On Efficient External-Memory Triangle Listing

Yi Cui, Di Xiao, and Dmitri Loguinov

***Abstract*—Discovering triangles in large graphs is a well-studied area; however, both external-memory performance of existing methods and our understanding of the complexity involved leave much room for improvement. To shed light on this problem, we first generalize the existing in-memory algorithms into a single framework of 18 triangle-search techniques. We then develop a novel external-memory approach, which we call *Pruned Companion Files* (PCF), that supports operation of all 18 algorithms, while significantly reducing I/O compared to the common methods in this area. After finding the best node-traversal order, we build an implementation around it using SIMD instructions for list intersection and PCF for I/O. This method runs 5-10 times faster than the best available implementation and exhibits orders of magnitude less I/O. In one of our graphs, the program finds 1 trillion triangles in 237 seconds using a desktop CPU.**

I. Introduction

Enormous size of modern datasets poses scalability challenges for a variety of algorithms and applications. One particular area affected by the explosion of big data is *graph mining* and, more specifically, motif discovery in large networks. Motifs are important building blocks of real-life networks in biology, physics, chemistry, sociology, and computer science [14], [16], [26], [27], [39], [42]. They capture *local* composition of graphs and allow reasoning about the underlying construction processes that result in the observed phenomena. Three-node cycles (i.e., triangles) have received the most attention, attracting research interest for over 35 years [20] and developing many applications in graph theory [4], [28], [40], [41], [43], bioinformatics [21], [27], computer graphics [12], databases [3], and social networks [5], [6], [10], [46].

Until recently [44], little was known about the CPU cost of triangle listing, its behavior under different acyclic orientations, and comparison across the different methods. Much of the previous work [1], [17], [24] utilized $O(.)$ bounds that were exactly the same for all involved methods (i.e., vertex/edge iterators). As it turns out [44], there are 18 algorithms for traversing the nodes of a triangle and handling the neighbors, which can be reduced to four equivalence classes from the CPU-cost perspective, each with its own optimal orientation. However, *external-memory* triangle listing remains largely unexplored. Given the same 18 options, how many different I/O classes are there, what node permutations do they require, and is it possible for some methods to simultaneously achieve optimal CPU and I/O complexity using the same orientation?

If m is the number of edges and M is RAM size, previous implementations [2], [13], [17], [22] operate with a simple I/O model that requires reading the graph m/M times, for a total overhead of m^2/M. In theoretical development, better bounds can be achieved using random coloring of the graph [18], [30]; however, there are no implementations that use this method and the constants inside its bound $O(m^{1.5}/\sqrt{M})$ are unknown. What makes these two approaches similar is that their performance does not depend on the traversal order within each triangle or preprocessing manipulations applied to the graph, which leaves little for additional investigation.

Instead, we show below that there exists a technique for graph partitioning that maps the 18 triangle-listing algorithms into six distinct classes, each of which possesses different I/O performance characteristics that depend on the acyclic orientation of the original graph. We call this framework *Pruned Companion Files* (PCF) and demonstrate how all 18 methods can be combined under an umbrella of a single algorithm. Taking into account both I/O and CPU cost [44], we discover 16 unique ways to perform triangle listing in external memory, none of which were known before.

While accurate modeling of I/O complexity is difficult, we are still able to identify the best partitioning scheme, deduce its optimal permutation, and prove that the amount of data read from disk is $\min(m^2/M, O(m))$ in random graphs with Pareto degree sequences, where shape parameter $\alpha > 4/3$. Note that this is the first result with linear I/O bounds under constant memory size. In contrast, both of the previous techniques [17], [30] require M to scale at least as fast as m to achieve the same performance. We also demonstrate that our partitioning scheme keeps the number of list intersections and table lookups unchanged compared to RAM-only methods, which means that its runtime remains constant for all M as long as I/O is not the bottleneck.

To test these developments in practice, we build an implementation that combines PCF with a novel application of SIMD to edge iterator. Our solution, which we call PaCiFier, is benchmarked on a variety of real-world graphs, including four new ones that have not been examined for triangles before. Our densest graph contains over 1T triangles, while the largest has over 100B edges. Results show that PaCiFier is $1-2$ orders of magnitude faster than the best vertex iterator [17] and $5-10$ times faster than the best edge iterator [13]. More importantly, it achieves $10-50$ times lower I/O complexity when RAM size is small compared to m.

II. Generalized Iterators (GI)

Recent work [44] created a taxonomy of 18 vertex and edge iterators. They use figures to highlight the intuitive differences among the methods; however, the lacking formal treatment makes it difficult to extend these results to external-memory scenarios. We therefore introduce a new description framework, which we call *Generalized Iterators* (GI), that

All authors are with the department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843 USA (email: {yicui, di, dmitri}@cse.tamu.edu).

explicitly encodes the traversal order in each triangle. This allows us to parameterize a single algorithm to cover execution of all alternative methods.

A. Redundancy Elimination

Naive triangle-listing algorithms do not enforce order among the neighbors, which results in extremely inefficient operation. Besides discovering each triangle $3! = 6$ times, there are serious repercussions stemming from the fact that the number of pairs checked at each node is a quadratic function of its degree. Even on relatively small graphs, this can lead to $1000\times$ more overhead than necessary [44].

The redundancy can be eliminated by converting the graph into a directed version, in which quadratic complexity applies only to the out-degree (or in-degree, depending on the method), whose second moments are kept significantly smaller than those of undirected degree. Assume the nodes are first shuffled using some algorithm and sequentially assigned IDs from sequence $(1, 2, \ldots, n)$. This creates a total order across the nodes and is often called *relabeling*. A directed graph is then created, where out-neighbors of each node have smaller labels and in-neighbors have larger. This step is called *acyclic orientation*. Finally, in the directed graph, triangles Δ_{xyz} are listed in ascending order of the new labels, i.e., $x < y < z$.

This procedure generalizes all previous efforts in the field, some of which perform only relabeling [24], [34], [36] and others only orientation [2], [13], [17], [22], [34], [37], [38]. The drawbacks of not doing both are discussed in [44].

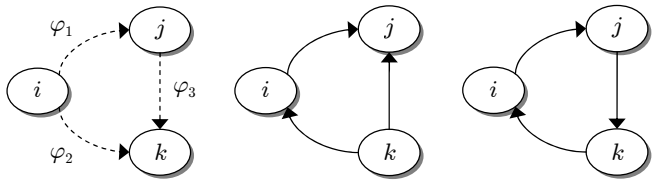
B. Relabeling

Consider a simple (i.e., no self-loops) undirected graph $G = (V, E)$ with n nodes and m edges. Define θ to be a permutation of node IDs that starts with the ascending-degree order and re-writes the label of each node in position i to $\theta(i)$. Among the $n!$ possibilities, there are several named permutations [44], which include *ascending-degree* $\theta_A(i) = i$, *descending-degree* $\theta_D(i) = n + 1 - i$, *round-robin*

$$\theta_{RR}(i) = \begin{cases} \lceil \frac{n+i}{2} \rceil & i \text{ is odd} \\ \lfloor \frac{n-i}{2} \rfloor + 1 & i \text{ is even} \end{cases}, \tag{1}$$

and *complementary round-robin* $\theta_{CRR}(i) = \theta_{RR}(n + 1 - i)$, each of which optimizes a different class of triangle-listing methods [44]. The difference in CPU cost between the best and worst permutations can be orders of magnitude. Even worse, this ratio may be unbounded as $n \to \infty$ [44]. For a given permutation θ, define its *reverse* to be $\theta'(i) = n + 1 - \theta(i)$. This is a useful concept that allows detection of equivalence classes later in the paper.

Suppose G_θ is the relabeled graph under permutation θ. Its construction typically requires sorting the degree sequence of G using θ, re-writing the source nodes of each list, inverting the graph using external memory, and re-writing the source nodes again. It is also common during this process to drop all nodes with degree one since they cannot be part of a triangle.



(a) general case (b) $+--$ (c) $+-+$

Fig. 1. Search-order operators in triangle listing.

C. Orientation

Define N_i to be the adjacency list of node i in G_θ and $d_i = |N_i|$ to be its undirected degree. In general, $i \notin N_i$ because the graph is simple. Suppose the neighbors within each N_i are sorted ascending by their ID and G_θ is kept as a sequence of pairs $\{(i, N_i)\}_{i=1}^n$. Our next goal is to define notation that allows splitting arbitrary sets into values smaller/larger than a given pivot. The most immediate use is construction of in/out lists in the directed graph, but we will encounter other applications shortly.

