SAND2019-10088C

Skip the Intersection: Quickly Counting Common Neighbors on Shared-Memory Systems

Xiaojing An*, Kasimir Gabert*, James Fox, Oded Green, David A. Bader
Georgia Institute of Technology, Atlanta, Georgia
Email: anxiaojing, kasimir, foxjas, ogreen @gatech.edu, bader@cc.gatech.edu
*The first two authors contributed equally to the work.

***Abstract*—Counting common neighbors between all vertex pairs in a graph is a fundamental operation, with uses in similarity measures, link prediction, graph compression, community detection, and more. Current shared-memory approaches either rely on set intersections or are not readily parallelizable. We introduce a new efficient and parallelizable algorithm to count common neighbors: starting at a wedge endpoint, we iterate through all wedges in the graph, and increment the common neighbor count for each endpoint pair. This exactly counts the common neighbors between all pairs without using set intersections, and as such attains an asymptotic improvement in runtime. Furthermore, our algorithm is simple to implement and only slight modifications are required for existing implementations to use our results. We provide an OpenMP implementation and evaluate it on real-world and synthetic graphs, demonstrating no loss of scalability and an asymptotic improvement. We show intersections are neither necessary nor helpful for computing all pairs common neighbor counts.**

I. Introduction

Computing the number of neighbors shared between two vertices, or the common neighbor count, is an important kernel in graph analytics: its direct applications include link prediction[1], graph compression[2], and uncovering topological modules[3]; it is foundational for vertex similarity measures[4], including the Jaccard Index[5], Resource Allocation[6], Adamic/Adar[7], cosine similarity[8], and topological overlap[3]; and it is computationally related to triangle counting, which is of considerable interest[9].

A straightforward way of computing the common neighbor count between any pair of vertices is to compute the size of the intersection of those vertices' adjacency lists. This allows each pair to be computed independently of other pairs, and to date this has been the primary approach used in existing shared memory algorithms and implementations. We show that such set-intersection based approaches have an unnecessary asymptotic factor in run-time complexity related to the degree of the graph, and viewing the problem through a different lens allows for a parallel, scalable algorithm that exactly computes common neighbors, and derivatives, without set intersections.

Consider a graph $G = (V, E)$, where $n = |V|$ represents the number of vertices and $m = |E|$ the number of edges. Without additional insights, getting the common neighbor counts between all pairs of vertices requires $O(n^2)$ set intersections, as each pair may have a non-zero common neighbor count. Prior work[10], [11], [2] have made the additional observation that the output will always be zero between two vertices unless there exists a *wedge*, or a path of length two, between them. Incorporating this insight reduces the number of required set intersections and brings the overall complexity cost down from $O(n^2\Delta)$ to $O(m\Delta^2)$, where Δ is the maximum degree in the graph, and with p processors to $O(n\Delta^3/p)$.

We demonstrate that we can do away with the set intersection completely. This is an asymptotic improvement and results in a parallelizable shared-memory algorithm running with $O(m\Delta)$ total operations and $O(n\Delta^2/p)$ per processor.

Our key insight stems from realizing that finding wedges[10], [11], [2], which is done to reduce the intersections from $O(n^2)$ to $O(m\Delta)$, already implicitly counts the common neighbors. Each wedge not only represents a non-zero between the endpoints, but represents exactly one of the common neighbor relationships between them. So, we keep track of the number of wedges iterated over for each endpoint, and this exactly counts common neighbors. This provides an asymptotic improvement by a factor of the maximum degree, which is crucial for graphs with many high degree vertices, and also allows for effective parallelism. This work confirms Gustavson's more general matrix multiplication results[12].

Our main contributions are as follows:

- We propose a new algorithm that is parallel and asymptotically faster than prior set intersection based methods
- We provide OpenMP implementations of our new algorithm and a competitive parallel set intersection approach
- We evaluate them on real-world and synthetic graphs demonstrate that our new algorithm is asymptotically faster than prior set intersection based approaches

The remainder of this paper is structured as follows. Section II describes related work, Section III presents notation, definitions, and current approaches, Section IV presents our wedge iteration algorithm and Section V describes our OpenMP and GAP[13] implementation. Section VI contains our experiments and results and Section VII concludes.

Funding was provided in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086, the U.S. Department of Homeland Security under Cooperative Agreement No. 2017-ST-061-QA001-01, and by the Doctoral Studies Program at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the funding agencies.

