Parallel Triangles and Squares Count for Multigraphs Using Vertex Covers

Luca Cappelletti[1(✉)], Tommaso Fontana[1], Oded Green[3], and David Bader[2]

[1] University of Milan, Milan, Italy
luca.cappelletti1@unimi.it
[2] New Jersey Institute of Technology, Newark, USA
[3] Georgia Tech, Atlanta, GA, USA

Abstract. Triangles and squares count are widely-used graph analytic metrics providing insights into the connectivity of a graph. While the literature has focused on algorithms for global counts in simple graphs, this paper presents parallel algorithms for global and per-node triangle and square counts in large multigraphs. The algorithms have linear improvements in computational complexity as the number of cores increases. The triangle count algorithm has the same complexity as the best-known algorithm in the literature. The squares count algorithm has a lower execution time than previous methods. The proposed algorithms are evaluated on six real-world graphs and multigraphs, including protein-protein interaction graphs, knowledge graphs and large web graphs.

Keywords: Graph · Multigraph · Triangles · Squares · Count

1 Introduction

The study of complex networks and their properties has been an active area of research in recent years. One of a network's most fundamental and well-studied properties is its clustering coefficient [13], which measures the fraction of triangles in a network, where a triangle is defined as three nodes that are all connected. The computation of the clustering coefficient [7] is a crucial step in many graph analytics tasks, including community detection and link prediction.

The original vertex-cover-based algorithms for counting triangles and squares, as described in [4], used vertex covers to reduce the number of set intersections and avoid unnecessary element comparisons. While these algorithms were shown to be much more efficient than traditional baselines, there are still several areas for improvement.

Self-loops or multiple edges between nodes, i.e., when the graph is a multigraph, are common in real-world and knowledge graphs. Both original algorithms assume that these features were either removed or not present. Other algorithms in the literature that handle multigraphs are approximated and implicitly remove the multi-edges instead of considering them [6,11]. All these algorithms only

J. Mikyška et al. (Eds.): ICCS 2023, LNCS 14076, pp. 635–646, 2023.
https://doi.org/10.1007/978-3-031-36027-5_50

provide the global counts of triangles and squares, respectively, but in many use cases, the per-node count would be more valuable.

This paper presents an updated parallel version of the algorithms presented in [4], addressing the above-mentioned shortcomings. Specifically, our algorithms support graphs containing self-loops and multigraphs and provide the number of triangles and squares per node. The updated algorithms' asymptotic worst-case computational complexities are equal to or lower than the original algorithms in real-world sparse graphs. All algorithms are implemented as part of the GRAPE [1] library, and the experiments are provided as library tutorials.[1]

2 Notation

A graph $G = (V, E)$ is composed of a set of nodes V and a set of edges E. A node $v \in V$ has neighbours $\mathcal{N}(v)$ and has degree $d(v)$ equal to the cardinality of its neighbours, $|\mathcal{N}(v)|$. When we sequentially iterate over a node's neighbors, we assume that they are sorted, as is common in several graph data structures.

In a multigraph, the neighbors of a node $v \in V$, $\mathcal{N}(v)$ may be a multiset, i.e., a set with repeated elements. Given a node $w \in V$ and a multiset $\mathcal{N}(v)$, we denote the multiplicity function $m_{\mathcal{N}(v)}(w) : V \to \mathbb{N}$ of as the number of times a node w appears in the neighbourhood $\mathcal{N}(v)$.

In the per-node version of the algorithms, we use atomic instructions [5]. Atomic instructions are low-level hardware operations guaranteed to complete without affecting other memory operations. They are helpful in multi-threaded and concurrent programming, allowing multiple threads to access and modify shared memory locations without the risk of race conditions and data corruption. An atomic fetch add is a specific type of atomic operation that retrieves the current value stored in a memory location and adds a specified value to it, returning the original value. This operation is used to increment the value of a shared memory location in a thread-safe manner without the risk of two or more threads interfering with each other. In real-world sparse graphs, the risk of multiple write attempts using atomic fetch add is very low, as the graph is sparse, and thus there are fewer interactions between nodes. We will denote atomic fetch-add operations as $\mathrel{+}=_A$.

