TurboGraph++: A Scalable and Fast Graph Analytics System

Seongyun Ko
POSTECH, Republic of Korea
syko@dblab.postech.ac.kr

Wook-Shin Han*
POSTECH, Republic of Korea
wshan@dblab.postech.ac.kr

ABSTRACT

Existing distributed graph analytics systems are categorized into two main groups: those that focus on efficiency with a risk of out-of-memory error and those that focus on scale-up with a fixed memory budget and a sacrifice in performance. While the former group keeps a partitioned graph resident in memory of each machine and uses an in-memory processing technique, the latter stores the partitioned graph in external memory of each machine and exploits a streaming processing technique. Gemini and Chaos are the state-of-the-art distributed graph systems in each group, respectively.

We present TurboGraph++, a scalable and fast graph analytics system which efficiently processes large graphs by exploiting external memory for scale-up without compromising efficiency. First, TurboGraph++ provides a new graph processing abstraction for efficiently supporting neighborhood analytics that requires processing multi-hop neighborhoods of vertices, such as triangle counting and local clustering coefficient computation, with a fixed memory budget. Second, TurboGraph++ provides a balanced and buffer-aware partitioning scheme for ensuring balanced workloads across machines with reasonable cost. Lastly, TurboGraph++ leverages three-level parallel and overlapping processing for fully utilizing three hardware resources, CPU, disk, and network, in a cluster. Extensive experiments show that TurboGraph++ is designed to scale well to very large graphs, like Chaos, while its performance is comparable to Gemini.

ACM Reference Format:
Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196915

1 INTRODUCTION

The fast growing size of graphs has encouraged many researchers to focus on improving scalability. We focus on scalability as an ability to process very large graphs without problems, such as the out-of-memory error, which is critical in assessing distributed graph analytics systems. There are mainly two approaches. One approach is to increase the number of machines until distributed memory machines can hold the input graph. Representative systems include GraphX [13], Pregel+ [41], GRAPE [11], and Gemini [44]. However, due to various factors, such as replicated vertices and edges, and intermediate results, the required memory size can be much larger than the input graph size. Thus, exact calculation of the memory size required for processing a specific query on a certain graph is difficult, leaving the systems vulnerable to crashes with the out-of-memory error. The other approach is to leverage external memory for scale-up. Representative systems include Chaos [30] and HybridGraph [37]. However, this approach typically exhibits poor efficiency compared with distributed in-memory graph analytics systems.

Efficiency is also an important and necessary factor, for assessing distributed graph analytics systems, which must not be sacrificed to scalability. Chaos is the first scalable, external-memory distributed system that processes queries without the out-of-memory error. However, it is much slower than recent in-memory systems, such as Gemini or Pregel+. Chaos relies heavily on disk and incurs excessively many I/Os for messaging. Recently, another external-memory system, HybridGraph, has been proposed for reducing disk I/Os with a hybrid message switching mechanism. However, HybridGraph is still much slower than the in-memory systems.

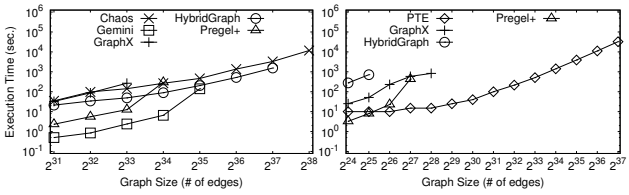
We observe that none of the state-of-the-art systems achieves both scalability and efficiency at the same time for processing even a simple query such as PageRank. Figure 1 (a) shows experimental results of the representative state-of-the-art systems for processing PageRank with 25 machines for doubling the graph size. Although Chaos can process up to the largest graph without crashing, the longest execution time shows its poor efficiency. While HybridGraph marginally improves Chaos, it is still much slower than Pregel+ or Gemini. We observe that Chaos and HybridGraph often wait for the I/Os for disk and network while the I/O bandwidths are not fully utilized. HybridGraph fails to load the graph when the number of edges reaches 2^{38} due to the out-of-memory error. Pregel+ and Gemini incur the disk swap or crash due to the out-of-memory error at large graphs showing limited scalability. GraphX performs the worst in terms of both scalability and efficiency.

