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 PARBUTTERFLY 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, PARBUTTERFLY 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, PARBUTTERFLY 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 PARBUTTERFLY.

We perform a comprehensive evaluation of all of the algorithms in PARBUTTERFLY 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 PARBUTTERFLY, that enables us to obtain new parallel algorithms for butterfly counting and peeling. PARBUTTERFLY 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 PARBUTTERFLY framework provides different ways to aggregate wedges in

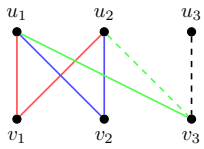

Figure 1: The butterflies in this graph are $\{u_1, v_1, u_2, v_2\}$, $\{u_1, v_1, u_2, v_3\}$, and $\{u_1, v_2, u_2, v_3\}$. The red, blue, and green edges each produce a wedge, which share the same endpoints, u_1 and u_2. Any two of these wedges form a butterfly. However, the dashed edges produce another wedge, $\{u_2, v_3, u_3\}$, which has different endpoints, u_2 and u_3. This wedge does not form a butterfly with any of the previous wedges.

parallel, including sorting, hashing, histogramming, and batching. Also, we can speed up butterfly counting by ranking vertices and only considering wedges formed by a particular ordering of the vertices. PARBUTTERFLY supports different parallel ranking methods, including side-ordering, approximate and exact degree-ordering, and approximate and exact complement-coreness ordering. These orderings can be used with any of the aggregation methods. To further speed up computations on large graphs, PARBUTTERFLY also supports parallel approximate butterfly counting via graph sparsification based on ideas by Sanei-Mehri *et al.* [48] for the sequential setting.

In addition, PARBUTTERFLY provides parallel algorithms for peeling bipartite networks based on sequential 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 Dhulipala *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 α,[1] PARBUTTERFLY gives a counting algorithm that takes $O(\alpha m)$ expected work and $O(\log m)$ span with high probability (w.h.p.).[2] Using a parallel Fibonacci heap that we design, PARBUTTERFLY gives a vertex-peeling algorithm that takes $O(\min(\text{max-b}_v, \rho_v \log n) + \sum_{v \in V} \deg(v)^2)$ expected work and $O(\rho_v \log^2 n)$ span w.h.p., and an edge-peeling algorithm that takes $O(\min(\text{max-b}_e, \rho_e \log n) + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ 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 ρ_v and ρ_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 PARBUTTERFLY 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 PARBUTTERFLY 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 PARBUTTERFLY.
(**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 PARBUTTERFLY 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 $N_2^G(v)$ 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), $(u_2, v_2) \in$

[1] The arboricity of a graph is defined to be the minimum number of disjoint forests that a graph can be partitioned into.

[2] By "with high probability" (w.h.p.), we mean that the probability is at least $1 - 1/n^c$ for any constant $c > 0$ for an input of size n.

E. 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*** α of a graph is the minimum number of spanning forests needed to cover the graph. In general, α 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 ε, 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.

PARBUTTERFLY Framework for Counting

(**1**) *Rank vertices*: Assign a global ordering, rank, to the vertices.
(**2**) *Retrieve wedges*: Retrieve a list W of wedges (x, y, z) where $\text{rank}(y) > \text{rank}(x)$ and $\text{rank}(z) > \text{rank}(x)$.
(**3**) *Count wedges*: For every pair of vertices (x_1, x_2), how many distinct wedges share x_1 and x_2 as endpoints.
(**4**) *Count butterflies*: Use the wedge counts to obtain the global butterfly count, per-vertex butterfly counts, or per-edge butterfly counts.

Figure 2: PARBUTTERFLY Framework for Counting

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.

