LOTUS: Locality Optimizing Triangle Counting

Mohsen Koohi Esfahani
mkoohiesfahani01@qub.ac.uk

Peter Kilpatrick
p.kilpatrick@qub.ac.uk

Hans Vandierendonck
h.vandierendonck@qub.ac.uk

Queen's University Belfast
Northern Ireland, United Kingdom
https://blogs.qub.ac.uk/GraphProcessing/

Abstract

Triangle Counting (TC) is a basic graph mining problem with numerous applications. However, the large size of real-world graphs has a severe effect on TC performance.

This paper studies the TC algorithm from the perspective of memory utilization. We investigate the implications of the skewed degree distribution of real-world graphs on TC and make novel observations on how memory locality is negatively affected. Based on this, we introduce the *LOTUS* algorithm as a structure-aware TC that optimizes locality.

The evaluation on 14 real-world graphs with up to 162 billion edges and on 3 different processor architectures of up to 128 cores shows that Lotus is 2.2–5.5× faster than previous works.

CCS Concepts: • **Theory of computation** → **Graph algorithms analysis**; **Shared memory algorithms**; **Massively parallel algorithms**.

Keywords: Graph Algorithms, High Performance Computing, Memory Locality, Triangle Counting, Real-World Graphs, Graph Mining, Clique Problem

1 Introduction

Triangle Counting (TC) is one of the fundamental problems in graph processing which is used in several fields of science, humanities, and technology [11, 12, 20, 24, 29, 30, 54, 57, 73, 75]. Different algorithms and optimizations have been proposed in the literature [1, 5, 8, 12, 23, 27, 31, 34, 37, 48, 62]. However, efficient TC is still a challenge for large and fast-growing real-world graph datasets.

The structure of real-world graphs poses further challenges to TC. Many real-world graphs derived from social networks, the internet, and the world-wide web show a skewed or heavy-tailed degree distribution, following a **power-law distribution**: a very small fraction of vertices (known as hubs) are connected to a disproportionately large fraction of edges of the graph.

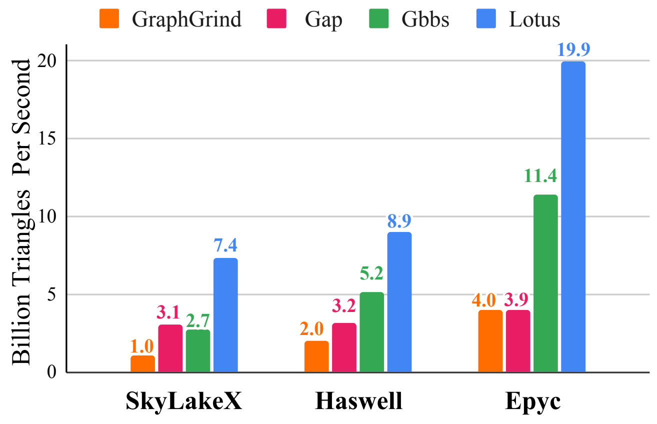


Figure 1. Average TC rates

In this paper, we study the effects of power-law degree distribution of the graphs on TC. We identify a number of **shortcomings** in state-of-the-art TC algorithms and **opportunities** for improvement, in particular: (1) poor memory locality, especially during traversal of triangles containing non-hub vertices, (2) (lack of) compactness of the graph topology representation, and (3) opportunity to prune searches that cannot uncover triangles.

The **key observation** behind this work is based on the skewedness of the degree distribution of graphs, which has specific consequences for TC: (1) on average, 93.4% of the triangles include a hub vertex, and (2) edges between hub vertices shape a network that is, on average, 1809× denser than the whole graph. These observations lead us to see hubs as a key to efficient TC.

This paper presents the design of **LOTUS**, a structure-aware and locality-optimizing TC algorithm. Lotus improves TC by distinguishing 4 types of triangles (depending on whether they include 3, 2, 1 or 0 hubs) and splits TC into multiple phases, corresponding to the types of triangle. Each TC phase is designed as a stand-alone TC algorithm, using bespoke, compact data structures. The algorithms and data structures are designed such that **random memory**

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508402

accesses, a key challenge for memory locality in graph processing, are targeted towards a small data structure in each TC phase to utilize hardware caches in the best possible way.

We evaluate Lotus with TC algorithms in frameworks such as GraphGrind [66], GAP [10], and GBBS [26] on 14 graphs of up to 162 billion edges and on 3 processor architectures: Intel SkyLakeX, Intel Haswell, and AMD Epyc with 32-128 cores. Figure 1 summarizes the average TC rates based on the end-to-end execution time including preprocessing step for the datasets with fewer than 10 billion edges.

The **contributions** of this work are:

- Studying the effects of skewed degree distribution of real-world graph datasets on TC,
- Analysing the relationship between different vertex classes from the perspective of memory utilization in common neighbour computation,
- Introducing Lotus, a novel structure-aware TC algorithm that optimizes locality for skewed degree graphs by separating processing hub edges from non-hub edges,
- Introducing Squared Edge Tiling as a graph partitioning algorithm for optimizing load balance in TC, and
- Evaluating the Lotus algorithm in comparison to previous works.

This paper is structured as follows: Section 2 explains key background materials. Section 3 explores opportunities for improving TC by introducing novel features of power-law graphs. Section 4 introduces the Lotus algorithm which is evaluated in Section 5. Section 6 discusses further related work and avenues for future work are presented in Section 7.

2 Background

2.1 Terminology

An undirected graph $G = (V, E)$ has a set of vertices V, and a set of edges E between these vertices. Edge (v, u) is the symmetric edge of edge (u, v), where $u < v$.

N_v is the set of neighbours of vertex v, $N_v^< = \{u \in N_v | u < v\}$, and $N_v^> = \{u \in N_v | u > v\}$. We use the CSX (compressed sparse rows or columns) representation [59].

Vertices are divided into (1) *hub* and (2) *non-hub* vertices. The Lotus algorithm (Section 4.2) specifies how a vertex is selected as hub. An edge can be in one of 3 forms: (1) *hub to hub* edge, (2) *hub to non-hub* edge, or (3) *non-hub to non-hub* edge. A *hub edge* is an edge with at least one hub endpoint. A non-hub edge is an edge without any hub as its endpoints. A triangle is called *hub triangle* if at least one of its vertices is a hub vertex.

2.2 TC Algorithms

Three main TC algorithms are summarized in [62]:

- The *Node iterator* algorithm enumerates each pair of neighbours of a vertex and checks if they are connected,
- The *Edge iterator* algorithm searches for common neighbours of two endpoints of each edge, and

Algorithm 1: Forward algorithm

```
Input: $G(V, E)$
Output: Triangles
1 $G'(V', E')$ = reorder_by_degree($G$);
2 $triangles = 0$;
3 par_for $v \in V'$
4     par_for $u \in N_v^<$
5         $triangles += |N_v^< \cap N_u^<|$;
6 return $triangles$;
```

- The *Forward* algorithm sorts vertices by their degrees in descending order and identifies common neighbours between a vertex and each of its neighbours.

Algorithm 1 is an improved version of the Forward algorithm introduced in [27] that we use as the baseline algorithm. For each vertex v and each $u \in N_v^<$, $N_v^< \cap N_u^<$ specifies the number of triangles including u and v. By limiting neighbours to $N^<$, a triangle is counted only once and the execution time is reduced as only half of the edges are processed. The intersection is performed using merge join [62], bitmap lookup [48], hashing [23, 62], or binary search [31].

3 Motivation

This section presents the key observations that underpin the design of Lotus.

3.1 Low Locality in Processing Non-Hub Vertices

The Forward algorithm (Algorithm 1) uses degree ordering to accelerate TC. Degree ordering improves load balance and reduces the number of comparisons occurring in the intersection operation [5].

For each edge, only one of (u, v) or (v, u) needs to be present in the graph as the symmetric edges are redundant for TC. Degree ordering decides that (v, u) is retained if $v < u$. By consequence, only vertices v with a smaller ID ($v < u$) are stored in the neighbour list of a vertex u. The neighbour list of a hub thus only contains hubs, and the neighbour list of a non-hub contains both hub and non-hub neighbours.

This is illustrated in an example graph (Figure 2) where edges have the same color as the vertex they are assigned to by degree ordering. Edges between vertex 3 and hub vertices 0 and 1 are assigned to vertex 3 and vertex 3 also has an edge to vertex 2 which is a non-hub.

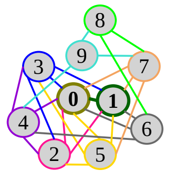


Figure 2. An example graph - hubs: 0, 1

An immediate consequence of this organization is that the neighbour lists of hubs (containing hub-to-hub edges) are very frequently accessed. Indeed, hubs have many neighbours, which by construction are mostly non-hubs. Each time a non-hub vertex v that is a neighbour of a hub u is processed, the neighbour list of the hub is accessed (Algorithm 1, Line 5). Frequent accesses to the hubs' neighbour lists prompt the processor cache to **maintain hub neighbour lists in cache**. Column 2 of Table 1 shows that these neighbour lists, consisting of hub-to-hub edges, include 18.1% of the edges of the graph.