Suppose $\mathbb{N}$ is the set of natural numbers and consider two finite sets $S, T \subseteq \mathbb{N}$. Then, let

$$(T, S)^+ = \{j \in S | j \leq \max(T)\} \tag{2}$$

be a subset of S that is bounded from above by the largest value in T. When T consists of a single element i, we simply write $(i, S)^+$. Similarly, define

$$(T, S)^- = \{j \in S | j \geq \min(T)\} \tag{3}$$

to contain elements of S no smaller than the minimum in T. Then, the out-list of i in the oriented graph is given by $N_i^+ := (i, N_i)^+$, while the corresponding in-list by $N_i^- := (i, N_i)^-$.

When the $+/-$ operator is specified by a variable φ, i.e., $(T, S)^\varphi$, we say that *S is φ-oriented by T*. This notation can be extended to other graph concepts. For example, G_θ^φ consists of tuples $\{(i, N_i^\varphi)\}$, where i is the source node and N_i^φ is its neighbor list, and $d_i^\varphi := |N_i^\varphi|$ is the corresponding degree in the directed graph. Define $1 - \varphi$ to be the *inverse* of operator φ, i.e., a plus becomes a minus and vice versa. It is then not difficult to see that $G_{\theta'}^\varphi$ is identical to $G_\theta^{1-\varphi}$, i.e., reversing the permutation is equivalent to inverting the orientation.

D. Search Order

Given six different ways to permute the nodes of a triangle, we next show how φ allows us to describe the various trajectories during search that result in exactly one listing of each triangle. Suppose i is the first visited node by an algorithm, $j \in N_i$ is the second, and $k \in N_i$ is the last one. The larger/smaller relationship between these nodes is what differentiates the various traversal orders. All possible combinations are captured by Fig. 1(a), where each dashed arrow represents a φ-relationship between the two neighboring nodes. If labeled with a plus, a dashed arrow indicates that the source node is *larger* than the destination. The roles are reversed when the label is a minus. Note that unlike our earlier notation Δ_{xyz}, where the order $x < y < z$ was fixed, the

Algorithm 1: Generalized vertex iterator

```
1  Function GVI (φ̄)
2      build hash table H with all directed edges from G_θ^{φ3}
3      for i = 1 to n do
4          X = (i, N_i)^{φ1}  ◁ neighbors of i in G_θ^{φ1} (hit list)
5          Y = (i, N_i)^{φ2}  ◁ same in G_θ^{φ2} (local list)
6          foreach j ∈ X do
7              Y' = (j, Y)^{φ3}  ◁ set Y φ3-oriented by j
8              foreach k ∈ Y' do
9                  if (j, k) ∈ H then report triangle Δ_{sort(ijk)}
```

Algorithm 2: Generalized lookup edge iterator

```
1  Function GLEI (φ̄)
2      for i = 1 to n do
3          X = (i, N_i)^{φ1}  ◁ neighbors of i in G_θ^{φ1} (hit list)
4          Y = (i, N_i)^{φ2}  ◁ same in G_θ^{φ2} (local list)
5          add elements of Y to hash table H
6          foreach j ∈ X do
7              Z = (j, N_j)^{φ3}  ◁ neighbors of j in G_θ^{φ3} (remote list)
8              Z' = (i, Z)^{φ2}  ◁ set Z φ2-oriented by i
9              foreach k ∈ Z' do
10                 if k ∈ H then report triangle Δ_{sort(ijk)}
11         empty H
```

Algorithm 3: Generalized scanning edge iterator

```
1  Function GSEI (φ̄)
2      for i = 1 to n do
3          X = (i, N_i)^{φ1}  ◁ neighbors of i in G_θ^{φ1} (hit list)
4          Y = (i, N_i)^{φ2}  ◁ same in G_θ^{φ2} (local list)
5          foreach j ∈ X do
6              Z = (j, N_j)^{φ3}  ◁ neighbors of j in G_θ^{φ3} (remote list)
7              Y' = (j, Y)^{φ3}  ◁ set Y φ3-oriented by j
8              Z' = (i, Z)^{φ2}  ◁ set Z φ2-oriented by i
9              K = Intersect (Y', Z')
10             foreach k ∈ K do report triangle Δ_{sort(ijk)}
```

relationship between (ijk) is fluid, i.e., changed by parameter $\bar{\varphi} = (\varphi_1, \varphi_2, \varphi_3)$.

Once the $\bar{\varphi}$ vector is chosen, the dashed arrows become oriented and are replaced with solid lines that specify greater-than relationships among the nodes. One example is shown in Fig. 1(b), where $k > i > j$. A simple rule to remember is that a $+$ keeps the direction of the dashed arrow, while a $-$ reverses it. Out of the $2^3 = 8$ possible $\bar{\varphi}$ vectors, two produce loops, such as the one in Fig. 1(c). These are invalid because they lead to a contradiction, e.g., $k > i > j > k$. The remaining six combinations are studied next.

E. Algorithms

In Algorithm 1, we create the *generalized vertex iterator* (GVI) that can handle all valid $\bar{\varphi}$ vectors. The method starts by populating all directed edges from $G_\theta^{\varphi_3}$ into a hash table. The reason for using φ_3 is that the algorithm performs lookups of (j, k) against H, which we know from Fig. 1(a) have relationship φ_3. Then, for each node i, GVI creates two sets – the *hit list* X, from which j will be drawn, and the *local list* Y consisting of neighbors k that may complete a triangle. From Line 6, the algorithm examines every node $j \in X$, orients Y using φ_3 with respect to j, and checks the resulting pairs (j, k) against the hash table. Note that Line 7 is important for eliminating the possibility of redundancy.

The next technique is the *generalized lookup edge iterator* (GLEI) whose operation is presented in Algorithm 2. The main difference begins in line 5, where GLEI populates the local list Y into a small hash table H. For each $j \in X$, the method constructs a *remote list* Z consisting of j's neighbors according to φ_3, orients it by φ_2 with respect to i, and checks its members against H. GLEI and GVI perform the same number of memory hits [44], with the only difference being the time needed to clear the hash table in Line 11.

The last method is the *generalized scanning edge iterator* (GSEI), which is described by Algorithm 3. It relies on sequential traversal of neighbor lists to perform set intersection in Line 9. This is in contrast to GLEI that uses hash tables for this purpose. The rest of the algorithm is quite similar. Before intersecting local and remote lists (Y, Z), the method orients them in Lines 7-8 to be consistent with Fig. 1(a). Note that the former is done by GVI and the latter by GLEI. In practice, orientation of the local list Y imposes no additional overhead since j monotonically increases within the loop, which is a consequence of $N_i^{\varphi_1}$ being sorted ascending. However, certain GSEI traversal orders require a binary search in the remote list Z to locate i [44].

TABLE I
TAXONOMY OF VERTEX/EDGE ITERATORS

GVI	GLEI	GSEI	Binary Search	Vector $\bar{\varphi}$	i	j	k
T_1	L_1	E_1	No	$++$	z	y	x
T_2	L_2	E_2	No	$-+$	y	z	x
T_3	L_3	E_3	No	$---$	x	y	z
T_4	L_4	E_4	No	$+-$	z	x	y
T_5	L_5	E_5	Yes	$+--$	y	x	z
T_6	L_6	E_6	Yes	$--+$	x	z	y

F. Taxonomy

A combination of Algorithms 1-3 comprises our *Generalized Iterators* (GI) framework. Analysis above shows that each of the main algorithms (i.e., GVI, GLEI, GSEI) admits six traversal orders and that this classification is exhaustive (i.e., no other patterns are possible). Table I assigns names to all methods based on their $\bar{\varphi}$, specifying whether the edge iterators require a binary search and how to relate (ijk) to (xyz). In prior literature, T_1 can be found in [17], [22], [38], E_1 in [2], [13], [37], E_2 in [24], [34], E_3 in [7], [8], and E_5 in [36]. Methods T_1-T_3, E_1, E_3, E_4 are listed in [29].

While there are 18 techniques total, their CPU cost can be reduced to just four non-isomorphic classes [44]; however, this may no longer hold when I/O is taken into account. What can be said for sure is that reversing θ, or similarly inverting $\bar{\varphi}$, produces an identical method from the I/O standpoint. This allows reduction of scope to a subset of methods that cannot be converted into each other through inversion of $\bar{\varphi}$.

