Parallel Edge-based Sampling for Static and Dynamic Graphs

Kartik Lakhotia[†], Rajgopal Kannan[‡], Aditya Gaur[†], Ajitesh Srivastava[†], Viktor Prasanna[†]

{[†]University of Southern California, [‡]Army Research Lab-West }
Los Angeles, USA
{klakhoti,adityaga,ajiteshs,prasanna}@usc.edu,rajgopal.kannan.civ@mail.Mil

ABSTRACT

Graph sampling is an important tool to obtain small and manageable subgraphs from large real-world graphs. Prior research has shown that Induced Edge Sampling (IES) outperforms other sampling methods in terms of the quality of subgraph obtained. Even though fast sampling is crucial for several workflows, there has been little work on parallel sampling algorithms in the past.

In this paper, we present parIES - a framework for parallel Induced Edge Sampling on shared-memory parallel machines. parIES, equipped with optimized load balancing and synchronization avoiding strategies, can sample both static and streaming dynamic graphs, while achieving high scalability and parallel efficiency. We develop a lightweight concurrent hash table coupled with a space-efficient dynamic graph data structure to overcome the challenges and memory constraints of sampling streaming dynamic graphs. We evaluate parIES on a 16-core (32 threads) Intel server using 7 large synthetic and real-world networks. From a static graph, parIES can sample a subgraph with > 1.4B edges in $< 2.5s$ and achieve upto $15.5\times$ parallel speedup. For dynamic streaming graphs, parIES can process upto 86.7M edges per second achieving $15\times$ parallel speedup.

CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**; **Vector / streaming algorithms**; *Concurrent algorithms.*

KEYWORDS

Parallel Graph Sampling, Streaming Graphs, Induced Edge Sampling, Dynamic Graph Data Structure, Big Data

ACM Reference Format:
Kartik Lakhotia[†], Rajgopal Kannan[‡], Aditya Gaur[†], Ajitesh Srivastava[†], Viktor Prasanna[†]. 2019. Parallel Edge-based Sampling for Static and Dynamic Graphs . In *Proceedings of the 16th conference on Computing Frontiers (CF '19), April 30-May 2, 2019, Alghero, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3310273.3323052

1 INTRODUCTION

Graph analysis forms the backbone for many applications in social media, WWW, bioinformatics and transportation that generate and process large real-world graphs. Often these graphs are too large to process on a single machine, however, despite the development of highly optimized distributed graph databases and processing frameworks [5, 12, 14, 19, 32, 34], many analytics algorithms cannot run efficiently and directly on these extremely large graphs. Thus approximate analytics on smaller, representative subgraphs have become a popular alternative [2, 25, 30, 31, 33]. Sampling is a widely used method for generating manageable subgraphs – many network repositories that provide datasets for analysis and benchmarking, store subgraphs sampled from real-world graphs[17, 18]. Fast representative graph sampling (i.e graph sampling done fast while simultaneously ensuring the sampled subgraph remains a *good* representative of the original) is therefore essential for the smooth processing of many crucial programs, especially within mission-critical workflows.

In spite of the numerous applications, there has been little work on *parallelizing* graph sampling. Sequential sampling of very large graphs can be unacceptably slow, especially in cases when a graph has to be repeatedly sampled or when the sampling throughput must match the incoming rate of a streaming graph. Existing works on parallel sampling are tailored towards application-specific algorithms that extract subgraphs conforming to the objective in the corresponding problem, for example, finding gene functionality clusters [7, 8]. Recently, an FPGA accelerator [28] was proposed for deletion sampling methods [16] that remove vertices and/or edges from the original graph to create a subgraph. However, the accelerator only caters to static in-memory graphs and takes $\approx 50s$ to sample a $150M$ edge graph, further emphasizing the need for efficient parallelization of graph sampling techniques.

In this paper, we focus on Induced Edge Sampling (IES) [3, 24]. Edge Sampling (ES) randomly samples edges, thus mitigating the downward degree bias of node sampling. Additionally, induction on the sampled nodes further improves the connectivity in the sampled subgraph G_s. IES has been shown to outperform many sophisticated state-of-the-art node sampling and topology based algorithms, such as forest fire or snowball sampling [3], in terms of preserving graph structure.

Parallelizing edge sampling is challenging as the parallel threads need to track and update a common goal in terms of target graph size. If implemented naively, this can limit the scalability of a parallel sampler. Furthermore, if the graph G is too large to fit in memory or dynamically changing with time (which is often the case with real-world graphs), a streaming algorithm must be applied that can work with strict memory space restrictions and perform sampling and induction in a single pass over the incoming stream of edges.

CF '19, April 30-May 2, 2019, Alghero, Italy
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6685-4/19/05…$15.00
https://doi.org/10.1145/3310273.3323052

Stream sampling is especially challenging to parallelize as the sampled vertex set and induced subgraph G_s need to be dynamically maintained, while processing the edge stream at a high rate.

To this purpose, we develop parIES - an efficient framework for parallel Induced Edge Sampling on shared-memory systems. parIES consists of 2 algorithms - parTIES and parPIES that effectively utilize the parallel computing resources provided by off the shelf multi-core architectures to sample static in-memory and streaming dynamic graphs, respectively. parTIES utilizes a checkpointing mechanism that allows all threads to continuously track the state of G_s without synchronizing. For parPIES, we further develop a light-weight hash table coupled with a space-efficient dynamic graph data structure, that concurrently and efficiently update the sampled subgraph G_s, while respecting the memory constraints on a streaming algorithm. To the best of our knowledge, this is the first work that comprehensively targets parallel edge-based sampling of *both* static and dynamic/streaming graphs, allowing billion scale graphs to be sampled in a few seconds.

The major contributions of our work are as follows:

(1) We develop an asynchronous scalable parallel Totally Induced Edge Sampling (parTIES) algorithm that efficiently executes Edge Sampling and Induction using optimized load balancing and synchronization avoidance mechanisms.
(2) To facilitate efficient sampling of streaming networks, we develop a dynamic graph data structure coupled with a parallel hash table that allows concurrent insertions, deletions, replace and search. Unlike conventional hash tables, our approach does not require rehashing, thereby ensuring scalable and highly parallel execution of updates.
(3) Using these data structures, we create parPIES - parallel Partially Induced Edge Sampling algorithm for dynamic graphs. parPIES generates the sampled subgraph in a single pass through the edge stream, while satsifying the memory constraints on a streaming algorithm.

2 SAMPLING STATIC GRAPHS

2.1 Background

Many applications such as graph sparsification, visualization and analytics require sampling a subgraph $G_s(V_s, E_s)$ from the larger graph $G(V, E)$. Edge sampling and vertex sampling are two very well-known sampling techniques [3, 24]. In many practical scenarios, the objective is to sample a subgraph comprised of a target number of vertices $|V_s|$ followed by inducing some or all of the edges in V_s (*partial* versus *total* induction). While nodes in an induced subgraph have inherently smaller degrees than the original, it has been shown that random edge sampling with induction improves the connectivity in G_s (by selecting high degree nodes with larger probability) and brings its distributions (degree, path length, clustering coefficient etc.) closer to G [3].

