DPTL+: Efficient Parallel Triangle Listing on Batch-Dynamic Graphs (Cover Letter)

Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, Xuemin Lin

We would like to thank the reviewers for their insightful and invaluable comments. We have revised the paper carefully according to the comments. A summary of major revisions is given below, followed by the point-to-point response made to each reviewer.

1) Rewrite the motivation part to justify the importance of improving the efficiency for batch update setting.
2) Improve the experiment part to include more explanations and newly designed experiments.
3) Through detailed discussions and experiments, we compare our proposed technique to more general dynamic graph processing techniques.
4) Proof-read and improve presentation.

Please kindly find our point-to-point responses below.

RESPONSE TO REVIEWER #1

Comment 1.1 **(D1)** Lack of novelty: I am not sure that (2) counts as a contribution of this paper. The idea of applying hashing to speedup intersections is quite known and even the authors own previous paper [13] applies the same idea. This should have made clear by the authors, as a reader might overestimate the novelty of this step. Moreover, given that DPTL (which does not use hashing) does not achieve very exciting results makes me question the significance of contributions of this paper.

Response. Yes, hashing is a well-known method for speeding up the intersection. In the revision, we clarify that our second contribution is the design of a novel triangle listing strategy such that we can use a time-efficient hashing technique: bitmap hashing in our problem setting to improve *both time complexity and practical performance*. In Section III-C of the revision, we stress that although the time complexity of lookup operation in various hashing techniques is $O(1)$, only the hashing technique without collision (e.g., bitmap hashing) can check the existence of an element with *exactly one* look-up operation. As mentioned in [12] and verified in our newly designed experiments using different hashing schemes (Fig. 9 in the revision), there is no clear advantage to apply the traditional hashing techniques (with collisions) to support the edge-oriented triangle listing algorithms, compared to the sort-merge based approach. In the revision, we stress that we can immediately enhance the search performance by using bitmap hashing techniques. However, the bitmap hashing technique needs n bits where n is the number of vertices. As stressed in the revision (Section III-C), we cannot afford to pre-build hash tables for all vertices or build the tables on the fly for every pivot edge processed. Thus, the key challenge in our algorithm is how to design a triangle listing strategy such that the bitmap hashing can be efficiently used in our problem setting, leading to our second contribution in this paper. Specifically, we design a new triangle listing strategy such that we build a bitmap hash table *only* once for each pivot vertex u processed, and ensure that $deg(u) \geq deg(v)$ is true for each pivot edge (u, v) with pivot vertex u.

To further justify the second contribution, we design a new experiment in Fig. 9 of the revision. Particularly, we use the standard C++ hash table implementation in STL for the neighborhood intersection computation of DPTL, namely DPTL-h1, and build a hash table for every vertices. Recall that we use sort-merge for the intersection computation in DPTL. In addition to the large memory required, DPTL-h1 does not show superior performance in terms of processing time on *uk-2005* graph compared to DPTL, while our proposed DPTL+ significantly outperforms DPTL under all settings.

Regarding the performance of DPTL, in the revision we highlight that (1) DPTL can ensure that each updated triangle is output exactly once, and only the updated edges are involved in the neighborhood intersection of two ending vertices, which is important specially when the batch size is small; (2) In the experiments, we stress that DPTL significantly outperforms the baseline algorithms when the batch size is small. We also explain why its performance becomes less competitive when the batch size grows, compared to the baseline algorithms. Note that this problem is addressed by DPTL+ in our paper, as the second contribution. ■

Comment 1.2 **(D2)** Parallelization speed-ups seem low: Another point that is a bit surprising to me and deserved more attention in the manuscript is the low speed-up due to the parallelization. As the authors mention, the task is embarrassingly parallel, but figure 12 does not reflect that.

Response. We would like to remark that the processing time reported in figure 12 (now Fig. 8 in the revision) is the total execution time, which consists of two parts: (1) update of the graph and (2) list of the updated triangles. The graph update time is the same for all algorithms. We make these clear in the revision.

We notice that graph update, although very fast, still takes a certain portion of the total execution time if the triangle listing is fast. Moreover, compared to triangle listing, it is difficult to achieve a good parallelization speed-up for graph update because of a large number of write operations involved. These make the parallel speed-up of the total execution time not very impressive when the triangle listing time is fast.
In the revision,

(1) We decompose the total execution time in the Fig. 8. We report that, in terms of triangle listing time which is the focus of our study, our proposed algorithms can achieve a good speed-up. It is reported that the speed-up of the *triangle listing time* of DPTL+ with 32 threads is 21, 10, 16, 11 on graphs uk-2014-host, uk-2005, twitter-2010, sk-2005 respectively, compared to DPTL+ with single thread. The corresponding speed-up of DPTL (resp. AOT*) is 28 (resp. 6), 19 (resp. 2), 20 (resp. 7), 21 (resp. 4), respectively.
(2) We improve the parallel implementation of graph update in the revision. Particularly, we try different OpenMP parallel schedule compiling options and find that the static option is faster than the dynamic option used in our previous implementation. Therefore, we use the static option for graph update when testing all algorithms in the revision. This can enhance the speed-up of the total execution time when graph update occupies a certain portion of the total time. For instance, the speed-up of execution time of *uk-2014-host* is 7 times in the previous submission, and now it is 18 in the revision. As shown in Fig 8 of the revision, the speed-up of the *processing time* of DPTL+ with 32 threads is 18, 6, 12, 7 on graphs *uk-2014-host*, *uk-2005*, *twitter-2010*, *sk-2005* respectively, compared to DPTL+ with single thread. The corresponding speed-up of DPTL (resp. AOT*) is 25 (resp. 6), 17 (resp. 2), 18 (resp. 7), 16 (resp. 4), respectively. ■

Comment 1.3 **(D3)** Some experiments are inconclusive: The third negative point about this paper is the way that the workloads were constructed. The proposed method gains decrease with batch size but edges are selected at random, For small batches, it is not clear how many triangles will actually change. If such number is very small, then a user would probably increase the batch size, making your method less attractive. The number of modified triangles should also be reported in the results.

Response. Thanks for your suggestion. In the revision,
(1) As suggested, we add a new figure (Fig. 6) to report the number of modified triangles relative to the batch-size. It shows that there are a large number of modified triangles even with a small batch-size. For instance, in graph *sk-2005*, there are around 36 million and 411 million modified triangles for batch-size of 0.01 percent and 0.1 percent, respectively.
(2) In the revision, we also report that a significant portion of the graph is affected even for a small batch-size. For example, the size of an *influenced graph* of *sk-2005* for batches size of 0.01 percent and 8 percent are 1,082 million and 1,717 million respectively.
(3) We notice that the graph *stackoverflow* used in the experiments is a temporal graph where each edge is associated with a timestamp. In the revision, we design a new experiment (Fig. 11) to evaluate the performance of the algorithms on different batch selection methods: random sampling method and time based method in Fig. 11 (a) and Fig. 11 (b), respectively. Though the running time of each algorithm is different on two batch selection methods, we observe the same performance rankings and similar trends of the algorithms in two figures. ■

Comment 1.4 (1) Batch-Dynamic Graph definition of G'; (2) a typo in section 3B; (3) confusing notation in experiments; (4) moving correctness proof to the appendix; (5) more discussion about counting version of the problem.

Response. Thanks for your suggestion. We have modified the paper in the revision accordingly. ■

RESPONSE TO REVIEWER #2

Comment 2.1 **(D1)** According to Section III.B, the baseline AOT* must compute the intersection of every edge in the influenced graph, including both edges in the update batch and in the original graph. Is there a way to prune early in this case, e.g., skipping an edge if it is not in the update batch? A single bit per vertex may simplify this process.

Response. In the revision, we show an example in Section III-A that we have to compute the neighborhood intersection of an original edge in AOT* since it is chosen as the pivot edge by the algorithm. This implies that we cannot simply skip an edge that is not in the update batch for AOT*. Actually, the main contribution of DPTL is the design of a new updated triangle listing strategy such that we only need to compute the neighborhood intersection on the edges in the update batch without missing any result. ■

Comment 2.2 **(D2)** The proposed algorithm is a little narrow and does not apply to other graph processing problems. For example, the following two papers proposed more general frameworks of incremental graph processing:

1) Incremental Graph Pattern Matching, SIGMOD'11
2) GraphIn: An Online High Performance Incremental Graph Processing Framework, Euro-Par 2016

The paper should make qualitative/quantitative comparison to these previous works. In particular, how does the proposed algorithm compare to more general graph processing frameworks mentioned in D2? If they are different, can the proposed algorithm be integrated into an existing framework?

Response. Thanks for your suggestion. In the revision,
(1) We add a new subsection (Section VII-B) to discuss if our proposed techniques can be effectively integrated into the general subgraph processing framework. Through detailed discussion, we show that, same as existing triangle listing algorithms, the core of the proposed techniques in our paper is specific to the triangle structure, and hence cannot be used for general subgraph processing framework. For instance, the key of the orientation technique designed for DPTL and DPTL+ is to use an updated edge as pivot edge to uniquely identify a triangle such that we output each triangle exact once, but we cannot do this on larger subgraphs. Similarly, for larger graph patterns, multiple pivot edges will be involved in the neighborhood

intersection computation, we cannot ensure $deg(u) \geq deg(v)$ for every pivot edge $u \rightarrow v$, and hence cannot guarantee that the time complexity of $\Theta(\min\{deg(u), deg(v)\})$ for each edge $u \rightarrow v$.
(2) In the revision, we also stress that the problem of triangle enumeration problem itself has many real-life applications, and many efficient triangle listing algorithms have been proposed under different settings.
(3) We also add a new subsection (Section VI-C) in the related work to briefly introduce the incremental subgraph pattern matching algorithms. We show that the problem of triangle listing is a special case of subgraph enumeration, and hence any subgraph enumeration algorithm can be immediately used for triangle enumeration by setting the query pattern as a triangle. Nevertheless, their algorithms are not specially designed for the triangle listing, and it is not promising to directly apply them to the problem of triangle listing. When we worked on the paper revision, we got source code from authors of SIGMOD'11 paper [29]. We also note that the author of Euro-Par'16 paper [30] replied that the project had not been open-sourced. Not surprisingly, the performance of SIGMOD'11 paper is not efficient when it is used on the problem of triangle listing on batch-updated graphs, and even cannot compete with our naive algorithm due to some overhead for the support of general graph processing. For instance, on the ego-Facebook graph from SNAP (https://snap.stanford.edu/data) with $4,039$ nodes and $176,468$ edges, the implementation from SIGMOD'11 takes 684 seconds under our default settings. While DPTL+ only takes 0.04 seconds. Considering the SIGMOD'11 paper is not designed for triangle listing and none of the graphs deployed in this paper can be properly run by their implementation when the pattern is set to a triangle, we did not include this comparison in the experiments of this paper. ■

RESPONSE TO REVIEWER #3

Comment 3.1 **(D1)** The authors fail to justify the importance of why they need to improve the efficiency for batch update setting. The authors aim to propose efficient algorithms for solving the triangle listing problem in batch update setting. However, as pointed out by the authors in the Introduction: "many applications do not require strong real-time response, and can therefore afford to postpone updates for a later time or to process updates on a fixed periodic basis." Therefore, the efficiency issues for batch update may not be very important. The authors should justify why they need to improve the efficiency for batch update setting.