For example, keeping only methods that utilize G_θ^+ for remote edges, i.e., φ_3 is the plus operator, would eliminate rows $(3, 4, 5)$ in Table I. In that case, Fig. 2 shows the position of the remaining 9 methods on a 2D plane, where the columns share the CPU cost, while the rows do the same for I/O. We use analysis from [44] to position the columns in order of increasing CPU complexity, with T_1 being the best and E_6

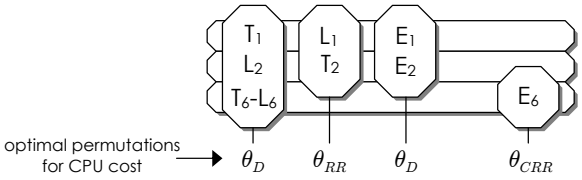


Fig. 2. Four CPU and three I/O classes.

being the worst; however, it is currently unknown if the rows do in fact differ in cost, whether they can be split into multiple subrows depending on additional factors, and how their I/O relates to each other. This is our next topic.

III. Pruned Companion Files (PCF)

This section presents a general family of disk-based algorithms that supports all of the methods in Table I. It also aims to achieve better I/O complexity than prior approaches.

A. Overview

It is important to discuss the performance objectives of external-memory algorithms before explaining our solution. There are four metrics that contribute towards the runtime of a method and its ability to handle large graphs. The first is the *triangle-identification time*, which consists of lookups against H in GVI/GLEI and intersection in GSEI (i.e., Lines 9, 10, 9, respectively). For a method $\mathcal{M}$, suppose $c_n(\mathcal{M}, \theta)$ is the number of elementary operations, which we call the *CPU cost*, and $r(\mathcal{M})$ is the speed of these operations in nodes/sec. For a fixed pair (i, j), the CPU cost equals $|Y'|$ for GVI, $|Z'|$ for GLEI, and $|Y'| + |Z'|$ for GSEI. Then, the triangle-identification time is given by $c_n(\mathcal{M}, \theta)/r(\mathcal{M})$.

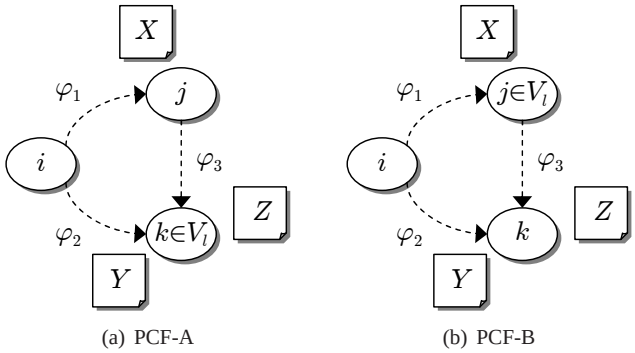
The second metric is the amount of I/O performed. Because all reads are sequential, this overhead is measured by the length of adjacency lists across all graphs participating in the algorithm. The third metric is the *number of lookups based on hit list* X (i.e., Lines 6, 6, 5), which is generally a function of the partitioning scheme. This is in contrast to RAM-only operation, where this value is always fixed at m, i.e., the number of edges in G_θ. Finally, the last parameter is the *minimum amount of RAM supported by the method*.

It is possible that some of these metrics are tradeoffs of each other; however, if an ideal algorithm exists, it would simultaneously beat the other methods in all four categories.

B. Graph Partitioning

Because GSEI explicitly maintains remote and local lists, both GVI and GLEI can be viewed as its special cases that replace one of the lists with a hash table. For example, GVI uses H in place of scanning Z, while GLEI does the same for scanning of Y. As a result, any I/O partitioning scheme that handles GSEI can be adopted to work with the other two algorithms without incurring additional overhead. Therefore, our description of I/O techniques targets Algorithm 3.

In general, triangle-partitioning schemes work by placing one (or more) edges in some RAM buffer and then scanning the disk for discovery of the remaining edges that complete each triangle. Since node j and its neighbors k must be retrieved using random access, one crucial observation is that all methods require the *remote* edge (j, k) to be present in RAM, while the other two lists (X, Y) may be streamed from disk sequentially. This framework, coupled with general φ and the algorithms developed in this section, is what we call *Pruned Companion Files* (PCF).



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

Fig. 3. Graph partitioning.

Assume the set of nodes V is divided into p pair-wise non-overlapping and jointly exhaustive sets $\mathbf{V} = (V_1, \ldots, V_p)$. In a method we call PCF-A, we split $G_\theta^{\varphi_3}$ along the destination node of each pair $(j, N_j^{\varphi_3})$ to create a set of *remote-edge graphs*

$$G_\theta^r(l) = \{(j, N_j^{\varphi_3} \cap V_l)\}, \tag{4}$$

where $l = 1, 2, \ldots, p$. In a method we call PCF-B, we do the same along the source nodes

$$G_\theta^r(l) = \{(j, N_j^{\varphi_3}) | j \in V_l\}. \tag{5}$$

These technique are illustrated in Fig. 3 and their properties are given by the next result.

Theorem 1: Algorithms 1-3 operating over PCF-A/B find each triangle exactly once. Furthermore, the triangle-identification cost $c_n(\mathcal{M}, \theta)$ remains constant for all p.

Proof: First notice that every edge (j, k) belongs to a unique partition $G_\theta^r(l)$. Then, replacing $G_\theta^{\varphi_3}$ with $G_\theta^r(l)$ in Algorithms 1-3 and repeating for all $l = 1, 2, \ldots, p$, we immediately obtain that no triangle is missed or counted more than once.

To show that the triangle-counting overhead remains constant, we focus on GSEI, with the other methods being similar. Fix a node j and assume the length of its neighbor list Z after orientation by node i in Line 8 is given by q_{ij}. Note that list Y' is independent of the partitioning scheme and can be ignored. For RAM-only operation, the intersection cost related to j can be expressed as

$$\sum_{(i,j) \in G_\theta^{\varphi_1}} q_{ij}. \tag{6}$$

In PCF-A, assume the length of Z oriented by i in partition l is given by $q_{ij}(l)$. This leads to an overall cost for j

$$\sum_{l=1}^{p} \sum_{(i,j) \in G_\theta^{\varphi_1}} q_{ij}(l). \tag{7}$$

Since the partitions are mutually disjoint and exhaustive, it must be that for all i

$$\sum_{l=1}^{p} q_{ij}(l) = q_{ij}, \tag{8}$$

which yields the same cost in (7) as in (6) after changing the order of summations.

In PCF-B, the analysis is even simpler. Because j appears as the source node in exactly one partition, it experiences the same overhead (6) in that partition and zero in all others. ■

This result shows that partitioning does not create any additional list-intersection operations, which allows us to focus on the remaining three objectives in the rest of the paper.

C. Partition Balancing

Assume M is the RAM size. To achieve the smallest p, each partition size $|G_\theta^r(l)|$ must equal M, which requires explicit balancing. Note that splitting the range $[1, n]$ into $p = m/M$ equal-size bins fails to accomplish this objective since permutation θ is degree-dependent. For example, with θ_D, smaller node IDs indicate larger degree. Therefore, nodes in the first bin may bring significantly more (or less depending on φ_3) edges into $G_\theta^r(l)$ than those in the last bin.

Balancing is accomplished by setting up boundaries $a_1, a_2, \ldots, a_{p+1}$ such that a node is included in V_l if and only if it belongs to $[a_l, a_{l+1})$. While $a_1 = 1$ and $a_{p+1} = n+1$ are obvious, the other values require more attention. For PCF-A in Fig. 3(a), notice that inclusion of k into V_l implies that all edges from list $N_k^{1-\varphi_3}$ are placed into $G_\theta^r(l)$. Therefore, we must select the boundaries such that

$$\sum_{k=a_l}^{a_{l+1}-1} d_k^{1-\varphi_3} = M, \tag{9}$$

which can be accomplished in one pass over $G_\theta^{1-\varphi_3}$. For PCF-B in Fig. 3(b), the roles of j, k are reversed, which leads to

$$\sum_{j=a_l}^{a_{l+1}-1} d_j^{\varphi_3} = M. \tag{10}$$

Balancing in PCF-A and B is equally fast, except the former requires existence of an inverted version of $G_\theta^{\varphi_3}$.

D. Companion Files

The fastest previous implementations [2], [13], [17], [22] use a framework that would scan the entire file $G_\theta^{\varphi_1}$ to obtain hit lists X and $G_\theta^{\varphi_2}$ for local lists Y. When $\varphi_1 = \varphi_2$, these files coincide, which cuts the overhead by half compared to other vectors $\bar{\varphi}$. Nevertheless, the amount of I/O produced by these schemes is still quite substantial, i.e., $mp = m^2/M$. Instead, our approach is to prune lists X, Y to be optimally suited for each partition l and write them into special *companion* files $G_\theta^c(l)$. Each of them, when paired with the corresponding remote-edge graph $G_\theta^c(l)$, allows identification of all triangles with either k (PCF-A) or j (PCF-B) in V_l.