In addition, graph analytics systems need to support neighborhood-centric analytics which process the multi-hop neighborhoods around each vertex [29]. Examples include triangle counting and Clustering Coefficient. Since the vertex-centric systems do not directly support the multi-hop neighborhood-centric analytics, a typical workaround implementation for processing those queries requires users to encode the neighborhoods of vertices into messages [12, 29]. For example, when processing triangle counting, each vertex generates a message containing a list of its neighbors and sends it to all the neighbors. In this case, the total size of messages can reach up to $\Sigma\, d_i^2$, where d_i is the degree of the i-th vertex. Hence, the memory requirement for processing neighborhood-centric analytics on existing vertex-centric systems can easily exceed orders of magnitude larger than the input graph, leading to the notorious out-of-memory error [27, 29, 32, 33, 43].

*Corresponding author

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196915



(a) PR. **(b) TC.**

Figure 1: Execution times for PageRank and triangle counting while doubling graph size on 25 machines with 32GB RAM.

Figure 1 (b) shows experimental results for processing the triangle counting query. Note that Chaos and Gemini do not support programming model APIs to implement it. Pregel+ shows the best performance at the smallest graph, but its execution time rapidly increases at size 2^{27} due to excessive disk swaps. Pregel+ and GraphX crash at size 2^{28} and size 2^{29}, respectively, due to the out-of-memory error. HybridGraph, an external-memory system, also crashes at size 2^{26} with the out-of-memory error because it holds the messages of vertices belonging to a block in memory to avoid disk I/Os while assuming that those messages fit in memory. Only PTE [27] can process up to the largest graph. PTE is the distributed algorithm tailored for triangle counting. It is implemented in Spark/Hadoop.

For distributed graph processing, the cost and quality of graph partitioning are also important performance factors. Representative graph partitioning schemes include vertex-cut [12, 15], edge-cut [22, 24, 34, 36], hybrid-cut [9]. A common objective of such graph partitioning schemes is to minimize the communication cost assuming that the network cost is the main bottleneck in processing queries. However, according to our analysis on the network bandwidth utilization in the above experiments, none of the existing systems saturates the network link bandwidth of our modern cluster. It confirms the claim that the old assumption that the network is the main bottleneck no longer holds in the modern cluster [6]. We need to reconsider the objectives of graph partitioning to improve the overall performance of a distributed graph analytics system.

A natural and challenging question is whether it is possible to develop a scalable and efficient graph analytics system. To solve this problem, we present TurboGraph++ by carefully designing crucial system components to fully utilize all hardware resources.

TurboGraph++ is the first distributed graph analytics system with no compromising of scalability or efficiency. We introduce a novel graph processing abstraction called nested windowed streaming for scalable and efficient processing of neighborhood-centric analytics. In neighborhood-centric analytics, the total size of neighborhoods around vertices can easily exceed the available memory budget. In order to bound the memory usage, we introduce the concept of nested windows.

We next propose a novel partitioning scheme, called balanced buffer-aware partitioning (BBP), for ensuring balanced workloads across machines with reasonable partitioning cost. Applying the regular grid partitioning scheme (p-by-p grids) to a graph could lead to seriously imbalanced workloads among machines, since each machine can end up having significantly different numbers of edges. BBP sorts the vertices by their degrees and distributes them among p machines in a round-robin manner. That is, it generates $p \times 1$ partitions of edges in a cluster. Then, BBP further divides each partition by $q \times r$ subpartitions, where q is determined by the available memory budget, and r is determined by p, q, and the number of NUMA nodes on each machine.

We next propose a novel processing strategy, called *three-level parallel and overlapping processing* (3-LPO), for fully utilizing all hardware resources (CPU, disk, and network) in each machine. Depending on the graph analytics type, latencies of a CPU computation task, a disk I/O task, and a network I/O task could vary. Thus, it is difficult to fully parallelize and overlap the three different tasks. Our 3-LPO carefully aligns the three tasks in a way that I/O tasks neither interfere with nor block the computation tasks. This way, the query processing time is nearly determined by the most bottlenecked hardware resource among CPU, disk, and network.

In summary, this paper makes the following key contributions.

- We develop a distributed graph analytics system without compromising scalability or efficiency. We present a new graph processing abstraction, called nested windowed streaming, for processing neighborhood-centric analytics without the out-of-memory error.
- We introduce a balanced buffer-aware partitioning with a reasonable cost to achieve balanced workloads across machines.
- In order to fully utilize all hardware resources as much as possible in each machine, we present three-level parallel and overlapped processing of CPU computation, disk I/O, and network I/O in order to maximize the overall efficiency.
- We demonstrate that TurboGraph++ is designed to scale well to handle large-scale graphs using 25 machines, like Chaos, while its performance is comparable to Gemini.