Response. As suggested, in the revision we rewrite the introduction to justify why we need to improve the efficiency for batch update setting from two perspectives.
(1) Indeed, the efficiency is not very important from users' perspective if they do not have real-time requirements. In the revision, we stress that this is important for the throughout of the system. As discussed in the paper, we may significantly improve the average processing time by processing updates in a batch fashion. In practice, triangle listing is one of many tasks on the system. By developing efficient algorithm for batch-updated graph, we may reduce the use of system resource and hence enhance the throughout of the system.
(2) In the revision, we also highlight that, in some scenarios the update of the edges may arrive at the system in batches, therefore it is natural to develop efficient algorithms for bach update setting. For instance, rather than immediately sending every update, other parties may instead periodically send updated information to the system to save communication costs. ■

Comment 3.2 **(D2)** In experimental settings, both AOT*, DPTL, DPTL+ are the authors' previous work and current work. However, since triangle listing is the famous problem, I really doubt whether some other research studies, e.g., [21], can be properly adapted to this setting.

Response. Yes, triangle listing is a famous problem and there are many existing works under different settings. In the revision, we provide more discussions and design a new experiment to justify that we already use reasonable baselines in the paper.
(1) We added a new subsection (Section VII-A) to discuss how to adapt existing triangle listing algorithms to our setting. Clearly, all existing main memory triangle listing algorithms can be immediately applied on the influenced graph to enumerate updated triangles. We can also consider other studies under other settings such as external memory (e.g., [17] ([21] in previous submission)) or distributed (e.g.,[16]) computing environments. For instance, the graph is partitioned into different groups in [17] such that each partition can be fit into the main memory. After computing the triangles in each individual group, they need to compute the triangles across different groups. To compute the triangles across two groups A and B, we may regard this as the problem of triangle listing in a batch-update setting; that is, A is the original graph and B is the batch-updates. However, this is not the research focus of [17], and the existing main memory triangle listing algorithm is directly used on the extended subgraph, where the definition of extended subgraph is similar to the influenced graph in our paper. Similarly, no special optimization is considered on this aspect in other studies. Thus, as to our best knowledge, a reasonable baseline is to apply the state-of-the-art main memory triangle listing algorithms on the influenced graph.
(2) We add a new experiment (Fig. 10 in the revision) in which two representative main memory triangle listing algorithms, CF [8] and KClist [13], replace AOT for listing triangles on the influenced graph. However considering that neither CF nor KClist are state-of-the-art, it is reported that their corresponding algorithms under our settings, namely CF* and KClist*, are clearly outperformed by AOT*. ■

Comment 3.3 Typos and incomplete sentence.

Response. Thanks, we fixed the problem and conducted a thorough proof-reading in the revision. ■

DPTL+: Efficient Parallel Triangle Listing on Batch-Dynamic Graphs

Michael Yu[†], Lu Qin[‡], Ying Zhang[‡], Wenjie Zhang[†], Xuemin Lin[†]

[†]University of New South Wales, Australia
[‡]AAII, University of Technology Sydney, Australia
{mryu, zhangw, lxue}@cse.unsw.edu.au {lu.qin, ying.zhang}@uts.edu.au

***Abstract*—Triangle listing is an important topic in many practical applications. We have observed that this problem has not yet been studied systematically in the context of batch-dynamic graphs. In this paper, we aim to fill this gap by developing novel and efficient parallel solutions. Specifically, given a graph G and a batch-update of edges B, we report the updated triangles (deleted triangles and new triangles) resulting from the batch of updates. We notice that it is cost expensive to directly apply state-of-the-art triangle listing algorithms because they are designed to enumerate the complete set of triangles from a given graph, whereas only the updated ones are the relevant output for our problem setting. In this paper, we developed an efficient algorithm, namely DPTL, based on a newly designed orientation technique, which only outputs the updated triangles while ensuring that each triangle solution is identified without any duplicate solutions. We follow up by taking advantage of a graph's degree distributions and designed a more sophisticated algorithm, namely DPTL+. We show that DPTL+ can achieve the best performance in terms of both practical performance and theoretical time complexity. Our comprehensive experiments over 28 real-life large graphs show the superior performance of the DPTL+ algorithm when compared against DPTL and two baseline solutions. Theoretically, we also show that DPTL+ has a time complexity of $\Theta(\sum_{\langle u,v\rangle \in B} \min\{deg(u), deg(v)\}+m)$ where $deg(x)$ is the degree of a vertex x, and m is the number of edges adjacent to the vertices in the batch-update. This time complexity is more promising than that of other solutions.**

I. Introduction

The *triangle-listing problem* is a fundamental problem in the area of graph analytics, where the complete set of all triangle structures in a simple undirected graph is returned. In network analysis, triangle listing algorithms serve as an important tool. Triangle structures provide valuable connective information about objects that are closely connected. It has a wide array of applications and can be seen playing a critical role in topics such as community search [1], [2], role discovery [3], structural clustering [4], higher-order graph clustering [5], web spam discovery [6], and other real-world applications that practically benefit from the mining of triangle structures.

Batch-Dynamic Graph Motivation. Dynamic graphs are increasingly prevalent as many applications have to face high volumes of updates on graphs that are constantly changing. A typical dynamic graph application receives updates and immediately processes them. Nevertheless, in some scenarios, users do not require strong real-time response, and can therefore afford to postpone updates for a later time or to process updates on a fixed periodic basis. We may significantly improve the average processing time by processing updates in a batch fashion. In practice, triangle listing is one of the many tasks on the system. By developing efficient algorithm for batch-updated graph, we may reduce the use of system resource and hence enhance the throughout of the system. In some applications the update of the edges may arrive the system in batch, and it is natural to develop efficient algorithms for bach update setting. For instance, instead of immediate sending every update, other parties may periodically send updated information to the system to save the communication costs. We also notice the increasing prevalence of computational devices that support multiple-core processing, and batched operations processing is a good candidate to take advantage of parallel algorithm designs [7].

By considering all above factors, we find that there exists a natural motivation to design efficient parallel triangle listing algorithms that consider batch-updates of dynamic graphs. Some graph algorithms are developed in the literature for a variety of graph problems in a batch-dynamic context. However, as far as we are aware, there has not been any systematic study into the problem of triangle listing on batch-dynamic graphs, not to mention any parallel implementations. To fill this important gap, in this paper we develop efficient parallel triangle listing algorithms on batch-dynamic graphs. Specifically, given a graph G and a batch-update B, we aim to develop an efficient parallel algorithm to report the *updated triangles* (i.e., deleted original triangles and inserted new triangles) resulting from the batch-update.

Challenges. Listing triangle structures on a dynamic graph is different to that on static graphs. A major challenge for triangle listing on dynamic graphs is to list only the solutions that are affected by the update; In comparison, existing triangle listing algorithm for static graphs only perform a complete re-computation of all triangle solutions in an updated graph. Another major challenge is how an algorithm has to manage many types of triangles whilst ensuring a superior time complexity. Finally, keeping expensive neighborhood intersection operations to a minimal number of edges whilst ensuring the correctness of the solution, and preventing duplicated triangles from appearing in the output is also a challenge.

Contribution. In this paper, we propose to study this yet to be explored problem and make the following contributions.

- This is the first systematic work for the problem of triangle-listing on a batch-dynamic graph setting.
- We design a parallel triangle-listing algorithm for

TABLE I
THE SUMMARY OF NOTATIONS

Notation	Definition
$G = (V, E)$	undirected graph with vertices V and edges E
$\vec{G}$	directed graph with vertices V and directed edge $\vec{E}$
B	batch-update consists of a set of inserted/deleted edges
$B^+(B^-)$	inserted (deleted) edges in batch-update B
Δ_G	set of triangles in graph G
$V(G)$	set of vertices in graph G
u, v, w, x, y, z	vertices in graph
(u, v)	undirected edge with vertex u and v
$\langle u, v\rangle$, $u \to v$	directed edge from vertex u to v
(u, v, w)	triangle with vertices u, v and w
$deg(u)$	degree of vertex u
$deg^+(u)$	out-degree of vertex u in an oriented graph

batch-dynamic graphs, named *Dynamic Parallel Triangle-Listing (DPTL)* by designing a new graph orientation technique, with a time complexity of $\Theta(\sum_{\langle u,v\rangle \in B} \max\{deg(u), deg(v)\})$, where $deg(x)$ denotes the degree of the vertex x, and B is the set of edges in the batch-update. Note that DPTL outputs each updated triangle exactly once, and only conducts neighborhood intersection operation for updated edges.

- We propose an improvement algorithm to DPTL, namely *DPTL+*, by developing a novel triangle listing strategy such that we can use a time-efficient hashing technique: bitmap hashing in our problem setting to improve *both time complexity and practical performance*. Particularly, we are able to bound the amount of computation with a good time complexity of $\Theta(\sum_{\langle u,v\rangle \in B} \min\{deg(u), deg(v)\} + m)$, where m is the number of edges adjacent to the vertices in the batch-update.
- We conduct an extensive performance study using 28 real-world large graphs up to billion scale, and demonstrate the high efficiency of our proposed solutions compared to the baseline methods.

II. BACKGROUND

In this section, we give a formal introduction of the problem studied. We first cover the set of notations that appear in this paper, a summary of which is shown in Table I.