Consider Algorithm 4, which is our one-pass solution to creating both companion and remote-edge files. If tuples $\{(i, N_i)\}$ are sorted by the source node i, Lines 3-5 simultaneously construct the three lists (X, Y, Z) by scanning multiple files in parallel; otherwise, only methods with $\varphi_1 = \varphi_2 = \varphi_3$ are supported. In Lines 7-14, the algorithm prepares the necessary lists for each partition l. Among these, Line 8 can be explained with the help of Fig. 3(a). Notice that PCF-A can $(1 - \varphi_3)$-orient set X with respect to V_l without losing any relevant nodes j. Similarly Line 13 uses an observation from Fig. 3(b) that PCF-B can φ_3-orient Y with respect to V_l without omitting any essential nodes k.

Algorithm 4: One-pass graph partitioning

```
1  Function PartitionGraph (method, φ̄, V)
2      for i = 1 to n do
3          X = (i, N_i)^{φ1}  ◁ hit list from G_θ^{φ1}
4          Y = (i, N_i)^{φ2}  ◁ local list from G_θ^{φ2}
5          Z = (i, N_i)^{φ3}  ◁ remote list from G_θ^{φ3}
6          for l = 1 to p do ◁ go through each partition
7              if method = PCF-A then
8                  X = (V_l, X)^{1−φ3}  ◁ hit list oriented by V_l
9                  Y = Y ∩ V_l  ◁ keep only nodes in V_l
10                 Z = Z ∩ V_l  ◁ keep only nodes in V_l
11             else
12                 X = X ∩ V_l  ◁ keep only nodes in V_l
13                 Y = (V_l, Y)^{φ3}  ◁ local list oriented by V_l
14                 Z = Z · 1_{i∈V_l}  ◁ Z if i ∈ V_l and ∅ otherwise
15             Y' = Y  ◁ local list to be written to G_θ^c(l)
16             if Z ≠ ∅ then
17                 write record (i, Z) into G_θ^r(l)
18                 if φ1 = φ3 then
19                     X = X\Z  ◁ further prune X
20                 if φ2 = φ3 then
21                     Y' = Y\Z  ◁ further prune Y
22             if X ≠ ∅ and Y ≠ ∅ and |X ∪ Y| ≥ 2 then
23                 write record (i, X, Y') to G_θ^c(l)
```

In Lines 18-19, where $\varphi_1 = \varphi_3$ indicates that sets X and Z may overlap, the algorithm drops redundant edges from X. The same operation applies to Y in Lines 20-21. Finally, the companion file receives triple (i, X, Y') if both hit list X and local list Y are non-empty, and there exist at least two nodes $j \in X$ and $k \in Y$ such that $j \neq k$.

Note that when $\varphi_1 = \varphi_2$, it is possible for X to overlap with Y. An important aspect of these cases is that Y is always φ_3-oriented against X. If additionally $Y' \neq \emptyset$, either $X \subseteq Y'$ or $Y' \subseteq X$ holds. Not only that, but the smaller list is always either at the bottom or top of the larger one. In such cases, only their union $X \cup Y'$ is written to disk, with an additional field indicating the offset that separates them. Algorithm 4 omits this detail to prevent clutter, but actual implementations should take it into account.

The main search function is shown in Algorithm 5. One noteworthy aspect is Line 8, which handles X being in RAM for PCF-A, and Line 10, which does the same for PCF-B. In the latter case, only nodes $j \in V_l$ should be included in the hit list, which explains the need for additional pruning. Since X being in RAM implies that Y is too, Line 11 uses $N_i(l)$ as the local list. Processing of individual nodes is given by Algorithm 6, which is identical to the corresponding section of GSEI, except it finds Z via the hash table rather than from the full graph $G_\theta^{\varphi_3}$.

Algorithm 5: Disk-based GSEI

```
Function FindTriangles (φ̄)
    for l = 1 to p do
        load G_θ^r(l) = {(i, N_i(l))} in RAM
        build hash table H to map each i to its neighbor list N_i(l)
        if φ_1 = φ_3 then ◁ possible for parts of X to be in RAM
            foreach (i, N_i(l)) in RAM do
                if method = PCF-A then
                    X = N_i(l) ◁ unrestricted hit list
                else
                    X = N_i(l) ∩ V_l ◁ restrict hit list to V_l
                ProcessOneNode (φ̄, i, X, N_i(l))
        while not EOF(G_θ^c(l)) do
            read one record (i, X, Y) from companion G_θ^c(l)
            if Y = ∅ then
                Y = H.find(i) ◁ local list must be in RAM
            ProcessOneNode (φ̄, i, X, Y)
        empty H
```

Algorithm 6: Modified GSEI intersection

```
Function ProcessOneNode (φ̄, i, X, Y)
    foreach j ∈ X do
        Z = H.find(j)  ◁ remote list is always in RAM
        Y' = (j, Y)^{φ_3}  ◁ set Y φ_3-oriented by j
        Z' = (i, Z)^{φ_2}  ◁ set Z φ_2-oriented by i
        K = Intersect (Y', Z')
        foreach k ∈ K do report triangle Δ_{sort(ijk)}
```

IV. Analysis

This section examines the introduced methods in comparison to each other. Our objective is to select a technique and its permutation so as to simultaneously maximize performance across all four criteria, if possible.

A. Overview

From this point on, we parameterize PCF with a specific $\bar{\varphi}$ from Table I by adding the corresponding row index. As before, we consider only rows $1, 2, 6$. When the A/B designation is non-essential, we omit it. For example, PCF-2 refers to $\bar{\varphi} = (-++)$ under both A/B, while PCF-2A narrows it down to the A partitioning scheme.

This creates the six I/O mechanisms in Table II, where $i \to j$ signifies the out-list neighbor relationship, i.e., $j \in N_i^+$, and $i \leftarrow j$ the opposite, i.e., $j \in N_i^-$. Note that PCF-1A and 2A place two edges in RAM and load the third one from disk. This explains why their local list Y is always omitted from companion files. The remaining four techniques do the opposite – one edge is contained in $G_\theta^r(l)$ and two in $G_\theta^c(l)$. In three of these cases, edge direction is kept the same between X and Y, which ensures that either $X \subseteq Y'$ or $Y' \subseteq X$, with only one of them actually written to disk. Method PCF-2B is the lone exception with its $X \cap Y' = \emptyset$.

Table III summarizes the pruning rules and specifies the contents of each companion list. Notice that PCF-1B uses stricter conditions for achieving $X, Y \neq 0$ than PCF-1A and its $X \cup Y'$ is the same or smaller, which indicates that it out-performs its counterpart. Assuming θ_D, further scrutiny of companion lists in Table III reveals that PCF-1A produces less I/O than any of the remaining four methods, with PCF-6A/6B being essentially identical to each other.

TABLE II
Summary of PCF Algorithms Using Remote Graph G_θ^+

PCF	$G_\theta^r(l)$	Condition	X	Y'
1A	$(y,z) \to x$	$x \in V_l$	$z \to y$	$\emptyset$
2A	$(y,z) \to x$	$x \in V_l$	$y \leftarrow z$	$\emptyset$
6A	$z \to y$	$y \in V_l$	$x \leftarrow z$	$x \leftarrow y$
1B	$y \to x$	$y \in V_l$	$z \to y$	$z \to x$
2B	$z \to x$	$z \in V_l$	$y \leftarrow z$	$y \to x$
6B	$z \to y$	$z \in V_l$	$x \leftarrow z$	$x \leftarrow y$

TABLE III
Composition of Companion Lists in PCF

PCF	X	Y	Y'
1A	$N_i^+ \cap [a_{l+1}, n]$	$N_i^+ \cap V_l$	$\emptyset$
1B	$(N_i^+ \cap V_l) \cdot \mathbf{1}_{i \geq a_{l+1}}$	$N_i^+ \cap [1, a_{l+1})$	Y
2A	N_i^-	$N_i^+ \cap V_l$	$\emptyset$
2B	$N_i^- \cap V_l$	N_i^+	$Y \cdot \mathbf{1}_{i \notin V_l}$
6A	$N_i^- \cap [a_l, n]$	$N_i^- \cap V_l$	Y
6B	$N_i^- \cap V_l$	$N_i^- \cap [1, a_{l+1})$	Y

B. Modeling I/O

Additional insight can be gleaned from bounding the size of companion files. Assume u_{il} is the length of i's hit list X' in $G_\theta^c(l)$ and v_{il} is that of $Y' \backslash X'$. Then, the total amount of companion I/O (in edges) is $H^c = H_X^c + H_Y^c$, where

$$H_X^c = \sum_{i=1}^{n} \sum_{l=1}^{p} u_{il}, \quad H_Y^c = \sum_{i=1}^{n} \sum_{l=1}^{p} v_{il}, \tag{11}$$

and that for remote-edge graphs is

$$H^r = \sum_{i=1}^{n} |G_\theta^r(l)| = m. \tag{12}$$

Since H^r is constant for all $\bar{\varphi}$, comparison across the various approaches in Table II needs to involve only H^c. Closed-form derivation of accurate models for (11) currently appears intractable. Even ballparking the scaling rate is quite elusive for certain extremely heavy-tailed degree distributions [44]. Instead, we offer bounds achievable in two worst-case scenarios and leave more precise modeling for future work. Assume $H^c(k)$ refers to the companion overhead of PCF-k and consider the next result.