The sequential induced edge sampling algorithm given in [3] randomly selects edges from E until a desired number $k = |V_s|$ of distinct vertices in V have been sampled. This is followed by a *total induction* phase that iterates over all the edges in E for induction in V_s. Such algorithms assume the graph is *static* and small enough that both V and E are always accessible through calls to main memory.

Naive parallelization of these algorithms entails concurrent threads having to probe and update a shared collective target. Such heavy synchronization can limit scalability and even make parallel execution slower than sequential execution. Further, the induction phase requires $\Omega(E)$ work, making it very inefficient for the cases when k/V is very small.

2.2 Asynchronous parallel Totally Induced Edge Sampling (ASparTIES)

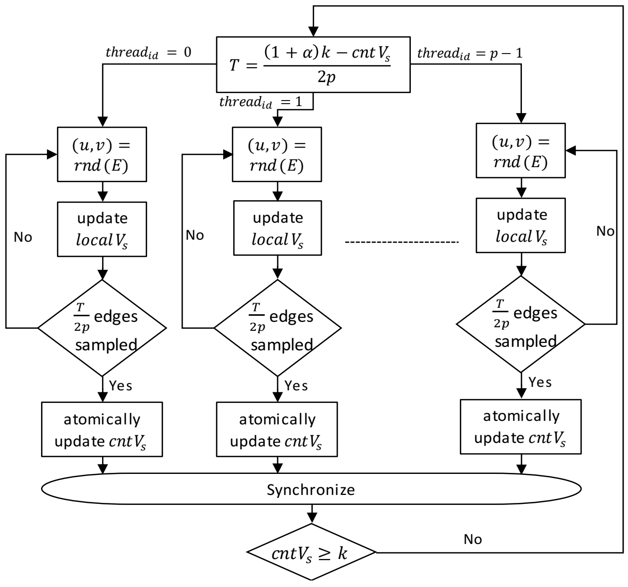

Figure 1: SparTIES: Synchronous Parallel Edge Sampling in parTIES using p threads

We develop an asynchronous, fast, efficient and load-balanced parallel algorithm for Totally Induced Edge Sampling (ASparTIES) for static graphs. Before describing this, we first outline a simple, *loosely synchronous* parallelization - SparTIES (Fig. 1). The key idea is to assign each thread a fixed amount of work in each iteration and iterate until $|V_s| \geq k$. Specifically, SparTIES restricts the amount of edges selected by a thread in an iteration to $\frac{(1+\alpha)k - cntV_s}{2p}$, where p denotes the number of concurrent threads and $cntV_s$ is the global sampled vertex size just before that iteration. α, $0 < \alpha \ll 1$ is an edge oversampling factor to drive sampling towards the vertex target. As shown in lemma 1, α does not have to be too large for reasonable target sizes on most graphs. Each thread maintains a private local array $localV_s$ of sampled vertices *unique* to it. During an iteration, for each sampled edge (u, v), a thread uses an atomic Compare-And-Swap (CAS) instruction to update both its $localV_s$ and a global boolean array of vertex status ($status[]$) if u and/or v are new samples. Threads only have to synchronize and update $cntV_s$ at the end of an iteration, thus limiting the amount of synchronization. When $cntV_s$ becomes greater than k, each thread copies its $localV_s$ into a global V_s array and proceeds to the induction phase. During induction, each thread is allocated $cntV_s/p$ sampled vertices to induce edges from. Similar to the sampling phase, threads use a private $localE_s$ array. Via the CSR matrix, only edges that are incident on at least one sampled node are inspected for induction. This makes our implementation efficient even for small k/V.

SparTIES has some drawbacks. For real-world graphs with power law degree distribution, the static task allocation can introduce load imbalance while the use of dynamic local vectors can slow down execution. Further, making α small to reduce oversampling can increase the amount of synchronization and limit scalability. Finally, the parallel efficiency of parTIES is also affected by the *type* of sampling. Intuitively, *sampling with replacement* (from E) has less overhead but requires more iterations to reach the sample target. Consider the fraction $\eta = |E_s|/|E|$ of edges that need to be sampled in order to obtain a subgraph of vertex size $|V_s| = \beta|V|$. Let $c_\eta = e^{-\eta}$. Let D_v be a random variable denoting the degree of a vertex v in G. Then it is straightforward to show the following (proof omitted for brevity):

Lemma 1. *(1) Sampling with replacement: The expected fraction of edges to be sampled to achieve a targeted fraction β of vertices is obtained by solving $\mathbb{E}[(c_\eta)^{D_v}] \approx 1 - \beta$. (2) Sampling without replacement: $\mathbb{E}[\eta] \leq \beta$.*

We resolve these drawbacks by developing ASparTIES: a fully asynchronous algorithm for static edge sampling with the following key design features: Each thread avoids conflicts by reserving work in rounds of *vertex chunks* (of size *BS*) through atomic updates of a global checkpoint on the size of V_s. A thread then samples random edges asynchronously until it obtains exactly *BS* vertex samples (after which it moves to its next round), or until it views $|V_s| \geq k$. Thus the algorithm samples exactly k vertices. A similar process is carried out for the induction phase. With *dynamic task allocation*, *in-place storage* and *conflict-free updates* of sampled vertices and induced edges, ASparTIES significantly improves load balance and avoids inefficiencies associated with dynamic private vectors.

Algorithm 1 shows the pseudocode of ASparTIES. The key function is *update* (lines 18-21), used by each thread t to reserve for itself an exclusive space of size *BS* in V_s. An atomic Fetch-and-add (FAA) on global counter $cntV_s$, executed once t has locally gathered and inserted *BS* number of new vertices into V_s (lines 4-10), initiates the next round for t. Lastly, the induction phase parallelizes tasks over vertices in V_s, following a similar process (lines 11-17).

Note that if $p \ll k$, all threads keep working until the last round, thus ensuring efficient *load balancing* in ASparTIES. Also note that the work done in the induction phase dominates the sampling phase as sampling only selects a subset of edges to be induced. Induction probes all edges incident on at least one vertex in V_s, thus expected work is $O(kd)$, where d is the average degree of V_s. Assuming that *status* array is initialized beforehand, the expected time complexity of ASparTIES is $O(\frac{kd}{p})$ if all operations are executed in parallel. However, on a CREW machine, multiple FAAs to the same variable are sequentialized. Choosing $BS = \theta(p)$ limits the number of rounds and consequently, the number of sequential steps incurred to $O(\frac{kd}{p})$. Thus, the scalability and asymptotic time complexity of the algorithm is not affected by the checkpointing.

Finally, ASparTIES implements sampling without replacement with low overheads ($O(1)$ work per edge). When the number of sampled edges is large, it allocates edge partitions to threads. Threads sample edges from their respective partitions and move sampled edges to the end of array to remove them from the edge set.