The flip-side of this is that **there is a low opportunity for the cache to retain neighbour lists of non-hubs**. While each non-hub neighbour list is accessed less frequently, together they constitute 81.9% of the edges of power-law graphs (Table 1, Columns 3 and 5).

3.2 Lack of Compactness of Graph Topology

In graph algorithms like breadth-first search, single-source shortest-path, and connected components, the main memory access challenge consists of random memory accesses to vertex data, which typically consists of 1–64 bits per vertex. Random memory accesses thus target a data set of size proportional to the number of vertices.

In contrast, the data accessed per vertex consists of the neighbour lists, i.e., the graph topology, in TC. Thus, **random memory accesses in TC target a much larger data set of size proportional to the number of edges**. This shows why achieving memory locality in TC is both more challenging and more important.

Driven by this comparison, we question whether it is possible to represent neighbour lists more compactly, without incurring overheads. The key to this is again in the hubs: the number of hubs is very few, but the majority of the IDs in neighbour lists of any vertex refer to hubs. In the power-law graphs used in this study, **1% of vertices are connected to 72.9% of edges** (Table 1, Column 4).

Drawing on the principles of coding and compression theory [35], **it is wasteful to represent highly frequently occurring IDs using the same bitwidth as rarely occurring IDs**. The penalty for doing so is inefficient cache utilization. While we reference coding theory to explain the problem, **it is important to design techniques that do not incur runtime overhead** to read graph topology data, as this is the main operation in TC.

3.3 Fruitless Searches

Out of the many neighbour list intersections performed, a relatively low number of triangles is found. Conservatively, all combinations need to be investigated to ensure all triangles are counted. However, there is some knowledge we can use to identify fruitless searches.

Assume there is a set of vertices $S \subset V$ where we know ahead of time that $N_v \cap S = \{\}$. Let $F = N_u \cap S$, then we know ahead of time that there exists no triangle (f, u, v), where $f \in F$ as f cannot be a neighbour of v. This follows from $N_v \cap N_u = N_v \cap (N_u \setminus F)$.

The most important S, that fits in well with the power-law graphs, is the set of hubs. As an example, in Figure 2 vertex 8 processes its neighbour 6 and loads its edges {0, 1, 4}. As 8 is not connected to a hub (0 or 1), it can be inferred that no triangle exists including vertices 8, 6 and any of the hubs.

Dataset	Hub Edges (%)			Non-hub to Non-hub Edges (%)	Hub Triangles (%)	Relative Density of Hubs Sub-graph	Fruitless Searches (%)
	Hub to Hub	Hub to Non-hub	Total				
LJGrp	4.7	76.9	81.5	18.5	99.9	467	78.1
Twtr10	43.5	29.8	73.3	26.7	99.6	4,347	64.0
Twtr	26.3	60.3	86.6	13.4	99.7	2,627	72.2
TwtrMpi	19.1	53.5	72.7	27.4	99.4	1,911	67.8
Frndstr	6.0	25.3	31.3	68.7	47.3	600	36.9
SK	4.9	75.6	80.5	19.5	97.0	490	56.4
WbCc	37.0	35.9	72.8	27.2	99.6	3,695	47.1
UKDls	14.2	63.9	78.2	21.8	98.8	1,423	39.9
UU	12.5	61.9	74.4	25.6	96.2	1,252	31.7
UKDmn	12.8	64.9	77.7	22.3	96.6	1,279	39.1
Average	18.1	54.8	72.9	27.1	93.4	1,809	53.3

Table 1. Topological characteristics of hubs (1% of vertices with maximum degrees selected as hubs)

In other words, **accessing hub edges cannot result in a triangle during processing non-hub vertices that have no edges to hubs** ($N_v \cap Hubs = \{\}$). However, hub edges are frequently accessed in processing these non-hub vertices. We measured what fraction of accessed edges point to hubs when processing these vertices (Table 1, Column 8). On average, **53.3% of memory accesses are performed to hub edges that can be avoided**. This data was collected using merge join intersection. Deploying binary search intersection [31] also reduces these memory accesses (Section 6.3).

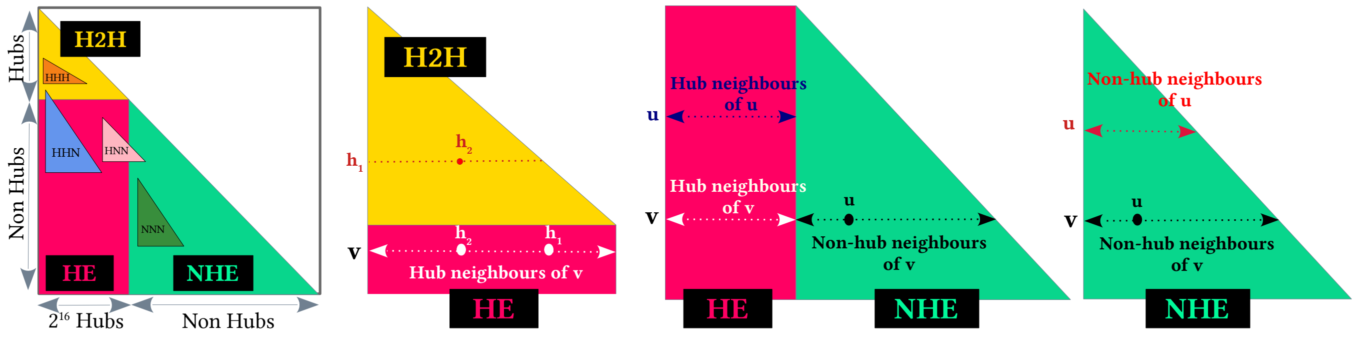
This shows that **if we *know* that v is not connected to a hub, it is possible to prune the search** and the power-law structure of graphs is helpful to prevent accessing a large fraction of edges in processing non-hub vertices.

3.4 Highly Dense Hubs Sub-graph

We have already observed that hubs are few and incident to 72.9% of the edges. Interestingly, as each vertex in a triangle must have two incident edges, these statistics become **even more skewed** when considering hubs. Column 6 of Table 1 ("Hub Triangles") shows the percentage of triangles containing at least one hub. It shows that, on average, **93.4% of the triangles contain at least one hub**.

This observation is an immediate result from the tight connections between hubs. We define the relative density (RD) of a sub-graph $S = (V', E')$ where $V' \subset V$ and $E' = E \cap (V' \times V')$, as $RD_S = \frac{|E'|/(|V'|^2)}{|E|/(|V|^2)}$. Column 7 of Table 1 reports the RD_H, where H is the set of hubs and shows that **hub-to-hub edges create a dense sub-graph that is 1809 times more dense than the full graph**.

These statistics demonstrate that (i) hubs should be central to the design of a TC algorithm; (ii) the high density of the hubs sub-graph invites a highly compact data structure to store this sub-graph.



(a) Adjacency matrix **(b)** HHH & HHN TC **(c)** HNN TC **(d)** NNN TC

Figure 3. Lotus adjacency matrix and TC steps

4 LOTUS

4.1 Lotus Idea

In order to manage the challenges surrounding hubs, Lotus distinguishes 4 types of triangles:

- **HHH**: triangles between 3 hub vertices,
- **HHN**: triangles between 2 hub and 1 non-hub vertices,
- **HNN**: triangles between 1 hub and 2 non-hub vertices,
- **NNN**: triangles between 3 non-hub vertices.

Triangle counting is organized in three steps, each using a bespoke algorithm and data structure designed to capture the corresponding characteristics of locality.

4.1.1 Counting HHH and HHN Triangles.

To count triangles with at least two hub vertices, the key question is if two hubs are connected? To check if two hubs are neighbours, Lotus exploits two features of power-law graphs: (1) hubs have a dense connection to each other (Section 3.4), and (2) there is a small number of hubs (Section 3.2).

Lotus represents the adjacency information of hubs in a dense bit array, with 1 bit per pair of hubs. As there are few hubs, the bit array can be retained in cache. Using this bit array, Lotus iterates over all distinct pairs of hub neighbours of each vertex and identifies triangles if two hubs in a pair are connected.

4.1.2 Counting HNN Triangles.

HNN triangles have two non-hub vertices that are connected to a hub. Hence, the most frequently occurring edges, and thus the ones most frequently queried, are hub edges. As such, Lotus organizes the search around iterating non-hubs and their non-hub neighbours, and then querying if they have a common hub neighbour.

To perform HNN TC efficiently, Lotus stores the hub neighbours separately from the non-hub neighbours. Moreover, based on the observation that hubs are few but frequently mentioned (Section 3.2), we store the hub neighbours using fewer bits per ID.

4.1.3 Counting NNN Triangles.

In this step, Lotus identifies common non-hub neighbours between a non-hub vertex and its non-hub neighbours. Lotus avoids loading hub edges (Section 3.3) and only processes non-hub edges in this step as it separately stores hub and non-hub edges.

