Counting Induced 6-Cycles in Bipartite Graphs

Jason Niu
University at Buffalo
Buffalo, NY, USA
jasonniu@buffalo.edu

Jaroslaw Zola
University at Buffalo
Buffalo, NY, USA
jzola@buffalo.edu

Ahmet Erdem Sarıyüce
University at Buffalo
Buffalo, NY, USA
erdem@buffalo.edu

ABSTRACT

Various complex networks in real-world applications are best represented as a bipartite graph, such as user-product, paper-author, and actor-movie relations. Motif-based analysis has substantial benefits for networks and bipartite graphs are no exception. The smallest non-trivial subgraph in a bipartite graph is a $(2, 2)$-biclique, also known as a butterfly. Although butterflies are succinct, they are limited in capturing the higher-order relations between more than two nodes from the same node set. One promising structure in this context is the induced 6-cycle which consists of three nodes on each node set forming a cycle where each node has exactly two edges. In this paper, we study the problem of counting induced 6-cycles through parallel algorithms. To the best of our knowledge, this is the first study on induced 6-cycle counting. We first consider two adaptations based on previous works for cycle counting in bipartite networks. Then, we introduce a new approach based on the node triplets and offer a systematic way to count the induced 6-cycles. Our final algorithm, BatchTripletJoin, is parallelizable across root nodes and uses minimal global storage to save memory. Our experimental evaluation on a 52 core machine shows that BatchTripletJoin is significantly faster than the other algorithms while being scalable to large graph sizes and number of cores. On a network with $112M$ edges, BatchTripletJoin is able to finish the computation in 78 mins by using 52 threads.

CCS CONCEPTS

• **Mathematics of computing → Hypergraphs**.

KEYWORDS

induced 6-cycle, bipartite, hypergraph, parallel

ACM Reference Format:
Jason Niu, Jaroslaw Zola, and Ahmet Erdem Sarıyüce. 2022. Counting Induced 6-Cycles in Bipartite Graphs. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3545008.3545076

1 INTRODUCTION

The growing interest in bipartite graphs derives from the applications which model the relationships between two distinct groups [8, 14–16, 22]. In a bipartite graph, the node set is divided into two disjoint and independent sets U and V such that every edge connects a node in U to one in V. For example, recommendation networks are often represented as a bipartite graph with users as one node set and items as the other [25]. Bipartite graphs are also used to model hypergraphs where entities take part in group relations, such as actor-movie [39], author-paper [27], and company-board member [31] connections. Despite their representation power, bipartite graphs are understudied because most graph algorithms, including motif analysis, are focused on the traditional unipartite graphs. One solution is to project bipartite graphs to obtain the unipartite representation but this comes at a cost of significant information loss and inflated graph size [7, 34]. Hence, it is essential to design algorithms that directly work on the bipartite graphs.

Motif-based analysis is shown to have significant benefits for various graph mining tasks [4, 34, 35]. The smallest non-trivial motif in a bipartite graph is a butterfly ($(2, 2)$-biclique), also known as a four-cycle and a rectangle [32, 38]. Butterflies are shown to be an effective building block for community structure and used in various graph mining tasks in bipartite graphs [5, 19, 41]. Sequential and parallel algorithms are designed for butterfly counting in offline and online scenarios [11, 32, 33, 37, 38]. However, butterflies capture the higher-order relations between only two nodes from the same node set. Alternative measures also have limited success, for example a $(3, 3)$-biclique suffers from the prohibitive cost of computation [8] and a 3-path (i.e., butterfly minus an edge) is unable to model cohesion [31]. There is a need to go for larger

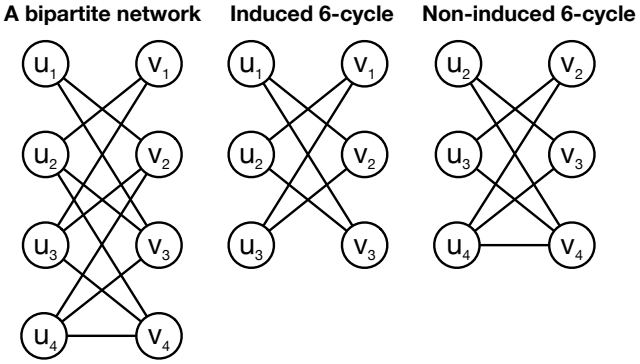

Figure 1: A toy bipartite network G, an induced 6-cycle in G, and a non-induced 6-cycle in G. The induced 6-cycle (u_1, u_2, u_3, v_1, v_2, v_3) consists of exactly six edges and the degree of each node is exactly 2. The non-induced 6-cycle (u_2, u_3, u_4, v_2, v_3, v_4) has an additional edge, making the degree of two nodes three. The induced 6-cycle does not include a butterfly whereas the non-induced 6-cycle contains two butterflies: u_2, u_4, v_3, v_4 and u_3, u_4, v_2, v_4. Both the left and right projections of an induced or non-induced 6-cycle result in triangles; each node pair is related since they share a neighbor in G. However, induced 6-cycles are the smallest bipartite structure that enables such projections.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545076

bipartite motifs which can model higher-order relations while being computationally affordable.

One promising structure in this context is the **6-cycle**, proposed by Opsahl [28] to model the triadic closure in bipartite networks. A 6-cycle consists of three nodes on each node set forming a cycle. Recently, Yang et al. introduced algorithms for counting *non-induced* 6-cycles [40]. While their algorithms are efficient, they ignore the inducedness constraint which is a key to get more informative results by avoiding combinatorial explosion. In general, it is well known that induced motifs (also known as graphlets) are more useful than non-induced motifs in real-world applications, such as anomaly detection, but also remain more challenging to compute [30, 36]. In an *induced* 6-cycle, each node has exactly two edges. There is no butterfly (or biclique) since each pair of nodes (from the same set) shares only one neighbor. An induced 6-cycle relates three nodes in the same node set to each other by forming a triangle in the projections with the minimal number of edges (see Figure 1). In that respect, induced 6-cycles offer a more distilled perspective than butterflies or bicliques. However, counting induced 6-cycles is more challenging than non-induced 6-cycles since one has to account for the lack of certain edges to ensure inducedness.

In this work, we present parallel algorithms to count induced 6-cycles in bipartite graphs. To the best of our knowledge, there is no prior work on counting induced 6-cycles. Due to the high computational cost, we use the affordances of shared-memory parallelization for practical runtime performance. We first consider two previous studies on cycle counting in bipartite networks and adapt them for parallel induced 6-cycle counting. In particular, we use the breadth-first search idea from [13] and wedge join technique from [37]. We show that those approaches have prohibitive time and space costs, and are therefore not scalable for large bipartite networks. As a solution, we propose counting induced 6-cycles over node triplets (three nodes on the same node set). Node triplets offer a systematic way to count induced 6-cycles in batches, thus avoiding duplicate work and enabling time-space tradeoffs for faster computation. We further consider space improvements by minimizing global storage and reduction of set intersection/difference operations when designing the BatchTripletJoin algorithm. In all our algorithms, we expose embarrassingly parallel computations in the coarse level and also make use of a preprocessing routine to assign better workloads for the threads. Preprocessing filters out the redundant parts of the graph while keeping the induced 6-cycle count the same and performs graph reordering to increase efficiency. We perform an extensive experimental evaluation on real-world networks and investigate the runtime and memory usage performance of our algorithms along with strong and weak scalability studies.

Our contributions can be summarized as follows:

- **Preprocessing.** We consider several techniques to filter and reorder the graph to speed up the induced 6-cycle counting. These techniques are applied to all our algorithms.
- **Adapting cycle counting algorithms.** Since this is the first study on induced 6-cycle counting, we propose two parallel adaptations of prior works on cycle counting to find the total number of induced 6-cycles.
- **Counting by node triplets.** We give a new approach based on counting induced 6-cycles for node triplets. We show the relationship between node triplets and induced 6-cycles through a pattern of set operations.
- **Improving the runtime and memory usage.** We introduce BatchTripletJoin, an improved space-efficient algorithm for node triplets that uses reduced set operations.
- **Evaluation on real-world networks.** We evaluate all our algorithms on various real-world bipartite networks. We compare the runtime of our algorithms for differing number of cores to demonstrate high scalability and practical runtimes. On a network with more than half a billion edges, BatchTripletJoin finishes the computation in 13.2 hours by using 52 threads.

