Improving I/O Complexity of Triangle Enumeration

Yi Cui, Di Xiao, Daren B.H. Cline, and Dmitri Loguinov

***Abstract*—In the age of big data, many graph algorithms are now required to operate in external memory and deliver performance that does not significantly degrade with the scale of the problem. One particular area that frequently deals with graphs larger than RAM is *triangle listing*, where the algorithms must carefully piece together edges from multiple partitions to detect cycles. In recent literature, two competing proposals (i.e., Pagh and PCF) have emerged; however, neither one is universally better than the other. Since little is known about the I/O cost of PCF or how these methods compare to each other, we undertake an investigation into the properties of these algorithms, model their I/O cost, understand their shortcomings, and shed light on the conditions under which each method defeats the other. This insight leads us to develop a novel framework we call Trigon that surpasses the I/O performance of both previous techniques in all graphs and under all RAM conditions.**

I. Introduction

Triangle listing is a field of graph mining that aims to identify all three-node cycles in undirected graphs G. This problem has many applications in theory and practice [2], [3], [4], [5], [8], [11], [26], [35], [36], [39], [42], including areas outside of computer science [14], [15], [17], [23], [25], [34], [38]. Due to the scale of modern graphs (i.e., billions/trillions of edges) and anticipated emergence of even bigger datasets in the future, reducing I/O complexity during graph manipulation has become an important topic.

Triangle listing involves two components – *in-memory search*, whose purpose is to find all relevant motifs (i.e., triangles) within portions of the graph loaded in RAM, and *graph partitioning*, whose responsibility is to chunk G into such pieces that ensure no triangle is missed or discovered more than once. In-memory search entails verification of neighboring relationships between all pairs of candidate nodes. The majority of these solutions [1], [6], [7], [12], [16], [18], [21], [29], [30], [31], [32], [33] can be expressed under the umbrella of 18 vertex/edge-iterator algorithms [9], [40], where a single method E_1 has emerged as a clear winner.

In graph partitioning, however, the situation is more interesting. As of this writing, the two most-efficient approaches to splitting the graph are a coloring scheme called Pagh [27] and the PCF framework from [9]. The main caveat is that the former has lower I/O bounds on complete graphs, while the latter on sparse, i.e., neither one is better than the other. Besides I/O, execution time also depends on the amount of hash-table lookups, which is a function of the partitioning algorithm. This raises a possibility that some methods might exhibit less I/O, but require more CPU cost.

A shorter version of this paper appeared in IEEE ICDM 2017.

All authors are with Texas A&M University, College Station, TX 77843 USA (yicui@cse.tamu.edu, di@cse.tamu.edu, dcline@stat.tamu.edu, dmitri@cse.tamu.edu).

It currently remains unclear under what specific conditions Pagh is better than PCF in terms of I/O, which of them should be chosen for a particular G, why one approach may have inherent advantages over the other, and whether it is possible to design a single algorithm that can perform better than both of these techniques. If so, how does one decide on its parameters in order to achieve the smallest runtime? Our goal in the paper is to address these questions.

A. Overview of Results

We start by analyzing the asymptotics of I/O in Pagh and PCF, aiming to achieve an understanding of their strengths and weaknesses. While the former has a simple model, the overhead of the latter is a complex function of the acyclic orientation θ, the resulting directed graph G_θ, and specific traversal order of nodes in each triangle. We derive the exact overhead of PCF; however, this formula proves difficult for closed-form analysis. We therefore obtain tight upper/lower bounds on its growth rate, which are then used in the comparison against Pagh.

This analysis shows how the scaling rate of average degree, memory size, and variance of out-degree affect which method is better. In general, PCF has the highest advantage when the graph is sparse, the variance of out-degree is small, and RAM is growing slowly with the number of edges m. Pagh wins when these conditions are reversed. As the number of nodes $n \to \infty$, our results demonstrate that under the best scenario for PCF, it beats Pagh by a factor of n. In the worst case, it loses by a factor of $\sqrt{n}$. We also prove existence of graphs where PCF scales I/O no faster than Pagh for *all* memory sizes; however, the opposite is possible as well.

Our investigation reveals that each method brings a significant amount of redundant edges into RAM, but *they do so under different conditions*. This gives hope that a single method can combine the strengths of these techniques and simultaneously avoid their individual drawbacks. To this end, we first generalize graph partitioning to cover all possible ways to execute vertex/edge iterators in external memory. Not surprisingly, both Pagh and PCF, as well as previous techniques based on MGT [12], [16], are all special cases of this unifying framework. Under its umbrella, we then create a particular scheme, which we call *Trigon*, that leverages the lessons learned from the preceding analysis. We show that Trigon's I/O is never worse, and in many cases much better, than either of its predecessors. Not only that, but it is also the first method that allows balancing between I/O and CPU cost in order to achieve the smallest runtime.

II. Related Work

The issue of optimal speed for in-memory algorithms appears to be settled. In the last decade, the fastest techniques

have come from the family of vertex/edge iterators [1], [12], [16], [18], [21], [30], [31], [32], [33]. The former methods typically rely on hash tables to perform neighbor checks, where the speed is limited by that of random memory access. On the other hand, scanning edge iterators can be implemented using vectorized CPU intrinsics, which are not bottlenecked by RAM latency. With 128-bit SIMD and list compression, it is feasible to achieve two orders of magnitude faster neighbor verification [9]. In the taxonomy of 18 vertex/edge iterators, method E_1 [1], [12], [32] is by far the best technique [9], [40].

In external memory, early methods used a variety of techniques, including disk seeking [10], [24], MapReduce [8], [28], [33], general graph libraries [13], [20], and iterative graph shrinkage [7], which are difficult to summarize here analytically. Due to their low efficiency, however, these approaches are not considered competitive today. In more recent development, algorithms have been streamlined to access the disk only sequentially, allowing comparison using just the amount of edges read from disk and RAM size M. In MGT [16], the graph is split into equal-size chunks. After each is loaded into RAM, the graph is scanned again to discover the missing edges that complete triangles with the portion already in RAM. Ignoring small terms, MGT reads m^2/M edges.

This result was superseded by a method we call Pagh [27], which achieves a strictly better asymptotic bound $O(m^{1.5}/\sqrt{M})$. We review its operation in more detail below. A different approach is proposed in [9], where a set of six PCF algorithms covers all 18 vertex/edge iterators in external memory. While there is no model for PCF I/O, upper bounds show that it is currently the only method that can achieve linear complexity under constant M.

III. Preliminaries

Assume a simple undirected graph $G = (V, E)$ with n nodes and m edges. Detection of triangles requires a large number of neighbor checks, whose complexity is normally a quadratic function of undirected degree. This overhead can be substantially reduced by performing an acyclic orientation on G, which makes cost depend on the much-smaller *directed* degree. In recent literature [40], orientation is modeled as some permutation θ that decides the direction of each edge. Specifically, each node u is placed into a new location $\theta(u)$, the permuted sequence of nodes is relabeled from 1 to n, and all edges are directed from larger to smaller node IDs. This splits each neighbor list N_u into out-neighbors N_u^+ and in-neighbors N_u^-, with the corresponding graphs G_θ^+ and G_θ^-. Note that adjacency lists are sorted by the new node label.

Throughout the paper, we use orientation θ_D that arranges the nodes in *descending* order of undirected degree d_u. This permutation, also known as *largest-first* in graph theory [22], [37], is optimal for both the fastest edge iterator E_1 and its corresponding PCF algorithms in [9], [40]. Since Pagh's performance is independent of θ, this choice does not affect its I/O. Letting $Y_u = |N_u^-|$ and $X_u = |N_u^+|$ be the respective in/out-degrees of u in directed G_θ, it follows that $X_u + Y_u = d_u$ and $\sum_{u=1}^n X_u = \sum_{u=1}^n Y_u = m$.

After orientation, E_1 searches for all directed triangles $\triangle_{uvw}$, where $u > v > w$. This is done by calling Algorithm 1 for each source node u in G_θ^+. The CPU cost consists of the number of hash-table lookups to retrieve N_v^+ and the size of intersection in Line 3. For in-memory operation, the former is just $\gamma(n) = m - n$, while the latter is given by [40]

$$\rho(n) = \sum_{u=1}^{n} \Big(\frac{X_u(X_u - 1)}{2} + X_u Y_u \Big). \tag{1}$$

Algorithm 1: Method E_1 processing source node u in memory

```
1 foreach v ∈ N_u^+ do ◁ visit all out-neighbors
2     find N_v^+ using a hash table
3     W = Intersect(N_u^+, N_v^+) ◁ intersect two sorted out-lists
4     foreach w ∈ W do report Δ_uvw
```

To emphasized the importance of keeping track of lookups, consider the example of Twitter [19]. With $m = 1.2$B edges, Algorithm 1 requires 60 seconds worth of lookups using one core of an Intel i7. The time to perform 511B intersections is an additional 250 seconds. Increasing the lookup cost just 5 times shifts the bottleneck to the hash table and causes triangle search to increase the runtime from 310 to 550 seconds.

When E_1 is used in external memory, the partitioning scheme must ensure that all three edges of a triangle are eventually present in RAM *at the same time*. This can be accomplished by holding one of them in RAM and streaming the other two from disk (e.g., MGT [16], PCF-1B [9]), keeping two in RAM and streaming the third one (e.g., PCF-1A [9]), or loading all three simultaneously [27], [29]. Because of the random lookup needed to obtain N_v^+, it does not currently appear feasible to stream all three edges.

Note that all methods require the same amount of I/O to store the found triangles. We therefore focus on the cost needed to create this list, which is what differentiates the various approaches.

IV. Analysis of Pagh

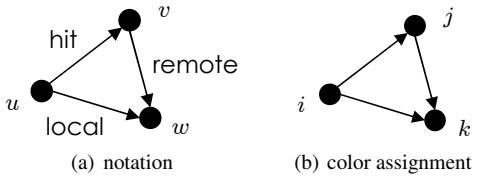
The original Pagh algorithm [27] has certain details omitted from the paper, while others are sketched at a high level. While I/O complexity of this method has known bounds in the $O(.)$ notation [27], numerical comparison between the algorithms, as well as implementation, both require the missing constants. Additionally, since coupling of Pagh to E_1 has not been discussed before, we perform this extension as well.

A. Algorithm

Pagh assigns to each node u a uniformly random color ϕ_u drawn from a set $1, 2, \ldots, c$, where $c = \sqrt{m/M}$ and M is RAM size in edges. Then, all nodes are split into c subsets $V_1, \ldots, V_c$ such that

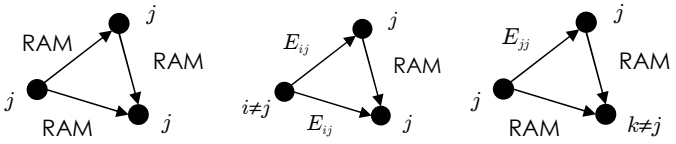
$$V_i = \{u \in V \mid \phi_u = i\}. \tag{2}$$

This can be visualized with the help of Fig. 1. Part (a) shows a directed triangle (uvw), as seen by Algorithm 1, with three uniquely-identifiable edges. While they have several different names in previous literature, we follow the notation of [9] for compatibility with E_1.From u's perspective, edge (uv) results in a *hit* on the hash table, (uw) participates in *local*



(a) notation (b) color assignment

Fig. 1. Directed triangle ($u > v > w$).



