Technical Report

UCAM-CL-TR-866
ISSN 1476-2986

Number 866

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

PDTL: Parallel and distributed triangle listing for massive graphs

Ilias Giechaskiel, George Panagopoulos, Eiko Yoneki

April 2015

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

PDTL: Parallel and Distributed Triangle Listing for Massive Graphs

Ilias Giechaskiel
University of Cambridge
Cambridge, UK
Email: Ilias.Giechaskiel@cl.cam.ac.uk

George Panagopoulos
University of Cambridge
Cambridge, UK
Email: gmp37@cam.ac.uk

Eiko Yoneki
University of Cambridge
Cambridge, UK
Email: Eiko.Yoneki@cl.cam.ac.uk

***Abstract*— This paper presents the first distributed triangle listing algorithm with provable CPU, I/O, Memory, and Network bounds. Finding all triangles (3-cliques) in a graph has numerous applications for density and connectivity metrics. The majority of existing algorithms for massive graphs are sequential processing and distributed versions of algorithms do not guarantee their CPU, I/O, Memory or Network requirements. Our Parallel and Distributed Triangle Listing (PDTL) framework focuses on efficient external-memory access in distributed environments instead of fitting subgraphs into memory. It works by performing efficient orientation and load-balancing steps, and replicating graphs across machines by using an extended version of Hu et al.'s Massive Graph Triangulation algorithm. As a result, PDTL suits a variety of computational environments, from single-core machines to high-end clusters. PDTL computes the exact triangle count on graphs of over 6B edges and 1B vertices (e.g. Yahoo graphs), outperforming and using fewer resources than the state-of-the-art systems PowerGraph, OPT, and PATRIC by $2\times$ to $4\times$. Our approach highlights the importance of I/O considerations in a distributed environment, which has received less attention in the graph processing literature.**

I. Introduction

Graphs have become important abstractions to model real-world situations, ranging from social relationships to communication, web, and road networks, but such graphs are becoming increasingly massive and and will soon reach billions of vertices and trillions of edges, making in-memory algorithms insufficient for computing graph properties. One such property that has gained the attention of the graph processing community is the *number of triangles* in the graph, which from a theoretical point of view can be seen as a special case of counting cycles of given length, or finding complete subgraphs. From a more practical perspective, finding all triangles in a graph is crucial for metrics such as the *clustering coefficient* [24] and the similar *transitivity ratio* [18], which can be used to find high-density nodes, and to detect fake accounts in social networks [25], as well as web spam and content quality [5]. Triangle enumeration is also necessary as a sub-algorithm for *dense neighborhood discovery* [23], *triangular connectivity* [4], and finding the *k-trusses* of graphs [22]. As a result, triangle listing and counting has a wide range of applications, but research has been limited on either external memory considerations, or the creation of parallel frameworks (Section II). However, in this paper we show that it is possible to combine both approaches with substantial improvements in performance. Our Parallel and Distributed Triangle Listing (PDTL) framework (Section IV) extended Hu et al.'s Massive Graph Triangulation (MGT) algorithm [13] in order to work in the distributed environment by duplicating the graph across each machine and has provable bounds on CPU, I/O, Memory, and Network utilization. By further parallelizing the orientation step, and by intelligently distributing the load across each processor, we are able to optimize our algorithm to the point where it computes the exact triangle counts on graphs with billions of edges or vertices $2\times$ to $4\times$ faster than the state-of-the-art frameworks, using considerably fewer resources, and exhibiting scalability across multiple processors and machines (Section V). In summary, our contributions are as follows:

- We create a general framework for triangle listing and counting for both distributed and single-machine systems. Our algorithm is the first triangle listing algorithm that provides efficient and well-understood bounds on CPU, I/O, Memory, and Network utilization, across multiple environments (Theorem IV.3).
- We uncover hidden assumptions in the proofs and implementation of the closed-source MGT algorithm. We modify the algorithm to correspond to its implementation, and prove that our modifications do not alter its theoretical efficiency (Section IV-A).
- We introduce further optimizations in the orientation (Definition III.2) and distribution steps of our algorithm reducing bottlenecks, without adding complexity.
- We conduct extensive experiments, and show that our algorithm is highly scalable across multiple cores and machines, with low memory requirements, even for graphs with hundreds of millions of edges, and multiple billion edges (Section V). In particular, over the standard Twitter dataset [15], our algorithm is 4 times faster than PATRIC [3], 3 times faster than OPT [14], and 2 times faster than PowerGraph [10], which are the state-of-the-art frameworks in distributed and multicore triangle counting.

II. Related Work

Dementiev [9] and Menegola [17] first introduced **external-memory** algorithms for triangle counting, but their algorithms had high I/O overheads. The first algorithms with reasonable performance for triangle listing were introduced by Chu and

Cheng [8], and relied on *graph partitioning* to achieve an I/O complexity of $\mathcal{O}\left(\frac{|E|^2}{MB}+\frac{T}{B}\right)$, under certain assumptions on the structure of the graph. However, MGT by Hu et al. [13] exhibits the same performance without any additional assumptions, and was proven to be superior in practice. Finally, Pagh and Silvestri recently proposed a new algorithm for triangle counting (but not listing) which has an I/O complexity of $\mathcal{O}\left(\frac{E^{1.5}}{\sqrt{MB}}\right)$, and improves the given bounds by a factor of $\min\left(\sqrt{E/M},\sqrt{M}\right)$ [19].

The first **dedicated parallel** triangle counting framework, PATRIC [3], uses graph partitioning and message passing for exact counting. It is not I/O-efficient, but it proposes multiple novel load balancing mechanisms, which are calculated in parallel and do not pose a bottleneck. Even so, PATRIC requires that each partition fits in memory, and targets datacenters, with hundreds of processors and high dedicated RAM per processor. OPT [14] is a disk-based, single machine system that exploits I/O and multi-core CPU parallelism, and performs favorably compared to distributed triangle-counting frameworks.

In terms of **general-purpose frameworks**, there are multiple MapReduce algorithms for counting triangles, the best of which is CTTP [20]. Even so, MapReduce algorithms produce too much intermediate networking data, and are considerably slow: CTTP takes $2\times$ longer on the Twitter dataset [15] using 40 nodes compared a single-core MGT. PowerGraph [10] is a general-purpose vertex-oriented framework that is the fastest for triangle counting among existing alternatives, while PSgL [21] proposes novel methods for generic subgraph listing, but is $6\times$ slower than PowerGraph on the Twitter graph.

Overall, we see that there is a divide between using external memory and parallelizing the algorithm, but as we show in Section V, by combining the two approaches PDTL is $4\times$ faster than PATRIC, $3\times$ faster than OPT, and $2\times$ faster than PowerGraph, while providing theoretical guarantees, and not running out of memory for larger graphs.

III. Preliminaries

A. Definitions

All graphs $G=(V,E)$ on $n=|V|$ vertices and $m=|E|$ edges are assumed to be undirected (bi-directional) and simple. For every $u\in V$, we denote by $N_G(u)=\{v:(u,v)\in E\}$ the set of its *neighbors* (or *adjacency list*) and by $d_G(u)=|N_G(u)|$ its *degree*. Note that we may omit the qualifier G when doing so is clear. Finally, for simplicity, we also identify V with $[n]=\{0,\ldots,n-1\}$.

Definition III.1 (Triangle)**.** *Given an undirected graph $G=(V,E)$, a triangle is a set of three vertices $\{u,v,w\}\subseteq V$, such that all of (u,v), (v,w) and (w,u) are edges in E.*

Finding the set K of all such triangles is called triangle *listing*, and merely reporting on their number $T=|K|$ is called triangle *counting*.