4.2 Lotus Graph Structure

In order to achieve the targets explained in Section 4.1, Lotus creates a special graph structure that consists of:

- **Number of Hubs**: Lotus selects the 64K (2^{16}) vertices with the highest degrees as hubs.
- **Hub to Hub (H2H) Bit Array**: Lotus represents hub-to-hub edges using this bit array. Since each hub has edges only to hubs with lower IDs, H2H is a triangular array (instead of a 2D square array). In this way, for hub vertices $h1$ and $h2$ where $h1 > h2 \geq 0$, the bit with index $h1(h1-1)/2 + h2$ specifies if $h1$ has an edge to $h2$.
- **Hub Edges (HE) Sub-graph**: This sub-graph represents all hub edges of the graph. It is stored in CSX format. As Lotus selects 64K hubs, each edge (neighbour ID) in HE sub-graph is represented in 16 bits. For vertex v, HE represents all hub neighbours h of v where $h < v$.
- **Non-Hub Edges (NHE) Sub-graph**: This sub-graph represents edges from each vertex v to its non-hub vertices u, where $u < v$. This sub-graph is also in CSX format, but unlike HE, NHE assigns 32 bits memory space per edge.

Figure 3a shows the adjacency matrix of Lotus. The 4 types of triangles are illustrated to demonstrate in which range their endpoints sit. Note that hub-to-hub edges are recorded twice: once in the HE sub-graph and once in the H2H, which overlaps HE in the figure.

4.3 Lotus Preprocessing

Lotus creates its graph structure in a preprocessing step before counting triangles. Algorithm 2 shows how Lotus creates its graph.

4.3.1 Creating Relabeling Array.

Lotus assigns the first consecutive IDs to hub vertices, therefore it is necessary to relabel vertices. Line 1 creates the relabeling array. The **create_relabeling_array**() function selects the hub vertices with highest degrees and assigns the first IDs to them.

Algorithm 2: Lotus Preprocessing

```
   Input: $G(V, E)$
   Output: LotusGraph
1  $RA$ = create_relabeling_array($G$);
2  $hubs\_count = (1 \ll 16)$;
3  TBitArray $H2H(hubs\_count)$;
4  Graph < $ushort$ > $HE$;
5  Graph < $uint$ > $NHE$;
6  par_for $v_{old} \in V$
7      $v_{new} = RA[v_{old}]$;
8      Array < $ushort$ > $he$;
9      Array < $uint$ > $nhe$;
10     for $u_{old} \in N_{v_{old}}$ do
11         if $u_{old} == v_{old}$ then
               /* self-edge */
12             continue;
13         $u_{new} = RA[u_{old}]$;
14         if $u_{new} > v_{new}$ then
               /* symmetric edge */
15             continue;
16         if $u_{new} < hubs\_count$ then
               /* hub neighbour */
17             $he$.push($u_{new}$);
18             if $v_{new} < hubs\_count$ then
                   /* hub neighbour of a hub */
19                 $H2H$.set($v_{new}, u_{new}$);
20         else
               /* non-hub neighbour */
21             $nhe$.push($u_{new}$);
22     $HE$.setEdges($v_{new}, he$);
23     $NHE$.setEdges($v_{new}, nhe$);
24 return ($hubs\_count, H2H, HE, NHE$);
```

In addition to hub vertices, there are a number of high-degree vertices. If they are assigned large IDs, the number of comparisons when processing NNN triangles is increased (Section 3). So, Lotus assigns the first consecutive IDs to 10% of vertices with the highest degrees instead of only 64K ones.

The remaining IDs are assigned to non-hub vertices in the same order as the main graph. In this way, Lotus prevents destroying the initial locality of graphs, which is a known artefact from degree ordering [44, 68, 72].

create_relabeling_array() returns an array that is indexed by the original ID of a vertex and the value at that index specifies the new ID of that vertex.

4.3.2 Creating Bit Array and Sub-graphs.

Line 3 initializes the H2H triangular bit array storing hub-to-hub edges by allocating memory of $hubs\_count * (hubs\_count - 1)/2$ bits size and setting all bits to zero.

Lines 4 and 5 initialize sub-graphs for **HE** and **NHE** where the size of each edge is 16 and 32 bits, respectively.

Lines 6-23 process each vertex in the graph. Lines 8-9 initialize the **he** and **nhe** arrays to contain hub and non-hub neighbours of a vertex, respectively.

Each neighbour of a vertex is considered in Lines 11-21 and self-edges and symmetric edges are ignored (Lines 11-15). Similar to the baseline algorithm (Section 2.2), Lotus does not process symmetric edges and limits neighbours of a vertex to the ones that have lower IDs. This restricts the neighbour list of vertex v_{new} to $N^{<}_{v_{new}}$.

The neighbour is assigned to **he** (Line 17), if it is a hub neighbour. In this case, the H2H bit array is set if the vertex and its neighbour are both hubs (Line 19). If the neighbour is a non-hub vertex, it is added to **nhe** (Line 21).

After processing all edges of a vertex, Lines 22-23 call *setEdges*() method that sorts the neighbour lists **he** and **nhe** and assigns them to the relevant vertex (v_{new}) of **HE** and **NHE** sub-graphs, respectively.

In Lines 5 and 9, 32-bit vertex ID is sufficient for public data sets as they have fewer than 2^{32} vertices. However, for datasets with greater number of vertices, 64-bit IDs can be used without losing the benefits of Lotus.

4.4 Counting Triangles in Lotus

Algorithm 3 shows how Lotus counts triangles:

4.4.1 HHH and HHN.

Lotus creates all distinct pairs between hub neighbours of a vertex (Lines 3-4) and if two hubs of a pair are connected (Line 5), a triangle has been found. Note that the bit array is laid in "$h1$-major" format, ensuring that bits for subsequent $h2$ values are placed in consecutive locations. Moreover, as $h1$ changes in the outer loop on Line 3, the calculation $h1(h1-1)/2$ is reused as $h2$ changes in the inner loop in Line 4.

Figure 3b shows counting HHH and HHN triangles for vertex v with hub neighbours h_2 and h_1. The existence of triangle (h_2, h_1, v) is validated by checking if h_2 has an edge to h_1 in the H2H sub-graph.

4.4.2 HNN.

Lotus finds common hub neighbours between each non-hub vertex and its non-hub neighbours. Line 7 iterates over all vertices. For each non-hub vertex v, its non-hub neighbours such as u are considered (Line 8), and each common hub neighbour of u and v forms a triangle (Line 9).

In Figure 3c, for vertex v and its non-hub neighbours such as u (that are in NHE sub-graph), hub neighbours of u and v (that are in HE sub-graph) are matched.

4.4.3 NNN.

Lines 10–12 are similar to the Forward algorithm to find NNN triangles in the NHE. Lotus uses merge join for intersection as the neighbour lists of non-hub vertices are relatively short. This prevents overheads imposed by other solutions (Section 6.3).

Algorithm 3: Counting Triangles in Lotus

```
Input: LotusGraph $L(hubs\_count,\ H2H,\ HE,\ NHE)$
Output: Triangles
1  $triangles = 0$;
   /* Counting HHH and HHN triangles */
2  par_for $v \in HE.V$
3      par_for $h1 \in HE.N_v$
4          for $h2 \in \{h \in HE.N_v \mid h < h1\}$ do
5              if $H2H.isSet(h1, h2)$ then
6                  $triangles++$;
   /* Counting HNN triangles */
7  par_for $v \in NHE.V$
8      par_for $u \in NHE.N_v$
9          $triangles += |HE.N_v \cap HE.N_u|$;
   /* Counting NNN triangles */
10 par_for $v \in NHE.V$
11     par_for $u \in NHE.N_v$
12         $triangles += |NHE.N_v \cap NHE.N_u|$;
13 return $triangles$;
```

In Figure 3d, for vertex v and for its non-hub neighbours such as u (that are in NHE), non-hub neighbours of u and v (that are in NHE) are matched.

4.5 How Does Lotus Improve Locality?

In counting HHH and HHN triangles, Lotus reads hub neighbours of a vertex in sequential accesses and iterates over all pairs of hub neighbours (Lines 3-4 of Algorithm 3). In other words, **Lotus accesses the neighbour list of a vertex only for processing that vertex**. The neighbour lists are streamed through cache. Sequentially streamed accesses are prefetched by hardware in timely fashion. Only the H2H bit array is used (Line 5) for random accesses to topology data. By concentrating random accesses on the H2H bit array, **the range of data accessed randomly is significantly reduced**.

This increases the frequency of cache hits. Table 7 shows that graph datasets in this study have edges with topology size of 0.42 - 12.30 Gigabytes in the CSX format, but the H2H size is less than 256 Megabytes. Moreover, **H2H stores edges in an addressable format** that facilitates efficient checking if two hubs are connected in constant time, and just a few instructions. Section 5.7 shows that 64 Megabytes cache space suffices to satisfy 90% of accesses to H2H.

In Algorithm 3, Lotus has two similar nested loops for counting HNN and NNN triangles in Lines 7-8 and 10-11. These loops iterate over the same domain (the neighbour lists of NHE). Lotus keeps the body of these loops (intersections at Lines 9 and 12) separate (as opposed to fusing the loops). Two contradictory effects need to be traded-off:

- Random memory accesses are made to $HE.N_u$ (Line 9) and $NHE.N_u$ (Line 12). Reuse of this data before eviction from the cache is possible. If we were to fuse the loops in Lines 7–12, then reuse of this data would become less likely, as the total volume of randomly accessed data, and thus the working set size, will increase.
- The cost of traversing the NHE sub-graph itself (fetching $NHE.V$ and $NHE.N_v$) is low as this data is streamed in sequentially. The NHE topology is relatively small as it contains only 27% of edges on average (Table 1, Column 7).