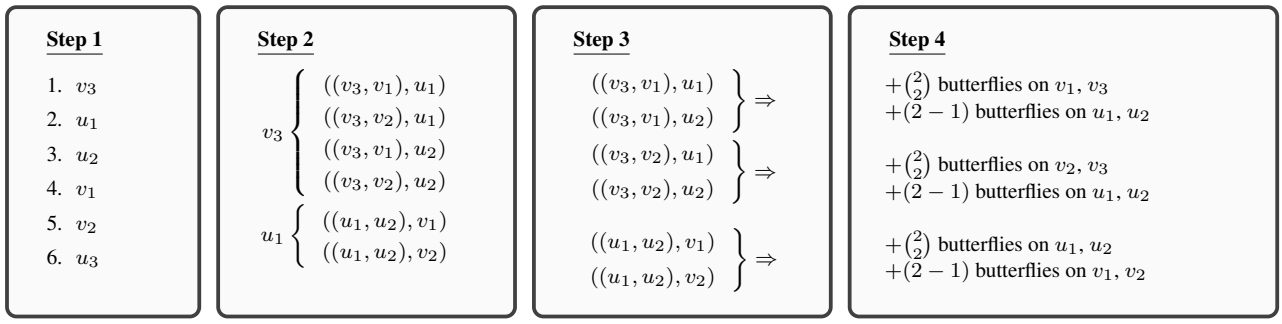

Figure 3: We execute butterfly counting per vertex on the graph in Figure 1. In Step 1, we rank vertices in decreasing order of degree. In Step 2, for each vertex v in order, we retrieve all wedges where v is an endpoint and where the other two vertices have higher rank (the wedges are represented as $((x, z), y)$ 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.[3] 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.[4]

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

[3] For total counts, butterfly counts can simply be computed and summed in parallel directly.

[4] Note that this is not feasible for partially-parallel batching, so in that case, the only option is to use atomic adds.

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.

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

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 Sariyüce 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.

Algorithm 1 Preprocessing

1: **procedure** PREPROCESS($G = (U, V, E), f$)
2: $V' \leftarrow$ SORT($U \cup V, f$) ▷ Sort vertices in increasing order of rank according to function f
3: Let u's rank $R[u]$ be its index in V'
4: $E' \leftarrow \{(R[u], R[v]) \mid (u, v) \in E\}$ ▷ Rename vertices to their rank
5: $G' = (V', E')$
6: **parfor** $u \in V'$ **do**
7: $N^{G'}(u) \leftarrow$ SORT($\{v \mid (u, v) \in E'\}$) ▷ Sort neighbors by decreasing order of rank
8: Store $\deg_u(u)$ and $\deg_v(u)$ for all $(u, v) \in E'$
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, $\deg_v(u)$, to be the number of neighbors $u' \in N(u)$ such that $\mathsf{rank}(u') > \mathsf{rank}(v)$. We also define a modified neighborhood, $N_v(u)$, to be the set of neighbors $u' \in N(u)$ such that $\mathsf{rank}(u') > \mathsf{rank}(v)$.

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*

$$\sum_{u' \in N_2(u)} \binom{|N(u) \cap N(u')|}{2}. \tag{4.1}$$

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

$$\sum_{u' \in N(v) \setminus \{u\}} (|N(u) \cap N(u')| - 1). \tag{4.2}$$

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 $(u, v) \in E$, each butterfly that (u, v) is contained within has additional vertices $u' \in U, v' \in V$ and additional edges $(u', v), (u, v'), (u', v') \in E$. Thus, iterating over all $u' \in N(v)$ (where $u' \neq u$), it suffices to count the number of vertices $v' \neq v$ such that v' is adjacent to u and to u'. In other words, it suffices to count $v' \in N(u) \cap N(u') \setminus \{v\}$. This gives us precisely $\sum_{u' \in N(v) \setminus \{u\}} (|N(u) \cap N(u')| - 1)$ 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** $u_1 \in V$ **do**
5: **parfor** $i \leftarrow 0$ to $\deg_{u_1}(u_1)$ **do**
6: $v \leftarrow N(u_1)[i]$ ▷ $v = i^{\text{th}}$ neighbor of u_1
7: **parfor** $j \leftarrow 0$ to $\deg_{u_1}(v)$ **do**
8: $u_2 \leftarrow N(v)[j]$ ▷ $u_2 = j^{\text{th}}$ neighbor of v
9: $W[I(i, j)] \leftarrow ((u_1, u_2), 1, v)$ ▷ (u_1, u_2) are the endpoints, v is the center of the wedge
10: **return** W

