Parallel Batch-Dynamic 3-Vertex Subgraph Maintenance

Ho Tin Fan, Alvin Lu

Abstract

Counting certain subgraphs is a fundamental problem that is crucial in recognizing patterns in large graphs, such as social networks and biological interactomes. However, many real world graphs are constantly evolving and are subject to changes over time, and previous work on efficient parallel subgraph counting algorithms either do not support dynamic modifications or do not extend to general subgraphs. This paper presents a theoretically-efficient and demonstrably fast algorithm for parallel batch-dynamic 3*-vertex subgraph counting, and the underlying data structure can be extended to counting* 4*-vertex subgraph counts as well. The algorithm maintains the h-index of the graph, or the maximum h such that the graph contains h vertices with degree at least h, and uses this to update subgraph counts through an efficient traversal of two-paths, or wedges. For a batch of size b, the algorithm takes* $O(bh)$ *expected amortized work and* $O(\log(bh))$ *span with high probability.*

1. Introduction

Graphs are often used to represent complex networks of interactions, and subgraph counting is a fundamental problem that allows us to find specific patterns within these interactions. They can be crucial in social media and advertising networks, which find patterns that connect users with people they may know and suggest content similar to what has been previously viewed [18]. Other applications include searching for common interactions between biomolecules in interactomes for medical purposes, and coordinating transportation and travel routes to reduce cost and time [18]. Moreover, constant streams of data make these real-world graphs quickly evolve and change over short periods of time, so developing a dynamic model that supports real-time updates is important. In addition, most modern computers have multiple processors, and since serial algorithms are unable to efficiently utilize all available resources when running on large real-world graphs, which can often contain millions of subgraphs, it is imperative to develop efficient parallel algorithms. In particular, we focus on shared-memory parallelism, where multiple threads can simultaneously access the same memory, because all publicly available graphs can fit in a commodity shared-memory machine. In this paper, we design a new algorithm that maintains 3-vertex subgraph counts in a batch-parallel and dynamic setting. For example, given a graph G and a batch of updates (including edge insertions and deletions), we find, in parallel, the new number of triangles.

Previous work on dynamic subgraph counting has focused mostly on static graphs, such as Ahmed *et al.*'s Parallel Graphlet Decomposition (PGD) framework that counts the number of 4-vertex subgraphs [1], and Pinar *et al.*'s sequential algorithm ESCAPE which counts all 5-vertex subgraphs [17]. In under 5 minutes, ESCAPE is able to find the exact number of all 5-vertex subgraphs on a 6 core machine in a graph with over 21 million edges. However, static graphs cannot support real-time updates without recomputation, which could be expensive on large graphs that constantly change over time. Dhulipala *et al.* developed an algorithm for parallel batch-dynamic k-clique counting in $O(b\sqrt{b+m})$ expected work and $O(log(b+m))$ span w.h.p. with b as the size

of the batch and m as the number of edges in the graph, [1] but it does not extend to general subgraphs [7]. Our algorithm is based on the sequential algorithm by Eppstein and Spiro, which maintains 3-vertex subgraph counts in a dynamic setting in $O(h)$ amortized time for each edge insertion or deletion, where h is the **h-index** of a graph [10]. We define the h-index as the largest possible h such that there are at least h vertices with degree greater than or equal to h. In their paper, they also prove that h must be between m/n and $\sqrt{2m}$.

In this paper, we develop a batch-parallel dynamic algorithm to maintain the count of 3-vertex subgraphs, and give a complete evaluation for triangles specifically. The algorithm is composed of two parts. The first part dynamically maintains the h-index of a graph, and keeps track of the set H, which stores the h vertices with degree at least h. Given a batch of b edges to insert or delete, we recompute the h-index in $O(b)$ expected work and $O(\log b)$ span w.h.p. H is then used to optimize triangle counting in the second part. Using the set H, the algorithm, which would have taken $O(bn)$ expected work and $O(log(bn))$ span w.h.p. where n is the number of vertices in the graph, is reduced to $O(bh)$ expected amortized work and $O(log(bh))$ span w.h.p.

We implemented our framework using the Graph Based Benchmark Suite library (GBBS) [8], and explored several different data structures for practical optimizations, specifically for computing the h-index. One option is to use a dynamic array of dynamic arrays. This would take up space proportional to the largest degree. Another method would be to store a combination of a dynamic array of dynamic arrays and a parallel hash table of dynamic arrays. If a vertex has a degree less than or equal to a given threshold, it is stored in the dynamic array of dynamic arrays. Otherwise, it is stored in the parallel hash table. This would use memory proportional to the number of different degrees in the graph, which is bounded by the largest degree. Our experiments showed that using the parallel hash table was slightly faster in most cases.

We tested our implementation using a 30-core machine with two-way hyperthreading, on real-world graphs from the Stanford Network Analysis Project (SNAP) [12]. Our framework counted triangles for graphs up to 3 million edges in under 7 minutes. It was then able to maintain updates with 100 batches of 100,000 edges each in less than 5 minutes.

In summary, we develop:

1. A new batch-parallel algorithm for dynamic 3-vertex subgraph counting
2. Strong theoretical guarantees for running time
3. A practical implementation that can be extended to any 3-vertex subgraph
4. An experimental evaluation of dynamic triangle counting on real-world graphs using a 30-core instance with 2-way hyperthreading

All of our code can be found at `https://github.com/CodeTiger927/gbbs`.

2. Related Work

There has been prior work on subgraph counting, mostly focusing on static graphs. In particular, there has been extensive work on triangle counting algorithms. Shun and Tangwongsan developed a shared-memory parallel algorithm to count and approximate the number of triangles in large scale graphs [21]. For the Yahoo graph with 6 billion edges, it takes less than 1.5 minutes to find the exact

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

number of triangles, and under 10 seconds to approximate the number to 99.6% accuracy. Zhang *et al.* designed the LiteTE framework which also counts triangles but under distributed memory model instead [24]. Makkar *et al.* created a dynamic parallel GPU algorithm to count triangles [13]. While it takes less time to update and find the exact number of triangles than our algorithm does, it cannot be extended to other subgraphs. In addition to these, there has been much other work on triangle counting such as [15, 22, 23, 3, 16, 4].

Subgraph counting has also been extended to larger subgraphs as well. Ahmed *et al.* [1] designed an algorithm to count 4-vertex subgraphs in parallel, while Elenberg *et al.* [9] achieved similar results using a different framework (GraphLab PowerGraph). Pinar *et al.*'s sequential ESCAPE framework finds exact counts for all 5-vertex subgraphs [17]. There has also been work on (2, 2)-biclique counting, such as Sanei-Mehri *et al.*'s sequential implementation [19] and Shi and Shun's parallel algorithm and framework [20]. Dhulipala *et al.* introduced a batch-parallel algorithm for finding k-cliques in static graphs in $O(b\sqrt{b+m})$ expected work and $O(log(b+m))$ span w.h.p. where b is the size of the batch being updated and m is the number of edges in the graph [7]. However, their algorithm for clique counting cannot be easily generalized to other kinds of subgraphs.