3 Computation of Vertex Covers

A vertex cover $\hat{V} \subseteq V$ is a subset of vertices in a graph such that each edge has at least one endpoint in the vertex cover. The algorithms use vertex covers to minimize the number of required set intersections. Any vertex cover suffices for the purpose, and there are different heuristics to obtain them. Three heuristics were explored based on different node sorting methods and whether to add one or both nodes of an edge to the vertex cover. Obtaining a vertex cover has a complexity of $O(|E|)$, which is negligible compared to the algorithms' complexity.

[1] https://github.com/AnacletoLAB/grape/tree/main/tutorials.

The paper explores three vertex cover schemas: Natural, Decreasing node degree, and Increasing node degree. The natural schema uses the order of nodes as they are loaded into the graph and adds both the edge source and destination. The Decreasing node degree schema sorts the nodes by decreasing node degree, prioritizing nodes with more edges, and only inserts the source nodes. The Increasing node degree schema sorts the nodes by increasing node degree, prioritizing nodes with fewer edges, and only inserts the source nodes.

4 Counting Triangles

We start by describing the global triangle count (Algorithm 1), which takes as input a graph $G = (V, E)$ and a vertex cover $\hat{V} \subseteq V$.

The counter t is initialized to zero, representing the number of triangles times three. It loops in parallel over all vertices in the cover $v_1 \in \hat{V}$ (Line 2). The key insight is that, by definition, every triangle has at least two nodes in the vertex cover [4]. Requiring the first two nodes to be in cover allows us to reduce the total necessary comparisons in the inner loops. For each vertex v_1, it loops over all of its neighbors in the vertex cover $v_2 \in \mathcal{N}(v_1) \cap \hat{V}$ (Line 3). Since we assume the neighbors are sorted if v_2 is greater than or equal to v_1 (in the case of self-loops), the loop is stopped early (Line 4), and thus halves the computational requirements. For v_2, the algorithm loops over all common neighbors of v_1 and v_2, $v_3 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_2)$, which are the nodes that close the triangle (Line 6). To avoid self-loops, the iteration is skipped if v_3 equals v_1 or v_2, which are excluded from the set. To account for triangles composed by multigraph edges, we compute the multiplicities product of the v_3 node in the neighborhoods of the other two nodes, i.e., $c = m_{\mathcal{N}(v_1)}(v_3) m_{\mathcal{N}(v_2)}(v_3)$ (Line 7). If v_3 is in the cover, the counter t is incremented by c (Line 9) because it will be re-encountered two other times. Conversely, if v_3 is not in the vertex cover $\hat{V}$, the counter t is incremented by $3c$ (Line 11) because the node will not be visited again. The algorithm concludes by returning the number of triangles, i.e., the counter divided by three $t/3$. Since the computation of each outer loop are independent, distributed approaches such as map-reduce are possible.

Time Complexities. The computation of the algorithm can be distributed up to $p = |\hat{V}|$ cores. The two inner loops require $O(d_{cover}^2)$ to iterate over all the in-cover neighbors of v_1, which requires at most d_{cover} to compute. The v_3 loop iterates the intersection of the neighbors of v_1 and v_2, which requires at most d_{cover}. The time complexity of the algorithm is $O(|\hat{V}| d_{cover}^2 / p)$.