II. Related work

Many existing implementations for computing common neighbor counts and similarity indices are based on set intersections. NetworkX[14] is a Python package for studying complex networks, and includes many vertex similarity indices targeting link prediction, all using set intersection. Igraph[15] is a sequential C++ network analysis library that also implements several similarity indices. It computes the Jaccard and Sörensen indices using set intersection. For Adamic/Adar and cocitation is uses a midpoint wedge iteration method which achieves the time complexity of our methods, but is not memory-scalable, will not support efficient parallel implementations, and cannot effectively support top-k results and other applications. Sparkling-Graph[16] implements common neighbor counting and Adamic/Adar for link prediction using set intersections. Neo4j[17] is a graph database system that supports the Jaccard Index, among other indices, and its implementation is set-intersection based. Song et al. [18] compares common neighbor count and Adamic/Adar with other methods for link prediction and computes these methods again by measuring the size of set intersections.

Common neighbors and related similarity indices have also been developed for distributed systems. Zadeh et al. [19] uses sampling based on non-zero elements of columns to avoid exact calculation of all pairs, and accurately estimate cosine similarities using MapReduce. Apache Flink[20] implements the Jaccard Index using similar ideas. Garcia et al. [21] applies a similar strategy as in Flink, but in a ScaleGraph based implementation for common neighbor count. However, the authors note that due to communication and synchronize overhead, distributed system implementations are not more efficient than OpenMP based set intersection implementations and we show that they may only be required when the graph no longer fits in shared memory.

The problem of all pairs common neighbors can also be formulated as a matrix multiplication, that is given the adjacency matrix A, computing A^2 and considering the non-diagonal entries. Gustavson [12] presents an efficient sequential formulation for sparse matrix-matrix multiplication. Numerous parallel implementations [22], [23], [24], [25] have built on this essential approach, with key differences in how accumulation of row results are handled (among other optimizations). Combinatorial BLAS [26], Kokkos [27] and Intel MKL[28] are examples of broader libraries and/or programming models tackling sparse kernels, with SpGEMM being one instance. We note that our graph-centric method for common neighbors can be extended to general SpGEMM, with similarities to existing SpGEMM methodology and vice versa. However, as our focus is the common neighbors problem, our methods were developed independently and optimized accordingly. Our algorithm is able to extend to multiple similarity indices, including Jaccard Indices, Adamic/Adar, and top-k variations that do not have established SpGEMM analogues. We compare with the Intel MKL[28] SpGEMM implementation, as it is a high-performance implementation that's often compared against in other SpGEMM papers. We discuss our methodology and results in Section VI.

Diamond sampling has been proposed to quickly and approximately find the largest common neighbors[29]. This work states that directly computing common neighbors is infeasible for large graphs, however our new approach can be extended to such applications with exact results on massive graphs due to the parallel structure of our algorithm and its suitability for filtering unnecessary computation on only the top-k results. We leave such top-k extensions to future work.

Despite the considerable body of work on common neighbors and similarity indices, there still remains a need for efficient algorithms [30], [31]. This work focuses on computing all vertex pairs' indices on shared memory systems. Our implementations for the all common neighbors problem are $O(\Delta)$ faster than any other shared-memory parallel implementations and are shown to scale to large graphs.

III. Background

A. Formal Notation and Problem Definition

Following is the notation used throughout this paper. Let $G = (V, E)$ be a graph, where V is the vertex set and E is the edge set, with $|V| = n$ and $|E| = m$. The adjacency list of a vertex u is given by Γ_u. The maximum degree of any vertex in the graph is given by $\Delta(G) := \Delta$. A *wedge* is a graph with three vertices and two edges, or a length-2 path. For consistency, we only present algorithms assuming graphs that are undirected. However, our algorithm can easily extend to directed graphs.

Common neighbor counting finds, for each pair of vertices $u, v \in V$, the number of neighbors that u and v have in common. Conceptually, the output is a matrix of size $O(n^2)$ where each u, v-th entry correspond to the number of common neighbors shared by vertices u and v. Importantly, u and v need not necessarily be neighbors themselves. The case where u and v are restricted to being neighbors is the *triangle counting* problem, which has received considerable attention in the literature [32], [33], [34].

More precisely, let the common neighbor count between u and v, denoted by $\mathrm{CN}_{u,v}$, be given by

$$\mathrm{CN}_{u,v} = |\Gamma_u \cap \Gamma_v| .$$

For the remainder of this work, we assume the graph is stored in a sparse matrix format with unsorted adjacency lists, so iterating the neighbor list for a vertex is $O(\Delta)$ and retrieving the degree of a vertex is $O(1)$.

B. Common Neighbor Counting Based on Set Intersection

Most existing work on all pairs common neighbor count are based on set intersections. Common neighbor counting using set intersections proceed in two steps: first identifying possible vertex pairs; then, computing a set intersection with each pair of vertices' respective adjacency lists.

