Accelerating Clique Counting in Sparse Real-World Graphs via Communication-Reducing Optimizations

Amogh Lonkar  
*Computer Science & Engineering*  
University of California, Santa Cruz  
Santa Cruz, CA, USA  
alonkar@ucsc.edu

Scott Beamer  
*Computer Science & Engineering*  
University of California, Santa Cruz  
Santa Cruz, CA, USA  
sbeamer@ucsc.edu

Abstract—Counting instances of specific subgraphs in a larger graph is an important problem in graph mining. Finding cliques of size $k$ ($k$-cliques) is one example of this NP-hard problem. Different algorithms for clique counting avoid counting the same clique multiple times by pivoting or ordering the graph. Ordering-based algorithms include an ordering step to direct the edges in the input graph, and a counting step, which is dominated by building node or edge-induced subgraphs. Of the ordering-based algorithms, kClist is the state-of-the-art algorithm designed to work on sparse real-world graphs. Despite its leading overall performance, kClist’s vertex-parallel implementation does not scale well in practice on graphs with a few million vertices.

We present CITRON (Clique counting with Traffic Reducing Optimizations) to improve the parallel scalability and thus overall performance of clique counting. We accelerate the ordering phase by abandoning kClist’s sequential core ordering and using a parallelized degree ordering. We accelerate the counting phase with our novel subgraph construction approach which reduces memory traffic to improve scaling bottlenecks. Our sorted, compact neighbor lists improve locality and communication efficiency which results in near-linear parallel scaling. CITRON significantly outperforms kClist while counting moderately sized cliques, and thus increases the size of graph practical for clique counting.

We have recently become aware of ArbCount [1], which often outperforms us. However, we believe that the analysis included in this paper will be helpful for anyone who wishes to understand the performance characteristics of $k$-clique counting.

Index Terms—Graph algorithms, $k$-clique counting, graph ordering, performance analysis

I. INTRODUCTION

Clique finding is a well-researched topic in the graph community and it has many interesting practical applications for community detection [2]–[5] and social network analysis [6], [7]. In recent years, the data mining community has incorporated clique finding into deep learning classifiers to enhance recommender systems in social networks [8], [9]. Clique finding is used prominently in bioinformatics, and researchers have used graph models to find variants in gene sequences [10], efficiently group related genes in a database [11] and perform protein structure analysis [12]. The rapid growth in the number of social media users and the large size of genome data has amplified the need for high performance systems capable of analyzing these huge networks to count or enumerate their $k$-cliques.

Clique finding represents only a single problem in Graph Pattern Mining (GPM). Motifs are other general graph patterns that also interest researchers in this space. Various GPM frameworks have been developed to count repeating instances of given patterns within the search graph. These include Pangolin [13], Peregrine [14], Arabesque [15], Fractal [16], and others. The aforementioned frameworks use more generalized algorithms and provide their own APIs for counting instances of arbitrary, user-given patterns. In contrast, there are also algorithms specialized for counting of $k$-cliques, such as kClist [17], Pivoter [18], and Arb-Count [1]. By virtue of being custom designed to solve a specific problem, these algorithms generally perform better than the general-purpose GPM frameworks.

KClust [17] is the current state-of-the-art clique counting algorithm, as it is algorithmically efficient and significantly outperforms prior work. However, when applied to large graphs, it demonstrates poor parallel scalability, especially for modest clique sizes. We carefully analyze kClist and identify scaling bottlenecks in both of its main phases: ordering & counting. The ordering phase directionalizes the graph in order to avoid redundantly counting the same cliques multiple times in the subsequent counting phase. KClust uses a core ordering [19] to reduce the maximum degree in order to obtain the greatest reduction in the amount of algorithmic work in the counting phase. Since the ordering phase consumes a substantial portion of the execution time, and computing a core ordering is inherently sequential, it limits scalability due to Amdahl’s Law. The parallel speedup of kClist’s counting phase quickly saturates, and we observe speedups as poor as $5 \times$ on 24 cores for certain graphs. We find the counting phase is generally memory-bound due to poor locality and the sheer volume of memory traffic.

We present CITRON (Clique Counting with Traffic Reducing Optimizations), a high performance clique counting algorithm. Due to its better parallel scalability, it significantly outperforms kClist, especially for modest clique sizes. We find the use of a core ordering unnecessary, and a parallelizable ordering such as a degree ordering not only greatly accelerates the ordering phase, but it also results in negligible slowdowns in the counting phase in practice. We accelerate the counting phase with our novel subgraph construction approach which
materially reduces the amount of memory accesses and improves their spatial locality.