Theorem 2: The PCF I/O complexity (in edges) is upper-bounded by

$$H^c(1) \leq \sum_{i=1}^{n} \min\Big(\frac{d_i^+ - 1}{2}, p-1\Big) d_i^+, \tag{13}$$

$$H^c(2) \leq \sum_{i=1}^{n} \min(d_i^+, p) d_i^-, \tag{14}$$

$$H^c(6) \leq \sum_{i=1}^{n} \min\Big(\frac{d_i^- + 1}{2}, p\Big) d_i^-, \tag{15}$$

where d_i^+ is the out-degree of i and d_i^- is the in-degree.

Proof: We only consider PCF-A since PCF-B uses similar arguments and produces the same bounds. It is not difficult to see that PCF-1A writes $H^c = H_X^c$ edges to companion files since its pruned hit lists Y' are always empty. First, notice that a list cannot be split into more than p chunks. Due to removal

of overlap $X \cap Z$, we can do even better – the last partition V_p produces a hit list X only for neighbors $j \geq a_{p+1} = n+1$. Since no label can exceed n, there are actually at most $p-1$ partitions where $u_{il} \neq 0$. Therefore, $\sum_{l=1}^{p} u_{il} \leq (p-1)d_i^+$.

Our second observation is that an out-list cannot be split into more than d_i^+ files. Then, the worst case arises when each V_l consists of a single node, where partition l contains the largest $d_i^+ - l$ out-neighbors of i. Thus,

$$\sum_{l=1}^{p} u_{il} \leq \sum_{l=1}^{d_i^+} u_{il} \leq \sum_{l=1}^{d_i^+} (d_i^+ - l) = \frac{d_i^+(d_i^+ - 1)}{2}, \qquad (16)$$

which combined with the first case yields (13).

For PCF-2A, the first case is very similar, except it uses the in-degree d_i^- and fails to remove the overlap $X \cap Z$. The second case writes the full in-neighbor list exactly d_i^+ times, which yields the result in (14). Finally, PCF-6A operates similar to 1A, except it uses the in-degree and fails to prune the lists as efficiently. Due to these small differences, its bound (15) is not perfectly symmetrical to (13). ■

Using [44], we obtain that the I/O bound of PCF-1 is minimized by the descending-degree permutation θ_D, that of PCF-2 by round-robin θ_{RR}, and that of PCF-6 by ascending-degree θ_A. Furthermore, under their respective optimal permutations, (14) is strictly worse than (13). The bound of PCF-6 under θ_A rivals that of PCF-1 under θ_D, although it is still slightly higher due to a less-efficient pruning of overlap between (X, Y) and Z. The worst permutations corresponding to (13)-(15) are θ_A, θ_{CRR} and θ_D, respectively [44].

For the asymptotics, let $D_n \sim F_n(x)$ be the random degree of a node in a graph of size n. As $n \to \infty$, suppose $F_n(x) \to F(x)$ and let $D \sim F(x)$. Then, under θ_D and $E[D^{4/3}] < \infty$, the scaling rate of (13) is *no worse than linear* [44]

$$H^c(1) \leq \min\Big(\frac{m^2}{M}, O(m)\Big). \qquad (17)$$

For example, Pareto distributions $F(x) = 1 - (1 + x/\beta)^{-\alpha}$ satisfy this requirement iff $\alpha > 4/3$. For PCF-2 and θ_{RR}, the rate (17) holds iff $\alpha > 1.5$ [44]. Note that (17) is strictly better than m^2/M from prior implementations [2], [13], [17], [22]. When M is a constant, it is also better than theoretical results of [18], [30] whose $O(m^{1.5}/\sqrt{M})$ bound cannot be linear unless M grows at least as fast as m.

Based on Table III, Theorem 2, and symmetry of PCF-1A/6B and 1B/6A, Fig. 4 places the I/O of the various methods in relationship to each other under different permutations. When we do not differentiate between the PCF variants A/B of a given method, it is usually because they have similar I/O. From the picture, it emerges that PCF-1B with θ_D is globally the most efficient technique.

C. I/O Comparison

For an illustration of the ideas presented earlier in this section, we employ the commonly considered Twitter graph [23] with 41M nodes and $m = 1.2$B edges. The file occupies 9.3 GB and its adjacency lists contain $2m = 2.4$B node IDs. We start with Table IV, which shows the size of companion files H^c. Observe that the predicted best-case permutations

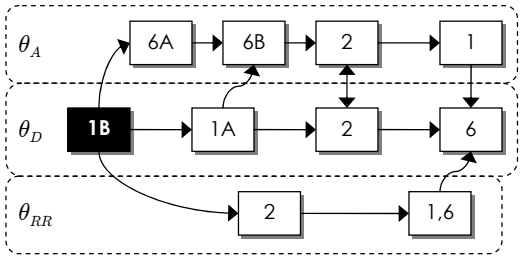


Fig. 4. Better-than relationships across the I/O of various PCF methods.

TABLE IV
TWITTER I/O (IN BILLION EDGES) UNDER 16 MB OF RAM

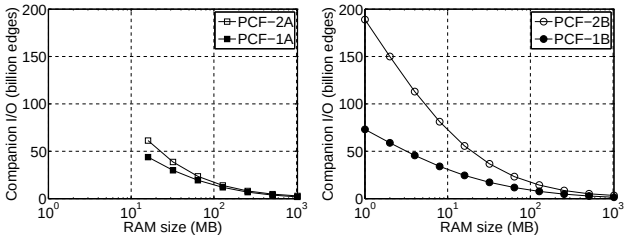
Permutation	1A	1B	2A	2B	6A	6B
θ_D	43.8	24.5	61.3	55.6	119.1	126.8
θ_{RR}	94.1	83.0	51.0	51.7	83.6	94.2
θ_A	125.7	118.4	54.8	61.7	25.5	44.2

in each column (highlighted in gray) agree with earlier analysis. Additionally, notice that reversal of θ swaps PCF-A/B, switches PCF-1 to PCF-6, and maps PCF-2 back to itself. These effects were expected based on (13)-(15). Even though PCF-1 and PCF-6 are close under their optimal permutations, the former comes out ahead for the reasons discussed above.

We now examine how the methods scale as $M \to 0$. We dismiss PCF-6 due to its similarity to PCF-1. We also fix θ_D since it achieves the best CPU cost among the methods in Fig. 2. We vary RAM size from 1 GB down to 1 MB and plot the result in Fig. 5, where PCF-A cannot go lower than 16 MB due to inability to fit the largest in-degree into RAM. Observe that not only is PCF-1 more efficient than PCF-2, but the gap between the two grows as M decreases. As $M \to 0$ and $p \to \infty$, both methods converge towards their upper bounds, which are 150B in (13) and 360B in (14) [44], the figure shows that PCF-1 is getting there at a slower pace than PCF-2.

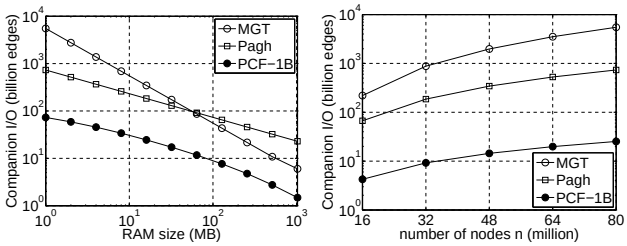
We next analyze the scaling rate of our best method PCF-1B against the two previous models of I/O. Recall that the m^2/M technique was proposed by MGT [17], while the $O(m^{1.5}/\sqrt{M})$ bound is due to Pagh *et al.* [30]. Since there is no actual implementation for the latter, it is difficult to assess the constants inside $O(.)$. We thus take some liberty in assuming how this method would work in practice. It randomly colors the nodes using $c = \sqrt{m/M}$ unique values and splits the edges into c^2 files based on the color of source/destination nodes. It then combines three files of colors (ij, jk, ki) and runs MGT over the result. Since the size of each combined subgraph is $3m/c^2$, the I/O cost of the method is $9m^{1.5}/\sqrt{M}$, which accounts for all c^3 combinations of triplets (ij, jk, ki). While [30] deals with undirected graphs, whose size is $\sum_{i=1}^{n} d_i = 2m$ edges, we assume the method can be applied to G_θ^+. Thus, both MGT and Pagh use $m = 1.2$B in their respective models.