Naively, all n^2 vertex pairs have to be considered in the first step. This is prohibitively expensive and means that common neighbor counting won't scale in most cases. Other work

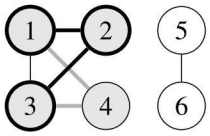

Fig. 1. A simple graph. An optimized set-intersection based approach first finds all neighbors connected by a wedge and then computes pair-wise set intersections between them. Starting at 1, all possible wedges are $123, 143, 132, 134$. We had the insight that discovering each possible wedge is enough to compute the common neighbor counts. A set intersection approach would first find 123, then compute $\Gamma_1 \cap \Gamma_3 = \{2,3,4\} \cap \{1,2,4\}$ to get $\mathrm{CN}_{1,3} = |\{2,4\}| = 2$. Next, 143 would be visited, but since $\mathrm{CN}_{1,3}$ is non-zero no second set intersection would run, resulting in wasted graph iteration work or extraneous set intersection work. As there is no wedge between 1 and 5, no set intersection will be performed for that pair, showing the improvement over the naïve n^2 approach.

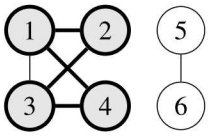

Fig. 2. The same simple graph as Figure 1, and a continuation of common neighbor counting from vertex 1 using wedge iteration. Since both 123 and 143 are iterated, the common neighbor count $\mathrm{CN}_{1,3} = 1 + 1 = 2$, and no asymptotically additional work nor set intersections are required. We have rephrased common neighbor counting from calculating set intersections between each pair of vertices to iterating the graph to find all wedge subgraphs, which is already being performed in order to find non-zero vertex pairs.

have reduced the number of vertex pairs in the first step by only considering vertex pairs that are connected by a wedge, meaning all pairs that share at least one common neighbor are considered. That is, all wedges are iterated through and for each *distinct* resulting pair a set intersection between their adjacency lists is performed. This results in $O(n\Delta^2)$ vertex pairs[10], [11], [2], a significant improvement over $O(n^2)$ pairs. We identify a tighter bound at $O(m\Delta)$ pairs.

In Figure 1, we demonstrate common neighbor counting via set intersection for a simple graph of six vertices $V = \{1, \ldots, 6\}$. To calculate the common neighbor counts starting at vertex 1, efficient prior approaches begin by enumerating the wedges 123, 143, 132, and 134. Then, for each wedge, if the output has not been computed yet (it is zero) a set intersection is performed to get the common neighbor count for that pair. Note that the pair $(1, 3)$ will be visited twice (and importantly the common neighbor count is also 2) but during the second visit no set intersection will be performed. Figure 2 highlights our new, more efficient approach.

The pseudocode for wedge iteration set-intersection based common neighbor counting is given in Algorithm 1. Various optimizations and strategies have been developed for the actual set intersection operation, mostly in the context of triangle counting [35], [36], [37], [38], [39], [40]. $O(\Delta)$ is the best time complexity known for set intersection, therefore the total time complexity for set-intersection based common neighbors is $O(m\Delta^2)$[10], [11], [2].

The memory required for storing the entire output is $O(\min\{m\Delta, n^2\})$. Since we are dealing with sparse graphs, with $m = O(n^2)$, $O(m\Delta)$ memory is needed.

C. Other Indices and Applications

Common neighbor count is the foundation for and can be extended to many other vertex similarity measures. Our approach is able to extend to at least five other common neighbor count based indices: Jaccard Index, Cosine Similarity, Sørenson Index, Resource Allocation, and Adamic/Adar. Such extensions are left to future work. Vertex similarity is a key kernel for applications such as link prediction, graph compression and structure extraction, graph ordering, and community detection[1], [2], [3].

Link prediction, as the name suggests, predicts the existence of "missing" links, due to errors or incomplete data. Numerous approaches have been applied studied for link prediction, including using vertex or edge attributes, graph topology, and machine-learning models. Despite its simplicity, common neighbor counts and its indices perform surprisingly well on many real-world networks and have been shown to beat more sophisticated approaches in link prediction[41], [11].

Graph compression is a process that reduces a graph's memory or disk usage. Navlakha et al. [2] propose a common neighbor based graph compression strategy, which outperforms other contemporary approaches.

Graph ordering finds a permutation of vertex ordering that improves locality of access to vertex data. While this problem is NP-hard[42], many heuristics have been proposed and a greedy common neighbor counting approach[42] provided the highest quality ordering compared to other contemporary approaches.

Common neighbors counting is also used in similarity-based community detection algorithms[43], including to identify hierarchical communities in metabolic networks[3].

IV. Wedge Iteration Common Neighbor Counting

In this section, we introduce our new algorithm for counting common neighbors. This approach is asymptotically faster than set intersection-based methods and remains scalable. It is based on a simple observation: for every wedge, or length-2 path of vertices, the middle vertex in the wedge is a common neighbor of the endpoint vertices.

