PivotScale: A Holistic Approach for Scalable Clique Counting

Amogh Lonkar Scott Beamer
Computer Science & Engineering
University of California, Santa Cruz
Santa Cruz, CA, USA
{alonkar, sbeamer}@ucsc.edu

***Abstract*—Counting cliques of size k (k-cliques) in a graph is an important problem in graph pattern mining. Due to a combinatorial explosion in the amount of work, counting large cliques in real-world networks is challenging, as leading parallel approaches become untenable for even modestly large clique sizes (e.g. k=10). Pivoter, a recent algorithm, is able to scale to much larger clique sizes due to its superior algorithmic complexity. While it has leading single-thread performance, its naive parallel implementation results in poor parallel speedups. Efforts to optimize its parallel performance on CPUs are absent in the current literature.**

We present PivotScale, a scalable approach to accelerate exact clique counting. Our approach scales with both the number of cores as well as the clique size k. During the initial ordering phase, we introduce a heuristic to select which parallel ordering approach will result in the fastest overall execution time. In the subsequent counting phase, we increase scalability by reducing memory usage. Our high-performance parallel implementation outperforms prior work and demonstrates near-linear parallel scaling for up to 64 threads on large real-world social networks.

***Index Terms*—Graph algorithms, clique counting, parallel scaling, performance analysis**

I. Introduction

Clique finding is a well-studied problem with many interesting practical applications including community detection [1]–[4] and social network analysis [5], [6]. A k-clique is a subgraph of k vertices where each vertex is connected to every other vertex. In recent years, the data mining community has incorporated clique finding into deep learning classifiers to enhance recommender systems in social networks [7], [8]. Clique finding is used prominently in bioinformatics where researchers use graph models to find variants in gene sequences [9], efficiently group related genes in a database [10], and analyze protein structures [11]. The rapid growth of social media and the large size of genomic data have amplified the need for high-performance systems capable of analyzing these massive networks to count or enumerate their k-cliques.

Clique finding represents only a single problem in the Graph Pattern Mining (GPM) domain. There are many GPM frameworks [12]–[16], and they use generalized algorithms to provide APIs to count arbitrary subgraph patterns of interest (motifs). By contrast, algorithms specialized to count k-cliques [17]–[19] generally perform better.

Clique counting is a challenging but rich problem to explore. Searching for cliques involves considering various combinations of vertices which results in a combinatorial explosion of algorithmic work. The prevalence of large cliques in real-world networks (Figure 1) necessitates a combination of algorithmic efficiency and practical performance to process them.

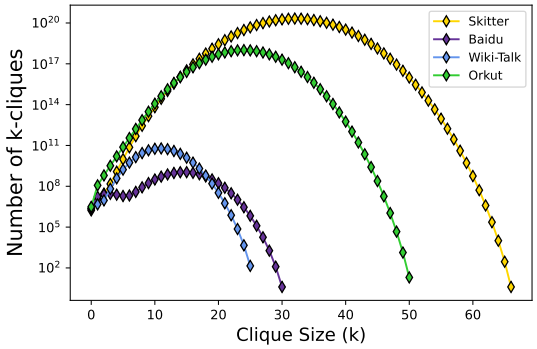

Fig. 1. Frequency distribution of k-cliques in different graphs (Table I). Large real-world graphs frequently have large cliques, but enumeration-based methods scale exponentially with clique size (k) making them unsuitable. Pivoting-based counting methods (this work) are algorithmically efficient.

Leading clique counting algorithms typically contain two main phases: *ordering*, which converts the undirected input graph into a directed acyclic graph (DAG), and *counting*, which is dominated by recursively building induced subgraphs. Different algorithms approach the counting phase differently, and they can be classified as enumeration-based [17], [19] or pivoting-based [18], [20]. Enumeration-based algorithms are generally faster for counting smaller cliques, while pivoting-based algorithms are faster for larger cliques.

Large cliques are common in practice, and surprisingly, larger cliques can even be more common than smaller cliques (Figure 1). This counterintuitive result arises because a clique of size n contains $\binom{n}{k}$ k-cliques. That combinatoric quantity is maximized when $k \approx \frac{n}{2}$. Thus, if there is a large maximal clique, there are a surprising number of still reasonably large cliques contained within it. For example, a 24-clique contains over 2.7 million 12-cliques. This highlights why counting large cliques is challenging. Increasing the target clique size not only deepens the search space, but it also increases the number of cliques to be counted.

As core counts continue to increase in modern machines,

improving parallel performance becomes increasingly essential to process graphs at scale. Given that multiple factors affect clique counting performance, we take a holistic approach to improve its parallel scalability and overall performance on CPUs. We analyze both the ordering and counting phases of the current state-of-the-art pivoting-based clique counting algorithm Pivoter [18] to understand how these factors impact performance and guide the creation of our targeted optimizations. In this work, we present *PivotScale*, a fast and scalable exact clique counting algorithm, and make the following contributions:

- We explore the tradeoffs between ordering quality and counting phase time. We consider a traditional core ordering, a parallel degree ordering, a parallel core ordering approximation, a parallel k-core decomposition based ordering, and a novel parallel centrality ordering. Our parallel core ordering approximation performs well and typically produces the same maximum out degree as the traditional core ordering.
- Depending on the input graph topology, a different ordering will produce the fastest overall time, so we present a runtime heuristic to quickly select the best ordering.
- To increase the parallel scalability of the counting phase, we present a subgraph data structure which greatly reduces memory usage while still allowing for fast access to neighbor lists.
- Combining our heuristic and the optimizations in the ordering and counting phases, we achieve the first scalable pivoting-based clique counting algorithm on CPUs.

We evaluate PivotScale on a suite of real-world graphs and compare it to Pivoter and other prior work. PivotScale achieves near-linear parallel scaling up through 64 threads for the entire clique counting process. We also compare against a GPU implementation of pivoting, GPU-Pivot [20]. PivotScale scales better with increasing clique size (k) which allows it to outperform the GPU when counting larger cliques.

II. Background

A. Preliminaries

For a given undirected input graph G, we want to count the number of cliques within it. A *clique* is a completely connected subgraph, i.e. each vertex is directly connected to every other vertex in the subgraph. A k-clique is a clique of exactly k vertices. The input graph G consists of a vertex set, $V(G)$, and an edge set, $E(G)$. Each vertex u in G has a neighborhood $N(u)$ which is the set of vertices that u is directly connected to. The size of a neighborhood is the degree $d(u) = |N(u)|$.

To reduce the amount of work to count cliques, efficient algorithms transform G into a DAG $\vec{G}$. With the DAG, each clique is only counted once in its canonical form instead of $k!$ times. Given a total ordering ω, *directionalizing* transforms the graph G to $\vec{G}$ by removing the edge $v \to u$ from $E(G)$ if $\omega(v) \geq \omega(u)$ and keeping only the edge $u \to v$ in $E(\vec{G})$. Edges are thus directed from a lower ω to a higher ω vertex. Pivoter uses a core/degeneracy ordering, which guarantees the lowest maximum out-degree (core value) of a vertex in $\vec{G}$. 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 since it requires removing vertices sequentially [21]. Alternatively, a degree ordering compares vertices by degree and uses the identifier as a tiebreaker (example in Figure 2), and computing it is easily parallelized.

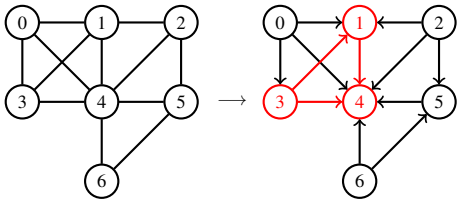

Fig. 2. Converting an undirected input graph (left) to a directed acyclic graph (right) with a degree-based ordering. Furthermore, the highlighted (red) portion on the right indicates the subgraph induced by vertex 0.

A major step in counting cliques is building a vertex-induced subgraph. This subgraph represents the current scope of the graph being explored which can potentially be a clique. The induced subgraph contains the neighbors of the target vertex and any edges between them, but it does not include the target vertex itself. The highlighted portion in Figure 2 is the subgraph induced for vertex 0. We denote the subgraph induced by vertex u on $\vec{G}$ to be $\vec{g_u}$ with $V(\vec{g_u}) = N(\vec{u})$ and $E(\vec{g_u}) = \{(v_1, v_2) \mid (v_1, v_2) \in E(\vec{G}) \wedge v_1 \in N(\vec{u}) \wedge v_2 \in N(\vec{u})\}$.

B. Pivoting-Based K-Clique Counting

Both clique counting approaches (enumeration and pivoting) recursively build subgraphs in the counting phase, but they differ in which subgraphs they build in each recursion level. Enumeration-based methods build a subgraph for every vertex in the subgraph at that level.

In contrast, Pivoter [18], the leading pivoting-based algorithm for k-clique counting, prunes away redundant work in the search space by judiciously selecting which subgraphs to build based on a *pivot* vertex. Pivoter counts maximal cliques using the Bron-Kerbosch algorithm [22]–[24]. Once a maximal clique of size n has been found, the number of k-cliques present in the maximal clique can be calculated by the formula $\binom{n}{k}$. Since Pivoter's worst case execution time does not depend on k, it is more efficient than enumeration-based methods for counting large cliques. Pivoter is able to count instances of 100-cliques and larger in real-world networks. In contrast, leading enumeration-based algorithms typically take an unreasonable amount of time when $k \geq 8$ (Figure 12).