Algorithm 1 Asynchronous parTIES with dynamic scheduling

```
    ${V_s, E_s} = \phi$, ${cntV_s, cntE_s, \forall i \in V : status[i]} = 0$
1:  BS = $p \times const$
2:  for $tid = 1, 2...p$ do in parallel                    ▷ Edge Sampling
3:      $cnt_{local} = 0$,  $update(cnt_{local}, cntV_s, BS)$
4:      while $cnt_{local} < k$ do
5:          $(u, v) = rnd(E)$
6:          if $CAS(\&status[u], 0, 1)$ then
7:              $V_s[cnt_{local}] = u$
8:              $update(cnt_{local}, cntV_s, BS)$
9:          Execute lines 6-8 for vertex v
10:         $E = E - \{(u, v)\}$
11: for $u \in V_s$ do in parallel                           ▷ Induction
12:     $cnt_{local} = 0$
13:     $update(cnt_{local}, cntE_s, BS)$
14:     for each $(u, v) \in E$
15:         if $v \in V_s$ then
16:             $E_s[cnt_{local}] = (u, v)$
17:             $update(cnt_{local}, cntE_s, BS)$
18: function $update(\&cnt_{local}, \&cnt_{global}, BS)$
19:     if $(cnt_{local} = 0) \lor ((cnt_{local} + 1)\%BS = 0)$ then
20:         $cnt_{local} = FAA(\&cnt_{global}, BS)$
21:     else $cnt_{local} + +$;
```

Initialization cost can be amortized across multiple runs of ASparTIES. Resetting the *status* array requires $O(k)$ work and $O(\frac{k}{p})$ time. Resetting the end pointers of partitions to recover initial edge set requires $O(p)$ work and $O(1)$ time. The optimizations discussed in ASparTIES will also be useful in sampling dynamic streaming graphs as we will see in the next section.

3 SAMPLING DYNAMIC GRAPHS

3.1 Background & Challenges

Graphs representing real-world networks may often be extremely large to fit on the main memory of a machine. Furthermore, the graph could be dynamically evolving over time, requiring updates to be processed as they arrive and without visiting the same edge repeatedly. In such situations, the graph may only be accessible in a *streaming* fashion i.e. as a randomly ordered stream of edges, possibly coming from a disk. Algorithms that assume the ability to store and randomly access elements from V and E are unsuitable for (parallel) sampling (of) such streaming graphs. Rather, streaming algorithms that can process edges on the fly, performing *both* sampling and induction in a *single* pass over the edge stream are necessary.

Recently, a *sequential* streaming algorithm labeled Partially Induced Edge Sampling (PIES) was proposed in [3]. For completeness, we provide a brief description of PIES below. PIES samples from a stream of incoming edges and maintains a partially induced subgraph. Initially, PIES selects all edges and adds them to G_s as long as $|V_s| < k$, a given parameter. Subsequently, the t^{th} streamed edge $e_t = (u, v)$ is inducted into G_s if either 1) both u and v are already in V_s or 2) e_t gets selected with probability $\frac{m}{t}$, where $m = |E_s|$ at the end of the initial phase. This is shown to be equivalent to random edge sampling[3]. If e_t is selected and $u \notin V_s$ ($v \notin V_s$, resp.), then a random vertex $i \in V_s$ (j, resp.) is selected and u (v, resp.) replaces that vertex. Vertex i (j, resp.) is then deleted along with all its existing incident edges in E_s. Edge e_t is then inducted into E_s. Note that the sample-induced graph is of necessity, *partially* induced, since induction is dependent on the current state of V_s. Vertices can appear and disappear from V_s but edges, once deleted/dropped, are lost from the stream and will not be represented even if their vertices are present in the final sampled set.

To the best of our knowledge, neither [3] nor any other work discusses how such streaming edge sampling can be efficiently implemented or parallelized ([3] does not provide any parallel algorithm details). We note that any parallel streaming algorithm for induced sampling should have the following (minimal) attributes:

(1) *Concurrent Dynamic Graph Update Capability*: The sampled subgraph G_s must be dynamically maintained, allowing *concurrent* insertion, replacement and search of vertices and edges.
(2) *Space-Efficient Operation*: Data structures used in the algorithm must allocate and operate on $O(|V_s| + |E_s|)$ main memory. The algorithm shouldn't assume access to $\theta(|V| + |E|)$ mainmemory.

Thus we require a data structure that can store the current sampled subgraph G_s as well as rapidly search it for the presence/absence of a vertex[1]. Clearly, we cannot use a $\theta(V)$ dense *status* array (like in parTIES) for search as it would violate the space-efficiency requirement. For storing the dynamic subgraph G_s, sophisticated frameworks that support graph updates, such as STINGER[9], Boost[13], Galois[22], would seem natural choices. However, a fundamental problem associated with these structures is that upon a *vertex deletion*, they only delete the incident edges and do not reclaim the complete memory associated with that vertex (eg. pointers to adjacency list). Directly using them requires at least $\theta(V)$ main memory. Moreover, we can utilize a simplified structure that just satisfies the requirements of our streaming algorithm and avoids performance overheads of a comprehensive database.

To solve both problems together, we propose to use an unordered array of size $\theta(k)$ to store V_s, *tightly coupled* with a sparse set data structure for rapid vertex search, specifically, a concurrent hash table. The coupling between the two structures ensures *consistent* insertions/deletions from both V_s and the hash table so that at any point in time, a vertex is perceived as either present or absent in both (by concurrent threads).

There are several separate chaining based parallel hash tables[21, 26, 29] that could potentially be used for this purpose. However, such hash tables store linked lists that require pointer chasing for search and constant memory management to allocate and free the elements in the list. While hopscotch hashing[15] and phase concurrent deterministic hash tables[27] provide very fast searches and updates, insertions and deletions in such hash tables displace pre-existing elements and hence, concurrent replace operations required in sampling are not supported. Some open addressing based hash tables can process concurrent insertions, deletions and searches[10, 23]. However, the hash table of[23] is very complex and cannot implement a dictionary. In [10], deletions are soft and create "tombstones" that insertion and search skip over. Since we require a large number of inserts and deletes in our streaming edge-sampling algorithm, such a table will quickly get filled mandating frequent rehashing which can become a performance bottleneck.

[1]This data structure is required for the intermediate and final output of sampling. It is not to store the input dynamic graph which is streamed in as an edge list.

3.2 Dynamic Graph Data Structure

Our design of a space-efficient and fast dynamic graph structure is motivated by the following key observations about subgraph updates in the parallel streaming edge-sampling algorithm: 1) *All* edge deletions are initiated by a vertex deletion. 2) The first k vertex operations are unbalanced insertions, followed by only replacements.

Keeping the above algorithm properties in mind, we propose to store the dynamic graph G_s as a k length array of structs - $vA[]$ and *relabel the vertices on-the-fly* before they are inserted. The new label of a vertex is simply its location (index) in the $vA[]$ array. Each element of $vA[]$ contains the following:

- $vId \rightarrow$ the original id of the vertex in G. The vId of all elements in $vA[]$ constitute the sampled vertex set V_s.
- $<>\ adj \rightarrow$ a dynamic array containing current set of vertices in V_s that are adjacent to vId, labeled using their original ids in G. Note that this is only the set of adjacent vertices discovered in the edges induced since vId's last (re)insertion into V_s. The adj of all elements in $vA[]$ constitute the induced edge set E_s.
- $lock \rightarrow$ a thread must acquire the $lock$ before modifying adj.