Our algorithm then iterates over each wedge and increments the common neighbor count for each endpoint pair. For conceptual clarity we present a sequential algorithm, and defer details about parallelism to Section V.

Algorithm 1 Common neighbors counting by intersecting adjacency lists of all vertex pairs in wedges

```
1: procedure WEDGE-SET-INTERSECTION-CN
2:     Initialize CN_{u,v} ← 0, ∀u, v ∈ V
3:     for v ∈ V do
4:         for u ∈ Γ_v do
5:             for w ∈ Γ_u do
6:                 if CN_{v,w} = 0 and v > w then
7:                     CN_{v,w} = |Γ_v ∩ Γ_w|
```

Algorithm 2 Wedge iteration common neighbors counting

```
1: procedure WEDGE-ITERATION-CN
2:     Initialize CN_{u,v} ← 0, ∀u, v ∈ V
3:     for v ∈ V do
4:         for u ∈ Γ_v do
5:             for w ∈ Γ_u do
6:                 if v > w then
7:                     CN_{v,w} ← CN_{v,w} + 1
```

Figure 2 provides an example to illustrate how to take advantage of iterating wedges. Unlike in the set-intersection example in Figure 1, visiting the wedge 143 is not wasted work; instead it is used to increment the common neighbor count between 1 and 3. By doing this, there is no need for a set intersection and no additional graph operations are performed.

It is set intersection free, naturally amenable to sparse output, supports applications such as only computing the top-k results, and is readily parallelizeable.

The pseudo-code can be found in Algorithm 2.

a) Complexity Analysis: By combining loops in lines 3, 4 and 5, we get the number of loops as

$$\Theta\left(\sum_{(v,u)\in E}(|\Gamma_v|+|\Gamma_u|)\right)=\Theta\left(\sum_{v\in V}|\Gamma_v|^2\right)=O(m\Delta).$$

Line 6 and 7 are $O(1)$, and so the total is $O(m\Delta)$.

To understand the memory complexity, it is important to note that this algorithm starts from an endpoint, as opposed to starting from a midpoint. Computing 2-hop neighbors from an endpoint means that only $O(n)$ memory is required to accumulate all possible outputs that involve the given endpoint. The $O(n)$ memory allocated can then also be recycled for latter iterations. As such, the overall memory requirement is $O(n+m\Delta)$ for computation and storage.

b) Algorithm Correctness: We provide a simple argument of correctness.

Proof. We want to show $\mathrm{CN}_{u,v}=|\Gamma_u\cap\Gamma_v|$ is equal to the number of wedges with u and v as endpoints. Then, iterating all wedges will successfully count the common neighbors. Let $u,v\in V$ be distinct. The proof proceeds by definition.

A wedge consists of two edges and three vertices, all distinct. Suppose there are k wedges with u and v as endpoints. Let the midpoints be $x_1,\ldots,x_k$, each distinct. Then, $x_j\in\Gamma_u$ and $x_j\in\Gamma_v$ for $1\le j\le k$, due to each edge in the wedge. So, $\mathrm{CN}_{u,v}\ge k$. Suppose $\mathrm{CN}_{u,v}>k$. Then, there exists some x' such that $x'\in\Gamma_v\cap\Gamma_u$, with u,v,x' all distinct and $\{u,x'\},\{v,x'\}\in E$. This then forms a $k+1$ wedge, a contradiction with there being k wedges. □

V. Implementations and Optimizations

We implemented our algorithm using OpenMP within the GAP[13] framework. We support both sparse output and counting only, and to perform a fair comparison with the wedge iteration set intersection based methods, we implemented an efficient set intersection method as well.

Algorithm 3 Wedge based common neighbors counting with sparse matrix output

```
1: procedure ENDPOINT-CN
2:     ▷ data initialization
3:     parallel-for each thread do
4:         owner[n] ← {0, ...}        ▷ ownership sign
5:         buffer[n] ← {0, ...}            ▷ cn counts
6:         out[n] ← {NULL, ...}     ▷ sparse output
7:         pos[n] ← {0, ...}       ▷ storing position
8:     end parallel-for
9:     parallel-for v ∈ V do
10:        length ← 0
11:        for u ∈ Γ_v do                    ▷ neighbors
12:            for w ∈ Γ_u do       ▷ second neighbors
13:                if v ≤ w then
14:                    continue
15:                o ← owner[w]
16:                if buffer[w] ≠ 0 and o ≠ v then
17:                    ▷ offload data
18:                    out[o][pos[o]] ← (w, buffer[w])
19:                    pos[o] ← pos[o] + 1
20:                    buffer[w] ← 0      ▷ reset buffer
21:                if buffer[w] = 0 then
22:                    owner[w] ← v                ▷ claim
23:                    length ← length + 1
24:                buffer[w] ← buffer[w] + 1      ▷ update
25:        out[v] ← malloc(length)
26:    end parallel-for
27:    ▷ offload remaining data
28:    parallel-for each thread do
29:        for i ∈ [n] do
30:            if buffer[i] ≠ 0 then
31:                o ← owner[i]
32:                out[o][pos[o]] ← (i, buffer[i])
33:                pos[i] ← pos[i] + 1
34:    end parallel-for
```