Algorithm 1: Triangle counts

Input : $G = (V, E)$, cover $\hat{V} \subseteq V$
Output: Graph-wide triangles t

$t \leftarrow 0$;
for $v_1 \in \hat{V}$ **do in parallel**
    for $v_2 \in \mathcal{N}(v_1) \cap \hat{V}$
        if $v_2 \geq v_1$ **then**
            break;
        for $v_3 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_2) \setminus \{v_1, v_2\}$
            $c = m_{\mathcal{N}(v_1)}(v_3) \cdot m_{\mathcal{N}(v_2)}(v_3)$;
            if $v_3 \in \hat{V}$ **then**
                t+=c;
            else
                t+=$3c$;
return t / 3;

4.1 Per Node Triangle Count

In the per-node count (Algorithm 2) we have a vector of atomic counters $\boldsymbol{t}$, one for each node. The triangle count for v_1 is always incremented by the multiplicity factor c (Line 8). If v_3 is not in the cover $\hat{V}$, the triangle count for v_2 and v_3 is also incremented by c. Using atomic additions ensures that each node's triangle count is updated safely, even with concurrent access from multiple threads. Finally, the algorithm returns the vector $\boldsymbol{t}$ of triangle counts per node.

The time complexity of the per-node algorithm remains $O(|\hat{V}| d^2_{cover}/p)$. However, to achieve perfect parallelization using atomic instructions, the processes should simultaneously modify the same counters as little as possible. This is possible in sparse real-world graphs. Still, the algorithm will behave worse than sequentially in degenerate cases, such as cliques, as simultaneous modification will result in the eviction of cache lines and CPU stalls, adding time overhead.

Algorithm 2: Per node count

Input : $G = (V, E)$, cover $\hat{V} \subseteq V$
Output: Vector of triangles $\boldsymbol{t}$ per node

1 $\boldsymbol{t} \leftarrow$ vector with $|V|$ atomic zeros;
2 **for** $v_1 \in \hat{V}$ **do in parallel**
3   **for** $v_2 \in \mathcal{N}(v_1) \cap \hat{V}$
4   **if** $v_2 \geq v_1$ **then**
5   **break**;
6   **for** $v_3 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_2) \setminus \{v_1, v_2\}$
7             $c = m_{\mathcal{N}(v_1)}(v_3) \cdot m_{\mathcal{N}(v_2)}(v_3)$;
8             $\boldsymbol{t}[v_1]$+=$_A c$;
9   **if** $v_3 \notin \hat{V}$ **then**
10                 $\boldsymbol{t}[v_2]$+=$_A c$;
11                 $\boldsymbol{t}[v_3]$+=$_A c$;
12 **return** $\boldsymbol{t}$;

5 Counting Squares

We describe the global square count (Algorithm 3), for a graph $G = (V, E)$ and a vertex cover $\hat{V} \subseteq V$.

The algorithm from [4] employed a double iteration on the vertex cover to check all the pairs of nodes in the cover and the intersection of their neighbors. We can speed up the square counts on sparse graphs by skipping the pairs of nodes that would produce empty intersections. In our approach, we iterate once $v_1 \in \hat{V}$ on the vertex cover and on the second-order neighbors of v_1 in the vertex cover, i.e., $v_3 \in \hat{V}_{v_1}$ where $\hat{V}_{v_1} = \bigcup_{v_2 \in \mathcal{N}(v_1)} \mathcal{N}(v_2) \cap \hat{V}$. By definition, we will only iterate on a pair of nodes in the cover with at least one common neighbor. We want to efficiently iterate on the set of *unique* second-order neighbors in the cover $\hat{V}_{v_1}$; to do so, we need to keep track of the visited nodes $\bar{V}$ to avoid counting squares multiple times. In our implementation to represent $\bar{V}$, we used a bitmap with $|V|$ bits for each thread which is cleared at the start of each new root node v_1. The algorithm initializes the counter s to zero, representing the number of squares times two. It loops in parallel over all vertices in the vertex cover $v_1 \in \hat{V}$ (Line 2). For each vertex v_1, it loops over all of its neighbors $v_2 \in \mathcal{N}(v_1)$ (Line 3). If v_2 equals v_1, we skip to the next neighbor to avoid self-loops. For each v_2, we iterate on all its neighbor in the vertex cover $v_3 \in \mathcal{N}(v_2) \cap \hat{V}$. Since we assume the neighbors are sorted if v_3 is greater than v_1, the loop is stopped early (Line 6), which is done to avoid checking twice the same node and roughly halves the time requirements. We have to skip self-loops $v_3 = v_2$, backward edges $v_3 = v_1$, and already visited nodes $v_3 \in \bar{V}$. Then, we add v_3 to the visited nodes $\bar{V}$ (Line 8).