The dynamic graph interface provides several functions to modify V_s and E_s :
1) `insertVs`: adds vertices to V_s if $|V_s| < k$ or else return -1 indicating completion of initial phase. The mechanism to reserve locations for threads in $vA[]$ is similar to algorithm 1.
2) `replaceVs`: replaces the vertex at a given location with input vertex v and deletes the adjacency list of replaced vertex.

In addition, it also has a `push_edge`(pos, v) function to append v to $vA[pos].adj$; a `lockSpec`/`lockRand` functions to acquire lock at a specified/random location; and an `unlock` function to release the lock at a specified location.

Since $vA[]$ is unordered, we use a hash table for efficient vertex search. Note that the hash table must function as a dictionary, not only confirming the presence/absence of a vertex in V_s, but also storing its new label (location in $vA[]$).

3.3 Hash Table

Clearly, a hash table suitable for streaming edge-sampling must support concurrent insertions, searches and replacements. Moreover, *consistency* in the number of copies visible in *both* hash table and $vA[]$ (either 0 or 1) is essential for correct functioning of the sampling algorithm. To ensure consistent thread-safe updates between V_s and the hash table, we need to resolve several challenges. For example, during replacements, multiple threads may want to add a vertex u to V_s and might attempt to replace different vertices in $vA[]$, on behalf of the same vertex u. In such cases, we must prevent multiple incorrect deletions from $vA[]$ and the hash table along with multiple insertions of u. Further, any operations in the interval between committing a replacement in the hash table and commiting in the $vA[]$ should not result in erroneous updates to G_s.

Thus, our lightweight hash table design is tailored towards our specific stream sampling problem. *Delete* operations are designed to create empty cells that can be reclaimed by future insertions (rather than creating tombstones[10]) and clean cells where a search can terminate. Given that there are only replacements after first k vertices are sampled, such hash table will not require rehashing throughout sampling. To this purpose, we create our hash table as an array of structs-$HT[]$, in which each element has the following components:

- $status \in \{V, E, B\}$ $\longrightarrow$ indicating the presence (V) or absence (E) of a vertex at the location or Busy (B), indicating a deletion/insertion in progress.
- vId $\longrightarrow$ original vertex label in G; used as hash table key. $status$ and vId are stored as a single word.
- $label$ $\longrightarrow$ location of vId in $vA[]$ (i.e new label of vId).
- $lock$ $\longrightarrow$ for fine-grained locking of the element.
- cnt $\longrightarrow$ the *running count* of the number of vertices currently present in the hash table that hashed on the respective or a preceding location but are stored further ahead.

Let $HT[pos]$ denote an arbitrary element in the hash table. We define an element as clean if $HT[pos].cnt = 0$.

Claim 1. *A search must terminate if it encounters a clean element.*

Algorithm 2 describes our main hash table functions.
1) `search` starts from location h and terminates at either a valid (success) or clean (failure-ref. Claim 1) element.
2) `replace` inserts a given input vertex v and removes a randomly selected vertex from both hash table and V_s. Starting from the hashed position $HT[\text{hash}(v)]$, it first locks it to prevent a concurrent thread from inserting the same vertex and then searches for an empty location. If it encounters v, it terminates, returning the $label$ of v. Otherwise, if an empty location pos is found, it atomically sets it to busy and searches further ahead for v (this is needed only if a clean element was not found earlier) If the search for v succeeds it resets the lock and status and returns the label (line 28-29). If the search fails (lines 19-27), it locks a random location in $vA[]$ and replaces the vertex $prev$ occupying it with the new vertex v. $prev$ is then deleted and v inserted in the hash table. cnt is incremented at appropriate locations in the hash table before releasing the lock on $HT[\text{hash}(v)]$ to reflect the insertion of v.
3) `delete` searches for and deletes the input vertex v from the hash table. If $\text{hash}(v) = h$ and v is found at $pos = h + x$, it atomically decrements $HT[i].cnt\ \forall\ i \in \{h, ..., h + x - 1\}$.

The hash table also has an `insert` function to be used in the initial phase of the streaming algorithm. Under any sequence of calls to `insert` and `replace`, the following will hold true (we define a vertex as *searchable* if `search` can successfully find it in the hash table):

Claim 2. *Any vertex committed in V_s is searchable.*

Proof. Since insertion is completed in the hash table before unlocking $vA[]$, any vertex in V_s is definitely present in the hash table. Let v be a vertex such that $\text{hash}(v) = h$ and v is placed at $HT[h + x]$ and consider $HT[i].cnt\ \forall\ i \in \{h, ...h + x - 1\}$. Clearly, in the absence of deletions, $HT[i] \geq 1$ and hence, v is searchable. In the presence of deletions, consider a vertex $u \neq v$ to be deleted such that $\text{hash}(u) = h'$ and u is placed at $HT[h' + x']$. Deletion of $u \neq v$ can decrease $HT[i].cnt$ by at most 1 if $i \in \{h', ...h' + x' - 1\}$. Before `delete(u)` executes, $HT[i].cnt \geq 2$ because insertion of u and v have incremented $HT[i].cnt$ by 1, each. Hence, after deletion of u, $HT[i].cnt \geq 1$ and v is still searchable. ◻

Algorithm 2 Hash Table functions

```
    $G_s \longrightarrow$ dynamic graph; hash() $\longrightarrow$ hashing function
 1: function SEARCH($v$)
 2:     $pos$ = hash($v$)
 3:     do
 4:         if $HT[pos].\langle vId, status\rangle = \langle v, V\rangle$ then
 5:             return $HT[pos].label$
 6:         if $HT[pos].cnt = 0$ then return $-1$
 7:         $pos = pos + 1$
 8:     while $pos \neq$ hash($v$)
 9:     return $-1$
10: function REPLACE($v$, $G_s$)
11:     $pos$ = hash($v$), $h$ = hash($v$)
12:     while $CAS(\&HT[h].lock, 0, 1) = 0$ {}
13:     do
14:         $e = HT[pos]$
15:         if $e.\langle vId, status\rangle = \langle v, V\rangle$ then
16:             $HT[h].lock = 0$
17:             return $e.label$
18:         if $CAS(\&e.\langle vId, status\rangle, \langle *, E\rangle, \langle *, B\rangle)$ then
19:             $vlabel$ = search($v$)
20:             if $vlabel < 0$ then
21:                 $e.label = G_s$.lockRand()
22:                 $prev = G_s$.replaceVs($v$, $e.label$)
23:                 delete($prev$)
24:                 $e.\langle vId, status\rangle = \langle v, V\rangle$
25:                 for $loc = h, h+1, ..., pos-1$ do
26:                     $FAA(\&HT[loc].cnt, 1)$
27:                 $HT[h].lock = 0$, $G_s$.unlock($e.label$)
28:                 return $e.label$
29:             $e.status = E$, $HT[h].lock = 0$
30:             return $vlabel$
31:         $pos = pos + 1$
32:     while $pos \neq h$
33:     $HT[h].lock = 0$, return $-1$
34: function DELETE($v$)
35:     $pos$ = hash($v$), $h$ = hash($v$)
36:     do
37:         $e = HT[pos]$
38:         if $CAS(e.\langle vId, status\rangle, \langle v, V\rangle, \langle *, E\rangle)$ then
39:             for $loc = h, h+1, ..., pos-1$ do
40:                 $FAA(\&HT[loc].cnt, -1)$
41:         if $HT[pos].cnt = 0$ then return
42:         $pos = pos + 1$
43:     while $pos \neq h$
```