We evaluate CITRON on a suite of real-world graphs and compare it to kClist and other prior works. For counting triangles, we achieve speedups of $18.16 - 208.27 \times$ (Geometric Mean: $102.436 \times$) in the ordering phase and speedups of $1.1 - 2.6 \times$ (Geometric Mean: $1.798 \times$) in the counting phase, resulting in total speedups of $14.37 - 39.37 \times$ (Geometric Mean: $18.517 \times$) over kClist.

We have recently become aware of ArbCount [1], which outperforms us as seen in Table IV. However, we believe that the analysis included in this paper will be helpful for anyone who wishes to understand the performance characteristics of k-clique counting.

II. BACKGROUND

For a given undirected input graph $G$, we want to count the number of cliques present of a given size $k$. A clique is a completely connected subgraph, i.e., each vertex is connected to every other vertex in the subgraph. The input graph consists of a vertex set, $V(G)$, and an edge set, $E(G)$. For a $k$-clique $C$ present in the input graph, $V(C) \subseteq V(G)$, $E(C) \subseteq E(G)$ and $|V(C)| = k$. Each vertex $u$ in $G$ has a neighborhood, which is the set of the vertices with which $u$ shares an edge. The neighborhood of $u$ in $G$ is indicated by $N(u)$ and its size is the degree $d(u) = |N(u)|$. We treat $G$ as an undirected graph, so $\forall v \in N(u)$, then $u \in N(v)$.

To reduce the amount of work done when counting cliques, ordering-based algorithms transform $G$ into a directed acyclic graph (DAG), which we denote by $\bar{G}$. This step adds work to direct edges to remove cycles in order to avoid counting the same clique multiple times. Consequently, any vertex $u$ in the new DAG can have two types of neighbors: in-neighbors and out-neighbors. In-neighbors are those vertices in $N(u)$ for which $u$ is the destination vertex of the shared edge. Conversely, $u$’s out-neighbors are those vertices in $N(u)$ where $u$ is the source vertex of the edge. While counting cliques, we only consider the out-neighbors of a vertex $u$ in $\bar{G}$, denoted by $N(u)$.

To reduce the amount of work done when counting cliques, ordering-based algorithms transform $G$ into a directed acyclic graph (DAG), which we denote by $\bar{G}$. This step adds work to direct edges to remove cycles in order to avoid counting the same clique multiple times. Consequently, any vertex $u$ in the new DAG can have two types of neighbors: in-neighbors and out-neighbors. In-neighbors are those vertices in $N(u)$ for which $u$ is the destination vertex of the shared edge. Conversely, $u$’s out-neighbors are those vertices in $N(u)$ where $u$ is the source vertex of the edge. While counting cliques, we only consider the out-neighbors of a vertex $u$ in $\bar{G}$, denoted by $N(u)$. The number of out-neighbors of a vertex $u$ is called its out-degree and is written as $d(u)$.

Given a total ordering $\omega$, directionalizing transforms the graph $G$ to $\bar{G}$ by removing the edge $u \rightarrow v$ from $E(G)$ if $\omega(u) \geq \omega(v)$ and keeping only the edge $v \rightarrow u$ in $E(\bar{G})$. Edges are thus directed from a lower $\omega$ to a higher $\omega$ vertex. kClist uses a core ordering, which guarantees the lowest maximum out-degree of a vertex in $\bar{G}$. While this approach reduces the amount of work done while counting cliques, it requires a fair bit of effort to compute, and cannot be parallelized [19]. Alternatively, a degree ordering uses the degree to compare vertices and uses the identifier as a tie breaker (example in Figure 1). We describe the degree ordering as:

$$\omega : \omega(u) > \omega(v) \text{ if } (d(u) > d(v)) \lor (d(u) = d(v)) \land (u > v)$$

A major step in counting cliques is building a vertex-induced subgraph. The induced subgraph contains the vertices in the neighborhood of the target vertex and any edges between them. The induced subgraph does not include the target vertex itself. The highlighted portion in Figure 1 shows the vertex 0 induced subgraph in the example graph. We denote the subgraph induced by vertex $u$ on $\bar{G}$ to be $\bar{g}_u$ with $V(\bar{g}_u) = N(u)$ and $E(\bar{g}_u) = \{(v_1, v_2) | (v_1, v_2) \in E(\bar{G}) \land v_1 \in N(u) \land v_2 \in N(u)\}$.