Definition III.2 (Degree-Based Order, Orientation)**.** *Given an undirected graph $G=(V,E)$, the degree-based order $\prec$ on V is defined as follows: $u\prec v$ if and only if $d(u)<d(v)$ or $d(u)=d(v)$ and $u<v$. We define the directed graph $G^*=(V,E^*)$, called G's orientation, by $(u,v)\in E^*$ if and only if $(u,v)\in E$ and $u\prec v$.*

Because $\prec$ is a strict total order, the orientation uniquely associates the triangle $\{u,v,w\}$ where $u\prec v\prec w$ with the tuple (u,v,w):

Definition III.3 (Cone Vertex, Pivot Edge [13])**.** *Given a triangle (u,v,w) with $u\prec v\prec w$ in G^*, we call u its cone vertex, and (v,w) its pivot edge.*

The arboricity $\alpha(G)$ of a graph G is the minimum number of edge-disjoint forests needed to cover its edges [7] and satisfies the following properties:

Theorem III.4 (Arboricity bounds [7])**.** *The arboricity of a graph $G=(V,E)$ satisfies:*

1) $\alpha\le\left\lceil\sqrt{|E|}\right\rceil$
2) $\alpha=\mathcal{O}(1)$ *if G is planar*
3) $\sum_{(u,v)\in E}\min\{d(u),d(v)\}\le\mathcal{O}(\alpha|E|)$

Note that the $T\le\frac{1}{3}\sum_{(u,v)\in E}\min\{d(u),d(v)\}$, where T is the number of triangles, as any edge can appear in at most $\min\{d(u),d(v)\}$ triangles, so $T=\mathcal{O}(\alpha|E|)$. As a result, it is beneficial to have a runtime dependent on $\alpha(G)$, because it is at most $\left\lceil\sqrt{|E|}\right\rceil$, but can be $\mathcal{O}(1)$ for planar graphs.

Finally, we remind the reader of Aggarwal and Vitter's I/O complexity analysis methodology [2], which depends on the block size B: in accessing N elements in order, the disk performs $scan(N)=\Theta(N/B)$ I/Os, whereas random access can require $\Omega(N)$ I/Os in the worst case. Sorting takes $sort(N)=\Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ I/Os by external mergesort, where M is the memory size.

IV. PDTL

We assume a computational environment of N nodes, each of which has P processors, with M bytes of memory for each of the processors, so that by choosing these parameters appropriately, we can model a high-end data center, with multiple processors per machine, or even just a single computer with low available memory. In Section IV-A we explain the baseline single-core MGT algorithm and our modifications, while in Section IV-B we explain our parallel and distributed PDTL system and prove its theoretical properties.

A. Massive Graph Triangulation

Algorithm 1 presents the MGT algorithm [13] under the *small-degree assumption*: that every vertex $v\in V$ has $d_{G^*}(v)\le cM/2$ for some implementation-specific constant $c<1$.[1] The idea behind MGT is that given an oriented graph G^* one can find all triangles by loading consecutive edges into memory and iterating over all vertices u and their out-edges

[1] We refer the reader to [13] and our code discussing and implementing a way to remove this assumption.

to find all triangles with cone vertex u and pivot edge loaded into memory. By using hash structures on the loaded edges and $N(u)$ this can be done in a CPU- and I/O- efficient way. However, as we illustrate in Section IV-A1, the high-level algorithm does not correspond to its implementation, so we modify MGT, and show in Section IV-A2 that our modifications do not alter the algorithm's efficiency.

Algorithm 1 MGT

Input: An oriented $G^* = (V, E^*)$
Output: All triangles in G
while there are edges in E^* to be read **do**
  Read the next cM edges into memory
  Create hash structures on the edges
  for $u \in V$ **do**
    Read $N(u)$ from disk
    Construct hash structures on $N(u)$
    Report triangles with cone u and pivot in memory
    Release the structures on $N(u)$

1) Modifications: Although only a binary for MGT is available at [11], during our experimentation we hypothesized that the implementation of MGT does not use explicit sets, but arrays. Indeed, if the adjacency list for any given vertex is not sorted, the given implementation misses triangles, though the manual [12] does not make mention of such requirements. Clearly, if any types of sets were constructed, this need for a sorted adjacency list would not be present. This belief was further verified by our own implementation, where using sets and maps of any kind, from C++'s `std::unordered_set` to Google's `google::dense_hash_set`, made our implementation more than $10\times$ slower.

Consequently, our implementation deviates from the proposed high-level Algorithm 1 by utilizing sorted arrays instead of sets. Specifically, it is assumed (for compatibility with the MGT implementation [11]) that the file format for the graph is such that if $v < w$ then $N_G(v)$ comes before $N_G(w)$ and if additionally $v, w \in N_G(u)$ then v comes before w in $N_G(u)$, with these properties preserved after orientation.

Following the notation in [13], let E_{mem} denote the set of edges in memory, V_{mem} its endpoints, V^+_{mem} those $v \in V_{mem}$ that have *outgoing* edges in E_{mem}, and $N^+(u) = N(u) \cap V^+_{mem}$. In other words, $V^+_{mem} = \{u \in V | \exists v \in V : (u,v) \in E_{mem}\}$, and for symmetry let $V^-_{mem} = \{u \in V | \exists v \in V : (v,u) \in E_{mem}\}$, with $V_{mem} = V^+_{mem} \cup V^-_{mem}$. Additionally let $v_{low} = \min_{v \in V^+_{mem}} v$ and $v_{high} = \max_{v \in V^+_{mem}} v$.

Because the graph is sorted, we know that if $v < v_{low}$ or $v > v_{high}$, then $v \notin V^+_{mem}$. As a result, we can split E_{mem} into two arrays: `edg` which stores the sequence of out-neighbors and `ind` that stores the degree of v and offset into `edg` at location $v - v_{low}$. In other words, the out-edges of v (provided it is in memory) are stored at $E_v =$`edg[ind[`$v - v_{low}$`]]`.

Moreover, because $|N^+(u)| \leq |N(u)| \leq d^*_{max}$, and because we know d^*_{max} from the orientation step, $N(u)$ and $N^+(u)$ can also be represented by static arrays of size d^*_{max}, called `nm` and `nmp` respectively.

As a result, the modified MGT (Algorithm 2) works as follows on the sorted and oriented graph: it loads the next $\Theta\left(\frac{|E|}{M}\right)$ edges into `edg` and `ind` as indicated above. Then, it iterates over the entire graph vertex by vertex, and for each vertex u, it does the following:

1) Stores $N(u)$ into the array `nm`
2) Evaluates $N^+(u)$ into the array `nmp` by iterating over $v \in N(u)$ and checking `ind` to see if it has any out-neighbors
3) For each $v \in N^+(u)$, it reports triangle (u,v,w) with cone u and pivot w for each $w \in N(u) \cap E_v$

Algorithm 2 Modified MGT

Input: A sorted, oriented $G^* = (V, E^*)$
Output: All triangles in G
while there are edges in E^* to be read **do**
  Read the next $c'M$ out-neighbors into `edg` and store in `ind` the degrees and offsets
  for $u \in V$ **do**
    Read $N(u)$ from disk to array `nm`
    Write $N^+(u)$ to `nmp` using `nm` and `ind`
    for $v \in$ `nmp` **do**
      for $w \in$ `nm` $\cap$ `edg[ind[`$v - v_{low}$`]]` **do**
        Report (u,v,w)
    Clear `nm` and `nmp`

2) Analysis: First of all, because $\Theta(M)$ edges are loaded at each step there are $h = \Theta(|E|/M)$ iterations, and each iteration performs $\frac{|E|}{B}$ I/Os to read over the graph. Additionally, the cost of outputting T triangles is $\frac{T}{B}$, for a total I/O complexity of $\mathcal{O}\left(\frac{|E|^2}{MB} + \frac{T}{B}\right)$.

For the CPU complexity, we note that checking whether $v \in V^+_{mem}$ amounts to checking whether `indices[`$v - v_{low}$`]` has a positive degree, which is a $\mathcal{O}(1)$ operation, so construction of E_{mem}, and V^+_{mem} (together with clearing it) takes $\Theta(|E_{mem}|) = \Theta(M)$ time. Construction of $N(u)$ and $N^+(u)$ thus also takes $\Theta(|N(u)|) = \Theta(d_{G^*}(u))$ time. Since each edge is examined once in a single iteration, each iteration incurs time $\mathcal{O}(|E|)$ for construction of these structures, for a total of $\Theta\left(|E|^2/M\right)$ time.