A. Notations and Problem Definition

Let $G = (V, E)$ be an undirected simple graph, where V refers to its set of vertices, and E refers to its set of edges. $V(G)$ and $E(G)$ are also used to denote the sets V and E in graph G respectively. The number of vertices and the number of edges are denoted as n and m where $n = |V|$ and $m = |E|$, respectively. We denote the set of neighbors of any vertex u in an undirected graph G as $N(u)$. We denote the degree of vertex u in G as $deg(u)$, where $deg(u)$ is $|N(u)|$. For directed graphs $\vec{G} = (V, \vec{E})$, we use $\vec{E}$ to denote the set of directed edges. Directed edges are denoted as $\{\langle u, v\rangle\}$ or $\{u \to v\}$ where u and v are the starting and ending vertex respectively. For vertices in a directed graph, the set of outgoing-neighbors of vertex u in $\vec{G}$ is denoted as $N^+(u)$, the out-degree of u is denoted as $deg^+(u) = |N^+(u)|$. Likewise, the in-neighborhood of vertex u in $\vec{G}$ is denoted as $N^-(u)$, and its in-degree as $deg^-(u) = |N^-(u)|$. We refer to an undirected edge as (u, v) between vertices u and v . A **triangle** consisting of vertices u, v and w is denoted by (u, v, w), where a triangle is defined as a set of three vertices that are fully connected.

Batch-Dynamic Graph. In this paper, we consider a *batch-dynamic* setting where a *batch* of updates contains a mix of edge insertion or edge deletion operations. We use B to denote the set of updated edges from a batch-update. The set of inserted edges is denoted using B^+ and the set of deleted edges is denoted using B^-. We make the assumption that B^+ and B^- are disjoint (i.e., $B^- \cap B^+ = \emptyset$), since any common edge(s) found in both sets can be removed from each batch-update without affecting the final triangle list. In particular, we use $G' = (V, E \cup B^+ - B^-)$ to denote the updated graph after a batch-update B is applied to the original graph G which adds new edges in B^+ and deletes edges in B^-. Note that we only consider edge updates in the paper. This is because insertions or deletions of the graph vertices can also be expressed using edge updates.

Influenced Graph. To output a list of every triangle affected by a batch-update, we have to consider the neighborhood of two vertices from every updated edge. With reference to Fig.1 (a), given an updated edge (b, d) (dash line) we identify the updated triangle (b, c, d) by checking all vertices that belong to the neighborhood of the vertices b and d (i.e., vertices a, b, c and d). In this paper, we use V_B^1 to denote the set of vertices incident to at least one updated edge in B. We also use V_B^2 to denote the set of vertices in V_B^1 and all vertices in G that are adjacent to any vertex in V_B^1. In this paper, we will use G_{inf} to denote a graph containing (1) the induced subgraph of G regarding the vertices in V_B^1, and (2) the neighbors of V_B^1 (i.e., V_B^2) and all edges between V_B^1 and V_B^2. Namely, we refer to G_{inf} as the *influenced graph* in relation to a batch-update B. It is apparent that we only need to consider the influenced graph G_{inf} to list all updated triangles. For simplicity, when referring triangle listing computations on an updated graph, we are considering only the influenced graph where there is no chance for ambiguity.

Problem Statement. Given an undirected simple graph $G = (V, E)$, we aim to list the complete set of all newly inserted and deleted triangles after a given batch of edge updates B. We also aim to develop a main-memory parallel algorithm that has a good time complexity and practical performance. Our algorithm should also produce a correct and complete list of new/deleted triangles after a batch of edge updates. More specifically, let Δ_G and $\Delta_{G'}$ denote the set of all triangles in the original graph G and all the triangles in a graph G' after a batch-update is processed. The triangle solutions we list include all triangles from $\{\Delta_{G'} - \Delta_G\}$ (new triangles inserted from update) and those from $\{\Delta_G - \Delta_{G'}\}$ (triangles deleted from update).

B. Orientation technique

One of the key challenges in triangle listing is in ensuring that each triangle solution gets listed exactly once. To address this issue, some *orientation techniques* have been proposed and adopted in the literature relating to triangle listing algorithms. The key idea is to generate a directed (i.e., oriented) acyclic graph $\vec{G}$ from an initially undirected input graph G [8]. Each undirected edge is mapped to a directed edge where its

direction (i.e., orientation) is decided by the relative rank of its endpoints based on a vertex-ordering (e.g., vertex IDs or vertex degree) [8]. Given a triangle (u, v, w), we refer to a vertex u as a *pivot vertex* if u has two out-going edges to vertices v and w (i.e., $u \rightarrow v$ and $u \rightarrow w$) in the oriented graph; We refer to a directed edge $u \rightarrow v$ as *pivot edge* if the vertex v has one incoming edge and one out-going edge. We note that each triangle in the undirected graph is associated with only one pivot vertex and one pivot edge since there exists no cyclic triangle in any oriented graph. Fig.1 shows an example of an oriented graph, where Fig.1(b) shows the directed version of the undirected graph from Fig.1(a) using an orientation based on a vertex ID ordering. For instance, the triangle (b, c, d) is reported when b is processed as pivot vertex with pivot edge $b \rightarrow c$.

The advantage of the oriented technique is two-fold: Firstly, by simply processing all pivot vertices using the above procedure, it already guarantees that each triangle is generated only once without having to otherwise perform any pruning for the removal of duplicate triangle solutions. Secondly, algorithm can be easily made parallel by processing sets of pivot vertices independently. To our best knowledge, the AOT algorithm [9] is the state-of-the-art main memory triangle listing algorithm with a time complexity of $\Theta(\sum_{\langle u,v\rangle \in \vec{E}} \min\{deg^+(u), deg^+(v)\}))$, where $\vec{E}$ and $deg^+(x)$ denote the set of directed edges in an oriented graph and the out-degree of vertex x respectively.

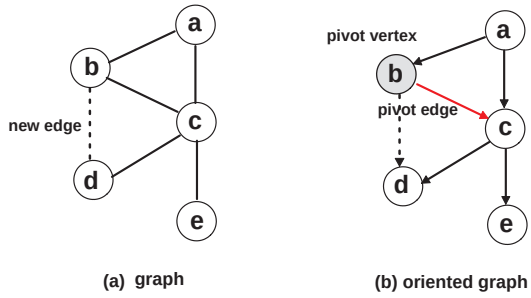

Fig. 1. Graph Orientation

III. Insertion-only batch-dynamic graphs

In this section, we simplify the problem to that of processing only batches containing only edge insertions, because batch deletion operations can be processed in a very similar fashion. We follow up by summarizing the final algorithm in a mixed batch operation setting in the next section. In Section III-A, we discuss two baseline solutions and show their limitations. Section III-B and Section III-C introduce two solutions for triangle listing on batch-dynamic graphs.

A. Warm-up

It is apparent that each updated triangle solution will include at least one edge from the edge-update batch. As a result, an intuitive and straightforward solution would be to iterate through each update edge (u, v) and list all affected triangles found when checking the neighborhood of the two endpoints. When the updated edges in B are processed *sequentially*, as illustrated in Algorithm 1, each new triangle will output only once because an updated triangle only outputs when the last new edge arrives. The time complexity of this serial **Naive** algorithm is $\Theta(\sum_{\langle u,v\rangle \in B} \max\{deg(u), deg(v)\}))$, where $deg(x)$ denotes the degree of vertex x on the influenced graph G_{Inf}. Despite the simple effective design of Algorithm 1, it is difficult to achieve an efficient parallel implementation. If we consider directly processing new edges in parallel at Line 1 of Algorithm 1, we are likely to overlook a significant portion of new triangle solutions. As an example, any new triangle with two new edges can easily be neglected if those two new edges are processed at the same time but on different threads. There is no efficient trivial solution to address this issue. We are able to account for every unique new triangle if we process the new edges in parallel on the updated graph after inserting all new edges from a given batch-update. Processing the set of new-edges across multiple threads on this updated graph will produce the complete set of new triangles. However, a drawback is that duplicate triangles will unavoidably be produced. The removal of duplicate triangles may take $O(n_\Delta \lg(n_\Delta))$ time where n_Δ is the number of updated triangles. In practice, n_Δ is significantly greater than the number of edges in B (e.g., tens of billions of triangles for million scale graphs). As shown in our initial experiments, a parallel implementation on multiple threads is much slower than the serial (i.e.,single thread) algorithm under our experimental settings, despite taking into account that the removal of duplicates can be processed in parallel.

Algorithm 1: Naive Method

```
   Input  : G : undirected graph; B : batch-update
   Output : new triangles generated after batch-update
1  for (u, v) ∈ B do
2      for w ∈ N(u) ∩ N(v) do
3          print (u,v,w);
4      Update G;
```

Another solution can be to apply a state-of-the-art main memory triangle listing algorithm against the influenced graph G_{Inf}, we refer to this solution as **AOT***. The time complexity of AOT* is $\Theta(\sum_{\langle u,v\rangle \in E_{Inf}} \min\{deg(u), deg(v)\}))$, where E_{Inf} denotes the set of edges in the influenced graph G_{Inf}. Although this algorithm is efficient in terms of listing all triangles in G_{Inf}, it will encounter existing triangles from the original graph G during the computation. For instance, the edge $\langle b, c\rangle$ in Fig. 1(b) is not an updated edge, but it will be chosen as pivot edge for neighborhood intersection computation. In comparison, only the updated edges are processed for Algorithm 1 (i.e., join operation for the neighbors of two ending vertices at Line 2), whereas every edge will be processed in AOT^* to ensure the correctness of the algorithm. The advantage of AOT^* is that the parallel implementation is readily available, whereas this is not the case for the naive algorithm.

B. Dynamic Parallel Triangle List Algorithm

In this subsection, we introduce our solution based on a newly developed orientation technique, namely DPTL. We begin by explaining our motivation, followed by descriptions of the algorithm. We finish with a thorough analysis split into

3 parts, where we cover the space cost, time complexity as well as the correctness of our algorithm.

(1) Motivation. As discussed in Section III-A, the parallel implementation of naive algorithm will inevitably produce duplicate triangle solutions, the overhead of sorting and removing duplicates is a significant bottleneck that leads to a poor performance. Where by applying an orientation technique on an influenced graph, the parallel implementation of AOT^* is able to avoid issues of duplication solutions, but it ultimately has to compute an intersection for every edge in the influenced graph, which includes original edges that do not participate in the formation of any new/updated triangle solutions.