We treat each vertex in parallel and use dynamic task scheduling and parallel reductions. Each processor allocates a buffer of size n. To provide an efficient implementation, we perform "lazy offloading" of the results. The key idea is to accumulate the results corresponding to a vertex and then add an "owner" to each element in buffer. Instead of resetting the buffer after every vertex's calculation is finished the algorithm moves forward with a dirty buffer. Whenever a buffer element is accessed, we check the owner status: if it is a different owner the data is offloaded, and then after ownership is claimed the value is updated. After all vertices have been processed, a final $O(n)$ pass is made through the buffer to offload any remaining results. The pseudocode for our OpenMP implementation can be found in Algorithm 3.

Most existing shared memory implementations we could find performed all pairs set intersection and stored results as dense output. These implementations are much slower than

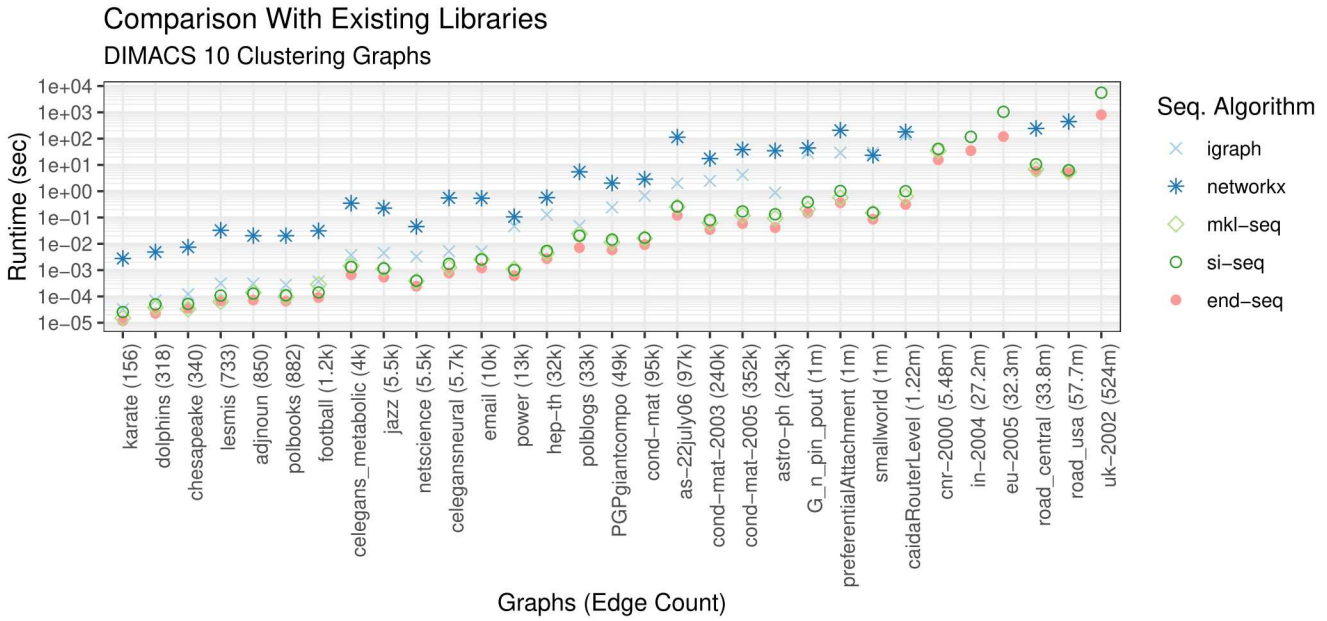

Fig. 3. Performance comparison on small graphs with our wedge iteration algorithm (end) against our set intersection implementation (si) and other libraries. We limit execution to be sequential to compare algorithmically with the inherently sequential algorithms. Our parallel sparse output algorithm runs on the most difficult graph here, `uk_2002`, in under 45 seconds. MKL, igraph, and NetworkX were not able to run successfully, either due to bugs or a time cutoff of two days, on several graphs as noted by missing points. Sequentially we are faster than other approaches including the highly optimized Intel MKL.