TC Step	Random Accesses	Edge Size	Total Size of Edges
HHH & HHN	$H2H$	1 bit	256 Megabytes
HNN	$HE.E$	16 bits	$\|HE.E\| * 2$ Bytes
NNN	$NHE.E$	32 bits	$\|NHE.E\| * 4$ Bytes

Table 2. Random memory accesses in Lotus TC

Lotus improves locality by dividing TC into three steps and in each step dedicates cache to a smaller special data structure that is most frequently needed. Table 2 summarizes which data structure is accessed in random order. Section 5.3 shows that Lotus reduces last level cache misses by 2.1× and DTLB misses by 34.6×, on average.

4.6 Graph Partitioning and Load Balancing in Lotus

Edge Tiling [56] improves load balance by splitting the edge list of high-degree vertices into smaller parts and scheduling these on different concurrent threads.

In Line 3 of Algorithm 3, the amount of work each neighbour ($h1$) performs depends on its offset from the first neighbour. As a consequence, we cannot divide work between threads by assigning the same number of neighbours to each thread.

In order to parallelize the loop in Line 3 of Algorithm 3, Lotus introduces **Squared Edge Tiling** that creates partitions with equal work complexity for neighbours of a vertex.

For vertex v with $|N_v|$ neighbours, the total work is $|N_v| * (|N_v| - 1)/2$ and if the total work performed from the first neighbour until the i-th neighbour is f fraction of the total work, where $0 < f < 1$, then:

$$i * (i-1)/2 = f|N_v|(|N_v| - 1)/2 \ ,$$

or

$$i = (\ 1 + \sqrt{f(2|N_v| - 1)^2 + 1 - f}\ )/2 \ .$$

Since $|N_v| \gg f$,

$$\frac{i}{|N_v|} \approx \frac{1 + \sqrt{f(2|N_v| - 1)^2}}{2|N_v|} \approx \sqrt{f} \ ,$$

or

$$i \approx |N_v| * \sqrt{f} \ .$$

Using this formula, we can identify the boundaries to partition the loop by changing f. As an example, for partitioning total work for a vertex with 100 neighbours into 5 partitions, the partition borders will be 0, $100 * \sqrt{0.2} = 45$, $100 * \sqrt{0.4} = 63$, $100 * \sqrt{0.6} = 77$,$100 * \sqrt{0.8} = 89$, and 100.

While the number of triangles may vary per tile, the effort per tile is balanced. Lotus performs squared edge tiling

	SkyLakeX	Haswell	Epyc
CPU Model	Intel Xeon Gold 6130	Intel Xeon E5-4627	AMD Epyc 7702
CPU Frequency	2.10 GHz	2.6 GHz	2 GHz
Sockets	2	4	2
NUMA Nodes	2	4	8
Total CPU Cores	32	40	128
Hyperthreading	No	No	No
L1 Cache	32 KB / 1 core	32 KB / 1 core	32 KB / 1 core
L2 Cache	1 MB / 1 core	256 KB / 1 core	512 KB / 1 core
L3 Cache	22 MB / 16 cores	25.6 MB / 10 cores	16 MB / 4 cores
Total L3 Cache	44 MB	102.4 MB	512 MB
Total Memory	768 GB	1,024 GB	2,048 GB

Table 3. Machines

Dataset	Name	Type	Source	|V| (M)	|E| (B)	|Triangles|
LJGrp	LiveJournal	SN	KN	7	0.22	141,388,608
Twtr10	Twitter 2010	SN	NR	21	0.53	17,295,646,010
Twtr	Twitter	SN	NR	28	0.96	13,734,746,881
TwtrMpi	Twitter-MPI	SN	NR	41	2.41	34,824,916,864
Frndstr	Friendster	SN	NR	65	3.61	4,173,724,142
SK	SK-Domain	WG	LWA	50	3.64	84,907,040,872
WbCc	Web-CC12	WG	NR	89	3.87	417,026,090,229
UKDls	UK-Delis	WG	LWA	110	6.92	663,713,224,204
UU	UK-Union	WG	LWA	133	9.36	453,830,915,490
UKDmn	UK-Domain	WG	KN	105	6.60	286,701,284,103
MClst	MetaClust	BG	HM	282	42.8	5,588,867,541,009
ClWb12	ClueWeb12	WG	LWA	978	74.7	1,995,295,290,765
WDC14	WDC 2014	WG	WDC	1,724	124	4,587,563,913,535
EU15	EU Domains	WG	LWA	1,071	161	15,338,196,409,949

Table 4. Datasets

Dataset	SkyLakeX					Haswell					Epyc				
	BBTC	GGrnd	GAP	GBBS	Lotus	BBTC	GGrnd	GAP	GBBS	Lotus	BBTC	GGrnd	GAP	GBBS	Lotus
LJGrp	4.1	4.7	6.4	2.5	**1.0**	8.6	3.9	6.8	1.4	**1.1**	2.4	2.5	6.6	**0.5**	0.8
Twtr10	62.4	74.2	32.7	32.8	**6.7**	15.9	50.0	28.9	25.3	**6.7**	31.5	21.6	45.0	9.0	**4.1**
Twtr	98.0	77.0	32.1	32.1	**10.0**	122.9	56.3	28.5	25.1	**9.8**	81.3	25.8	20.3	9.4	**6.1**
TwtrMpi	377.7	282.2	80.5	90.5	**36.8**	234.5	129.2	67.6	72.4	**33.7**	333.3	67.2	38.8	25.9	**18.2**
Frndstr	129.5	129.1	70.5	76.4	**56.7**	176.8	111.7	69.5	67.5	**54.6**	59.9	33.3	27.4	24.5	**23.8**
SK	246.3	56.5	28.8	19.5	**7.3**	871.5	37.5	29.0	8.2	**6.5**	246.5	19.5	21.0	3.3	**2.9**
WbCc	602.0	649.0	121.1	233.8	**64.2**	362.2	279.0	118.9	170.1	**57.9**	534.5	134.1	92.1	51.7	**21.9**
UKDls	-	383.3	67.7	80.0	**32.7**	-	141.5	68.3	48.7	**26.1**	-	58.6	89.8	38.6	**12.2**
UU	-	134.9	61.6	74.4	**29.3**	-	86.9	56.8	38.6	**22.1**	-	43.8	36.0	15.0	**9.5**
UKDmn	-	123.9	50.3	53.6	**19.9**	-	58.0	48.5	24.9	**15.9**	-	32.6	32.4	10.3	**7.2**
Lotus Avg. Speedup	11.3×	7.4×	3.0×	2.8×		24.6×	4.6×	3.1×	2.0×		22.1×	4.5×	5.3×	1.7×	

Table 5. End to end TC execution times in seconds - GGrnd: GraphGrind - Failed attempts are shown by dash - Avg. Speedup is arithmetic mean over Lotus speedup for each dataset

during the preprocessing step. Values of $\sqrt{f}$ are fixed for different vertices as f indicates the fraction of work and for dividing work into p partitions, $f = \frac{k}{p}$, where $0 < k < p$. So, values of $\sqrt{f}$ are pre-calculated and reused in calculating the partition boundaries of different high-degree vertices.

Section 5.8 shows that squared edge tiling provides 2.7× speedup in processing HHH and HHN triangles.

5 Evaluation

5.1 Experimental Setup

5.1.1 Machines. We present experiments on 3 machines with different processor architectures, listed in Table 3. The machines use CentOS 7.

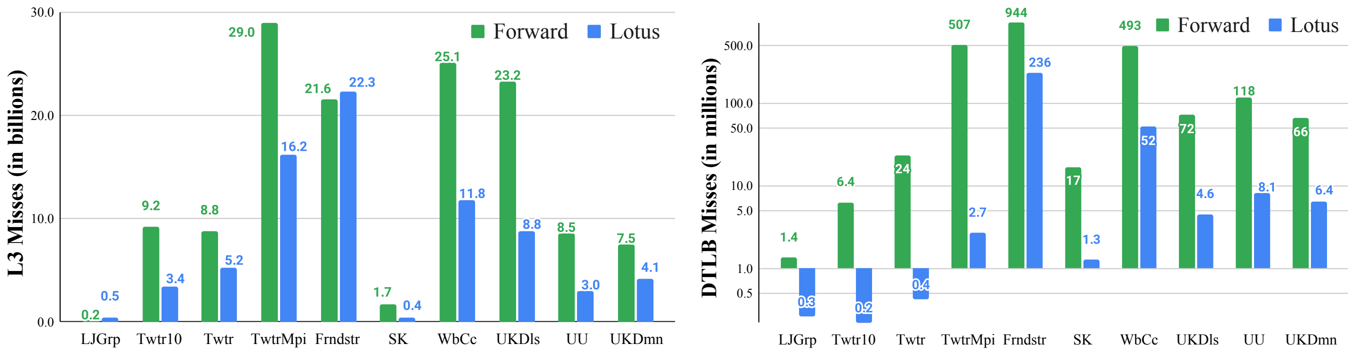
5.1.2 Datasets. Table 4 shows the datasets and their sources: "*Konect*" (KN) [15, 46, 55], "*NetworkRepository*" (NR) [18, 21, 22, 58, 64], "*Laboratory for Web Algorithmics*" (LWA) [15–18, 47], "*HipMCL*" (HM) [9, 65], and "*Web Data Commons*" (WDC) [49, 52, 53]. Datasets types are Social Network (SN), Web Graph (WG), or Bio Graph (BG). Numbers of edges are in billions and numbers of vertices are in millions, counted after removing zero degree vertices. Graphs are represented in Compressed Sparse Row/Column [59] with $|V| + 1$ index values of 8 bytes per index value and $|E|$ neighbour IDs of 4 bytes each. Lotus uses 2 bytes for neighbour IDs of HE.