Claim 3. *There exists exactly one copy of vertex v in the Hash table if and only if $v \in V_s$.*

Proof. If $v \in V_s$, by claim 2, v is searchable and hence, exists in the hash table. Assume that there are multiple copies of a vertex v in the hash table. Since `replace` and `insert` lock $HT[\text{hash}(v)]$, the multiple copies must have been inserted sequentially one-by-one. By claim 2, after first insertion of v, subsequent attempts to insert v will find the existing copy and abort. Hence, no other copy of v would be created in the hash table. If $v \notin V_s$, either v was never inserted in which case, it is not in the hash table, or v was replaced from V_s. If v was removed, by claim 2, the `delete` function would have successfully found and deleted one copy of v. Since, there existed only one copy of v, after deletion, v must have been completely removed from the hash table. □

Claim 3 implies that vertices in the hash table will always be consistent with $vA[]$. During the transient phase of a function, vertices in $vA[]$ may differ from the hash table but since the corresponding location in $vA[]$ is locked, it does not result in erroneous updates to G_s.

3.4 Parallel Partially Induced Edge Sampling

Using the data structures described above, we develop a high throughput parallel Partially Induced Edge Sampling (parPIES) algorithm for sampling streaming dynamic graphs(algorithm 3). parPIES uses hash table functions to manipulate vertices in V_s and dynamic graph functions to populate E_s.

Each thread, when idle, exclusively reads a small batch of edges from the incoming stream and then processes them. In the first phase, parPIES calls `insert` for vertices of all edges (u, v), which inserts the vertices in $vA[]$ if they were not already present. All edges (u, v) are induced by pushing v to the adjacency list at location of u in $va[]$. When k vertices have been sampled, `insert` returns -1 terminating this phase.

In the second phase, for every edge (u, v), parPIES first calls `search(`u`)`. If the search fails, with probability $\frac{m}{uv_{id}}$, it inserts u in V_s by calling `replace(`u`)`. The same procedure is repeated for v as well. If u and v are either found or inserted in V_s, the edge is induced. However, before pushing v to the adjacency of u_l (new label of u), a check is performed on $vA[u_l].vId$ as a concurrent replace may have changed it.

Note that if u is deleted, all edges $\{(u, *)\}$ are removed (algorithm 3). However, spurious edges $(v, u) \mid v \in V_s$ continue to exist in E_s as they are stored in the adjacency of v. To prevent memory overflow from spurious edges, we perform a *cleanup* if *clean_cond* is asserted. *clean_cond* can be set if either a given number of edges have been processed since the last *cleanup* or if $|E_s|$ crosses a threshold (determined by available memory). During a *cleanup*, vertices in all adjacencies are searched for in the hash table and removed if not found . Reading an exclusive batch of edges from the graph stream, tracking uv_{id}, $|V_s|$ and $|E_s|$ variables and the global *clean_cond* condition uses checkpointing described in algorithm 1.

Extensions: Algorithm 3 can be easily tweaked for total induction on static graphs with only streaming access (eg. massive disk-resident graphs). If the graph is not changing, we can run multiple passes over the edge stream. The first pass will simply execute sampling using algorithm 3 without induction. The second pass will induce edges incident on vertices in the hash table, executing total induction over V_s.

Algorithm 3 parPIES pseudocode

```
    $G_s \rightarrow$ dynamic graph object,   $vSet \rightarrow$ hash table object
    $uv_{id} \rightarrow$ position of edge $(u, v)$ in input edge stream
    $clean\_cond = 0,\ V_s = \phi,\ E_s = \phi$
 1: for $i \in 0, 1...p-1$ do in parallel                ▷ first phase
 2:     while $|V_s| < k$ do
 3:         read $E'$ exclusive edges from stream
 4:         for each $(u, v) \in E'$
 5:             $u_l = vSet$.insert($u, G_s$)
 6:             if $u_l < 0$ then goto line 11
 7:             Execute lines 5-6 for $v$
 8:             $G_s$.lockSpec($u_l$)
 9:             $G_s$.pushEdge($u_l, v$)
10:             $G_s$.unlockSpec($u_l$)
11: $m = |E_s|$
12: for $i \in 0, 1...p-1$ do in parallel                ▷ second phase
13:     while graph is streaming do
14:         while $clean\_cond = 0$ do
15:             read $E'$ exclusive edges from stream
16:             for each $(u, v) \in E'$
17:                 $r = uniform\_random(0, 1)$
18:                 $u_l = vSet$.search($u$)
19:                 if $(r < m/uv_{id}) \land (u_l < 0)$ then
20:                     $u_l = vSet$.replace($u, G_s$)
21:                 else if $u_l < 0$ then process next edge
22:                 Execute lines 18-21 for $v$
23:                 $G_s$.lockSpec($u_l$)
24:                 if $(G_s.vA[u_l].vId = u)$ then
25:                     $G_s$.pushEdge($u_l, v$)
26:                 $G_s$.unlockSpec($u_l$)
27:         __synchronize()__
28:         clean spurious edges
29:         reset $clean\_cond$
30:         __synchronize()__
```

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

We conduct experiments on a 16-core machine with 2×2.6GHz 8-core Intel Xeon E5-2650 processors (256KB L2 and 20MB L3 cache) and 128GB main memory. The processors are enabled with 2-way hyperthreading (total 32 threads). The memory bandwidth of our machine as measured by the STREAM benchmark, is 55.2GBps for Copy and 61.4GBps for Add. All codes are written in C++ and compiled using G++ 4.7.1 with OpenMP v3.1 on Ubuntu 14.04 OS.

We use 7 large real world and synthetic graph datasets for performance evaluation; table 1 summarizes their characteristics. We only focus on undirected graphs and report time and memory performance of the algorithms discussed in this paper.

Implementation Details: In parTIES, edge deletion (line 13, algorithm 1) is only implemented if sampling fraction $\frac{k}{V} \geq 0.5$. This is because overlap in randomly selected edges is significant only if a large fraction of total edges are needed to sample target number of vertices. In parPIES, we instantiate the hash table with size $2k$ and thus, the maximum load factor is 0.5. This was done to mimic the strict memory constraints for a streaming algorithm. The adjacency lists are implemented using C++ STL vectors. Unless specified otherwise, the programs run on all 16 cores with hyperthreading (16h), sample 20% vertices i.e. $k = 0.2|V|$ and parPIES executes 1 cleanup just before writing the output.

Table 1: Graph Datasets

Dataset	Description	#Nodes(M)	#Edges(M)	Degree
soclj [17]	LiveJournal (social)	4.85	68.48	14.1
gplus [11]	Google Plus (social)	28.94	462.99	16
rmat25 [6]	Synthetic graph	33.55	536.87	16
pld [20]	Paylevel-Domain (hyperlink)	42.89	623.06	14.53
rmat26 [6]	Synthetic graph	67.11	1073.74	16
twitter [17]	Follower network (social)	61.58	1468.36	23.84
sd1 [20]	Subdomain graph (hyperlink)	94.95	1937.49	20.4

For measuring the throughput of parPIES, we randomly shuffle the edges and store them in a queue. To emulate the streaming aspect of dynamic graphs, threads only pop edges from the top of the queue until the queue is empty.