We initialize the multiplicity counters of neighbors of v_1 in cover v_{in}, out of cover v_{out}, and the sums of the squared multiplicities v_{in^2}, v_{out^2}. We iterate over each common neighbour of v_1 and v_3 excluding the nodes v_1, v_3 themselves. We compute the product of multiplicities of v_4 in v_1 and v_2. If the node v_4 is in cover $v_4 \in \hat{V}$, this multiplicity and its square are added to the counters v_{in} and v_{in^2}, conversely, they are added to v_{out} and v_{out^2}.

We add to the s counter the four counters to obtain the number of squares involving v_1, v_3, v_4, v_2 is counted as part of v_4 nodes. Since we will not encounter multiple times the nodes outside of the cover forming squares with v_1 and v_3, we need to account for the squares they form with themselves $v_{out}^2 - v_{out^2}$, which are all pairs of *distinct* nodes, the squares they form with the in cover nodes $2v_{out}v_{in}$, and the squares formed by nodes in cover $(v_{in}^2 - v_{in^2})/2$, which will be encountered twice. The algorithm concludes by returning the number of squares, $s/2$.

Time Complexity. The algorithm's three inner loops require $O(d_{cover}^2 d_{graph})$ because the algorithm will iterate over all the in-cover neighbors of v_1, which requires at most d_{cover} to compute. The v_3 loop has to compute the neighbors of v_2, which takes at most d_{graph}. The v_4 loop computes the intersection of the neighbors of v_1 and v_2, which will require at most d_{cover}. Therefore, the time complexity of the algorithm is $O(|\hat{V}| d_{cover}^2 d_{graph}/p)$, for $p \leq |\hat{V}|$. This analysis ignored the costs relative to the set $\bar{V}$ due to its strict dependency on the implementation details and because it was negligible in our experiments. This analysis ignored the costs relative to the set $\bar{V}$ due to its strict dependency on the implementation details. A sensible choice may be to use a bitmap paired with a vector, the bitmap for fast reading and updating, and the vector to keep track of the words of memory to reset. These require $O(1)$ time for reading and updating it. The time needed to reset it is proportional to the number of elements in it. This would add a multiplicative factor to the complexity, which in the worst case is $O(d_{cover}d_{graph})$. In practice, this operation is bottle-necked by the memory bandwidth of RAM, so even for large bitmaps, the resetting is practically negligible compared to loops.

Algorithm 3: Square counts

Input : $G = (V, E)$, cover $\hat{V} \subseteq V$
Output: Number of squares s

$s \leftarrow 0$;
for $v_1 \in \hat{V}$ **do in parallel**
  $\bar{V} \leftarrow \emptyset$;
  for $v_2 \in \mathcal{N}(v_1) \setminus \{v_1\}$
    for $v_3 \in \mathcal{N}(v_2) \cap \hat{V} \setminus \{v_1, v_2\} \setminus \bar{V}$
      if $v_3 > v_1$ **then**
        break;
      $\bar{V} \leftarrow \{v_3\} \cup \bar{V}$;
      $v_{in}, v_{out}, v_{in^2}, v_{out^2} \leftarrow 0$;
      for $v_4 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_3) \setminus \{v_1, v_3\}$
        $c = m_{\mathcal{N}(v_1)}(v_4) \cdot m_{\mathcal{N}(v_3)}(v_4)$;
        if $v_4 \in \hat{V}$ **then**
          $v_{in} \mathrel{+}= c$;
          $v_{in^2} \mathrel{+}= c^2$;
        else
          $v_{out} \mathrel{+}= c$;
          $v_{out^2} \mathrel{+}= c^2$;
      $s \mathrel{+}= v_{out}^2 - v_{out^2} + {}^{(v_{in}^2 - v_{in^2})}/_2 + 2v_{out}v_{in}$;
return ${}^{s}/_{2}$;