Set intersection of two ordered sets of size m, n takes time $\mathcal{O}(m+n)$ using a naive set intersection, thus the total complexity for the triangle operations is

$$\sum_{i=1}^{h} \sum_{u \in V} \sum_{v \in N_i^+(u)} d_{G^*}(u) + d_{G^*}(v)$$

where $N_i^+(u)$ denotes $N^+(u)$ in the i-th iteration. First, note that any v is in at most 2 (consecutive) $N_i^+(u)$ for any given u. This is due to the small degree assumption, because if the adjacency is split the first time, the second time it will entirely fit in memory. Thus, we can change the order of summation as follows:

$$
\begin{aligned}
&\sum_{i=1}^{h}\sum_{u\in V}\sum_{v\in N_i^+(u)} \\
=\ &\sum_{u\in V}\sum_{i=1}^{h}\sum_{v\in N_i^+(u)} \\
\leq\ &2\sum_{u\in V}\sum_{v\in N^+(u)}
\end{aligned}
$$

Examining each term separately (and ignoring the factor of 2 for clarity):

$$\sum_{u\in V}\sum_{v\in N^+(u)} d_{G^*}(u) = \sum_{u\in V} d_{G^*}^2(u)$$

Additionally,

$$
\begin{aligned}
&\sum_{u\in V}\sum_{v\in N^+(u)} d_{G^*}(v) \\
=\ &\sum_{v\in V} d_{G^*}(v)\left(d_G(v) - d_{G^*}(v)\right) \\
=\ &\sum_{v\in V} d_G(v)\cdot d_{G^*}(v) - \sum_{v\in V} d_{G^*}^2(v)
\end{aligned}
$$

because $d_G(v) - d_{G^*}(v)$ represents the number of incoming vertices to v. The sums of $d_{G^*}^2(v)$ cancel out, so we need to calculate $\sum\limits_{v\in V} d_G(v)\cdot d_{G^*}(v)$. This is where the arboricity becomes useful (Theorem IV.1 is adapted from — but is not identical to — the one given in [13]):

Theorem IV.1 (Ordering)**.** $\sum\limits_{v\in V} d_G(v)\cdot d_{G^*}(v) = \mathcal{O}\left(\alpha|E|\right)$

Proof.

$$
\begin{array}{lcl}
\sum\limits_{v\in V} d_G(v)\cdot d_{G^*}(v) & = & \sum\limits_{v\in V}\sum\limits_{u\in N^+(v)} d_G(v) \\
 & = & \sum\limits_{(v,u)\in E^*} d_G(v) \\
\text{(by orientation)} & \leq & \sum\limits_{(v,u)\in E} \min\{d(v), d(u)\} \\
\text{(by Theorem III.4)} & = & \mathcal{O}\left(\alpha|E|\right)
\end{array}
$$

□

Note that the sorting of the original file takes $\mathcal{O}\left(sort\left(|E|\right)\right)$ I/Os and $\mathcal{O}\left(|E|\ln|E|\right)$ CPU time [2], while the orientation itself takes $\mathcal{O}\left(scan(|E|)\right)$ I/Os and $\mathcal{O}\left(|E|\right)$ CPU time, provided that the entire degree array can fit in memory.[2] If not, in the worst case (e.g. for the complete graph K_n), a vertex has a neighbor in every block. As a result, for each node, there must be $\mathcal{O}\left(|V|/B\right)$ I/Os, for a total of $\mathcal{O}\left(|V|^2/B\right)$ I/Os, just for the degree file. Because $|E| = \mathcal{O}\left(|V|^2\right)$, the total complexity is $\mathcal{O}\left(scan\left(|V|^2\right)\right)$ I/Os and $\mathcal{O}\left(|E|\right)$ CPU time. This does not make a difference in dense graphs (except for the asymptotic constant), but it is still a point of omission for the analysis presented in [13]. Consequently, the overall complexity is identical to that of the baseline MGT, and is summarized in Theorem IV.2.

[2]The degrees and adjacency lists for all vertices are stored in separate files of sizes $|V|$ and $|E|$ respectively (Section V-B).

Theorem IV.2 (MGT Complexity)**.** *In summary, our implementation of MGT has an I/O complexity of*

$$\mathcal{O}\left(\frac{|E|^2}{MB} + \frac{T}{B}\right)$$

and CPU complexity of

$$\mathcal{O}\left(\frac{|E|^2}{M} + \alpha|E|\right)$$

If the graph is not already sorted, an additional $\mathcal{O}\left(sort(|E|)\right)$ I/Os and $\mathcal{O}\left(|E|\log|E|\right)$ computations are needed, and if $|V| < M$, $\mathcal{O}\left(scan(|V|^2)\right)$ I/Os are necessary to orient it.

B. Distributed Framework

Our distributed protocol works as follows: every machine is sent a copy of the entire graph, and every available processor is allocated a (contiguous) set of edges S, and is responsible for finding all triangles in the graph which contain pivot edges in S, by using MGT. This is significantly different from the existing parallel triangle-counting systems, where different machines are responsible for different subsets of the vertices.

1) Description: In our framework, a *master* machine delegates responsibility to the N *client* machines (including itself), and combines their results. Because the orientation step need only occur once, it is the responsibility of the master to apply the degree-based order to the graph in question, before sending it over the network. The master then sends the oriented graph to each client, together with the indices that each processor is responsible for. Each core processes the adjacency list between the specified indices. The client combines the triangle counts (and possibly the triangle lists if necessary), and sends these back to the master, which atomically sums the results.

Our PDTL framework is oblivious to how the orientation step is performed, and what specific (contiguous) subset of edges is assigned to each processor. In a naive implementation, orientation is performed sequentially, and edges are split equally to all processors. However, our master parallelizes the orientation, and includes a load-balancing step to equalize the time taken for triangle counting in each of the processors. More concretely, for **multicore orientation**, the master reads the entire degree array into memory (provided $|V| < PM$), and each core performs the orientation on a contiguous set of edges, which are then concatenated. **Load balancing** similarly calculates the number of in-edges for each vertex after orientation (equal to $d_G(v) - d_{G^*}(v)$), and splits the edges equally amongst the processors so that the are still contiguous, and the sum of these in-degrees are approximately the same among all processors. This provides an estimate for the average size of $N^+(u)$, and thus the number of required intersections.

Our protocol is illustrated in Figure 1. For clarity, the master process is duplicated, and is shown to run on a separate machine from the clients. In our illustration, boxes represent different processes and clients, while ovals within boxes represent threads. Lines between boxes represent network traffic, with solid lines representing requests, and dotted lines answers. Finally, $C_{i,j}$ represents the "configuration" for

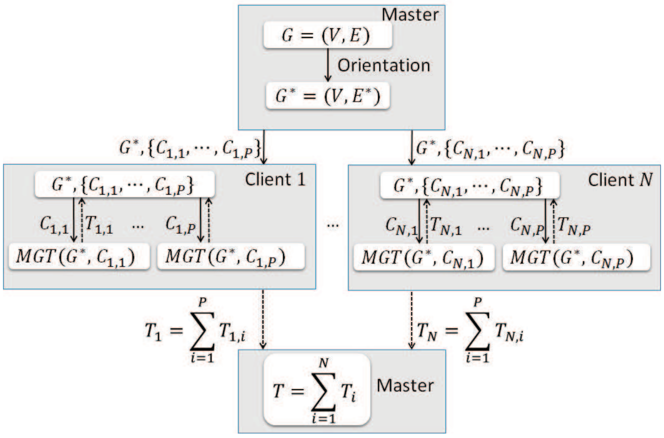

Fig. 1: PDTL protocol overview

processor j on machine i: the memory allocated for that thread, together with the section of the graph for which the processor is responsible. Note that the master starts the triangle counting operations before the network transfer has finished, so sending the graph does not pose a big bottleneck in practice.

2) Analysis: A problem with distributed algorithms using graph partitioning is that they assume each partition can fit in memory. Though for smaller graphs this may be the case, in dense graphs, such as the complete K_n, this is no longer true. Such algorithms require $\Theta\left(n^2\right)$ memory on *each* of the processors, and each of the NP processors must receive the entire graph. However, our algorithm requires memory proportional to the maximum degree, and the graph is only duplicated once per the N nodes. As a result, PDTL has lower network traffic and is preferable for dense graphs, and is also able to accommodate more computational environments. More concretely, PDTL incurs $\Theta\left(N \cdot (P + |E|) + T\right)$ **network** traffic in total, where T is the total number of triangles in the graph (or 0 for triangle counting), due to the communication cost per processor, and the duplication across the N nodes.