3. Preliminaries

In this section, we introduce the notations we use throughout this paper, the models we compute under, and primitive functions that we use in our algorithm.

Graph Notation We represent a graph $G = (V, E)$ to be an unweighted symmetric graph with a set of n vertices V and a set of m edges E. Each vertex $v \in G$ has a set of neighbors denoted as $N(v)$ and a degree of $\text{DEG}(v)$. Our algorithm also uses a specific subgraph known as a **wedge**, or 2-path. We call the two vertices at the ends of the path as **endpoints**, and the vertex in the middle as the **center node**. Finally, $batch$ is an update with size b, and it will be specified whether it represents a collection of vertices or edges.

Shared Memory Model We use the shared memory model where multiple processors share the same memory address space. Parallelism is achieved by allowing concurrent access to memory in a single step [5].

Work and Span Model We represent our parallel computations in a directed acyclic graph (DAG) where each node is an operation, and an outgoing edge to another node shows dependency. The **work** of our algorithm is defined as the total number of operations or vertices in the DAG. The **span** is the maximum number of operations on a dependency chain or the longest path in the DAG. We define our algorithm to be **work-efficient** if its work is the same as the best sequential algorithm for the specific problem. Finally, our running time is bounded by $work/P + O(span)$ where P is the number of processors [6].

Primitives We assume atomic **compare-and-swap (CAS)** for our algorithms. We use the following parallel primitives in our algorithm. **Reduce** takes in an array of size n and a binary associative function f where it computes f over all elements in the array with $O(n)$ work and $O(\log n)$ span. Given an array of elements with size n, **prefix sum** returns an array of the same size but each element in the new array is equal to the sum of all elements before it in the old one. It computes in span of $O(\log n)$ and work of $O(n)$. **Filter** takes in an array of elements and a boolean function f, and filters out all elements v, such that $f(v)$ is false. The ordering of the remaining elements is maintained. This also takes $O(n)$ work and $O(\log n)$ span. **Radix/Integer sort** sorts an array of n

integers in ascending or descending order in $O(n)$ expected work and $O(\log n)$ span w.h.p. [14]. Finally, **parallel hashing** hashes a list of n elements in $O(n)$ expected work and $O(\log n)$ span w.h.p. to achieve fast random access [11].

4. Dynamic Parallel H-index

In this section, we introduce our batch-parallel dynamic algorithm for maintaining the h-set of a graph. This is then used for our batch-parallel dynamic triangle counting algorithm, which is described in Section 5.

We prove strong theoretical guarantees for our batch-parallel dynamic algorithm that maintains the h-index of a graph. The h-index of a graph or h is the largest number such that there are h vertices with degree at least h. The set H stores these h vertices. In our dynamic h-set framework, we support two functions: INSERT adds a batch of vertices in parallel, and ERASE deletes a batch of vertices. When we insert or delete an edge and change the degrees of vertices, we first ERASE them, update the degrees of the vertices based on the modified edges, and finally INSERT the vertices again. To accomplish this, we maintain the following data structures.

- A parallel hash table C that maps each degree to a dynamic array of vertices with that degree. So, $C[i] := \{v \in G \mid \text{DEG}(v) = i\}$.
- The integer h represents the largest possible number such that there are at least h vertices with degree greater than h.
- The implicit set H which stores h vertices such that for all v in H, $\text{DEG}(v) \geq h$. This set is not explicitly maintained, but is implicitly taken to be the h vertices with highest degree in C, and we refer to H throughout this paper.
- The size t representing the number of vertices v such that $v \in H \cap C[h]$. These vertices prevent h from increasing.

With these data structures, we now describe the algorithm to maintain and use them for inserting and removing vertices from h-set.

Theorem 4.1. *Given a batch of b vertex insertions or deletions, the h-set functions* INSERT *and* ERASE *take $O(b)$ expected work and $O(\log b)$ span with high probability to update h.*

4.1. Updating C

During both insertion and deletion of vertices from h-set, we need to update all the data structures, specifically C as it needs to keep track of all vertices. Algorithm 1 allows us to insert a *batch* of vertices to the data structure C in parallel. We first sort the batch based on degree in descending order (Line 2). Then, we look for blocks of vertices with the same degree, storing indices which end a block. Inside an array of size b for *batch*'s indices, *idx*, we store all indices from 0 to $b-1$ (Lines 3). We then filter *idx* so we only have indices at the boundaries of the blocks (the vertex at that index has a different degree than the next one) (Line 4). We iterate through *idx* in parallel, and the *start* of each block is $idx[i-1]+1$ or 0 if i is 0. The *end* is $idx[i]$ (Lines 5–10). We then add all elements between $batch[start]$ and $batch[end]$ inclusive to the appropriate entry in C (Lines 11–12). The sort of batch and later on, the filtering of *idx* (which also has size b) (Lines 2–4) limits the time complexity to $O(b)$ expected work and $O(\log(b))$ span w.h.p. The parallel for loop (Lines 5–12)

Algorithm 1 C Updates

```
 1: procedure ADDTOC(batch, G)            ▷ Takes in batch of vertices to add and the graph G
 2:     Sort batch based on degree in descending order
 3:     Initialize array idx to be all the indices from 0 to b − 1
 4:     Filter idx so that the only remaining entries are indices of vertices in batch with a different
    degree than the vertex in the next index
 5:     for (parallel) i = 0...|idx| do
 6:         if i = 0 then
 7:             Create and initialize the index start ← 0
 8:         else
 9:             Create and initialize the index start ← idx[i − 1] + 1
10:         Create and initialize the index end ← idx[i]
11:         degree ← batch[start]
12:         Add all elements of from batch[start] to batch[end] to C[degree] in parallel
```

only takes $O(b)$ work and $O(1)$ span because there are a total of b elements to add. Thus, the total bounds are $O(b)$ expected work and $O(\log(b))$ span w.h.p.

Deletion of elements from C follows similarly, and we define a procedure REMOVEFROMC to be the exact same as ADDTOC except that Line 12 removes, instead of adding, all elements between $batch[start]$ and $batch[end]$ from the corresponding entry in C using a parallel filter. The time complexity is the same as that of ADDTOC.