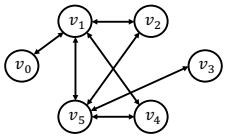
2 NESTED WINDOWED STREAMING MODEL

We model a graph as a directed graph $G = (V, E)$ where V and E represent a set of vertices and edges, respectively. Each vertex can have attributes. An undirected graph can be expressed as a directed graph with pairs of edges in both directions. A path from u to v in G is a sequence of vertices u, v_1, v_2, ..., v_{k-1}, v such that (u, v_1), (v_1, v_2),..., and (v_{k-1}, v) are edges in E. The vertex v and (v_{k-1}, v) are an ending vertex and an ending edge of the path, respectively. The length of a path is the number of edges on it. A simple path is a path in which all vertices are distinct.

We now formally define a class of queries, *the k-walk neighborhood query*, for processing the neighborhood-centric analysis. When $k = 1$, the query is used for processing the one-hop neighborhood of each vertex.

2.1 The k-walk Neighborhood Query

A vertex v is *k-hop reachable* from a vertex u if there exists a shortest path from u to v such that its length is less than or equal to k. The *k-hop neighborhood* of a set of vertices $\{v_1, \cdots, v_i\}$ is an induced subgraph of G consisting of all vertices k-hop reachable from $v_1, \cdots$, or v_i. A *walk* is more general than a simple path in that the same vertex can appear several times in a path. The *k-reachable walk set* of a vertex u, denoted as $W^k(u)$, is a partially ordered set consisting of 1) a set of all possible walks from u whose length is less than or equal to k and 2) a binary relation $\leq$ meaning that given two walks w_1 and w_2, $w_1 \leq w_2$ if w_1 is a prefix of w_2. Figure 2 shows



(a) A data graph G.

$W^1(v_0)$={(v_0,v_1)}

$W^2(v_0)$={(v_0,v_1), (v_0,v_1,v_0), (v_0,v_1,v_2),(v_0,v_1,v_4),(v_0,v_1,v_5)}

(b) Examples of $W^k(v_0)$.

Figure 2: A data graph G and examples of W^k.

```
foreach vertex u ∈ V_a do
    foreach walk w = (w', v') ∈ W^k(u) do
        Compute(w', v');
    end
end
```

Figure 3: The k-walk neighborhood query.

some examples of $W^k(v_0)$ in a data graph G. $W^1(v_0)$ is equivalent to the set of edges of v_0. $W^2(v_0)$ is the union of $W^1(v_0)$ and a set of additional walks of length two from the ending vertices of the walks in $W^1(v_0)$. Note that (v_0, v_1, v_0) is included, since the same vertex can be visited multiple times by the definition of the walk.

Given a set V_a of active vertices, a k-walk neighborhood query traverses the k-hop neighborhood of each vertex in V_a and performs computations. Here, the graph traversal refers to a process of visiting vertices in a graph. The representative one-walk neighborhood query is PageRank, where it accesses the one-hop neighborhood, i.e., the adjacency list of each active vertex during traversal. Thus, PageRank is naturally implemented with the vertex-centric programming model. To support the multi-hop (k>1) neighborhood query, the vertex-centric programming model requires construction of the multi-hop neighborhood in the local state of each vertex.

In order to support general graph traversal for the k-hop neighborhood in TurboGraph++, we introduce an important class of graph queries, called the *k-walk neighborhood query*, that enumerates every walk w in $W^k(v)$ and does some computation for each w. Note that a user can provide constraints to enumerate necessary walks only. Given a walk $w = (w', v')$, one can read or update attributes of the vertices in w' and can update attributes of v'.

The pseudo code in Figure 3 explains the semantics of the k-walk neighborhood query using the k-reachable walk set. Here, since $W^k(u)$ includes all possible walks in the k-hop neighborhood of u, we can process the k-walk neighborhood query by enumerating and processing all such walks. The walks in $W^k(u)$ are ordered according to its partial order. Consider $W^k(u)$. If a walk $w = (w', v')$ is the current walk, we guarantee that w' has been traversed and is still accessible. Then, Compute(w', v') accesses the vertex v' as well as the walk w' and performs a user-defined computation. If $k = 1$, w' is u. Here, in order to reduce the number of Compute calls, we can group the walks in $W^l(u)$ by the source vertex v of the ending edges and call Compute with v and its adjacency lists. Thus, with one function call of Compute, we can process many walks by iterating over edges of v.