Algorithm 3 Parallel work-efficient butterfly counting per vertex

1: **procedure** COUNT-V-WEDGES(W)
2: $(R, F) \leftarrow$ GET-FREQ(W) ▷ Aggregate W and retrieve wedge frequencies
3: Initialize B to store butterfly counts per vertex
4: **parfor** $i \leftarrow 0$ to $|F| - 1$ **do**
5: $((u_1, u_2), d) \leftarrow R[i]$ ▷ u_1 and u_2 are the wedge endpoints
6: Store $(u_1, \binom{d}{2})$ and $(u_2, \binom{d}{2})$ in B ▷ Store butterfly counts per endpoint
7: **parfor** $j \leftarrow F[i]$ to $F[i+1]$ **do**
8: $(\_, \_, v) \leftarrow W[j]$ ▷ v is the wedge center
9: Store $(v, d-1)$ in B ▷ Store butterfly counts per center
10: $(B, \_) \leftarrow$ GET-FREQ(B) ▷ Aggregate B and get butterfly counts
11: **return** B
12: **procedure** COUNT-V($G = (U, V, E)$)
13: $G' = (V', E') \leftarrow$ PREPROCESS(G)
14: $W \leftarrow$ GET-WEDGES(G') ▷ Array of wedges
15: **return** COUNT-V-WEDGES(W)

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 $\mathrm{O}(\alpha m)$ wedges. GET-WEDGES (Algorithm 2) takes $\mathrm{O}(\alpha m)$ work and $\mathrm{O}(\log m)$ 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 histogramming, as discussed in Section 3.1. For an input of length n, GET-FREQ takes $\mathrm{O}(n)$ expected work and $\mathrm{O}(\log n)$ span w.h.p. using any of the three aggregation methods [18, 23, 26]. We have $\mathrm{O}(\alpha m)$ wedges, and so GET-FREQ takes $\mathrm{O}(\alpha m)$ expected work and $\mathrm{O}(\log m)$ 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* $\mathrm{O}(\alpha m)$ *expected work and* $\mathrm{O}(\log m)$ *span w.h.p.*

Note that this is a better worst-case work bound than the work bound of $\mathrm{O}(\sum_{v \in V} \deg(v)^2)$ using side order. In the worst-case $\mathrm{O}(\alpha m) = \mathrm{O}(m^{1.5})$ while $\mathrm{O}(\sum_{v \in V} \deg(v)^2) = \mathrm{O}(mn)$. 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 $\binom{|N(u) \cap N(u')|}{2}$ 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.

Algorithm 4 Parallel work-efficient butterfly counting per edge

1: **procedure** COUNT-E-WEDGES(W)
2: $(R, F) \leftarrow$ GET-FREQ(W) ▷ Aggregate W and retrieve wedge frequencies
3: Initialize B to store butterfly counts per edge
4: **parfor** $i \leftarrow 0$ to $|F| - 1$ **do**
5: $((u_1, u_2), d) \leftarrow R[i]$ ▷ u_1 and u_2 are the wedge endpoints
6: **parfor** $j \leftarrow F[i]$ to $F[i+1]$ **do**
7: $(\_, \_, v) \leftarrow W[j]$ ▷ v is the wedge center
8: Store $((u_1, v), d-1)$ and $((u_2, v), d-1)$ in B
9: $(B, \_) \leftarrow$ GET-FREQ(B) ▷ Aggregate B and get butterfly counts
10: **return** B
11: **procedure** COUNT-E($G = (U, V, E)$)
12: $G' = (V', E') \leftarrow$ PREPROCESS(G)
13: $W \leftarrow$ GET-WEDGES(G') ▷ Array of wedges
14: **return** COUNT-E-WEDGES(W)

THEOREM 4.1. *Butterfly counting per vertex can be performed in* $\mathrm{O}(\alpha m)$ *expected work and* $\mathrm{O}(\log m)$ *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* $\mathrm{O}(\alpha m)$ *expected work and* $\mathrm{O}(\log m)$ *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 $O(m^2 + m \cdot \text{max-b}_e)$ work, where max-b_e 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 time 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 implementations 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,[5] 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.

[5] Julienne is work-efficient in the context of k-core.