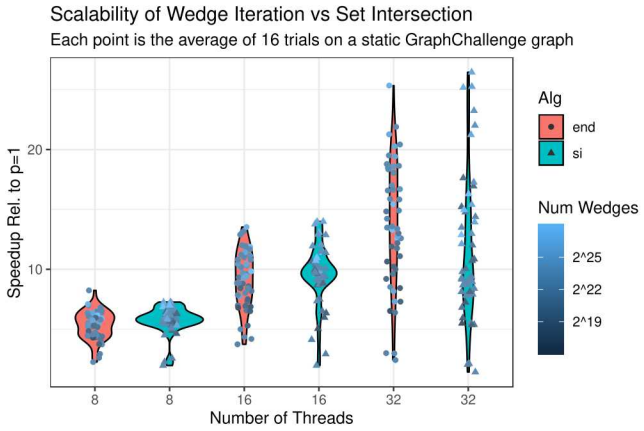

Fig. 4. The number of threads is varied for both our wedge iteration implementation (end) and our set intersection implementation (si). Each point represents the average of 16 trials running common neighbor counting on a GraphChallenge static challenge graph[9]. The first violin plot (red) shows the probability density of the runtime points from our algorithm and the second (green) shows the probability density of the set iteration runtime points, and the raw data points are in the middle. Our algorithm remains similarly scalable, input graph dependent, while providing an asymptotic run-time improvement.

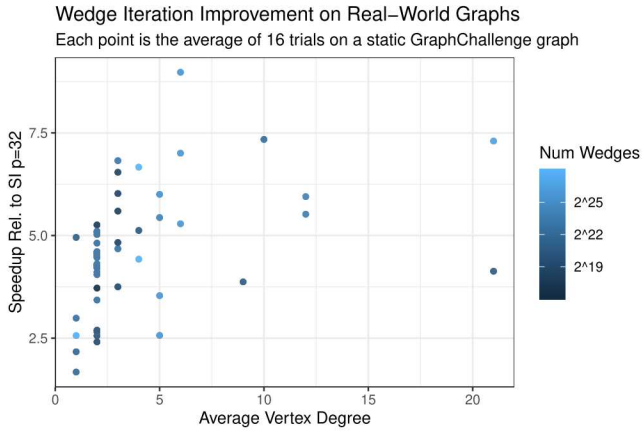

Fig. 5. We show the speedup of wedge iteration over set intersection based methods on the SNAP and other real-world graphs from the static GraphChallenge dataset[9]. In all cases our algorithm is faster.

the wedge iteration set intersection approach (Algorithm 1) and cannot scale to large graphs. Besides comparing with existing libraries, we also implemented Algorithm 1 within our OpenMP system and applied further optimizations to provide a fairer comparison, including using a lookup table for set intersection, memory pre-allocation, dynamic scheduling, and lazy offloading.

VI. Experiments and Results

In this section we describe the experiments we performed to evaluate our algorithm. We compare our algorithm against the shared memory dense midpoint and set intersection algorithms provided by igraph[15] and NetworkX[44] respectively along with the highly optimized Intel MKL[28]. We developed a GAP wrapper for common neighbors using Intel MKL. For our NetworkX evaluation, we implemented a Python implementation of Algorithm 1. For fairness we additionally evaluated against our own set intersection implementation. We ensured correctness by comparing the resulting common

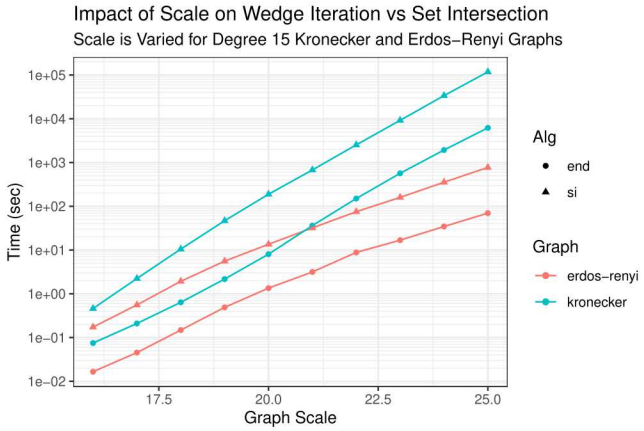

Fig. 6. The scale of the graph is varied. Above scale 25 terabytes of storage are be required to store outputs, and so larger graphs are good candidates for applications such as returning only the top-k results. In both cases our wedge iteration (end) is around an order of magnitude faster than si and as the degree is held constant, the asymptotic difference is not yet apparent.