5.1 Per Node Version

We have a vector of atomic counters $\boldsymbol{s}$, one for each node. Since the number of squares contributed by v_1, v_3 and all $v_4 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_3)$ is obtained from the factor of multiplicities of each $v_4 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_3)$, we need first to compute the counters of the nodes in cover v_{in} and the nodes out of cover v_{out}), and afterward dispense the number of squares among the nodes properly. The necessity to iterate twice on the neighbors $v_4 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_3)$ effectively duplicates the time requirements of the per-node algorithm. The counts of the node v_1 and v_3, which are the root vertex cover nodes, are incremented by the number of squares they form with the in-vertex and out-of-vertex nodes, $v_{out} \cdot v_{in}$. Each node $v_4 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_3)$ count is incremented depending on the multiplicity factor c and whether it is in cover or not. Nodes in the cover will be re-encountered, while nodes outside will be only encountered once alongside the root nodes v_1 and v_3. We double the number of squares deriving from other out-of-cover nodes to account for the latter nodes encountered once. Since in the number of out-of-cover nodes v_{out}, we also count the node's multiplicity factor c, we must subtract that twice. We observe that by summing the obtained square, the total will be four times the total number of squares obtained from the global algorithm. Analogously to the global version, the per-node algorithm is distributable. The time complexity of the per-node algorithm remains $O({}^{|\hat{V}| d_{cover}^2 d_{graph}}/_{p})$.

Algorithm 4: Per node count

```
Input  : G = (V, E), cover V̂ ⊆ V
Output: Number of squares s per node
s ← vector with |V| atomic zeros;
for v1 ∈ V̂ do in parallel
    V̄ ← ∅;
    for v2 ∈ N(v1) \ {v1}
        for v3 ∈ N(v2) ∩ V̂ \ {v1, v2} \ V̄
            if v3 > v1 then
                break;
            V̄ ← {v3} ∪ V̄;
            v_in, v_out ← 0;
            for v4 ∈ N(v1) ∩ N(v3) \ {v1, v3}
                c=m_N(v1)(v4) · m_N(v3)(v4);
                if v4 ∈ V̂ then
                    v_in+=c;
                else
                    v_out+=c;
            s[v1]+=_A v_out v_in;
            s[v3]+=_A v_out v_in;
            for v4 ∈ N(v1) ∩ N(v3) \ {v1, v3}
                c=m_N(v1)(v4) m_N(v3)(v4);
                if v4 ∈ V̂ then
                    s[v4]+=_A c(v_out + v_in − c);
                else
                    s[v4]+=_A c(2(v_out − c) + v_in);
return s;
```

6 Experiments

Experiments were conducted on a computer with an *AMD Ryzen 9 3900x CPU* (12 cores, 24 threads) and 128 GB RAM using six real-world graphs, including protein-protein interaction graphs, knowledge graphs, and web graphs. Table 1 summarizes the datasets, including the graph ID used in all result tables.

Table 1. Summary of the datasets' main characteristics

Graph id	Graph name	Nodes	Edges	d_{graph}
1	Saccharomyces Cerevisiae [12]	$7K$	$1M$	$2.7K$
2	Homo Sapiens [12]	$20K$	$6M$	$7.5K$
3	Mus Musculus [12]	$22K$	$7M$	$7.6K$
4	KGCOVID19 [9]	$570K$	$18M$	$122K$
5	Friendster [10]	$65M$	$1.8G$	$5K$
6	ClueWeb09 [2,10]	$1.6G$	$7.8G$	$6.4M$