Outline. We present preliminary definitions and notation in Section 2 and summarize the prior work on motif counting in bipartite networks in Section 3. Then, we give a series of preprocessing techniques to speed up induced 6-cycle counting in Section 4 and adaptations of two cycle counting algorithms for induced 6-cycle counting in Section 5. Next, we present our two main algorithms based on the use of node triples in Section 6. We give our experimental evaluation in Section 7 and conclusion in Section 8.

2 PRELIMINARIES

We work on a simple and undirected bipartite graph $G = (U, V, E)$ where U is the set of nodes in the left set, V is the set of nodes in the right set, and E is the set of edges. The neighbors of a node v is denoted by $N(v)$. The degree of a node v ($|N(v)|$) is $d(v)$ and the average degree of nodes in U and V are $\langle d_U \rangle$ and $\langle d_V \rangle$, respectively. Also, we use $\langle d2_U \rangle$ to denote the average number of distance-2 neighbors of nodes in U. We denote $|U| + |V|$ as n (number of nodes) and $|E|$ as m (number of edges). The summation of all elements in a list X is denoted as $sum(X)$. For parallel time and space complexities, we represent the number of processing units as p.

An **induced 6-cycle** is a set of six nodes $u_1, u_2, u_3 \in U$ and $v_1, v_2, v_3 \in V$ and six edges as follows (w.l.o.g):
(i) $(u_1, v_2), (u_1, v_3), (u_2, v_1), (u_2, v_3), (u_3, v_1), (u_3, v_2)$ edges exist;
(ii) $(u_1, v_1), (u_2, v_2), (u_3, v_3)$ edges do **not** exist.

If only (i) holds, it is a **non-induced 6-cycle**. In a given bipartite network G, we find the total number of instances of induced 6-cycles. In an induced 6-cycle instance, two vertices are connected if and only if they are also connected in G. The degree of a node is exactly two in an induced 6-cycle and at least two in a non-induced 6-cycle. Figure 1 gives an example for both.

We define a **wedge** as a 2-path composed of two **endpoint** vertices $u_1, u_2 \in U$ and a **center** vertex $v \in V$ with edges $(u_1, v), (u_2, v) \in E$ (we always consider the endpoints in the left set and the center vertex in the right set). An induced 6-cycle is made up of three wedges connected to each other in a cyclic way. We use $W(x)$ to denote the set of wedges where x is the smaller of the two endpoints (in U) and $W(x, y)$ to denote the set of wedges whose endpoints are x and y. The total number of wedges centered on V is equal to $\sum_{v \in V} \binom{d(v)}{2}$ and denoted by $|W|$. The average $W(u)$ for all $u \in U$ is represented as $\langle W_U \rangle$.

3 RELATED WORK

In this section, we review various related works on finding motifs in bipartite networks.

Algorithm 1: PREPROCESSING (G)

Input: $G = (U, V, E)$: graph
Output: $G' = (U', V', E')$: processed graph

1 $G \leftarrow$ 2-core of G
2 **if** $|U| > |V|$ **then** $Swap(U, V)$ // Ensure $|U| < |V|$
// Sort the nodes in U by inc. count of wedges
3 $X \leftarrow SortbyWedgeCounts(U)$
4 Let x's rank $R[x]$ be its index in X
5 **parallel foreach** $u \in U$ **do** add $R[u]$ to U'
6 $V' \leftarrow V$
// $N'(x)$ is the neighbors of node x in G'
// In both loops, neighbors sorted in descending order
7 **parallel foreach** $u \in U$ **do**
$N'(R[u]) \leftarrow Sort(\{v | (u, v) \in E\})$
8 **parallel foreach** $v \in V$ **do**
$N'(v) \leftarrow Sort(\{R[u] | (u, v) \in E\})$
9 **return** G'

Counting Short Cycles in Bipartite Networks. A cycle in a bipartite network is considered to be short if its length k follows $g \leq k \leq 2g - 2$ where g is the length of the smallest cycle in the graph. The objective here is to count all non-induced short cycles in bipartite networks. A message-passing algorithm was proposed by Karimi and Banihashemi [20] which iteratively passes messages across a node's neighbors to count all short cycles within a bipartite graph. Dehghan and Banihashemi [13] proposed an algorithm to count short cycles by applying breadth-first search to all nodes in either the left or right set of a bipartite network.

Butterfly Counting. In this problem, the objective is to count the number of butterflies in bipartite networks. A butterfly is the smallest cycle in bipartite networks and has a variety of applications such as document clustering [14] and link spam detection [16]. The first work for butterfly counting is by Wang et al. who introduced a counting scheme which uses the number of wedges containing each node in the left set to calculate the total butterfly count [38]. Sanei-Mehri et al. improved upon Wang et al.'s algorithm by computing the number of wedges for each node in the set with lower runtime cost [32]. The set with the higher sum of squares of the degrees for each node is selected. Along with the exact counting algorithms, they also proposed randomized algorithms which can approximate the number of butterflies in bipartite networks. In another work, Sanei-Mehri et al. introduced streaming algorithms to count butterflies in graph streams [33]. Shi and Shun [37] recently designed a parallel butterfly counting algorithm which modified Chiba and Nishizeki's wedge retrieval process [11] to enable parallelization.

6-Cycle Counting. The problem of counting 6-cycles in bipartite networks has only recently been studied for large bipartite networks. Yang et al. introduced algorithms to count the number of non-induced 6-cycles, which they denote as bi-triangles [40]. Their algorithms are based on combining wedges and super-wedges, with the former being 2-paths and the latter being 3-paths. They also introduce local 6-cycle counting algorithms which count the number of 6-cycles containing a specified node or edge. In our work, we consider **induced 6-cycle counting**, which is more challenging and promising for real-world applications.

4 PREPROCESSING

We make use of a generic preprocessing step in all our algorithms which formats the graph to speed up computations (Section 4). To speed up the computation for large bipartite graphs, we can shrink and reformat the graph such that the induced 6-cycle count stays the same. In PREPROCESSING, outlined in Algorithm 1, we give a computation that takes as input a bipartite graph and outputs another bipartite graph that filters out some parts of the input and reorders the nodes and neighbor lists. We first update the input graph to only consider the nodes and edges that are in a 2-core, which is a maximal connected subgraph in which all nodes have a degree of at least 2 (line 1). Since all the nodes in an induced 6-cycle have a degree of at least 2, we can simply ignore the nodes outside the 2-core, thus reducing the size of the graph. Afterwards, if necessary, we swap the left (U) and right sets (V) to ensure that the left set (U) has the smaller number of nodes (line 2). We always parallelize based on U in our counting algorithms, hence making it the smaller set increases the number of induced 6-cycles that are processed in batches for each thread. Next, we reorder each node $u \in U$ in increasing order of wedges from u (lines 3 - 5). The wedge count for a node u is $\sum_{v \in N(u)} d(v) - 1$ (we consider the wedges where u is an end-point, as defined in Section 2). Note that Shi and Shun [37] showed that reordering the graph using approximate degree ordering or degeneracy ordering yields efficient results and here we consider wedge count based ordering in a similar spirit. Finally, we sort each neighbor list in descending order of node ids (lines 7 - 8). This enables linear time set intersection and difference operations. We evaluate the impact of our techniques in PREPROCESSING as well as various node reordering schemes in Section 7.4.

Time and space complexity. In PREPROCESSING, core decomposition (line 1) takes $O(m)$ time [6]. The swapping of left and right sets (line 2) is $O(1)$ if pointers are swapped instead of the contents themselves. Finally, reordering the nodes and sorting neighbor lists in descending order (lines 3-8) takes $O(|X| \log |X| + m \log m)$ time where $|X| = min(|U|, |V|)$. Overall, PREPROCESSING takes $O(m \log m)$ time. Asymptotically, it never becomes the bottleneck in any of our counting algorithms. The space complexity for storing the processed graph and temporary variables is $O(n + m)$.

5 ADAPTING CYCLE COUNTING

Since induced 6-cycle counting has not been studied before, we start by proposing two adaptations inspired by the previous works for cycle counting in bipartite networks. The first is a modified version of breadth-first search to count induced 6-cycles, which Dehghan and Banihashemi also used to count short cycles [13] (Section 5.1). The second is based on the parallel wedge retrieval algorithm proposed by Shi and Shun, which was used for butterfly counting [37] (Section 5.2).