After the DAG is generated by a core ordering, Pivoter starts counting the cliques (Algorithm 1). For each vertex, Pivoter discovers maximal cliques that contain it. Because of directionalization, each maximal clique is discoverable from only one root vertex and thus only counted once. Pivoter maximizes the clique C by recursively adding a vertex w to it and constraining the subgraph to only contain neighbors of w. To reduce the amount of work, the *pivot* (highest-degree

Algorithm 1 Pivoter Algorithm for Counting k-cliques

```
1: function CountCliques(G, k)
2:     G⃗ ← DirectByCoreOrder(G)                    ▷ Directionalize
3:     count ← 0
4:     for all v ∈ V(G⃗) in parallel do
5:         g⃗_v ← InduceSubgraph(G⃗, v)
6:         count += CountRecurse(g⃗_v, {v}, 0)
7:     return count
8: function CountRecurse(g⃗_v, C, np)
9:     if V(g⃗_v) = ∅ ∨ |C| − np = k then
10:        return  C(np, k−|C|+np)
11:    p ← FindPivot(g⃗_v)                          ▷ Pivot
12:    count ← 0
13:    for all w ∈ {V(g⃗_v) \ N(p)} do              ▷ Includes p
14:        g⃗_w ← InduceSubgraph(g⃗_v, w)
15:        if w = p then                            ▷ Processing pivot
16:            count += CountRecurse(g⃗_w, C∪{w}, np+1)
17:        else
18:            count += CountRecurse(g⃗_w, C ∪ {w}, np)
19:    return count
```

vertex in the subgraph), is chosen at each level. Pivoting saves work by not building subgraphs for the neighbors of the pivot since they will be handled by the pivot's subgraph. The recursion branches accordingly to also handle non-neighbors of the pivot. The actual counting (line 10) requires knowing the number of pivots taken so far in the recursion (np). We discuss subtle details of Algorithm 1 in Section V-A.

As an implementation optimization, Pivoter builds a fresh subgraph for the first level, and then mutates that subgraph for subsequent levels since they may only slightly shrink the subgraph. Since the original DAG is left unmodified (only the subgraph is modified), each vertex can be processed in parallel (line 4). Mutating subgraphs instead of building them fresh for each recursive call saves a substantial amount of work, but the challenge is that these subgraph mutations must be reversible. In particular, sets of reversible changes must be able to accumulate and be undone like a stack to match the tree of recursive calls. Naturally, this mutation-intensive code can be difficult to implement correctly, but it is performance critical for this problem. The time spent inducing subgraphs by mutation (lines 5 & 14) and then reversing those mutations (not shown), makes up the bulk of the computational work.

C. Efforts to Parallelize Pivoter

The original Pivoter publication focuses on its sequential implementation, but it also includes a naively parallelized implementation [18]. The authors note that their parallel implementation demonstrates that their algorithm can be parallelized, but it is not optimized for performance. In our experiments, we measure modest parallel speedups ($< 4\times$) on 64 threads in the counting phase, and its overall scalability is further hindered by the use of a sequential algorithm for ordering. Currently, the only optimized parallel implementation of pivoting-based clique counting is GPU-Pivot presented by Almasri et al. [20]. To the best of our knowledge, no literature currently exists for parallelizing Pivoter for CPUs.

Counting cliques on GPUs requires a careful balance between exposing sufficient parallelism to keep a large number of threads busy without significantly increasing memory consumption by storing intermediate data per-thread. In GPU-Pivot [20], a vertex or an edge is assigned to a block of threads (warp). Finding the pivot vertex is the only algorithmic work that is parallelized within a warp. Once the pivot has been found, all threads in the warp sequentially build the same induced subgraph. This avoids increasing memory consumption by not building multiple induced subgraphs in parallel within a warp. To further optimize memory consumption, GPU-Pivot stores the entire adjacency matrix in a binary-encoded format. This requires building a new subgraph per recursive call, resulting in more algorithmic work. Since counting cliques on large dense graphs involves recursively building many induced subgraphs, building a single subgraph per-warp complicates fully utilizing the GPU's parallel capabilities when scaling up the clique size.

In contrast, we leverage the additional memory capacity of CPUs to store a subgraph per thread. While our structure is not as memory efficient as the binary-encoded subgraph, we avoid building a new subgraph per level because we are able to reuse our subgraphs with reversible mutations. This improves scalability on large dense graphs with many cliques.

III. Parallelizing the Ordering Phase

When scaling to larger target clique sizes, limiting the maximum out-degree in the directionalized graph is essential for performance. The counting effort required per vertex is superlinear with respect to degree, so limiting the maximum out-degree created during the ordering phase can greatly reduce the amount of algorithmic work in the subsequent counting phase. For this reason, Pivoter employs a core ordering which guarantees the lowest maximum out-degree [18]. Even though computing a core ordering is sequential, for moderate or larger clique sizes, the time it saves in the counting phase typically outweighs the time a fast parallel ordering (to create an inferior directionalized graph) would save in the ordering phase (Table III). Since the time spent in the counting phase typically depends on the maximum out-degree produced by the ordering, we use the maximum out-degree to determine the quality of an ordering.

A. Parallel Core Ordering Approximation

While maintaining ordering quality, we accelerate the ordering phase since computing an ordering sequentially can consume a significant fraction of the overall time (Algorithm 2). We take inspiration from an existing parallel approximation presented by Besta et al. applied to graph coloring [25]. While the core ordering algorithm used in Pivoter removes a single vertex (with the least degree) at a time, the approximation removes vertices in bulk over multiple parallel rounds. The approximation selects vertices for removal if their degree is

Algorithm 2 Parallel Core Ordering Approximation

```
1: function APPROXCOREORDER(G, ϵ)
2:     level ← 0
3:     while |V(G)| > 0 do
4:         remove ← {}                          ▷ Vertices to remove this round
5:         δ ← |E|/|V|                          ▷ Average degree
6:         for all u ∈ V(G) in parallel do
7:             if d(u) < (1 + ϵ) × δ then
8:                 remove ← remove ∪ {u}
9:                 ω[u] ← level                 ▷ Assign rank to vertex
10:        V(G) ← V(G) \ remove                 ▷ Remove vertices
11:        for all u ∈ remove in parallel do
12:            for all v ∈ N(u) do
13:                d(v) ← d(v) − 1              ▷ Update degrees
14:        level ← level + 1
15:    return ω
```

less than $(1+\epsilon)\delta$, where ϵ is an error parameter and δ is the average degree of the remaining graph. It then removes the vertices and updates the degrees of their neighbors. This approach processes and removes a large fraction of the vertices in parallel during the first few rounds.

The parameter ϵ allows for a tradeoff between ordering quality and parallelism. Increasing ϵ causes more vertices to be removed each round, which increases the amount of parallelism but lowers the quality of the ordering. Setting ϵ sufficiently high produces an ordering similar to a degree ordering, while setting ϵ sufficiently low results in an approximation of the core ordering. In our experiments, we sweep various values for the parameter ϵ and find that ordering quality and counting performance is typically best when the maximum out-degree is equal to that produced by the core ordering (Section VI-C).

When we use our ordering approximation to directionalize the graph, we need a tiebreaker since the rankings produced are not unique (they are based on the round removed). An effective tiebreaker is to first use the original degree and then the vertex identifiers if necessary.

B. *Parallel* k-*core Ordering*

The process of peeling vertices to generate the core ordering is also similar to computing a graph's *k*-core decomposition. A *k*-core of a graph is a maximal connected subgraph such that every vertex in the subgraph has a degree of at least *k*. A *k*-core decomposition assigns the largest *k* to each vertex for which *k*-core that vertex belongs to. We can use the *k*-core decomposition to produce an ordering that directs edges from vertices with a lower *k*-core number to vertices with a higher *k*-core number. However, since multiple vertices can share a *k*-core number, a tiebreaker is required. We use the same tiebreaker as our parallel core approximation: degree first, and vertex identifier second. There are already parallel algorithms to compute a *k*-core decomposition such as ParK [26] and PKC [27].

Depending on the ϵ parameter, our core approximation will use a greater or fewer number of rounds than a parallel k-core decomposition, and thus a greater or fewer number of distinct rankings. When ϵ is sufficiently small that our core approximation produces more distinct rankings over a greater number of rounds, it can produce a higher-quality ordering.

C. Centrality-Based Ordering

To better appreciate what characteristics make an ordering approximation effective, we analyze the differences between the core and degree orderings. Both orderings often produce mosty similar directionalizations, and they differ for only about 3–5% of edges. Those differences typically involve high-degree vertices. The degree ordering will rank simply based on the initial degree, while the core ordering effectively also considers the degrees of the neighbors. Since the core ordering peels the lowest degree vertex remaining at a time, the vertices with the highest rankings will also have the highest degree neighbors. Thus, a core ordering ranks vertices mostly by *importance* akin to PageRank [28].

We propose using Eigenvector centrality to quickly rank *important* vertices in the graph [29]. The centrality ordering is simple to compute and is easy to parallelize, since it only requires summing up the scores of each vertex's neighbors' scores. This is even simpler than PageRank [28] since it does not require normalizing the scores. With a few iterations (3), the centrality ordering produces a maximum out-degree that lies between that of the core ordering and the degree ordering. While the centrality-based ordering never results in the fastest overall performance, it is always faster than the slower ordering between core and degree. This demonstrates that a successful ordering should not only consider degrees, but *importance* as well. It also demonstrates that importance can be quickly approximated.