Since the **master** is responsible for orienting the graph according to the degree-based order, it incurs $\mathcal{O}\left(scan(|E|)\right)$ I/Os and $\mathcal{O}\left(|E|\right)$ CPU time, assuming there is enough memory to hold V, as explained in Section IV-A2. This is true even for multicore orientation, as the graph is read once over all cores, but with an additional $\mathcal{O}\left(P\right)$ term: one for each of the cores. For load balancing, the vertex degrees are read once, with $PM = \mathcal{O}\left(|E|\right)$ edges sampled, and the results are stored for NP processors using $\mathcal{O}\left(scan(|V|)\right)$ I/Os and $\mathcal{O}\left(|V| + max(|E|, PM)\right) = \mathcal{O}\left(|E|\right)$ CPU time. The master is also responsible for adding the triangle counts received (in parallel) and also concatenating the triangle listing (sequentially), for a CPU complexity of $\mathcal{O}\left(N + T\right)$ and an I/O complexity of $\mathcal{O}\left((T + N)/B\right)$, as there might be an additional block for each of the N machines.

Since each processor is responsible for a unique (contiguous) section of the graph, there are no repeated **computations**. The chunk that each processor is responsible for has size $S = \frac{|E|}{NP}$, and each processor must make $R = \left\lceil \frac{S}{M} \right\rceil$ iterations over the graph.[3] During these iterations, the graph is read once for creation of the vertex structures, and contributes $\mathcal{O}\left(|E|\right)$ processing time. Though it would be impossible to calculate exactly the amount of computations performed in each iteration for counting triangles as it depends closely on the graph structure, we know by the proof of Theorem IV.2 that over all processors, these computations sum to $\mathcal{O}\left(\alpha \cdot |E|\right)$.[4] Consequently, total computations across all processors are

$$\mathcal{O}\left(NP \cdot \left\lceil \frac{|E|}{NPM} \right\rceil \cdot |E| + \alpha \cdot |E|\right) = \mathcal{O}\left(NP \cdot |E| + \frac{|E|^2}{M} + \alpha \cdot |E|\right)$$

because $\left\lceil \frac{|E|}{NPM} \right\rceil \leq \frac{|E|}{NPM} + 1$.

The **I/O** complexity is also easy to find. As above, each processor makes $R = \left\lceil \frac{|E|}{NPM} \right\rceil$ iterations over the graph, and outputs a variable number of triangles t, making its I/O complexity equal to $\mathcal{O}\left(R \cdot scan(|E|) + scan(t)\right)$. As a result, the total I/O over all processors is

$$\mathcal{O}\left(NP\frac{|E|}{B} + \frac{|E|^2}{MB} + \frac{T}{B}\right)$$

One of the important distinctions between PDTL and frameworks which load entire subgraphs in memory is that in PDTL, $|E|$ can still be larger than the total amount of available memory NPM. Moreover, we see that when $NPM > |E|$, we can reduce M to $\frac{|E|}{NP}$ without affecting any individual processor, whereas the total amount of memory needed in frameworks like PATRIC and PowerGraph can exceed $|E|$, due to overlapping subgraphs. Finally, it is important to note that the limiting factor after the graphs have been sent to all machines is the processor responsible for the highest number of triangles, so increasing the total number of processors is usually preferable, even with the same amount of total memory, as we also identify in Section V. Our findings are summarized in Theorem IV.3:

Theorem IV.3 (PDTL Complexity)**.** *Using the convention that T represents the number of triangles in the case of triangle listing and 0 in the case of triangle counting, and assuming $\frac{|E|}{NP} > d^*_{max}$, PDTL incurs across all cores a total of:*

- $\Theta\left(NP + N|E| + T\right)$ *Network traffic*
- $\mathcal{O}\left(NP|E| + \frac{|E|^2}{M} + \alpha|E|\right)$ *CPU computations*
- $\mathcal{O}\left(NP\frac{|E|}{B} + \frac{|E|^2}{MB} + \frac{T}{B}\right)$ *I/Os*

V. Evaluation

Due to the wide range of environments in which PDTL can run, our extensive experiments cover single-core, limited memory machines to multi-machine, multi-core, large memory clusters. We discuss our setup and methodology in Section

[3]For the load-balanced approach, this is true in summation, not for individual processors.

[4]If S is smaller than the maximum degree, the complexity changes to $\mathcal{O}\left(NP \cdot \alpha \cdot |E|\right)$ as a single vertex can be split across NP machines, but this would be atypical.

V-A, and introduce our datasets in Section V-B. We discuss the pre-processing and orientation operations in Section V-C, and compare them to those of competing algorithms. In Section V-D we discuss the core properties of our PDTL algorithm, in both the local and distributed environments, including the effects of load balancing. In Section V-E, we compare our algorithm against MGT, OPT, and PowerGraph extensively, and show that PDTL demonstrates superior performance.

A. *Setup and Methodology*

To illustrate the breadth of environments in which PDTL supports triangle counting, we conducted experiments in multiple different clusters and machine configurations:

- **Amazon EC2**: We used 4 Amazon EC2 `c3.8xlarge` instances, each of which contained 32 vCPU units, 60GB of memory, and were connected using a 10 Gigabit Ethernet network. For the PowerGraph measurements, we rented 4 Amazon EC2 `r3.8xlarge` that are similar to `c3.8xlarge` instances, but have 244GB of memory in order to satisfy PowerGraph's memory requirements.
- **Local Cluster**: More distributed experiments were conducted in a local 4-node Linux cluster machine running 8 Virtual Xen nodes, each with 4 cores of an Intel Xeon E5607, 40GB of memory and a Samsung 840 SSD.
- **Local Multicore**: Additional multicore experiments were conducted in a local machine running Linux with 2 AMD Opteron 6344 CPUs for a total of 24 cores, 256GB of memory, and a Samsung 840 SSD.
- **Local Multicore Windows**: Since we only had access to an OPT [14] Windows binary, we used a Windows box, with performance similar to the above, having 2 Intel Xeon E5-2420 CPUs with support for 24 concurrent threads, 128GB memory, and a Samsung 840 SSD.

Our code was compiled with `G++`, using the `-O3` optimization option, and explicitly cleared disk caches before each experiment was run. Though our code fully supports triangle listing, our experiments only measured counting time, to allow comparison with alternative implementations. To account for random variation and general fluctuations, we repeated each experiment 3 times, and present the averages here. More extensive tables of our results can be found in the Appendices.

B. *Datasets*

Table I lists the real and synthetic graphs used for our experiments. Our synthetic RMAT graphs are scale-free graphs produced by the RMAT generator [6], such that RMAT-n contains 2^n vertices and 2^{n+4} edges. Our triangle counts for real graphs have been verified to be correct. Specifically, the Orkut and LiveJournal graphs have reported triangle counts on the SNAP repository [16] that match our values, while the Twitter and Yahoo datasets are in agreement with OPT [14].

Our PDTL framework assumes that graphs are in binary, bi-directional format, with degrees of vertices and their out-edges in separate files. Moreover, we assume that edges are sorted by source and destination, partly for compatibility with the original MGT binary [11]. Since all efficient graph storage techniques operate on binary data, and all counting algorithms require efficient access to neighbors of a vertex, we exclude any time to convert a graph to this format from our discussion.[5] However, because the degree-based ordering is non-standard, we consider the orientation cost separately, and include it in our overall measurements. Similarly, we include copying costs from the master to the clients, to illustrate that our algorithm runs faster, even including graph duplication. Though other architectures such as NFS or HDFS were considered, we store a graph copy locally, since each graph is read at least once per processor. As we see in Table III, the average copying time is up to $10\times$ less than total processing time.

C. *Preprocessing*

Table II presents the time orientation took in our Local Multicore machine with 24 cores, compared to PowerGraph's setup time and OPT's database creation, whose pre-processing steps are much slower. Figure 2 shows a $5.2\times$ speed-up of multicore orientation over the single-core solution, and illustrating that our SSD is capped at 16 threads and 500MB/s.