2.2 Processing Model

Challenges. Although the vertex-centric model has been shown to be useful for programming various graph analytics, it is restricted to some simple analytics that only access the one-hop neighborhood of each vertex [16, 29]. It is not well-suited for the k-walk neighborhood queries because its restricted vertex-level abstraction leads to difficulties in programming, high communication overhead, and a large amount of memory for maintaining a multi-hop neighborhood in the local state of each vertex [16, 29].

Our goal is to provide a processing model that can efficiently process the k-walk neighborhood query with a fixed size memory and deliver a simple and intuitive programming model. For efficient support for the k-walk neighborhood query with a fixed size memory, we develop an efficient streaming model, so that we can process very large data with a fixed size memory. Thus, the first challenge is how we abstract vertices and their adjacency lists as streams and efficiently enumerate all k-walks in a streaming fashion.

The second challenge is how to minimize the cost of constructing a stream. Since graph data are stored in the disk, we have to perform sequential scans rather than random seeks and avoid repeatedly loading the same graph data in order to minimize the cost of constructing a stream. Unless we enumerate walks in W^k carefully, it can incur many random seeks as DFS-based graph traversal.

The last challenge is how to naturally extend an existing programming model to support the k-walk neighborhood query. This is of practical importance since programmers can easily accommodate the new programming model.

Proposal. We propose a novel processing model called the *nested windowed streaming model* (NWSM). To address the first challenge, NWSM regards a sequence of vertex attribute values and a sequence of the adjacency lists as a vertex stream and an adjacency list stream, respectively. We regard a sequence of the generated updates to the ending vertex of each walk during the user computation as an update stream. Each update is represented as a pair of a target vertex ID and an update value.

In order to enumerate all walks (w', v') in W^k and to access all attribute values of the vertices in w', NWSM nests k pairs of vertex stream and adjacency list stream inside one another. The vertex stream at level l, vs^l, contains the vertex attribute values of the source vertices for the ending edges of walks of length l in W^k where $1 \le l \le k$. The adjacency list stream at level l, $adjs^l$, contains the ending edges of the walks of length l in W^k where $1 \le l \le k$. The window is defined based on memory size in order to allocate a fixed sized memory space for a window. The stream data are inserted into the size-based window until memory size reaches its maximum limit. That is, we divide each stream into disjoint windows so that each window can be safely loaded in memory. A vector is allocated for each attribute of the vertex so that only relevant attributes can be loaded in memory if necessary.

Now, we address the second challenge. If we follow edges of the vertices one by one as in DFS-based graph traversal and load the next level streams accordingly, the cost of constructing the next level stream can be expensive; 1) we may have to load the same vertices and their adjacency lists multiple times even at the same level and 2) we may suffer from random disk seeks due to the random access order. In order to handle this problem, we propose a concept of *mark-and-backward-traversal*. Given a window at level l, when we want to construct the stream for level $l + 1$, we first mark vertices of interest for level $l + 1$ by accessing the outgoing edges at the level l window. Then, we construct the streams for those

```
SCATTER(Int l, VertexStream vs^l, AdjStream adjs^l):
 1: foreach VertexWindow vw_i^l ∈ vs^l with i ∈ [1, q] do
 2:     foreach AdjWindow adjw_{i,j}^l ∈ adjs_i^l with j ∈ [1, n] do
 3:         foreach adjlist adj ∈ adjw_{i,j}^l do
 4:             adj_scatter^l(adj.src, Iterator(adj));
 5:         end
 6:         if (l ≠ K) then
 7:             SCATTER(l+1, VertexStream(voi^{l+1}),
                AdjStream(voi^{l+1}));
 8:         end
 9:     end
10: end

GATHER(VertexStream vs, UpdateStream us):
 1: Generate substream us_1, ..., us_q for vertex windows
    vw_1, ..., vw_q, respectively, where for every update u in us_i
    (1 ≤ i ≤ q) we have u.vid ∈ vw_i
 2: foreach VertexWindow vw_i for i ∈ [1, q] do
 3:     foreach Update u ∈ us_i do
 4:         vertex_gather(u.dst, u.val);
 5:     end
 6: end

APPLY(VertexStream vs):
 1: foreach VertexWindow vw_i for i ∈ [1, q] do
 2:     foreach Vertex v ∈ vw_i do
 3:         vertex_apply(v);
 4:     end
 5: end
```

Figure 4: Scatter-Gather-Apply with NWSM.

vertices and their adjacency lists by performing two sequential scans, one for each, respectively. Here, whenever a window at level l is loaded in memory, we can follow incoming edges of the vertices performing the backward traversal to enumerate all the walks that can be enumerated in memory. In this way, we avoid the repeated accesses to the same vertices and edges, and random disk seeks.