6.1 Impact of Vertex Cover Schema

In this section, we present the results of our evaluation of the performance of the triangle and square counting algorithms for various vertex covers. Table 2 provides information on the vertex cover size and time requirements of six different graphs using the three vertex cover schemas described in Sect. 3: natural, decreasing, and increasing. The size of the vertex cover for each graph using each schema is given in the $|\hat{V}|$ column and the maximum degree of each vertex in the graph is given in the d_{cover} column, the percentage of vertices covered by the vertex cover is given in the % column. Finally, the time it took to compute the vertex cover using the given schema is in the Time column. The table indicates that the vertex cover size can vary depending on the schema. The decreasing schema typically produces the smallest vertex cover, and the increasing schema produces the largest. The time it takes to compute the vertex cover also varies depending on the schema used, with the decreasing schema typically being the slowest and the increasing schema typically being the fastest, beating even the **natural** approach, which does not involve any sorting procedures, contrarily to the other two schemas. The table also shows that as the size of the graph increases, the time it takes to compute the vertex cover generally increases as well. Nevertheless, it remains a fraction of the time necessary to compute the same graph's triangles or squares counts.

Table 2. Vertex cover size by vertex cover schema

	Natural				Decreasing				Increasing			
Id	$\lvert\hat{V}\rvert$	d_{cover}	%	*Time*	$\lvert\hat{V}\rvert$	d_{cover}	%	*Time*	$\lvert\hat{V}\rvert$	d_{cover}	%	*Time*
1	6240	2729	93%	77 ms	5720	2729	85%	90 ms	6393	2092	95%	70 ms
2	19200	7507	98%	2 ms	18475	7507	94%	2 ms	19384	6940	99%	1 ms
3	20756	7669	94%	3 ms	19524	7669	88%	3 ms	21300	7296	96%	1 ms
4	217K	122K	38%	12 ms	180K	122K	31%	50 ms	540K	22K	94%	22 ms
5	37M	5214	57%	6 s	31M	5214	48%	15 s	65M	3507	99%	6 s
6	456M	6444K	27%	52 s	277M	6444K	16%	171 s	1672M	2M	99%	106 s

Our experiments revealed that the choice of vertex cover has a significant impact on the performance of the triangle counting algorithm. Table 3 shows the execution time and the number of counted triangles for each vertex cover, both in the global and per-node versions. Notably, the algorithm achieved the fastest performance when using the increasing vertex cover, followed by the natural and decreasing vertex covers. This can be attributed to the fact that the increasing vertex cover, while being the least efficient in terms of the number of nodes covered, effectively excludes the nodes with higher degrees, which can substantially reduce the algorithm's time requirements by a quadratic factor. The choice of vertex cover should therefore be carefully considered when applying our algorithm to real-world graphs, especially those with a high degree of heterogeneity in their node degrees.

Table 3. Triangle counts time by vertex cover

Id	Number of Triangles	Natural		Decreasing		Increasing	
		Global	*Per node*	*Global*	*Per node*	*Global*	*Per node*
1	48834553	231 ms	208 ms	228 ms	207 ms	226 ms	291 ms
2	399408889	2442 ms	2313 ms	2431 ms	2434 ms	2424 ms	2317 ms
3	713495427	3752 ms	3518 ms	3822 ms	3693 ms	3720 ms	3549 ms
4	402950936	3290 ms	3081 ms	3575 ms	3807 ms	2812 ms	2669 ms
5	4173724142	248 s	248 s	250 s	259 s	250 s	244 s
6	31013019123	293 m	301 m	296 m	305 m	43 m	43 m

In Table 4, we present the results of the square counting algorithm using three different vertex covers. The table shows the time taken and the number of squares counted for each strategy. Interestingly, our results suggest that there is no clear optimal vertex cover strategy for this algorithm. This implies that the algorithm's performance is not highly dependent on the choice of vertex cover.