Algorithm 5 Parallel vertex peeling (tip decomposition)

```
1:  procedure UPDATE-V(G = (U, V, E), B, A)
2:      Initialize W to be an array of wedges
3:      parfor u_1 ∈ A do
4:          parfor v ∈ N(u_1) do
5:              parfor u_2 ∈ N(v) where u_2 ≠ u_1 do
6:                  Store ((u_1, u_2), 1, v) in W   ▷ (u_1, u_2) is the key, 1 is the frequency
7:      B' ← COUNT-V-WEDGES(G, W)
8:      Subtract corresponding counts B' from B
9:      return B
10: procedure PEEL-V(G = (U, V, E), B)   ▷ B is an array of butterfly counts per vertex
11:     Let K be a bucketing structure mapping U to buckets based on # of butterflies
12:     f ← 0
13:     while f < |U| do
14:         A ← all vertices in next bucket (to be peeled)
15:         f ← f + |A|
16:         B ← UPDATE-V(G, B, A)                ▷ Update # butterflies
17:         Update the buckets of changed vertices in B
18:     return K
```

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 $\sum_{v \in X} \binom{\deg(v)}{2}$ is minimized. Without loss of generality, let U be this bipartition.

Vertex peeling takes as input the per-vertex butterfly counts from the PARBUTTERFLY counting framework. We create a bucketing structure mapping vertices in 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 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

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 $O(\sum_{v \in U} \deg(v)^2)$. 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 ρ_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 $O(\rho_v \log^2 m)$ 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 Sariyüce and Pinar's [49] sequential algorithm, which takes $O(\text{max-b}_v + \sum_{v \in V} \deg(v)^2)$ 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 ρ_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 $O(\text{max-b}_v + \sum_{v \in V} \deg(v)^2)$ expected work and $O(\rho_v \log^2 n)$ span w.h.p. The work bound matches the work bound of Sariyüce 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* $O(\min(\text{max-b}_v, \rho_v \log n) + \sum_{v \in V} deg(v)^2)$ *expected work and* $O(\rho_v \log^2 n)$ *span w.h.p., where* max-b$_v$ *is the maximum number of per-vertex butterflies* ρ_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 $u_2 \in N(v_1) \backslash \{u_1\}$ and then find the intersection