4.2 Results

Table 2 shows the performance of different phases of the sampling algorithms in parIES. For parTIES, we report the time of sampling and induction phases. For parPIES, table 2 reports the overall execution time of algorithm 3 (in_sampling) and a modified streaming algorithm that only executes edge sampling without induction (sampling). We also measure the time for search with sampling (s_sampling) by allowing vertex search for all the edges but removing memory operations i.e populating/clearing adjacency lists. For reference, we also report the execution time of an optimized sequential PIES algorithm[2].

We observe that sequential and parallel ASparTIES outperforms the corresponding versions of SparTIES, for all the datasets. The sequential implementation of ASparTIES benefits from the in-place storage of samples that eliminates memory operations required by dynamic vectors. The parallel speedup over all datasets averages 14.2× for asynchronous ASparTIES compared to 4× for synchronous parTIES. This is because of the inherent dynamic task allocation policy of ASparTIES that minimizes load imbalance and ensures that all threads work till the end of each phase. For the largest dataset *sd1*, ASparTIES creates a sampled subgraph of size 1442M edges (fig.2) in $<$ 2.5 seconds. For further evaluation of totally induced sampling, we will only use ASparTIES and refer to it as simply parTIES. We also note that parTIES spends $>$ 75% of the execution time in induction. This is because sampling reaches its target after touching a small subset of edges incident on V_s but induction has to probe all the edges incident on any vertex in V_s. Further, parTIES is able to induce upto 74% of the total edges in G by sampling only 20% of the vertices.

parPIES processes dynamic graph streams with a large average throughput of 63.6 million edges per second (MEPS). With 4B indices for vertices, this translates to $\approx$ 0.5GB of graph stream processed every second. parPIES also exhibits high parallel efficiency by achieving an average 14.74× speedup on 16 hyperthreaded cores. Compared to seqPIES, single threaded execution of parPIES is upto 30% slower on small graphs and comparable on large graphs. With 32 threads, parPIES executes $11.6\times - 17.2\times$ faster than seqPIES.

[2]Among the several hash maps tried (C++ STL, Boost, Google, hopscotch), Tessil hopscotch [1] was fastest and was used in sequential version of PIES (seqPIES). However, it does not support concurrent writes and cannot be used for parallel sampling. Execution time of sequential TIES implementation is almost same as single-threaded ASparTIES and we do not separately report it for brevity.

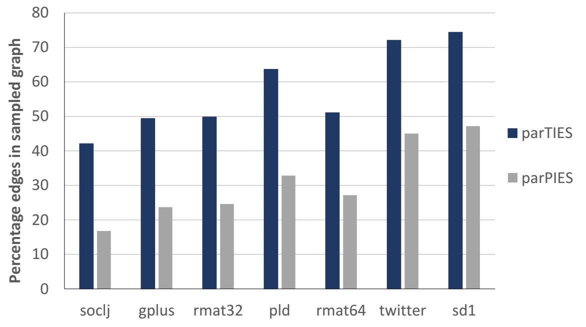

Figure 2: Total edges induced by parTIES and parPIES

For *soclj*, parPIES is able to process as much as 86.7 MEPS. This is because it induces fewer edges for *soclj* compared to other datasets and hence, performs less memory operations associated with inserting and clearing edges. Conversely, for *twitter* and *sd1*, parPIES is able to induce $>$ 45% of the edges in spite of only partial induction (fig.2). It is also evident from the fact that for sampling only, throughput achieved for *twitter* (619 MEPS) and *sd1* (602 MEPS) is comparable to *soclj* (571MEPS). For large datasets, we also observe that execution time of s_sampling is $<$ 40% of the time of in_sampling. This implies that $>$ 60% of the time in parPIES is spent on pushing edges and freeing the memory occupied by adjacency lists.

Quality of Sampled Graph: Also note that the number of edges induced by parPIES is lesser than parTIES (fig.2). The difference arises due to partial induction in parPIES where edges once dropped are not recovered even if the corresponding vertices are present in the final sampled vertex set.

Interestingly for very sparse graphs, partial induction in parPIES may result in better representative subgraphs than parTIES because total induction can overestimate the degree distribution[3]. Fig. 3 shows the degree distribution (cumulative density function) of the five real world datasets and their corresponding sampled versions. For all of these datasets, degree distribution of parPIES sampled subgraph is closer to the original, in comparison to the parTIES sampled subgraph. We also note that the distribution of sampled graphs obtained by parTIES and parPIES is almost identical to the corresponding sequential algorithms given in [3]. Thus, we obtain parallel speedup from parTIES and parPIES without altering the quality of sampled subgraph. For a detailed analysis of output quality, we refer the readers to [3].

Scaling: Tables 3 and 4 show the parallel speedup with increasing number of threads, for parTIES and parPIES, respectively. Both of our parallel algorithms are highly scalable and achieve close to 16× speedup when executed on 16 cores with hyperthreading (16(h)). We observe that the average speedup with 2 threads is only $1.5 - 1.6\times$. This is because sequential implementation does not incur any parallelism overheads. As mentioned in sections 2 and 3, we use shared variables between threads to checkpoint the global state of the program. For a multithreaded execution, the shared variables are updated by multiple cores incurring costs associated with cache coherency mechanisms. From 2 to 16 cores, the scaling is linear with speedup almost doubling with every 2-fold increase in the number of threads. Hyperthreading on 16 cores further accelerates

Table 2: Time (in seconds) for different phases in parTIES and parPIES for $k = 0.2|V|$. (16h) indicates 16 cores with 2-way hyperthreading and (1) indicates single-threaded execution

Algorithm	Phase	socLj		gplus		rmat25		pld		rmat26		twitter		sd1	
		(1)	(16h)	(1)	(16h)	(1)	(16h)	(1)	(16h)	(1)	(16h)	(1)	(16h)	(1)	(16h)
SparTIES	sampling	0.11	0.02	0.89	0.12	1.08	0.13	2.07	0.21	2.35	0.29	4.66	0.37	6.07	0.61
	induction	0.64	0.23	5.16	1.32	6.66	1.9	10.2	2.77	16.47	3.78	24.6	6.73	33	9.2
	total	0.75	0.25	6.05	1.44	7.74	2.03	12.27	2.98	18.82	4.07	29.26	7.1	39.07	9.81
ASparTIES	sampling	0.1	0.01	0.93	0.06	1.09	0.07	2.24	0.14	2.55	0.15	4.47	0.27	7.23	0.43
	induction	0.52	0.03	4.1	0.26	5.7	0.35	8.2	0.65	13.13	0.98	17.75	1.46	23.66	2
	total	0.62	0.04	5.03	0.32	6.77	0.42	10.45	0.78	15.67	1.13	22.2	1.73	30.88	2.43
seqPIES	in_sampling	9.17	-	88.3	-	110.2	-	149.9	-	247.5	-	386.5	-	621.8	-
parPIES	sampling	1.82	0.12	13.19	0.88	14.5	0.97	19.03	1.26	29.32	1.84	35.19	2.37	52.18	3.22
	s_sampling	8.55	0.4	63.6	3.76	63.92	4.38	70.54	4.78	130.9	9.06	148.6	10.18	194.6	13.58
	in_sampling	12	0.79	105.1	7.06	122.3	8.28	150.7	10.64	266.8	18.18	376.9	26.08	541.4	35.94