D. Tradeoffs Impacting Which Ordering is Best

In our analysis, we find that different orderings perform better during the counting phase for different graph topologies. The fastest overall execution, including both ordering and counting, is typically achieved by the core approximation with $\epsilon = -0.5$ or the degree ordering (Figure 8). This challenges the expectation that core ordering is always better for pivoting. Although it is conceivable that a faster to compute ordering might result in a lower total time than the parallel core approximation, it is surprising it is faster specifically in the counting phase. Our analysis reveals that counting with the core ordering executes fewer instructions, but counting with the degree ordering executes instructions faster (Section VI-B). In other words, the core ordering has the expected algorithmic advantage, but the degree ordering has a practical speed advantage. Thus, which ordering is faster depends on whether the core ordering saves a sufficient amount of algorithmic work to overcome the degree ordering's speed advantage.

We next analyze why the degree ordering sometimes counts faster despite both orderings using the same counting implementation. Our analysis reveals that counting with a degree

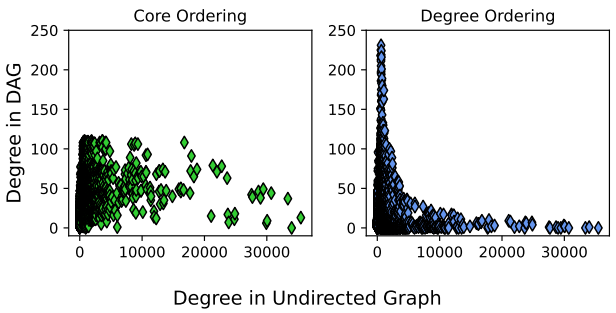

Fig. 3. Differences in degree distribution after directionalizing using a core ordering (left) and a degree ordering (right) on the Skitter graph.

ordering executes instructions faster due to fewer cache misses (Section VI-B). Both orderings start from the same graph, and the resulting DAGs have the same average degree, but their degree distributions can differ (Figure 3). In practice, the degree ordering does not achieve as low of a maximum out degree as the core approximation, but those extra edges reduce the degrees elsewhere.

The amount of work to process a vertex in the DAG depends on how many subsequent recursive function calls it produces (Algorithm 1). High-degree vertices result in a greater breadth of calls initially, and vertices in larger cliques result in a greater depth of calls. That initial breadth of calls from a high-degree vertex results in more reuse (cache locality) for the induced subgraph ($g_{\vec{v}}$). Thus, the degree ordering's speed advantage is due to it enjoying more reuse due to its edges being concentrated in higher-degree vertices. Counting with the core approximation is faster when the algorithmic work advantage is substantial, and empirically, we observe that occurs with graphs with more cliques. The greater recursive call depth from those cliques compounds the amount of work (Table II).

E. Heuristic to Select Best Ordering

Our insights from the last section indicate that the degree ordering is faster overall when the graph has relatively few cliques. Thus, a heuristic can decide which ordering to select if it can coarsely predict the prevalence of cliques. Cliques are pockets of density in an otherwise sparse graph. Naturally, a large clique requires its members to have at least moderately high degrees. In an *assortative* network, high-degree vertices are more likely to be connected to other high-degree vertices, and social networks are assortative [30]. Large cliques typically emerge in assortative networks as the density of high-degree vertices encourages their formation.

We present a simple heuristic leveraging our assortativity insight. In the original graph, we identify the highest degree vertex, and consider the highest degree a of its neighbors. If a is large, it means the highest degree vertex has a high-degree neighbor, and the graph is likely assortative and thus more likely to have many cliques. To adapt to different graphs, we normalize to the number of vertices. Empirically (Table IV), we find if $a/|V| \geq 0.0015$, it is better to use the core approximation (large cliques are likely), and otherwise, use the degree ordering.

To add robustness to our heuristic, we also test the assortativity of the graph by measuring the number of common neighbors between the highest-degree vertex and its highest-degree neighbor. Empirically, we find that over 10% of the neighbors are common between the two vertices when there are sufficiently many cliques to merit the core ordering approximation. We also find that the degree ordering is more advantageous for smaller graphs ($|V| < 1M$), where ordering is a significant fraction of the total time (Table III). In summary, we select the core approximation if the graph is sufficiently large and if either $a/|V| \geq 0.0015$ or over 10% of neighbors are common between the two vertices, and the degree ordering otherwise.

IV. Improving Counting Phase Scalability

The counting phase is important to optimize since it is the longest phase of clique counting. We base our counting approach on Pivoter, the most algorithmically efficient pivoting implementation [18]. While processing a vertex, it first builds an induced subgraph and then modifies it in subsequent recursive calls to count all of the cliques originating from that vertex. This approach is compatible with a vertex-parallel strategy, since the induced subgraphs that are modified are also independent. A vertex-parallel strategy brings up two potential concerns: load imbalance due to the skewed degree distributions of real-world graphs and increased memory requirements for large thread-local structures.

We analyze the potential for load imbalance by both attempting to improve it and measuring how much time each thread is working. Sweeping various scheduling parameters such as task granularity (chunk sizes) and scheduler types (static, dynamic, cyclic) does not fully improve parallel scalability. Additionally, we measure the time required for each thread during the entire counting phase while executing with 64 threads. The coefficient of variance (the ratio of standard deviation to the mean) for the execution time of each thread across the entire suite of input graphs is 0.03. Through our analysis, we find that load balance is at most a minor factor, and memory usage hinders scalability to a much greater extent.

The induced subgraph data structure is performance critical for counting cliques, and since it is thread-local, its memory consumption is also important. The original Pivoter code represents the subgraph with an adjacency list that has an array of size $|V(G)|$ pointing to inner arrays that store the neighbors of the associated vertices (Figure 4A). This dense index array allows for constant-time access to neighbor lists, and we refer to this approach as *PivotScale (dense)*. With minor adjustments to memory allocation, we observe that such a dense structure often performs well, but the $|V(G)|$-sized index consumes substantial memory when there is one subgraph per thread. Note that if the number of threads is greater than the average degree of the graph, these indices alone will consume more memory than the original graph.

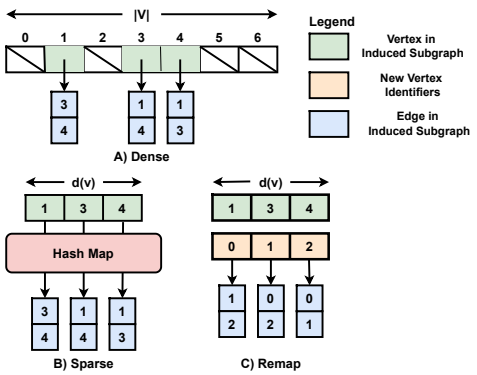

Fig. 4. Comparison between different subgraph structures for storing the first-level induced subgraph for vertex 0 (Figure 2) in PivotScale. Only the new vertex identifiers and associated edges are stored in C) Remap. Our default implementation uses C) Remap.

To compress the subgraph and improve cache locality, we only index vertices with non-zero degree in the subgraph with a hash map to reach their neighbor lists (Figure 4B). Since the number of non-zero degree vertices in the subgraph is bounded by the maximum out degree in the DAG (which is on the order of 100s, as opposed to $|V(G)|$, which is on the order of millions), this may even allow the entire subgraph to fit in cache. We refer to this approach as *PivotScale (sparse)*. For large graphs like Friendster, this optimization is able to overcome the scaling plateau from 32 threads to 64 threads.

In our experiments, we observe a lookup into a hash map to be $\approx 1.2\times$ slower than a direct array access. To combine the memory efficiency of a compressed subgraph structure with the benefit of fast indexing, we present a new subgraph structure. Instead of using a hash map, we remap the vertex identifiers in the first level subgraph induced by v to the compact range $[0, d(v))$ and use a dense index (Figure 4C). We only do this remapping step in the first level of the recursion, and reuse the new identifiers in subsequent levels since the additional space savings are modest. We refer to this approach as *PivotScale (remap)*, and we use it by default due to its leading performance and scalability.

V. Implementation Details

A. Implementation Subtleties for Pivoter

There are some additional details to the Pivoter algorithm and our implementation of it that are not conveyed in Algorithm 1 due to their complexity to express concisely. First, additional logic can be placed to allow the recursion to terminate early when the subgraph remaining combined with the clique size so far is too small to reach k (before line 9). Second, Algorithm 1 counts cliques of a target size k, but the original Pivoter algorithm can actually count the occurrences of every clique size up through k with only a modest amount of additional work to compute additional binomial coefficients in a loop (modify lines 9–10). We omit this detail since the prior work we compare to does not have this ability, but our released code provides an additional version capable of this.

There is a subtlety regarding how and when to direct the graph and subgraphs. Technically, pivoting and directionalization are independent concepts [20]. In this work, we take Pivoter's approach since we find it to perform the best on multicore [18]. Our approach within Algorithm 1: directionalize the graph first (line 2), symmetrize the subgraph when building the first level (line 5), maintain edge directions when building a subgraph for the pivot (line 16), and direct by vertex identifiers for non-neighbors of the pivot (line 18).