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 (u1, v1) ∈ A do
4:          parfor u2 ∈ N(v1) where u2 ≠ u1 do
5:              N ← INTERSECT(N(u1), N(u2))
6:              Store ((u2, v1), |N| − 1, _) in B'
7:              parfor v2 ∈ N where v2 ≠ v1 do
8:                  Store ((u1, v2), 1, _) in B'
9:                  Store ((u2, v2), 1, _) in B'
10:     (B'', _) ← GET-FREQ(B')
11:     Subtract corresponding counts in B'' from B
12:     return B
13: procedure PEEL-E(G = (U, V, E), B)    ▷ 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 ← 0
16:     while f < m do
17:         A ← all edges in next bucket (to be peeled)
18:         f ← f + |A|
19:         B ←UPDATE-E(G, B, A)                ▷ Update # butterflies
20:         Update the buckets of changed edges in B
21:     return K
```

$N(u_1) \cap N(u_2)$. 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 Sariyüce 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 $O(\min(\deg(u), \deg(u')))$ work (by scanning

through the smaller list in parallel and performing lookups in the larger list). This gives us $O\big(\sum_{(u,v)\in E}\sum_{u'\in N(v)} \min(\deg(u),\deg(u'))\big)$ expected work. As in vertex peeling, to analyze the span of PEEL-E, we define ρ_e 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 $O\big(\rho_e \log^2 m\big)$ w.h.p.

Similar to vertex peeling, if the maximum number of per-edge butterflies is $\Omega\big(\rho_e \log m\big)$, which is likely true in practice, then the work of our algorithm is faster than the sequential algorithm by Sariyüce and Pinar [49]. The work of their algorithm is $O\big(\text{max-b}_e + \sum_{(u,v)\in E}\sum_{u'\in N(v)} \min(\deg(u), \deg(u'))\big)$, where max-$b_e$ is the maximum number of butterflies per-edge (assuming that their intersection is optimized).

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 $q \leq \rho_e$, we have that max-b_e is at most $\rho_e \log m$ (if this does not occur, we know that max-b_e is greater than $\rho_e \log m$, and we finish the algorithm as described above). Then, we terminate and restart the algorithm using the original bucketing structure of Dhulipala *et al.* [19], which will give an algorithm with $O\big(\text{max-b}_e + \sum_{(u,v)\in E}\sum_{u'\in N(v)} \min(\deg(u),\deg(u'))\big)$ expected work and $O\big(\rho_e \log^2 m\big)$ span w.h.p. Our work bound matches the work bound of Sariyüce 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* $O\big(\min(\text{max-b}_e, \rho_e \log m) + \sum_{(u,v)\in E}\sum_{u'\in N(v)} \min(deg(u), deg(u'))\big)$ *expected work and* $O\big(\rho_e \log^2 m\big)$ *span w.h.p., where* max-b_e *is the maximum number of per-edge butterflies and* ρ_e *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 [10, 36] and we compile our programs with g++ (version 7.3.1) using the `-O3` flag. We test our algorithms on real-world bipartite graphs from the Koblenz Network Collection (KONECT) [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 [48] and Sariyüce and Pinar's [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 Sariyüce and Pinar [49] are serial implementations for global and local butterfly counting, respectively. PGD [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.[6] 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 in 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

[6] By "large enough," we mean graphs for which the sequential counting algorithms take more than 2 seconds to complete.

Dataset	Abbreviation	$\lvert U\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	# butterflies	ρ_v	ρ_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	298,492,670,057	—	—
Discogs label-style	discogs	270,771	1,754,823	5,302,276	3,261,758,502	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	—	—
Wikipedia edits (en)	enwiki	21,416,395	3,819,691	122,075,170	2,036,443,879,822	—	—
Delicious user-item	delicious	33,778,221	833,081	101,798,957	56,892,252,403	165,850	—
Orkut	orkut	8,730,857	2,783,196	327,037,487	22,131,701,213,295	—	—
Web trackers	web	27,665,730	12,756,244	140,613,762	20,067,567,209,850	—	—