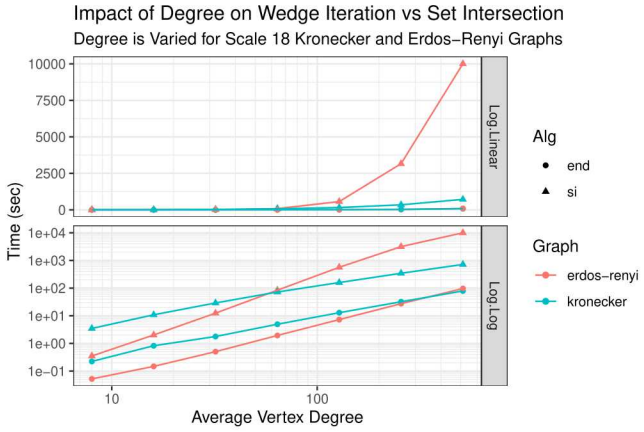

Fig. 7. The graph degree is varied. Note that Kronecker graphs have a skewed degree distribution resulting in more small degree vertices, and so asymptotic improvement is not as significant, showing that the improvement is related to the number of larger degree vertices. The asymptotic improvement can be seen clearly by comparing end and si in the Erdős-Rényi graphs. This further shows that our analysis is not tight to the maximum degree and the asymptotic improvement is in fact related to the degree distribution.

neighbor counts against NetworkX, igraph, and MKL.

Our experiments were run on commodity servers with dual socket Intel Xeon E5-2683 v4 at 2.10GHz processors with 32 cores across four NUMA nodes with 20M L3 cache and 512 GB of RAM running Ubuntu 16.04. In all cases besides the sequential comparisons, we relabeled the input graphs in decreasing order by degree, a common way to balance workload[45], [46].

a) Comparison with Other Libraries: For a fair comparison, we limit our parallel algorithm to run sequentially and evaluate it against other available shared memory libraries, which do not support parallel execution, on small graphs from the DIMACS10 clustering graph challenge[47]. Figure 3 contains the results of this experiment. Our implementations consistently outperform igraph and NetworkX, and as the graph scale increases our *sequential* speedup increases, from 2 to 220. For many of the larger graphs, neither igraph nor NetworkX were able to return any results due to memory constraints. We remain competitive with the highly optimized Intel MKL, beating it with sequential execution in every case except for `eu-2005`. We note that MKL's SpGEMM does not account for symmetry and produces more output pairs than necessary. Similar to other reported problems with MKL [48], we were not able to successfully run it when the number of vertices exceeded one million, and so it did not finish in all instances such as `in-2004`.

b) Evaluating Algorithm Behavior: Figure 4 shows that our scalability remains similar to the older set intersection based method and Figure 5 shows that we have speedup across all static GraphChallenge real-world graphs[9]. To further understand our algorithm's scalability and behavior, we run with different synthetic graphs, both Kronecker[49] with the Graph500 parameters[50] and Erdős-Rényi graphs. These synthetic graphs were chosen as they have different degree distributions and so help understand how tight the $O(\Delta)$ asymptotic improvement is by comparing Δ and the average degree. The results can be found in Figure 6 and Figure 7. As the scale increases, our new algorithm performs progressively better. Furthermore, as the degree increases and the scale remains the same, our algorithm improves, showing the asymptotic improvement with degree. With Kronecker graphs our improvement is slower than with Erdős-Rényi as the degree distribution is skewed.

VII. Conclusion

Graphs are crucial data structures that represent varied data such as biological systems, road maps, and social networks. The analysis of graphs play an increasing role in understanding and reasoning about such data. One fundamental analytic is the number of common neighbors between vertices. This has a wide range of applications, including predicting potential links, uncovering topological modules, and detecting communities. A significant amount of attention has been paid to evaluating the effectiveness of common neighbors, but little attention on computationally scaling common neighbor counting from small graphs to the massive graphs seen today. Until now, there have been no practical solutions for these classes of analytics in shared-memory graph workflows. We identify that set intersections are unnecessary and the process of determining which vertices to consider already produces the common neighbor count outputs.

In our work, we propose a new algorithm that is both asymptotically faster than set intersection methods and parallelizable, unlike prior approaches, and we anticipate effectively extending to top-k applications for multiple indices on massive scale graphs in future work. We implement our algorithm using OpenMP and demonstrate its performance and scalability. We show that set intersections add an unnecessary asymptotic factor when computing all pair common neighbor counts.

REFERENCES

[1] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American society for information science and technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[2] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph summarization with bounded error," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 419–432.

[3] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabási, "Hierarchical organization of modularity in metabolic networks," *Science*, vol. 297, no. 5586, pp. 1551–1555, 2002.

[4] E. A. Leicht, P. Holme, and M. E. Newman, "Vertex similarity in networks," *Physical Review E*, vol. 73, no. 2, p. 026120, 2006.

[5] P. Jaccard, "Nouvelles recherches sur la distribution florale," *Bull. Soc. Vaud. Sci. Nat.*, vol. 44, pp. 223–270, 1908.