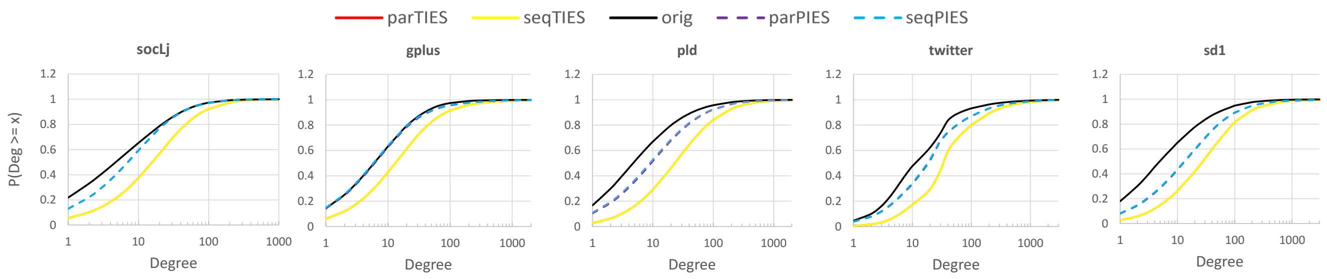

Figure 3: Degree CDF of original graphs and subgraphs sampled by sequential and parallel PIES and TIES algorithms

Table 3: parTIES parallel speedup over single thread

	Number of threads					
	(1)	(2)	(4)	(8)	(16)	(16h)
socLj	1	1.57	3.19	5.91	10.42	15.81
gplus	1	1.61	3.27	6.40	10.99	15.70
rmat32	1	1.62	3.19	5.85	10.59	16.06
pld	1	1.54	3.19	5.48	9.97	13.30
rmat64	1	1.63	3.23	6.09	11.15	13.88
twitter	1	1.59	3.12	5.60	10.15	12.86
sd1	1	1.60	3.19	5.84	10.51	14.33

Table 4: parPIES parallel speedup over single thread

	Number of threads					
	(1)	(2)	(4)	(8)	(16)	(16h)
socLj	1	1.51	2.87	5.73	9.61	15.10
gplus	1	1.58	2.94	5.57	10.01	14.89
rmat32	1	1.48	2.78	5.38	9.66	14.78
pld	1	1.45	2.76	5.31	9.48	14.16
rmat64	1	1.44	2.83	5.42	9.52	14.68
twitter	1	1.48	2.76	5.41	9.58	14.45
sd1	1	1.44	2.82	5.51	10.06	15.06

parPIES by an average of 1.52× and parTIES by 1.37×. The scaling is also quite consistent for all the datasets.

Sampling Fraction: is the ratio of the number of sampled vertices to the total vertices in original graph i.e. $\frac{k}{|V|}$. Fig.4 and 5 depict the variation in execution time with sampling fraction, for parTIES and parPIES, respectively. Note that the execution time increases more rapidly for parTIES than parPIES. This is because unlike parTIES that only processes edges incident on a sampled node, parPIES processes all the edges irrespective of the sampling fraction.

For parPIES, the increase in execution time is mostly a consequence of larger number of edges induced and the resulting memory operations. For *soclj*, induction increases drastically with sampling fraction and so does the execution time. For large graphs *twitter* and *sd1*, $> 45\%$ edges are induced with $k = 0.2|V|$ only and further increase in k marginally affects $|E_s|$. Hence, the execution time increases only 2× with an 8-fold increase in the sampling fraction.

Within parTIES, we observed that the execution time of sampling phase increases more rapidly as compared to induction. Intuitively, this happens because if a lot of vertices are already sampled, it becomes harder for a randomly selected edge to sample new vertices. On the other hand, induction phase probes lot of edges even with small k because edge sampling preferentially selects high degree vertices. As k is increased, the new vertices sampled have relatively small expected degree. For all our test cases, execution time of induction phase was still significantly higher than sampling.

parPIES Cleanups: As shown in algorithm 3, parPIES performs periodic *cleanups* of spurious edges to avoid memory overflow. Let $E_s(t)$ be the edge set at any time t (inclusive of spurious edges) during the execution of parPIES and T be the total execution time. Since

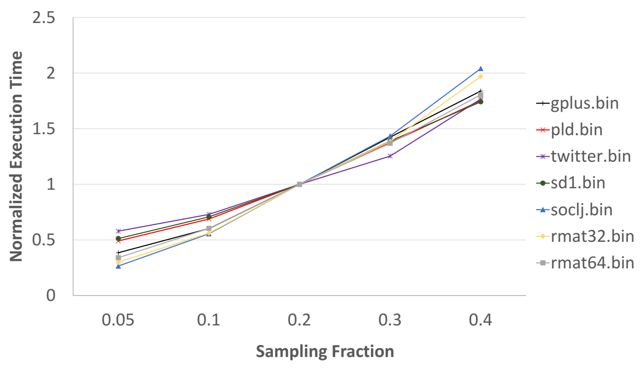

Figure 4: parTIES execution time (normalized for every dataset with time for $k = 0.2|V|$) vs sampling fraction

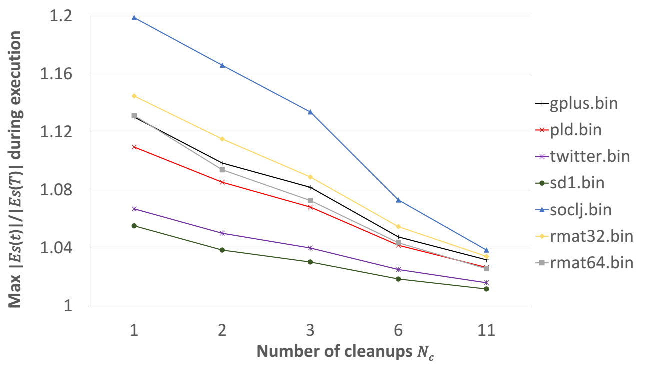

Figure 6: Maximum edges stored during execution of parPIES (normalized by output edge count) vs N_c

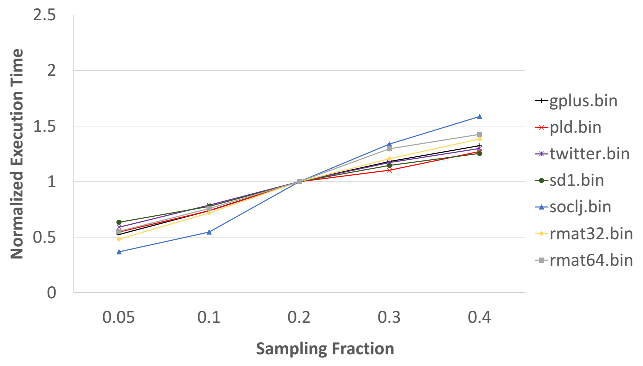

Figure 5: parPIES execution time (normalized for every dataset with time for $k = 0.2|V|$) vs sampling fraction