Figure 5: These are relevant statistics for the KONECT [34] graphs that we experimented on. Note that we only tested peeling algorithms on graphs for which Sariyüce 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 ρ_v and ρ_e data, and these entries are represented by a dash.

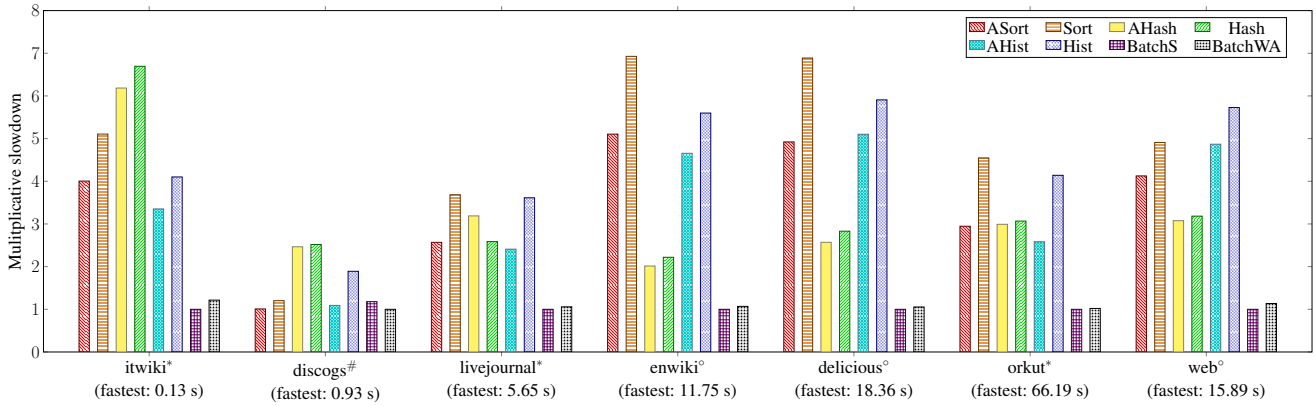

Figure 6: These are the parallel runtimes for butterfly counting per vertex, considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking,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

	Total Counts					Per-Vertex Counts			Per-Edge Counts		
Dataset	PB T_{48h}	PB T_1	Sanei-Mehri et al. [48] T_1	PGD [2] T_{48h}	ESCAPE [46] T_1	PB T_{48h}	PB T_1	Sariyüce and Pinar [49] T_1	PB T_{48h}	PB T_1	Sariyüce and Pinar [49] T_1
itwiki	0.10*	1.38*	1.63	1798.43	4.97	0.13*	1.43*	6.06	0.37*	3.24°	19314.87
discogs	0.90#◊	1.36°	4.12	234.48	2.08	0.93#◊	1.53°	96.09	0.59°	5.01*	1089.04
livejournal	3.83*	35.41*	37.80	> 5.5 hrs	139.06	5.65*	36.22*	158.79	10.26°	105.65*	> 5.5 hrs
enwiki	8.29°	68.73*	69.10	> 5.5 hrs	151.63	11.75°	75.10*	608.53	16.73°	167.69*	> 5.5 hrs
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	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	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.

counting. 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.

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

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 α 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 $O(\sum_{v \in V} \deg(v)^2)$ work. Zou [65] develop the first algorithm for butterfly peeling per edge, with $O(m^2 + m \cdot \text{max-b}_e)$ work. Sariyüce and Pinar [49] give algorithms for butterfly peeling over vertices and over edges, which take $O(\text{max-b}_v + \sum_{v \in V} \deg(v)^2)$ work and $O(\text{max-b}_e + \sum_{u \in U} \sum_{v_1, v_2 \in N(u)} \max(\deg(v_1), \deg(v_2)))$ 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 1.8×10^{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 $O(\sum_{(u,v) \in E} (\deg(v) + \sum_{u' \in N(v)} \deg(u')))$, 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

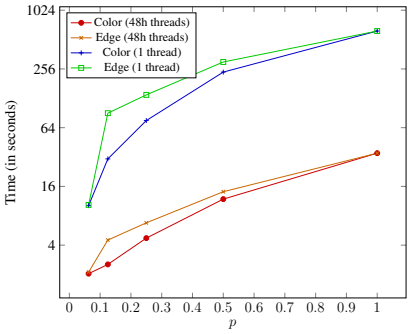

Figure 8: These are the runtimes for colorful and edge sparsification over probabilities p. We considered both the runtimes on 48 cores hyperthreaded and on a single thread. We ran these on orkut, using simple batch aggregation and side ranking.

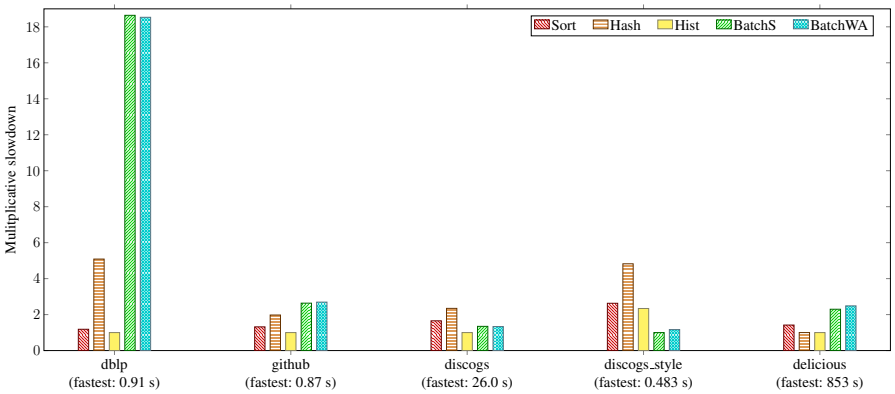

Figure 9: These are the parallel runtimes for butterfly vertex peeling with different wedge aggregation methods (these runtimes do not include the time taken to count butterflies). All times are scaled by the fastest parallel time, as indicated in parentheses. Also, note that the runtimes for discogs_style represent single-threaded runtimes; this is because we did not see any parallel speedups for discogs_style, due to the small number of vertices that were peeled.

	Vertex Peeling			Edge Peeling		
Dataset	ParButterfly T_{48h}	ParButterfly T_1	Sariyüce and Pinar [49] T_1	ParButterfly T_{48h}	ParButterfly T_1	Sariyüce and Pinar [49] T_1
dblp	0.91°	2.40°	2.06	2.06°	16.90°	6.93
github	0.87°	1.03°	1.15	5.25*	18.00*	18.82
discogs	26°	53.10*	157.14	306*	2160*	2149.54
discogs_style	0.48*	0.48*	14826.16	2380#	15600*	16449.56
delicious	853°	1900*	2184.27	—	—	—

Figure 10: These are runtimes in seconds for parallel and single-threaded butterfly peeling from ParButterfly and serial butterfly peeling from Sariyüce and Pinar [49]. Note that these runtimes do not include the time taken to count butterflies. For the runtimes from ParButterfly, we have noted the aggregation method used; * refers to simple batching, # refers to sorting and ° refers to histogramming.

counting (e.g., [6, 7, 14, 19, 24, 25, 27, 29, 33, 37, 43, 44, 45, 54, 55, 56, 58, 63], among many others).

7 Conclusion

We have designed a framework 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 ParButterfly framework is built with modular components that can be combined for practical efficiency. 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