III. RELATED WORK

One of the fastest sequential algorithms for clique counting is provided by Chiba and Nishizeki [20]. First, it ranks the vertices in descending order of degree. Next, it searches the induced subgraph of each vertex recursively $k$-2 times where $k$ is the size of the desired clique. Once the recursion is complete, the vertex is removed from the graph to avoid duplicate work. The execution time of this algorithm is $O(km\omega(G)^{-k-2})$, where $m$ is the number of edges in the input graph and $\omega(G)$ is the arboricity, i.e., the minimum number of forests into which the edges can be partitioned. While the algorithm is simple and efficient, the process of removing vertices from the graph makes it sequential and suboptimal for dealing with the massive real-world graphs of today.

In recent years, more work has been done to parallelize clique counting to boost performance. Danisch et al. present kClist [17], the current state-of-the-art parallel algorithm for counting and listing cliques. kClist is able to parallelize the counting step by first converting the input graph into a directed acyclic graph (DAG). Directionalizing the graph ensures kClist counts each clique only once without resorting to sorting the vertices and then removing them one by one sequentially while counting like Chiba and Nishizeki’s algorithm. kClist uses a core ordering to rank vertices for directionalizing the graph to be used in the counting phase. The algorithm for computing the core ordering from Matula and Beck [19] guarantees the smallest maximum out-degree in the graph. This is significant, because the maximum degree determines the execution time of the counting phase and a smaller value means less algorithmic work and a more efficient algorithm.

![Fig. 1. Converting an undirected input graph (left) to directed acyclic graph (right) by a degree-based ordering. Furthermore, the highlighted (red) portion on the right indicates the subgraph induced by vertex 0.](image-url)
Pivoter [18], the leading pivoting-based algorithm for k-clique counting. Their algorithm compresses the recursion tree and stores the canonical form as a Succinct Clique Tree (SCT), eliminating the need for ordering the graph. Lastly, Almasri et al. [24] implement ordering and pivoting-based k-clique counting algorithms on GPUs. Aside from the aforementioned dedicated solvers, various GPM frameworks [13]–[16] are also capable of counting cliques in large graphs. Peregrine [14] is a pattern-aware framework, meaning it can avoid expensive computation by only exploring relevant subgraphs. Peregrine achieves this by analyzing the input pattern (i.e., k-cliques) to develop a search plan and use it to traverse the graph.

Algorithm 1 Core Ordering

1: function OrdCore(G)
2:   \( r \leftarrow 0, n \leftarrow |V(G)| \)  \( \triangleright \) Array to hold rank of each node
3:   \( R \leftarrow [0, 0, ..., 0] \)
4:   while \( n \neq 0 \) do
5:     Sort nodes by degree
6:     \( r \leftarrow r + 1 \)
7:     \( \text{minDegNode} \leftarrow \) first element from sorted list
8:     \( R[\text{minDegNode}] \leftarrow n - r \)
9:     \( V(G) \leftarrow V(G) \setminus \text{minDegNode} \)
10:    Update degrees of all \( v \in N(\text{minDegNode}) \)
11:   \( n \leftarrow n - 1 \)
12: return \( R \)
13: Build Graph \( \bar{G} \) using \( R \)

Once the ranking of each vertex is computed (Algorithm 1), the graph is directionnalized and edges \( u \rightarrow v \) are only included if and only if \( R(u) < R(v) \). Similar to Chiba and Nishizeki’s algorithm, producing the core ordering is also sequential since it requires removing a vertex from the graph in each iteration.

After the DAG is generated, kClist starts counting the cliques (Algorithm 2). It recursively builds subgraphs until a certain depth and then adds up the degrees of vertices in the last recursion layer. Since the DAG is not being modified any further, each vertex can be processed in parallel to reduce execution time.