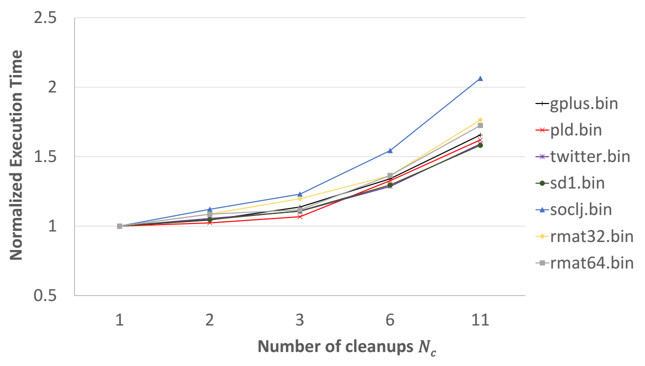

Figure 7: Execution time of parPIES (normalized by time with only one cleanup) vs N_c

a cleanup is always done just before outputting G_s, $E_s(T)$ has no spurious edges. To evaluate the effect of number of cleanups N_c on

(1) Storage (fig.6) $\longrightarrow$ we measure $\frac{\max_{t\in(0,T]} |E_s(t)|}{|E_s(T)|}$ vs N_c. We observed that $\approx 90\%$ of the total space requirement of parPIES is due to induced edges and hence, $\max_{t\in(0,T]} |E_s(t)|$ is a good indicator of main memory consumed.
(2) Performance (fig.7) $\longrightarrow$ we measure T vs N_c

The storage required for edges reduces as N_c increases. We observe that *cleanups* have a large impact on the $E_s(t)$ for *soclj* which requires 20% extra edge storage if no intermittent cleanup is done. This can be intuitively explained by the small fraction of edges in the output subgraph for *soclj* (fig.2) that increase the relative overhead of spurious edges. Contrarily, large graphs *twitter* and *sd1* require very little extra storage for spurious edges.

For almost all datasets, the execution time grows by only $50-70\%$ with an $11\times$ increase in N_c. This empirically shows the efficiency of search operations in our hash table data structure.

5 CONCLUSION & FUTURE WORK

In this paper, we introduced parIES - the first parallel Induced Edge Sampling framework that uses synchronization avoiding strategies and novel data structures to overcome the parallelization challenges in sampling and memory constraints of a streaming algorithm. For future work, we will explore better memory management schemes for the dynamic subgraph structure (such as thread-safe non-blocking vector containers[4]).

Also, there are several interesting directions to pursue in terms of different sampling algorithms and objectives , such as extensions to distributed implementations. Note that sampling on distributed memory systems poses a different set of challenges. Locking and global atomics may become a major bottleneck. A bulk synchronous approach such as sparTIES may be more suitable in such a scenario. For sampling a streaming graph, instead of processing small chunks from the graph stream, edges could be batched in larger groups and distributed across the machines for processing.

Acknowledgements: We thank the anonymous reviewers for their careful reading of our manuscript and their insightful suggestions. We also express gratitude to our shepherd Dr. Ana Lucia Varbanescu, whose comments and advice have been of great help in improving the quality of this paper.
This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086, National Science Foundation (NSF) under Contract Numbers CNS-1643351 and ACI-1339756 and Air Force Research Laboratory under Grant Number FA8750-18-2-0034. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, NSF or AFRL. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] 2016. https://tessil.github.io/2016/08/29/hopscotch-hashing.html. (Aug. 2016).

[2] Nesreen K Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. 2014. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1446–1455.

[3] Nesreen K Ahmed, Jennifer Neville, and Ramana Kompella. 2014. Network sampling: From static to streaming graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)* (2014).

[4] Antal Buss, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M Amato, Lawrence Rauchwerger, et al. 2010. STAPL: standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. ACM.

[5] Vito Giovanni Castellana, Alessandro Morari, Jesse Weaver, Antonino Tumeo, David Haglin, Oreste Villa, and John Feo. 2015. In-memory graph databases for web-scale data. *Computer* 48, 3 (2015), 24–35.

[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM.

[7] Kathryn Dempsey Cooper, Kanimathi Duraisamy, Hesham Ali, and Sanjukta Bhowmick. 2011. A parallel graph sampling algorithm for analyzing gene correlation networks. *Procedia Computer Science* (2011).

[8] Kathryn Dempsey, Kanimathi Duraisamy, Sanjukta Bhowmick, and Hesham Ali. 2012. The development of parallel adaptive sampling algorithms for analyzing biological networks. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE.

[9] David Ediger, Robert McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE.

[10] Hui Gao, Jan Friso Groote, and Wim H Hesselink. 2005. Lock-free dynamic hash tables with open addressing. *Distributed Computing* (2005).

[11] Neil Zhenqiang Gong, Wenchang Xu, Ling Huang, Prateek Mittal, Emil Stefanov, Vyas Sekar, and Dawn Song. 2012. Evolution of social-attribute networks: measurements, modeling, and implications using google+. In *Proceedings of the 2012 Internet Measurement Conference*. ACM.

[12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: distributed graph-parallel computation on natural graphs.. In *OSDI*, Vol. 12. 2.

[13] Douglas Gregor and Andrew Lumsdaine. 2005. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)* (2005).

[14] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 8, 9 (2015), 950–961.

[15] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *International Symposium on Distributed Computing*. Springer.

[16] Vaishnavi Krishnamurthy, Michalis Faloutsos, Marek Chrobak, Jun-Hong Cui, Li Lao, and Allon G Percus. 2007. Sampling large Internet topologies for simulation purposes. *Computer Networks* (2007).

[17] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM.

[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[19] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[20] 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* (2015).

[21] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM.

[22] 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*. ACM, 456–471.

[23] Chris Purcell and Tim Harris. 2005. Non-blocking hashtables with open addressing. In *International Symposium on Distributed Computing*. Springer.

[24] Davood Rafiei. 2005. Effectively visualizing large networks through sampling. In *Visualization, 2005. VIS 05. IEEE*. IEEE.

[25] Ryan A Rossi, Rong Zhou, and Nesreen K Ahmed. 2018. Estimation of Graphlet Counts in Massive Networks. *IEEE Transactions on Neural Networks and Learning Systems* (2018).

[26] Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)* (2006).

[27] Julian Shun and Guy E Blelloch. 2014. Phase-concurrent hash tables for determinism. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM.

[28] Usman Tariq, Umer I Cheema, and Fahad Saeed. 2017. Power-efficient and highly scalable parallel graph sampling using FPGAs. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE.

[29] Josh Triplett, Paul E McKenney, and Jonathan Walpole. 2011. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming.. In *USENIX Annual Technical Conference*, Vol. 11.

[30] 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*. ACM, 837–846.

[31] Bin Wu, Ke Yi, and Zhenguo Li. 2016. Counting triangles in large graphs by random sampling. *IEEE Transactions on Knowledge and Data Engineering* 28, 8 (2016), 2013–2026.

[32] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.

[33] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2018. Accurate, Efficient and Scalable Graph Embedding. *arXiv preprint arXiv:1810.11899* (2018).

[34] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System.. In *OSDI*. 301–316.