Counting cliques by recursively building subgraphs requires several sets in addition to the adjacency list for the topology. In prior implementations, these sets are canonically: vertices under consideration for the clique (P), vertices currently in the clique (R), and vertices no longer under consideration (X). These sets have the invariants that each vertex is in exactly one set and $P \cup R \cup X = V(G)$. By implementing these sets together and leveraging the invariants, the implementations can provide constant-time operations for testing set membership, iteration, and moving between sets [24].

To be able to reverse a mutation requires retaining information. Implementations typically place some of the information needed for reversing in the function call frame, so it naturally tracks the recursive calls. The remaining data needed to reverse a mutation can be recomputed by carefully examining the mutated graph. For example, recently deleted edges can be swapped from just beyond the active portion of neighbor lists.

B. Innovations Specific to PivotScale

We started our implementation from GAP Benchmark Suite reference code [31]. In addition to implementing our novel contributions, we focused on both optimizing the performance of subgraph mutations as well as encapsulating their complexity. Instead of continuing to use the P-R-X structure, we streamlined the object to have only a single sparse set to hold the subgraph's active vertices (P). We do not need set R since we are only counting cliques instead of listing them, and we also do not need set X to maintain the invariants. To provide constant-time membership checks for our sparse set (an array), we also use a dense bitmap. On the platforms we evaluate, we find using a byte per entry instead of a bit per entry performs even better. These lookups are frequent, and the remapping has already sufficiently shrunken the size of the dense structure.

Since the subgraphs are so frequently induced and mutated, we identified a need to make those operations fast. Our initial implementations were algorithmically fast, but experienced slowdowns in practice due to frequent memory allocations and deallocations. Although the operations are fast in isolation, if done repeatedly, they can cause memory fragmentation and the memory allocator can become a bottleneck. Thus, we redesigned our objects to reuse memory allocations to avoid the need for new allocations. This required vigilant tracking of mutations to make it easy to not only undo a subgraph induction, but to even reuse the entire subgraph object for a new vertex. The various arrays within these data structures are

Graph	Description	|V| (M)	|E| (M)	δ	k_{max}
DBLP	Citation network [32], [33]	0.3	1.1	3.7	114
As-Skitter	Internet topology [33], [34]	1.7	11.1	6.5	67
Baidu	Links between web pages [35], [36]	2.2	17.8	8.5	31
Wiki-Talk	Network of Wikipedia users [33], [37], [38]	2.4	9.3	3.9	26
Orkut	Social network [32], [33]	3.1	117.2	37.8	51
LiveJournal	Social network [32], [33]	4.0	34.7	8.1	-
Web-Edu	Links between .edu web pages [39]	9.9	46.2	2.4	449
Friendster	Social network [32], [33]	65.6	1,806.1	27.5	129

TABLE I

INPUT GRAPHS USED IN THE EVALUATION INCLUDING THE AVERAGE DEGREE (δ) AND SIZE OF THE LARGEST CLIQUE (k_{max}).

Graph	Normalized Instruction Count	Normalized Function Calls	Normalized LLC MPKI	Normalized IPC
DBLP	1.00	1.02	0.92	1.00
As-Skitter	1.52	1.44	0.66	1.04
Baidu	1.00	1.01	0.92	1.07
Wiki-Talk	1.36	1.35	0.83	1.01
Orkut	1.07	1.08	0.86	1.00
LiveJournal	1.28	1.21	1.09	0.97
Web-Edu	1.26	1.31	0.74	1.04
Friendster	1.00	1.02	0.88	1.04
geometric mean	1.16	1.17	0.85	1.02

TABLE II

COUNTING PHASE OF DEGREE ORDERING NORMALIZED TO CORE ORDERING. COUNTING WITH A DEGREE ORDERING ALWAYS EXECUTES MORE INSTRUCTIONS, BUT TYPICALLY EXECUTES THEM FASTER DUE TO FEWER CACHE MISSES (MPKI). LIVEJOURNAL IS A VERY CHALLENGING GRAPH DUE TO THE EXTREMELY LARGE NUMBER OF CLIQUES, AND THE CORE ORDERING HAS BETTER PRACTICAL PERFORMANCE IN ADDITION TO SUPERIOR ALGORITHMIC EFFICIENCY.

all vectors, so they can grow when needed but do not need to initially over-allocate. However, most of the time, the current need is less than prior allocations, so they can be efficiently reused.

Choosing to perform a vertex identifier remap operation only at the first level subgraph operation makes our implementation efficient. The hash maps have a significant computational cost, so using them frequently, as done by the sparse subgraph, becomes the dominant operation. The dense subgraph benefits from no computational overheads to access elements, but the majority of the elements are unused, so the cache capacity is used ineffectively. By performing a remap operation at the first level, we shrink the identifier range to be sufficiently small that dense structures are practical, and we pay the overhead of the hash map only once rather than for every graph operation.

VI. EVALUATION

A. Experimental Setup

We perform our experiments on a single-socket AMD EPYC 9554 (Genoa) which has: 64 $\times$ 3.1 GHz physical cores, 256 MB of shared L3 cache, and 768 GB of RAM. We use 64 threads unless specified otherwise. Our implementation uses C++20 with OpenMP, and is compiled with g++ version 12.2.0 and optimization level -O3.

We use a variety of input graphs to evaluate the performance of our optimizations (Table I). Since clique finding is used heavily in social network analysis, we select graphs commonly used in this subfield to make our analysis more consistent with prior work. All graphs are unweighted and symmetrized to initially be undirected. Unless specified otherwise, all experiments are for counting 8-cliques and our implementation uses our remapped subgraph structure.

B. Analyzing Prior Ordering Techniques

To appreciate the impact of the ordering phase on the overall execution time, we first consider the prior ordering approaches: core ordering and degree ordering (Table III). Although the ordering phase is usually only a modest portion of the overall execution time, it can have a significant impact because it can greatly affect the counting phase. We observe that in most cases, the core ordering results in faster counting times. It reduces the amount of algorithmic work in the counting phase by generally producing a lower maximum out-degree. For DBLP, Baidu and Friendster, we observe that a degree ordering actually results in a marginally faster counting time while being significantly faster in the ordering phase.

To investigate why a degree ordering results in better counting performance on certain graphs, we use hardware performance counters to profile the counting phase (Table II). The degree ordering always executes more instructions due to more recursive function calls, which is expected since it has a higher maximum out degree. A higher maximum degree causes larger induced subgraphs, and consequently, more instructions to process those larger subgraphs. Larger, denser subgraphs also generally lead to more recursive function calls with increased reuse, i.e. locality. That locality allows the degree ordering to execute instructions faster due to fewer cache misses (last-level cache misses per kilo-instructions (LLC MPKI)).

For the graphs the degree ordering counts faster (DBLP, Baidu, and Friendster), we observe the number of function calls and instructions executed are close to those from the core ordering (Table II). These graphs do not have as many densely connected subgraphs, resulting in comparable work for both orderings. Since the degree ordering counts faster and there is no algorithmic advantage for the core ordering, the degree ordering is faster overall. This demonstrates that a lightweight ordering can produce the fastest overall time, so our heuristic should consider a degree ordering is needed.

C. Analyzing Ordering Phase Tradeoffs

To achieve a high-quality ordering in less time than the inherently sequential core ordering, we consider the parallel core approximation (Section III-A), the parallel *k*-core (Section III-B), and the centrality orderings (Section III-C) from this work. The ϵ parameter for our parallel core approximation provides a tradeoff between ordering quality and parallelism. We consider many values of ϵ but report only the most representative ones for space concerns. For the parallel *k*-core ordering, we measure the time to generate the ordering using PKC [27], and we import that ordering into our implementation to measure its counting time.

We first compare the orderings by quality as measured by the maximum out degree (Figure 5). The ϵ parameter allows our core approximation to achieve a range of quality, by setting it sufficiently low (-0.5) to match the core ordering

Graph	Core Ordering				Degree Ordering			
	Ordering Time (s) (1 thread)	Counting Time (s) (64 threads)	Total Time (s) (64 threads)	Max. Out-Degree	Ordering Time(s) (64 threads)	Counting Time (s) (64 threads)	Total Time (s) (64 threads)	Max. Out-Degree
DBLP	0.03	0.02	0.05	113	0.00	0.02	**0.02**	113
As-Skitter	0.32	0.53	**0.85**	111	0.01	1.73	1.74	231
Baidu	0.61	0.19	0.80	78	0.02	0.18	**0.19**	298
Wiki-Talk	0.15	0.86	**1.01**	131	0.01	2.69	2.70	340
Orkut	3.11	19.99	23.10	253	0.05	22.93	**22.98**	535
LiveJournal	1.34	2562.86	**2564.20**	360	0.02	3619.24	3619.26	524
Web-Edu	1.25	1.04	2.29	448	0.02	2.09	**2.11**	448
Friendster	126.36	58.26	184.62	304	1.68	56.24	**57.92**	868

TABLE III

COUNTING 8-CLIQUES COMPARING SEQUENTIAL CORE AND DEGREE ORDERINGS IN TERMS OF ORDERING TIME, ORDERING QUALITY (MAXIMUM OUT DEGREE), AND COUNTING TIME. THE FASTEST OVERALL TIMES ARE BOLDED. THE CORE ORDERING IS GUARANTEED TO PRODUCE THE LOWEST MAXIMUM OUT-DEGREE, WHICH TYPICALLY REDUCES THE WORK IN THE COUNTING PHASE.