Graphs	d^*_{max}	PDTL	PowerGraph	OPT
LiveJ1	687	1.4s	-	1m46.8s
Orkut	535	3.6s	25.7s	43.6s
Twitter	4,102	32.8s	3m53.2s	7m17.6s
Yahoo	1,540	3m55.6s	-	-
RMAT-26	2,964	29.3s	3m33.0s	15m10.3s

TABLE II: Preprocessing time: PDTL (Orientation), PowerGraph (Setup), OPT (Database Creation)

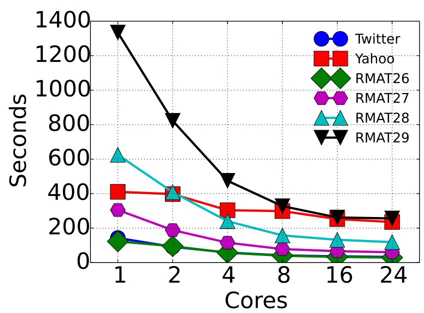

Fig. 2: PDTL in Local Multicore: Orientation

D. *PDTL Properties*

In this section we examine the properties exhibited by PDTL without comparing it to other systems.

1) Local: We tested weak scaling of PDTL in the Local Multicore machine, by increasing the number of cores, but keeping the total amount of memory constant at 128GB, as shown in Figure 3. Specifically, using 2 cores halves calculation times, and this effect persists at a decreasing rate. Synthetic graphs exhibit better speedups due to their scale-free nature, as does the Twitter graph. However, due to its structure, the Yahoo graph only exhibits a $5\times$ speedup at 24 cores, compared to a $13\times$ speedup for the other graphs.

[5] Note that OPT [14] requires that the input be sorted by vertex degree which is not included in the measurements, so we believe this is a fair starting point.

Graph	Nodes	Edges	Triangles	Size	AvDeg	STD	MaxDeg	Source
soc-LiveJournal1	4.8M	68.0M	285,730,264	365MB	17.8	52	20,334	[16]
com-Orkut	3.1M	117.2M	627,584,181	917MB	76.0	155	33,313	[16]
Twitter	61.6M	1.5B	34,824,916,864	9.4GB	57.7	402	2,997,487	[15]
Yahoo	1.4B	6.6B	85,782,928,684	59GB	17.9	279	7,637,656	[1]
RMAT-26	67.1M	1.1B	51,559,452,522	8.4GB	61.2	632	430,269	[6]
RMAT-27	134.2M	2.1B	114,007,006,286	17GB	63.6	601	676,199	[6]
RMAT-28	268.4M	4.3B	251,913,686,661	34GB	66.0	660	1,062,289	[6]
RMAT-29	536.9M	8.6B	556,443,109,053	68GB	69.0	782	1,665,635	[6]

TABLE I: Graphs used for the experiments

Graph	1 node	2 nodes		3 nodes		4 nodes	
	Total time	Total time	Avg copy time	Total time	Avg copy time	Total time	Avg copy time
Twitter	2m44.2s	2m07.4s	13.5s	1m56.0s	16.2s	1m49.0s	19.1s
Yahoo	6m37.9s	6m04.9s	1m46.0s	5m38.6s	1m52.4s	7m13.8s	3m06.4s
RMAT-26	6m10.4s	3m29.7s	14.7s	2m46.8s	16.7s	2m31.3s	19.0s
RMAT-27	14m01.1s	7m59.5s	27.9s	6m13.1s	27.9s	5m06.6s	33.3s
RMAT-28	31m16.8s	17m57.4s	52.1s	13m34.4s	57.2s	11m12.1s	1m08.3s
RMAT-29	1h17m24.5s	42m11.0s	1m46.6s	31m00.3s	2m18.4s	26m05.0s	2m34.6s

TABLE III: PDTL in EC2: Total time and average copy time per remote node

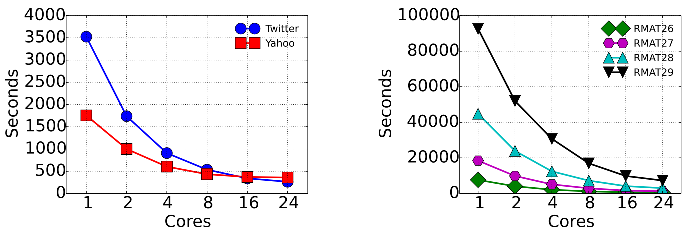

Fig. 3: PDTL in Local Multicore: Total Time

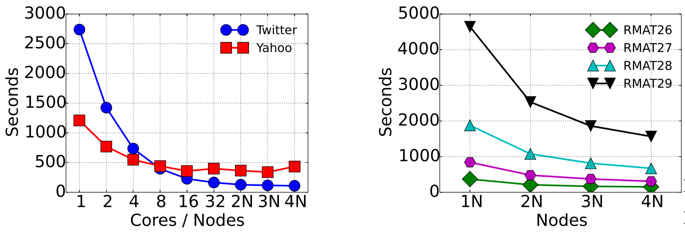

Fig. 4: PDTL in EC2: Total Time

2) Distributed: We also ran distributed experiments in Amazon EC2, with 1GB of memory/core, as shown in Figure 4. We observe the following:

- The Twitter graph shows good scalability, while the Yahoo graph, being sparser and having a low average degree, does not benefit from adding more than 16 cores. Consequently, the structure of each graph heavily affects processing time, as also indicated by our analysis.
- RMAT graphs are much denser and more computationally intensive. This results in good scalability even up to 128 cores (4 nodes), as copy overhead (included) is negligible.
- Comparing total computation time (Table III) and orientation time (Table II), further illustrates the unusual behavior of the Yahoo graph compared to Twitter and the RMAT graphs. Specifically, even though orientation only represents a small proportion of overall runtime for the latter two graphs, it comes close to 50% for the Yahoo graph on 3 nodes.
- Table III also details the copy times (averaged over the number of non-master nodes) for each of our graphs. As expected, this number scales with increasing graph size (recall Yahoo is larger than RMAT-28, but smaller than RMAT-29), and with increasing number of nodes (due to more limited network bandwidth). The Yahoo graph also presents an anomaly in the copying of the graph which results in much higher than expected increase in copy time for 4 nodes due to improper I/O balancing of the master node (Section V-D4): since the Yahoo graph results in heavy I/O due to its structure, it causes an initial I/O bottleneck when the master is performing its computations while also copying the graph to the nodes.

3) Memory: To identify the effect of limited memory, we ran experiments in our Local Cluster varying the number of nodes, and the total amount of memory available per node (fixing $P = 4$ cores/node). As can be seen in Figure 5, the effect of limiting memory is negligible, and as a matter of fact more memory can lead to slightly higher costs due to array initialization overhead, as indicated in Section IV-B2.

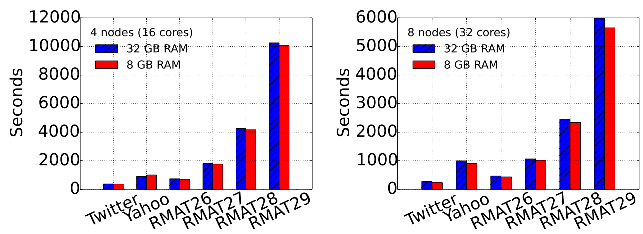

Fig. 5: PDTL in Local Cluster: Memory vs. Calc Time

4) I/O and CPU: Despite the fact that PDTL is an external-memory algorithm, in our Amazon EC2 experiments we discovered that it is not I/O-bound. Specifically, we measured the total I/O for different numbers of cores and nodes on the Yahoo and Twitter graphs, and we found that it represents a small percentage of the computation time (Figure 6). As explained in Section IV-B2 and verified here, the absolute time spent on I/O operations increases as the number of cores increases, and is tied to the concrete graph structure, as indicated by the difference between the Twitter and Yahoo graphs.

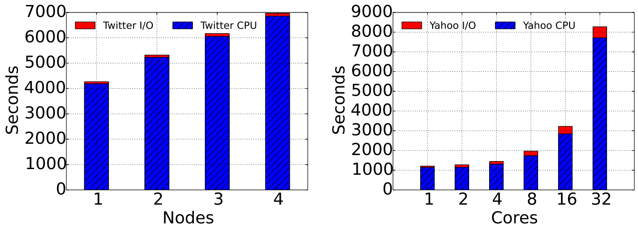

Fig. 6: PDTL in EC2: Total CPU and I/O breakdown for various number of cores and nodes

Figures 7 and 8 show the per node I/O and CPU breakdown for Twitter and Yahoo respectively. We notice that for the Twitter graph, our load-balancing mechanism works fairly well, and there is no correlation between the CPU and the I/O operation times. However, the Yahoo graph is heavily skewed, and higher I/Os appear at the nodes with highest computation times, further illustrating the point that the concrete graph structure heavily influences the overall runtime of our algorithm.