4.2. h-set Insertions

For an insertion of *batch* to h-set, as shown in Algorithm 2, we start by sorting *batch* in descending order by degree (Line 2). In *aboveH*, we store the number of vertices with degree greater than or equal to h, which is of size $h + |C[h]| - t$. We update *aboveH* to include the new vertices in batch with degree at least h with a serial binary search on *batch* (Lines 3–4). The new vertices are then added to C (Line 5) with the procedure described in Algorithm 1. We create a sequence *sum* where each number at index i, is the number of vertices that can no longer be in H if h is increased by i. To do this, we run a parallel prefix sum on the sizes of each entry in C from h to $h+b$ inclusive, and store the resulting array in *sum* (Lines 6–9). For each index, we can increase h if *aboveH* minus the number of vertices lost by increasing it is at least the new h. We create a boolean array M in parallel with the same size as *sum* where each value at i is the boolean value of $(aboveH - sum[i] \geq h + i)$ (Line 10). The index of the first false is when we can no longer increase h. To find this index, we store an array of indices, *indexM*, where each value is $b+1$ (or any number larger than the maximum index) if the entry of M is true, or the index of the entry if it is false (Line 11). Using a parallel reduce on *indexM*, we find the smallest index of a false. We store that index in *increase*, and we add *increase* to h (Lines 12–13). The number of vertices above the new h is *aboveH* minus the number of vertices lost which is $sum[increase - 1]$ or 0 if h did not increase (Lines 14–15). t is then $h + |C[h]| - aboveH$ (Line 16).

The initial sort of batch on Line 2 takes $O(b)$ expected work and $O(\log b)$ span w.h.p. Since the binary search used on Line 4 is serial, both the work and span are $O(\log b)$. The update to C (Line 5) has time complexity $O(b)$ expected work and $O(\log b)$ span w.h.p as shown previously in Section 4.1. The prefix sum on Line 9 is computed over *sum* which has size at b, meaning it is also done in $O(b)$ work and $O(\log b)$ span. Creating the arrays M and *indexM* both only take $O(b)$ work and

Algorithm 2 h-set Insertion

```
1:  procedure INSERT(batch, G)                         ▷ Takes in batch of vertices and graph G
2:      Sort batch in descending order based on the vertex degree
3:      Create and initialize aboveH ← h + |C[h]| − t
4:      Add to aboveH the number of vertices in batch that have degree greater than the current h
    (with a serial binary search)
5:      ADDTOC(batch, G)                                              ▷ Add vertices to C
6:      Create an array of numbers sum and initialize each value to 0
7:      for (parallel) each entry F from C[h] to C[h+b] do
8:          Set the corresponding entry in sum to be |F| if F exists
9:      PREFIXSUM(sum)                                                ▷ In-place prefix sum
10:     Create boolean array M and initialize M[i] to (aboveH − sum[i] ≥ h + i + 1) for each index
    i in M
                                                              ▷ Find index of the first false
11:     Create an array of numbers indexM and initialize indexM[i] to i if M[i] is false, otherwise
    to b + 1 (or any large sentinel value)
12:     Create and initialize increase ← smallest index in indexM (use a parallel reduce)    ▷ Find
    smallest number
13:     h ← h + increase
                                                                              ▷ Update t
14:     if increase > 0 then
15:         aboveH ← aboveH − sum[increase − 1]        ▷ Update aboveH after |H| has changed
16:     t ← h + |C[h]| − aboveH
```

constant span. The parallel reduce used to find the smallest index on Line 12 takes $O(b)$ work and $O(\log b)$ span. The remaining algorithm can be completed in constant time. Hence, the insertion algorithm has bounds $O(b)$ expected work and $O(\log b)$ span w.h.p., limited by the sort (Line 2), update of C (Line 5), prefix sum (Line 9), and parallel reduce (Line 12).

4.3. h-set Deletions

Similar to the vertex insertion algorithm in Section 4.2, for vertex deletions in Algorithm 3, we also start by sorting *batch* in descending order and computing what *aboveH* will be (Lines 2–4). We then remove all the vertices in batch from C (Line 5) with the variant of Algorithm 1 described in Section 4.1. We make a sequence *sum* where each number is the number of vertices H will gain by decreasing h. Since h cannot be negative, the greatest h can change to is $\max(0, h-b)$. We take the prefix sum of the sizes of the entries of C from $h-1$ to $\max(0, h-b)$ inclusive, and put the result in *sum* (Lines 6–9). For each index, we can stop decreasing h when *aboveH* plus the number of vertices gained is greater than the decreased h. We create a boolean array M in parallel with size $|sum|+1$ where $M[0]$ is the boolean value of $(aboveH \geq h)$, and each entry at index i from 1 and beyond is $(aboveH + sum[i-1] \geq h - i)$ (Lines 10–11). The index of the first true is the first possible h, meaning we won't have to decrease it anymore. We find the smallest index of a true and store it in *decrease* by using the same way we did in insertions to find *increase* (Lines 12–14). We update *aboveH* so it gains the $sum[decrease-1]$ vertices added to H in the deletion (Lines 15–16). t is then $h + |C[h]| - aboveH$ (Line 17).

The algorithm for deletions is almost identical to the insertion algorithm described in Section

Algorithm 3 h-set Removal

```
 1: procedure ERASE(batch, G)                ▷ Takes in a batch of vertices and a graph G
 2:     Sort batch in descending order based on the degree
 3:     Create and initialize aboveH ← h + |C[h]| − t
 4:     Add to aboveH the number of vertices in batch that have degree greater than the current h
        (with a serial binary search)
 5:     REMOVEFROMC(batch, G)                                    ▷ Remove vertices from C
 6:     Create the array of numbers sum and initialize each value to 0
 7:     for (parallel) each entry F from C[h] to C[max(0, h − b)] do
 8:         Set the corresponding entry in sum to be |F| if F exists
 9:     PREFIXSUM(sum)                                                ▷ In-place prefix sum
10:     Create boolean array M and initialize M[0] ← (aboveH ≥ h)
11:     Initialize M[i] to (aboveH + sum[i − 1] ≥ h − i) for all positive indices i of M
                                                                ▷ Find index of the first true
12:     Create an array of numbers indexM and initialize indexM[i] to be i if M[i] is true, otherwise
        to b + 1 (or any large sentinel value)
13:     Create and initialize decrease ← smallest index in indexM (use a parallel reduce)
14:     h ← h + decrease
                                                                                ▷ Update t
15:     if decrease > 0 then
16:         aboveH ← aboveH + sum[decrease − 1]          ▷ Update aboveH after h has changed
17:     t ← h + |C[h]| − aboveH
```

4.2, albeit with some key changes to the specific numbers stored. The size of each array and the functions used still have the same bounds. Thus, the proof from Section 4.2 also applies to deletions, and the function ERASE processes a batch in $O(b)$ work and $O(\log b)$ span w.h.p.