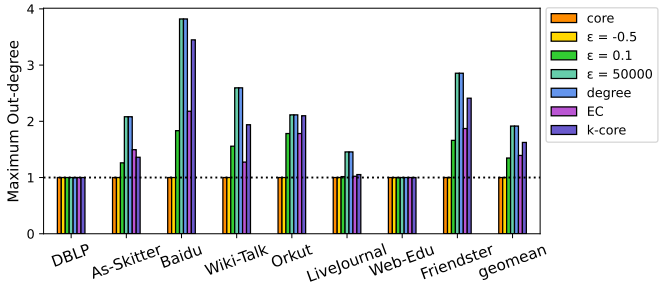

Fig. 5. Normalized maximum out-degree, a measure of ordering quality. All values are normalized to the maximum out-degree of core ordering. If the ϵ parameter for our parallel core approximation is sufficiently low, the ordering closely matches the core ordering in quality.

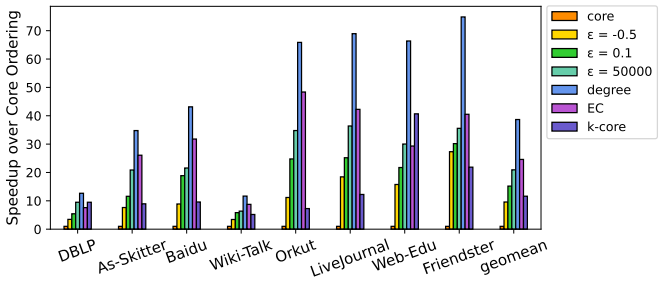

Fig. 6. Ordering time speedup over the core ordering. On larger graphs, our core approximation is significantly faster. In addition to being fast, our approximation with $\epsilon = -0.5$ produces the same maximum out-degree as the core ordering, which results in the counting phase time between both to be comparable. Degree ordering is always the fastest ordering, but it does not always result in the fastest counting times.

or setting it sufficiently high (50,000) to match the degree ordering. Even with a low ϵ, the approximation is able to remove more than 50% of vertices in the first round without a measurable decrease in quality. By ranking vertices based on importance, the centrality ordering (EC) produces a reasonable quality ordering that lies in between that of the core and degree orderings. There are some graphs (DBLP, Web-Edu) where the ordering quality remains the same for all values of ϵ. Further analysis of the DAG topologies show similar degree distributions, but the degree ordering still has marginally more higher degree vertices. We also observe that the parallel *k*-core ordering has a consistently worse ordering quality than our core approximation orderings with low ϵ.

Comparing the ordering times demonstrates how expensive a core ordering is (Figure 6). The degree ordering is the fastest since it only requires a single parallel round and the centrality ordering is quite quick since it only requires 3 rounds. Our core approximation is parallelized, but it still must synchronize between rounds, so varying ϵ and thus the number of rounds, impacts the ordering time. With $\epsilon = -0.5$, the ordering requires 160–6033 rounds, which still allows for a $9.58\times$ speedup over the exact sequential core ordering. The speedup is larger for larger graphs, and our approximation is able to produce essentially the same ordering quality more quickly. Increasing $\epsilon = 0.1$ greatly reduces the number of rounds (8–15), but that additional ordering speedup may not be worth the loss in ordering quality. The parallel *k*-core ordering is faster to produce than the core approximation ordering with $\epsilon = -0.5$,

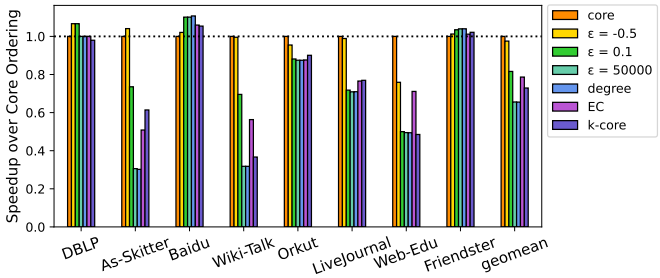

Fig. 7. Counting time speedup over core ordering for counting 8-cliques. The core ordering and our parallel core approximation generally are the fastest due to their algorithmic efficiency. Graphs like DBLP, Baidu and Friendster benefit more from a degree ordering. Note DBLP is the smallest graph and is more easily perturbed by minor overheads.

but slower than the degree or centrality-based orderings.

We observe that the core ordering typically results in the best counting times (Figure 7). The low maximum out-degrees it produces results in smaller subgraphs, and less work to build and process those subgraphs. While the other orderings (core-approximation, degree-based, centrality-based, parallel *k*-core) are faster to compute, the resulting DAG topology results in more work in the counting phase due to the higher maximum out-degree. However, graphs like DBLP, Baidu and Friendster have comparable counting phase performance across the different orderings, and thus present an opportunity for using a lightweight ordering.

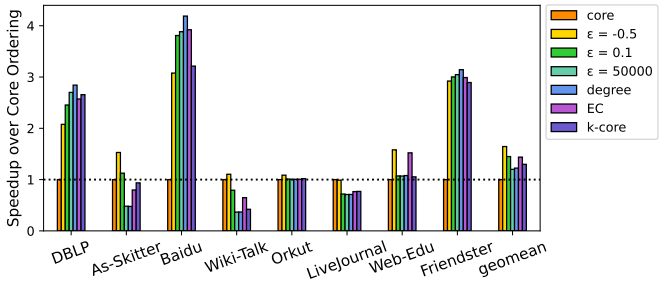

Fig. 8. Total execution time for counting 8-cliques speedup over core ordering. In graphs where core ordering results in the fastest counting, our parallel core approximation with $\epsilon = -0.5$ is much faster overall due to time saved by computing the ordering in parallel.

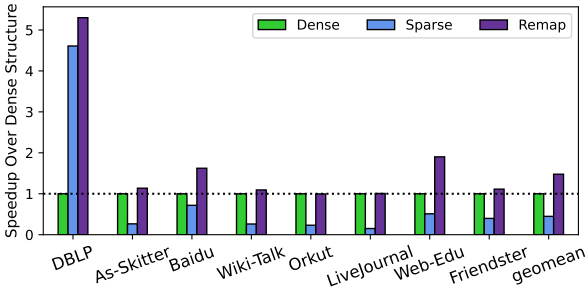

Fig. 9. Performance of different subgraph structures in PivotScale normalized to the dense structure for counting 8-cliques with 64 threads. The remapped structure provides the fast access of the dense structure and the memory compression of the subgraph structure, resulting in the best overall performance.

Combining the ordering and counting phases, we also measure the total time required to count 8-cliques using each ordering (Figure 8). In graphs where the core ordering is advantageous, we observe that the parallel core approximation with $\epsilon = -0.5$ results in the best overall times since the ordering is faster to compute. Besta et al. find setting $\epsilon = 0.1$ to be a good compromise for their graph coloring application. By contrast, our clique counting application is more sensitive to ordering quality and we find the additional ordering time to be worthwhile. Surprisingly, graphs like DBLP, Baidu and Friendster have better overall performance with the degree ordering, despite the ordering quality being inferior in some cases (Section III-D). Depending on the value of ϵ selected, we can tune our approximation towards core or degree. The centrality and k-core orderings are in the middle in overall performance as they are never the fastest or slowest.

D. Analyzing PivotScale Subgraph Data Structures

We evaluate the different subgraph data structures in PivotScale on memory consumption and performance. We measure the memory consumption for the entire process[1], and find the subgraph data structures can greatly contribute to it. Using 64 cores, the process using dense structures requires 811.67 MB for the smallest graph (DBLP) and 265.69 GB for the largest graph (Friendster). As expected, we find that our compact sparse and remapped structures reduce memory consumption by $6.63 - 40.24\times$ (geometric mean: $17.39\times$). Consequently, our sparse and remapped structures result in $1.24 - 77.00\times$ (geometric mean: $9.98\times$) fewer cache misses (LLC MPKI). With these optimizations, we are able to fit a larger set of the subgraph in cache, resulting in better locality. As core count, and consequently contention for shared memory increases, our remapped structure becomes a more scalable solution. The remapped structure is also fast to access (Figure 9), so we use it in our main implementation.

[1]Maximum memory usage collected via the `Maximum resident set size` reported by the command `/usr/bin/time -v`.

Graph	Best Ordering	a	|V| (M)	a/|V|	Common Fraction	Heuristic Time (s)
DBLP	degree	296	0.3	0.0010	0.72	0.00
As-Skitter	core	33,982	1.7	0.0200	0.84	0.01
Baidu	degree	2,867	2.2	0.0013	0.00	0.01
Wiki-Talk	core	10,520	2.4	0.0044	0.11	0.01
Orkut	core	29,657	3.1	0.0945	0.12	0.01
LiveJournal	core	1,705	4.0	0.0004	0.20	0.01
Web-Edu	core	18,293	9.9	0.0019	0.90	0.04
Friendster	degree	3,117	65.6	0.0000	0.00	0.24

TABLE IV
ORDER-SELECTING HEURISTIC INPUTS, MEASUREMENTS, AND DECISIONS FOR COUNTING 8-CLIQUES. OUR HEURISTIC SELECTS OUR CORE APPROXIMATION IF THE GRAPH IS LARGE ENOUGH ($|V| > 1M$), AND IF $a/|V| > 0.0015$ OR IF THERE ARE MORE THAN 0.10 COMMON NEIGHBORS, AND USES A DEGREE ORDERING OTHERWISE. OUR HEURISTIC ALWAYS SELECTS THE CORRECT ORDERING. THE TIME TO COMPUTE THE HEURISTIC IS TINY.