5.1 Counting by Breadth-First Search

One of the more common methods for finding cycles in a graph is through breadth-first search (BFS) [13]. The idea is to simply perform a traversal for a few levels and determine the number of cycles that the root node takes part in. To count induced 6-cycles, we introduce the NODEJOIN algorithm, outlined in Algorithm 2. Given

Algorithm 2: NodeJoin (G)

Input: $G(U, V, E)$: graph
Output: *count*: number of induced 6-cycles

```
1  G ← Preprocessing (G)
2  counts ← []                                   // |U| values
3  parallel foreach u1 ∈ U do
4      S ← ∅  // Hashmap of node pairs (from U) to values
5      foreach v2, v3 ∈ N(u1) s.t. v2 > v3 do
6          H ← ∅                                 // Set of nodes
7          foreach u3 ∈ N(v2)\N(v3) and u3 > u1 do
8              add u3 to H
9          foreach u2 ∈ N(v3)\N(v2) and u2 > u1 do
10             foreach u3 ∈ H do
                   // S stores the number of v1s (see Fig.
                      2)
11                 if (u2, u3) ∉ S then
12                     S[(u2, u3)] ← |N(u2) ∩ N(u3)\N(u1)|
13                 counts[u1] ← counts[u1] + S[(u2, u3)]
14 count ← sum(counts)                           // Parallel reduction
15 return count
```

a bipartite graph $G = (U, V, E)$, NodeJoin counts the induced 6-cycles by performing a limited BFS from each vertex $u \in U$ up until a depth level of three. Figure 2 illustrates the BFS tree where $u_1 \in U$ is the root node. $v_2, v_3 \in V$ are two of u_1's neighbors, hence put at level one. In level two, we find a neighbor of v_2 which is not connected to v_3, denoted by u_3 (and vice versa, denoted by u_2). In the last level, we find a common neighbor of u_2 and u_3, denoted by v_1, which is not connected to u_1.

For each root node $u_1 \in U$, the same u_2, u_3 pair may appear in multiple induced 6-cycles containing u_1. To avoid duplicate processing, we use the container S (line 4 in Algorithm 2) to store the number of nodes in $N(u_2) \cap N(u_3)\backslash N(u_1)$, which corresponds to v_1s in the last level of the BFS tree (line 12). We also ensure an ordering such that $u_3 > u_1$ and $u_2 > u_1$ (lines 7 and 9) to break the symmetry and thus prevent the duplicate processing of node pairs from U. Note that the *counts* list contains the number of induced 6-cycles counted *through* each vertex; it is not the actual count for each vertex. The sum of *counts* gives the total induced 6-cycle count (line 14). NodeJoin has a coarse-grained parallelism where the root nodes in U are shared among the threads (line 3).

A significant drawback of NodeJoin is the recomputation of set intersections across BFS trees. Multiple root nodes u_1 may participate in an induced 6-cycle with the same u_2 and u_3 node pair, resulting in the recomputation of $N(u_2) \cap N(u_3)$ (line 12). One solution would be to store each set intersection for all pairs of root nodes, but it will have prohibitive space usage for large networks.

Time complexity. We express the costs in terms of average node degrees, $\langle d_U \rangle$ and $\langle d_V \rangle$, to enable a tight analysis. There are $|U|$ iterations performed in total which go over each node $u \in U$ (line 3). The loop in line 5 iterates over node pairs in u_1's neighbor list, corresponding to $O(\binom{\langle d_U \rangle}{2})$ iterations. The cost of lines 7 and 8 is $O(\langle d_V \rangle)$. Lines 9 and 10 take $O(\langle d_V \rangle)$ iterations each, for a total of $O(\langle d_V \rangle^2)$ iterations. Computing the set operations in line 12 takes $O(\langle d_U \rangle)$ time because $N(u_2) \cap N(u_3)\backslash N(u_1)$ can be computed in linear-time by simultaneously going over the neighbor lists of u_2, u_3, and u_1 (neighbor lists are kept sorted in descending order, see Section 4). Overall, the total time of NodeJoin is $O(|U| \cdot \binom{\langle d_U \rangle}{2} \cdot (\langle d_V \rangle + \langle d_V \rangle^2 \cdot \langle d_U \rangle))$ which is equal to $\boldsymbol{O(m \cdot \langle d_U \rangle^2 \cdot \langle d_V \rangle^2)}$. The parallel time complexity of NodeJoin is simply $O(1/p \cdot m \cdot \langle d_U \rangle^2 \cdot \langle d_V \rangle^2)$ since it is embarrassingly parallel.

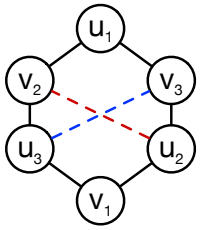

Figure 2: NodeJoin's BFS tree. The dotted lines represent the edges which do not exist. The BFS tree goes from top to bottom with u_1 being the root node and node v_1 at depth level three. We check for the lack of the blue edge in line 7 and of the red edge in line 9 in Algorithm 2.

Space complexity. In addition to the $O(m)$ space taken by the graph, NodeJoin uses one global container *counts* (line 2) and two local containers S (line 4) and H (line 6) per thread to store various auxiliary information. *counts* stores $|U|$ values. S stores $O(\binom{\langle d_V \rangle}{2})$ values which in the worst case can be $O(|U|^2)$. H stores up to $|U|$ nodes. Therefore, the space complexity of NodeJoin is $O(m + p \cdot (|U| + |U|^2 + |U|) = O(p \cdot |U|^2)$. Note that in practice we observe that this is a loose bound and the actual memory footprint is much smaller (see Section 7.3).

5.2 Counting by Wedges

An alternative way to count induced 6-cycles is by aggregating wedges. Since induced 6-cycles are composed of three overlapping wedges (see Figure 3), we can reduce the cost of computation by operating on wedges rather than nodes. Shi and Shun proposed to use (and store) wedges for counting butterflies [37]. We can count induced 6-cycles using a similar wedge retrieval technique while taking advantage of the patterns associated with inducedness.

We describe our wedge based counting algorithm WedgeJoin in Algorithm 3. WedgeJoin simply goes over triples of wedges and counts the ones that form an induced 6-cycle. Wedge retrieval (lines 2-6) is based off of Shi and Shun's [37] algorithm and enables the parallel processing of wedges. The parallel container W allows for fast access of wedges based on endpoints. Unlike their algorithm, we only find wedges with endpoints in U instead of the entire node set. In our implementation, W is a list of all the wedges in the graph such that each wedge consists of two nodes from U (endpoints) and one node from V (center). We partition W based on the smaller endpoint (u_1) and sort each partition with respect to the larger endpoint (u_2). For all nodes $u_1 \in U$, W enables the retrieval of all wedges with endpoints u_1, u_2 and center v_3 such that $u_1 < u_2$.

WedgeJoin finds cycles of wedges (u_1, v_3, u_2), (u_2, v_1, u_3), and (u_1, v_2, u_3) such that $u_1 < u_2 < u_3$. Line 8 (in Algorithm 3) iterates over all blue wedges (u_1, v_3, u_2) and line 10 iterates over all green wedges (u_2, v_1, u_3) (Figure 3). The lack of edges (u_1, v_1), (u_2, v_2), and (u_3, v_3) is needed to satisfy the inducedness (the dashed black edges in Figure 3). To ensure that the blue and green wedge pair satisfy the inducedness constraint, we first check for the nonexistence of (u_1, v_1) and (u_3, v_3) in line 13. Afterwards, we traverse all red wedges (u_1, v_2, u_3) (Figure 3) in line 15. Finally, we check for the last unwanted edge (u_2, v_2) in line 17 and increment the induced 6-cycle count of the blue wedge.