In order to address the third challenge, NWSM chooses the scatter-gather-apply (GAS) model [12, 30, 31] among the existing programming models due to its popularity and extends it to support the k-walk neighborhood query. Our extended programming model is a generalization of the vertex-centric GAS model. Our programming model delivers a much more intuitive way for programming for the k-walk neighborhood queries.

Scatter-Gather-Apply with NWSM. Figure 4 shows execution semantics of NWSM using the concepts of the stream and window. vs^l is partitioned into disjoint q ($q \geq 1$) vertex windows (vw_1^l,...,vw_q^l) for $i \in [1, q]$. Then, $adjs^l$ is split into disjoint q ($q \geq 1$) adjacency list substreams ($adjs_1^l$,...,$adjs_q^l$) such that we have the range of source vertex IDs of the adjacency lists in $adjs_i^l$ in vw_i^l. Each $adjs_i^l$ is partitioned into n disjoint ($n \geq 1$) adjacency list windows ($adjw_{i,1}^l$,...,$adjw_{i,n}^l$) according to the memory size for $adjw^l$.

Scatter: The SCATTER can enumerate necessary walks and execute the user-defined computations. It takes as the input parameters the current level l, vs^l, and $adjs^l$. It can generate the update stream as the output during the computations. Note that when $l = 1$, vs^1 and $adjs^1$ correspond to the active vertex stream and the corresponding adjacency list stream in each machine, respectively.

Given vs^l, we iterate over the vw_i^l ($1 \leq i \leq q$) of vs^l (Line 1). Then, with vw_i^l, we stream $adjs_i^l$ over the n disjoint windows $adjw_{i,j}^l$ ($1 \leq j \leq n$) of $adjs_i^l$ (Line 2). That is, given vw_i^l and $adjw_{i,j}^l$, we load the subgraph that include the vertices in vw_i^l and their adjacency lists in $adjw_{i,j}^l$. Then, we execute the user-defined function, $adj\_scatter^l$, with each adjacency list in $adjw_{i,j}^l$ (Line 3-5). Here, $adj\_scatter$ marks some vertices at voi^{l+1} when $l \in [1, k-1]$ and executes the computation when $l = k$. After that, if $l \neq k$, it nests the vs^{l+1} and $adjs^{l+1}$ using voi^{l+1} (Line 6-8).

In this way, in order to enumerate W^k, NWSM nests k pairs of the streams (vs^l and $adjs^l$) by recursively calling SCATTER. The l-th call of SCATTER processes the vs^l and $adjs^l$ and recursively defines the voi^{l+1} for $l \in [1, k-1]$. This recursive process continues up to the last k-th level. That is, by nesting k pairs of streams (vs^l and $adjs^l$) using the k pairs of windows (vw_i^l and $adjw_{i,j}^l$), we load the subgraphs that include a subset of W^k and enumerate all the walks in it. Now, when the l-th window reaches the end of its stream for $l \in [2, k]$, the prior $(l-1)$-th window slides. The recursive process repeats until all windows reach the ends.

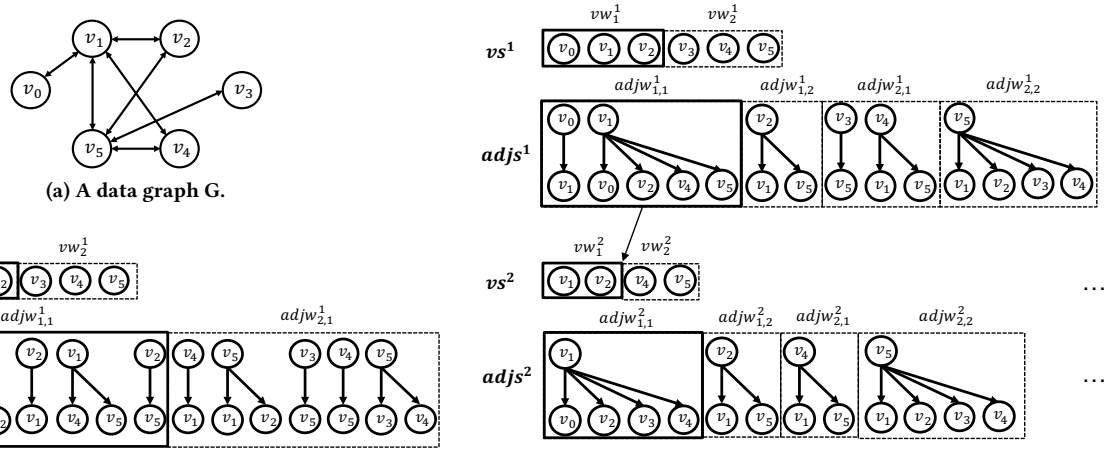
When processing the adjacency lists in the stream, the two types of adjacency list iterating modes can be used: a partial and full adjacency list mode. In the partial adjacency list mode, the adjacency list includes a subset of adjacent neighbors of each vertex. It can be used when the computation unit is an edge such as PageRank and SSSP. In the full adjacency list mode, the adjacency list includes all adjacent neighbors of each vertex. It must be supported to efficiently process subgraph matching queries such as triangle counting, where fast intersection of adjacency lists is necessary.