E. Validating Our Order-Selecting Heuristic

Our heuristic can quickly compute whether a given graph will benefit from the algorithmic efficiency of our core approximation or the increased locality and faster ordering of a degree ordering (Table IV). To determine whether the target clique size (k) impacts the best ordering, we measure the total execution time of the core approximation with $\epsilon = -0.5$, the degree ordering, and our heuristic selecting the ordering for counting varied clique sizes (Figure 10). On larger graphs, the degree ordering is usually faster for $k = 4$ due to the speed of ordering. However, once the clique size is sufficiently large for pivoting to be faster ($k \geq 8$), which ordering is best does not change. Since the total execution time for pivoting does not change significantly with k, our heuristic does not consider k.

F. Parallel Scaling of k-Clique Counting with PivotScale

We evaluate the parallel scaling of our optimizations when counting 6-cliques and 12-cliques considering all of our subgraph structures on all input graphs (Figure 11). We report self-relative speedups, i.e. performance relative to the single-thread performance of that implementation. These speedups include all associated times, including the time to compute the heuristic as well as the ordering and counting phases. Since the counting phase dominates the execution time, the scalability of the counting phase primarily determines the overall parallel scalability.

We observe that all implementations of PivotScale achieve near-linear scaling for most graphs. In Baidu and Web-Edu, we observe that scaling for PivotScale's dense structure plateaus

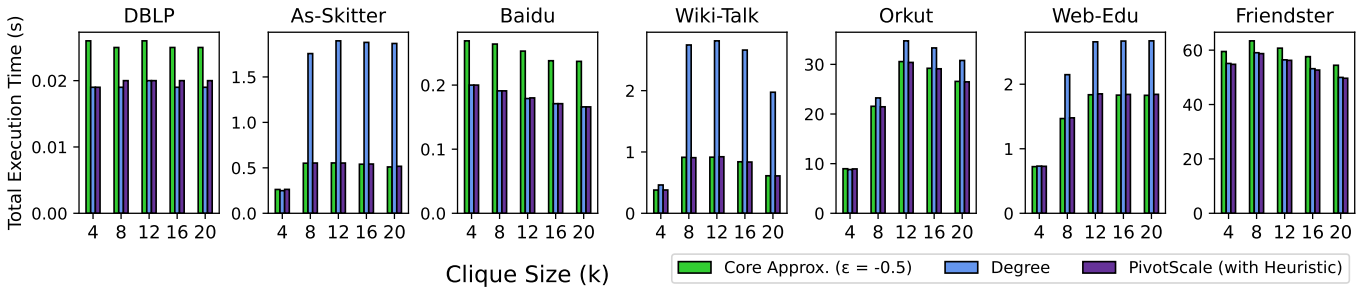

Fig. 10. Total execution time for counting varied clique sizes using only our core approximation, only the degree ordering, or the ordering selected by our heuristic. Our heuristic always selects the correct ordering and it does not add significant overhead. Using our heuristic to select a better ordering results in a $0.99 - 1.43\times$ (geometric mean: $1.10\times$) speedup over using only our core approximation ordering.

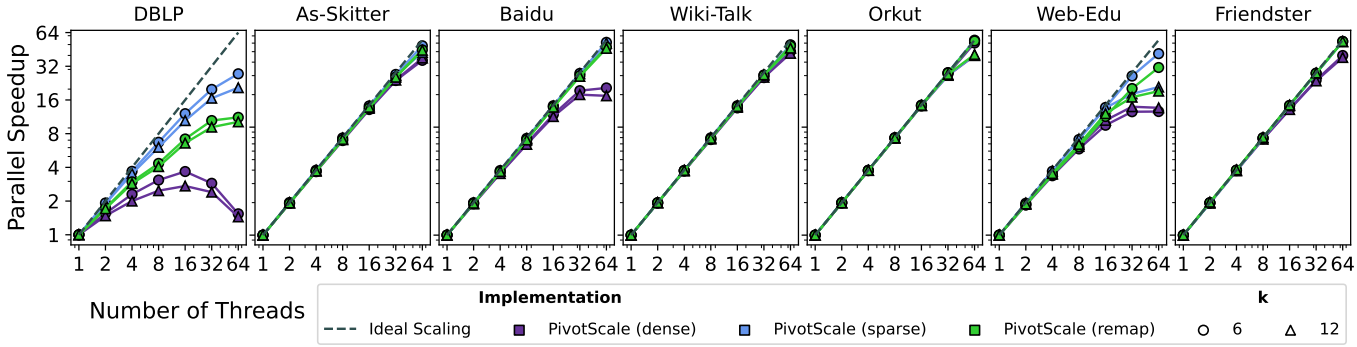

Fig. 11. Parallel scalability of PivotScale with different subgraph structures for counting 6-cliques or 12-cliques. The time for each run includes the time to compute the heuristic, the ordering phase, and the counting phase. Scaling is self-relative, i.e. normalized to the single core performance of that implementation. For most graphs, all of our PivotScale structures scale enjoy good scalability. For Baidu and Web-Edu, memory becomes a bottleneck for our dense implementation at 32 threads, but our more compact sparse and remapped structures avoid this and scale linearly even beyond 32 threads. DBLP is small so all implementations have insufficient parallelism. PivotScale (remap) is the fastest in absolute performance.

at 32 threads. In contrast, by compressing the subgraph using PivotScale's sparse or remapped structures, we are able to achieve linear scaling for both of those graphs. This greatly increased scalability is largely enabled by the more efficient memory use of our subgraph data structures.

Scaling for DBLP plateaus beyond 8 threads as it is a small graph that does not have sufficient parallel work for the duration of the run. Testing various schedulers with different task granularities does not improve performance. Despite poor scalability, the total execution time is very small (0.04 s for our remapped structure). Our remapped subgraph structure is the fastest overall (Figure 9), so we use it in the rest of the evaluation.

G. Comparison Against Prior Work

We compare PivotScale against prior work for counting cliques of various sizes (Figure 12 & Table V). When reporting the execution time, we include all preprocessing, including the time required to compute our heuristic and directionalize the graph. Consistent with prior work, we exclude the time required to read and build the undirected input graph. Each execution uses 64 threads, and we report the average time of two trials to account for variance. We use GPU-Pivot's reported execution times for the NVIDIA Volta V100 and Ampere A100 GPUs, but those times do not include cliques larger than $k = 11$ [20].

We observe that Arb-Count's execution time increases greatly with clique size (k) for most graphs, while the pivoting approaches increase much more slowly (Figure 12), and this is consistent with prior findings. A pivoting approach has a high fixed cost irrespective of target clique size, and counting larger cliques incurs nearly negligible additional time. Initially, Pivoter starts to outperform the enumeration approach (Arb-Count) on CPUs at $k = 10$. Most notably, our work with PivotScale's parallel scalability enables pivoting to be more advantageous earlier, at $k = 8$ for larger graphs. PivotScale generally outperforms or at least matches the performance of GPU-Pivot.

We also observe that the total execution time for GPU-Pivot increases significantly with target clique size (k) on clique-rich graphs such as As-Skitter and Orkut (Figure 12). We suspect this is due to an increased number of intersection operations while building the subgraph in each recursive level. Additionally, building a single subgraph per-warp does not allow for efficient use of the GPU's hardware resources, inhibiting scalability. Our approach is more scalable for counting large cliques in challenging graphs, and the execution time for PivotScale does not increase as significantly with k. Both

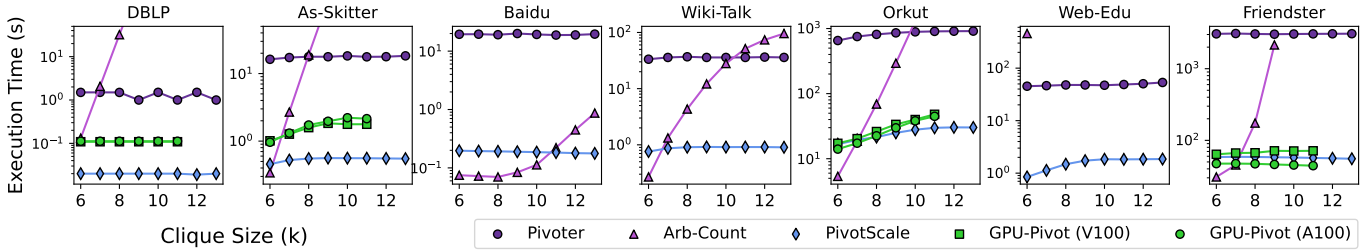

Fig. 12. Total execution time (on a log scale) required for counting cliques of different sizes on the input graphs for each of the CPU algorithms (Pivoter [18], Arb-Count [19], PivotScale) and GPU-Pivot running on an NVIDIA Volta V100 and an NVIDIA Ampere A100 GPU. All CPU implementations are executed using 64 threads. Lower is better. GPU-Pivot does not report times for $k > 11$. We observe that the lone enumeration-based algorithm (Arb-Count) takes longer for higher values of k. In contrast, the pivoting-based approaches typically do not get slower for higher k. Due to improved parallel scaling, PivotScale is much faster than Pivoter, despite requiring constant time for various k. This allows the inflection point at which pivoting starts to win to decrease from $k = 10$ to $k = 8$ on larger graphs. Since the pivoting algorithm is unable to fully utilize the hardware capabilities of a GPU, we do not observe a significant improvement in performance of GPU-Pivot from the V100 to the A100. Due to better scaling, PivotScale routinely outperforms GPU-Pivot with the exception of lower k on Friendster with the A100. The raw data is available in Table V.