(a) case 1 (b) case 2 (c) case 3

Fig. 2. Special cases in Pagh.

Algorithm 2: Graph partitioning in Pagh

```
for u = 1 to n do
    i = φ_u  ◁ color of source node
    for j = 1 to c do ◁ color of destination nodes
        N⁺_uj = N⁺_u ∩ V_j ◁ out-neighbors of color j
        write (u, N⁺_uj) to subgraph E⁺_ij
```

intersection at u, and (vw) is part of *remote* intersection. The mapping to colors is shown in part (b) of the figure, where i refers to the color of the largest node, j to that of the middle, and k to that of the smallest.

The edges of $G^+_\theta = (V, E^+_\theta)$ are partitioned into c^2 subsets $\{E^+_{ij}\}$ according to the color of source/destination nodes, i.e.,

$$E^+_{ij} = \{(u, v) \in E^+_\theta \mid \phi_u = i, \phi_v = j\}. \tag{3}$$

This is demonstrated in Algorithm 2, which splits the out-graph into tuples (u, N^+_{uj}), where N^+_{uj} contains u's out-neighbors of color j. Note that the expected size of each V_i is n/c and that of E^+_{ij} is m/c^2 edges. After this preprocessing step, Pagh suggests using MGT [16] to find triangles in each of the c^3 triples $(E^+_{ij}, E^+_{jk}, E^+_{ik})$, where the remote edge belongs to E^+_{jk}. MGT relies on vertex iterator T_1 [9], which is $15 - 80$ times slower than E_1 on real graphs. Additionally, it does not by default handle heterogeneous partitions (i.e., hit/remote/local edges all being stored separately). To create a fully working system, we need a few refinements.

B. Pagh+

Assuming partitions are well-balanced, i.e., all have size M within some tolerance, MGT can be combined with E_1 to efficiently solve the problem. Algorithm 3, which we call Pagh+, loads remote edges E^+_{jk} into RAM and then scans the other two subgraphs. Since Algorithm 2 writes source nodes in the same order for all subgraphs, Pagh+ can obtain both hit and local lists of each u by concurrently reading E^+_{ij} and E^+_{ik}. The resulting system detects each triangle once and performs no more intersections than in-memory E_1. Note that we skipped discussing cases when some of the colors are duplicate; however, our implementation handles them efficiently (i.e., without reading unnecessary files).

Theorem 1. *Pagh+ needs $I_P(n) = (2c - 1)m$ edges of I/O.*

Proof: First notice that Algorithm 3 loads each remote subgraph once, for a total I/O cost of m. The remaining overhead comes from hit/local edges, which we consider next. While there are c^3 possible triples (ijk), there are three special cases. The first one is shown in Fig. 2(a), where all three edges are in RAM. This results in no additional cost beyond E^+_{jj}.

Algorithm 3: Pagh+ handling one remote graph

```
load E⁺_jk = {(v, N⁺_vk)} in RAM; set up hash table to source nodes
for i = 1 to c do
    while file E⁺_ij not empty do
        load (u, N⁺_uj) from E⁺_ij and (u, N⁺_uk) from E⁺_ik
        foreach v ∈ N⁺_uj do ◁ visit all neighbors in the hist list
            find remote list N⁺_vk using the hash table
            W = Intersect(N⁺_uk, N⁺_vk) ◁ local/remote lists
            foreach w ∈ W do report Δ_uvw
```

The second configuration in Fig. 2(b) has file E^+_{jj} loaded in RAM and the remaining color i is not equal to j. There are $c(c-1)$ such cases, each requiring $|E^+_{ij}|$ I/O. The last special case in Fig. 2(c) involves $c(c-1)$ files E^+_{jk}, each producing $|E^+_{jj}|$ I/O. The remaining scenarios are outside the scope of Fig. 2. There are $c(c-1)$ files E^+_{jk} such that $j \neq k$, each of which can be coupled with $c - 1$ values of $i \neq j$. This yields $c(c-1)(c-1)$ cases that load $2m/c^2$ edges each.

Combining the various terms, we get

$$m + \frac{m}{c^2}(0 + 2c(c-1) + 2c(c-1)(c-1)), \tag{4}$$

which simplifies to $2cm - m = (2c-1)m$. ∎

Since $(2c-1)m = 2m^{1.5}/\sqrt{M} - m$, Pagh+ has the best multiplicative constant in the literature. The closest alternative [29] uses the undirected graph G, assigns direction to *colors* rather than edges, and increases c to $\sqrt{5m/M}$ such that certain combinations of subgraphs fit in RAM. This leads to $\sqrt{5}m^{1.5}/\sqrt{M} \approx 2.2m^{1.5}/\sqrt{M}$ total I/O, which is worse than the result above. Another potential drawback to this approach is usage of undirected graphs, where E_1 has to perform unnecessary intersections [40].

It is also simple to obtain the number of hash-table lookups in Pagh+. When they become a CPU bottleneck, E_1 may essentially deteriorate into T_1 and lose its advantages. The next result shows that this value is linear in c.

Theorem 2. *Pagh+ performs $\gamma_P(n) = cm$ lookups.*

Proof: Denote by $X_{uj} = |N^+_{uj}|$ the out-degree of u with respect to neighbors of color j. Now, notice that the size of hit lists processed by Algorithm 3 for all (j, k) equals

$$\sum_{k=1}^{c}\sum_{j=1}^{c}\sum_{i=1}^{c}\Big(\sum_{u=1}^{n}\mathbf{1}_{\phi_u=i}\,X_{uj}\Big), \tag{5}$$

where $\mathbf{1}_A$ is an indicator of event A. Swapping the order of summations, this becomes

$$\sum_{k=1}^{c}\sum_{j=1}^{c}\sum_{u=1}^{n}\Big(\sum_{i=1}^{c}\mathbf{1}_{\phi_u=i}\,X_{uj}\Big) = \sum_{k=1}^{c}\sum_{j=1}^{c}\sum_{u=1}^{n}X_{uj}$$

Algorithm 4: PCF-1A graph partitioning

```
for u = 1 to n do ◁ iterate over all nodes
    for i = 1 to p do ◁ go through each partition
        H_ui = N_u^+ ∩ [a_{i+1}, n]  ◁ pruned hit list
        L_ui = N_u^+ ∩ [a_i, a_{i+1})  ◁ local/remote list
        if L_ui ≠ ∅ then
            write (u, L_ui) to G_θ^r(i)  ◁ remote file i
            if H_ui ≠ ∅ then
                write (u, H_ui) to G_θ^c(i)  ◁ companion file i
```

Algorithm 5: PCF-1B graph partitioning

```
for u = 1 to n do ◁ iterate over all nodes
    for i = 1 to φ_u − 1 do ◁ go through each partition below u
        H_ui = N_u^+ ∩ [a_i, a_{i+1})  ◁ hit list
        L_ui = N_u^+ ∩ [1, a_{i+1})  ◁ local list
        if H_ui ≠ ∅ and |L_ui| ≥ 2 then
            write (u, L_ui) to G_θ^c(i)  ◁ save to companion file i
    if N_u^+ ≠ ∅ then ◁ out-degree non-zero?
        write (u, N_u^+) to G_θ^r(φ_u)  ◁ remote file of u's color
```

$$= \sum_{k=1}^{c} \sum_{u=1}^{n} X_u. \tag{6}$$

Leveraging the fact that $\sum_{u=1}^{n} X_u = m$, we get the statement of the theorem. ∎

C. Discussion

Slightly unbalanced partition sizes $|E_{ij}^+|$ due to randomness of color assignment are a minor issue in practice. However, when the graph contains nodes with large degree, Pagh requires a different algorithm. One example is the star graph, where all nodes connect to a center node of some color k. To avoid optimizations that discard (ijk) if any of the subgraphs is empty, the star graph can be augmented with c^2 random edges between the leaf nodes. Neglecting small terms, Algorithm 2 produces c partitions of size $m/c \gg M$. In fact, two of the three subgraphs involving color k have size m/c. Pagh+ cannot be applied here, but MGT can be modified to handle any triple (ijk) with I/O complexity $2(m/c)^2/M = 2m$. Repeating this c^2 times for all (ij) produces a total of $2m^2/M$. Depending on m and M, this result can be significantly worse than in Theorem 1.

Pagh [27] handles this case by isolating nodes of degree larger than $\sqrt{mM}$ into a separate category. Each of them requires sorting up to m edges on disk. Since there are no more than c such nodes, the I/O can be bounded by $c \cdot \text{sort}(m) \sim cm \log m / \log M$ edges. If RAM scales as some power of m, as assumed in [27], we get the usual $O(m^{1.5}/\sqrt{M})$; however, the hidden constants may be non-negligible. But more importantly, the CPU cost for sorting the graph c times may be quite hefty.

On the bright side, Pagh does not impose much restriction on minimum RAM or disk size. Setting $c = n$, it is possible to create subgraphs that contain just one edge each, resulting in $O(1)$ memory consumption. Furthermore, its disk-space requirement is only m edges. However, when c^3 is large, Pagh has to read many small files and its I/O speed may be adversely affected by disk seeking.

V. Analysis of PCF

The I/O complexity of PCF is quite peculiar due to the dependency on the underlying graph. This section develops the methodology and insight that not only sheds light on PCF, but also helps later with comparison against Pagh+ and design of our new method.

A. Operation

PCF [9] is a suite of six algorithms 1A, 1B, 2A, 2B, 6A, 6B. All of them partition the graph along the remote edge of the corresponding in-memory algorithm (i.e., E_1, E_2, and E_6). In the notation of Fig. 1(a), these are (vw) for 1A/1B, (uw) for 2A/2B, and (uv) for 6A/6B. The A variants split based on the destination node of the remote edge, while the B versions do the same on the source node. After preprocessing, PCF sequentially loads chunks of G_θ^+ in RAM and scans so-called *pruned companion files* to obtain the missing edges.

Method E_1 requires PCF-1A/1B, which we review and analyze next. Both of them start by dividing the set of nodes V into $p = m/M$ non-overlapping subsets $V_1, \ldots, V_p$. PCF utilizes *sequential* partitions such that $u \in V_i$ iff $u \in [a_i, a_{i+1})$, where boundaries $\{a_i\}$ are determined by load-balancing either the in-degree (1A) or out-degree (1B) of each partition to equal memory size M. To be consistent with other parts of the paper, we say that nodes in V_i have color i. We also use the same function ϕ_u to map u to its color.