The result for Twitter and $M \to 0$ is shown in Fig. 6(a). After the initial jump, PCF-1B becomes parallel to Pagh's curve $1/\sqrt{M}$. Both of them scale significantly better than MGT's inverse linear function. In Fig. 6(b) we use random graphs with a Pareto degree distribution ($\alpha = 1.5$, $E[D] = 30$) to examine the scaling rate of I/O as $n \to \infty$. In this range,



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

Fig. 5. Scaling rate of PCF-1 on Twitter under θ_D.



(a) Twitter (b) random graphs ($M = 1$ MB)

Fig. 6. Comparison against prior methods.

PCF-1B is roughly linear, while the other two methods grow significantly faster. As n increases, the ratio of MGT to PCF-1B jumps from 51 to 219, while that for Pagh from 15.5 to 29.3. To put this in perspective, $n = 80$M nodes requires 25B edges of I/O for PCF-1B, 734B for Pagh, and 5.5T for MGT.

D. CPU-I/O Tradeoffs

As it turns out, Fig. 2 splits into 16 different CPU-I/O complexity classes, i.e., two (A/B) for each of the 8 unique GI methods, with T_6-L_6 being a single entity. In the past, it was believed that GVI and GLEI were functionally identical. However, this is not the case when I/O is taken into account. For example, T_1 shares the I/O cost with L_1, but at lower CPU complexity. Similarly, it shares the CPU cost with L_2-L_6, while imposing less I/O. In the same vein, it was unknown until now whether E_1 and E_2 were interchangeable. Results above confirm that they are not.

These observations are emphasized using Table V, where each I/O cell reports the best number achieved by either PCF-A or B. Observe that the best GVI is T_1, which exhibits optimal CPU and I/O complexity under θ_D. The decision is also easy for GSEI, where E_1 is the top contender. On the other hand, GLEI must choose which of the two objectives is more important – L_1 has the best I/O and L_2 the best CPU cost, both under θ_D. Other GLEI combinations are much worse.

E. Lookups and Minimum RAM

Recalling that PCF-B prunes X such that $X \subseteq V_l$ holds, while PCF-A does not, the next result follows immediately.

Theorem 3: PCF-A issues H_X^c hit-list lookups and requires $M \geq \max_i d_i^{1-\varphi_3}$. PCF-B performs exactly m lookups and requires $M \geq \max_i d_i^{\varphi_3}$.

In graphs with heavy-tailed degree and $M \ll m$, it is common that the hit list size $H_X^c \gg m$ (e.g., see Table IV). Therefore, for small RAM size, PCF-B should have a noticeably better CPU performance than PCF-A. In fact, its number of hash-table hits is optimal as it equals that in RAM-only algorithms.

TABLE V
CPU-I/O COMPLEXITY CLASSES IN TWITTER UNDER 16 MB OF RAM

Under CPU-optimal permutation				Under I/O-optimal permutation			
Perm	GI	CPU	I/O	Perm	GI	CPU	I/O
θ_D	T_1	150B	24B	θ_D	T_1	150B	24B
	L_2	150B	56B		L_1	360B	24B
	T_6-L_6	150B	119B		E_1	511B	24B
	E_1	511B	24B	θ_{RR}	T_2	255B	51B
	E_2	511B	56B		L_2	63T	51B
θ_{RR}	L_1	255B	83B		E_2	63T	51B
	T_2	255B	51B	θ_A	T_6-L_6	123T	25B
θ_{CRR}	E_6	63T	45B		E_6	123T	25B

In terms of restrictions on RAM, all considered methods PCF-1/2/6 have a plus for φ_3, which means that PCF-A lower-bounds M by the largest *in-degree*, while PCF-B by the largest *out-degree*. It is well-known that θ_D keeps the latter no larger than $\sqrt{2m}$; however, its maximum in-degree equals $\max_i d_i$, which can be significantly higher, i.e., up to $n-1$. Therefore, PCF-B under θ_D is definitively less restrictive than PCF-A. When the permutation is reversed, the bounds on in/out degree are swapped and PCF-A becomes better than PCF-B. Finally, θ_{RR} has both maximum in/out degree equal to $\max_i d_i$, which makes this permutation equally bad in both PCF-A/B.

F. Summary

From the analysis above, two methods T_{1B} and E_{1B} emerge as clear winners within their respective classes (i.e., hash tables and scanning intersection). Among the 18 methods, they achieve the smallest companion I/O, perform the minimal number of hit-list lookups, impose the lowest RAM requirements, do not need to invert $G_\theta^{\varphi_3}$ during creation of $\{V_l\}$, and obtain (X, Y, Z) from a single file in Algorithm 4.

We next consider which of them has a smaller runtime. There are two aspects involved – the relative CPU cost

$$w_n := \frac{c_n(E_1, \theta_D)}{c_n(T_1, \theta_D)} \tag{18}$$

and the relative speed $s = r(E_1)/r(T_1)$. While [44] proves existence of random graphs where $w_n \to \infty$ as $n \to \infty$, ratio w_n is only $2-3$ in real graphs commonly studied in this area. Given that s is at least 20 on modern CPUs, it is conclusive that scanning edge iterators will remain the best option until graphs are discovered with significantly larger w_n.

V. IMPLEMENTATION

We now build a fast implementation of E_{1B} that takes advantage of SIMD for scanning the lists and PCF-B for I/O. We call this method PaCiFier and make it available in [11].

A. Intersection

Since E_{1B} spends almost all of its CPU time on intersection, it is crucial to address this bottleneck first. With support for SIMD in modern CPUs, we can exploit data-level parallelism

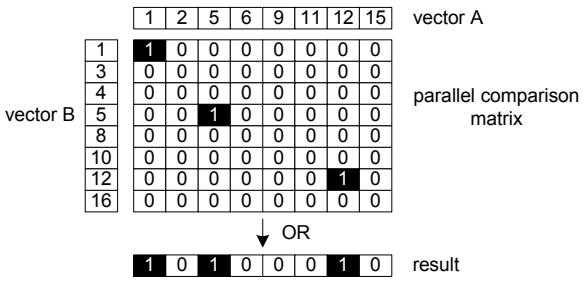


Fig. 7. Parallel intersection with STTNI.

TABLE VI
SINGLE-CORE SPEED (INTEL I7-3930K @ 4.4 GHZ)

	Speed (M/sec)
Hash table	19
Naive scalar intersection	264
Branchless intersection	416
SIMD 32-bit intersection	1,119
SIMD 16-bit intersection	1,801

and achieve a significant speedup compared to traditional CPU-based methods. We adopt the technique from [35], which utilizes STTNI intrinsics from SSE 4.2. They work on two 128-bit vector registers, treating them as four 32-bit or eight 16-bit integers. Fig. 7 shows how STTNI builds an all-to-all comparison matrix and outputs a vector of matches using just one instruction. While 32-bit intersection is fast, better results can be procured by compressing labels into 16-bit numbers. This is performed by grouping node IDs into chunks that share the same upper 16 bits. For each chunk, PaCiFier additionally keeps its length and a list of the lower two bytes from each original label. This works well because all vertices are sequentially relabeled and adjacency lists are kept in ascending order. Besides almost doubling intersection speed, this method reduces graph size by approximately 50%.

For lists that are shorter than some threshold (e.g., 16), both compression and 16-bit intersection do not work well. In these cases, we keep the lists in 32-bit format and apply the branchless scalar (i.e., non-SIMD) intersection from [19]. A benchmark of these operations together with the Google Hash Table are shown in Table VI. With 1.8B operations/sec, PaCiFier's ratio s is a whopping 94.7. This places even more doubt that T_{1B} will be competitive in the near future, especially given that RAM bandwidth scales much faster than latency [33], i.e., s will continue increasing.