4.4. Retrieving H and P Partition

We now prove the bounds for retrieving the set H given the h-index and describe a partition within H that will further optimize triangle counting.

The set H is only implicitly stored in the h vertices with largest degree. We iterate through each entry in C that is greater than h, and compute a prefix sum over each of the sizes. Using the prefix sum, we then add all the vertices to H in order. Finally, we also include the t elements from $C[h]$ to account for the remaining vertices. Since there are a total of h different vertices and at most h different entries of C to iterate over, the prefix sum will have worst case complexity of $O(h)$ work and $O(\log h)$ span w.h.p. Adding all the vertices will then only take constant span and $O(h)$ work. Thus, we can retrieve H given the h-index in $O(h)$ expected work and $O(\log n)$ span w.h.p.

We are also able to check in expected constant time if a vertex is in H by comparing its degree to h. If it is larger, then it must be in H. If it is equal to h, we check if the vertex is among the first t elements of $C[h]$. Otherwise it is not in H.

To minimize the changes to H, we define a set P to store all vertices with degree at least twice the h-index. This set, similar to H is also implicitly stored in C and can be retrieved in a similar fashion. The bounds are also be the same. This set is used in place of H for triangle counting in Section 5.

5. Triangle counting

In this section, we describe a data structure that dynamically maintains the number of triangles using the h-set data structure described in Section 4. Each *batch* consists of either edges to be inserted or deleted and is processed in parallel. Note that however, we substitute H with the P partition as described in Subsection 4.4, which is necessary to obtain our bounds. The data structure consists of the following components:

- Count T, representing the number of triangles in the graph.
- A parallel hash table W mapping from two vertices (u,v), to the number of nodes in $\{x|x \in V \setminus H, (x,v) \in E, \text{ and } (x,u) \in E\}$, or the number of wedges with (u,v) as endpoints and the center node outside of H.

Algorithm 5 contains the pseudocode for updating triangle counts given a batch of edge insertions. We will later show that processing a batch of edge deletions is symmetric.

Theorem 5.1. *We can perform batch-parallel dynamic triangle counting in $O(bh)$ expected work amortized and $O(\log b + \log h)$ span with high probability, where b is the size of the batch and h is the h-index of the graph after inserting the edges in insertion and before deleting the edges in deletion.*

Proof. On Line 22, we find the sum of W for all edges in the *batch*, which can be done by a parallel reduce and takes $O(b)$ work and $O(\log b)$ span. Then, since we only check endpoints that are not in H (Line 21–27), it has at most $O(h)$ neighbors, therefore retrieving these edges takes $O(h)$ work and $O(1)$ span for each of these edges. Since there are h edges in *batch* the total work is also $O(bh)$. Finally, we need an extra parallel reduce to sum all the $O(bh)$ values. Hence, the total work is $O(bh)$ with $O(\log(bh))$ span. Lines 14–19 are also similar. For each edge, we iterate through all the elements of h-set, which takes $O(bh)$ work. We again use a parallel reduce to find the sum, which takes $O(\log(bh))$ span. Additionally, Line 24 is executed $O(h)$ times per edge, so it takes $O(bh)$ work and $O(1)$ span w.h.p. Since the set size is at most $O(bh)$, adding the sets into the wedges map W on Line 20 takes $O(bh)$ work and $O(\log(bh))$ span with w.h.p. Algorithm 4 also shares the same complexity. Since the algorithm only iterates through nodes outside of H, each node has at most $O(h^2)$ pairs of neighbors. The P partition of H, which is the subset of vertices in H with degree $\geq 2h$, changes only $O(\frac{1}{h})$ times on average per element in the batch update. This is proven in Eppstein and Spiro's work [10]. Using this result and P instead of H, the total expected work is $O(bh)$ amortized. Note that the span is unaffected as $O(\log h^2) = O(\log h)$. Thus, the span is $O(\log b + \log h)$ with high probability due to the parallel sort.

Therefore in total, the algorithm takes $O(bh)$ expected work amortized and $O(\log b + \log h)$ span with high probability. □

We start by inserting all the edges into our graph and adding them to our h-set data structure. Note that we are storing the batch of edges as a parallel hash table *batch*, which allows quick retrieval and checking if an edge is newly added or not. Then, to compute the updated triangle count, we find the number of triangles created due to the new batch of inserted edges. All newly created triangles can be be divided into three cases, depending on whether the vertices are in H, and if more than two of its edges are in the inserted batch.

The first case we must handle is triangles with two edges that already exist prior to the insertion, such that the node shared by the two preexisting edges is not in H. To update the number of triangles obtained in this case, it is sufficient to iterate through each of the edges (u,v) in our *batch*, and add $W(u,v)$, the number of wedges with u and v as endpoints, to the total count (Line 12), since all of these wedges would be completed by this newly inserted edge.

For the second case, we start by iterating through each of the edges $(u,v) \in batch$ in parallel (Line 13). We check whether each vertex u is in H or not. If it is not in H, we retrieve w such that $w \in N(u) \setminus H$ and $w \neq v$, and check if these three vertices (u,v,w) form a triangle. We then repeat the same process for vertex v (Line 14). Note that this will result in over-counting. For instance, if (u,v,w) forms a triangle and (u,v) is an added edge, then we will encounter w once while processing vertex u, and we will encounter w again while processing vertex v. Thus, we further categorize all triangles into eleven types by whether the edges are newly inserted or preexisting, and the position of the vertices in H. For each type of such triangles, we assign a **duplicate coefficient** D, which represents the number of times that we over-count each type of triangle. Every time we count a triangle, we add $1/D$ to the total count instead of 1. The eleven types of triangles and their associated duplicate coefficients can be found in Figure 1. Finally, we use a parallel reduce to sum the number of newly formed triangles (Lines 21–27).

The third case (Line 14–19) is triangles with at least one newly inserted edge and have nodes inside the h-set that are opposite to at least one of the new edges. We once again iterate through the edges in *batch* in parallel. For each of these edges $(u,v) \in batch$, we iterate through the vertices $w \in H$, and check if (u,v,w) form a triangle. If so, then we increment the total triangle count. This similarly over-counts the total number of triangles, so we again use the categorization described in Figure 1, adding $1/D$ to the count instead of 1.

After obtaining the corrected triangle counts, we must now update the number of newly formed wedges (Lines 20 and 24). We iterate through the neighbors for each vertex u of the edge (u,v) in the *batch*. For each neighbor $w \in N(u)$, we add 1 to $W(v,w)$.