Algorithm 2 kClist Algorithm for Counting k-cliques

1: \( \text{count} \leftarrow 0 \)
2: \( \bar{G} \leftarrow \text{OrdCore}(G) \)  \( \triangleright \) Directionalize input graph
3: \( \text{Counting}(k, \bar{G}, \text{count}) \)
4: function Counting(l, \( \bar{G} \), count)
5: if \( l = 2 \) then
6:   for all nodes in \( \bar{g} \) do
7:     count \leftarrow count + d(node)
8: else
9:   for all \( v \in V(\bar{G}) \) in parallel do
10:      Build subgraph \( g_{node} \)
11:      Counting(l-1, \( g_{node} \), count)
12: return count

The worst case execution time for kClist is \( O(km c(G)^{k-2}) \), where \( c(G) \) is the core value, i.e. the smallest maximum out-degree generated by ordering the vertices. Since \( c(G) \leq 2a(G) - L \), this is an improvement in the execution time over Chiba and Nishizeki for large values of \( k \) even without considering parallelization.

Finocchi et al. [21] present a scalable algorithm for counting k-cliques using the MapReduce framework. They use a degree ordering to direct the graph. More recently, Shi et al. [1] present Arb-Count, a new work-efficient parallel algorithm with polylogarithmic span. Aside from the core and degree ordering, they also implement the Barenboim-Elkin [22] and Goodrich-Pszona [23] orderings. Jain and Seshadhri present Pivoter [18], the leading pivoting-based algorithm for k-clique counting.

IV. Parallelizing the Ordering Phase

As mentioned earlier, the counting phase requires the undirected input graph to be converted into a DAG, and kClist uses the core ordering (Algorithm 1) from Matula and Beck [19] to directionnalize the graph. It ranks vertices in the graph by their effective degree. The vertices are initially sorted on the basis of their degrees and stored in a min-heap. The top element is popped and assigned the lowest rank (set to the number of vertices currently in the graph). When a vertex is removed from the graph, the degrees of its neighbors are updated and the process repeats until no vertices remain in the graph. Each edge is directed from a lower-ranked vertex to a higher-ranked vertex. This ordering is attractive because it produces the lowest maximum-degree in the graph, allowing the counting phase to theoretically perform closer to its optimal bound. However, the algorithm’s vertex removal and degree update step make it inherently sequential.

We find that using a degree-based ordering both requires less computation than a core ordering and can be parallelized. The degree ordering considers vertices’ original degrees and does not require a sequential process of eliminating vertices one-by-one. To directionnalize the graph based on the degree ordering (Algorithm 3), we compare the degree of the two vertices connected by each edge. The edge is directed from the lower-degree vertex to the higher-degree vertex. If the two degrees are equal, then the edge is directed from the vertex with the lower vertex identifier to that with the higher vertex identifier. While each edge can theoretically be considered in parallel, we find vertex-parallelism is sufficient and it simplifies the code. Since our input graph is symmetrized, we only need to do a single check for the degree and vertex identifier comparison.

In theory, a degree-based ordering is not guaranteed to produce the lowest maximum degree, potentially increasing the work required during the counting phase. However, the time required to calculate a degree-based ordering is significantly smaller than that for computing a core ordering, especially after parallelizing. In practice, we find the degree ordering not only does not significantly slow down the counting phase, it can even accelerate it (Table II).
Algorithm 3 Building a DAG using a Degree-Based Ordering in CITRON

1: function GENORDERING(G)
2:     degrees ← (0, 0, ..., 0)
3:     for all u ∈ V(G) in parallel do
4:         for all v ∈ N(u) do
5:             if (d(u) < d(v)) ∨ (d(u) = d(v) ∧ u < v) then
6:                 degrees[u] ← degrees[u] + 1
7:     offsets ← PrefixSum(degrees)
8:     for all u ∈ V(G) in parallel do
9:         for all v ∈ N(u) do
10:            if (d(u) < d(v)) ∨ (d(u) = d(v) ∧ u < v) then
11:               neighbors[offsets[u]] ← v
12:               offsets[u] ← offsets[u] + 1
13:     return G

V. IMPROVING COUNTING PHASE SCALABILITY

To reduce the effect of confounding factors in our experiments, we first implement the kClist algorithm within the GAP Benchmark Suite reference code (gapbs) [26] and we use it as a baseline. The execution time of our implementation differs only 1.27 – 6.40% from the publicly released kClist code (Mean: 3.72%, Variance: 5.98%). Our re-implementation effort allows us to systematically look at the impact of optimizations in isolation without also having to consider other differences in the rest of the code bases.

A. Scaling Bottlenecks in kClist

Our baseline implementation scales poorly for high thread counts (Figure 6) like the released vertex-parallel implementation of kClist. Danisch et al. suggest load-balancing issues that arise from skewed degree distributions in sparse real-world graphs as a possible explanation for the poor scalability of their vertex-parallel algorithm. Through our analysis, we find load imbalance is at most a minor factor, and other bottlenecks hinder scalability to a much greater extent.