Dataset	Epyc	
	GBBS	Lotus
MClst	1,415.2	**784.5**
ClWb12	81.7	**29.9**
WDC14	170.1	**85.7**
EU15	449.3	**256.9**
Lotus Avg. Speedup	2.1×	

Table 6. End to end TC execution times in seconds

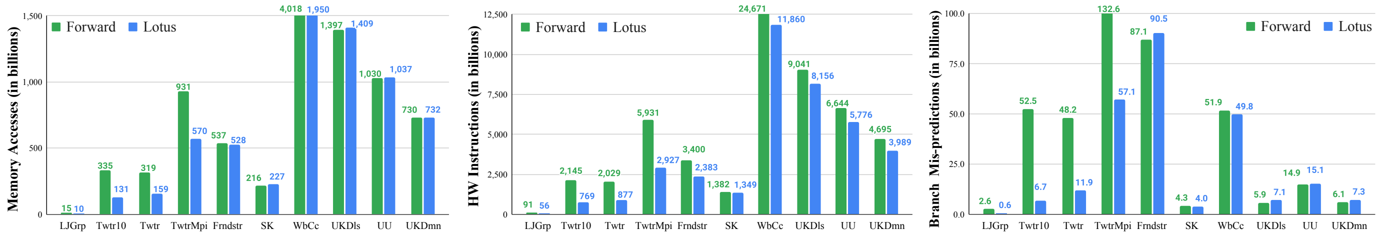
5.1.3 Lotus. We implemented Lotus in the `C` language using the `pthread`, `libnuma`, and `papi` [70] libraries. We use the interleaved NUMA memory policy and to have a better load balance[61], we apply work-stealing for parallel processing of graph partitions as described in Section 4.6. We use the master-worker model for managing parallel threads and `futex` syscall for thread synchronization. We compiled the source code using the `gcc-9.2` compiler with `-O3` flag.

5.1.4 Frameworks and TC algorithms. We use the following algorithms/implementations for evaluating Lotus:



(a) Last level cache misses **(b)** DTLB misses (Log scale vertical axis)

Figure 4. Comparison of last level cache misses and DTLB misses [SkyLakeX]



(a) Memory accesses (load & store instructions) **(b)** Hardware instructions **(c)** Branch mis-predictions

Figure 5. Comparison of hardware events [SkyLakeX]

1. BBTC [76] (commit 88fe6bc) that improves load balancing in TC through better partitioning.
2. Edge iterator in GraphGrind [66, 67] (commit 5099761).
3. Forward algorithm implementation in GAP [10] (commit 6ac1afd), as a study of graph frameworks [7] shows TC performance of other graph processing frameworks that do not use vectorization, are close to GAP (±10%) .
4. TC of GBBS [26] (commit 38964eb) that improves [63] by parallelizing the intersection in the Forward algorithm.

All algorithms use degree ordering to accelerate TC and we report end-to-end execution time.

5.2 Comparison to Previous Works

Tables 5 compares Lotus execution time with other TC algorithms for graphs smaller than 10 billion edges.

This table shows that the speedup obtained by Lotus on the Epyc architecture with 128 cores is less than on the other architectures. This is due to the total on-chip cache size. The Epyc system has two sockets with 512MB total L3 cache, which is 12 times larger than the L3 cache on the SkyLakeX machine. This large L3 cache captures a significantly higher fraction of memory accesses, and poses lesser challenges relating to memory locality. As a result, speedup obtained by Lotus is less, due to the larger cache size.

Table 6 shows the results of Lotus in comparison to GBBS on the Epyc machine and for graphs greater than 10 billions edges. This shows that Lotus delivers better speedups for larger graphs.

On average, **Lotus is 19.3 times faster than BBTC, 5.5 times faster than GraphGrind, 3.8 times faster than GAP, and 2.2 times faster than GBBS**.

5.3 Has Lotus Improved Locality?

In Section 4.5, we explained how Lotus improves locality. To evaluate the locality effects of Lotus, we compare the last level cache misses and DTLB misses of Lotus and Forward algorithms on the SkyLakeX machine in Figure 4a, and Figure 4b. **Lotus reduces last level cache misses by up to 4.0× and on average by 2.1×. DTLB misses are also reduced by up to 56× and on average by 34.6×.**

Besides improving locality, Lotus is also a more efficient algorithm throughout. Figure 5 compares hardware events for execution of Lotus and Forward algorithms. It shows that, on average, **Lotus reduces memory accesses (load and store instructions) by 1.5×, hardware instructions by 1.7×, and branch mis-predictions by 2.4×**.

5.4 Execution Breakdown

Figure 6 displays the breakdown of Lotus execution time and shows time passed in (1) preprocessing, (2) counting HHH and HHN triangles, (3) counting HNN triangles, and (4) counting non-hub triangles.

It shows that, on average, **19.4% of the total execution time is passed in preprocessing**. Moreover, on average, **40.4% of the triangle counting time is passed in counting non-hub triangles**.

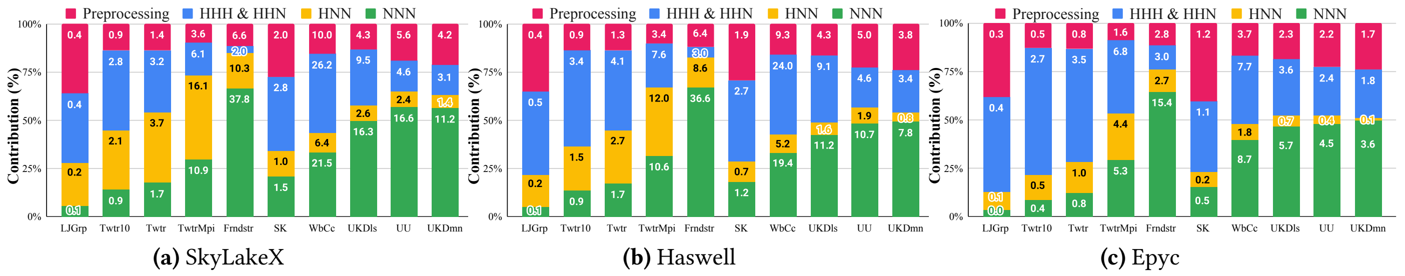


Figure 6. Lotus execution breakdown (numbers on each bar are in seconds)

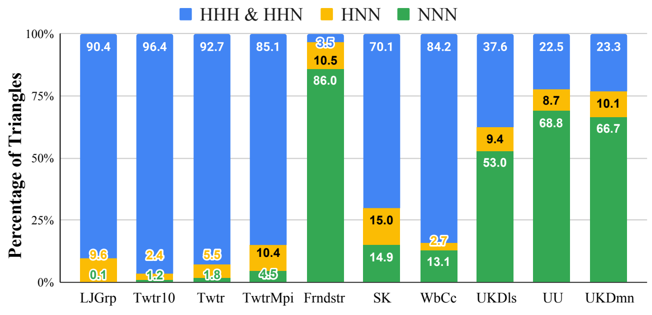


Figure 7. Contribution of triangles

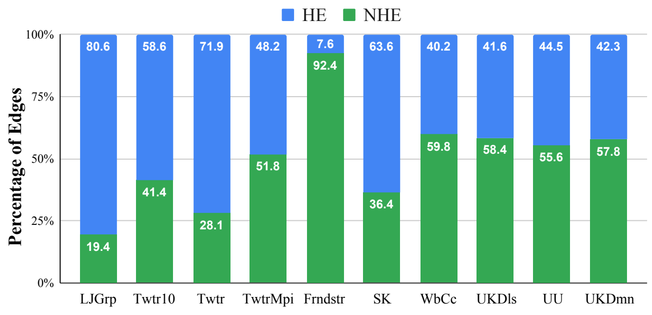


Figure 8. Percentage of edges in HE and NHE sub-graphs

Figure 7 compares the number of hub and non-hub triangles counted by Lotus. It shows that, on average, **68.9% of the triangles are counted as hub triangles in Lotus** and 31.1% as non-hub triangles.

Figure 8 compares the number of edges in HE and NHE sub-graphs. It shows that, on average, **Lotus processes 50.1% of edges as hub edges**. The number of triangles and edges are different from Table 1 as 1% of vertices have been selected there as hubs.

5.5 Less Power-Law Graphs

Figures 8 and 7 show that less power-law graphs may not benefit from Lotus as other datasets. For example, the `Friendster` dataset has a relatively low skewness and the highest degree is 5K. However, Lotus selects a constant number of hubs (64K). By consequence, only 7.6% of the edges connect to these hubs and Lotus spends most of the TC time in counting non-hub triangles (Figure 6).