Graph	Algorithm	k=6	k=7	k=8	k=9	k=10	k=11	k=12	k=13
DBLP	Pivoter	1.50	1.00	1.50	1.00	1.50	1.00	1.50	1.50
	Arb-Count	0.13	2.07	32.11	450.86	$> 2h$	$> 2h$	$> 2h$	$> 2h$
	GPU-Pivot (V100)	0.11	0.11	0.11	0.11	0.11	0.11	-	-
	GPU-Pivot (A100)	0.11	0.11	0.11	0.11	0.11	0.11	-	-
	PivotScale	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**
As-Skitter	Pivoter	16.26	17.27	17.77	17.74	18.26	17.69	17.78	18.29
	Arb-Count	**0.38**	2.51	18.34	125.52	754.08	4189.38	$> 2h$	$> 2h$
	GPU-Pivot (V100)	1.01	1.27	1.59	1.84	1.78	1.78	-	-
	GPU-Pivot (A100)	0.96	1.31	1.73	1.97	2.22	2.15	-	-
	PivotScale	0.46	**0.52**	**0.55**	**0.56**	**0.56**	**0.56**	**0.55**	**0.55**
Baidu	Pivoter	19.44	19.52	19.11	20.03	19.31	18.85	18.94	19.57
	Arb-Count	**0.07**	**0.07**	**0.07**	**0.08**	**0.11**	0.22	0.45	0.90
	PivotScale	0.20	0.19	0.19	0.19	0.19	**0.18**	**0.18**	**0.18**
Wiki-Talk	Pivoter	33.42	35.91	36.91	35.93	35.91	35.93	36.45	35.95
	Arb-Count	**0.28**	1.32	4.60	13.24	28.60	51.30	73.87	95.76
	PivotScale	0.76	**0.87**	**0.91**	**0.92**	**0.91**	**0.91**	**0.91**	**0.90**
Orkut	Pivoter	654.13	753.08	812.71	858.04	889.39	904.02	909.91	912.99
	Arb-Count	**5.35**	18.58	69.89	281.03	1294.34	$> 2h$	$> 2h$	$> 2h$
	GPU-Pivot (V100)	17.23	20.33	26.18	33.64	39.96	48.10	-	-
	GPU-Pivot (A100)	14.05	**17.32**	22.48	29.82	38.22	44.82	-	-
	PivotScale	16.72	19.48	**21.47**	**24.97**	**27.91**	**29.83**	**30.32**	**30.20**
Web-Edu	Pivoter	45.29	46.36	47.84	47.82	47.25	48.79	50.47	53.35
	Arb-Count	456.47	$> 2h$	$> 2h$	$> 2h$	$> 2h$	$> 2h$	$> 2h$	$> 2h$
	PivotScale	**0.85**	**1.13**	**1.48**	**1.73**	**1.84**	**1.83**	**1.84**	**1.86**
Friendster	Pivoter	3,064.48	3,097.26	3,054.73	3,032.45	3,050.13	3,063.23	3,070.55	3,080.26
	Arb-Count	**30.77**	**44.19**	166.53	2132.27	$> 2h$	$> 2h$	$> 2h$	$> 2h$
	GPU-Pivot (V100)	63.87	66.54	67.06	71.40	71.05	71.45	-	-
	GPU-Pivot (A100)	47.32	47.41	**47.07**	**46.12**	**45.22**	**44.31**	-	-
	PivotScale	58.48	58.88	58.69	58.12	57.66	56.87	**56.19**	**55.40**

TABLE V

TOTAL EXECUTION TIME FOR COUNTING (IN SECONDS) k-CLIQUES USING PIVOTER [18], ARB-COUNT [19], GPU-PIVOT [20] AND PIVOTSCALE. WE USE THE TIMES REPORTED BY GPU-PIVOT IN THEIR PAPER. EVERY OTHER ALGORITHM IS EXECUTED USING 64 THREADS ON THE SAME MACHINE (CPU) UNDER THE SAME CONDITIONS. THE EXECUTION TIMES REPORTED INCLUDE ANY PREPROCESSING, INCLUDING GRAPH ORDERING, BUT IGNORE GRAPH READING TIMES. THE FASTEST EXECUTION TIMES ARE DENOTED IN BOLD.

GPU-Pivot and PivotScale do not show significant variation in execution time relative to k on graphs where work does not increase significantly (e.g. DBLP and Friendster).

H. Comparison on LiveJournal

LiveJournal is a computationally challenging graph with many cliques and we perform this evaluation separately since using less than an optimized implementation at full parallelism results in unreasonable execution times. We observe that the execution time of both implementations increases significantly with k for LiveJournal (Figure 13), compared to As-Skitter and Orkut, other relatively challenging graphs (Figure 12). The original Pivoter code includes an optimization to terminate early to count smaller cliques, specifically for LiveJournal. LiveJournal's density causes the recursion depth to grow dramatically with the target clique size k. Increasing k from 6 to 11 causes 942.16× more recursive function calls for LiveJournal, but for As-Skitter and Orkut, the same increase in k only results in 1.61× more calls.

We also observe that the execution time of PivotScale is less than that of GPU-Pivot for all k (Figure 13 and Table VI). In their paper, the authors of Pivoter report an execution time

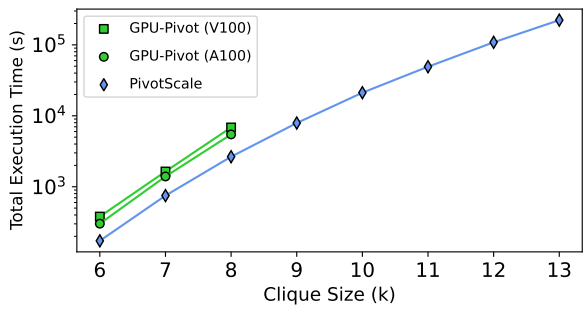

Fig. 13. Total execution times for GPU-Pivot and PivotScale for LiveJournal while scaling clique size. Since this graph has many cliques, the execution times for both increases dramatically with k, unlike the other input graphs.

k	Number of k-cliques	PivotScale	GPU-Pivot (V100)	GPU-Pivot (A100)
6	10,990,740,312,954	**172.92**	379.88	301.77
7	449,022,426,169,164	**750.00**	1,639.54	1,396.37
8	16,890,998,195,437,619	**2,650.87**	6,850.99	5,467.18
9	587,802,675,586,713,160	**7,906.71**	-	-
10	18,973,061,151,392,022,301	**21,172.76**	-	-
11	568,916,187,227,810,700,115	**49,213.59**	-	-
12	15,868,894,086,996,727,006,147	**108,621.55**	-	-
13	412,397,238,639,623,631,270,670	**223,130.87**	-	-

TABLE VI
NUMBER OF K-CLIQUES IN LIVEJOURNAL FOR VARIED k AND THE ASSOCIATED EXECUTION TIME (IN SECONDS) FOR PIVOTSCALE AND GPU-PIVOT. THE BEST EXECUTION TIME IS DENOTED IN BOLD. GPU-PIVOT DOES NOT REPORT EXECUTION TIMES FOR $k > 8$. PIVOTSCALE IS FASTER THAN GPU-PIVOT FOR ALL REPORTED k. THIS IS THE FIRST WORK TO REPORT COUNTS FOR $k > 10$.

of 5.9 days for counting 10-cliques in LiveJournal [18]. With PivotScale's improved scalability, we are able to count 10-cliques in under 6 hours using 64 threads. This is the first work to report counts for cliques larger than $k = 10$ in LiveJournal (Table VI).

Pivoting algorithms are more suited for counting large cliques in graphs and enumeration algorithms perform well for smaller cliques. A hybrid algorithm which performs well for all clique sizes can easily be implemented by switching with a simple heuristic e.g. $(k \geq 8)$. Due to its improved parallel scalability in both ordering and counting phases, PivotScale typically outperforms current state-of-the-art CPU and GPU algorithms for large clique sizes ($k \geq 8$).

VII. RELATED WORK

Chiba and Nishizeki present one of the fastest sequential algorithms for clique counting [40]. The algorithm is simple and efficient, but it includes a sequential step which prevents it from scaling to massive real-world graphs.

Finocchi et al. present a scalable clique counting algorithm that uses a degree ordering and is built on MapReduce [41]. KClist was the first work to parallelize Chiba and Nishizeki's algorithm by separating the ordering and counting phases [17]. More recently, Shi et al. present Arb-Count, a work-efficient parallel algorithm with polylogarithmic span [19]. It is the state-of-the-art enumeration-based clique counting algorithm. Aside from the core and degree ordering, they also implement the Barenboim-Elkin [42] and Goodrich-Pszona [43] orderings. Lonkar and Beamer present communication reducing optimizations to improve the scalability of enumeration-based clique counting [44].