Finally, we need to adjust the number of wedges after updating the h-set as well. There are two cases that we must consider:

- Case 1: Nodes that are promoted into H due to either increase in degree or decrease in the h-index. W must no longer keep track of wedges with these nodes as the center node.
- Case 2: Nodes that are removed from H because of either decrease in degree or increase in h-index. W must add all the wedges with these nodes as the center node.

For the first case, we create a new set consisting of all the elements in H before the batch of edge insertions on Line 6. Then, we iterate through the updated H, checking if they were in H prior to the edge updates. When we find a node that was not previously in H, we subtract 1 from all wedges $W(u,v)$ for all pairs of its neighbors (u,v). (Line 11).

The second case is similar to the first case, but instead of iterating through the updated H, we instead iterate through the vertices in H prior to the update, and check if they are still in H. For each node that is no longer present in H, we add 1 to all wedges $W(u,v)$ for all pairs of its neighbors (u,v) (Line 7).

Maintaining the triangle count under a batch of edge deletions is similar to that for a batch of edge insertions. Our algorithm, DELETEEDGES, strictly follows Algorithm 5, with the following changes. The first difference is the count obtained from Line 22. For a batch of inserted edges, it under-counts the number of triangles, while it over-counts for a batch of deleted edges. Thus,

Algorithm 4 Adjusting the wedges map W

```
 1: procedure ADJUSTWEDGES(beforeH,H)                    ▷ Adjust the wedges table W
 2:     A ← ∅                                   ▷ A stores the wedges that need to be removed
 3:     B ← ∅                                      ▷ B stores the wedges that need to be added
 4:     for (parallel) c ∈ beforeH do
 5:         if c ∉ H then
 6:             for (parallel) u,v ∈ N(c) do
 7:                 Add (u,v) to B
 8:     for (parallel) c ∈ H do
 9:         if c ∉ beforeH then
10:             for (parallel) u,v ∈ N(c) do
11:                 Add (u,v) to A
12:     Subtract A from W
13:     Add B into W ▷ Since W is a parallel hash table that supports atomic operation, this can be
    done in parallel without contention problems.
```

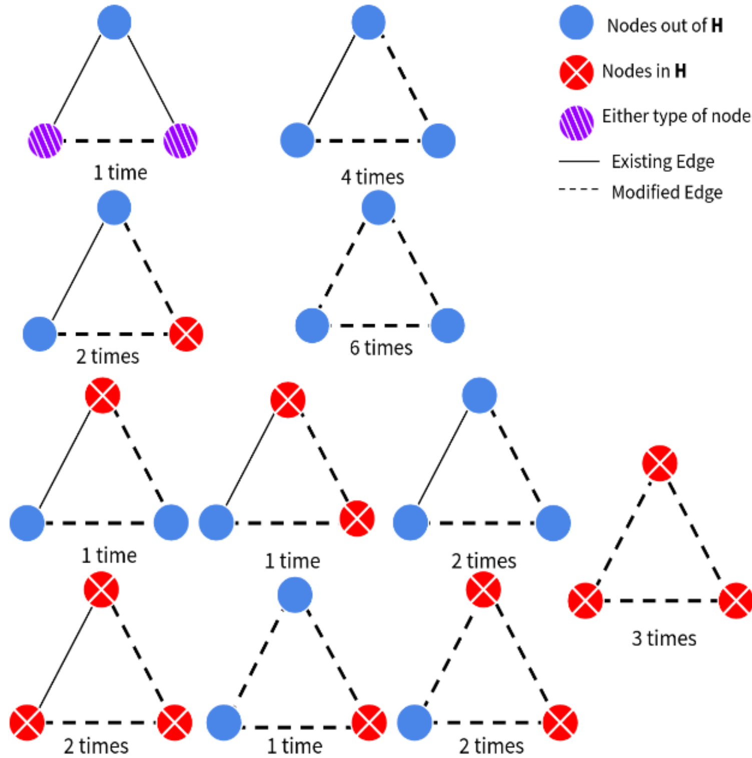

Figure 1. All triangles can be categorized into eleven types, depending on if some of the nodes are in H or not, and if more than one edge is newly added. The times attached to each triangle represent the duplicate coefficient of that triangle, which is the number of times our algorithm would discover each of these triangles.

instead of subtracting, we add the number of triangles obtained from our second (Line 14) and third (Line 1) cases of triangles. Furthermore, instead of deleting the edges from the graph and h-set on Lines 7 and 8, we must perform these operations after counting all the triangles instead, right before Line 20, in order to identify removed triangles.

Algorithm 5 Triangle Counting

```
1: W : (u,v) → Number of wedges with endpoints (u,v)
2: T ← 0                                                    ▷ Number of triangles
3: G ← ∅                                                    ▷ Dynamic symmetric graph
4: Initialize a H to be a batch-parallel dynamic h-set data structure
5: procedure ADDEDGES(batch, G, H)   ▷ Takes in batch of edges, dynamic graph G, and h-set framework H
6:     beforeH ← H.GETELEMENTS() ▷ Retrieving the h-set elements before inserting the new edges
7:     Add batch to G
8:     H.INSERT(batch)                                      ▷ Insert batch into the h-set
9:     ADJUSTWEDGES(beforeH, H)
10:    X ← ∅                                                ▷ The new set of wedges
11:    for (parallel) (u,v) ∈ b do
12:        T ← T + W(u,v)
13:        TRIANGLECOUNTINGHELPER(u,v,w,X,H,T)   ▷ Focusing on node u, counting triangles with more than one added edges.
14:        TRIANGLECOUNTINGHELPER(v,u,w,X,H,T)   ▷ Focusing on node v, counting triangles with more than one added edge.
15:    for (parallel) (u,v) ∈ b do                          ▷ Triangles with nodes in the HSet
16:        for (parallel) w ∈ H do
17:            if (u,w) and (v,w) ∈ E then
18:                D ← duplicate coefficient of the triangle (u,v,w)
19:                T ← T + 1/D
20:    Insert X into W
21: procedure TRIANGLECOUNTINGHELPER(u,v,w,X,H,T)
22:    if u ∉ H then
23:        for (parallel) w ∈ N(u) do
24:            Add (v,w) into X
25:            if (u,v,w) forms a triangle then
26:                D ← duplicate coefficient of the triangle (u,v,w) ▷ The coefficient of the eleven types of triangles can be found in Figure 1
27:                T ← T + 1/D
```

6. Implementation

Our algorithm was implemented and tested using the Graph Based Benchmark Suite (GBBS) [8]. We explored several different ways to store the data structures and various practical optimizations.