One may wonder if it is possible to develop a new technique that (1) lists each updated triangle only once in the parallel implementation; while all the more, (2) only computes the intersection for the updated edges in B exclusively. We find that the existing orientation technique introduced in Section II-B naturally resolves the first goal, however the orientation technique means that every edge will have to be processed once to avoid missing any updated-triangle solution, and therefore does not address the second goal. This motivates us to develop a more sophisticated orientation technique to satisfy both desired properties.

In addition to considering a direction of the initially undirected edge in an oriented graph, we also consider whether the edge is a newly updated edge from B or an edge originally from graph G. Note that by considering this additional property, each edge now has four possible states depending on its direction as well as edge origin (i.e., from B or from G). We note here the formation of cyclic triangle (clock-wise or anti clock-wise) is impossible given the nature of orientation technique. Our key idea is to identify a unique *pivot edge* for each updated triangle to perform the intersection of its starting and ending vertices, such that we can prevent a unique triangle from being processed multiple times.

Since each updated new triangle contains at least one new edge in B, as illustrated in Fig. 2, we can categorize the updated triangles into one of three types, where each triangle solution belongs to exactly one of the three. With reference to Fig. 2, Δ_1, Δ_2, and Δ_3 refer to updated triangles with one, two and three new edges in B, respectively. The role of an edge in a triangle is either a new edge or an original edge. We label these two edge roles differently. If the line is solid, it represents an edge from graph G; otherwise it is dashed and represents an edge from an update batch B. A pivot edge is highlighted with red color. Moreover, an undirected line is equivalent to the edge having both directions.

Below, we show how to choose the pivot vertex and pivot edge for triangles in three categories. We leave the proof of the correctness to the end of this subsection.

- (Case 1, Δ_1) For each triangle in Δ_1, we choose the starting vertex of the new edge and the new edge as the pivot vertex and pivot edge, respectively. As shown in Fig. 2(a), vertex u is the pivot vertex and the new edge $u \to v$ is the pivot edge.
- (Case 2, Δ_2) For each triangle in Δ_2, we choose the vertex x with two new edges as the pivot vertex. The edge $x \to y$ or $y \to x$ is used as pivot edge where y is the starting vertex of the original edge. As shown in Fig. 2(b), vertex u is the pivot vertex, and the new edge $u \to v$ or $v \to u$ is the pivot edge.
- (Case 3, Δ_3) For each triangle in Δ_3, we choose the vertex x with two out-going new edges as the pivot vertex. The edge $x \to y$ is used as pivot edge where the vertex y has one in-going edge and one out-going edge. As shown in Fig. 2(c), vertex u is the pivot vertex, and the new edge $u \to v$ is the pivot edge.

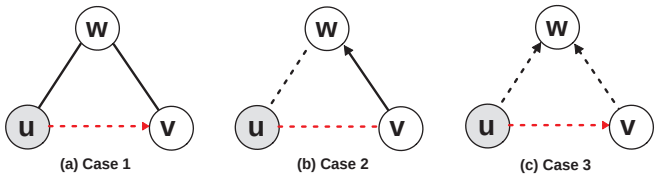

Fig. 2. Orientation technique for DPTL

(2) Description of the Algorithm (DPTL). We take the time to reiterate that this part considers the case where the batch-set B consists only of new edges. Additionally, we use G to denote the relevant influenced graph after including the batch-update for ease of presentation. We apply the degree based orientation technique [10], [11], [8] to convert the influenced graph G into a directed influenced graph $\vec{G}$. Particularly, we use $deg_{\vec{G}}(u)$ and $deg^{+}_{\vec{G}}(u)$ to denote the degree and out-degree of a vertex u in the oriented graph $\vec{G}$. For an undirected edge (u, v) in G, it becomes a directed edge $u \to v$ if $deg_{\vec{G}}(u) >= deg_{\vec{G}}(v)$ where a tie is broken by vertex IDs.

A pseudo-code of the method is shown in Algorithm 2 where three cases shown in Fig. 2 are carefully considered. The algorithm takes in two parameters: the oriented influenced graph $\vec{G}$ and the set of edges from a batch-update B. Let $V(B)$ denote the vertices in the new edges B, we can process them as pivot vertices in a parallel fashion. For each pivot vertex $u \in B$, we output its corresponding updated triangles (Lines 1-16). At Lines 3-6, all triangles with u as pivot vertex and new edge $u \to v$ (i.e., Case 1) as the pivot edge are output, where $N^{+}_{B}(u)$ denotes the out-going neighbors of u in $\vec{G}$ where (u, v) is a new edge in B. At Lines 8-11, all triangles with u as pivot vertex with two new edges (i.e., Case 2) are output, where $N_B(u)$ denotes both out-going and in-going neighbors of u in $\vec{G}$ where (u, v) is a new edge in B. Note that both $u \to v$ and $v \to u$ may be used as pivot edges. The third case is handled at Line 13-16 where each triangle consists of three new edges, with u and $u \to v$ as pivot vertex and pivot edge, respectively.

***Example** 1:* We further explain the DPTL algorithm by a running example illustrated in Fig. 3. When v_0 is processed as the pivot vertex, the triangle (v_0, v_1, v_9) is reported in Case 1 with $v_0 \to v_1$ as the pivot edge since (v_0, v_9) and (v_1, v_9) are original edges and v_9 is the neighbor of both v_0 and v_1. The triangle (v_0, v_1, v_7) is output under Case 2 with $v_0 \to v_7$ as the pivot edge. When v_5 is processed as the pivot vertex, the triangle (v_5, v_0, v_6) is reported under Case 2 with $v_5 \leftarrow v_0$ as the pivot edge. With v_6 as the pivot vertex, the triangle (v_6, v_4, v_5) is output under Case 3 with $v_6 \to v_4$ as the pivot edge.

Algorithm 2: DPTL

Input : $\vec{G}$: the oriented influenced graph after batch-update
Input : B: new edges in batch-update
Output : new triangles resulting from batch-update

```
1  for each u ∈ V(B) in parallel do
2      /*CASE 1*/
3      for v ∈ N_B^+(u) do
4          W ← N_Ḡ(u) ∩ N_Ḡ(v) ;
5          for w ∈ W do
6              print (u, v, w);
7      /*CASE 2*/
8      for v ∈ N_B(u) do
9          W ← N_B(u) ∩ N_Ḡ^+(v) ;
10         for w ∈ W do
11             print (u, v, w);
12     /*CASE 3*/
13     for v ∈ N_B^+(u) do
14         W ← N_B^+(u) ∩ N_B^+(v) ;
15         for w ∈ W do
16             print (u, v, w);
```

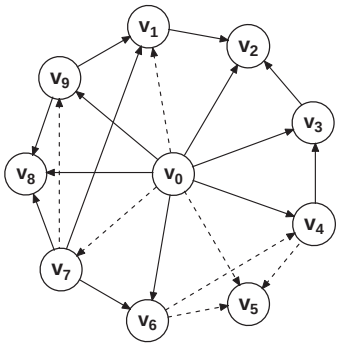

Fig. 3. Running Example

(3) Space Complexity. DPTL is very space efficient because no extra data structure is required in Algorithm 2. Thus the space complexity of DPTL is $O(m)$ where m is the number of edges in the influenced graph.

(4) Time Complexity. The dominant cost of Algorithm 2 is the intersection of neighbor sets for two ending vertices of the pivot edge. Assuming the neighbors of each vertex are already sorted, the time complexity for each pivot edge is $\Theta(\max\{deg(u), deg(v)\})$. Thus, the time complexity is $\Theta(\sum_{\langle u,v\rangle \in B} \max\{deg(u), deg(v)\})$ since only the edges in B are used as pivot edges.

As we output each updated triangle exactly once, and there is no resource competition, the parallel implementation of Algorithm 2 is straightforward where the pivot vertices at Line 1 can be processed in parallel.

(5) Correctness Analysis. Please refer to Section IX-A for correctness analysis of Algorithm 2.

C. Advanced Dynamic Parallel Triangle List Algorithm

In this subsection, we further enhance the performance of DPTL algorithm by carefully considering the degree distribution information in graphs. This additional technique is added to produce a new algorithm, namely **DPTL+**.

(1) Motivation. When analyzed as a serial algorithm, DPTL shares the same time complexity with the naive algorithm , where the dominant cost of both algorithms is in the set-intersection of the neighborhoods for each new edge (u, v) in B, leading to a time complexity of $\Theta(\max(deg(u), deg(v)))$ as the sort-merge is used for intersection operation. It is well known that we may improve the above time complexity to $\Theta(\min(deg(u), deg(v)))$ by applying hashing techniques. Suppose we have a hash table H that is initialized to the neighborhood of a vertex u (i.e., $N(u)$), we are then able to compute $N(u) \cap N(v)$ of an edge by looking up each neighbor of v (i.e., $N(v)$). If we **ensure** that $deg(u) \geq deg(v)$ is true, it comes to the time complexity $\Theta(\min(deg(u), deg(v)))$. Ensuring that $deg(u) \geq deg(v)$ is achievable if we always choose the endpoint vertex with larger degree as the vertex u for each new edge processed.

A immediate implementation is to pre-build and maintain hash tables for every vertices in the graph. Note that although the time complexity of lookup operation in existing hashing techniques is $O(1)$, only the hashing technique *without collision* (e.g., bitmap hashing) can check the existence of an element with *exactly one* look-up operation. As mentioned in [12] and verified in the experiments, there is no clear advantage to apply the traditional hashing techniques (with collisions) to support the edge-oriented triangle listing algorithms, compared to the sort-merge based approaches. Moreover, more space is required if we pre-build hash tables for all vertices.

This motivates us to apply the time-efficient hashing technique: bitmap hashing. However, the bitmap hashing technique needs n bits where n is the number of vertices, and we cannot afford to pre-build bitmap hash tables for all vertices or build the tables on the fly for every pivot edge processed. To avoid this, we can amortize the cost of hash table constructions by instead building a bitmap hash table once for each pivot vertex u when it is processed, and looking up the neighbors of the other vertex v for each pivot edge with exact one look-up operation. However, where previously possible if we build a hash-table for each edge, we lose the ability to ensure that $deg(u) \geq deg(v)$ is true for each pivot edge (u, v) with pivot vertex u. The current algorithm, DPTL, cannot address this problem.