File G_θ^+ is split into p disjoint subgraphs $G_\theta^r(1), \ldots, G_\theta^r(p)$ that contain all remote edges (vw) matching the corresponding color. Specifically, (vw) is written into $G_\theta^r(i)$ iff $w \in V_i$ in PCF-1A and $v \in V_i$ in PCF-1B. The corresponding companion files $G_\theta^c(i)$ contain nodes u and their hit/local lists, but only if they are relevant to partition i. For example, PCF-1B skips node u unless it has at least one neighbor of color i and another neighbor with a smaller ID. While [9] has a comprehensive algorithm that covers all six methods, it may be difficult to parse. We therefore find it useful to show the minimal versions of PCF-1A and 1B using Algorithms 4-5.

B. Model

Since $\sum_{i=1}^{p} |G_\theta^r(i)| = m$ is fixed, the main open question is companion I/O, i.e., $\sum_{i=1}^{p} |G_\theta^c(i)|$. For a source node u, suppose ϕ_{us} is the color of its s-th out-neighbor in sorted order. For a given list N_u^+, denote by R_{us} the number of colors to the left of position s, excluding the color of s, and by R'_{us} the number to the right, but not counting u's own color

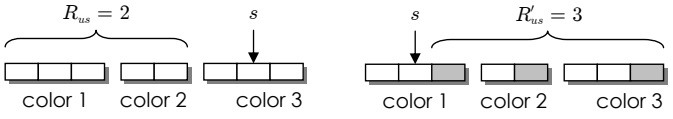
$$R_{us} = |\{\phi_{ut} \mid t < s, \phi_{ut} \neq \phi_{us}\}|, \tag{7}$$

$$R'_{us} = |\{\phi_{ut} \mid t > s, \phi_{ut} \neq \phi_u\}|. \tag{8}$$

With this in mind, consider the next result.

Theorem 3. *The companion I/O of PCF-1A is given by*

$$I_A(n) = \sum_{u=1}^{n} \sum_{s=1}^{X_u} R_{us} \tag{9}$$



(a) PCF-A (b) PCF-B

Fig. 3. Colors among N_u^+ in PCF.

and that of PCF-1B by

$$I_B(n) = \sum_{u=1}^{n} \Big[R'_{u1} + \sum_{s=1}^{X_u} R'_{us} \Big]. \tag{10}$$

Proof: In PCF-1A, consider some source node u and color i. As long as the local list $L_{ui} = N_u^+ \cap V_i \neq \emptyset$, all out-neighbors with labels at least a_{i+1} are saved to disk in Algorithm 4. Therefore, from a perspective of some fixed position $s \in [1, X_u]$ in the out-list N_u^+, the number of times this node is written to disk equals the number of non-empty local lists in positions $[1, s-1]$, excluding those that contain s. An example is shown in Fig. 3(a), where s is written twice. This happens to be the number of distinct colors, except ϕ_{us}, among the nodes preceding s, which equals R_{us} in (7). Taking a summation over all u and s yields (9).

For PCF-1B, we first have to remove neighbors of color ϕ_u from consideration since these edges are found in RAM (i.e., included in the remote graph). Once this is done, notice that Algorithm 5 writes a node in position s into R'_{us} files as part of some local list. Fig. 3(b) shows one such example. However, there is one exception for $s \geq 2$. The *last* node of each color (within u's neighbor list) has overhead $R'_{us} + 1$, where the extra 1 accounts for s being included in the hit list of companion file $G^r_\theta(\phi_{us})$. The affected neighbors are shown in Fig. 3 using shading. Putting the pieces together,

$$I_B(n) = \sum_{u=1}^{n} \Big[R'_{u1} + \sum_{s=2}^{X_u} \Big(R'_{us} + \mathbf{1}_{\phi_{u,s+1} \neq \phi_{us}} \Big) \Big], \tag{11}$$

where condition $\phi_{u,X_u+1} \neq \phi_{u,X_u}$ is always true (i.e., we always count an extra 1 for the very last node in N_u^+). Rearranging the terms, we get

$$I_B(n) = \sum_{u=1}^{n} \Big[\sum_{s=1}^{X_u} R'_{us} + \sum_{s=2}^{X_u} \mathbf{1}_{\phi_{u,s+1} \neq \phi_{us}} \Big]. \tag{12}$$

Now notice that the sum of indicator variables yields the number of unique colors in positions $[2, X_u]$. Since this value is R'_{u1}, we obtain (10). ∎

Note that (9)-(10) are exact. While R_{us} and R'_{us} appear symmetric to each other, there is a subtle difference. PCF-1A load-balances using in-degree, while PCF-1B using out-degree. Hence, their color assignments are not directly comparable to each other. However, on real graphs, PCF-1B commonly demands less I/O [9]. Additionally, it requires a lot fewer lookups. For the next result that shows this, define $R_u = R_{u,X_u} + 1$ to be number of colors in N_u^+.

Theorem 4. *The number of hash-table hits in PCF-1A is*

$$\gamma_A(n) = I_A(n) + m - \sum_{u=1}^{n} R_u, \tag{13}$$

and that in PCF-1B is

$$\gamma_B(n) = m - n. \tag{14}$$

Proof: PCF-1A writes only pruned hit-lists, which produce $I_A(n)$ lookups when they are loaded back to RAM. Additionally, a portion of each hit list is removed by Algorithm 4 and kept in RAM as part of the local list L_{ui}. In fact, the entire L_{ui}, except its first node, is part of the hit list for node u. Adding the two terms together yields (13).

For PCF-1B, every node in N_u^+ is part of the hit list, except the one in position $s = 1$. Writing

$$\gamma_B(n) = \sum_{u=1}^{n} (X_u - 1), \tag{15}$$

we immediately get (14). ∎

Note that $\gamma_A(n)$ can be orders of magnitude larger than m, while $\gamma_B(n)$ is always optimal (i.e., the same as in-memory E_1). Further problems of PCF-1A include a requirement that RAM size be no smaller than the largest in-degree $\max_u Y_u$, which can be as large as $n-1$. In contrast, PCF-1B only needs $M \geq \max_u X_u$, whose largest value under descending-degree permutation θ_D stays bounded by $\sqrt{2m}$. While PCF-1A can be dismissed for now as being inferior, we later come back to it and explain what features the new method shares with it.

C. Bounds

Computing the exact I/O formula (10) requires processing the entire G_θ^+ and splitting all m edges into colors. In certain cases, this may be too expensive, especially if repeated many times (e.g., in an iterative search for optimal parameters). To overcome this issue, we derive simple upper bounds that require one pass over the out-degree sequence $\{X_u\}$.

Theorem 5. *For a given out-degree sequence $\{X_u\}$, the expected size of companion I/O in PCF-1B is bounded by*

$$E[I_B(n)] \leq \sum_{u=a_2}^{n} \zeta_u \Big[X_u - \zeta_u + 1 + (\zeta_u - 2) q_u^{X_u - 1} \Big], \tag{16}$$

where $q_u = 1 - 1/\zeta_u$ and $\zeta_u = \phi_u - 1$.

Proof: Note that uniformly random, rather than sequential, color assignment can only make R'_{us} stochastically larger. Therefore, replacing R'_{us} with some other variable Q_{us} that uniformly draws from among ζ_u colors can yield only larger I/O in expectation. Since $Q_{us} = 0$ for $u < a_2$, we get

$$E[I_B(n)] \leq \sum_{u=a_2}^{n} \Big(E[Q_{u1}] + \sum_{s=1}^{X_u} E[Q_{us}] \Big). \tag{17}$$

To expand this, continue assuming random color choices and define

$$W_{usi} = \sum_{t=s+1}^{X_u} \mathbf{1}_{\phi_{ut} = i} \tag{18}$$

to be the number of u's out-neighbors to the right of s that have color i. Conditioning on the out-degree sequence, each

W_{usi} is Binomial$(X_u - s, 1/\zeta_u)$, where $E[W_{usi}] = X_u/\zeta_u$ and $P(W_{usi} \geq 1) = 1 - (1 - 1/\zeta_u)^{X_u - s}$. Then,

$$Q_{us} = \sum_{i=1}^{\zeta_u} \mathbf{1}_{W_{usi} \geq 1} \tag{19}$$

is the number of uniform colors to the right of s. Setting $q_u = 1 - 1/\zeta_u$, we get

$$E[Q_{us}] = \zeta_u P(W_{usi} \geq 1) = \zeta_u (1 - q_u^{X_u - s}). \tag{20}$$

Next, observe that

$$\begin{aligned}\sum_{s=1}^{X_u} E[Q_{us}] &= \zeta_u \sum_{s=1}^{X_u} (1 - q_u^{X_u - s}) = \zeta_u \Big(X_u - \sum_{s=0}^{X_u - 1} q_u^s \Big) \\ &= \zeta_u \Big(X_u - \frac{1 - q_u^{X_u}}{1 - q_u} \Big) \\ &= \zeta_u \Big[X_u - \zeta_u (1 - q_u^{X_u}) \Big]. \end{aligned} \tag{21}$$

Adding $E[Q_{u1}]$ to the last result, we get

$$\begin{aligned} E[I_B(n)] &\leq \sum_{u=a_2}^{n} \zeta_u \Big[X_u - \zeta_u + \zeta_u q_u^{X_u} + 1 - q_u^{X_u - 1} \Big] \\ &\leq \sum_{u=a_2}^{n} \zeta_u \Big[X_u - \zeta_u + 1 + (\zeta_u - 2) q_u^{X_u - 1} \Big], \end{aligned} \tag{22}$$

where we use the fact that $\zeta_u q_u = (\zeta_u - 1)$. ∎

Bound (16) holds in expectation; however, there are adversarial graphs and color assignments that may violate it. Therefore, our second bound is deterministic, but somewhat looser in sparse graphs. It shows a more clear dependency of I/O on the second moment of out-degree.

Theorem 6. *The companion I/O of PCF-1B is bounded by*

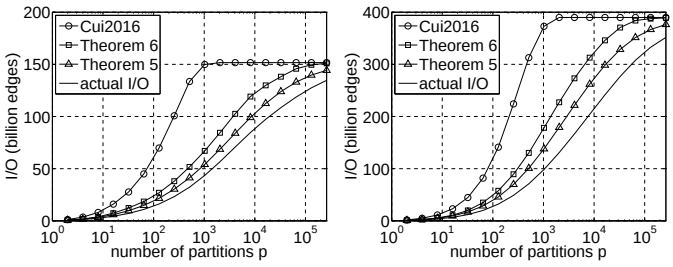
$$I_B(n) \leq \sum_{u=1}^{n} \min\Big(\frac{(X_u - 1)(X_u + 2)}{2}, X_u \zeta_u \Big). \tag{23}$$

Proof: Trivially, $R'_{us} \leq \min(X_u - j, \zeta_u)$. Thus, we get

$$\begin{aligned} I_B(n) &\leq \sum_{u=1}^{n} \Big[\min(X_u - 1, \zeta_u) + \sum_{s=1}^{X_u} \min(X_u - s, \zeta_u) \Big] \\ &= \sum_{u=1}^{n} \Big[\min(X_u - 1, \zeta_u) + \sum_{j=1}^{X_u - 1} \min(s, \zeta_u) \Big] \\ &\leq \sum_{u=1}^{n} \min\Big(X_u - 1 + \sum_{s=1}^{X_u - 1} s, X_u \zeta_u \Big), \end{aligned} \tag{24}$$

which becomes (23) after expanding the inner sum. ∎

Note that [9] also obtains an upper-bound on $I_B(n)$; however, they neglect the standalone term R'_{u1} in (10). This issue notwithstanding, their bound is a special case of (23) where $\zeta_u = \phi_u - 1$ is replaced by $p - 1$. Fig. 4 shows a comparison between that result and our models, where we use Twitter from [19] and IRL-domain from the authors of [9].