Algorithm 3: WedgeJoin (G)

Input: $G(U, V, E)$: graph
Output: *count*: number of induced 6-cycles

```
1  G ← Preprocessing (G)
2  W ← ∅                                   // Parallel container of wedges
3  parallel foreach u1 ∈ U do
4      foreach v3 ∈ N(u1) do
5          foreach u2 ∈ N(v3) s.t. u2 > u1 do
6              add (u1, v3, u2) to W (sorted by endpoints)
7  counts ← []                                          // |W| values
8  parallel foreach w1 ∈ W do
9      (u1, v3, u2) ← w1      // Blue wedge in Fig. 3; u1 < u2
10     foreach w2 ∈ W(u2) do
11         (u2, v1, u3) ← w2        // Green wedge in Fig. 3;
            u2 < u3
12         // Speedup #1: If v3 ∈ N(u3), skip all the
              successive wedges with the same endpoints;
              also no need to check v3 ∉ N(u3) for such
              wedges
13         if v1 ∉ N(u1) and v3 ∉ N(u3) then
14             // Speedup #2: Reuse the count below for all
                  the successive green wedges with the same
                  pair of endpoints
15             foreach w3 ∈ W(u1, u3) do
16                 (u1, v2, u3) ← w3 // Red wedge in Fig. 3
17                 if v2 ∉ N(u2) then counts[w1]++
18 count ← sum(counts)                          // Parallel reduction
19 return count
```

We have two speedups for faster computation, mentioned in lines 12 and 14. In the first speedup, we aim to skip the processing of some green wedges with particular endpoints. Note that in W, the wedges with the same smaller-endpoint (u_1) are sorted with respect to their larger-endpoint (u_2). This means that while going over the green wedges (w_2) in line 10, where u_2 is the smaller-endpoint, we may encounter successive green wedges with the same pair of endpoints, u_2 and u_3 (where the center point (v_1) from V is different). In that case, if $v_3 \in N(u_3)$ happens to be true, we can skip processing all such successive green wedges with endpoints u_2, u_3 because the (u_3, v_3) edge violates the inducedness condition. We can do this by simply keeping a flag and temporary variable to remember the larger-endpoint (u_3) from the last processed green wedge. This way we do not check whether $v_3 \notin N(u_3)$ again and again. More importantly, if $v_3 \in N(u_3)$, we skip processing all the green wedges with the same pair of endpoints u_2, u_3 . In the second speedup, we again take advantage of the successive green wedges with the same pair of endpoints. We perform the computation in lines 15 to 17 once for a w_1, w_2 pair and reuse the induced 6-cycle count for all successive w_1, w_2' pairs where w_2 and w_2' share the same endpoints. We use both speedups in our implementation.

WedgeJoin computes the true count of induced 6-cycles by finding three wedges where: (1) **Nodes on the left are unique:** Lines 9 and 11 enforce uniqueness by establishing an ordering $u_1 < u_2 < u_3$; (2) **Nodes on the right are unique:** Each node on right is the center of a traversed wedge (lines 9, 11, and 16) which contain two endpoints—through the inducedness checks in lines 13 and 17, we prove uniqueness by checking if a pair of endpoints from all three traversed wedges connects to multiple nodes on right; (3) **The six induced edges exist** (the blue, green, and red edges in Figure 3)**:** Since all nodes are unique, each traversed wedge (lines 8, 10, and 15) contains two of the induced edges; (4) **The three non-induced edges do not exist** (the dashed black edges in Figure 3)**:** We have explicit conditions on lines 13 and 17 corresponding to the three inducedness checks. Finally, since we go over all triples of wedges, all the induced 6-cycles are counted.

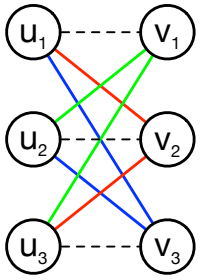

Figure 3: A cycle of three wedges (red, blue, and green) and the lack of any other edge are needed to form an induced 6-cycle. The dashed edges must be nonexistent; including any of those would make the 6-cycle non-induced.

Time complexity. Wedge retrieval (lines 2-6) traverses over all wedges which have endpoints in U and takes $O(|U| \cdot \langle d_U \rangle \cdot \langle d_V \rangle) = O(m \cdot \langle d_V \rangle)$ time. Starting in line 8, we iterate over all the wedges, taking $O(m \cdot \langle d_V \rangle)$ iterations. The loop on line 10 finds the wedges where a node $u \in U$ is the smaller endpoint, which corresponds to $O(\langle W_U \rangle)$ iterations. We find the third wedge in line 15, which takes $O(\langle d_U \rangle)$ iterations. Line 17 simply takes $O(1)$ time. Overall, WedgeJoin takes $O(m \cdot \langle d_V \rangle + m \cdot \langle d_V \rangle \cdot \langle W_U \rangle \cdot \langle d_U \rangle)$ which is equal to $\boldsymbol{O(m \cdot \langle d_V \rangle \cdot \langle W_U \rangle \cdot \langle d_U \rangle)}$ time (divided by p when parallelized). Comparing with NodeJoin, which has $O(m \cdot \langle d_U \rangle^2 \cdot \langle d_V \rangle^2)$ time, whether $\langle W_U \rangle$ is smaller than $\langle d_U \rangle \cdot \langle d_V \rangle$ determines if WedgeJoin is faster than NodeJoin. However, as we see in Section 7, even in real-world networks where $\langle W_U \rangle$ is larger than $\langle d_U \rangle \cdot \langle d_V \rangle$, the constant time speedups implemented in WedgeJoin causes WedgeJoin to run significantly faster than NodeJoin.

Space complexity. The global containers *counts* (line 7) and W (line 2) takes space equivalent to the number of wedges, which is much more than the $O(m)$ space required for the graph. Each wedge takes $O(1)$ space and there are $O(|W|)$ wedges in total, which also corresponds to the total space complexity of WedgeJoin.

6 NODE TRIPLETS FOR FASTER COUNTING

In this section, we propose a new technique that considers node triplets to count the induced 6-cycles. We define a node triplet to be a grouping of three unique nodes such that all nodes are in the same set (U or V) and there exists a 4-path connecting the three nodes. Inspired by Yang et al.'s approach for non-induced 6-cycles [40], we derive a formula to find the number of induced 6-cycles for a given node triplet and compute the total count by going over all node triplets. The formula lets us systematically avoid the duplicate work and engage in time-space tradeoffs for faster computation. We first introduce the TripletJoin algorithm in Section 6.1 which simply applies the formula for all node triplets and also stores the set of common neighbors for fast computation. Then, we present our final algorithm, BatchTripletJoin, in Section 6.2 which improves TripletJoin by storing common neighbors more efficiently and reducing set operations.

Algorithm 4: TripletJoin (G)

Input: $G\,(U, V, E)$: graph
Output: *count*: number of induced 6-cycles

1 $G \leftarrow$ Preprocessing (G)
2 $counts \leftarrow []$ // $|U|$ values
// For each node pair in U, common neighbors stored in S
3 $S \leftarrow \emptyset * |U|$ // $|U|$ hashmaps of nodes to sets
4 **parallel foreach** $u_1 \in U$ **do**
5 **foreach** $v_j \in N(u_1)$ **do**
6 **foreach** $u_i \in N(v_j)$ *s.t.* $u_i > u_1$ **do**
7 add v_j to $S[u_1][u_i]$
8 **parallel foreach** $u_1 \in U$ **do**
9 $H \leftarrow \emptyset$ // Distance-2 neighbors of u_1 with greater id
10 **foreach** $v_j \in N(u_1)$ **do**
11 **foreach** $u_i \in N(v_j)$ *s.t.* $u_i > u_1$ **do**
12 add u_i to H
13 **foreach** $u_2, u_3 \in H$ *s.t.* $u_3 > u_2$ **do**
14 **if** $u_3 \in S[u_2].keySet()$ **then**
15 $counts[u_1] \leftarrow counts[u_1] + (|S[u_1][u_2] \backslash N(u_3)|) \cdot (|S[u_1][u_3] \backslash N(u_2)|) \cdot (|S[u_2][u_3] \backslash N(u_1)|)$
16 $count \leftarrow sum(counts)$ // Parallel reduction
17 **return** *count*

6.1 Counting by Node Triplets

Node triplets offer a systematic way to count the induced 6-cycles. Given that there are exactly six edges in an induced 6-cycle and no two nodes share more than one neighbor, we can derive a formula to find the number of induced 6-cycles for a given node triplet:

Theorem 1. *Given a bipartite network $G = (U, V, E)$ and three unique nodes u_1, u_2, and $u_3 \in U$, the number of induced 6-cycles containing the node triplet (u_1, u_2, u_3) is:*

$$|N(u_1) \cap N(u_2) \backslash N(u_3)| \cdot |N(u_1) \cap N(u_3) \backslash N(u_2)| \cdot |N(u_2) \cap N(u_3) \backslash N(u_1)| \quad (1)$$