To tackle this problem, we are motivated to design an new triangle listing strategy to take advantage of the bitmap hash technique. Since we have already set that $deg(u) \geq deg(v)$ is true for each directed edge $u \rightarrow v$ due to our usage of the orientation technique, which implies that we can only use edge $u \rightarrow v$ as the pivot edge for the pivot vertex u (resp. $u \leftarrow v$ as the pivot edge for the pivot vertex v). We will be able to ensure the processing cost for each pivot edge $u \rightarrow v$ or $v \rightarrow u$ is $\Theta(\min(deg(u), deg(v)))$, together with the hash table construction cost $\Theta(m)$ in total. Regarding the Algorithm 2, we already use the edge $u \rightarrow v$ as the pivot edge for pivot vertex u in Case 1 (Fig. 2(a)) and Case 2 (Fig. 2(c)). However, as shown in Fig. 2(b), we may use the edge $u \leftarrow v$ as the pivot edge when u is the pivot vertex. In the running example in Fig. 3, when v_7 is processed as the pivot vertex, the triangle (v_7, v_0, v_9) is identified by pivot edge $v_7 \leftarrow v_0$ with $deg(v_7) > deg(v_0)$. To alleviate this issue, we re-design the process strategy of the algorithm as illustrated in Fig. 4 where

Cases 1 and 3 are the same as DPTL illustrated in Fig. 2. Particularly, we further consider the Case 2 in two scenarios. For Case 2.1, we choose the vertex u with two out-going new edges as the pivot vertex and $u \to v$ as the pivot edge where v is the ending vertex of the original edge. For Case 2.2, we choose the vertex u with one out-going new edge and one out-going original edge as the pivot vertex, and the new edge $u \to v$ as the pivot edge. Now we always use $u \to v$ as the pivot edge under four cases when u acts as a pivot vertex.

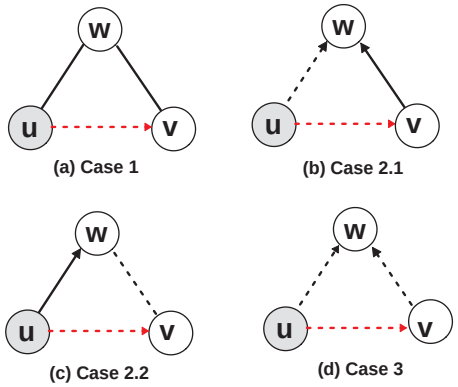

Fig. 4. Orientation technique for DPTL+

(2) Description of the Algorithm (DPTL+). DPTL+ considers four types of triangles shown in Cases 1, 2.1, 2.2 and 3 of Fig. 4. We note in advance the similarities of the triangle types to those considered by DPTL, where the definition for triangles with three update-edges and that with one update-edge remain the same.

A pseudo-code is shown in Algorithm 3, where DPTL+ also iterates through vertices found in B at Line 1, as was the case in DPTL. For each pivot vertex u processed, a bitmap hash table H is also initialized based on its neighbors in $N(u)$ at the start. Particularly, for each out-going edge $u \to v$, H_v is set to $\oplus$ (Lines 3-4). For each in-coming edge $u \leftarrow v$, H_v is set to $\ominus$ (Lines 5-7). For each pivot vertex u, all triangle patterns in Case 1 are listed in Lines 11 though 14. Note that the look-up operations $H_w = \oplus$ or $H_w = \ominus$ for $w \in N_{\vec{G}}(v)$ and $v \in N_B^+(u)$ implies that $w \in N_{\vec{G}}(u) \cap N_{\vec{G}}(v)$ where $u \to v$ is the pivot edge. Similarly, all triangle patterns in Case 2.1 are listed in Lines 16 through 19; all triangle patterns in Case 2.2 are listed in Lines 21 through 24, and all triangle patterns in Cases 3 are listed in Lines 26 through 29.

***Example** 2:* With respect to the running example in Fig. 3, we show how triangles in Case 2.1 and 2.2 are uniquely identified. Note that in DPTL, triangle (v_0, v_7, v_9) is reported with v_7 as its pivot vertex and $v_7 \leftarrow v_0$ as its pivot edge. In this new case for DPTL+, (v_0, v_7, v_9) now corresponds to the Case 2.2 with v_0 as pivot vertex and $v_0 \to v_7$ as the pivot edge. Same for the triangles (v_0, v_5, v_6) and (v_0, v_5, v_4). Triangle (v_0, v_1, v_7) corresponds to Case 2.1, which is reported with pivot vertex v_0 and pivot edge $v_0 \to v_7$ in both DPTL and DPTL+.

(3) Space Complexity. In addition to the influenced graph G, we only need to maintain a hash table H per thread for its active pivot vertex. Since a vertex will be only processed once as the role of pivot vertex, the space time complexity

Algorithm 3: DPTL+

```
Input  : G⃗: the oriented influenced graph after batch-update
Input  : B: new edges in batch-update
Output : new triangles resulting from batch-update
1  for u ∈ V(B) in parallel do
2      H ← ∅;
3      for v ∈ N⁺_G⃗(v) do
4          H_v ← ⊕;
5      for v ∈ N_G⃗(v) do
6          if H_v ≠ ⊕ then
7              H_v ← ⊖;
8      for v ∈ N⁺_B(v) do
9          H_v ← ⊎;
10     /*CASE 1*/
11     for v ∈ N⁺_B(u) do
12         for w ∈ N_G⃗(v) do
13             if H_w = ⊕ or H_w = ⊖ then
14                 print (u, v, w);
15     /*CASE 2.1*/
16     for v ∈ N⁺_B(u) do
17         for w ∈ N⁺_G⃗(v) do
18             if H_w = ⊎ then
19                 print (u, v, w);
20     /*CASE 2.2*/
21     for v ∈ N⁺_B(u) do
22         for w ∈ N_B(v) do
23             if H_w = ⊕ then
24                 print (u, v, w);
25     /*CASE 3*/
26     for v ∈ N⁺_B(u) do
27         for w ∈ N⁺_B(v) do
28             if H_w = ⊎ then
29                 print (u, v, w);
```

of DPTL+ is $O(pm)$ where m is the number of edges in the influenced graph G, and p is the number of threads at running time.

(4) Time Complexity. It takes $\Theta(deg(u))$ time to initialize the hash table for each pivot vertex u. The total cost spent on hash table initialization is $\Theta(m)$ time where m is the number of edges in the influenced graph G. For each pivot edge processed at Lines 12, 17 ,22 and 27, the cost is $\Theta(\min\{deg(u), deg(v)\})$ because we have $deg(u) \geq deg(v)$ under all cases. All in all, the time complexity of the DPTL+ (Algorithm 3) is $\Theta(\sum_{\langle u,v\rangle \in B} \min\{deg(u), deg(v)\} + m)$ which is more competitive than that of DPTL (Algorithm 2).

Same as DPTL, DPTL+ only outputs each updated triangle once, and there is no resource competition between threads, the parallel implementation of Algorithm 3 is straightforward where the pivot vertices at Line 1 can be processed in parallel.

(5) Correctness Analysis. Please refer to Section IX-B for the correctness analysis of Algorithm 3.

IV. Insertion/Deletion Mixed Batch-Dynamic Graph

In this section, we discuss the final algorithm for triangle listing on batch-dynamic graphs, where both inserted and

deleted edges are considered. We use G_1 and G_2 to denote the original graph and the graph after batch-update B. By Δ_{G_1} and Δ_{G_2}, we denote the set of triangles in G_1 and G_2, respectively.

According to the problem definition, we are only interested in the updated triangles after the batch-update, and we do not need to care about the order in which the updated triangles are output. Thus, the final algorithm consists of two phases: (1) oriented original graph $\vec{G_1}$ and the batch of deleted edges B^- as input of DPTL or DPTL+; and (2) oriented updated graph $\vec{G_2}$ and the batch of inserted edges B^+ as the input of DPTL or DPTL+. The deleted triangles and new triangles resulting from the batch-update will be output in phases (1) and (2), respectively.

As there is no mutual edge between the inserted edges B^+ and deleted edges B^-, the updated graph consists of three types of edges: original edges, inserted edges, and deleted edges (if we do not explicitly remove the deleted edges). For each deleted triangle pattern, there exists at least one deleted edge from B^- and none of the inserted edges from B^+ [1] since we consider batch deletion first. Thus, all deleted triangles (i.e.,$\Delta_{G_1} - \Delta_{G_2}$) can be reported in Phase (1). Similarly, for each new triangle pattern, there exists at least one new edge from B^+ and none of the deleted edges from B^-[2]. Since all deleted edges are removed in G_2 after the computation in Phase (1), and all new edges are included in G_2; therefore G_2 contains only original edges and inserted edges, all new triangles (i.e.,$\Delta_{G_2} - \Delta_{G_1}$) can be output in Phase (2).

V. Experimental Setup

Algorithms. We compare our proposed algorithm with the following methods discussed in the paper. In total, there are 4 methods in this experimental setting.

- **Naive** (Algorithm 1 in Section III-A) Serial version that lists updated triangles one update-edge at a time. As discussed in Section III-A, the parallel version of the naive algorithm is much slower than the serial version with single thread under our experiment setting, this is due to its expensive cost in removing duplicate solutions.
- **AOT*** State-of-the-art triangle listing algorithm [9] applied on the influenced graph $\vec{G}$, as discussed in Section III-A.
- **DPTL** (Algorithm 2 in Section III-B) Our first algorithm using a new orientation technique.
- **DPTL+** (Algorithm 3 in Section III-C) DPTL with hash table and degree-ordering based lookup optimization.

Datasets. The dataset we use in this experiment includes a number of real-world sample graphs including some billion-scale graphs. Some sample graph data are neither simple nor undirected, and they are treated and processed as simple undirected graphs for the purpose of this experiment. Each directional edge is considered as an undirected edge, multi-edges and loops are also ignored. The final list of datasets used in this paper is in Table 2, where *uk-2005* is used as the default graph in the experiments.

[1]Otherwise, the triangle cannot be included Δ_{G_1}

[2]Otherwise, it cannot be included in Δ_{G_2}

Workload. To evaluate the algorithms under the batch-dynamic setting, we first perform a batched deletion followed by a subsequent batched insertion, and return the amount of total time spent. Particularly, we randomly extract edges from the graph to use as new edges and deleted edges for the batch-update, the graph consisting of the remaining edges is used as the original graph. The set of edges involved in both edge insertion and deletion batches is controlled based on the batch size, which is the percentage of a given graph's edge count and varies from 0.01% to 8% with default value 4%.