B. Relabeling and Orientation

For degree-based permutations, prior work sorts pairs (degree, ID) to establish a total order. This becomes a major bottleneck in preprocessing, especially for large graphs where these tuples do not fit in RAM. In contrast, we use a novel approach that decides the new labels without sorting the nodes. We first accumulate a histogram of degree frequency in one pass over pairs (i, d_i), which are kept separately from the adjacency lists $\{N_i\}$. Using a prefix sum of the histogram, we then establish the starting IDs for nodes of each unique degree value. Performing another scan of the tuples, we find

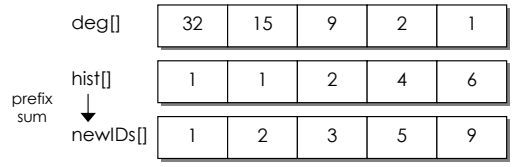


Fig. 8. Descending-degree relabel with a histogram.

the degree of each source node i in the histogram and create a mapping from old labels to the corresponding new IDs. This is shown in Fig. 8. Frequently accessed parts of the histogram typically fit in the L2 cache, which makes lookups against them extremely fast.

If the mapping fits in RAM, PaCiFier performs a scan over the adjacency lists and rewrites all edges in one-pass. Otherwise, it changes the source nodes, inverts the graph, and updates the source nodes again.

C. Parallelization

Scaling PaCiFier to multiple cores is rather straightforward. In Algorithm 5, the processing of each record $(i, X, Y) \in G^c_\theta(l)$ is an independent job, which allows multiple threads to work on different lists without interfering with each other. The lookup table H is read-only and can be safely shared by all worker threads without any locks. Assuming c available cores and hyper-threading, we run $2c$ worker threads and set the affinity mask to bind each thread to a dedicated core. This configuration ensures 100% CPU utilization for the entire execution and almost linear scalability with the number of cores (see below).

D. Evaluation Setup and Datasets

Experiments use a six-core Intel i7-3930K @ 4.4 GHz, Asus Rampage IV Extreme motherboard, and quad-channel DDR3 RAM @ 2133 MHz. We compare PaCiFier against four methods with available implementations – RGP [8], DGP [8], MGT [17], and PDTL [13]. For the first three techniques, we use a multi-threaded binary shared by the authors of [17].

We employ all standard graphs in the field – Live Journal (LJ) [17], US road maps (USRD) [17], Billion Triples Challenge (BTC) [15], WebUK [17], Twitter [23], and Yahoo [45]. Note that the original Yahoo graph has $n = 1.4$B, which reduces to 720M after removing zero-degree nodes. To cover a wider variety of options, we add two web crawls: IRLbot [25] and ClueWeb [9]. Out of the former, we extract domain, host, and IP-level graphs. Assuming $I(x)$ is the IP address of an authoritative nameserver for domain x, graph IRL-IP contains edges $I(x) \to I(y)$ iff $x \to y$ in IRL-domain, which may be useful for spam detection and ranking. The original ClueWeb dataset published online [9] does not contain any dynamic links and is limited to 7.9B edges [32]. We remedy this problem by running our HTML parser over all pages, which yields a much larger graph with 102B links. The new files can be downloaded from [11].

Table VII summarizes statistics of the graphs, where the old datasets require billion-scale intersection cost $c_n(E_1, \theta_D)$ and

TABLE VII
DATASET PROPERTIES

Graph	Nodes (n)	Degree sum ($2m$)	Triangles	w_n	$c_n(E_1, \theta_D)$	Size	$E[d_i]$	$\max_i d_i$	$\max_i d_i^+$
LJ	4,846,609	85,702,474	285,730, 264	3.01	2.1B	364 MB	17.7	20,333	685
USRD	23,947,347	57,708,624	438,804	2.37	25M	403 MB	2.4	9	4
BTC	164,660,997	772,822,094	28,498,939	1.59	3.5B	4.1 GB	4.7	1,637,619	646
WebUK	62,338,347	1,877,431,056	179,076,331,071	1.99	364B	7.5 GB	30.1	48,822	5,692
Twitter	41,652,230	2,405,026,390	34,824,916,864	3.38	511B	9.3 GB	57.7	2,997,487	4,102
Yahoo	720,242,173	12,869,122,070	85,782,928,684	1.47	433B	53.3 GB	17.9	7,637,656	1,540
IRL-domain	86,534,416	3,416,273,404	112,797,037,447	3.63	1.4T	13.3 GB	39.5	2,948,635	4,481
IRL-host	641,982,060	12,872,821,328	437,436,899,269	2.85	2.6T	52.7 GB	20.1	5,475,377	4,516
IRL-IP	1,588,925	1,636,848,800	1,032,158,059,864	3.17	4.2T	6.1 GB	1,030	669,776	8,915
ClueWeb	8,179,508,503	102,394,528,124	879,280,163,294	2.00	3.0T	358 GB	12.5	44,383,637	1,747

TABLE VIII
PREPROCESSING TIME (SECONDS)

Graph	MGT	PDTL	PaCiFier
LJ	2.2	1.0	1.7
USRD	2.0	1.4	2.0
BTC	18.8	11.6	8.9
WebUK	36.9	24.5	14.7
Twitter	88.9	38.4	24.5
Yahoo	295	276	149
IRL-domain	149	61.9	31.8
IRL-host	736	456	221
IRL-IP	33.9	19.1	8.5
ClueWeb	8,192	19,502	962

TABLE IX
RUNTIME (SECONDS) WITH 8 GB OF RAM

Graph	RGP	DGP	MGT	PDTL	PaCiFier
LJ	22.3	22.2	11.2	2.8	0.7
USRD	12.3	12.3	1.2	6.2	0.3
BTC	111	110	11.4	12.1	2.1
WebUK	1,299	891	599	93.6	17.1
Twitter	10,300	9,814	2,238	327	63.4
Yahoo	31,945	13,990	1,080	619	79.2
IRL-domain	17,717	16,919	5,946	849	148
IRL-host	–	–	11,099	1,773	367
IRL-IP	–	–	18,617	2,358	237
ClueWeb	–	–	*	13,782	1,737

TABLE X
RESULTS FROM PRIOR WORK

Type	Algorithm	Runtime (sec)		Cores or servers
		Twitter	Yahoo	
RAM-only	[36]	101	–	16
	[37]	55.9	77.7	40
External	PATRIC [2]	552	–	200
	OPT [22]	469	819	6
MapReduce	[10]	36,300	–	47
	GP [38]	28,980	–	1,636
	TTP [31]	12,780	–	47
	CTTP [32]	5,520	61,920	40

the new ones trillion-scale. The densest graph IRL-IP has an average degree $1{,}030$, contains over 1T triangles, and requires 4.2T intersection operations. ClueWeb comes in at a hefty 358 GB, but neither its number of triangles nor CPU cost can top those of IRL-IP. Also note that the longest out-list in the table occupies just 35 KB of RAM, far smaller than the longest undirected neighbor set (i.e., 177 MB).

E. Preprocessing Time

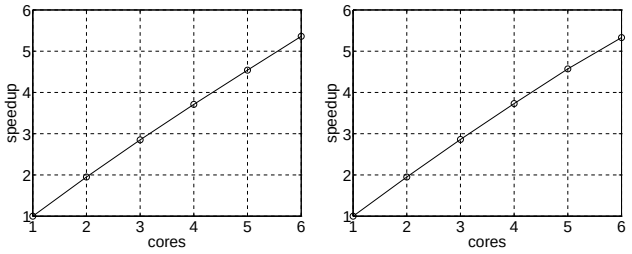
RGP/DGP do not require preprocessing, while the other three methods manipulates the input graph G into a suitable format prior to actual listing of triangles. It is common to time the two phases separately, especially since the former can be performed once and the latter repeated many times on the same preprocessed data. Table VIII shows the result using a RAID system capable of reads at 1 GB/s. Even though PaCiFier is the only one performing both relabeling and orientation, its usage of the histogram to avoid sorting makes it $2-8$ times faster than MGT and up to 20 times faster than PDTL.

F. Triangle-Listing Time

We run the next set of tests using an 8-GB RAM constraint, which ensures that I/O is not a bottleneck for our RAID. As a result, Table IX presents an evaluation of pure CPU efficiency of each algorithm. PaCiFier's performance is determined by the length of neighbor lists, i.e., efficiency of SIMD scanning. Compared to MGT, which implements T_1, its speedup varies from a factor of 13.6 on Yahoo to 78.6 on IRL-IP. In the latter graph, PaCiFier finds 1T triangles in 237 seconds, which translates into 17.7B neighbor checks/sec and 4.3B discovered triangles/sec using all six cores. Compared to PDTL, which is an optimized version of E_1 with MGT's partitioning scheme, PaCiFier achieves a $5-10\times$ faster runtime.