In our algorithm for dynamically maintaining the h-set in Section 4, we prove that H can be retrieved in $O(h)$ expected work and $O(\log h)$ span w.h.p. However, depending on how C is stored, the bounds for time and space may vary. The following are two implementations for C:

The first way, which we call **dyn_arr**, stores C as a dynamic array of dynamic arrays. Each index represents a degree and maps to the dynamic array of vertices with that degree. There is minimal overhead when accessing elements, but the worst case time complexity would be reduced to $O(n)$ expected work and $O(\log n)$ span w.h.p. because it will have to iterate through many empty arrays without vertices. In addition, its memory footprint is proportional to the largest degree which is bounded by $O(n)$.

The second option, which we call **threshold**, partitions the vertices given a threshold based on their degrees and stores them in two data structures. If a vertex has degree less than or equal to the threshold, it is stored in the dynamic array of dynamic arrays like in the first implementation. Else, the vertex is stored in a parallel hash table of dynamic arrays. While low degree vertices tend to be more clustered than sparse high degrees, a poorly chosen threshold could reduce the efficiency, and the threshold version does not support changes to the threshold after it is set. This version achieves the same bounds ($O(h)$ expected work and $O(\log h)$ span w.h.p.) as the theoretical algorithm but may be slower in practice due to overhead with hashing. The memory is also proportional to the different number of degrees in the graph.

We also added in an option to disable P partition for the h-set. Although the partition is necessary for the theoretical bounds, we found that in real-life graphs, very few vertices actually have a high enough degree to enter the partition.

Finally, although in the theoretical Algorithm 4, we used parallel hash tables to count elements from a set in parallel, in our actual code, we instead used parallel sort to perform this task. The algorithm first sorts/aggregate the list of elements. Since all the elements are next to each other, their count is (index of endpoint) - (index of start point) + 1, for each start point (which we identify when an element differs from the element before it), we subtract its indices in the array from the counter. We then add back the indices of the end points plus 1 to the counter, giving us the accurate count for each type of element. Since finding the start and endpoints are independent of each other, we can perform this in parallel, resulting in $O(S)$ expected work and $O(\log S)$ span with high probability, where S is the size of the set.

7. Experiments

We conducted our experiments on two graphs, DBLP and Youtube, which are graphs from the Stanford Network Analysis Project (SNAP [12]). The graph statistics are shown in Table 1. We ran our algorithm on a GCP cloud computing instance that has 30 cores with two-way hyperthreading and 240 GB of memory. The processors are 2.1GHz Intel Xeon Scalable Processors (Cascade Lake). We start each algorithm with the initial SNAP graph, and then generate batches of edges for insertion and deletion using the Barabási-Albert algorithm [2]. The algorithm takes in two values, the number of nodes to be modified and the number of edges per node to be inserted or deleted. For our experiments, we select 20,000 vertices uniformly at random, and insert or delete 5 edges for each of the selected vertices. Thus, each batch consists of 100,000 edges. If an already existing edge

Graph name	Number of vertices	Number of edges	Total triangle count	h-index
DBLP	317080	1049866	2224385	137
Youtube	1134890	2987624	3056386	547

Table 1. Statistics on the two graphs we tested on, DBLP and Youtube.

is generated, we remove that edge from the batch. However, since the two graphs we experiment on are sparse, this situation rarely occurs. Additionally, since the Barabási-Albert Model is generative, for batches of deletion, we had to instead use a not so true-to-life selection by randomly picking the edges from the graph.

Each batch of generated edges contains about 100,000 edges. Furthermore, the first half of each set of batches are edge insertions, and the second half are edge deletions. The memory resident set size (RSS) is taken right after all the batch insertions. Table 2 shows our running times for the DBLP graph, and Table 3 shows our running times for the Youtube graph.

The total batch time positively correlates with the batch size, and the time needed per batch also increases as the h-index increases. In addition, the experimental data does not indicate a significant difference in the running time between the threshold implementation and dynamic array implementation of h-set, as further demonstrated in Figures 2 and 4. This is because the max-degrees in both graphs are relatively small. However, the algorithm does display a tendency to be slower with P partition than without it. We believe this is because both DBLP and Youtube do not have vertices with high degree, therefore very few vertices actually have a high enough degree to become part of the P partition.

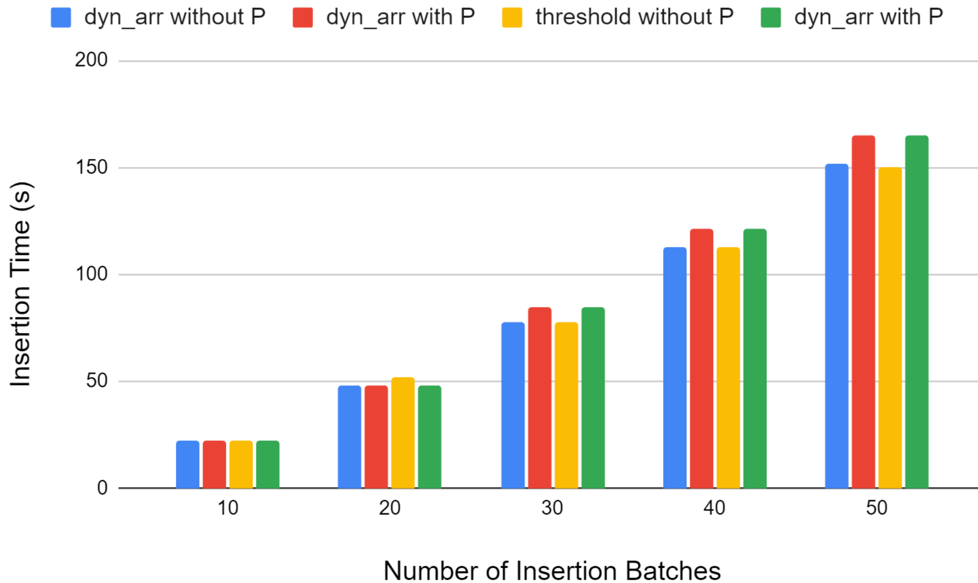

Figure 2. These are the running times of our batch-parallel dynamic triangle counting algorithm under batches of edge insertions on the DBLP graph, showing the relationship between batch size and running time.