In general, less power-law graphs can be categorized in two categories:

1. Social networks with **a great number of low-degree hubs** where the tight connection between high-degree vertices [44] allows improving performance by **recursively applying Lotus** and splitting the NHE sub-graph further in new H2H, HE and NHE components, similar to how iHTL extracts dense flipped blocks [42].
2. Graphs that have **a very small number of very high-degree hubs**, where the Forward algorithm is effective even without degree ordering. In processing low-degree vertices of these graphs, two types of memory accesses are performed:
   (i) Accesses to neighbour list of hub vertices that are easily maintained in the cache as hubs are rare but are accessed frequently (since they are neighbours to a great percentage of vertices), and
   (ii) Accesses to neighbour list of low-degree neighbours that a good spatial locality (which usually exists in graphs before degree reordering, especially in LWA graphs as a result of applying Layered Label Propagation [17]) results mostly in cache hits (since spatial locality assigns consecutive IDs to neighbours and necessitates consecutive processing of low-degree neighbours).
   For these graphs, it is necessary to check the degree distribution of the graph at the start of TC and to apply the Forward or edge-iterator algorithms if the graph is not skewed enough. GAP [10] uses the average degree of the graph and a sampling mechanism to compare the average and median degree of vertices.

5.6 Topology Data Size

Table 7 compares size of topology data in CSX format and Lotus. Since the Forward algorithm (Algorithm 1) uses only half of the edges, we have calculated sizes of CSX edges and CSX without symmetric edges.

Lotus affects the size of topology data in 3 ways:

- The HE and NHE sub-graphs require an index array each, adding $8(|V|+1)$ Bytes.
- Adding the H2H bit array, of fixed size (256 Megabytes).
- Reducing the size of hub IDs, which saves 2 bytes per edge in the HE sub-graph.

Dataset	CSX Edges (GB)	CSX (GB)	Lotus (GB)	Growth (%)
LJGrp	0.4	0.5	0.6	28.8
Twtr10	1.0	1.1	1.3	10.4
Twtr	1.8	2.0	1.8	-8.9
TwtrMpi	4.5	4.8	4.3	-10.8
Frndstr	6.7	7.2	7.7	6.7
SK	6.8	7.2	5.6	-21.6
WbCc	7.2	7.9	7.3	-6.8
UKDls	12.9	13.7	12.1	-11.9
UU	17.4	18.4	15.7	-14.5
UKDmn	12.3	13.1	11.5	-12.0

Table 7. Size of topology data (Gigabytes)

Dataset	H2H Density (%)	H2H Zero Cachelines (%)
LJGrp	0.20	62.51
Twtr10	2.83	5.72
Twtr	2.05	8.60
TwtrMpi	2.73	9.89
Frndstr	0.29	36.94
SK	1.04	91.74
WbCc	15.26	74.60
UKDls	2.56	93.31
UU	0.17	91.45
UKDmn	0.15	95.15

Table 8. Lotus H2H bit array characteristics

For graph datasets like `SK-Domain` where Lotus collects a greater number of edges as hub edges, the topology size is reduced more as HE size is reduced.

Table 7 shows that, on average, **Lotus reduces size of topology data by 4.1%**. Independently of reducing size, only a subset of the topology data is accessed in each phase, resulting in smaller working sets.

5.7 H2H Bit Array

H2H is a dense triangular adjacency array that lists edges between a hub and its hub neighbours with lower IDs. The first column of Table 8 shows that the density of H2H (fraction of non-zero bits) is between 0.2% and 15.3%.

We also measured how many 64-byte aligned blocks of H2H contain 512 zero bits (Table 8, column 3). In web graphs, 75–95% of H2H blocks contain no edges. Edges are thus tightly packed in cache blocks, which implies that **hubs in web graphs are mostly connected to a number of hubs**. In contrast, social networks exhibit a different behavior where 5–62% of the blocks are zero that shows edges are thus more dispersed throughout H2H.

To have a better understanding of how H2H is placed in cache, we measure how many accesses to H2H are satisfied by selecting the most frequently accessed cachelines. To this end, we sort cachelines based on how frequently they are accessed and we calculate the partial sum of their accesses.

Figure 9 shows that by storing one million cachelines of H2H in cache, more than 90% of accesses to H2H are satisfied. In other words, **64 Megabytes of cache space suffices to satisfy 90% of accesses to H2H**.

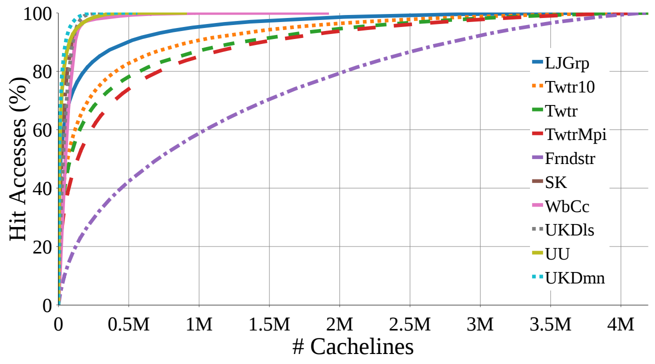


Figure 9. Percentage of accumulative memory accesses to most frequently accessed cachelines of H2H (M: Million)

Dataset	Edge Balanced (%)	Squared Edge Tiling (%)
Twtr10	32.1	1.0
TwtrMpi	32.6	0.7
SK	13.6	3.1
WbCc	83.3	1.3
UKDls	41.8	3.3

Table 9. Average idle time in percent of total execution time [SkyLakeX]

This shows that 90% of (h_1, h_2) pairs produced in Line 5 of Algorithm 3 access only 25% of H2H cachelines. In other word, accesses to the **H2H sub-graph benefit from a high level of locality**.

While using a hash table can be seen as an option for implementing H2H, Figure 9 shows that the high level of locality in memory accesses to H2H makes it suboptimal to use a hash table for H2H. A hashing mechanism imposes more instruction count per memory access, a higher memory footprint, and a higher preprocessing time.

5.8 Squared Edge Tiling

In Section 4.6, we introduced the squared edge tiling partitioning policy to provide better load balance in processing HHH and HHN triangles in Lotus algorithm. Lotus applies squared edge tiling for vertices with degree greater than 512 and divides the total work of each vertex between $p = 2 * \#threads$ partitions.

Table 9 shows the average idle time of threads in the first step of Lotus for two partitioning policies: edge balanced partitioning [67, 79] and squared edge tiling, when running on the SkyLakeX machine. Edge balanced divides edges into $256 * \#threads$ partitions. On average, **squared edge tiling provides 2.7× speedup in processing HHH and HHN triangles**.

6 Related Work

6.1 TC History

Itai and Rodeh [37, 38] use rooted spanning tree for TC. *AYZ* algorithm [1, 2] provides better computation complexity ($O(|E|^{1.41})$) in counting triangles of sparse graphs. It uses matrix multiplication for triangles formed by high-degree vertices and for triangles made by at least one low-degree vertex, AYZ algorithm acts like the node iterator algorithm (Section 2.2) and finds the directed paths of length 2 and checks if their endpoints are connected by an edge.

In addition to the 3 algorithms explained in Section 2.2, Schank and Wagner [62] present 3 improvements:

- *Node-iterator-core* algorithm prioritizes vertices with smaller degree and removes the vertex after processing,
- *Edge-iterator-hashed* algorithm uses a hash container to identify the common neighbours of the endpoints of each node, and
- *Forward-hashed* algorithm uses a hash container for finding common neighbours.

Latapy [48] presents the *new-vertex-listing* algorithm to improve the node iterator algorithm for high-degree vertices. For each vertex, it iterates over all its neighbours and finds the common neighbours using a bitmap. Based on this, Latapy presents the *new-listing* as an improvement to the *AYZ* algorithm.

Lotus makes several benefits from these algorithms:

- Similar to AYZ and new-listing, Lotus differentiates between hub and non-hub vertices, however, Lotus counts a triangle as hub triangle if it has at least one hub vertex, as the main target of Lotus is to prevent accessing hub edges when it is not required.
- Lotus uses a bitmap array like the new-vertex-listing algorithm does. However, Lotus does not use it for presenting edges of only a vertex, but for all edges between hubs.
- Lotus has also similarities with node-iterator-core algorithm as Lotus (1) counts triangles of hubs, (2) removes hubs and their edges from the graph (as they are not present in the NHE sub-graph), and (3) counts triangles between non-hub vertices in the NHE sub-graph.

6.2 Approximate and Streaming TC

Approximate and streaming TC has also been studied in the literature such as [11, 19, 39, 63, 71].

The Lotus algorithm can be used to accelerate counting hub triangles of a streaming graph and also to improve its precision. We know hubs create a large percentage of total triangles (Sections 3.4 and 5.4) and therefore in a streaming context, Lotus stores the H2H bit array in the memory and accelerates processing of hub edges that are streamed in.

6.3 Improvements to TC and Forward Algorithm

Using hash maps for accelerating neighbour matching has been studied in some works such as [48, 63]. In this context, using binary search has been proposed in [31] and [33] deploys branch-free binary search [40, 41]. [34] decides between merge-based search and binary search by considering degree of vertices.

[27] improves TC by removing vertices with degree 1 (that cannot shape a triangle) from the graph and by ordering vertices of the same degree based on their connection to hub vertices. [32] reduces branch misses by using radix binning. Fast (but with more memory complexity) common neighbour counting through iterating over all wedges is studied in [3]. TC has been one of the problems pursued by the Graph Challenge and [60] surveys a number of TC studies.