(a) Twitter ($m = 1.2$B) (b) IRL-domain ($m = 1.7$B)

Fig. 4. Model accuracy in PCF-1B.

D. Discussion

PCF-1B requires that the longest out-list fit in memory, i.e., $M \geq \max_u X_u$. While much better than in PCF-1A, this condition is stricter than in Pagh, which can work with constant M as $n \to \infty$. Additionally, PCF-1B needs enough disk space to write all companion files. In some cases, the read-only operation of Pagh may be preferable. Furthermore, it is common to exclude the preprocessing stage from comparison, because triangle enumeration can run multiple times over the same input (e.g., feeding the found Δ_{uwv} to different consumers on the fly). However, if this is not the case, all I/O of PCF-1B needs to be doubled. This is of no consequence to asymptotics, but we benchmark both stages separately in the experimental section.

On the positive side, PCF achieves deterministic load-balancing and its sequential color assignment brings many benefits compared to random colors in Pagh. First, contiguous coloring produces stochastically smaller R'_{us} because u's neighbors are more drawn towards colors with a large mass of degree. Since such colors are concentrated at the start of the range $[1, n]$, neighbor lists contain more duplicate colors than would be possible under uniform assignment. This effect is most pronounced on graphs with heavy-tailed degree. Second, due to sequential grouping of nodes into each color, splitting of neighbor lists in Algorithm 5 does not require a hash-table lookup for each edge. Similarly, when PCF-1B loads the remote graph into RAM, it can use an array of offsets instead of a hash table to perform retrieval of remote edges. Third, placing similar node IDs into individual partitions allows better compression of neighbor lists. This can save up to 50% on byte I/O. Similarly, [9] shows that SIMD intersection is 80% faster on compressed lists.

VI. Asymptotic Comparison

We are now interested in the conditions that cause each of the candidate methods to be better than the other. Deciding this for finite n requires a specific graph and computation of the various models/bounds from the previous section. Instead, we study cases of $n \to \infty$, which should provide a qualitative assessment of each method's capabilities and types of graphs they are most suited for.

A. Definitions

Suppose the average directed degree of the graph, i.e., m/n, grows proportionally to n^a, where $a \in [0, 1]$ is a constant. In

general, we write $f \sim g$ to mean that $f(n) = O(g(n))$ and $g(n) = O(f(n))$. Similarly, assume memory size $M_n \sim n^r$, where $r \in [0, 1+a]$ is also fixed. To ignore contribution from constants and slowly growing terms, we have the following definition.

Definition 1. *The scaling rate of a function $f(n)$ is given by*

$$\omega(f) = \lim_{n\to\infty} \log_n f(n), \tag{25}$$

as long as the limit exists and is finite.

For example, $f(n) = 5n^{2.3}/\log(n)$ has $\omega(f) = 2.3$. Since the scaling rate of m is $1+a$, Pagh has a very simple result

$$\omega(I_P) = \frac{3(1+a) - r}{2}. \tag{26}$$

However, the corresponding model for PCF is less obvious. We therefore perform a separate investigation into it.

B. Dynamics of PCF

We start with an upper bound on $\omega(I_B)$, which requires studying the second moment of out-degree. To this end, define

$$\pi_n = \sum_{u=1}^{n} X_u^2 \tag{27}$$

and consider the next result.

Theorem 7. *The scaling rate of* (27) *is $\omega(\pi) = 1 + 2a + \epsilon$, where $\epsilon \in [0, (1-a)/2]$.*

Proof: Suppose $\pi_n \sim n^{1+2a+\epsilon_n}$, where ϵ_n is some unknown function. Our goal is to put bounds on it. Assuming $E[X_u] = m/n$ is fixed, it is obvious that minimizing the variance of set $X_1, \ldots, X_n$ yields the lowest π_n. Since this is achieved by constant $X_u = m/n$, we get

$$\pi_n \geq n\frac{m^2}{n^2} \sim n^{1+2a}. \tag{28}$$

This shows that $\epsilon_n \geq 0$ must hold. To arrive at the upper bound on π_n, first notice that X_u cannot exceed the number of nodes preceding it (i.e., $u-1$). At the same time, X_u must be no larger than $2m/u$; otherwise, the degree sum $\sum_{v=1}^{u} d_v$ of the largest u nodes would exceed $2m$, which is impossible. As a result,

$$\begin{aligned}\pi_n &\leq \sum_{u=1}^{n} \min\Big(u-1, \frac{2m}{u}\Big)^2 \leq \sum_{u=1}^{\sqrt{2m}} u^2 + \sum_{u=\sqrt{2m}}^{n} \frac{(2m)^2}{u^2} \\ &\sim \frac{(2m)^{1.5}}{3} + 4m^2\Big(\frac{1}{\sqrt{2m}} - \frac{1}{n}\Big) \sim n^{3(1+a)/2}.\end{aligned} \tag{29}$$

Since we assumed that $\pi_n \sim n^{1+2a+\epsilon_n}$, we get that $\epsilon_n \leq (1-a)/2$. Letting $\epsilon_n \to \epsilon$ as $n \to \infty$, the statement of the theorem follows. ∎

Note that regular graphs (i.e., all degree equal to each other) yield $\epsilon = 0$ for all a. Another well-known case follows from [40]. Specifically, for a sequence of graphs $\{G_n\}$, define D_n to be a random variable with the same distribution as undirected degree in G_n. Then, assuming $E[D_n^{4/3}]$ converges to a finite constant as $n \to \infty$, these graphs also achieve $\epsilon = 0$. For more general cases, the family of dense-core graphs introduced next allows realization of any ϵ.

Theorem 8. *For any $\epsilon \in [0, (1-a)/2]$, there exists a graph with $\omega(\pi) = 1 + 2a + \epsilon$.*

Proof: Assume a graph where the first k_n nodes, each with degree $l_n \leq k_n$, link to nodes with labels $(1, 2, \ldots, l_n)$. All remaining nodes have degree two and link to nodes $(1, 2)$. Then, assuming $k_n \sim n^{z_1}$ and $l_n \sim n^{z_2}$, where $z_1 \geq z_2$ and $z_1 + z_2 \geq 1$, we get

$$E[D_n] = \frac{k_n l_n + 4(n-k_n)}{n} \sim n^{z_1+z_2-1} \tag{30}$$

and

$$\pi_n \sim \sum_{u=1}^{l_n} u^2 + \sum_{u=l_n}^{k_n} l_n^2 + n - k_n \sim k_n l_n^2 \sim n^{z_1+2z_2}. \tag{31}$$

Assume a is selected first and ϵ is selected second in the range $[0, (1-a)/2]$. Then, we can construct the system above using $z_1 = 1 - \epsilon$ and $z_2 = a + 1 - z_1$. Note that $z_1 + z_2 = a + 1 \geq 1$ is satisfied with any $a \geq 0$. Furthermore, condition $z_1 \geq z_2$ is equivalent to $2z_1 \geq a+1$, or $\epsilon \leq (1-a)/2$, which is satisfied by any valid ϵ. ∎

Leveraging the last two theorems finally produces a usable upper bound on the scaling rate of $I_B(n)$.

Theorem 9. *The rate of PCF-1B I/O is upper-bounded by*

$$\omega(I_B) \leq \min(1 + 2a + \epsilon, 2 + 2a - r). \tag{32}$$

Furthermore, in the worst-case of $\epsilon = (1-a)/2$, the graphs built in Theorem 8 reach (32) *for all a and r.*

Proof: From (23), it is clear that

$$I_B(n) \leq \min(\pi_n, (p-1)m) \leq \min\Big(\pi_n, \frac{m^2}{M}\Big). \tag{33}$$

Converting this into rates yields (32).

Next, for any a and $\epsilon = (1-a)/2$, the graphs introduced in Theorem 8 require $z_1 = z_2 = (1+a)/2$. Then, we can set $k_n = 2l_n$ and obtain that out-lists of source nodes $u \in [l_n, 2l_n]$ have $X_u = l_n$ neighbors and roughly $\zeta_u = l_n^2/M$ colors. Converting this into asymptotics, it follows that the out-degree of these nodes scales as z_2 and the number of colors as $2z_2 - r$. Therefore, when $z_2 < 2z_2 - r$, or equivalently $r < 1 - \epsilon = (1+a)/2$, PCF-1B has the same asymptotics as π_n. This makes $\omega(I_B) = 1 + 2a + \epsilon$. Otherwise, $I_B(n)$ scales as $k_n l_n \zeta_u \sim n^{2+2a-r}$. Both cases and the condition to switch between them are exactly the same as in (32). ∎

The graphs from Theorem 8 bring out the worst in PCF, to which we come back shortly. In the mean time, we show that it has a pretty impressive best-case as well.

Theorem 10. *In bipartite graphs, PCF-1B has I/O overhead $I_B(n) = m$ for all a and r, i.e., $\omega(I_B) = 1 + a$.*

Proof: Suppose the nodes are divided into two sections, which we call S_1 and S_2, of size k_n and $n - k_n$, respectively. Each node in S_1 connects to all nodes in S_2. We assume that $k_n < n/2$, i.e., the first section is smaller, and $k_n \sim n^a$. Notice that this graph has an average degree proportional to

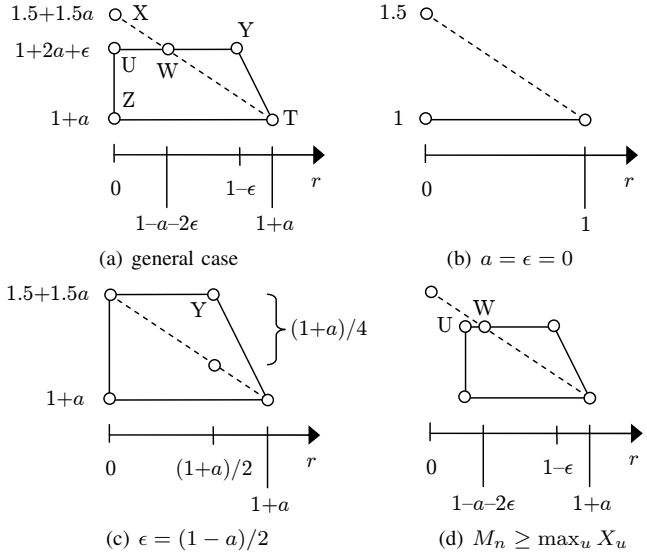
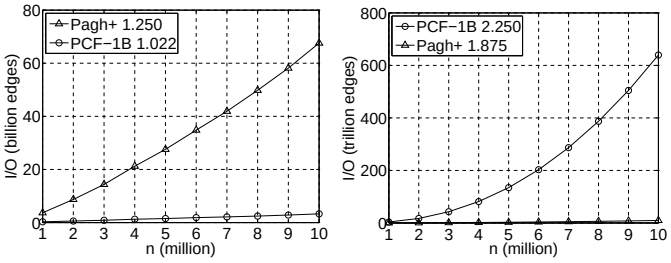


Fig. 5. Comparison of scaling rates.