Performance Measurement. In the experiments, we record the running time of the algorithms to evaluate their performance, which consists of graph batch-update time and the updated triangle listing time. Test instances that do not terminate within an hour (or 3600 seconds) are not included in the final figure. Note that we do not evaluate the space consumption because all 4 algorithms are space efficient.

Experimental Settings. In the experiments, all programs are implemented in c++ and compiled with g++-9. Regarding the batch-update of the graph, we update the adjacency lists of all vertices in parallel. In the implementation, we use static OpenMP parallel schedule compiling option. The source code for the AOT* algorithm is obtained by the authors in [9]. All experiments are performed on a 64 bit Linux machine with an Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz, the L1, L2 and L3 cache of 32K, 1024K, and 25344K respectively, with a RAM size of up to 376GB. The number of threads chosen for parallel execution varies from 1 to 32 with 8 threads as default.

A. Performance Evaluation

1) Effect of Diff. graphs: 28 datasets are deployed for the purposes of the experiment, and the running time of 4 algorithms is reported in Fig. 5 with default batch size (4%) and number of threads (8). We divide the results into two plots based on their edge-count for clarity, where the graphs with larger number of edges are evaluated in the second plot. In terms of total processing time, our DPTL+ algorithm performs the best: we find that its running time is consistently the least among the four methods tested. When comparing DPTL+ with its base design DPTL, the addition of the second technique does show an improvement in terms of the running-time. We are able to observe instances where the difference in running-time is less prominent, such as *soc-lastfm*, *lijournal-2008* and *friendster*. Although DPTL+ and DPTL consider the same set of update-edges during the iteration, DPTL+ does perform better than DPTL overall. While both DPTL and DPTL+ make use of the orientation technique, the DPTL algorithm does not make use of the additional degree distribution information which contributes towards a reduction in the overall intersection costs when considering the degree difference of edge endpoints. For the baseline algorithms AOT*, our algorithm DPTL performs faster than AOT* for most graphs. As reported in Fig. 5, the performance of the naive algorithm is not competitive under most of the settings, especially the large graphs.

2) Effect of Batch-Size: Where the thread count is fixed to the default of 8 threads, we measure the performance of all

TABLE II
STATISTICS OF 28 DATASETS.

Graph	#Nodes (M)	#Edges (M)	Max Degree	Avg Degree
soc-lastfm	1.19	9.04	5150	7
soc-digg	0.77	11.81	17643	15
youtube	3.22	18.75	91751	5
skitter	1.7	22.19	35455	13
higgs-twitter	0.46	25.02	51386	54
dbpedia	3.97	25.22	469692	6
web-hudong	1.98	28.86	61440	14
actor	0.38	30.08	3956	78
uk-2014-tpd	1.77	30.57	63731	17
flicker	1.62	30.95	27236	19
petster	0.62	31.39	80636	50
web-baidu-baike	2.14	34.03	97848	15
wiki-topcats	1.79	50.89	238342	28
sx-stackoverflow	6.02	56.37	44065	9
uk-2014-host	4.77	80.43	726244	16
ljournal-2008	5.36	99.03	19432	18
soc-orkut	3	212.7	27466	70
hollywood-2011	2.18	228.99	13107	105
indochina-2004	7.41	301.97	256425	40
wikipedia-link-en	12.15	576.52	962969	47
arabic-2005	22.74	1107.81	575628	48
uk-2005	39.46	1566.05	1776858	39
webbase-2001	118.14	1709.62	816127	14
it-2004	41.29	2054.95	1326744	49
twitter-2010	41.65	2405.03	2997487	57
friendster	124.84	3612.13	5214	28
sk-2005	50.64	3620.13	8563816	71
uk-2006-05	77.74	5271.7	4070242	67

four algorithms given batch sizes ranging from 0.01% to 8% of the graph being processed. We first report the number of modified triangles regarding different batch-size (percentage) on 6 graphs in Fig. 6. It is shown that the number of modified triangles is considerable large even with a small batch-size. For instance, in graph *sk-2005*, there are around 36 million and 411 million modified triangles for batch-size of 0.01 percent and 0.1 percent, respectively.

In Fig. 7, we demonstrate the processing time of the algorithms on four graphs. It is shown that the performance of DPTL+ consistently outperforms other competitors under all settings. With the growth of batch size, the margin between DPTL+ and DPTL becomes more significant, which implies that benefit of exploiting degree distribution is obvious on larger influenced graphs. We observe that the AOT* algorithm is not very sensitive to changes in batch size, this is because the size growth of influenced graph is slow on the batch-size. For example, the size of an *influenced graph* of *sk-2005* for batches size of 0.01 percent and 8 percent are 1,082 million and 1,717 million respectively. As DPTL can take immediate advantage of the small number of updated edges, it significantly outperforms AOT* when the batch-size is small.

3) Effect of the number of threads: Our existing evaluation of the discussed algorithms are based on a thread setting of 8 threads, we assess here the degree of parallelism by considering its execution on multiple thread counts: from single threaded operation up to 32 threads. As discussed previously, we reiterate that for the naive algorithm, the plots only depict its single threaded running time for all points along the x axis. Cases that exceed a running time of 3600 seconds are left out of the plot.

In Fig. 8, we decompose the total execution time of each algorithm into graph update time and triangle listing time, where all algorithms have the same graph update time. Here, we slightly abuse the notation by using the algorithm name to represent its triangle listing time. As expected, the performance DPTL+ is superior to other algorithms under all settings. With reference to Fig. 8, for larger datasets, such as *uk-2005* and *it-2004*, DPTL is not as fast as AOT* for lower thread settings. Once DPTL reaches a parallel execution of around 8 threads, it is able to overtake AOT* as the more efficient algorithm.

In terms of parallelization speed-up of triangle listing time, DPTL+, DPTL and AOT* demonstrate a good performance. It is reported that the speed-up of the *triangle listing time* of DPTL+ with 32 threads is 21, 10, 16, 11 on graphs uk-2014-host, uk-2005, twitter-2010, sk-2005 respectively, compared to DPTL+ with single thread. The corresponding speed-up of DPTL (resp. AOT*) is 28 (6), 19 (2), 20 (7), 21 (4), respectively.

We noticed that graph update, although very fast, still takes a certain portion of the total execution time if the triangle listing is fast. Moreover, compared to triangle listing, it is difficult to achieve a good parallel speed-up for graph update because of a large number of write operations involved. These factors make the parallel speed-up of the total execution time not very impressive when the triangle listing time is very fast (e.g., Fig 8(d)). Nevertheless, in terms of total processing time,

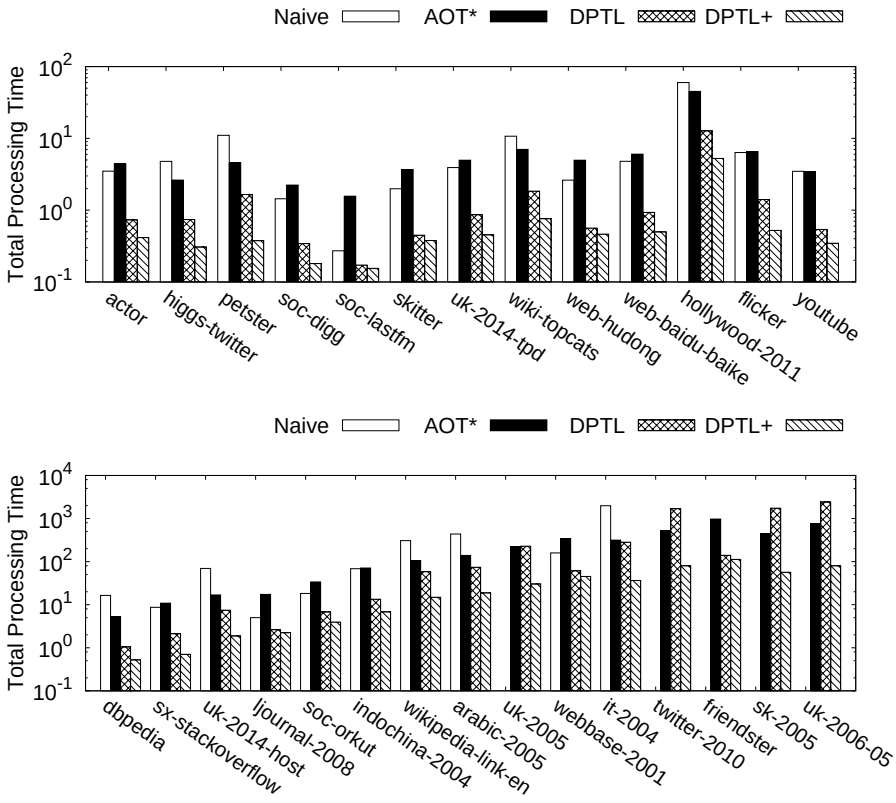

Fig. 5. Performance Analysis

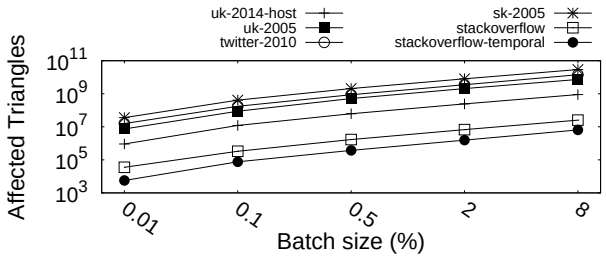

Fig. 6. #modified triangles on diff. batch-size

Fig. 8 shows that DPTL+ with 32 threads can still achieve a speed-up of 18, 6, 12, 7 on graphs uk-2014-host, uk-2005, twitter-2010, sk-2005 respectively, compared to DPTL+ with single thread. The corresponding speed-up of DPTL (resp. AOT*) is 25 (6), 17 (2), 18 (7), 16 (4), respectively.

As shown in Fig. 8 of the revision, the speed-up of the *processing time* of DPTL+ with 32 threads is 18, 6, 12, 7 on graphs uk-2014-host, uk-2005, twitter-2010, sk-2005 respectively, compared to DPTL+ with single thread. The corresponding speed-up of DPTL (resp. AOT*) is 25 (6), 17 (2), 18 (7), 16 (4), respectively.

B. Additional Experiments

In this subsection, we provide additional experiments for better understanding of the proposed methods.