6.4 Distributed and GPU-based TC

Distributed TC has been considered in studies such as [5, 6, 77], and GPU-based TC in [14, 25, 31, 33, 34, 76]. Patric [5] presents different types of partitioning for distributed TC and also a dynamic load balancing mechanism [6]. [76] studies block-based partitioning in TC. An evaluation of set intersection techniques has been studied in [13].

6.5 Locality Optimizing and Structure-Aware Algorithms

SDS Sort [28] introduces a parallel sorting algorithm for data with skewed distribution.

Graph relabeling algorithms such as Rabbit-Order [4], GOrder [74], SlashBurn [50], and CN-Order [51] optimize locality in SpMV-based graph processing. [44] analyzes the effects of reordering algorithms on different real-world graphs by investigating the connection between different vertex classes of the graphs. It is also explained how the structure of a power-law graph provides better *Push Locality* (in traversing a graph in the push direction), or *Pull Locality* (for traversing a graph in the pull direction).

While graph reordering algorithms provide better locality for non-hub vertices, they cannot improve locality of hub vertices in a pull traversal as much as other vertices [43]. Hubs have a great number of neighbours and consecutive processing of these neighbours reduces the opportunity for reusing loaded vertex data in cache.

iHTL [42] provides temporal locality in processing hub vertices and increases the *Effective Cache Size*[44] in SpMV-based graph processing algorithms by extracting dense subgraphs containing incoming edges to in-hubs and processing them in the push direction; while processing other edges in the pull direction. In this way, memory accesses for processing in-hubs, that have a few common destinations, are hit in cache and the destructive effect of processing in-hubs in the pull direction is prohibited.

Thrifty Label Propagation [45] optimizes memory accesses in identifying Connected Components (CC) of power-law graphs. Thrifty introduces *Zero Planting* and *Zero Convergence* techniques to accelerate label propagation and to prevent processing all edges of the graph in pull iterations. In this way, Thrifty processes only a small percentage of edges and delivers better performance than sampling CC algorithms like Afforest [69].

To provide better load balance in using CPU and GPU integrated devices, FinePar [78] assigns high-degree vertices to CPU while processing low-degree vertices by GPU. VEBO [68] introduces a partitioning algorithm that distributes high-degree vertices on different partitions, while trying to assign equal number of edges to partitions.

7 Conclusion and Future Work

This paper studies behaviours of real-world graphs in triangle counting and explains that the large fraction of edges connected to hubs suffer from low reuse.

We introduced the *LOTUS* algorithm based on common features of power-law graphs. Lotus processes hub edges separately from non-hub edges, which allows Lotus to count triangles in 3 steps. In each step, Lotus optimizes locality by concentrating random memory accesses on a data structure that contains more specific data in a much smaller size.

The evaluation shows that Lotus is 2.2–5.5× faster than previous works.

We propose the following extensions as future work:

- TC is the simplest form of the k-clique counting problem. We anticipate that the skewed statistics on triangles containing hubs will become even more skewed for larger cliques. It would be interesting to study how Lotus can be applied for counting larger cliques.
- Lotus improves locality in counting HNN triangles by reducing the size of topology data and avoiding interleaving hub and non-hub edges; however, locality of HNN may be further improved by applying blocking strategies [36] to limit domain of random accesses.
- Creating multiple HE sub-graphs may improve performance further, especially in graphs with many high-degree vertices (Section 5.5). It is an open question whether recognizing a higher number of distinct vertex types (two kinds of hubs and non-hubs) creates further opportunities to prune fruitless searches during HNN and NNN search.

Source Code Availability

Source code repository and further discussions are available online in https://blogs.qub.ac.uk/GraphProcessing/LOTUS-Locality-Optimizing-Triangle-Counting/ .

Acknowledgments

We are grateful to Prof. Paolo Boldi for giving guidance in using WebGraph framework.

We are grateful to Dr. Vaughan Purnell (QUB), Jose Sanchez Bornot (Ulster University), Luis Fernandez Menchero (QUB), and James McGroarty (QUB) for supervising the Kelvin High Performance Computing Centre.

This work is partially supported by the High Performance Computing center of Queen's University Belfast and the Kelvin supercomputer (EPSRC grant EP/T022175/1) and by DiPET (CHIST-ERA project CHIST-ERA-18-SDCDN-002, EPSRC grant EP/T022345/1).

First author is supported by a scholarship of the Queen's University Belfast and the Department for the Economy, Northern Ireland.

We are grateful for the constructive feedback and suggestions of our PPoPP'22 reviewers.

References

[1] Noga Alon, Raphael Yuster, and Uri Zwick. 1994. Finding and Counting Given Length Cycles (Extended Abstract). In *Proceedings of the Second Annual European Symposium on Algorithms (ESA '94)*. Springer-Verlag, Berlin, Heidelberg, 354–364.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and counting given length cycles. *Algorithmica* 17 (1997), 209–223. https://doi.org/10.1007/BF02523189

[3] Xiaojing An, Kasimir Gabert, James Fox, Oded Green, and David A. Bader. 2019. Skip the Intersection: Quickly Counting Common Neighbors on Shared-Memory Systems. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, USA, 1–7. https://doi.org/10.1109/HPEC.2019.8916307

[4] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, USA, 22–31. https://doi.org/10.1109/IPDPS.2016.110

[5] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. 2013. PATRIC: A Parallel Algorithm for Counting Triangles in Massive Networks. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management* (San Francisco, California, USA) *(CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 529–538. https://doi.org/10.1145/2505515.2505545

[6] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. 2015. A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, USA, 1839–1847. https://doi.org/10.1109/BigData.2015.7363957

[7] Ariful Azad, Mohsen Mahmoudi Aznaveh, Scott Beamer, Mark Blanco, Jinhao Chen, Luke D'Alessandro, Roshan Dathathri, Tim Davis, Kevin Deweese, Jesun Firoz, Henry A Gabb, Gurbinder Gill, Balint Hegyi, Scott Kolodziej, Tze Meng Low, Andrew Lumsdaine, Tugsbayasgalan Manlaibaatar, Timothy G Mattson, Scott McMillan, Ramesh Peri, Keshav Pingali, Upasana Sridhar, Gabor Szarnyas, Yunming Zhang, and Yongzhe Zhang. 2020. Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, USA, 216–227. https://doi.org/10.1109/IISWC50251.2020.00029

[8] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, USA, 804–811. https://doi.org/10.1109/IPDPSW.2015.75

[9] Ariful Azad, Georgios A. Pavlopoulos, Christos A. Ouzounis, Nikos C. Kyrpides, and Aydin Buluc. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research* 46, 6 (1 2018), 11. https://doi.org/10.1093/nar/gkx1313

[10] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015), 1–16. arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[11] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2008. Efficient Semi-Streaming Algorithms for Local Triangle Counting in Massive Graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Las Vegas,

Nevada, USA) *(KDD '08)*. Association for Computing Machinery, New York, NY, USA, 16–24. https://doi.org/10.1145/1401890.1401898

[12] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2010. Efficient Algorithms for Large-Scale Local Triangle Counting. *ACM Trans. Knowl. Discov. Data* 4, 3, Article 13 (Oct. 2010), 28 pages. https://doi.org/10.1145/1839490.1839494

[13] Christos Bellas and Anastasios Gounaris. 2022. Exploiting GPUs for fast intersection of large sets. *Information Systems* (2022), 101992. https://doi.org/10.1016/j.is.2022.101992

[14] Mauro Bisson and Massimiliano Fatica. 2017. Static graph challenge on GPU. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, USA, 1–8. https://doi.org/10.1109/HPEC.2017.8091034

[15] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Softw. Pract. Exper.* 34, 8 (July 2004), 711–726. https://doi.org/10.1002/spe.587

[16] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2018. BUbiNG: Massive Crawling for the Masses. *ACM Trans. Web* 12, 2, Article 12 (June 2018), 26 pages. https://doi.org/10.1145/3160017

[17] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. Association for Computing Machinery, New York, NY, USA, 587–596. https://doi.org/10.1145/1963405.1963488

[18] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[19] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. 2006. Counting Triangles in Data Streams. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Chicago, IL, USA) *(PODS '06)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/1142351.1142388

[20] Ronald S Burt. 2004. Structural holes and good ideas. *American journal of sociology* 110, 2 (2004), 349–399.

[21] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy, In ICWSM. *AAAI Conference on Weblogs and Social Media* 14.

[22] Charles L Clarke, Nick Craswell, and Ian Soboroff. 2009. *Overview of the trec 2009 web track*. Technical Report. DTIC Document.

[23] Jonathan Cohen. 2009. Graph Twiddling in a MapReduce World. *Computing in Science & Engineering* 11 (2009), 29–42.

[24] James S. Coleman. 1988. Social Capital in the Creation of Human Capital. *Amer. J. Sociology* 94 (1988), S95–S120.