(a) $a = \epsilon = 0, r = 1/2$ (b) $a = 1/2, \epsilon = 1/4, r = 3/4$

Fig. 6. Actual I/O with curve-fitted scaling rates.

n^a and that none of the nodes in S_1 have any out-neighbors in G_θ^+. Therefore, PCF-1B assigns them the same color 1. It then follows that the companion I/O from all nodes $u \in S_2$ is no more than m since each out-neighbor list N_u^+ has nodes of one color and thus can only produce output to one companion file $G_\theta^c(1)$. Furthermore, this result holds for all M_n. ■

Because PCF-1A load-balances partitions on the in-degree, rather than the out-degree, it fails to achieve the same benefits on bipartite graphs. Since PCF-1B cannot have less I/O than m, Theorem 10 shows that this bound is tight.

C. Analysis

We summarize the findings of this section using Fig. 5(a). The x-axis shows rate r at which RAM increases as $n \to \infty$. This value ranges from zero (i.e., constant M_n) to $1 + a$ (i.e., the entire graph fits in memory). On the y-axis, we have Pagh's scaling rate $\omega(I_P)$, represented by a dashed line, and the PCF-1B rate $\omega(I_B)$, given by the $UYTZ$ trapezoid. Pagh's curve is a straight line that comes from (26). On the other hand, the rate of PCF-1B is contained somewhere in the trapezoid, with each interior point possibly corresponding to some graph G. The upper boundary, delineated by segments UY and YT, is produced by graphs from Theorem 8. The lower boundary, shown by line ZT, is the bipartite graph from Theorem 10.

At $r = 0$, i.e., constant RAM, Pagh begins in point X that is always no lower than PCF-1B's worst initial point U. This happens because $1.5+1.5a \geq 1+2a+\epsilon$ for all $\epsilon \leq (1-a)/2$. As r increases, Pagh descends and eventually intersects with the upper bound of PCF-1B in point W. Therefore, in the range $[0, 1-a-2\epsilon)$, Pagh has no chance of beating PCF-1B, regardless of the actual G. Between points W and T, some of the graphs are solved quicker by Pagh and others by PCF-1B.

It can be seen from the figure that the largest gap between the two methods occurs at $r = 0$, where PCF-1B in point Z vanquishes Pagh in point X by $(1 + a)/2$. Using a complete bipartite graph with $a = 1$, this yields a factor of n improvement in favor of PCF-1B. Outside of this custom-tailored graph, a more realistic best-case scenario for PCF-1B consists of graphs with a constant average degree and $\epsilon = 0$. This is depicted in Fig. 5(b), where PCF-1B collapses the trapezoid into a single line and defeats Pagh for all r. The biggest gap occurs at $r = 0$, where PCF-1B has a factor of $\sqrt{n}$ less I/O.

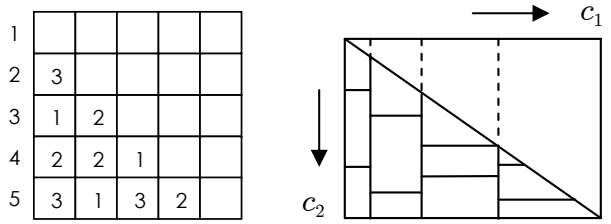
On the other hand, the best case for Pagh is $\epsilon = (1-a)/2$, which is shown in part (c) of the figure. In this situation, it beats the upper-bound of PCF-1B for all memory sizes. Consequently, knowing that G has a dense core similar to the graphs in Theorem 8, Pagh is the method of choice. The largest improvement is achieved in $r = (1+a)/2$, where Pagh undercuts the scaling rate of PCF-1B by $(1 + a)/4$. Since $a \leq 1$, point Y causes the most damage to PCF in complete graphs, i.e., when $a = 1$. On these, Pagh has smaller cost by a factor of $\sqrt{n}$.

The final caveat is shown in Fig. 5(d), where the trapezoid has its left boundary moved forward to reflect the fact that $M_n \geq \max_u X_u$ must hold for PCF-1B to work. While it is hard to predict how far point U shifts without access to the actual graph, we know it is no further than $r = (1 + a)/2$ since $\max_u X_u \leq \sqrt{2m}$. This may be to the left of W, as show in the picture, or to the right. In either case, Pagh wins by default for all r where PCF-1B is unable to execute.

To see some of these cases in practice, Fig. 6(a) shows the actual I/O of the two methods in a random graph with Pareto degree, where shape $\alpha = 1.5$ and average degree is 30. As predicted by our analysis and Fig 5(b), the asymptotic gap between the methods is $n^{1/4}$. Continuing to Fig. 6(b), we examine a dense-core graph from Theorem 8 whose average degree scales as $\sqrt{n}$, RAM size $M_n = n^{3/4}$, and $\epsilon = 1/4$. This puts the graph on the upper-bound of PCF, where the model suggests Pagh should win by $n^{3/8}$. Indeed, it does.

D. Discussion

We can now summarize the insight gained from dissecting both methods. Pagh's main pitfall is that it fails to exclude nodes u that obviously cannot be in any triangles of relevant color. For example, if u has out-neighbors of color j, but none of color k, it should not be used in conjunction with remote edges E_{jk}^+. This leads to epic redundancy when the graph is sparse, i.e., there are few colors among the neighbors. On the other hand, this strategy works well for dense graphs where



(a) general case (b) contiguous case

Fig. 7. Heterogenous 2D partitioning of remote edges.

Algorithm 6: UPI creating companion files

```
1  for u = 1 to n do
2      foreach v ∈ N_u^+ do ◁ iterate through all out-neighbors
3          foreach partition ℓ where v ∈ S_ℓ do ◁ v is a source in B_ℓ
4              Z_vℓ = {w | (v, w) ∈ B_ℓ}  ◁ take neighbors of v in B_ℓ
5              L_uvℓ = N_u^+ ∩ Z_vℓ  ◁ local list for (u, v) in partition ℓ
6              if L_uvℓ \ {v} = ∅ then continue
7              if u ∈ S_ℓ then ◁ u is also a source in partition ℓ
8                  L_uvℓ = ∅  ◁ all local nodes in E_ℓ^+
9                  if v ∈ D_ℓ then continue ◁ (u, v) already in E_ℓ^+
10             write (u, v, L_uvℓ) to companion graph C_ℓ^+
```

little pruning is necessary in the first place. The number of hash-table lookups proportional to c is also a concern.

On the other hand, the main downside of PCF lies in one-dimensional color partitioning. This creates a large number of colors p and causes unnecessary duplication of effort. Usage of 2D coloring could help reduce the number of files into which the out-neighbors must be written. This can be seen in (10), where making R'_{us} pick out of $\sqrt{p}$ colors, rather than p, would be a noticeable improvement.

VII. Trigon

Our investigation discovered that an ideal algorithm should prune unnecessary edges, be able to utilize $\sqrt{p}$ colors, deterministically load-balance partitions, leverage sequential colors for faster compression/intersection/lookups, handle star graphs without exorbitant overhead, operate with $O(1)$ RAM, and post lower I/O numbers than either of the current techniques. We offer such an approach next.

A. Generalized Coloring

All vertex/edge iterators [9] require the remote edge of enumerated triangles to be retrievable using random lookup in RAM. Therefore, for such methods to operate in external memory, the oriented graph must be split into at least $p = m/M$ chunks. For now, we ignore the issue of *how* partitioning should be done and focus on the general concepts that would allow the in-memory search to function properly. The framework developed below applies to all 18 methods from [9]; however, to keep the notation to a minimum, we only describe how it works with E_1.

Since G_θ^+ is oriented and without self-loops, only the lower half of the adjacency matrix has non-zero entries. Therefore, any edge partition can be viewed as a subset of

$$B = \{(u, v) \in \mathbb{N}^2 \mid v < u \leq n\}, \tag{34}$$

which is a collection of all integer pairs (u, v) such that $u > v$ and both numbers are no larger than n. Now suppose there exist sets $B_1, \ldots, B_p$ that form a partition on B, i.e., $B_\ell \subseteq B$ for all ℓ, $B_i \cap B_j = \emptyset$ for $i \neq j$, and $\cup_{\ell=1}^{p} B_\ell = B$. This is illustrated using Fig. 7(a), where a 5×5 adjacency matrix is split into three subgraphs. The number in each cell specifies the partition ℓ it belongs to.

Note that all previous methods are special cases of this formalization. For example, Pagh uses $B_{(j-1)c+k} = \{(u, v) \mid u \in V_j, v \in V_k\}$, where $c = \sqrt{p}$ is the number of colors. Both PCF methods utilize contiguous partitions shown in Fig. 7(b), where destinations are split into c_1 colors and sources nodes into $c_2 = p/c_1$. PCF-1A uses $c_1 = p$, while PCF-1B does the opposite, i.e., $c_1 = 1$.

Once partitions are decided, the edges of G_θ^+ must be separated into sets $E_1^+, \ldots, E_p^+$, where $E_\ell^+ = E_\theta^+ \cap B_\ell$ for $\ell = 1, 2, \ldots, p$ and the following condition enforced.

Definition 2. *A partition $\{B_\ell\}$ is called* admissible *with respect to G_θ^+ if it guarantees that $|E_\ell^+| = m/p$ for all ℓ.*

As discussed earlier, Pagh fails to produce admissible partitions on star graphs and similar structures. PCF-1A attempts to split the destinations into $c_1 = p$ colors in Fig. 7(b) and runs into the same problem. On the other hand, PCF-1B is able to produce admissible partitions in all G as long as $\max_u X_u \leq M$.

B. Unified Partitioned Iterator

Assume the edges of G_θ^+ have been separated into individual files. What remains is creation of companion files, which is done in a framework we call *Unified Partitioned Iterator* (UPI). Let $S_\ell = \{u \mid (u, v) \in B_\ell\}$ be the source nodes and $D_\ell = \{v \mid (u, v) \in B_\ell\}$ be the destination nodes in partition ℓ. For the example in Fig. 7(a), $S_3 = \{2, 5\}$. Operation of UPI is summarized in Algorithm 6. For each node u and its out-neighbor v, Line 3 finds all partitions ℓ where v is a source. Next, recalling Fig. 1(a), observe that the local list needs to be customized to include only those neighbors w of u that are possibly neighbors of v in B_ℓ. This is done in Lines 4-5. If the local list is empty or contains only v, then v cannot be u's hit node for partition ℓ and the algorithm moves on in Line 6.