Proof. Let unique nodes v_1, v_2, and $v_3 \in V$ be in an induced 6-cycle with u_1, u_2, and u_3 as depicted in the induced 6-cycle of Figure 3. The difference between a 6-cycle and an induced 6-cycle is that, in an induced 6-cycle, neither of v_1, v_2, and v_3 can be a common neighbor of all three nodes u_1, u_2, and u_3. Therefore, the number of induced 6-cycles containing u_1, u_2, u_3, v_1, and v_3 is the number of possible v_2s which are neighbors of u_1 and u_3 but not u_2. This can be represented as $|N(u_1) \cap N(u_3) \backslash N(u_2)|$. Likewise, the number of possible v_1s and v_3s are $|N(u_2) \cap N(u_3) \backslash N(u_1)|$ and $|N(u_1) \cap N(u_2) \backslash N(u_3)|$, respectively. The sets of $\{N(u_1) \cap N(u_2) \backslash N(u_3)\}$, $\{N(u_1) \cap N(u_3) \backslash N(u_2)\}$, and $\{N(u_2) \cap N(u_3) \backslash N(u_1)\}$ are mutually exclusive. Therefore, multiplying the size of these three sets gives the number of induced 6-cycles for the node triplet u_1, u_2, u_3. □

Algorithm 4 outlines the TripletJoin algorithm. Given a bipartite graph $G = (U, V, E)$, TripletJoin computes the number of participating induced 6-cycles for all node triplets of U and returns the total sum. TripletJoin processes at most $\binom{|U|}{3}$ node triplets, of which many may share multiple nodes, such as the same pair of nodes $u_1, u_2 \in U$. This may cause serious recomputation of $N(u_1) \cap N(u_2)$, which corresponds to a significant runtime cost. Therefore, we store the common neighbors of node pairs in U, i.e., $N(u_1) \cap N(u_2)\ \forall\ u_1, u_2 \in U$, in container S (lines 3-7). Then, for each u_1, we use a container H (line 9) to store all its distance-2 neighbors which are greater than itself. Afterwards, we obtain node triplets by iterating over unique node pairs in H and compute the induced 6-cycle count of each by Theorem 1 (line 15). This process of finding node triplets avoids going over all $\binom{|U|}{3}$ triplets by only processing the three nodes which form a 4-path (lines 10- 12: u_1-v_x-u_2 and u_1-v_y-u_3 for arbitrary v_x and v_y). Such node triplets are more likely to be a part of an induced 6-cycle when compared to an arbitrary node triplet in U.

Time complexity. Lines 4-7 iterate through all wedges of the graph, which takes $O(|W|)$ time. Then, for each node in U (line 8), we find its distance-2 neighbors (lines 9-12), taking $O(m \cdot \langle d_V \rangle)$ time in total. Line 13 traverses through pairs of distance-2 neighbors, which takes, on average, $O(\binom{\langle d2_U \rangle}{2})$ time. Line 15 does a computation based on Theorem 1, which takes an average of $O(\langle d_U \rangle)$ time. Therefore, TripletJoin has a time complexity of $O(|W| + |U| \cdot \binom{\langle d2_U \rangle}{2} \cdot \langle d_U \rangle) = \boldsymbol{O(m \cdot \langle d2_U \rangle^2)}$. As we see in Section 7, the time complexity of TripletJoin is typically much smaller than both NodeJoin ($O(m \cdot \langle d_U \rangle^2 \cdot \langle d_V \rangle^2)$) and WedgeJoin ($O(m \cdot \langle d_V \rangle \cdot \langle W_U \rangle \cdot \langle d_U \rangle)$) because the number of distance-2 neighbors of a node is often much less than the number of wedges it has. The parallel time complexity of TripletJoin is $O(1/p \cdot m \cdot \langle d2_U \rangle^2)$ since it is embarrassingly parallel.

Space complexity. In addition to the $O(m)$ space required for the graph and the $O(|U|)$ space required for the container *counts* (line 2), TripletJoin involves storing wedges (line 3) in the global scope, which takes $O(|W|)$ space and determines the total space complexity. Note that the local storage of distance-2 neighbors of a node $u \in U$ (line 9) only takes $O(p \cdot \langle d2_U \rangle)$ space, which is surpassed by the global storage of wedges and thus does not increase the total space complexity.

6.2 Faster Triplet Counting with Less Space

Here we consider three orthogonal improvements on top of TripletJoin for a more time and space efficient algorithm.

Storing size of intersections. By globally storing wedges in WedgeJoin and set intersections in TripletJoin, we are able to solve the recomputation issue of wedges and set intersections, respectively. However, for large graphs, the space required for this storage is prohibitive and may exceed the amount of available memory. To reduce the memory usage, we can make a more efficient use of global storage across loop iterations and local storage within loop iterations. Since local storage is temporary, its memory is freed (and thus can be reallocated) after each iteration, unlike global storage. Compared to TripletJoin, which stores the set intersections in global storage, we can only store the *sizes* of set intersections globally (not the sets) and use local storage only for the set intersections which directly relate to the associated loop iteration.

Reduced set operations. Another improvement is about the computation of induced 6-cycle counts for a node triple. In an induced 6-cycle, each node $v \in V$ has exactly two edges. We can use this to improve upon Theorem 1 by eliminating the need for the

Algorithm 5: BatchTripletJoin (G)

Input: $G(U, V, E)$: graph
Output: *count*: number of induced 6-cycles

```
1  G ← Preprocessing (G)
2  counts ← []                                   // |U| values
   // For each node pair in U, # of common neighbors
      stored in S
3  S ← ∅ * |U|            // |U| hashmaps of nodes to values
4  parallel foreach u1 ∈ U do
5      foreach vj ∈ N(u1) do
6          foreach ui ∈ N(vj) s.t. ui > u1 do
7              S[u1][ui] ← S[u1][ui] + 1
8  parallel foreach u1 ∈ U do
9      H ← ∅                  // Hashmap of nodes to node sets
10     foreach vj ∈ N(u1) do
11         foreach ui ∈ N(v) s.t. ui > u1 do
12             add vj to H[ui]
13     foreach u2, u3 ∈ H.keySet() s.t. u3 > u2 do
14         if u3 ∈ S[u2].keySet() then
15             x ← |H[u2] ∩ H[u3]|
16             counts[u1] ← counts[u1]+
               (|H[u2]| − x) · (|H[u3]| − x) · (S[u2][u3] − x)
17 count ← sum(counts)                           // Parallel reduction
18 return count
```

set difference operation, reducing the number of computations. As shown in Theorem 2, we can instead subtract the number of nodes which are connected to all three nodes in a node triplet. Therefore, instead of computing three $O(|V|)$ set difference computations, we can simply compute one $O(|V|)$ set intersection computation.

Theorem 2. *Given a bipartite network $G = (U, V, E)$, three unique nodes $u_1, u_2, u_3 \in U$, and $x = |N(u_1) \cap N(u_2) \cap N(u_3)|$, the number of induced 6-cycles containing the node triplet is:*

$$(|N(u_1) \cap N(u_2)| - x) \cdot (|N(u_1) \cap N(u_3)| - x) \cdot (|N(u_2) \cap N(u_3)| - x) \quad (2)$$

Proof. Given a set X, $|X| - |X \cap A| = |X \setminus A|$. Therefore, $|N(u_1) \cap N(u_2)| - x$ is equivalent to $|N(u_1) \cap N(u_2) \setminus N(u_3)|$ in Theorem 1. As such, Theorem 2 is correct by the same reasoning as Theorem 1. □

We consider the improvements above in BatchTripletJoin, outlined in Algorithm 5, which reduces the number of computations and is more efficient than TripletJoin in terms of memory usage. In lines 4-7, we compute and globally store the sizes of the set intersections between the neighbor lists for all u_1, u_i node pairs. Afterwards, we iterate through all u_1s (line 8) and store the non-empty set intersections between the neighbor lists of u_1 and all $u_i \in U$ s.t. $u_i > u_1$ (lines 9-12). Finally, we count the number of induced 6-cycles associated with each node triplet $\{u_1, u_2, u_3\}$ by Theorem 2 (line 16) and return the sum of all the counts. In our implementation, we force the compiler to vectorize the inner loops and it gave a slight improvement on the largest networks.