To analyze the impact of load imbalance, we both attempt to improve load balance as well as measure the amount of work done by each thread. We sweep various scheduling parameters such as task granularity (chunk sizes) and scheduler types (static, dynamic, cyclic), and we are not able to fully improve parallel scalability. Lumsdaine et al. show that a cyclic distribution (e.g. static schedule with chunk size of 1 in OpenMP) effectively allocates equal work among threads for triangle counting [27]. We test cyclic scheduling with our implementation, and we observe that parallel scaling is only moderately improved (Figure 2).

We measure the work performed by each thread by the number of inner loop iterations because each iteration consists of a uniform amount of work and the number of iterations while processing each vertex is dependent on the number of neighbors of that vertex. With a cyclic distribution, we are able to achieve uniform work distribution among threads. The normalized standard deviation (the ratio of the standard deviation to the mean) of work measured over 24 threads while counting triangles in Cit-Patents is 0.003.

Shifting our analysis to bottom up, performance counter guided profiling with VTune shows memory accesses to be the bottleneck as the baseline code is 49.52% memory bound while counting triangles on Cit-Patents and the CPU is stalled 47.6% of the clock cycles waiting on DRAM. After our optimizations (to be detailed), according to VTune, we reduce the memory bound to 32.92% and stalls due to DRAM to 30.6% for the same task. Due to the success of our interventions, we can confirm that memory is a significant bottleneck of the original code.

Fig. 2. Parallel scaling for counting triangles in Cit-Patents using cyclic work distribution and a dynamic scheduler in the Baseline implementation.

Through our analysis, we find there are three scaling bottlenecks in our baseline implementation that closely mimics kClist. First, the DAG is built in such a way that each directed edge is included twice. This unnecessarily increases the core value, and consequently, the sizes of all the data structures associated with the graph and the subgraph. Scalability is inhibited by allocating a large subgraph structure per thread, and memory accesses quickly become a bottleneck. Second, the original implementation includes a first-level remapping of vertex identifiers changing from the interval [0, |V(G)|] to [0, c(G) − 1] (c(G) is the core value of the graph). We presume this remapping is done to allow the vertex identifier to function as an indexing mechanism to aid in building subgraphs, as well as improving locality. We find that the remapping step increases memory traffic and is not required to accurately count cliques. Lastly, the reference implementation updates a label vector once a neighbor is “visited” so that edges can be included in the subgraph for the following recursion level. This adds an extra memory access while iterating over the neighbor lists. Instead, we find that performing a set
intersection of the sorted neighbor arrays is far more efficient. We find that the aforementioned implementation details place a heavy burden on memory by requiring more accesses with poor locality. By fixing these issues in the next subsection, we are able to significantly improve scalability and thus execution time.

B. Communication Reducing Optimizations

We introduce a number of optimizations to reduce the aforementioned scaling bottlenecks. First, we build our DAG without including duplicate edges and then build compressed adjacency lists for the subgraphs in the subsequent recursion levels. We remove the relabeling step while building the first-level subgraphs as well as the label-based intersection from kClist, and replace it with a set-based intersection on sorted neighbor lists. Finally, we implement certain heuristics to further reduce the amount of work done in the counting phase.

a) Compressing the Adjacency List: As opposed to kClist (which contains two copies of each edge), our generated DAG contains sorted adjacency lists without duplicate neighbors. This becomes beneficial while building the subgraphs during recursion, since we can simply append each new neighbor to the end of the array. kClist pessimistically reserves $c(G)$ spaces per node in the subgraph and then starts placing the neighbors of the $i^{th}$ neighbor at index $i \times c(G)$, making the adjacency list large and sparsely filled since all entries between indices $i \times c(G)$ and $(i+1) \times c(G)$ are unused (Figure 4). This inevitably leads to poor locality and increased cache misses. We can see the stark difference in how much memory is used for storing the first level subgraph by kClist and CITRON in Table III. We describe both subgraph building methods in more detail in Section 5.2.2.

b) Reducing Memory Traffic by Removing Label Array: The main step of the subgraph building process is finding which edges will exist in the subgraph by intersecting each vertices’ neighbor list in the original graph with $V(g)$. kClist accomplishes this by having a single label vector with an entry per node (Algorithm 4). The algorithm then loops over each node’s neighbor list in the subgraph twice. Initially, there is a check to see if the label of the neighbor is equal to the current node’s neighborhood in the original adjacency list. This further reduces the amount of work done in the counting phase.