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[2] N. K. Ahmed, J. Neville, R. A. Rossi, N. G. Duffield, and T. L. Willke. Graphlet decomposition: framework, algorithms, and applications. *Knowl. Inf. Syst.*, 50(3):689–722, 2017.

[3] N. K. Ahmed, T. L. Willke, and R. A. Rossi. Estimation of local subgraph counts. In *IEEE International Conference on Big Data*, pages 586–595, 2016.

[4] N. K. Ahmed, T. L. Willke, and R. A. Rossi. Exact and estimation of local edge-centric graphlet counts. In *International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 1–17, 2016.

[5] S. G. Aksoy, T. G. Kolda, and A. Pinar. Measuring and modeling bipartite graphs with community structure. *J. Complex Networks*, 5:581–603, 2017.

[6] S. Arifuzzaman, M. Khan, and M. Marathe. PATRIC:

A parallel algorithm for counting triangles in massive networks. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 529–538, 2013.

[7] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, 2015.

[8] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 16–24, 2008.

[9] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 189–199, 2010.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[11] S. P. Borgatti and M. G. Everett. Network analysis of 2-mode data. *Social Networks*, 19(3):243 – 269, 1997.

[12] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Motif counting beyond five nodes. *TKDD*, 12(4):48:1–48:25, 2018.

[13] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, Feb. 1985.

[14] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Eng.*, 11(4):29–41, July 2009.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[16] V. S. Dave, N. K. Ahmed, and M. Hasan. PE-CLoG: Counting edge-centric local graphlets. In *IEEE International Conference on Big Data*, pages 586–595, 2017.

[17] I. 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.

[18] L. Dhulipala, G. Blelloch, and J. 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.

[19] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 393–404, 2018.

[20] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis. Distributed estimation of graph 4-profiles. In *International Conference on World Wide Web (WWW)*, pages 483–493, 2016.

[21] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[22] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 721–732. VLDB Endowment, 2005.

[23] 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.

[24] O. Green, L. M. Munguia, and D. A. Bader. Load balanced clustering coefficients. In *Workshop on Parallel Programming for Analytics Applications*, pages 3–10, 2014.

[25] O. Green, P. Yalamanchili, and L. M. Munguia. Fast triangle counting on the GPU. In *Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2015.

[26] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.

[27] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *ACM SIGMOD International Conference on Management of Data*, pages 1587–1602, 2018.

[28] T. Hocevar and J. Demsar. A combinatorial approach to graphlet counting. *Bioinformatics*, pages 559–65, 2014.

[29] Y. Hu, H. Liu, and H. H. Huang. TriCore: Parallel triangle counting on gpus. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 14:1–14:12, 2018.

[30] S. Jain and C. Seshadhri. A fast and provable method for estimating clique counts using Turán's theorem. In *International Conference on World Wide Web (WWW)*, pages 441–449, 2017.

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

[32] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *International Conference on World Wide Web (WWW)*, pages 495–505, 2015.

[33] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task. Counting triangles in massive graphs with MapReduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014.

[34] J. Kunegis. KONECT: the Koblenz network collection. pages 1343–1350, 05 2013.

[35] M. Latapy, C. Magnien, and N. D. Vecchio. Basic notions for the analysis of large two-mode networks. *Social Networks*, 30(1):31 – 48, 2008.