Time complexity. BatchTripletJoin features three orthogonal improvements over TripletJoin but the time complexity does not change. Efficient usage of memory and reduction of set operations (line 16) only offer constant time speedup and thus does not affect the overall time complexity. Overall, the total time complexity of BatchTripletJoin is equal to TripletJoin, which is $O(m \cdot \langle d2_U \rangle^2)$.

Space complexity. BatchTripletJoin utilizes three global storage containers - one for storing the graph, one for the container *counts* (line 2), and one for the container S (line 3) - and one local container H (line 9). Storing the graph requires $O(m)$ space and *counts* uses $O(|U|)$ space to store $|U|$ values. S stores the size of the non-empty intersections of the neighbor lists of node pairs in U for which the space complexity is $O(|U| \cdot \langle d2_U \rangle)$. Note that in real-world networks, this is significantly smaller than the number of wedges, as we also show numerically in Section 7. The local container H stores edges and thus takes $O(p \cdot m)$ space. In total, the space complexity of BatchTripletJoin is $O(|U| \cdot \langle d2_U \rangle)$.

Table 1: Statistics of the real-world bipartite networks used in the experiments. *I6C* stands for number of induced 6-cycles.

Networks	$\|U\|$	$\|V\|$	m	*I6C*
DBLP (DB)	4,000,150	1,425,813	10,002,631	5.10 x 10^7
Github (GI)	56,555	123,345	440,237	1.37 x 10^{11}
IMDB (IM)	1,232,031	419,661	5,596,667	2.01 x 10^{10}
Kindle (KI)	1,406,890	430,530	3,205,467	5.20 x 10^9
Twitter (TW)	175,214	530,418	1,890,661	5.58 x 10^{11}
Movielens (ML)	69,878	10,677	10,000,054	1.69 x 10^{17}
Reuters (RE)	781,265	283,911	60,569,726	9.91 x 10^{18}
LiveJournal (LJ)	3,201,203	7,489,073	112,307,385	2.10 x 10^{18}

7 EXPERIMENTS

In this section, we evaluate our algorithms in Section 6 as well as the adaptations in Section 5 on real-world datasets from Konect [21] and ICON [3]. For brevity, we use NJ (NodeJoin), WJ (WedgeJoin), TJ (TripletJoin), and BTJ (BatchTripletJoin). Table 1 gives broad statistics of the datasets.

DBLP is the graph of authors and their papers [24]. Github connects users with their projects [9]. IMDB contains actors and the movies they played in [1]. Kindle is the network of books and the users who rated those books [18]. Twitter contains Twitter users and the tags they mentioned in their tweets [12]. Movielens is a network of users and the movies they rate [17]. Reuters contains story-word inclusions in Reuters news [23]. LiveJournal is the network of users and their group memberships [26].

We give the statistics of our real-world datasets after Preprocessing and the computed time complexities of our algorithms in Table

Net.	$\|U\|$	$\|V\|$	m	$\langle W_U \rangle$	$\langle d2_U \rangle$	NJ	WJ	TJ
DB	644K	1.92M	5.89M	12.5	5.67	4.67 x 10^9	2.07 x 10^9	**1.90 x 10^8**
GI	22.9K	34.3K	335K	1,423	813	**6.83 x 10^9**	6.80 x 10^{10}	2.21 x 10^{11}
IM	350K	497K	4.80M	347	282	**8.41 x 10^{10}**	2.20 x 10^{11}	3.81 x 10^{11}
KI	198K	370K	2.01M	231	201	**6.20 x 10^9**	2.58 x 10^{10}	8.14 x 10^{10}
TW	129K	138K	1.46M	161	117	2.07 x 10^{10}	2.80 x 10^{10}	**1.99 x 10^{10}**
ML	10.6K	69.9K	10.0M	222,291	4,589	1.83 x 10^{17}	3.01 x 10^{17}	**2.11 x 10^{14}**
RE	169K	781K	60.5M	20,920	899	4.65 x 10^{16}	3.51 x 10^{16}	**4.88 x 10^{13}**
LJ	2.19M	2.98M	107M	1,983	581	3.28 x 10^{14}	3.71 x 10^{14}	**3.61 x 10^{13}**

Table 2: Statistics of the networks after Preprocessing. For NodeJoin ($O(m \cdot \langle d_U \rangle^2 \cdot \langle d_V \rangle^2)$), WedgeJoin ($O(m \cdot \langle d_V \rangle \cdot \langle W_U \rangle \cdot \langle d_U \rangle)$), and TripletJoin ($O(m \cdot \langle d2_U \rangle^2)$), we give the numerical values for their time complexities and highlight the best in bold. Note that BatchTripletJoin has the same time complexity as TripletJoin.

Table 3: Runtime (in seconds) when using 1 and 52 threads. "–" denotes >24 hours. *RS* denotes the relative speedup of BatchTripletJoin over the second best algorithm on 52 threads.

	1 thread				52 threads				
Net.	NJ	WJ	TJ	BTJ	NJ	WJ	TJ	BTJ	*RS*
DB	15.5	8.87	7.78	4.88	3.00	2.30	2.72	1.65	1.39
GI	2,818	1,989	1,976	421	78.6	52.3	58.9	9.75	5.37
KI	1,167	782	795	224	31.9	21.0	41.2	6.30	3.34
IM	5,138	2,913	2,027	689	144	77.1	75.4	18.7	4.04
TW	1,615	524	226	81.7	43.6	13.7	9.82	2.47	3.98
ML	–	–	–	43,850	–	–	4,991	890	5.61
RE	–	–	–	85,229	–	–	12,822	1,890	6.78
LJ	–	–	–	–	–	–	30,035	4,682	6.42

2. TripletJoin (BatchTripletJoin) has the best time complexity for five of eight datasets.

All experiments are performed on a Linux operating system running on a machine with Intel Xeon Gold processor at 2.1 GHz and 1125GB DDR4 memory. The processor contains 4 sockets with each having 13 cores for a total of 52 cores. We implemented our algorithms in C++ with Intel TBB 2020.2 [29] and OpenMP 4.5 [10] and compiled using GCC 10.2.0 at the -O3 level. **Our implementation of all the algorithms is available at https://tinyurl.com/par6cycle-code.** We terminated the computation if it took more than 24 hours to finish, denoted by "-" in the results. We also denote the computations that go out of memory by "OOM".

We first consider the strong scaling performance of our algorithms in Section 7.1. Then, we analyze the weak scaling behavior in Section 7.2. Next, we look at the memory usage in Section 7.3. Lastly, we examine the impact of Preprocessing and how different choices translate to improvements in runtime in Section 7.4.

7.1 Strong Scaling Experiments

Here we provide the strong scaling experiments for all algorithms. Table 3 shows our runtime experiments on real-world networks using 1 and 52 threads. We also show the speedup of BatchTripletJoin compared to the second best algorithm in terms of runtime on 52 threads. NodeJoin and WedgeJoin are not able to finish computation even with 52 threads on the three largest networks: Movielens, Reuters, and LiveJournal. The runtimes for WedgeJoin are typically better than NodeJoin as suggested by the numerical calculation of time complexities in Table 2. For TripletJoin, the three largest networks timed out on a single thread but was completed on 52 threads. BatchTripletJoin has the best sequential and parallel runtimes. The only configuration where it could not finish the computation in 24 hours is the sequential run on LiveJournal. On the largest network with 112M edges (LiveJournal), BatchTripletJoin is able to finish the computation in 78 mins by using 52 threads. It is 6.4× faster than the next best algorithm, TripletJoin.

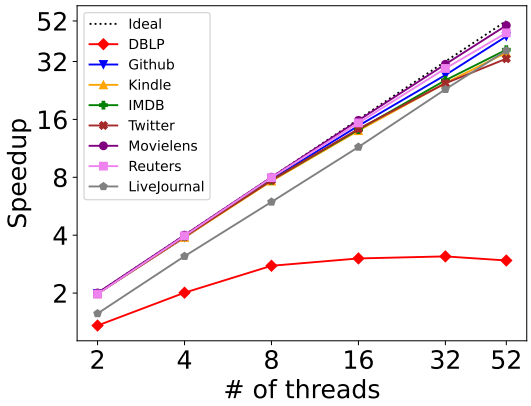

Figure 4: Speedup of BatchTripletJoin for strong scaling experiments on 2, 4, 8, 16, 32, and 52 threads when compared to a single thread. We also show the ideal speedup as a dotted line.