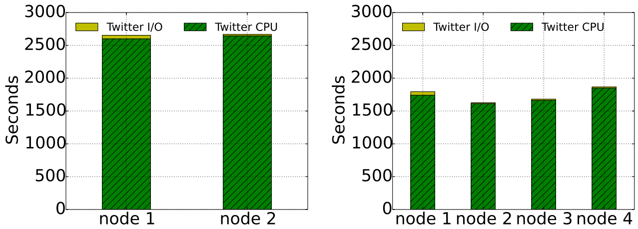

Fig. 7: PDTL in EC2: Twitter CPU and I/O breakdown

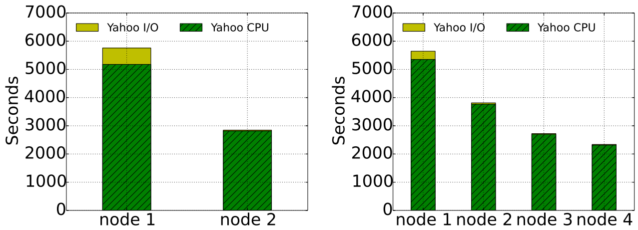

Fig. 8: PDTL in EC2: Yahoo CPU and I/O breakdown

5) Load Balancing: We compared in our Local Multicore machine the naive approach of allocating the same number of edges for each core to our load balancing solution. Figure 9 contains our findings for 16 and 24 cores, and clearly illustrates up to a $3\times$ improvement on the calculation time, even for the Yahoo graph.

Table IV details the total I/O and CPU computations for all processors within each node in Amazon EC2. As can be seen, our load-balancing mechanism leaves room for improvement, since the discrepancies between nodes increase as more nodes are added: even though there is only a 1% difference for 2

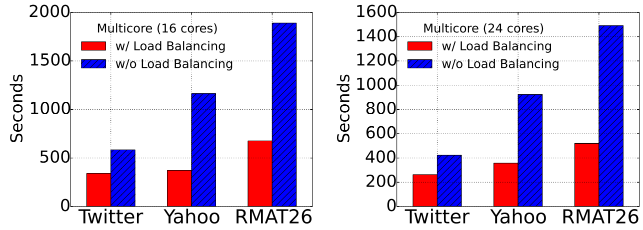

Fig. 9: PDTL in Local Multicore: Load Balancing

Twitter nodes, the difference increases to 13% for 4 nodes, while for Yahoo, the number increases from 87% to 130%.

E. PDTL Comparisons

In this section we examine our PDTL algorithm in the context of competing frameworks.

1) MGT: To compare PDTL against single-core MGT, we conducted experiments in Amazon EC2 nodes, looking at just the calculation times. Figure 10 shows that using just 2 processors halves the processing time for all real graphs, and using 32 cores provides a $16\times$ speedup for the Twitter graph. Figure 11 similarly shows that the speedup for calculations of distributed, multicore PDTL over MGT reaches up to $55\times$ with 4 nodes. This effect is especially pronounced for scale-free RMAT graphs, whereas speedups reach $30\times$ for Twitter, but only $4\times$ for Yahoo. It should be noted that the comparison here is against our implementation of MGT, because the provided MGT binary [11] misreported triangle counts for some of the larger graphs.[6] As a result, we cannot directly compare our implementation to the baseline one (or other external-memory algorithms, which were shown to be slower than MGT [13]), but for completeness we note that for small graphs the performance was similar to the given binary.

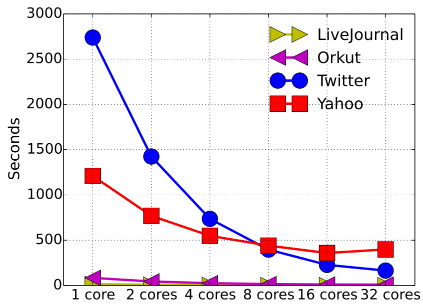

Fig. 10: PDTL in EC2: Single Node Performance

2) OPT: We compared our multi-core algorithm to OPT [14] in our two Local Multicore machines. We measured setup time (orientation for PDTL and database creation for OPT) and calculation time separately, and report our results when using 24 cores in Table V. With the exception of the LiveJournal dataset, our calculation time is always (and up to $2\times$) faster

[6] MGT reported $627,506,739$ triangles for Orkut (compared to $627,584,181$) and $559,420,538$ triangles for Twitter (compared to 34.8B).

Graph	2 nodes		3 nodes			4 nodes			
CPU									
Twitter	43m19.9s	44m03.0s	33m36.7s	32m00.4s	35m32.1s	29m01.0s	26m53.1s	27m43.5s	30m48.8s
Yahoo	1h26m17.7s	46m57.8s	1h28m16.2s	59m05.4s	40m12.4s	1h29m12.0s	1h02m45.8s	44m58.7s	38m37.0s
RMAT-26	1h26m13.9s	1h8m35.3s	1h4m23.1s	53m01.3s	45m58.7s	53m57.1s	44m34.3s	40m14.3s	34m59.3s
I/O									
Twitter	53.5s	23.7s	54.43s	22.4s	23.0s	53.5s	13.3s	17.8s	17.3s
Yahoo	9m40.8s	28.3s	7m2.4s	32.6s	24.7s	4m53.1s	48.1s	24.9s	22.4s
RMAT-26	2m35.2s	29.2s	2m23.5s	22.1s	20.9s	1m25.2s	19.6s	18.4s	15.3s

TABLE IV: PDTL in EC2: Per node total CPU and I/O breakdown

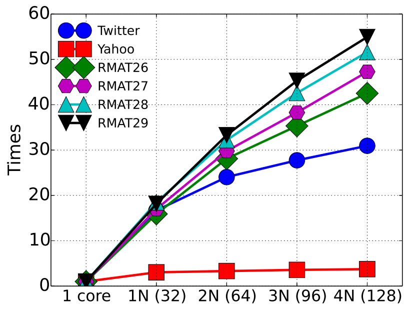

Fig. 11: PDTL in EC2: Speedup over MGT

than OPT's calculations, and our setup time is up to $75\times$ faster. When looking at the total time, PDTL is up to $3.5\times$ faster for large graphs (and $7.8\times$ faster for LiveJournal). As can be seen in Figure 12, these effects remain for any number of cores, though they are even more pronounced for fewer ones. We should note that the OPT binary we received occasionally gave inconsistent triangle counts and could not run with $M = 128GB$, hence the memory discrepancy in our testing.

Graph	PDTL Orientation	PDTL Calc	OPT Database	OPT Calc
LiveJ1	1.4s	12.4s	1m46.8s	3.3s
Orkut	3.6s	11.4s	43.6s	11.7s
Twitter	32.8s	4m22.9s	3m55.2s	7m17.6s
Yahoo	3m55.6s	5m57.9s	-	-
RMAT-26	29.3s	8m40.4s	15m10.3s	16m51.2s

TABLE V: Local Multicore: PDTL and OPT Performance

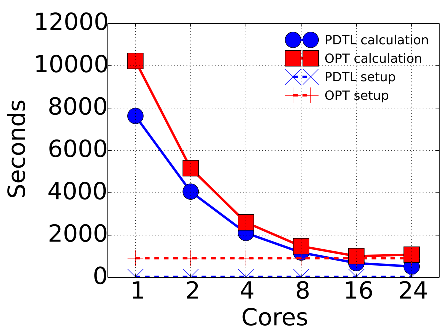

Fig. 12: Local Multicore: PDTL (128GB) and OPT (100GB) on RMAT-26

3) PowerGraph: We also compared our distributed framework to PowerGraph [10] in Local Cluster and Amazon EC2. To make a fair comparison, we consider two measures: the total runtime of both programs (including orientation in PDTL's case), and the pure calculation time (including load-balancing costs for PDTL). For PowerGraph, the calculation time is the reported time of the triangle counting algorithm. For PDTL, the overall calculation time corresponds to the maximum individual calculation time between the different nodes. This is because the nodes start calculating as soon as they receive the files and, thus, the calculation time of the "struggler" node determines entirely the overall calculation time. The value $total - calc$ thus represents the setup time for PowerGraph, while it represents a combination of network costs and workload imbalance for PDTL.