[6] T. Zhou, L. Lü, and Y.-C. Zhang, "Predicting missing links via local information," *The European Physical Journal B*, vol. 71, no. 4, 2009.

[7] L. A. Adamic and E. Adar, "Friends and neighbors on the web," *Social networks*, vol. 25, no. 3, pp. 211–230, 2003.

[8] G. Salton, "Automatic text processing: The transformation, analysis, and retrieval of," *Reading: Addison-Wesley*, 1989.

[9] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song *et al.*, "Static graph challenge: Subgraph isomorphism," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[10] G. P. Krawezik, P. M. Kogge, T. J. Dysart, S. K. Kuntz, and J. O. McMahon, "Implementing the jaccard index on the migratory memory-side processing emu architecture," *IEEE Proc. High Performance Extreme Computing (HPEC)*, 2018.

[11] V. Martínez, F. Berzal, and J.-C. Cubero, "A survey of link prediction in complex networks," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 69, 2017.

[12] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

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

[14] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[15] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal, Complex Systems*, vol. 1695, no. 5, pp. 1–9, 2006.

[16] https://sparkling-graph.github.io/, accessed: 2019-01-22.

[17] J. Webber and I. Robinson, *A programmatic introduction to neo4j*. Addison-Wesley Professional, 2018.

[18] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu, "Scalable proximity estimation and link prediction in online social networks," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. ACM, 2009, pp. 322–335.

[19] R. B. Zadeh and G. Carlsson, "Dimension independent matrix square using mapreduce," *arXiv preprint arXiv:1304.1467*, 2013.

[20] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[21] D. Garcia Gasulla, "Link prediction in large directed graphs," 2015.

[22] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 37.

[23] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *International Conference on High Performance Computing*. Springer, 2015, pp. 48–57.

[24] K. Rupp, F. Rudolf, and J. Weinbub, "Viennacl-a high level linear algebra library for gpus and multi-core cpus," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.

[25] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[26] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[27] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[28] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi(tm)*. Springer, 2014, pp. 167–188.

[29] G. Ballard, T. G. Kolda, A. Pinar, and C. Seshadhri, "Diamond sampling for approximate maximum all-pairs dot-product (mad) search," in *International Conference on Data Mining*. IEEE, 2015, pp. 11–20.

[30] https://stackoverflow.com/questions/50739165/link-prediction-for-big-graphs, accessed: 2019-01-31.

[31] https://cstheory.stackexchange.com/questions/16404/finding-two-vertices-with-the-most-least-common-neighbors, accessed: 2019-02-01.

[32] N. Alon, R. Yuster, and U. Zwick, "Finding and counting given length cycles," *Algorithmica*, vol. 17, no. 3, pp. 209–223, 1997.

[33] T. Schank and D. Wagner, "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study," in *Experimental & Efficient Algorithms*. Springer, 2005, pp. 606–609.

[34] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theoretical computer science*, vol. 407, no. 1-3, pp. 458–473, 2008.

[35] H. C. Sungpack Hong, Martin Sevenich, "Finding common neighbors between two nodes in a graph," US Patent 0 178 405.

[36] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[37] M. Bisson and M. Fatica, "High performance exact triangle counting on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3501–3510, 2017.

[38] O. Green, P. Yalamanchili, and L. Munguía, "Fast Triangle Counting on the GPU," in *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[39] O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader, "Logarithmic radix binning and vectorized triangle counting," *IEEE Proc. High Performance Extreme Computing (HPEC)*, 2018.

[40] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader, "Fast and adaptive list intersections on the gpu," *IEEE Proc. High Performance Extreme Computing (HPEC)*, 2018.

[41] P. Sarkar, D. Chakrabarti, and A. W. Moore, "Theoretical justification of popular link prediction heuristics." in *IJCAI proceedings-international joint conference on artificial intelligence*, vol. 22, no. 3, 2011, p. 2722.

[42] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1813–1828.

[43] Y. Pan, D.-H. Li, J.-G. Liu, and J.-Z. Liang, "Detecting community structure in complex networks via node similarity," *Physica A: Statistical Mechanics and its Applications*, vol. 389, no. 14, pp. 2849–2857, 2010.

[44] A. Hagberg, D. Schult, P. Swart, D. Conway, L. Séguin-Charbonneau, C. Ellison, B. Edwards, and J. Torrents, "Networkx. high productivity software for complex networks," *Webová strá nka https://networkx. lanl. gov/wiki*, 2013.

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

[46] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[47] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering," in *10th DIMACS Implementation Challenge Workshop*, 2012.

[48] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 693–702.

[49] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 985–1042, 2010.

[50] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Group (CUG)*, vol. 19, pp. 45–74, 2010.