Step 11 (Algorithm 4) is an optimization Danisch et al. [17] mention explicitly in their paper. All the out-neighbors of a node in the next level subgraph are moved to the start of that node’s neighborhood in the original adjacency list. This further increases the number of data writes required.

One of our major contributions in CITRON is eliminating extra memory accesses for the remapping and the label-based intersection while building subgraphs (Algorithm 5). By having a separate adjacency list and offset array for each subgraph layer, our data is more tightly packed in contrast to the spread out, sparse structure for kClist. Instead of reading and writing from a label array (with poor locality) to check for common neighbors, we perform set intersections between $V(G)$ and $N(u)$.

For the example graph from Figure 1 is 3. The adjacency list for kClist contains $c(G)$ entries reserved for each potential neighbor, i.e. $c(G) \times c(G)$ entries in total. The value of $c(G)$ for the example graph from Figure 1 is 3.
Algorithm 5 Optimized Subgraph Building in CITRON

1: function GENSUBGRAPH($\vec{G}$, $l$, $u$)
2:     $V(g_v) = N^1(u)$
3:     for all node $\in V(g_v)$ do
4:         for all neighbor $\in N(node)$ do
5:             outEdges $\leftarrow V(g_v) \cap N(neighbor)$
6:             $\text{adjList.append(outEdges)}$
7:             $d(node) \leftarrow d(node) + |outEdges|$
8:         offsets $\leftarrow \text{PartialSum}(d)$ $\triangleright$ for CSR
9:     return $g_u$

Fig. 4. First level subgraph for example graph (Figure 1) in the baseline implementation. There is also a separate array for storing the degrees of each vertex in the subgraph not pictured.

For simplicity, we assume that the neighbor lists in the DAG are sorted and there are no duplicate entries. After the initial remapping phase, the new vertex identifiers are $\{0, 1, 2\}$.

For high-degree vertices, we can see how kClist’s organization can be wasteful and require multiple accesses to fit the entire adjacency list into the cache, especially if there are more entries than a cache line can fit. kClist only does the remapping in the top-most recursion level and then swaps nodes to the front of that vertex’s allocated space ($i \times c(G)$).

In contrast, our approach skips the remapping and simply appends each vertex’s common neighbors to the end of the array. To get the required neighbor list for the intersection, we can simply slice the adjacency list from indices $\text{Offset}[\text{nodeIndex}]$ to $\text{Offset}[\text{nodeIndex}+1]$, where $\text{nodeIndex}$ is the index of the vertex in $V(G)$. Our compact adjacency list is similar to the Doubly Compressed Sparse Row (DCSR) format in CombBLAS [28], [29]. Additionally, a simple building process allows us to repeat it per level instead of swapping vertices and still enjoy a performance boost.

Our optimizations in CITRON greatly reduces the memory access and size requirements. Table III compares the number of array accesses required to build all of the first level subgraphs and the total memory (in bytes) used by the largest subgraph in each kClist and CITRON. The kClist subgraph is of a constant size for each vertex since the array lengths are determined by the core value. In contrast, CITRON has a variable subgraph size for each vertex since the adjacency list is built by appending neighbors to the end of a list. To
CITRON. W

BUILDING ALL THE FIRST LEVEL SUBGRAPHS AND THE MEMORY

COMPARISON BETWEEN THE NUMBER OF ARRAY ACCESSES WHILE
BUILDING ALL THE FIRST LEVEL SUBGRAPHS AND THE MEMORY

CONSUMED FOR THE BASELINE AND CITRON. WE MEASURE SIZES WITH

maintain fairness, we report the size of the largest subgraph
(induced by the vertex with the highest degree).

From Table III, we observe that CITRON requires fewer
array accesses for all of the input graphs. We can also see that
the difference in the memory sizes is quite significant, and
removing the relabeling step and the label array (used for the
intersections) allows us to save additional space in memory.
Although the original input graph is far larger than the space
reclaimed with our smaller subgraph data structures, its benefit
is that it reduces the working set of memory accesses, allowing
for more to fit in cache. Additionally, since each thread gets
its own private copy of the subgraph struct, the memory
required for storing the kClist subgraph can quickly become
a bottleneck for high thread counts.

c) Other Optimizations: We add a heuristic to skip
processing nodes if cliques are impossible to be found. If the
number of nodes in the subgraph is less than \((l - 2)\) at any
level in the recursion, then that subgraph will never contain a
k-clique.