Figure 13 shows that although calculation times are similar (with PDTL presenting an advantage as the graphs become bigger), with setup times, PDTL is more than $2\times$ faster. Table VI illustrates this point more clearly, and also highlights that for larger graphs, PowerGraph runs out of memory. This is especially noteworthy, given that PowerGraph experiments were run on nodes with 244GB of memory **each** for a total of 976GB, while our PDTL experiments were run using only 1GB/core (with much lower requirements) for a total of 128GB of memory. This validates our analysis in Section IV-B2, clearly illustrating that partitioning-based approaches do not work for large graphs, and that external-memory algorithms like PDTL are needed.

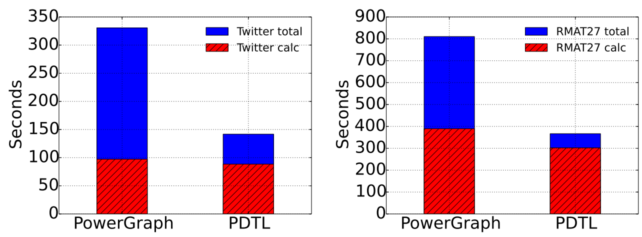

Fig. 13: EC2 (4N): PDTL and PG breakdowns

4) Other Frameworks: Although we could not obtain a copy of the PATRIC binary, the original paper [3] indicates that PATRIC counts the triangles in the Twitter graph in 9m24s using 200 cores, and 4GB of memory/core. In another recent experiment [14], PATRIC was run in a cluster of 31 nodes with 12 threads per node (372 threads total) and 2GB of memory/core for a time of 10m8s. In either case, we notice

Graph	PDTL		PowerGraph	
	Calc	Total	Calc	Total
Orkut	6.9s	11.7s	4.9s	30.6s
Twitter	1m28.5s	2m21.8s	1m37.3s	5m30.5s
Yahoo	5m23.9s	11m09.4s	F	F
RMAT-26	2m18.6s	3m00.6s	2m56.7s	6m29.7s
RMAT-27	5m02.1s	6m06.7s	6m29.5s	13m30.5s
RMAT-28	11m12.1s	13m10.9s	F	F
RMAT-29	25m33.5s	30m21.2s	F	F

TABLE VI: PDTL and PowerGraph in EC2. F represents an out-of-memory exception

that PDTL is $4\times$ faster using only 96 cores and 1GB of memory/core, and is still faster even when the number of cores is reduced to 8, again highlighting our fast performance under low memory requirements. Finally, it is worth briefly mentioning that MapReduce-based algorithms such as CTTP [20] are not competitive, spending 92m calculating Twitter triangles using 40 nodes with 4GB of memory/node.

VI. Conclusions and Future Work

In this paper, we presented our Parallel and Distributed Triangle Listing (PDTL) framework, the first distributed triangle listing and counting algorithm that focuses on external-memory I/O efficiency, but also provides theoretical CPU and Network guarantees. Our framework works well in a variety of computational environments, and is based on the recent MGT algorithm [13]. Key to our engineering approach is the combination of both a distributed setting and external memory. This gives us a high amount of parallelism whilst freeing us from the usual distributed constraint of fitting entire subgraph in memory. Our resulting implementation is scalable and performs especially well in low-memory scenarios. As graphs become larger, the requirement of fitting even parts of a graph in memory will no longer be viable, as we also verified experimentally: our algorithm was able to accommodate for massive graphs containing over 8 billion edges with little memory, while competing partitioning-based frameworks ran out of memory even using almost 1TB of RAM.

More generally, our extensive experiments demonstrate that PDTL is highly scalable across multiple cores and machines, with low memory requirements, even for graphs with hundreds of millions of vertices, and billions of edges. In particular, over the Twitter data set [15], our algorithm is faster than all of the state-of-the-art algorithms in distributed and parallel triangle counting: PDTL is $4\times$ faster than PATRIC [3], $3\times$ faster than OPT [14], and $2\times$ faster than PowerGraph [10].

Future work could focus investigation on different types of disks and file systems (for instance distributed file systems, or lazy evaluation), as a means of removing any copying bottlenecks that may exist. Such research could be informed by PowerGraph's general-framework, which fares better compared to triangle-specific systems. Even though its high memory requirements influence the results, it would be interesting to more formally investigate this. Additionally, more detailed investigations could try different techniques of load balancing, and provide a better understanding of the optimal number of machines and cores for any given graphs. As we identified in our experiments, scale-free graphs scale extremely well, even up to 8 machines, while the real-world Yahoo graph [1] exhibits a slowdown at even 4 nodes.

Overall, our framework provides a starting point towards many directions, including altering it for dynamic or approximate triangle counting, but more importantly for investigating other graph algorithms and processing systems which can benefit from our disk-based approach for large datasets.

Acknowledgments. We would like to thank Marton Havasi for his help with data processing. The research is part funded by the EPSRC DDEPI Project, EP/H003959, and by the Goulandris Scholarship at Magdalene College.

References

[1] Yahoo! Webscope. http://webscope.sandbox.yahoo.com/.

[2] Aggarwal, A., and Vitter, Jeffrey, S. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 9 (Sept. 1988).

[3] Arifuzzaman, S., Khan, M., and Marathe, M. PATRIC: A parallel algorithm for counting triangles in massive networks. CIKM '13, ACM.

[4] Batagelj, V., and Zaveršnik, M. Short cycle connectivity. *Discrete Mathematics 307*, 3-5 (2007).

[5] Becchetti, L., Boldi, P., Castillo, C., and Gionis, A. Efficient semi-streaming algorithms for local triangle counting in massive graphs. KDD '08, ACM.

[6] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: A recursive model for graph mining. SIAM '04.

[7] Chiba, N., and Nishizeki, T. Arboricity and subgraph listing algorithms. *SIAM J. Comput. 14*, 1 (Feb. 1985).

[8] Chu, S., and Cheng, J. Triangle listing in massive networks. *ACM Trans. Knowl. Discov. Data 6*, 4 (Dec. 2012).

[9] Dementiev, R. *Algorithm engineering for large data sets*. PhD thesis, Saarland University, 2006.

[10] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. PowerGraph: Distributed graph-parallel computation on natural graphs. OSDI'12, USENIX Association.

[11] Hu, X., Tao, Y., and Chung, C.-W. MGT executable binary. http://appsrv.cse.cuhk.edu.hk/~taoyf/paper/codes/trilist/trilist.zip.

[12] Hu, X., Tao, Y., and Chung, C.-W. MGT manual. http://appsrv.cse.cuhk.edu.hk/~taoyf/paper/codes/trilist/manual.

[13] Hu, X., Tao, Y., and Chung, C.-W. Massive graph triangulation. SIGMOD '13, ACM.

[14] Kim, J., Han, W.-S., Lee, S., Park, K., and Yu, H. Opt: A new framework for overlapped and parallel triangulation in large-scale graphs. SIGMOD '14, ACM.

[15] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (2010), ACM.

[16] Leskovec, J. SNAP: Stanford large network dataset collection. http://snap.stanford.edu/data/. Accessed: 2014-06-03.

[17] Menegola, B. An external memory algorithm for listing triangles. Tech. rep., Universidade Federal do Rio Grande do Sul, 2010.

[18] Opsahl, T., and Panzarasa, P. Clustering in weighted networks. *Social Networks 31*, 2 (2009).

[19] Pagh, R., and Silvestri, F. The input/output complexity of triangle enumeration. *CoRR abs/1312.0723* (2013).

[20] Park, H.-M., Silvestri, F., Kang, U., and Pagh, R. Mapreduce triangle enumeration with guarantees. CIKM '14, ACM.

[21] Shao, Y., Cui, B., Chen, L., Ma, L., Yao, J., and Xu, N. Parallel subgraph listing in a large-scale graph. SIGMOD '14, ACM.

[22] Wang, J., and Cheng, J. Truss decomposition in massive networks. *Proc. VLDB Endow. 5*, 9 (May 2012).

[23] Wang, N., Zhang, J., Tan, K.-L., and Tung, A. K. H. On triangulation-based dense neighborhood graph discovery. *Proc. VLDB Endow. 4*, 2 (Nov. 2010).