Line 7 checks if u itself participates in B_ℓ as a source node. If so, the entire local list is already included in E_ℓ^+, which Line 8 signals by emptying $L_{uv\ell}$. Additionally, it is possible that link (u, v) is also contained in the remote graph, which happens if v is a destination node in B_ℓ. Line 9 takes care of this condition. Finally, Line 10 saves the triple $(u, v, L_{uv\ell})$ into the companion file, which is done even if $L_{uv\ell}$ was previously emptied in Line 8.

Triangle search in UPI is shown in Algorithm 7. The only difference from Pagh+ is that each partition ℓ has its own companion file, from which nodes u, their hit neighbors v, and local lists $L_{uv\ell}$ are obtained.

Theorem 11. *UPI finds each triangle exactly once and exhibits no more intersection overhead than in-memory E_1.*

Algorithm 7: UPI processing one partition ℓ

```
1 load E_ℓ^+ = {(v, N_vℓ^+)} in RAM; set up hash table to source nodes
2 while companion file C_ℓ^+ not empty do
3     load (u, v, L_uvℓ) from C_ℓ^+
4     find remote list N_vℓ^+ using the hash table
5     W = Intersect(L_uvℓ, N_vℓ^+) ◁ local/remote lists
6     foreach w ∈ W do report Δ_uvw
```

Proof: Because the edges are partitioned into non-overlapping and exhaustive sets, detecting the same triangle multiple times or missing some of them is impossible. This is a consequence of the fact that remote edge (vw) belongs to exactly one partition ℓ.

We now consider the intersection overhead of Algorithm 6. The local intersection cost at node u can be written as

$$\sum_{v\in N_u^+}\sum_{\ell=1}^{p}|L_{uv\ell}| = \sum_{v\in N_u^+}\sum_{\ell=1}^{p}|N_u^+\cap Z_{vl}|. \tag{35}$$

Since $\{Z_{v1},\ldots,Z_{vp}\}$ is a partition of v's possible neighbor options $[1,v-1]$, we get that

$$\begin{aligned}\sum_{v\in N_u^+}\sum_{\ell=1}^{p}|N_u^+\cap Z_{vl}| &= \sum_{v\in N_u^+}|N_u^+\cap[1,v-1]|\\ &= \frac{X_u(X_u-1)}{2},\end{aligned} \tag{36}$$

which is exactly the same as in E_1.

Now suppose $Y_{v\ell}$ is the in-degree of v from hit lists in partition ℓ and let $X_{v\ell}$ be its out-degree in the remote graph E_ℓ^+. Since node v is hit $Y_{v\ell}$ times in ℓ, each causing a scan over $X_{v\ell}$ neighbors, the remote intersection overhead for v equals

$$\sum_{\ell=1}^{p}X_{v\ell}Y_{v\ell}\le Y_v\sum_{\ell=1}^{p}X_{vz}=X_vY_v. \tag{37}$$

Combining the upper bound in (37) with (36), we get the cost of E_1 in (1). ∎

The proof of this theorem shows that intersection cost can actually *reduce* as p increases. This happens because node v participates in remote intersection only when there is a hit-list edge (u,v) in the corresponding companion file. However, if u has no other neighbors smaller than v in partition ℓ, Line 6 of Algorithm 6 discards v as being ineligible. In practice, cost reduction only affects the X_uY_u term in (1) and happens only in partitioning schemes that break some of the out-lists N_u^+ across multiple E_ℓ^+ (i.e., Pagh and PCF-1A).

C. Trigon

We next decide how to achieve the best admissible partition within the general framework above. On one hand, it is theoretically possible to customize set $\{B_\ell\}$ to a particular G_θ^+ in order to achieve the absolute minimum I/O for that graph. However, this solution is expensive (i.e., NP-hard) as it requires steam-rolling through all possible subsets of m edges. Instead, we are interested in alternative approaches that can be computationally reasonable.

To this end, recall our discussion of PCF and Pagh, where random assignment of nodes into colors would have produced stochastically larger R_{us} and R'_{us} in (7)-(8). The best technique, which comes from PCF, is to group nodes of the same color together. This forces members of N_u^+ to pick color from a smaller range of options (i.e., those contained in $[1,u-1]$). Additionally, continuous colors simplify preprocessing, remove redundancy between local lists of different hit nodes v, and improve intersection/compression performance. At the same time, Pagh's lowering of c to $\sqrt{p}$ is appealing as well. Combining these ideas, it turns out that the design in Fig. 7(b) is the most sensible solution.

We call this approach *Trigon* and discuss its operation next. Since there are two colors involved (i.e., along the source and destination nodes), we call the one whose partitions are decided first *primary* and the other *secondary*. One option is to use c_1 primary and c_2 secondary colors, which is the case in Fig. 7(b). This approach starts by selecting vertical boundaries such that the number of edges contained in each primary color equals m/c_1. This is done by computing set $\{a_k\}_{k=1}^{c_1}$ such that

$$\sum_{u=a_k}^{a_{k+1}-1}Y_u=\frac{m}{c_1}, \tag{38}$$

where Y_u is the in-degree of u. Note that this is exactly how PCF-1A begins and that the $\{Y_u\}$ sequence is available during orientation of G, i.e., at no extra cost.

Then, for each primary color k, suppose boundaries $\{b_{kj}\}_{j=1}^{c_2}$ specify the corresponding ranges of secondary colors. This is accomplished by load-balancing the out-degree within each partition (kj), i.e.,

$$\sum_{u=b_{kj}}^{b_{k,j+1}-1}|N_u^+\cap[a_k,a_{k+1})|=M, \tag{39}$$

which is similar to PCF-1B. For Fig. 7(b), this means each vertical column has size m/c_1 and each rectangle fits in RAM. Note that if (38) fails to create enough partitions of primary color, e.g., on star-like graphs, the value of c_1 is lowered to match the particulars of G_θ^+. To compensate for the lack of vertical partitions, (39) automatically increases the number of secondary colors such that $c_1c_2=p$ continues to hold.

The second option is to reverse this process, i.e., use source nodes for primary colors. However, it is not difficult to see that this procedure offers no I/O benefits due to symmetry, but at the same time has a major drawback in inability to adapt c_1 to G_θ^+. Therefore, the configuration in Fig. 7(b) is preferred.

The Trigon split technique is shown in Algorithm 8, where we continue using color k for node w and color j for v to maintain compatibility with Fig. 1(b). The algorithm is pretty much self-explanatory, with the only caveat being Line 8. Under Trigon's coloring model, it is now possible for local list L_{uk} to contain nodes w larger than any hit node in H_{ukj}. They can never complete directed triangles in Fig. 1, which explains their removal.

D. Analysis

Suppose ϕ_u and ϕ_{us} are defined as before, except they now refer to respectively the *primary* color of u and that of its

Algorithm 8: Trigon writing companion files

```
for u = 1 to n do
    for k = 1 to c_1 do ◁ run thru primary colors
        L_uk = N_u^+ ∩ [a_k, a_{k+1})  ◁ local list for color k
        if L_uk ≠ ∅ then ◁ work to be done?
            for j = 1 to c_2 do ◁ run thru secondary colors
                H_ukj = N_u^+ ∩ [b_kj, b_{k,j+1})  ◁ hit list for pair (kj)
                if H_ukj ≠ ∅ and |L_uk ∪ H_ukj| ≥ 2 then
                    L_uk = L_uk ∩ [1, max(H_ukj)]  ◁ prune
                    if φ_u(k) = j then ◁ local list in RAM
                        write (u, H_ukj \ L_uk) to companion C_kj^+
                    else
                        write (u, H_ukj ∪ L_uk) to C_kj^+
```

s-th out-neighbor. In this notation, expression (7) still works for R_{us}. Similarly, R_u counts the number of primary colors in N_u^+. To handle the vertical dimension with c_2 colors, let $\varphi_u(k)$ be the *secondary* color of node u with respect to primary color k and assume $\varphi_{us}(k)$ is the same for u's out-neighbor s. Then, (8) is replaced with

$$R''_{us} = |\{\varphi_{ut}(\phi_s) \mid t > s, \varphi_{ut}(\phi_s) \neq \varphi_u(\phi_s)\}|, \tag{40}$$

which counts the number of secondary colors to the right of s, again excluding the color of u. Using the analysis of PCF-1B, the next result follows immediately.

Theorem 12. *The I/O complexity of Trigon is*

$$I_T(n) \approx \sum_{u=1}^{n} \Big[R''_{u1} + \sum_{s=1}^{X_u} (R_{us} + R''_{us}) \Big] \tag{41}$$

and the number of hash-table lookups is

$$\gamma_T(n) = \sum_{u=1}^{n} \sum_{s=1}^{X_u} R_{us} + m - \sum_{u=1}^{n} R_u. \tag{42}$$

With the exception of minor terms related to overlapping local/hit lists, the result in (41) is exact. To perform a self-check, notice that PCF-1A (i.e., $c_1 = p$) has $R''_{us} = 0$, which converts (41) into (9). For PCF-1B (i.e., $c_1 = 1$), we get $R_{us} = 0$ and $R''_{us} = R'_{us}$, which makes the Trigon model identical to (10). Intuitively speaking, (41) can be viewed as a sum of I/O in PCF-1A running with c_1 colors and PCF-1B with c_2 colors, although this is approximate since R''_{us} does not equal R'_{us} unless $c_1 = 1$. Recalling (13), also observe that (42) is exactly the number of lookups in PCF-1A under c_1 partitions, where more primary colors cause more CPU cost.

Following the proof of Theorem 6, there exists a simple bound on $I_T(n)$ that shows the impact of each color.

Theorem 13. *The Trigon I/O is upper-bounded by*

$$I_T(n) \leq \sum_{u=1}^{n} X_u h(X_u), \tag{43}$$

where $h(x) = \min(x/2, c_1) + \min(x/2, c_2 - 1)$.

The first term of $h(x)$ bounds the size of hit lists (and the number of lookups), while the second models the size of local lists. Additionally, since $h(x) \leq c_1 + c_2 - 1$, usage of $c_1 = c_2 = \sqrt{p}$ in (43) yields a looser bound

$$I_T(n) \leq \sum_{u=1}^{n} X_u(c_1 + c_2 - 1) = (2\sqrt{p} - 1)m, \tag{44}$$

which is the I/O cost of Pagh+ in Theorem 1. Since colors are sequential, Trigon beats (44) even in complete graphs, where it comes the closest to this bound, by roughly a factor of 2. It is also clear that (42) is upper bounded by $c_1 m$. Recalling Theorem 2, this makes $\gamma_T(n)$ better than the corresponding metric in Pagh+ for all $c_1 \leq \sqrt{p}$.

Under an appropriately-chosen c_1, Trigon is no worse than either of the previous methods; however, the best choice for the number of primary colors remains far from obvious.