1) Evaluate Diff. hash schemes: In Fig. 9, we use the standard C++ hash table implementation in STL (Standard Template Library) for the neighborhood intersection computation of DPTL, namely DPTL-h1, and pre-build a hash table

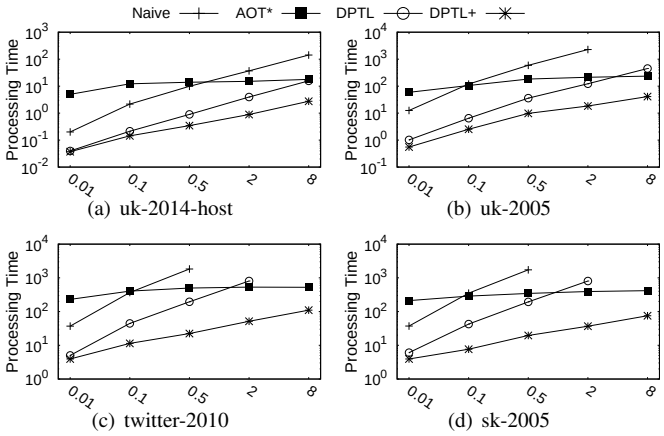

Fig. 7. Processing time w.r.t diff. batch-size

for every vertex. Recall that we use sort-merge and bitmap hashing for the intersection computation in DPTL and DPTL+, respectively. In addition to the large memory required, DPTL-h1 does not show superior performance in terms of processing time on uk-2005 graph compared to DPTL, while our proposed DPTL+ significantly outperforms DPTL under all settings.

2) Justification of the baseline: To justify that AOT* is a reasonable baseline, two representative main memory parallel triangle listing algorithms, CF [8] and KClist [13], replace AOT for listing triangles on the influenced graph. As expected, considering that they are not state-of-the-art, Fig. 10 shows that their corresponding algorithms, namely CF* and KClist*, are

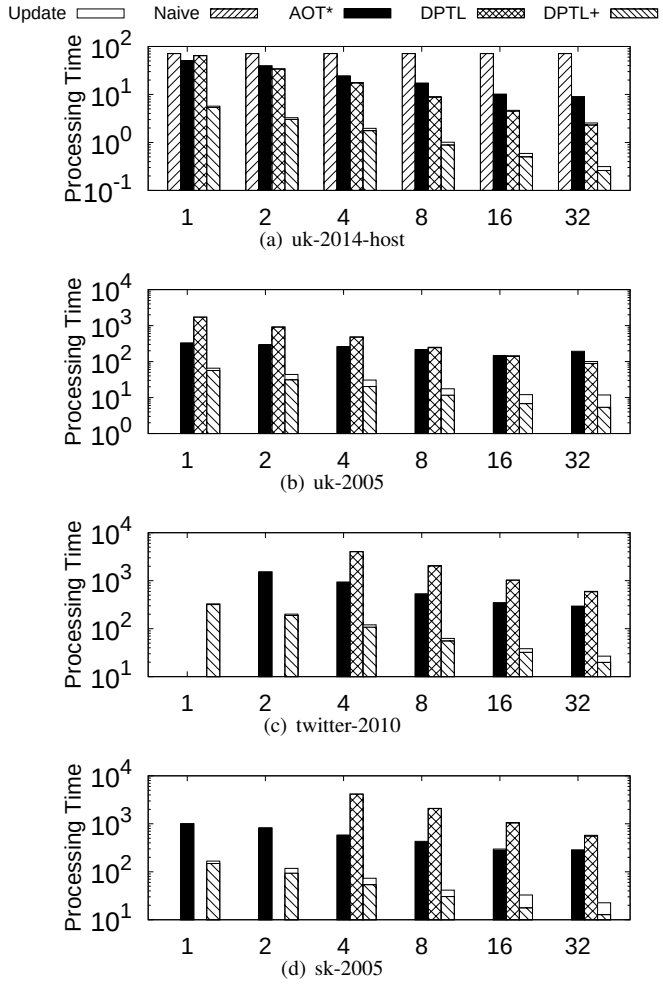

Fig. 8. Evaluating Parallel Performance

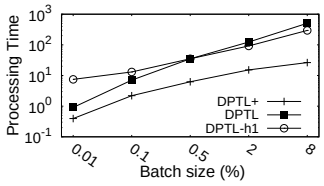

Fig. 9. Diff. hash schemes

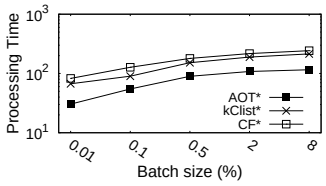

Fig. 10. Evaluate baselines

clearly outperformed by the AOT*.

3) Evaluate of batch selection: In the above experiments, we choose a batch of updates by randomly selecting edges from a graph. We notice that there is a timestamp on edges of the graph *stackoverflow*. To evaluate the effect of the batch selection methods, we compare the performance of the algorithms in Fig. 11(a) and Fig. 11(b), where random selection and time-based selection method are used respectively. Though the running time of each algorithm is different on two batch selection methods, we observe the same performance rankings and similar trends of the algorithms in two figures. Note that we also report the number of modified triangles for these two approaches in Fig. 6, denoted by *stackoverflow* and *stackoverflow-temporal*, respectively. It is shown that time based batch selection method (i.e., *stackoverflow-temporal*) has less number of modified triangles compared to the sampling based one (i.e., *stackoverflow*) under same batch size, but they have similar growth rate when the batch-size increase.

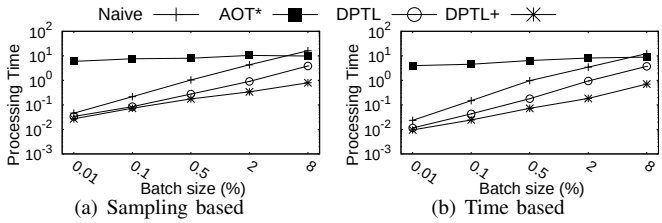

Fig. 11. Evaluating batch selection methods

VI. Related Work

In this section, we introduce existing works closely related to our problem studied.

A. Triangle Listing and Counting

As a special subgraph, the triangle is a fundamental structure that is the building block for subgraph patterns that are more complex, it is also widely used in many real-life applications. In particular, in-memory algorithms have been extensively studied in the literature (e.g.,[8], [13], [10], [11]). Triangle listing can be classified into two popular computing paradigms which are the edge-iterator [14] and the node-iterator [15]; both paradigms share a common asymptotic behavior [11]. Since then, triangle listing solutions have mostly been improvements and optimizations based on these two early paradigms. In more recent years, the topics of interest have shifted to parallel/distributed processing (e.g.,[12], [16]), efficient I/O external memory methods (e.g.,[17], [18]), and the asymptotic cost analysis of triangle listing in random graphs [19].

The triangle counting is a problem related to the triangle listing problem, which find the total number of triangles in a graph G. Compared to listing algorithms, counting algorithms find ways to compute the number without relying on the exploration of triangle instances. Many algorithms have been designed to count triangles (e.g., [20], [21], [22]). Approximate methods are useful for settings that handle large-scale graphs, or settings where a given approximation is as useful as knowing the exact triangle count (e.g., [23], [24], [25]).

B. Batch-update Algorithms on Dynamic Graphs

The are advantages of processing updates in batches when dealing with high volumes of changing data. Various algorithms that consider batche-updates have been investigated such as computing clustering coefficients [26], single-source shortest-path [27], dynamic connectivity problems [7] and those with GPU support [28]. Recently, the problem of computing the exact triangle count for batch-dynamic graph has been studied in [25]. However, their methods cannot be applied to listing triangle solutions as they do not need to explicitly list each triangle, and the duplication can be simply avoided by inclusive-exclusive principle.

C. Subgraph Enumeration on Dynamic Graphs

As a generalization of triangle listing algorithm, the problem of subgraph enumeration has been intensively studied in the

literature, which aims to find all occurrence of a query subgraph g in a graph G. In recent years, some incremental solutions (e.g., [29], [30], [31], [32], [33]) have been proposed to efficiently detect updated subgraph upon the updates of the graph. Though their techniques can be immediately used for the problem of updated triangle listing by simply setting the query subgraph as a triangle, this is not promising because (1) unlike existing triangle listing algorithms, they are not specifically designed for the triangle; and (2) their optimization techniques cannot directly benefit the triangle listing. For instance, one of the commonly used approaches is the partial result indexing. However, we may come up with an updated triangle for *any* two connected edges (i.e., wedges), and it is infeasible to index all of them.

VII. Discussion

A. Justification of the Baseline Algorithms

As shown in Section VI-A, triangle listing is a famous problem and there are many existing works under different settings. Clearly, all existing main memory triangle listing algorithms can be immediately applied on the influenced graph to enumerate updated triangles. We may also consider other studies under other settings such as external memory (e.g., [17]) or distributed (e.g., [16]) computing environments. For instance, the graph is partitioned into different groups in [17] such that each partition can be fit into the main memory. After computing the triangles in each individual group, they need to compute the triangles across different groups. To compute the triangles across two groups A and B, we may regard this as the problem of triangle listing in a batch-update setting; that is, A is the original graph and B is the batch-updates. However, this is not the research focus of [17], and the existing main memory triangle listing algorithm is directly used on the influenced subgraph. Similarly, no special optimization is considered on this aspect in other studies. Thus, as to our best knowledge, a reasonable baseline is to apply the state-of-the-art main memory triangle listing algorithms on the influenced graph, which is confirm by Fig. 10 in the experiments.

B. Compared to General Graph Matching Algorithms

As discussed in Section VI-C, it is infeasible to directly apply the general dynamic graph matching algorithms for the problem of triangle listing on batch-updated graphs because they are not specifically designed for triangle listing. One natural question is that if we can extended our proposed approach to enhance their performance. Same as existing triangle listing algorithms, the core of the proposed techniques in our paper is specific to the triangle structure, and hence cannot be used for general subgraph processing framework. For instance, the key of the orientation technique designed for DPTL is to use an updated edge as pivot edge to uniquely identify a triangle such that we output each triangle once, but we cannot do this on larger subgraphs. Similarly, for larger graph pattern multiple pivot edges will be involved in the neighborhood intersection computation, we cannot ensure $deg(u) \geq deg(v)$ for every pivot edge $u \rightarrow v$. Thus, we cannot guarantee that time complexity of $\Theta(\min\{deg(u), deg(v)\})$ for each edge $u \rightarrow v$.