Finally, we further decrease the amount of work done by
not building an adjacency list and computing the row offsets
in the penultimate recursion layer. This is possible because
we only require degrees of each node in the subgraph in the
current level instead of the nodes and their neighbors in the
next level. This also reduces the amount of memory required
for the last level subgraph.

VI. EXPERIMENTAL SETUP

A. Environment

We perform our experiments on a dual-socket Intel Xeon
Platinum 8260. Each socket has 24 physical cores running
at 2.40GHz with two-way hyperthreading (48 threads) and
35.75MB shared L3 cache. Our system has 768 GB RAM.
For the experiments, we use all thread contexts (96) unless
specified otherwise. We compile with g++ (version 9.3.0) with
optimization -O3 and use OpenMP. We also use Intel VTune
for analyzing our optimization’s improvements.

B. Graphs in this Work

We use a variety of input graphs to evaluate the performance
of our optimizations (Table I). These graphs are taken from
Stanford Network Analysis Project (SNAP) [25]. Since clique
finding is used heavily in social network analysis, we have
selected commonly used graphs to make analysis more ger-
mane. All graphs are unweighted and symmetrized to initially
be undirected.

VII. Evaluation

In this section, we analyze how the two different orderings
affect performance. We also compare the overall performance
of our optimizations in CITRON with both vertex-parallel and
edge-parallel versions of kClist, as well as a GPM framework
in Peregrine [14], a dedicated solver in Pivoter [18], and
a recently released code ArbCount [1]. All of the thread
counting experiments are performed by increasing the number
of threads (and cores) from 1 to 48. All other comparisons
are performed using all available threads (96). We perform at
least 2 trials and record the mean for each data point.

A. Ordering Comparison

As seen in Table II, a degree-based ordering may not always produce the lowest maximum-degree value. However,
its lightweight computation and parallelizability generally lead
to overall lower total execution times for clique counting. In
Table II, we also break down the ordering and counting times
for counting triangles \((k=3)\) in each graph and see how each
ordering impacts the total execution time. Both orderings are
applied on our optimized code running on 96 threads and the
fastest total time is denoted in bold.

The ordering time and total execution time are greatly
improved with the parallel degree-based ordering. Addition-
ally, we observe that the counting time for the degree-based
ordering is also less in most cases. To understand this, we
make a simple model for algorithmic work and animate it with
data from our executions. For triangles, the total work is given
by the sum of the product of each vertex’s degree with the sum
of each of its neighbor’s degrees as the algorithm traverses
those edges \((\sum_{u \in V(G)} d_u \cdot \sum_{v \in N(u)} d_v))\). The degree-based
ordering produces a smaller value for work than the core value
in most cases. As \(k\) gets larger, core ordering becomes more
beneficial. Increasing \(k\) increases the max recursive depth, and
the overall execution becomes more dominated by the highest
degree nodes.

B. Parallel Scaling of the Counting Phase

To compare the improvement in parallel scaling of our
optimizations with our baseline implementation, we count
3,4,5 and 6-cliques on the input graphs on increasing thread
counts up to 48 threads (Figure 6). We do not see major
improvement using hyperthreading, so we report data using
48 threads on 2 sockets and 1-24 threads on one socket. Each
plot shows the parallel speedup of the counting phase relative
to single thread of that implementation.

In the baseline scaling plots (Figure 6), we observe that
scaling improves as \(k\) increases. We believe the amortization
of scheduling overhead causes this phenomenon since more work
is required to count larger cliques than smaller cliques. We
can see that our optimizations in CITRON reduce this effect.
Especially in Orkut and cit-Patents, all \(k\) values scale equally
well. Most notably, our optimizations perform extremely well compared to kClist on cit-Patents, where we achieve near-linear scaling.

C. Total execution time Comparison

Table IV shows the overall execution times for counting cliques of various sizes on all the input graphs for each implementation. We use the released implementation of kClist rather than our baseline for fairness. We were unable to convert As-Skitter into the required graph format for Peregrine. After repeated effort, we are unable to use kClist’s Edge-Parallel implementation to count triangles on any graph. Peregrine is unable to count cliques of any size on Friendster as it times out.