E. Minimizing I/O

One big question is whether deploying $c_1 = \sqrt{p}$ is optimal for achieving the lowest I/O. This seems logical as it reduces the number of colors in each direction to their minimum. Because analysis of the accurate model (41) currently appears intractable, we only consider insight that might be gained from the upper-bound (43), which can be written as $E[Xh(X)]$ for some random variable X.

Since c_1 and c_2 are almost interchangeable in $h(x)$, it makes sense to study the following simplified problem. Suppose we are interested in minimizing

$$\xi(c) = E[X(\min(X, c) + \min(X, p/c))], \tag{45}$$

where X is a random variable that represents the out-degree of G_θ^+ and c is the number of primary colors.

Theorem 14. *If X has density $f(x)$,* (45) *is minimized by $c = 1$, $c = p$, or any solution to $g(c) = g(p/c)$, where*

$$g(y) = y \int_y^\infty x f(x) dx. \tag{46}$$

Proof: Suppose $X \sim F(x)$. Then, we can expand the expectation in (45) as

$$\begin{aligned} \xi(c) &= \int_0^\infty x(\min(x, c) + \min(x, p/c)) dF(x) \\ &= \int_0^c x^2 dF(x) + c \int_c^\infty x dF(x) + \int_0^{p/c} x^2 dF(x) \\ &\quad + \frac{p}{c} \int_{p/c}^\infty x dF(x). \end{aligned} \tag{47}$$

Differentiating with respect to c and applying Leibnitz's integration rule four times,

$$\begin{aligned} \frac{d\xi(c)}{dc} &= c^2 f(c) - c^2 f(c) + \int_c^\infty x f(x) dx - \frac{p^3 f(p/c)}{c^4} \\ &\quad + \frac{p^3 f(p/c)}{c^4} - \frac{p}{c^2} \int_{p/c}^\infty x f(x) dx \\ &= \int_c^\infty x f(x) dx - \frac{p}{c^2} \int_{p/c}^\infty x f(x) dx \\ &= \frac{g(c) - g(p/c)}{c}. \end{aligned} \tag{48}$$

Optimal c is either a solution to $g(c) = g(p/c)$ or lies on the boundary, i.e., $c = 1$ or $c = p$. ∎

Notice that $c = \sqrt{p}$ is a trivial solution to $g(c) = g(p/c)$. Furthermore, it is the *only* solution if $g(x)$ is monotonic. Outside of certain esoteric cases, this result shows that the optimal Trigon configuration is PCF-1A, PCF-1B, or $c_1 = \sqrt{p}$. However, there is no clear winner for all graphs G. The next example shows one such case.

Theorem 15. *If $X < 2\sqrt{p} - 1$ with probability 1, then $c = 1$ or $c = p$ is optimal in (45). On the other hand, if $X > 2\sqrt{p}-1$ with probability 1, then $c = \sqrt{p}$ is optimal.*

Proof: Because the objective function is symmetric in c, we only need to consider $c \in [1, \sqrt{p}]$. Any optimal solution c has an optimal counterpart $1/c$. Suppose $X \sim F(x)$ is defined in $[1, n-1]$ and rewrite (45) as

$$\xi(c) = \int_1^{n-1} x\xi(c, x)dF(x), \tag{49}$$

where $\xi(c, x) = \min(x, c) + \min(x, p/c)$. First suppose that $x \leq \sqrt{p}$, in which case $\xi(c, x)$ becomes $\min(x, c) + x$. This is trivially minimized by $c = 1$. Second, suppose $x > \sqrt{p}$, in which case we get $\xi(c, x) = c + \min(x, p/c)$. There are two subcases here – 1) $x < p/c$ yields $c + x$, where $c = 1$ is optimal; and 2) $x \geq p/c$ produces $c + p/c$, where $c = \sqrt{p}$ is best. In the former subcase, the lowest cost is $x+1$ and in the latter it is $2\sqrt{p}$. Therefore, $c = 1$ is better when $x < 2\sqrt{p}-1$, worse when $x > 2\sqrt{p} - 1$, and the two are equal otherwise.

As a result, if X is limited to $[1, 2\sqrt{p}-1]$, we get that (49) minimized by $c = 1$. On the other hand, if X is always larger than $2\sqrt{p} - 1$, the integral is minimized by $\sqrt{p}$. ∎

One example that falls under Theorem 15 are d-regular graphs. This is illustrated in Fig. 8(a) using a random graph with $d = 10$, $n = 10$M, and $p = 1024$. The I/O function of Trigon in this graph is an inverted cup, with the middle being the worst and the two boundaries being the best. This is one of the few cases where PCF-1A wins over PCF-1B. A more common scenario is given by Twitter in Fig. 8(b), where $c_1 = \sqrt{p} = 32$ is clearly optimal and PCF-1B beats PCF-1A.

If the program has access to G_θ^+, it can compute our models shown earlier in the paper and always make the right decision. However, if graph G_θ^+ cannot be examined before choosing c_1, the next result explains which choice would always be safer.

Theorem 16. *Usage of $c = \sqrt{p}$ in (45) yields at most double the optimal I/O. On the other hand, $c = 1$ or $c = p$ can be worse than optimal by a factor of $\sqrt{p}$.*

F. Minimizing Runtime

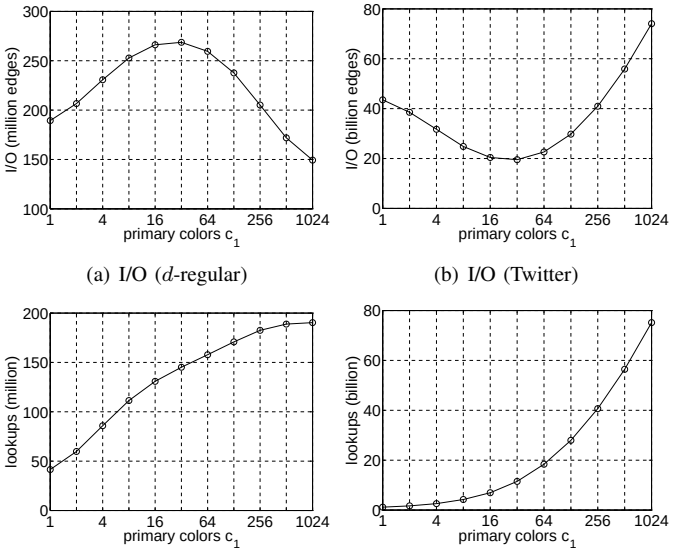
When achieving the quickest execution time is a priority, the choice of optimal c_1 may involve balancing conflicting objectives. This is exemplified by Fig. 8(c)-(d), where optimal points c_1 do not coincide with those in plots (a)-(b). Note that the x-axis is on a $\log_2$ scale and lookup growth is sublinear. On the d-regular graph, Trigon increases $\gamma_T(n)$ by 3.5 times between $c_1 = 1$ and $\sqrt{p}$. On Twitter, the number of lookups goes up by a factor of 9.8. As predicted earlier, both values are much smaller than Pagh's linear (i.e., 32-fold) increase.

With overlapped operation between CPU and I/O, the runtime is determined by the maximum of disk read time and in-memory operations. Define S_D, S_I, and S_H to be respectively the speed of the disk, intersection, and lookups (in edges/sec), which can be easily benchmarked on startup. Parameterizing $I_T(n)$ and $\gamma_T(n)$ with c_1, an objective might be to minimize

$$r(c_1) = \max\Big(\frac{I_T(n, c_1)}{S_D}, \frac{\rho(n)}{S_I} + \frac{\gamma_T(n, c_1)}{S_H}\Big) \tag{50}$$

where $\rho(n)$ is the intersection cost from (1).

To obtain $I_T(n, c_1)$ and $\gamma_T(n, c_1)$, one can use (41)-(42). Direct computation of these values may be costly; however, approximation (43), as well as its refinement using (16) or (23), work quite well. A binary search over $r(c_1)$ requires efficient computation of the models, i.e., without scanning the degree sequence $\{X_u\}$, which may not fit in RAM. Our approach is to create a short digest of the necessary information during construction of G_θ^+, which summarizes the in/out degree sequences of the graph. Since $\{X_u\}$ typically contains many runs of similar values $(X_u, X_{u+1}, \ldots, X_{u+s-1})$, each of them can be compressed into one entry that keeps track of the count s and the starting value X_u. As a result, minimization of (50) often takes negligible time.



(a) I/O (d-regular) (b) I/O (Twitter)

(c) lookups (d-regular) (d) lookups (Twitter)

Fig. 8. Trigon tradeoffs between I/O and lookups ($p = 1024$).

VIII. Evaluation

We finally come to the stage of putting the ideas developed in the previous section to work. To enable a fair comparison, we use C++ to implement Trigon and Pagh+ as separate modules that share the same in-memory and disk components (i.e., multi-threading, overlapped I/O, SIMD intersection). Setting $c_1 = 1$ in Trigon, we obtain PCF-1B. Therefore, the only difference between the three methods lies in their partitioning scheme. As PCF-1A is not competitive on most real-world graphs, we do not consider it here.

Out of the standard graphs used for triangle listing, we engage the six largest from [9]. Their characteristics are shown in Table I. In the last two rows, we add into the mixture best-case scenarios from Pagh and PCF.

TABLE I
GRAPH PROPERTIES

Graph	Nodes n	Edges m	Size (GB)	Triangles
Twitter [19]	41M	1.2B	9.3	35B
Yahoo [41]	720M	6.4B	53.3	86B
IRL-domain [9]	86M	1.7B	13.3	113B
IRL-host [9]	642M	6.4B	52.7	437B
IRL-IP [9]	1.6M	818M	6.1	1040B
ClueWeb [9]	8.2B	51B	358	879B
Complete	100K	5.0B	37.2	167T
Bipartite	100K	2.5B	18.6	0

TABLE II
I/O (BILLION EDGES)

Graph	p	Pagh+	PCF-1B	Trigon	RAM
Twitter	1,024	75.6	43.5	19.5	4.5 MB
Yahoo		392.3	25.5	25.5	23.2 MB
IRL-domain		104.8	98.4	33.8	6.2 MB
IRL-host		386.5	137.9	59.7	22.9 MB
IRL-IP		51.5	145.7	23.4	3.0 MB
ClueWeb		2,869.9	457.1	326.2	169.7 MB
Complete	10,000	995.0	15,742	493	1.9 MB
Bipartite		497.0	2.5	2.5	1.0 MB

TABLE III
PREPROCESSING AND ENUMERATION TIME (MINUTES)

Graph	Pagh+			Trigon		
	pre	run	total	pre	run	total
Twitter	3.3	144.0	147.4	14.8	10.0	24.8
Yahoo	27.8	1,296.4	1,324.2	35.5	19.1	54.6
IRL-domain	3.5	191.4	194.9	21.0	14.8	35.8
IRL-host	26.2	1,070.3	1,096.5	52.7	32.0	84.7
IRL-IP	0.2	31.7	31.9	12.1	8.7	20.8
ClueWeb	181.8	8,331.1	8,512.9	426.8	254.3	681.1
Complete	2.5	1,050.7	1,053.2	624.2	238.6	862.8
Bipartite	8.8	629.5	638.3	6.6	2.3	9.9

TABLE IV
NUMBER OF LOOKUPS (BILLION)

Graph	Pagh+	Trigon	Ratio
Twitter	38.4	11.5	3.5
Yahoo	199.2	19.9	10.0
IRL-domain	53.2	19.4	2.7
IRL-host	196.3	34.3	5.8
IRL-IP	26.2	13.2	2.0
ClueWeb	1,457.8	205.3	7.1
Complete	500.0	252.0	2.0
Bipartite	250.0	2.5	100.0

A. I/O

Performance of triangle listing depends on the ratio between graph size and available RAM, i.e., $p = m/M$. Since our I/O methods are quite efficient, this affords us an opportunity to examine scenarios where graphs are substantially larger than memory. In fact, this is the first paper that runs an actual implementation with RAM size that is $3-4$ orders of magnitude smaller than the oriented graph G_θ^+.