VIII. Conclusion

The triangle listing is a fundamental problem in graph analysis with a wide range of applications. Though this problem has been intensively studied on static graphs, this paper is the first to investigate the problem in the context of batch-dynamic graphs with batch edge insertion and deletion operations. An efficient parallel algorithm DPTL+ has been developed with the best theoretical time complexity and practical performance compared to other solutions.

References

[1] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips, "Tolerating the community detection resolution limit with edge weighting," *Physical Review E*, vol. 83, no. 5, 2011.

[2] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, "Defining and identifying communities in networks," *PNAS*, vol. 101, no. 9, 2004.

[3] B.-H. Chou and E. Suzuki, "Discovering community-oriented roles of nodes in a social network," in *Proc. of DaWaK'10*, 2010.

[4] X. Xu, N. Yuruk, Z. Feng, and T. A. Schweiger, "Scan: a structural clustering algorithm for networks," in *Proc. of SIGKDD'07*, 2007.

[5] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proc. of SIGKDD'07*, 2017.

[6] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient algorithms for large-scale local triangle counting," *TKDD*, vol. 4, no. 3, 2010.

[7] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala, "Parallel batch-dynamic graph connectivity," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 381–392.

[8] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, vol. 407, no. 1-3, 2008.

[9] M. Yu, L. Qin, Y. Zhang, W. Zhang, and X. Lin, "Aot: Pushing the efficiency boundary of main-memory triangle listing," 2020, accepted by DASFFA 2020.

[10] M. Ortmann and U. Brandes, "Triangle listing algorithms: Back from the diversion," in *Proc. of the Meeting on Algorithm Engineering & Expermiments*, 2014.

[11] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *International workshop on experimental and efficient algorithms*. Springer, 2005.

[12] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Proc. of ICDE'15*, 2015.

[13] M. Danisch, O. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs," in *Proc. of WWW'18*, 2018.

[14] V. Batagelj and A. Mrvar, "A subquadratic triad census algorithm for large sparse networks with small maximum degree," *Social Networks*, vol. 23, no. 3, 2001.

[15] A. Itai and M. Rodeh, "Finding a minimum circuit in a graph," *SIAM Journal on Computing*, vol. 7, no. 4, 1978.

[16] H.-M. Park, S.-H. Myaeng, and U. Kang, "Pte: Enumerating trillion triangles on distributed systems," in *Proc. of SIGKDD'16*, 2016.

[17] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proc. of KDD'11*, 2011.

[18] X. Hu, Y. Tao, and C. Chung, "I/o-efficient algorithms on triangle listing and counting," *ACM Trans. Database Syst.*, vol. 39, no. 4, 2014.

[19] D. Xiao, Y. Cui, D. B. Cline, and D. Loguinov, "On asymptotic cost of triangle listing in random graphs," in *PODS*. ACM, 2017, pp. 261–272.

[20] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proc. of SIGKDD'08*, 2008.

[21] A. Pavan, K. Tangwongsan, S. Tirthapura, and K. Wu, "Counting and sampling triangles from a graph stream," *PVLDB*, vol. 6, no. 14, 2013.

[22] T. G. Kolda, A. Pinar, T. D. Plantenga, C. Seshadhri, and C. Task, "Counting triangles in massive graphs with mapreduce," *SIAM J. Scientific Computing*, vol. 36, no. 5, 2014.

[23] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "DOULION: counting triangles in massive graphs with a coin," in *Proc. of SIGKDD'09*, 2009.

[24] D. Türkoglu and A. Turk, "Edge-based wedge sampling to estimate triangle counts in very large graphs," in *Proc. of ICDM'17*, 2017.

[25] D. Makkar, D. A. Bader, and O. Green, "Exact and parallel triangle counting in dynamic graphs," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 2017, pp. 2–12.
[26] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.
[27] R. Bauer and D. Wagner, "Batch dynamic single-source shortest-path algorithms: An experimental study," in *International Symposium on Experimental Algorithms*. Springer, 2009, pp. 51–62.
[28] O. Green and D. A. Bader, "custinger: Supporting dynamic graph algorithms for gpus," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–6.
[29] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu, "Incremental graph pattern matching," in *ACM SIGMOD*, 2011, pp. 925–936.
[30] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. S. Young, M. Wolf, and K. Schwan, "Graphin: An online high performance incremental graph processing framework," in *Euro-Par*, vol. 9833, 2016, pp. 319–333.
[31] L. Chen and C. Wang, "Continuous subgraph pattern search over certain and uncertain graph streams," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 8, pp. 1093–1109, 2010.
[32] Y. Li, L. Zou, M. T. Özsu, and D. Zhao, "Time constrained continuous subgraph search over streaming graphs," in *ICDE*, 2019, pp. 1082–1093.
[33] K. Kim, I. Seo, W. Han, J. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *SIGMOD*, 2018.

IX. Appendix

A. Correctness of DPTL

We show that the Algorithm 2 correctly outputs all updated triangles in the sense that (1) all updated triangles are reported, and none of the existing triangles are encountered in the computation; and (2) each updated triangle is reported exactly once.

Given a triangle (x, y, w) in the graph G, there are a total of $4 \times 4 \times 4 = 64$ possible patterns considering the edge directions and types (new or existing edges). Among them, the cyclic triangles (clockwise or anti clockwise) will be excluded, with a total number of $2 \times 2 \times 2 \times 2 = 16$. The triangles without new edge will be excluded as well with a total number of $2 \times 2 \times 2 = 8$. Considering there are 2 cyclic triangle patterns without any new edge, the number of triangle patterns in the output of the Algorithm 2 is $64 - 16 - 8 + 2 = 42$ following the inclusionexclusion principle.

In the following, we show that these 42 patterns will be uniquely reported in Algorithm 2. Let's start with Case 1, and assume (x, y) is the new edge. As shown in Fig. 12, there are 6 valid patterns. Patterns in Fig. 12(a)-(c) will only be output when x is the pivot vertex and $x \to y$ is the pivot edge. Patterns in in Fig. 12(d)-(f) are associated with pivot vertex y and pivot edge $y \to x$. Other patterns with new edge (x, z) or (y, z) can be detected in the same way. Note that there is no cyclic triangle due to the use of orientation technique and the output triangle has at least one new edge (i.e., exiting triangle will not be reported). Thus, 18 triangle patterns in Case 1 can be correctly output.

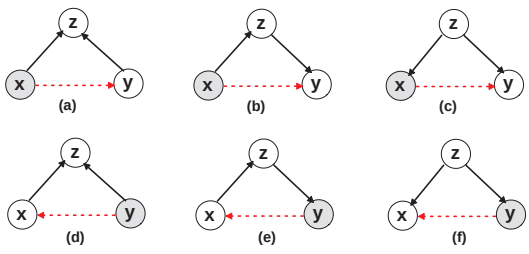

Fig. 12. Case 1 for DPTL

Regarding the Case 2 with two new edges, we assume (y, z) is the only original edge. As shown in Fig. 13, there are 6 valid patterns, and x is the pivot vertex. While the choice of pivot edge is different with $x \to y$ for Fig. 13(a), $y \to x$ for Fig. 13(b) and (c), $x \to z$ for Fig. 13(d), $z \to x$ for Fig. 13(e) and (f). With similar rationale to Case 1, another 18 triangle patterns in Case 2 can be reported correctly.

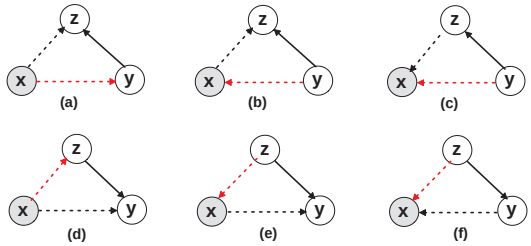

Fig. 13. Case 2 for DPTL

Regarding the Case 3 with three new edges, and we assume x is the vertex with two out-going edges. There are two valid triangle patterns as shown in Fig. 14 where x is the pivot vertex, and $x \to y$ is the pivot edge in Fig. 14(a) and $x \to z$ is the pivot edge in Fig. 14(b). With similar rationale to Cases 1 and 2, 6 valid triangle patterns in Case 3 can be reported correctly. Based on the above analysis, we can see all $18 + 18 + 6 = 42$ valid triangle patterns can be uniquely identified by their corresponding pivot vertices and pivot edges. So the correctness of Algorithm 2 follows.

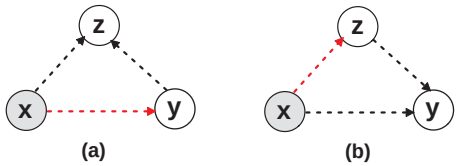

Fig. 14. Case 3 for DPTL

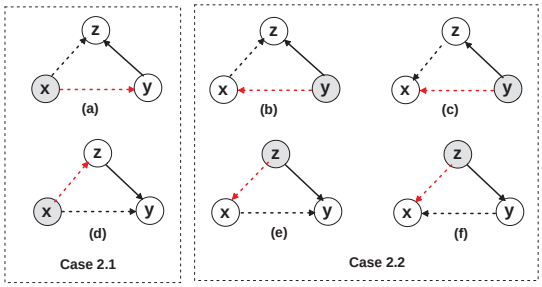

Fig. 15. Case 2.1 and 2.2 for DPTL+

B. Correctness of DPTL+

Since the Case 1 and 3 of the DPTL+ are the same as those of DPTL, we limit the analysis on the Case 2 where an updated triangle contains two new edges. Let edge (y, z) be the only original edge, compared to Fig. 13 (Case 2 of DPTL), the choice of some pivot vertices and pivot edges have been changed in DPTL+ as shown in Fig. 15. Particularly, the pivot vertex (x) and pivot edges ($x \to y$ and $x \to z$) remain the same in Fig. 15(a) and (d), which corresponds to the Case 2.1 in DPTL+. While other 4 triangle patterns fall in the Case 2.2 of DPTL+, and now vertex y becomes the pivot vertex with pivot edge $y \to x$ in Fig. 15(b) and (c), and the vertex z becomes the pivot vertex with pivot edge $z \to x$ in Fig. 15(e) and (f). Similarly, we can correctly find triangle patterns with two new edges where (x, y) or (x, z) is the original edge. So the 18 triangle patterns in Case 2 can be reported correctly.

Based on the above analysis and the correctness of DPTL, DPTL+ uniquely reports all 42 valid triangle patterns with their corresponding pivot vertices and pivot edges.