Number of Batches	Type of h-set	P Option	Static Graph Insertion Time (s)	Batch Insertion Time (s)	Batch Deletion Time (s)	Total Time (s)	Maximum Triangle Count	Memory RSS (MB)
20	dyn_arr	off	56.03	22.34	11.07	89.44	2,262,415	60,701
		on	44.29	24.23	11.54	80.06	2,262,357	61,279
	threshold	off	47.60	22.40	4.86	74.88	2,260,651	60,708
		on	54.88	24.39	4.55	83.83	2,259,658	61,323
40	dyn_arr	off	53.70	47.85	15.71	117.26	2,295,102	61,923
		on	55.23	52.02	16.73	123.98	2,298,303	62,690
	threshold	off	53.81	48.22	9.48	111.52	2,299,116	61,934
		on	53.58	51.47	9.36	114.41	2,297,295	62,651
60	dyn_arr	off	54.15	78.04	20.02	152.21	2,332,594	79,742
		on	54.87	84.62	22.65	162.14	2,334,947	69,357
	threshold	off	54.00	77.85	15.19	147.04	2,336,530	68,090
		on	55.83	83.54	16.99	156.37	2,338,294	56,749
80	dyn_arr	off	53.58	112.83	25.79	192.20	2,373,476	91,247
		on	53.50	121.14	30.92	205.56	2,373,479	94,276
	threshold	off	50.94	112.77	22.51	186.21	2,372,656	91,312
		on	53.22	121.98	26.71	201.91	2,373,153	94,307
100	dyn_arr	off	53.06	151.69	33.41	238.16	2,415,063	104,197
		on	55.87	165.01	37.68	258.56	2,414,351	62,096
	threshold	off	52.35	150.55	30.15	233.06	2,412,401	104,142
		on	52.99	164.64	35.06	252.69	2,413,794	85,196

Table 2. The table shows the specific elapsed times and the memory resident set size (RSS) after batch insertions for each type of h-set and the number of batches modified on the DBLP graph

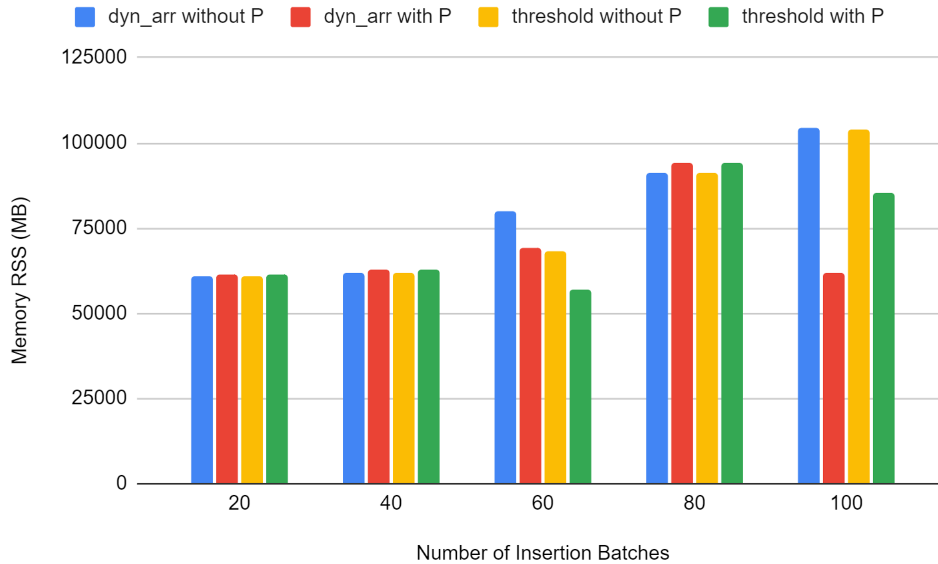

Figure 3. These are the memory RSS of our batch-parallel dynamic triangle counting algorithm under batches of edge insertions on the DBLP graph, showing the relationship between batch size and memory.

Number of Batches	Type of h-set	P Option	Static Graph Insertion Time (s)	Batch Insertion Time (s)	Batch Deletion Time (s)	Total Time (s)	Maximum Triangle Count	Memory RSS (MB)
20	dyn_arr	off	374.36	42.86	55.35	472.58	3,092,211	58,534
		on	422.12	48.62	61.26	532.00	3,092,096	62,031
	threshold	off	373.46	42.54	46.28	462.28	3,091,770	58,527
		on	422.03	48.97	50.51	521.51	3,094,339	62,025
40	dyn_arr	off	374.42	80.71	68.29	523.42	3,133,097	60,306
		on	423.18	93.84	74.79	591.81	3,129,056	64,146
	threshold	off	373.95	81.38	56.88	512.20	3,130,377	60,324
		on	423.46	94.76	62.88	581.10	3,129,724	64,227
60	dyn_arr	off	374.70	116.70	70.54	561.93	3,166,960	62,208
		on	422.33	137.93	78.10	638.35	3,167,823	66,418
	threshold	off	371.44	116.31	59.96	547.72	3,169,741	62,261
		on	422.47	137.04	65.92	625.42	3,164,377	66,422
80	dyn_arr	off	371.94	154.56	71.57	598.07	3,201,131	64,285
		on	425.55	177.23	79.43	682.21	3,203,756	68,686
	threshold	off	373.78	152.96	62.12	588.86	3,204,649	64,311
		on	421.57	178.44	70.26	670.27	3,205,214	68,773
100	dyn_arr	off	376.10	188.51	72.80	637.42	3,241,993	66,448
		on	424.62	220.62	82.95	728.18	3,240,969	71,255
	threshold	off	371.11	188.48	64.75	624.35	3,241,257	66,404
		on	422.27	219.65	75.93	717.84	3,240,507	71,176

Table 3. The table shows the specific elapsed times and the memory resident set size (RSS) after batch insertions for each type of h-set and the number of batches modified on the Youtube graph.

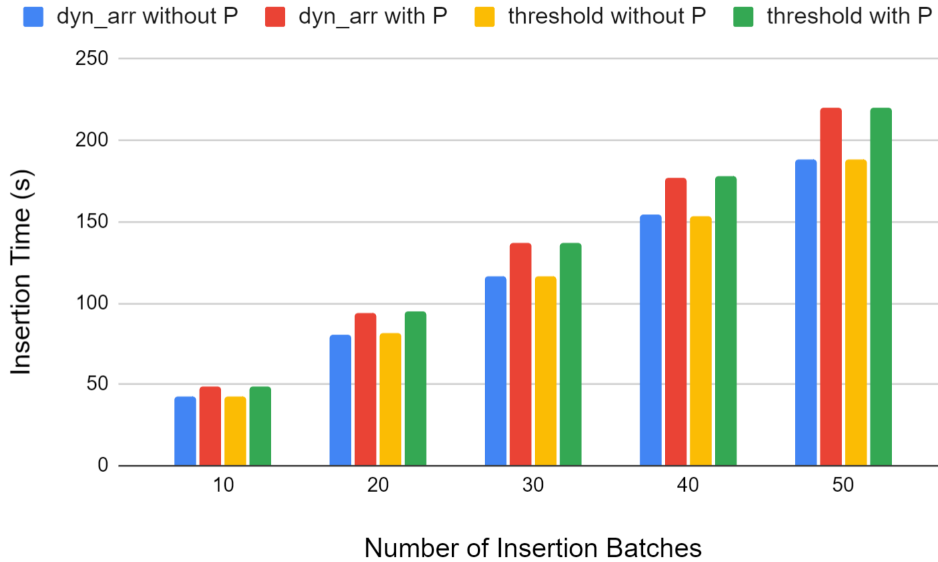

Figure 4. These are the running times of our batch-parallel dynamic triangle counting algorithm under batches of edge insertions on the Youtube graph, showing the relationship between batch size and running time.