Table 4. Square counts time by vertex cover

Id	Number of Squares	Natural		Decreasing		Increasing	
		Global	*Per node*	*Global*	*Per node*	*Global*	*Per node*
1	17223337716	2 s	6 s	2 s	6 s	2 s	6 s
2	250013165364	40 s	101 s	40 s	99 s	40 s	102
3	659991475347	48 s	126 s	48 s	124 s	49 s	129 s
4	709420799404	104 s	248 s	216 s	516 s	415 s	1058 s
5	465803364346	38.5 h	76 h	37.5 h	35 h	39 h	77 h

6.2 Scalability

To evaluate the scalability of our algorithms, we conducted a series of experiments with varying numbers of threads, including 1, 6, 12 (utilizing all cores), and 24 (using hyper-threading). As shown in Table 5, our algorithms demonstrated linear scaling with the number of cores, confirming their effectiveness in exploiting parallel processing resources. However, we observed some sub-linear scaling when hyper-threading was utilized. Nonetheless, our results demonstrate that our algorithms are highly scalable and capable of achieving significant performance improvements when executed on multi-core systems.

Table 5. Square and triangle count times with natural vertex cover per thread number

	Triangles								Squares							
	Global				*Per node*				*Global*				*Per node*			
Id	1	6	12	24	1	6	12	24	1	6	12	24	1	6	12	24
1	4 s	0.7 s	0.35 s	0.2 s	4 s	0.6 s	0.3 s	0.2 ms	36 s	6 s	3 s	2 s	80 s	16 s	8 s	6 s
2	46 s	8 s	4 s	2 s	42 s	7 s	3.5 s	2 s	12 m	113 s	47 s	40 s	26 m	5 m	147 s	101 s
3	68 s	11 s	6 s	4 s	63 s	10 s	5 s	4 s	14 m	2 m	68 s	48 s	30 m	6 m	3 m	126 s
4	55 s	9 s	5 s	3 s	52 s	9 s	4 s	4 s	27 m	5 m	134 s	104 s	57 m	10 m	5 m	248 s

7 Future Works

This paper presented parallel algorithms for global and per-node triangle and square counts in large multigraphs. While our proposed algorithms have shown improvements in computational complexity, there is still room for future work to optimize further and improve the efficiency of the algorithms.

Firstly, we have identified that the current time complexity of the square count algorithm is $O(|\hat{V}| d_{cover}^2 d_{graph}/p)$, and we have not yet found ways to exploit the vertex cover to reduce the number of checks on 2 of the four vertices of the graph. Future research could explore the design of better algorithms that leverage these two nodes to reduce the computational requirements further.

Secondly, while we focused on developing efficient algorithms for triangle and square counts, we have not explored other algorithms for larger circuits using vertex cover-based acceleration. By searching for efficient algorithms for larger circuits, solutions with lower computational requirements could be discovered that also apply to the count of squares and possibly even triangles.

Thirdly, while our proposed triangle count algorithm can process ClueWeb09 in 40 min, the square count algorithm still cannot process graphs with billions of nodes in reasonable wall times. Future work could investigate the use of GPU-accelerated implementations to close this gap and enable faster execution of the square count on large instances.

In addition to the optimization and improvement of the algorithms, another important avenue for future work is the exploration of the use of these tools in the context of real-world applications, such as graph clustering [8]. While our algorithms provide a fast and efficient way to count triangles and squares and, therefore, to calculate clustering coefficients, we have not yet fully investigated their potential in the analysis of biological graphs such as protein-protein interaction graphs. These graphs are of significant interest in bioinformatics and have important applications in drug discovery and disease diagnosis [3,14]. Future research could explore the application of our proposed algorithms to these types of graphs, and investigate how the resulting triangle and square counts and clustering coefficients could be used to gain insights into the structure and function of large dynamic biological systems. By leveraging the power and efficiency of our algorithms, we believe that our tools could have important implications for the analysis of real-world graphs and the discovery of new insights in a variety of fields.