In Figure 4, we plot the speedup of BatchTripletJoin on 2, 4, 8, 16, 32, and 52 threads. To show the speedup on LiveJournal, we ran BatchTripletJoin until completion, which took approximately 47.4 hours with a single thread. Comparing 52 threads to a single thread, there is approximately a 3x speedup for DBLP, 33x speedup for Twitter, 36x speedup for Kindle, IMDB, and LiveJournal, 44x speedup for Github and Reuters, and a 49x speedup for Movielens. DBLP, the network with the smallest induced 6-cycle count, does not scale after 8 threads. The larger networks, such as Movielens, Reuters, and LiveJournal, do not have a significant decline in scalability when using 52 threads, suggesting that they would continue scaling for even larger number of threads. Github, Movielens, and Reuters achieved the highest speedup due to their large induced 6-cycle counts in relation to graph size (Table 1), indicating a dense induced 6-cycle structure. With the speedup numbers consistently over 32x on 52 threads for the networks with the largest induced 6-cycle counts, BatchTripletJoin exhibits strong scalability.

7.2 Weak Scaling Experiments

Here we provide the weak scaling experiments for all algorithms. For weak scaling, we consider x duplicates of the original network when running on x number of threads. Runtime results on 52 threads for Github, Kindle, IMDB, Twitter, and Movielens are shown in Table 4. All the algorithms timed out in 24 hours for Reuters and LiveJournal on 52 threads. Also, DBLP achieved poor weak scaling results since Preprocessing accounts for over 90% of the time spent (more details in Table 6). BatchTripletJoin is the only algorithm that could compute the duplicated Movielens network, which has 520M edges, in under 24 hours (13.2 hours). WedgeJoin was unable to process the weak scaling experiments on 52 threads for Kindle, IMDB, and Movielens due to prohibitive space complexity. BatchTripletJoin outperforms the other algorithms significantly in terms of runtime.

We plotted the weak scaling behavior of BatchTripletJoin on 2, 4, 8, 16, 32, and 52 threads in Figure 5. We computed the ratio of runtime using x threads on the duplicated network to the sequential runtime on the original network. BatchTripletJoin performs best on the networks with the highest proportion of induced 6-cycles to graph size, Github and Movielens. Having approximately 60%

Alg.	GI*	KI*	IM*	TW*	ML*
NJ	4,152	1,695	7,534	2,429	–
WJ	2,845	*OOM*	*OOM*	850	*OOM*
TJ	3,264	2,352	3,529	573	–
BTJ	512	357	1,084	129	47,530

Table 4: Runtime (in seconds) of our algorithms on 52 duplicates of the original dataset using 52 threads. "–" denotes >24 hours and "*OOM*" means the computation run out of memory.

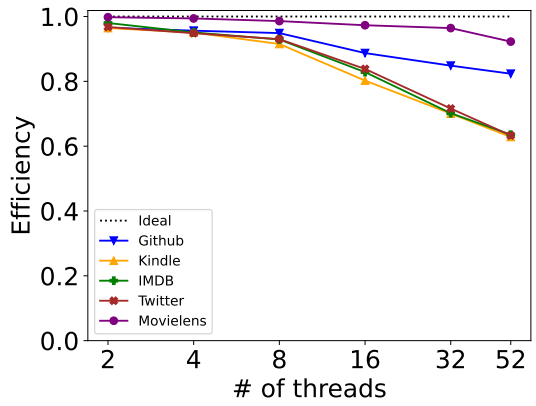

Figure 5: Efficiency of BatchTripletJoin for weak scaling experiments on 2, 4, 8, 16, 32, and 52 threads when compared to a single thread. We also show the ideal efficiency (= 1.0) as a dotted line.

efficiency or better even when the graph is 52 times the size of the original, BatchTripletJoin can be considered to be scalable in terms of weak scaling although there is room for improvement.

Table 5: Memory used (in gigabytes). "–" indicates that the experiment timed out after 24 hours.

Alg.	DB	GI	KI	IM	TW	ML	RE	LJ
NJ	0.54	1.54	0.69	1.52	0.56	-	-	-
WJ	0.70	0.67	1.10	2.68	0.51	-	-	-
TJ	0.71	1.64	3.47	8.53	1.39	17.1	32.6	125
BTJ	1.64	0.45	1.15	2.36	0.65	3.44	14.1	36.7

7.3 Memory Experiments

Here we give the memory usage results. We used the system activity reporter utility (SAR) and run it in the background during the computation. We measure the amount of used space in our machine every second and report the maximum amount used in Table 5. Unlike the other algorithms, NodeJoin does not use any global storage across parallel threads, and thus typically requires the least memory. BatchTripletJoin has a significantly smaller memory footprint than TripletJoin thanks to globally storing the sizes of set intersections instead of the entire set. This is also in line with the space complexities of the algorithms.

7.4 Analyzing the Preprocessing

We analyze the impact of the various techniques used in Preprocessing for the best performing algorithm BatchTripletJoin. As seen in Table 2, which lists the size and statistics of the graphs after preprocessing, there is a significant reduction in graph sizes which directly impacts the runtime of the algorithms.

We first look at how much time Preprocessing takes. Table 6 shows the proportion of runtime spent on Preprocessing for BatchTripletJoin on 52 threads. For DBLP, the dataset with the smallest number of induced 6-cycles, the majority of the time was spent on Preprocessing. Note that DBLP is the second largest network in terms of node count and has the third highest number of edges, so applying Preprocessing on the nodes and edges takes a significant amount of time compared to traversing the relatively small number of induced 6-cycles. This is the opposite for all the other networks - Preprocessing takes minimal time compared to the rest of the computation. It is even negligible for the three

Table 6: Fraction of runtime spent on Preprocessing for BatchTripletJoin on 52 threads.

DB	GI	KI	IM	TW	ML	RE	LJ
0.918	0.003	0.055	0.021	0.058	<0.001	0.001	0.002

Table 7: Ablation study for the three techniques in Preprocessing. Runtime results (in seconds) for BatchTripletJoin on 52 threads (best in bold). "–" denotes >24 hours.

Omission	DB	GI	KI	IM	TW	ML	RE	LJ
None	1.65	**9.75**	**6.30**	18.7	**2.47**	**890**	**1,890**	**4,682**
2-Core	2.39	10.9	8.11	18.6	1,809	943	2,059	–
Node Set Swaps	10.8	10.1	22.9	**6.67**	1,630	–	–	–
Node Reordering	**1.50**	11.7	8.04	20.6	3.86	967	5,189	50,109

largest networks - Movielens, Reuters, and LiveJournal. Preprocessing is the most useful when there is a significant number of induced 6-cycles in the graph, which is true for most networks in our dataset. Handling the networks with low induced 6-cycle density remains a challenge, as exemplified by DBLP.

Next, we perform an ablation study for the three techniques in Preprocessing. Table 7 shows the runtime on 52 threads when a section of Preprocessing is skipped. Applying all the techniques (denoted by "None") achieves the best result for six of the eight networks, including the three largest (Movielens, Reuters, and LiveJournal). With larger networks, the runtime savings from each Preprocessing section is more significant, in particular 2-core filtering and swapping node sets provide drastic gains.