[25] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2018. Multi-threaded Sparse Matrix-Matrix Multiplication for Many-Core and GPU Architectures. *CoRR* abs/1801.03065 (2018), 24. arXiv:1801.03065 http://arxiv.org/abs/1801.03065

[26] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Trans. Parallel Comput.* 8, 1, Article 4 (April 2021), 70 pages. https://doi.org/10.1145/3434393

[27] Evan Donato, Ming Ouyang, and Cristian Peguero-Isalguez. 2018. Triangle Counting with A Multi-Core Computer. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, USA, 1–7. https://doi.org/10.1109/HPEC.2018.8547540

[28] Bin Dong, Surendra Byna, and Kesheng Wu. 2016. SDS-Sort: Scalable Dynamic Skew-Aware Parallel Sorting. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (Kyoto, Japan) *(HPDC '16)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/2907294.2907300

[29] Jean-Pierre Eckmann and Elisha Moses. 2002. Curvature of co-links uncovers hidden thematic layers in the World Wide Web. *Proceedings of the National Academy of Sciences of the United States of America* 99 (2002), 5825–5829.

[30] Brooke Foucault Welles, Anne Van Devender, and Noshir Contractor. 2010. *Is a Friend a Friend? Investigating the Structure of Friendship Networks in Virtual Worlds*. Association for Computing Machinery, New York, NY, USA, 4027–4032. https://doi.org/10.1145/1753846.1754097

[31] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A. Bader. 2018. Fast and Adaptive List Intersections on the GPU. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, USA, 1–7. https://doi.org/10.1109/HPEC.2018.8547759

[32] Oded Green, James Fox, Alex Watkins, Alok Tripathy, Kasimir Gabert, Euna Kim, Xiaojing An, Kumar Aatish, and David A. Bader. 2018. Logarithmic Radix Binning and Vectorized Triangle Counting. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, USA, 1–7. https://doi.org/10.1109/HPEC.2018.8547581

[33] Chuangyi Gui, Long Zheng, Pengcheng Yao, Xiaofei Liao, and Hai Jin. 2019. Fast Triangle Counting on GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, USA, 1–7. https://doi.org/10.1109/HPEC.2019.8916216

[34] Yang Hu, Hang Liu, and H. Howie Huang. 2018. TriCore: Parallel Triangle Counting on GPUs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, USA, 171–182. https://doi.org/10.1109/SC.2018.00017

[35] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[36] Eun-Jin Im and Katherine A Yelick. 1999. Optimizing Sparse Matrix Vector Multiplication on SMP. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1999*. Citeseer, SIAM, USA, 9.

[37] Alon Itai and Michael Rodeh. 1977. Finding a Minimum Circuit in a Graph. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/800105.803390

[38] Alon Itai and Michael Rodeh. 1978. Finding a minimum circuit in a graph. *SIAM J. Comput.* 7, 4 (1978), 413–423.

[39] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. A Space-Efficient Streaming Algorithm for Estimating Transitivity and Triangle Counts Using the Birthday Paradox. *ACM Trans. Knowl. Discov. Data* 9, 3, Article 15 (Feb. 2015), 21 pages. https://doi.org/10.1145/2700395

[40] Paul-Virak Khuong and Pat Morin. 2017. Array Layouts for Comparison-Based Searching. *ACM J. Exp. Algorithmics* 22, Article 1.3 (May 2017), 39 pages.

[41] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA.

[42] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. Exploiting In-Hub Temporal Locality In SpMV-Based Graph Processing. In *50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP 2021)*. Association for Computing Machinery, New York, NY, USA, 10. https://doi.org/10.1145/3472456.3472462

[43] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. How Do Graph Relabeling Algorithms Improve Memory Locality?. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, USA, 84–86. https://doi.org/10.1109/ISPASS51385.2021.00023

[44] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. Locality Analysis of Graph Reordering Algorithms. In *2021 IEEE International Symposium on Workload Characterization (IISWC'21)*. IEEE Computer Society, USA, 101–112. https://doi.org/10.1109/

IISWC53511.2021.00020

[45] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. Thrifty Label Propagation: Fast Connected Components for Skewed-Degree Graphs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, USA, 226–237. https://doi.org/10.1109/Cluster48925.2021.00042

[46] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173

[47] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[48] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science* 407, 1 (2008), 458–473.

[49] Oliver Lehmberg, Robert Meusel, and Christian Bizer. 2014. Graph Structure in the Web: Aggregated by Pay-Level Domain. In *Proceedings of the 2014 ACM Conference on Web Science* (Bloomington, Indiana, USA) *(WebSci '14)*. Association for Computing Machinery, New York, NY, USA, 119–128. https://doi.org/10.1145/2615569.2615674

[50] Yongsub Lim, U Kang, and Christos Faloutsos. 2014. SlashBurn: Graph Compression and Mining beyond Caveman Communities. *IEEE Transactions on Knowledge and Data Engineering* 26, 12 (Dec 2014), 3077–3089. https://doi.org/10.1109/TKDE.2014.2320716

[51] Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. 2017. Using Complex-Network properties For Efficient Graph Analysis. In *International Conference on Parallel Computing, ParCo 2017 (Parallel Computing is Everywhere)*, Vol. 32. Foundation ParCo Conferences and Consortium Cineca, IOS Press Ebooks, Bologne, Italy, 413–422. https://doi.org/10.3233/978-1-61499-843-3-413

[52] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2014. Graph Structure in the Web — Revisited: A Trick of the Heavy Tail. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14 Companion)*. Association for Computing Machinery, New York, NY, USA, 427–432. https://doi.org/10.1145/2567948.2576928

[53] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The Graph Structure in the Web – Analyzed on Different Aggregation Levels. *The Journal of Web Science* 1, 1 (2015), 33–47. https://doi.org/10.1561/106.00000003

[54] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[55] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement* (San Diego, California, USA) *(IMC '07)*. ACM, New York, NY, USA, 29–42.

[56] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[57] Alejandro Portes. 1998. Social capital: Its origins and applications in modern sociology. *Annual review of sociology* 24, 1 (1998), 1–24.

[58] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, USA, 4292–4293.

[59] Youcef Saad. 1994. Sparskit: a basic tool kit for sparse matrix computations - Version 2.

[60] Siddharth Samsi, Jeremy Kepner, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Paul Monticciolo. 2020. GraphChallenge.org Triangle Counting Performance. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, USA, 1–9. https://doi.org/10.1109/HPEC43674.2020.9286166

[61] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 979–990. https://doi.org/10.1145/2588555.2610518

[62] Thomas Schank and Dorothea Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental and Efficient Algorithms*, Sotiris E. Nikoletseas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 606–609.

[63] Julian Shun and Kanat Tangwongsan. 2015. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, USA, 149–160. https://doi.org/10.1109/ICDE.2015.7113280

[64] Friendster social network. 2011. Friendster: The online gaming social network. archive.org/details/friendster-dataset-201107.

[65] Martin Steinegger and Johannes Söding. 2018. Clustering huge protein sequence sets in linear time. *Nature Communications* 9 (06 2018). https://doi.org/10.1038/s41467-018-04964-5

[66] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning. In *2017 46th International Conference on Parallel Processing (ICPP)*. ACM, USA, 181–190. https://doi.org/10.1109/ICPP.2017.27

[67] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/3079079.3079097

[68] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2018. VEBO: A Vertex- and Edge-Balanced Ordering Heuristic to Load Balance Parallel Graph Processing. *CoRR* abs/1806.06576 (2018), 1–13. arXiv:1806.06576 http://arxiv.org/abs/1806.06576

[69] Michael Sutton, Tal Ben-Nun, and Amnon Barak. 2018. Optimizing parallel graph connectivity computation via subgraph sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 12–21. https://doi.org/10.1109/IPDPS.2018.00012

[70] Daniel Terpstra, Heike Jagode, Haihang You, and Jack J. Dongarra. 2009. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer, 157–173. https://doi.org/10.1007/978-3-642-11261-4_11

[71] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. 2009. DOULION: Counting Triangles in Massive Graphs with a Coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09)*. Association for Computing Machinery, New York, NY, USA, 837–846. https://doi.org/10.1145/1557019.1557111

[72] Hans Vandierendonck. 2020. Graptor: Efficient Pull and Push Style Vectorized Graph Processing. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*.

ACM, New York, NY, USA, Article 13, 13 pages. https://doi.org/10.1145/3392717.3392753

[73] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *nature* 393, 6684 (1998), 440–442.

[74] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1813–1828. https://doi.org/10.1145/2882903.2915220

[75] Howard T Welser, Eric Gleave, Danyel Fisher, and Marc Smith. 2007. Visualizing the signatures of social roles in online discussion groups. *Journal of social structure* 8, 2 (2007), 1–32.

[76] Abdurrahman Yasar, Sivasankaran Rajamanickam, Jonathan W. Berry, and Ümit V. Çatalyürek. 2020. A Block-Based Triangle Counting Algorithm on Heterogeneous Environments. *CoRR* abs/2009.12457 (2020), 1–13. arXiv:2009.12457 https://arxiv.org/abs/2009.12457

[77] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V. Çatalyürek. 2018. Fast Triangle Counting Using Cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, USA, 1–7. https://doi.org/10.1109/HPEC.2018.8547563

[78] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. 2017. FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 27–38. https://doi.org/10.1109/CGO.2017.7863726

[79] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-Aware Graph-Structured Analytics. *SIGPLAN Not.* 50, 8 (Jan. 2015), 183–193. https://doi.org/10.1145/2858788.2688507