8 Conclusions

We have presented a set of parallel algorithms for counting triangles and squares in large multigraphs, which have demonstrated significant improvements in computational complexity compared to the best-known algorithms in the literature. Our algorithms achieve linear scaling with the number of available cores and have been evaluated on a range of real-world graphs and multigraphs, including protein-protein interaction graphs, knowledge graphs, and large web graphs. We have also shown that different vertex covers for square counts, both in the global and per-node versions, show no dominant option. In contrast, the increasing vertex covers heuristic is clearly dominant in the triangle counts. These findings could have important implications for optimising and designing future algorithms for counting triangles and squares in large multigraphs.

While our proposed algorithms have demonstrated significant improvements in computational complexity and efficiency, the limited scalability of the squares count algorithm on large instances highlights the need for future studies in high-performance computing settings. These could include exploring the use of GPUs and computing clusters to optimize the efficiency of the algorithm further and enable the processing of larger graphs in reasonable wall times.

Overall, our work contributes to the growing body of research on graph analytics and provides a valuable tool for researchers and practitioners working in a range of fields. By enabling fast and efficient counting of triangles and squares in large multigraphs, our algorithms have the potential to facilitate new insights and discoveries in areas such as bioinformatics, social network analysis, and web mining, among others. We hope that our work will inspire further research in this area and lead to new developments in the field of graph analytics.

References

1. Cappelletti, L., et al.: GRAPE: fast and scalable Graph Processing and Embedding (2021). arXiv: 2110.06196 [cs.LG]
2. Clarke, C.L., Craswell, N., Soboroff, I.: Overview of the trec: web track, p. 2009. Technical report DTIC Document (2009)
3. Friedel, C.C., Zimmer, R.L.: Inferring topology from clustering coefficients in protein-protein interaction networks. BMC Bioinf. **7**(1), 1–15 (2006)
4. Green, O., Bader, D.A.: Faster clustering coefficient using vertex covers. In: 2013 International Conference on Social Computing, pp. 321–330. IEEE (2013)
5. Part Guide: Intel® 64 and IA-32 architectures software developer's manual. In: Volume 3A: System Programming Guide 2.11 (2011)
6. Jha, M., Comandur, S., Pinar, A.: When a graph is not so simple: counting triangles in multigraph streams. Technical report Sandia National Lab. (SNL-CA), Livermore, CA (United States) (2013)
7. Li, Y., Shang, Y., Yang, Y.: Clustering coefficients of large networks. Inf. Sci. **382**, 350–358 (2017)
8. Nascimento, M.C.V., Carvalho, A.C.P.L.F.: A graph clustering algorithm based on a clustering coefficient for weighted graphs. J. Braz. Comput. Soc. **17**, 19–29 (2011)

9. Reese, J.T., et al.: KG-COVID-19: a framework to produce customized knowledge graphs for COVID-19 response. Patterns **2**(1), 100155 (2021). ISSN: 2666–3899. https://doi.org/10.1016/j.patter.2020.100155. https://www.sciencedirect.com/science/article/pii/S2666389920302038
10. Rossi, R.A., Ahmed, N.K.: Networkrepository: a graph data repository with interactive visual analytics. In: 29th AAAI Conference on Artificial Intelligence, Austin, Texas, USA, pp. 25–30 (2015)
11. Stefani, L.D., et al.: Triest: counting local and global triangles in fully dynamic streams with fixed memory size. ACM Trans. Knowl. Disc. Data (TKDD) **11**(4), 1–50 (2017)
12. Szklarczyk, D., et al.: The STRING database in 2021: customizable protein-protein networks, and functional characterization of user-uploaded gene/measurement sets. Nucleic Acids Res. **49**(D1), D605–D612 (2021)
13. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small- world'networks. Nature **393**(6684), 440–442 (1998)
14. Zaki, N., Efimov, D., Berengueres, J.: Protein complex detection using interaction reliability assessment and weighted clustering coefficient. BMC Bioinf. **14**(1), 1–9 (2013)