Li et al. provide an optimized parallel algorithm which orders the graph on a coloring-based method to improve counting phase perfromance [45]. The search space their work prunes is inherent to the enumeration approach, and pivoting already avoids that redundant work. Lastly, GPU-Pivot implements enumeration and pivoting-based k-clique counting algorithms on GPUs [20]. Aside from the aforementioned dedicated solvers, various GPM frameworks are also capable of counting cliques in large graphs [12]–[16].

In addition to exact clique counting, related problems like maximal clique enumeration and approximate clique counting have also been heavily studied. Maximal clique enumeration involves finding cliques which cannot be extended any further [22]–[24], [46], [47]. Other recent works attempt to approximate k-clique counts in graphs through sampling [48]–[51] or probabilistic hashing [52].

VIII. CONCLUSION

In this work, we take a holistic approach to improve the parallel scalability of clique counting. We provide in-depth analysis of how the ordering phase impacts the counting phase. With inspiration from prior work, we create the parallel core ordering approximation which allows PivotScale to benefit from faster ordering times without sacrificing algorithmic efficiency while counting. We use our analysis to develop a heuristic which predicts which ordering will lead to the fastest counting, and consequently, fastest overall execution time for a given input graph. We also improve the parallel scalability of counting by reducing memory usage and lookup overheads with a compact subgraph structure.

Pivoter made finding large cliques in large graphs practical, and our efficient parallelization furthers that scalability. PivotScale achieves total speedups of $25.66 - 110.58\times$ (Geometric Mean: $47.05\times$) over Pivoter while counting k-cliques. PivotScale outperforms the fastest enumeration implementation (Arb-Count) for a smaller clique size than before. On challenging graphs with many cliques, PivotScale scales better with clique size, allowing it to outperform GPU-Pivot while counting larger cliques.

Our optimizations and analysis can be applied to similar problems within Graph Pattern Mining. In this work, we focus on counting the total number of k-cliques, but simple changes to our code could easily enable per-vertex k-clique counts. To enable others to benefit from our optimizations, we open-source our code publicly at: https://github.com/ucsc-vama/pivotscale

ACKNOWLEDGMENT

We thank Seshadhri Comandur for suggesting this project and helpful guidance along the way. We also thank Sabyasachi Basu and Daniel Paul-Pena for their advice during the rebuttal. Research partially funded by NSF Award Number 2119154, Principles and Practice of Scalable Systems (PPoSS).

REFERENCES

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

[2] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, no. 7043, pp. 814–818, 2005.

[3] 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," *Systems Journal*, vol. 11, no. 1, pp. 250–259, 2017.

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

[5] L. Pan and E. E. . Santos, "An anytime-anywhere approach for maximal clique enumeration in social network analysis," in *International Conference on Systems, Man and Cybernetics (SMC)*, 2008, pp. 3529–3535.

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

[7] S. Manoharan and Sathish, "Patient diet recommendation system using k clique and deep learning classifiers," *Journal of Artificial Intelligence and Capsule Networks*, vol. 2, no. 2, pp. 121–130, 2020.

[8] K. X. P. Vilakone and D. Park, "Personalized movie recommendation system combining data mining with the k-clique method," *Journal of Information Processing Systems*, vol. 15, no. 5, pp. 1141–1155, Oct. 2019.

[9] T. Marschall, I. G. Costa, S. Canzar, M. Bauer, G. W. Klau, A. Schliep, and A. Schönhuth, "CLEVER: clique-enumerating variant finder," *Bioinformatics*, vol. 28, no. 22, pp. 2875–2882, 10 2012. [Online]. Available: https://doi.org/10.1093/bioinformatics/bts566

[10] E. T. T. Matsunaga, C. Yonemori and M. Muramatsu, "Clique-based data mining for related genes in a biomedical database," *BMC Bioinformatics*, vol. 10, no. 205, 2009.

[11] K. C. D. Bahadur, T. Akutsu, E. Tomita, T. Seki, and A. Fujiyama, "Point matching under non-uniform distortions and protein side chain packing based on an efficient maximum clique algorithm," *Genome Informatics*, vol. 13, pp. 143–152, 2002.

[12] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: a two-level framework for efficient graph pattern mining," in *International Conference on Supercomputing (ICS)*, 2021, pp. 378–391.

[13] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An efficient and flexible graph mining system on cpu and gpu," *VLDB*, vol. 13, no. 8, pp. 1190–1205, 2020.

[14] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: a pattern-aware graph mining system," in *European Conference on Computer Systems (EuroSys)*, 2020, pp. 1–16.

[15] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 425–440.

[16] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *International Conference on Management of Data (MOD)*, 2019, pp. 1357–1374.

[17] M. Danisch, O. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs," in *World Wide Web Conference (WWW)*, 2018, pp. 589–598.

[18] S. Jain and C. Seshadhri, "The power of pivoting for exact clique counting," in *International Conference on Web Search and Data Mining (WSDM)*, 2020, pp. 268–276.

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

[20] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, "Parallel k-clique counting on gpus," in *International Conference on Supercomputing (ICS)*, 2022, pp. 1–14.

[21] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 417–427, 1983.

[22] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[23] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theoretical computer science*, vol. 363, no. 1, pp. 28–42, 2006.

[24] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in sparse graphs in near-optimal time," in *International Symposium on Algorithms and Computation (ISAAC)*. Springer, 2010, pp. 403–414.

[25] M. Besta, A. Carigiet, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, and T. Hoefler, "High-performance parallel graph coloring with strong guarantees on work, depth, and quality," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2020, pp. 1–17.

[26] N. S. Dasari, R. Desh, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 9–16.

[27] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 1482–1491.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.

[29] P. Bonacich, "Factoring and weighting approaches to status scores and clique identification," *Journal of mathematical sociology*, vol. 2, no. 1, pp. 113–120, 1972.

[30] M. E. Newman, "Assortative mixing in networks," *Physical review letters*, vol. 89, no. 20, p. 208701, 2002.

[31] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[32] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *ACM SIGKDD workshop on mining data semantics*, 2012, pp. 1–8.

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

[34] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *International Conference on Knowledge Discovery in Data Mining (KDD)*, 2005, pp. 177–187.

[35] X. Niu, X. Sun, H. Wang, S. Rong, G. Qi, and Y. Yu, "Zhishi.me – weaving chinese linking open data," in *International Semantic Web Conference (ISWC)*. Springer, 2011, pp. 205–220.

[36] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[37] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Signed networks in social media," in *SIGCHI conference on human factors in computing systems*, 2010, pp. 1361–1370.

[38] ——, "Predicting positive and negative links in online social networks," in *International Conference on World Wide Web (WWW)*, 2010, pp. 641–650.

[39] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[40] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM Journal on Computing*, vol. 14, no. 1, pp. 210–223, 1985.

[41] I. Finocchi, M. Finocchi, and E. G. Fusco, "Clique counting in mapreduce: Algorithms and experiments," *Journal of Experimental Algorithmics (JEA)*, vol. 20, pp. 1–20, 2015.

[42] L. Barenboim and M. Elkin, "Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition," *Distributed Computing*, vol. 22, no. 5-6, pp. 363–379, 2010.

[43] M. T. Goodrich and P. Pszona, "External-memory network analysis algorithms for naturally sparse graphs," in *European Symposium on Algorithms (ESA)*. Springer, 2011, pp. 664–676.

[44] A. Lonkar and S. Beamer, "Accelerating clique counting in sparse real-world graphs via communication-reducing optimizations," *arXiv preprint arXiv:2112.10913*, 2021.

[45] R. Li, S. Gao, L. Qin, G. Wang, W. Yang, and J. X. Yu, "Ordering heuristics for k-clique listing." *VLDB*, 2020.

[46] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast algorithms for maximal clique enumeration with limited memory," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012, pp. 1240–1248.

[47] W. Deng, W. Zheng, and H. Cheng, "Accelerating maximal clique enumeration via graph reduction," *VLDB*, vol. 17, no. 10, pp. 2419–3431, 2024.

[48] S. Jain and C. Seshadhri, "Provably and efficiently approximating near-cliques using the turán shadow: Peanuts," in *The Web Conference*, 2020, pp. 1966–1976.
[49] X. Ye, R.-H. Li, Q. Dai, H. Chen, and G. Wang, "Lightning fast and space efficient k-clique counting," in *The Web Conference*, 2022, pp. 1191–1202.
[50] ——, "Efficient k-clique counting on large graphs: The power of color-based sampling approaches," *IEEE Transactions on Knowledge and Data Engineering*, 2023.
[51] L. Chang, R. Gamage, and J. X. Yu, "Efficient k-clique count estimation with accuracy guarantee," *VLDB*, vol. 17, no. 11, pp. 3707–3719, 2024.
[52] M. Besta, C. Miglioli, P. S. Labini, J. Tětek, P. Iff, R. Kanakagiri, S. Ashkboos, K. Janda, M. Podstawski, G. Kwaśniewski *et al.*, "Probgraph: High-performance and high-accuracy graph mining with probabilistic set representations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2022, pp. 1–17.