On real-world graphs, Table II shows that Pagh+ loses to PCF-1B in five out of the six cases, sometimes by as much as a factor of 15. The only graph where it wins is IRL-IP, which is quite dense (average degree 1,030). This is not surprising given our earlier analysis. If we consider preprocessing to be part of triangle listing and double the PCF-1B result, it becomes worse that Pagh+ in three cases. On the other hand, Trigon beats both previous methods on each of the graphs. Furthermore, even if its I/O is doubled, it still stays below Pagh+, in some cases by a wide margin.

On the complete graph and 10K partitions, Pagh+ has 15 times less I/O than PCF-1B. However, its overhead is still double that of Trigon, which follows from the dichotomy of sequential vs random coloring discussed earlier. On the bipartite graph, PCF-1B and Trigon both annihilate Pagh+ by issuing 200 times less I/O, which also agrees with our analysis.

B. Runtime

For the experiments, we use one machine with a six-core Intel i7-3930K (desktop CPU released in 2011). We equip this computer with a single 3-TB magnetic hard drive (Hitachi Deskstar 7K3000) that is capable of reads at 160 MB/s. We omit PCF-1B since slow I/O makes it predictably worse than Trigon. Instead, we compare against Pagh+ to investigate the impact of non-sequential colors, lookup load, and disk seeking. Furthermore, we consider the total delay, which includes the partitioning phase, as one of the measures of performance.

Table III shows the result. In the first four rows, Trigon completes triangle search $15-60$ times quicker than Pagh+. One notable example is Yahoo, where purely sequential I/O in Pagh+ would have been responsible for only 163 minutes (i.e., 392B edges, four bytes each, read at 160 MB/s). Instead, Pagh+ spends an additional 1,132 minutes (i.e., 18 hours) on lookups. A similar scenario occurs with ClueWeb in row six, where Pagh+ gets bogged down for 5 days just checking the hash table. Table IV confirms that Pagh+ requires substantially more random memory access than Trigon. The larger the hash-table size, the worse the lookup speed, which explains the huge runtime gap between the two methods on ClueWeb.

In dense-graph scenarios of Table III, Pagh+ is $3-5$ times slower than Trigon. Usage of 10K partitions for the complete graph creates a noticeable bottleneck in reading $c^3 = 1$M combinations of files. Analysis of the total delay, i.e., both preprocessing and triangle listing, shows a more favorable outcome for Pagh+; however, Trigon is still faster in all graphs, sometimes by a wide margin (e.g., $24\times$ on Yahoo).

IX. CONCLUSION

We analyzed I/O complexity of the best methods in the literature, compared their asymptotics, identified their inherent strengths and weaknesses, and developed a novel framework that surpassed the existing efforts in all performance measures relevant to triangle listing. Our approach works by trading I/O cost for lookups, which makes the method adaptable to whatever bottlenecks triangle listing may be facing in a particular hardware configuration.

REFERENCES

[1] S. Arifuzzaman, M. Khan, and M. Marathe, "PATRIC: A Parallel Algorithm for Counting Triangles in Massive Networks," in *Proc. ACM CIKM*, Oct. 2013, pp. 529–538.

[2] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs," in *Proc. ACM-SIAM SODA*, Jan. 2002, pp. 623–632.

[3] V. Batagelj and M. Zaveršnik, "Short Cycle Connectivity," *Elsevier Discrete Mathematics*, vol. 307, no. 3-5, pp. 310–318, Feb. 2007.

[4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs," in *Proc. ACM SIGKDD*, Aug. 2008, pp. 16–24.

[5] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips, "Tolerating the Community Detection Resolution Limit With Edge Weighting," *Physical Review E*, vol. 83, no. 5, p. 056119, May 2011.

[6] N. Chiba and T. Nishizeki, "Arboricity and Subgraph Listing Algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, Feb. 1985.

[7] S. Chu and J. Cheng, "Triangle Listing in Massive Networks and Its Applications," in *Proc. ACM SIGKDD*, Aug. 2011, pp. 672–680.

[8] J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, Jul.-Aug. 2009.

[9] Y. Cui, D. Xiao, and D. Loguinov, "On Efficient External-Memory Triangle Listing," in *Proc. IEEE ICDM*, Dec. 2016, pp. 101–110.

[10] R. Dementiev, "Algorithm Engineering for Large Data Sets," Ph.D. dissertation, Universität des Saarlandes, 2006.

[11] I. Fudos and C. M. Hoffmann, "A Graph-Constructive Approach to Solving Systems of Geometric Constraints," *ACM Transactions on Graphics*, vol. 16, no. 2, pp. 179–216, Apr. 1997.

[12] I. Giechaskiel, G. Panagopoulos, and E. Yoneki, "PDTL: Parallel and Distributed Triangle Listing for Massive Graphs," in *Proc. IEEE ICPP*, Sep. 2015, pp. 370–379.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed Graph-parallel Computation on Natural Graphs," in *Proc. USENIX OSDI*, 2012, pp. 17–30.

[14] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. Lin, "Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs," *PVLDB*, vol. 7, no. 13, pp. 1379–1380, Aug. 2014.

[15] T. Hocevar and J. Demsar, "A Combinatorial Approach to Graphlet Counting," *Bioinformatics*, vol. 30, no. 4, pp. 559–565, Feb. 2014.

[16] X. Hu, Y. Tao, and C. Chung, "Massive Graph Triangulation," in *Proc. ACM SIGMOD*, Jun. 2013, pp. 325–336.

[17] Z. R. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad, "Kavosh: A New Algorithm for Finding Network Motifs," *Bioinformatics*, vol. 10, no. 318, Oct. 2009.

[18] J. Kim, W. Han, S. Lee, K. Park, and H. Yu, "OPT: A New Framework for Overlapped and Parallel Triangulation in Large-Scale Graphs," in *Proc. ACM SIGMOD*, Jun. 2014, pp. 637–648.

[19] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, A Social Network or a News Media?" in *Proc. WWW*, Apr. 2010, pp. 591–600.

[20] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proc. USENIX OSDI*, 2012, pp. 31–46.

[21] M. Latapy, "Main-memory Triangle Computations for Very Large (Sparse (Power-law)) Graphs," *Elsevier Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 458–473, Nov. 2008.

[22] D. W. Matula and L. L. Beck, "Smallest-Last Ordering and Clustering and Graph Coloring Algorithms," *Journal of the ACM*, vol. 30, no. 3, pp. 417–427, Jul. 1983.

[23] L. A. A. Meira, V. R. Maximo, A. L. Fazenda, and A. F. D. Conceicao, "Acc-Motif: Accelerated Network Motif Detection," *IEEE/ACM Trans. Computational Biology and Bioinformatics*, vol. 11, no. 5, pp. 853–862, Apr. 2014.

[24] B. Menegola, "An External Memory Algorithm for Listing Triangles," Universidade Federal do Rio Grande do Sul, Tech. Rep., 2010.

[25] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network Motifs: Simple Building Blocks of Complex Networks," *Science*, vol. 298, no. 5594, pp. 824–827, Oct. 2002.

[26] M. E. Newman, D. J. Watts, and S. H. Strogatz, "Random Graph Models of Social Networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. Suppl. 1, pp. 2566–2572, Feb. 2002.

[27] R. Pagh and F. Silvestri, "The Input/Output Complexity of Triangle Enumeration," in *Proc. ACM PODS*, Jun. 2014, pp. 224–233.

[28] H. Park and C. Chung, "An Efficient MapReduce Algorithm for Counting Triangles in a Very Large Graph," in *Proc. ACM CIKM*, Oct. 2013, pp. 539–548.

[29] H.-M. Park, S.-H. Myaeng, and U. Kang, "PTE: Enumerating Trillion Triangles On Distributed Systems," in *Proc. ACM SIGKDD*, Aug. 2016, pp. 1115–1124.

[30] T. Schank and D. Wagner, "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study," in *Proc. WEA*, May 2005, pp. 606–609.

[31] M. Sevenich, S. Hong, A. Welc, and H. Chafi, "Fast In-Memory Triangle Listing for Large Real-World Graphs," in *Proc. ACM SNA-KDD*, Aug. 2014, pp. 1–9.

[32] J. Shun and K. Tangwongsan, "Multicore Triangle Computations without Tuning," in *Proc. IEEE ICDE*, Apr. 2015, pp. 149–160.

[33] S. Suri and S. Vassilvitskii, "Counting Triangles and the Curse of the Last Reducer," in *Proc. WWW*, Mar. 2011, pp. 607–614.

[34] N. H. Tran, K. P. Choi, and L. Zhang, "Counting Motifs in the Human Interactome," *Nature Communications*, vol. 4, p. 2241, Aug. 2013.

[35] N. Wang, J. Zhang, K.-L. Tan, and A. K. Tung, "On Triangulation-Based Dense Neighborhood Graph Discovery," *PVLDB*, vol. 4, no. 2, pp. 58–68, Nov. 2010.

[36] D. J. Watts and S. Strogatz, "Collective Dynamics of 'Small World' Networks," *Nature*, vol. 393, pp. 440–442, Jun. 1998.

[37] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *Comput. J.*, vol. 10, no. 1, pp. 85–86, Jan. 1967.

[38] S. Wernicke and F. Rasche, "FANMOD: A Tool for Fast Network Motif Detection," *Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, Feb. 2006.

[39] V. Willians and R. Williams, "Subcubic Equivalences Between Path, Matrix, and Triangle Problems," in *Proc. IEEE FOCS*, Oct. 2010, pp. 645–654.

[40] D. Xiao, Y. Cui, D. Cline, and D. Loguinov, "On Asymptotic Cost of Triangle Listing in Random Graphs," in *Proc. ACM PODS*, May 2017, pp. 261–272.

[41] Yahoo Altavista Graph, 2002. [Online]. Available: http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[42] Z. Yang, C. Wilson, X. Wang, T. Gao, B. Zhao, and Y. Dai, "Uncovering Social Network Sybils in the Wild," in *Proc. ACM IMC*, Nov. 2011, pp. 259–268.