Lastly, we check the impact of different node ranking choices. In line 3 of Preprocessing, we perform increasing wedge ordering on U. To test its performance compared to other ordering schemes, we conduct experiments using degree, degeneracy, and wedge orderings. Degree ordering ranks the nodes based on their degrees. Degeneracy ordering is an ordering of vertices given by repeatedly finding and removing vertices of smallest degree, also known as ordering by core numbers. Wedge ordering uses the number of 2-paths from each node. Table 8 shows the runtime with 52 threads on the decreasing and increasing versions of each of those ordering schemes. We have also included the runtime when no ordering is implemented, denoted as "None", to measure the impact of node ordering on speed. The increasing versions of each ordering scheme typically outperform the decreasing versions. In the increasing versions, the amount of work (i.e., number of induced 6-cycles) is more evenly distributed across parallel threads, preventing the runtime of one thread to dominate over the others. For example, in increasing degree ordering, the highest degree node is likely to participate in

Net.	None	Degree		Degeneracy		Wedge	
		Dec	Inc	Dec	Inc	Dec	Inc
DB	**1.50**	1.64	1.65	2.91	2.60	1.63	1.65
GI	11.7	10.9	10.7	10.7	10.9	10.5	**9.75**
KI	8.04	7.77	6.87	8.18	7.33	7.66	**6.30**
IM	20.6	21.0	18.8	22.6	20.5	21.0	**18.7**
TW	3.86	7.30	2.58	5.29	4.40	7.33	**2.47**
ML	967	909	932	968	989	**879**	890
RE	5,189	16,406	2,644	3,231	3,097	12,219	**1,890**
LJ	50,109	–	5,085	23,572	17,074	–	**4,682**

Table 8: Runtime (in seconds) for different orderings for BatchTripletJoin on 52 threads (best in bold). We show the decreasing and increasing variants for each. "–" denotes >24 hours.

more induced 6-cycles compared to a lower degree node. Since we process induced 6-cycles based on the node with minimum id and higher degree nodes are assigned a higher id, we process a lower proportion of induced 6-cycles for higher degree nodes in their parallel threads (and vice versa). Comparing individual ordering schemes, increasing wedge ordering outperforms the other ordering schemes in six of eight networks, including the two largest networks (`Reuters` and `LiveJournal`), which is why we consider it as the default ordering in Preprocessing.

8 CONCLUSION AND FUTURE WORK

We introduced efficient and scalable parallel algorithms to count induced 6-cycles in bipartite networks. To the best of our knowledge, this is the first inquiry in induced 6-cycle counting. Experiments on real-world bipartite networks show that our best algorithm, BatchTripletJoin, is highly parallelizable in relation to the number of processors and enables practical computation for large networks with up to half a billion edges; on the 52 times scaled `Movielens` network with a total of $520M$ edges, BatchTripletJoin finishes the computation in 13.2 hours by using 52 threads.

Although BatchTripletJoin exhibits strong performance, it is unable to compute some large networks in under 24 hours with 52 threads, such as the 52 times scaled `Reuters` and `LiveJournal` networks (3B-5B edges). It also shows poor scalability when the network has relatively few induced 6-cycles, as in the `DBLP` network. As a future work, we will investigate scaling our algorithm to larger networks with billions of edges. We will also extend our methods to handle the networks with low induced 6-cycle counts. One interesting question in this context is how quickly one can terminate the computation if the graph has no induced 6-cycles.

ACKNOWLEDGMENTS

This research was supported by NSF awards OAC-1845840 and OAC-2107089, and used resources from the Center for Computational Research at the University at Buffalo [2].

REFERENCES

[1] 2016. IMDb. https://www.imdb.com/interfaces/.
[2] 2021. Center for Computational Research, University at Buffalo. http://hdl.handle.net/10477/79221.
[3] Ellen Tucker Aaron Clauset and Matthias Sainz. 2016. The Colorado Index of Complex Networks. https://icon.colorado.edu/.
[4] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, Nick G Duffield, and Theodore L Willke. 2017. Graphlet decomposition: Framework, algorithms, and applications. *Knowledge and Information Systems* 50, 3 (2017), 689–722.
[5] Sinan G Aksoy, Tamara G Kolda, and Ali Pinar. 2017. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks* 5, 4 (2017), 581–603.
[6] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).
[7] Austin R Benson, Rediet Abebe, Michael T Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences* 115, 48 (2018), E11221–E11230.
[8] S. P. Borgatti and M. G. Everett. 1997. Network analysis of 2-mode data. *Social Networks* 19, 3 (1997), 243 – 269.
[9] Scott Chacon. 2009. The 2009 GitHub contest. https://github.com/blog/466-the-2009-github-contest.
[10] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.
[11] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.
[12] Munmun De Choudhury, Yu-Ru Lin, Hari Sundaram, Kasim Selcuk Candan, Lexing Xie, and Aisling Kelliher. 2010. How does the data sampling strategy impact the discovery of information diffusion in social media?. In *Fourth international AAAI conference on weblogs and social media*.
[13] Ali Dehghan and Amir H Banihashemi. 2019. Counting Short Cycles in Bipartite Graphs: A Fast Technique/Algorithm and a Hardness Result. *IEEE Transactions on Communications* 68, 3 (2019), 1378–1390.
[14] I.S. Dhillon. 2001. Co-clustering Documents and Words Using Bipartite Spectral Graph Partitioning. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '01)*. 269–274.
[15] D. C. Fain and J. O. Pedersen. 2006. Sponsored search: A brief history. *Bulletin of the American Society for Information Science and Technology* 32, 2 (2006), 12–13.
[16] D. Gibson, R. Kumar, and A. Tomkins. 2005. Discovering Large Dense Subgraphs in Massive Graphs. In *VLDB*. 721–732.
[17] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems* 5, 4 (2015), 1–19.
[18] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*. 507–517.
[19] Rajgopal Kannan, Viktor K Prasanna, Cesar Augusto Fonticielha De Rose, et al. 2020. RECEIPT: REfine Coarse-grained IndePendent Tasks for Parallel Tip decomposition of Bipartite Graphs. *Proceedings of the VLDB Endowment.* (2020).
[20] Mehdi Karimi and Amir H Banihashemi. 2012. Message-passing algorithms for counting short cycles in a graph. *IEEE Transactions on Communications* 61, 2 (2012), 485–495.
[21] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on World Wide Web*. 1343–1350.
[22] M. Latapy, C. Magnien, and N. Del Vecchio. 2008. Basic notions for the analysis of large two-mode networks. *Social Networks* 30, 1 (2008), 31 – 48.
[23] David D Lewis, Yiming Yang, Tony Russell-Rose, and Fan Li. 2004. Rcv1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research* 5, Apr (2004), 361–397.
[24] Michael Ley. 2002. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *International symposium on string processing and information retrieval*. Springer, 1–10.
[25] Xin Li and Hsinchun Chen. 2013. Recommendation as link prediction in bipartite graphs: A graph kernel-based machine learning approach. *Decision Support Systems* 54, 2 (2013), 880–890.
[26] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proc. of the 7th ACM SIGCOMM conference on Internet measurement*. 29–42.
[27] Mark EJ Newman. 2001. Scientific collaboration networks. I. Network construction and fundamental results. *Physical Review E* 64, 1 (2001), 016131.
[28] T. Opsahl. 2013. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. *Social Networks* 35, 2 (2013), 159 – 167.
[29] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.
[30] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* 23, 2 (2007), e177–e183.
[31] Garry Robins and Malcolm Alexander. 2004. Small worlds among interlocking directors: Network structure and distance in bipartite graphs. *Computational & Mathematical Organization Theory* 10, 1 (2004), 69–94.
[32] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2150–2159.
[33] Seyed-Vahid Sanei-Mehri, Yu Zhang, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. 2019. FLEET: butterfly estimation from a bipartite graph stream. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1201–1210.
[34] Ahmet Erdem Sarıyüce and Ali Pinar. 2018. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 504–512.
[35] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. 2013. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 2013 SIAM international conference on data mining*. SIAM, 10–18.
[36] C Seshadhri and Srikanta Tirthapura. 2019. Scalable subgraph counting: The methods behind the madness: WWW 2019 tutorial. In *Proceedings of the Web Conference (WWW)*, Vol. 2. 75.
[37] Jessica Shi and Julian Shun. 2020. Parallel algorithms for butterfly computations. In *Symposium on Algorithmic Principles of Computer Systems*. SIAM, 16–30.
[38] Jia Wang, Ada Wai-Chee Fu, and James Cheng. 2014. Rectangle counting in large bipartite graphs. In *2014 IEEE International Congress on Big Data*. IEEE, 17–24.
[39] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (1998), 440–442.
[40] Yixing Yang, Yixiang Fang, Maria E Orlowska, Wenjie Zhang, and Xuemin Lin. 2021. Efficient Bi-triangle Counting for Large Bipartite Networks. *PVLDB* 14, 6 (2021), 984–996.
[41] Qiuyu Zhu, Jiahong Zheng, Hao Yang, Chen Chen, Xiaoyang Wang, and Ying Zhang. 2020. Hurricane in bipartite graphs: The lethal nodes of butterflies. In *32nd International Conference on Scientific and Statistical Database Management*. 1–4.