Our speedups range from $0.39 - 53.04 \times$ (Geometric Mean: $6.19 \times$) over kClist, $1.42 - 498.65 \times$ (Geometric Mean: $10.71 \times$) over Peregrine and $0.21 - 335.12 \times$ (Geometric Mean: $45.08 \times$) over Pivoter. CITRON performs best for modest values of $k$ (Figure 7). The newly released ArbCount outperforms all frameworks most of the time.

We observe that a degree ordering greatly improves the execution time for smaller $k$ values as the overhead of computing the core ordering is very large compared to the counting time. However, as counting time increases with the clique size, this overhead becomes amortized and the core ordering leads to a smaller execution time. This is consistent with Shi et al.’s findings [1].

We suspect that load imbalance likely becomes an issue with our optimizations for higher $k$ values. With a degree-based ordering, the larger maximum degree increases the computational fan-out for building subgraphs in deeper recursions. Our asymptotic analysis also shows that the execution time of the counting phase also grows exponentially for increasing $k$. We parallelize a task size of 1 (i.e. each vertex). Initial analysis shows counting cliques in the induced subgraphs of large degree vertices require a significant fraction of the total execution time. Thus, it may be beneficial to parallelize subgraph building or further recursion levels for large degree vertices.

VIII. CONCLUSION

CITRON optimizes kClist’s vertex-parallel algorithm by optimizing memory communication and increasing parallelization, both proven techniques to improve graph algorithm performance [30]. CITRON reduces communication between the CPU and memory by more efficiently storing and accessing the subgraph structures in each recursion level of the counting phase. We are able to significantly improve parallel scaling and the time required to count cliques using this optimization. We also use a parallel, lightweight degree-based ordering to produce a DAG in the ordering phase. Even though we lose some efficiency in the counting phase by using the degree-based ordering instead of a core ordering, we observe improvements in total execution time because the degree ordering is faster to compute. CITRON achieves speedups of $18.16 - 208.27 \times$ (Geometric Mean: $102.436 \times$) in the ordering phase and speedups of $1.1 - 2.6 \times$ (Geometric Mean: $1.798 \times$) in the counting phase, resulting in total speedups of $14.37 - 39.37 \times$ (Geometric Mean: $18.517 \times$) over kClist while counting triangles.

While we focus on counting in this work, we are easily able to enumerate each clique with a few simple changes to our code. Since more research is in the field of Graph Pattern Mining, we hope that our optimization can be used as a performance benchmark to compare the performance of new clique counting algorithms.
The best execution time is denoted in bold. The best execution time other than Pivoter is italicized. *Both node parallel and edge parallel version of kClist segfaulted for Friendster so we report the values for our baseline implementation which has nearly identical performance to the kClist Node Parallel implementation.

**TABLE IV**

Summary of total execution time for counting cliques using kClist [17], CITRON, Peregrine [14] and Pivoter [18]. Each algorithm was executed using 96 threads on the same machine under the same conditions. The best execution time is denoted in bold. The best execution time other than Arbitrator is italicized. *Both node parallel and edge parallel version of kClist segfaulted for Friendster so we report the values for our baseline implementation which has nearly identical performance to the kClist Node Parallel implementation.

**ACKNOWLEDGMENT**

[1] J. Shi, L. Dhulipala, and J. Shun, “Parallel clique counting and peeling algorithms,” in *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 2021, pp. 135–146.

[2] E. Gregori, L. Lenzini, and S. Mainardi, “Parallel k-clique community detection on large-scale networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1651–1660, 2013.

[3] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, “Uncovering the overlapping community structure of nature and society,” *nature*, vol. 435, no. 7043, pp. 814–818, 2005.

[4] F. Hao, G. Min, Z. Pei, D.-S. Park, and L. T. Yang, “k-clique community detection in social networks based on formal concept analysis,” *IEEE Systems Journal*, vol. 11, no. 1, pp. 250–259, 2017.

[5] Y. Fang, K. Yu, R. Cheng, L. V. Lakshmanan, and X. Lin, “Efficient algorithms for densest subgraph discovery,” arXiv preprint arXiv:1906.00341, 2019.

[6] L. Pan and E. E. Santos, “An anytime-anywhere approach for maximal clique enumeration in social network analysis,” in *2008 IEEE International Conference on Systems, Man and Cybernetics*, 2008, pp. 3529–3535.

[7] R. A. Rossi, D. F. Gleich, and A. H. Gebremedhin, “Parallel maximum clique algorithms with applications to network analysis,” *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. C589–C616, 2015.