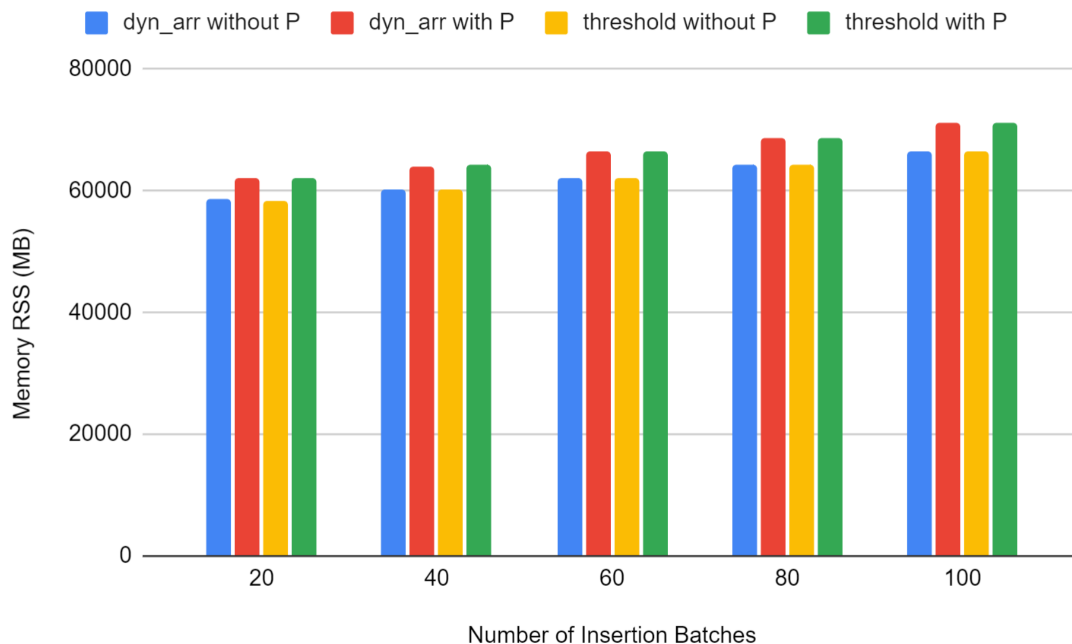

Figure 5. These are the the memory RSS of our batch-parallel dynamic triangle counting algorithm under batches of edge insertions on the Youtube graph, showing the relationship between batch size and memory.

8. Conclusion

We have presented a new work-efficient batch-parallel algorithm for dynamically maintaining the number of triangles in a graph. We showed from experiments that the algorithm follows our theoretical bounds and yields accurate results. An interesting future direction for our work would be to extend it to 4-vertex subgraph counting, as we are already maintaining the number of 2-paths from two vertices (u, v), which can be extended to count 4-vertex subgraphs. Furthermore, memory appears to be an obstacle in testing for larger graphs, so better memory optimizations for storing the count of 2-paths can be researched to apply our algorithm on much larger graphs.

9. Acknowledgement

We would like to first say a very big thank you to our mentor Jessica Shi for all the support and guidance she has provided us over the year. Also special thanks to Julian Shun for suggesting this research topic and pointing us towards the right direction. Finally, we would like to thank the MIT PRIMES program for making this amazing research opportunity possible.

References

[1] N. Ahmed, J. Neville, R. A. Rossi, and N. Duffield. Efficient graphlet counting for large networks. *2015 IEEE International Conference on Data Mining*, pages 1–10, 2015.

[2] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, Jan 2002.

[3] S. Arifuzzaman, M. Khan, and M. Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM International Conference on Information Knowledge Management*, CIKM ’13, page 529–538, New York, NY, USA, 2013. Association for Computing Machinery.

[4] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW '15, page 804–811, USA, 2015. IEEE Computer Society.

[5] G. E. Blelloch and B. M. Maggs. *Parallel Algorithms*, page 25. Chapman Hall/CRC, 2 edition, 2010.

[6] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, Apr. 1974.

[7] L. Dhulipala, Q. C. Liu, J. Shun, and S. Yu. Parallel batch-dynamic k-clique counting, 2020.

[8] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun. The graph based benchmark suite (gbbs). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA'20, New York, NY, USA, 2020. Association for Computing Machinery.

[9] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis. Distributed estimation of graph 4-profiles. *CoRR*, abs/1510.02215, 2015.

[10] D. Eppstein and E. S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. *CoRR*, abs/0904.3741, 2009.

[11] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 698–710, 1991.

[12] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[13] D. Makkar, D. A. Bader, and O. Green. Exact and parallel triangle counting in dynamic graphs. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 2–12, 2017.

[14] O. Obeya, E. Kahssay, E. Fan, and J. Shun. Theoretically-efficient and practical parallel in-place radix sorting. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 213–224, New York, NY, USA, 2019. Association for Computing Machinery.

[15] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *CoRR*, abs/1103.6073, 2011.

[16] H.-M. Park and C.-W. Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM International Conference on Information Knowledge Management*, CIKM '13, page 539–548, New York, NY, USA, 2013. Association for Computing Machinery.

[17] A. Pinar, C. Seshadhri, and V. Vishal. ESCAPE: efficiently counting all 5-vertex subgraphs. *CoRR*, abs/1610.09411, 2016.

[18] P. Ribeiro, P. Paredes, M. E. P. Silva, D. Aparicio, and F. Silva. A survey on subgraph counting: Concepts, algorithms and applications to network motifs and graphlets, 2019.

[19] S. Sanei-Mehri, A. E. Sariyüce, and S. Tirthapura. Butterfly counting in bipartite networks. *CoRR*, abs/1801.00338, 2018.

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

[21] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160, 2015.

[22] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 607–614, New York, NY, USA, 2011. Association for Computing Machinery.

[23] K. Tangwongsan, A. Pavan, and S. Tirthapura. Parallel triangle counting in massive streaming graphs. In *Proceedings of the 22nd ACM International Conference on Information Knowledge Management*, CIKM '13, page 781–786, New York, NY, USA, 2013. Association for Computing Machinery.

[24] 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.