The number of found triangles is consistent across the methods, except RGP/DGP fail to finish within 12 hours on several graphs, which we indicate with a dash. Additionally, MGT quits with an unrealistically small number of triangles (i.e., 170M) after spending 24K seconds on ClueWeb, which we show with an asterisk. Its traces point toward early termination before processing all of the partitions; however, unavailability of the source code prevents further analysis.

To put these results in perspective, Table X cites the runtime from prior work on Twitter and Yahoo. We split the algorithms into several categories – RAM-only, external-memory, and MapReduce. We report the number of utilized cores for the former two groups and cluster size for the last one. The first two methods in the table [36], [37] produce comparable numbers to those of PaCiFier, but using $3-6$ times more late-model Xeon cores. Due to their RAM-only operation, we do not consider them competitors for PaCiFier. The next two techniques [2], [22] are extensions of RGP/DGP and MGT to multiple machines. They are generally faster than their respective predecessors, but still far slower than PaCiFier. The



(a) Twitter (b) IRL-domain

Fig. 9. Speedup vs. number of cores (8 GB of RAM).

TABLE XI
RUNTIME (SECONDS)

Graph	RAM (MB)	MGT	PDTL	PaCiFier
Twitter	8,192	2,238	323	63.3
	4,096	2,248	327	63.2
	2,048	2,260	327	61.9
	1,024	2,285	347	61.0
	512	2,354	464	61.4
	256	2,487	1,003	67.2
IRL-domain	8,192	5,947	849	148
	4,096	5,976	851	144
	2,048	6,020	853	143
	1,024	6,090	898	143
	512	6,252	995	145
	256	6,540	1,484	149

final four methods [10], [38], [31], [32] in the table are entirely disappointing – 87 to 572 times slower than PaCiFier while consuming substantially more resources.

G. Parallelization Efficiency

We now examine how PaCiFier scales with the number of cores, which indicates how well the algorithm benefits from additional CPU resources. As discussed earlier in section V-C, PaCiFier's parallelization framework partitions the computation (i.e., triples (i, X, Y') from the companion file) into equal-sized jobs, which are processed lock-free by worker threads. As shown in Fig. 9, PaCiFier's runtime indeed scales almost linearly. The reason for a slightly suboptimal outcome is that certain auxiliary operations (e.g., indexing of $G^r_\theta(l)$ in Line 4 of Algorithm 5) are executed sequentially.

H. Effect of RAM: Bottlenecked by CPU

Next, we analyze the performance of each algorithm under varying RAM size. We showed earlier that PaCiFier's CPU cost was constant for all M. While the I/O complexity does increase as $M \to 0$, double buffering and prefetching can keep this overhead negligible until the disk becomes a bottleneck. Table XI supports this discussion – using our RAID system, PaCiFier completes in virtually the same amount of time for all M in the range between 256 MB and 8 GB. The initial drop in runtime can be explained by smaller lookup tables and better cache locality; however, as M decreases further, SIMD becomes less efficient and this effect is reversed. While MGT is not bottlenecked by I/O either, PDTL increases its runtime by $49-116\%$ at $M = 256$ MB. More interesting cases where the disk can no longer keep up with the computation are studied next.

TABLE XII
I/O COMPARISON

Graph	RAM (MB)	GP	TTP	MGT/PDTL	PaCiFier
Yahoo (in GB)	8,192	2,099	1,066	88.8	40.4
	4,096	3,271	1,599	177.6	47.6
	2,048	5,247	2,132	355.1	55.5
	1,024	7,632	3,198	710.2	64.8
	512	11,219	4,531	1,420	74.6
	256	16,408	6,663	2,841	84.4
ClueWeb (in TB)	8,192	47.4	19.2	3.91	0.69
	4,096	68.4	27.9	7.82	0.87
	2,048	99.8	40.2	15.6	1.10
	1,024	141.7	55.9	31.3	1.36
	512	204.6	80.4	62.6	1.64
	256	291.1	113.6	125	1.93

I. Effect of RAM: Bottlenecked by I/O

For comparison of disk activity, we use the exact model $m\lceil m/M \rceil$ for MGT/PDTL and compute the size of all companion files in PaCiFier by running Algorithm 4. Although DGP/RGP share the same $\Theta(m^2/M)$ asymptotic cost with MGT, these methods require two orders of magnitude more I/O due to slow convergence, which we omit from analysis. Instead, we contrast against MapReduce methods. The first one is GP [38], which uses at least $\rho = \lceil 3\sqrt{m/M} \rceil$ reducers and shuffles

$$\frac{30(\rho-1)(\rho-2)m}{\rho} \tag{19}$$

bytes of data [31]. A later method called TTP [31] reduces ρ by a factor of $\sqrt{3}$ and improves the shuffle to $20(\rho-1)m$.

Table XII shows the I/O in bytes on the two largest graphs under consideration. PaCiFier starts off beating GP/TTP by a factor of $32-78$ and MGT/PDTL by a factor of $3.7-9$. This advantage keeps accumulating as M decreases. Eventually, PaCiFier develops a $58-195\times$ lead over the former and $34-64\times$ over the latter as M reaches 256 MB. In the last scenario, the I/O phase of MGT/PDTL would require 34.5 hours to finish ClueWeb using our 1 GB/s RAID. With a magnetic hard drive (i.e., 100 MB/s read speed), this would take over two weeks. On the other hand, PaCiFier lowers these numbers to 32 minutes and 5.3 hours, respectively.

VI. CONCLUSION

The paper created a taxonomy of 18 triangle-listing methods using a unifying framework called Generalized Iterators (GI), developed a new set of algorithms called Pruned Companion Files (PCF) for external-memory operation of GI, and showed that it possessed better complexity than current implementations in the field. It then determined which of the 18 methods was the most efficient when both CPU and I/O objectives were taken into account and created a working solution that exhibited $5-10\times$ smaller runtime and orders of magnitude less I/O compared to the best previous technique.

REFERENCES

[1] N. Alon, R. Yuster, and U. Zwick, "Finding and Counting Given Length Cycles," *Algorithmica*, vol. 17, no. 3, pp. 209–223, Mar. 1997.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

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

[9] ClueWeb09 Dataset. [Online]. Available: http://www.lemurproject.org/clueweb09/.

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

[11] Y. Cui, D. Xiao, and D. Loguinov, "IRL Triangle Datasets and Code," Sep. 2016. [Online]. Available: http://irl.cs.tamu.edu/projects/motifs/.

[12] 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.

[13] 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.

[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] A. Harth, "Billion Triples Challenge Data Set," 2009. [Online]. Available: http://km.aifb.kit.edu/projects/btc-2009/.

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

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

[18] X. Hu, M. Qiao, and Y. Tao, "Join Dependency Testing, Loomis-Whitney Join, and Triangle Enumeration," in *Proc. ACM PODS*, May 2015, pp. 291–301.

[19] H. Inoue, M. Ohara, and K. Taura, "Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions," *PVLDB*, vol. 8, no. 3, pp. 293–304, Nov. 2014.

[20] A. Itai and M. Rodeh, "Finding a Minimum Circuit in a Graph," *SIAM Journal on Computing*, vol. 7, no. 4, pp. 413–423, 1978.

[21] 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.

[22] 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.

[23] H. Kwak, C. Lee, H. Park, and S. Moon, "Twitter Graph," 2010. [Online]. Available: http://an.kaist.ac.kr/traces/WWW2010.html.

[24] 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.

[25] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," *ACM Trans. Web*, vol. 3, no. 3, pp. 1–34, Jun. 2009.

[26] 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.

[27] 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.

[28] 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.

[29] M. Ortmann and U. Brandes, "Triangle Listing Algorithms: Back from the Diversion," in *Proc. ALENEX*, Jan. 2014, pp. 1–8.

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

[31] 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.

[32] H. Park, F. Silvestri, U. Kang, and R. Pagh, "MapReduce Triangle Enumeration With Guarantees," in *Proc. ACM CIKM*, Nov. 2014, pp. 1739–1748.

[33] D. A. Patterson, "Latency Lags Bandwidth," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, Oct. 2004.

[34] 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.

[35] B. Schlegel, T. Willhalm, and W. Lehner, "Fast Sorted-Set Intersection using SIMD Instructions," in *Proc. ADMS*, Sep. 2011.

[36] 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.

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

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

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

[40] 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.

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

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

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

[44] D. Xiao, Y. Cui, D. B. Cline, and D. Loguinov, "On Asymptotic Cost of Triangle Listing in Random Graphs," Texas A&M University, Tech. Rep. 2016-9-2, Sep. 2016. [Online]. Available: http://irl.cs.tamu.edu/publications/.

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

[46] 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.