[24] Watts, D., and Strogatz, S. Collective dynamics of 'small-world' networks. *Nature*, 393 (1998).

[25] Yang, Z., Wilson, C., Wang, X., Gao, T., Zhao, B. Y., and Dai, Y. Uncovering social network sybils in the wild. *ACM Trans. Knowl. Discov. Data 8*, 1 (Feb. 2014).

Appendix A
EC2: PDTL and OPT

PDTL experiments in EC2 use 16GB of total memory for 1 node, and 1GB/core for multiple nodes.

Graph	**Cores**						**Nodes**		
	1	2	4	8	16	32	2	3	4
CPU									
Twitter	45m02.1s	45m17.4s	46m17.5s	48m05.6s	50m08.8s	1h09m50.5s	1h27m12.6s	1h40m59.1s	1h54m16.3s
Yahoo	19m06.5s	19m14.3s	21m53.2s	28m58.0s	47m24.4s	2h08m33.9s	2h12m05.7s	3h06m23.3s	3h54m22.7s
I/O									
Twitter	36.0s	26.5s	36.5s	46.9s	1m10.0s	1m17.3s	1m27.6s	1m50.2s	1m52.3s
Yahoo	1m02.5s	2m02.7s	2m12.8s	3m48.2s	6m18.4s	9m20.1s	11m19.0s	9m10.6s	7m39.5s

TABLE VII: PDTL in EC2: Total CPU and I/O breakdown for various number of cores and nodes

Graph	**Cores**						**Nodes**		
	1	2	4	8	16	32	2	3	4
LiveJ1	15.1s	8.5s	5.3s	3.6s	2.8s	2.6s	3.0s	3.0s	3.1s
Orkut	1m23.8s	44.0s	25.1s	15.0s	10.4s	9.6s	8.4s	8.0s	8.1s
Twitter	45m38.5s	23m44.3s	12m15.7s	6m35.9s	3m47.4s	2m44.2s	2m07.4s	1m56.0s	1m49.0s
Yahoo	20m09.8s	12m49.4s	9m08.2s	7m20.1s	5m58.3s	6m37.9s	6m04.9s	5m38.6s	7m13.8s
RMAT-26	1h38m12.8s	-	-	-	-	6m10.4s	3m29.7s	2m46.8s	2m31.3s
RMAT-27	3h58m00.6s	-	-	-	-	14m01.1s	7m59.5s	6m13.1s	5m06.6s
RMAT-28	9h38m05.4s	-	-	-	-	31m16.8s	17m57.4s	13m34.4s	11m12.1s
RMAT-29	23h24m02.5s	-	-	-	-	1h17m24.5s	42m11.0s	31m00.3s	26m05.0s
Twitter (OPT)	53m55.1s	27m16.2s	13m49.8s	7m23.0s	5m00.1s	3m11.4s	N/A	N/A	N/A

TABLE VIII: EC2: PDTL and OPT (12GB)

Appendix B
PDTL Local Multicore: Runtime, Orientation, and Load balancing

Graph	d^*_{max}	**1 core**	**2 cores**	**4 cores**	**8 cores**	**16 cores**	**24 cores**
LiveJ1	687	3.5s	2.5s	2.2s	2.0s	1.6s	1.4s
Orkut	535	9.4s	6.8s	4.6s	3.7s	3.6s	3.6s
Twitter	4,102	2m22.9s	1m31.5s	58.6s	41.8s	36.4s	32.8s
Yahoo	1,540	6m50.5s	6m37.6s	5m03.7s	4m58.8s	4m12.5s	3m55.6s
RMAT-26	2,964	2m03.0s	1m35.2s	57.0s	39.9s	32.9s	29.3s
RMAT-27	3,855	5m05.1s	3m08.5s	1m55.5s	1m18.4s	1m06.3s	1m00.1s
RMAT-28	4,984	10m25.1s	6m48.6s	4m00.2s	2m37.7s	2m12.3s	1m58.8s
RMAT-29	6,389	22m14.6s	13m42.5s	7m54.6s	5m27.5s	4m21.7s	4m16.2s

TABLE IX: PDTL in Local Multicore: Orientation

Graph	**16 cores**		**24 cores**	
	w/ LB	w/o LB	w/ LB	w/o LB
Twitter	9m44.4s	5m39.6s	7m03.1s	4m22.9s
Yahoo	19m22.7s	6m10.8s	15m23.9s	5m57.9s
RMAT-26	31m30.6s	11m16.7s	24m51.9s	8m40.4s

TABLE X: PDTL in Local Multicore: Runtime with and without load balancing (128GB memory)

Graph	**1 core**	**2 cores**	**4 cores**	**8 cores**	**16 cores**	**24 cores**
LiveJ1	37.7s	27.9s	21.2s	16.8s	13.5s	12.4s
Orkut	1m46.9s	58.3s	33.4s	21.2s	13.7s	11.4s
Twitter	58m45.3s	28m57.9s	15m08.0s	8m55.0s	5m39.6s	4m22.9s
Yahoo	29m15.5s	16m40.6s	10m04.0s	7m13.0s	6m10.8s	5m57.9s
RMAT-26	2h07m08.2s	1h07m35.0s	35m03.0s	19m37.3s	11m16.7s	8m40.4s
RMAT-27	5h07m40.6s	2h44m08.1s	1h25m16.0s	48m15.0s	27m04.0s	20m52.9s
RMAT-28	12h26m43.8s	6h38m52.6s	3h27m21.0s	2h00m20.9s	1h08m54.8s	49m58.8s
RMAT-29	25h42m50.9s	14h26m31.1s	8h31m50.7s	4h42m16.4s	2h43m44.8s	2h01m15.3s

TABLE XI: PDTL in Local Multicore machine (128GB Memory)

APPENDIX C
LOCAL CLUSTER: POWERGRAPH AND PDTL RUNTIME

Graph	**Nodes** 2	4	8
LiveJ1	8.6s	10.9s	15.2s
Orkut	31.9s	29.7s	37.7s
Twitter	11m08.8s	7m58.9s	8m02.3s
Yahoo	22m18.8s	23m27.1s	40m07.3s
RMAT-26	22m34.8s	13m22.8s	11m43.7s
RMAT-27	54m15.7s	31m02.4s	24m55.7s
RMAT-28	2h9m18.4s	1h12m25.2s	55m50.2s
RMAT-29	5h16m07.5s	2h53m01.9s	2h04m23.7s

TABLE XII: PDTL in Local Cluster: 8GB of Memory/ Node

Graph	**Cores**		**Nodes**		
	1	4	2	4	8
LiveJ1	29.0s	13.6s	12.5s	13.0s	18.3s
Orkut	2m06.4s	41.3s	29.4s	31.0s	39.0s
Twitter	1h11m47.0s	19m33.3s	11m40.7s	8m17.5s	8m26.4s
Yahoo	40m42.0s	35m03.3s	21m28.5s	22m39.4s	37m44.8s
RMAT-26	2h39m36.8s	43m56.4s	22m41.9s	13m41.4s	11m53.3s
RMAT-27	6h22m37.6s	1h47m30.7s	54m35.8s	31m59.8s	25m59.3s
RMAT-28	15h34m26.6s	4h19m12.0s	2h10m46.6s	1h14m34.6s	57m21.2s
RMAT-29	37h33m37.0s	10h29m09.1s	5h16m57.3s	2h53m09.1s	2h5m03.6s

TABLE XIII: PDTL in Local Cluster: 32GB Memory/ Node

Graph	**PDTL**			**PowerGraph**	
	Orientation	Calc	Total	Calc	Total
LiveJ1	1.4s	22.5s	23.9s	7.0s	21.3s
Orkut	3.6s	17.7s	44.9s	18.9s	42.7s
Twitter	32.8s	5m02.5s	9m37.2s	6m33.9s	10m22.2s
Yahoo	3m55.6s	19m33.2s	42m11.5s	F	F
RMAT-26	29.3s	8m31.1s	11m55.2s	F	F
RMAT-27	1m00.1s	19m45.9s	26m31.3s	F	F
RMAT-28	1m58.8s	45m39.6s	59m36.1s	F	F
RMAT-29	4m16.2s	1h51m43.6s	2h15m48.9s	F	F

TABLE XIV: Local Cluster (7N): PDTL (32GB/N) and PowerGraph (40GB/N). F represents an out-of-memory exception