[36] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3), 2010.

[37] D. Makkar, D. A. Bader, and O. Green. Exact and parallel triangle counting in dynamic graphs. In *IEEE International Conference on High Performance Computing (HiPC)*, pages 2–12, 2017.

[38] D. Marcus and Y. Shavitt. Efficient counting of network motifs. In *IEEE International Conference on Distributed Computing Systems Workshops*, pages 92–98, 2010.

[39] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

[40] D. Mawhirter, B. Wu, D. Mehta, and C. Ai. ApproxG: Fast approximate parallel graphlet counting through accuracy control. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 533–542, 2018.

[41] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[42] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, Mar. 2018.

[43] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a MapReduce implementation. *Inf. Process. Lett.*, 112(7):277–281, Mar. 2012.

[44] H.-M. Park and C.-W. Chung. An efficient MapReduce algorithm for counting triangles in a very large graph. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 539–548, 2013.

[45] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh. MapReduce triangle enumeration with guarantees. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 1739–1748, 2014.

[46] A. Pinar, C. Seshadhri, and V. Vishal. ESCAPE: Efficiently counting all 5-vertex subgraphs. In *International Conference on World Wide Web (WWW)*, pages 1431–1440, 2017.

[47] R. A. Rossi, R. Zhou, and N. K. Ahmed. Estimation of graphlet counts in massive networks. *IEEE Trans. Neural Netw. Learning Syst.*, 30(1):44–57, 2019.

[48] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura. Butterfly counting in bipartite networks. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2150–2159, 2018.

[49] A. E. Sariyüce and A. Pinar. Peeling bipartite networks for dense subgraph discovery. In *ACM International Conference on Web Search and Data Mining (WSDM)*, pages 504–512, 2018.

[50] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269 – 287, 1983.

[51] J. Shi and J. Shun. Parallel algorithms for butterfly computations. *CoRR*, abs/1907.08607, 2019.

[52] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[53] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012.

[54] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *IEEE International Conference on Data Engineering (ICDE)*, pages 149–160, 2015.

[55] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *International World Wide Web Conference (WWW)*, pages 607–614, 2011.

[56] K. Tangwongsan, A. Pavan, and S. Tirthapura. Parallel triangle counting in massive streaming graphs. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 781–786, 2013.

[57] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1(2):75–81, Apr 2011.

[58] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. DOULION: Counting triangles in massive graphs with a coin. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 837–846, 2009.

[59] J. Wang, A. W.-C. Fu, and J. Cheng. Rectangle counting in large bipartite graphs. In *IEEE International Congress on Big Data*, pages 17–24, 2014.

[60] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang. Vertex priority based butterfly counting for large-scale bipartite networks. *PVLDB*, 12(10), June 2019.

[61] P. Wang, J. Zhao, X. Zhang, Z. Li, J. Cheng, J. C. S. Lui, D. Towsley, J. Tao, and X. Guan. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 30(1):73–86, Jan 2018.

[62] X. Xia. Efficient and scalable listing of four-vertex subgraphs. Master's thesis, Texas A&M University, 2016.

[63] Y. Zhang, H. Jiang, F. Wang, Y. Hua, D. Feng, and X. Xu. LiteTE: Lightweight, communication-efficient distributed-memory triangle enumerating. *IEEE Access*, 7:26294–26306, 2019.

[64] R. Zhu, Z. Zou, and J. Li. Fast rectangle counting on massive networks. In *IEEE International Conference on Data Mining (ICDM)*, pages 847–856, 2018.

[65] Z. Zou. Bitruss decomposition of bipartite graphs. In *Database Systems for Advanced Applications*, pages 218–233, 2016.