Gather: The GATHER function takes the update stream as the input stream and performs the aggregation. The updates generated by each machine during the scatter phase must be forwarded so that each update is located at the designated machine for its target vertex. The forwarded updates are made into substreams of us, us_1, ..., us_q, such that each update $u \in us_i$ ($1 \leq i \leq q$) has $u.vid \in vw_i$ (Line 1). Then, for vw_i (Line 2), we stream us_i and invoke the user-defined $vertex\_gather$ function that updates the attribute values of the target vertex (Line 3-5).

Apply: The APPLY function takes the updated vertex stream and recomputes the attribute values of vertices. For vw_i ($1 \leq i \leq q$) (Line 1), we invoke the user-defined $vertex\_apply$ function with each vertex $v \in vw_i$ (Line 2-4).

We explain how NWSM processes the PageRank and triangle counting queries (Figure 5). Figure 5 (a) shows a data graph G. Here, we suppose that vs^1 is divided into two disjoint windows vw_1^1 and vw_2^1. The range of vw_1^1 is $[v_0, v_2]$, and the range of vw_2^1 is $[v_3, v_5]$.

Figure 5 (b) shows an example of processing the PageRank query. We use the partial adjacency list mode. vs^1 includes all vertices in G, and $adjs^1$ contains all of their adjacency lists. $adjs^1$ are partitioned according to vw_1^1 and vw_1^2. Here, since we use the partial adjacency list, the adjacency lists in $adjw_{1,1}^1$ and $adjw_{2,1}^1$ are further partitioned by the range of their ending vertices. The scatter



(a) A data graph G.

(b) PageRank.

(c) Triangle counting with a partial order constraint ($u < v < w$).

Figure 5: Examples of vertex windows and adjacency list windows under NWSM.

function iterates the edges of the partial adjacency lists in $adjw^1_{1,1}$ and produces the updates to the ending vertices of the edges. Then, we slide to vw^1_2 and $adjw^1_{2,1}$ and repeat the same computation. The gather and apply phases proceed to accumulate the updates and update the PageRank values.

Figure 5 (c) shows an example of processing the triangle counting query. We use the full adjacency list mode. The vs^1 includes all vertices in G, and the $adjs^1$ contains all of their adjacency lists. The triangle counting query loads the two-hop neighborhood of vertices using the two nested windows and finds the triangles by counting the common neighbors of the two connected vertices. Here, we assume that each adjacency list window contains up to five edges.

At level one, we load vw^1_1 and $adjw^1_{1,1}$. Among the target vertices of the edges in $adjw^1_{1,1}$, we mark the vertices for level two which satisfy the partial order constraint ($u < v$ where v is a neighbor of u) to avoid counting the same triangle multiple times. At level two, we load vw^2_1 and $adjw^2_{1,1}$. Then, we perform the backward-traversal to find the parent vertices of the vertices in vw^2_1. We compute the intersection of the adjacency lists of u in vw^2_1 and its parent v in vw^1_1. In this case, there is no triangle found. Then, we slide to $adjw^2_{1,2}$ and find a triangle by intersecting the adjacency lists of $v2$ and $v1$, (v_1, v_2, v_5). We repeat processing the second vertex window at level one.

2.3 Programming APIs

We explain our programming APIs for programming the k-walk neighborhood query. We present two programming examples for PageRank and triangle counting in Appendix A.1.

Users do not need to be aware of the details of our execution model. Instead, they only need to know the general concepts of the scatter, gather, and apply phases as well as the semantics of the k-walk neighborhood query in Figure 3.

Figure 6 shows our programming API. Users can define a schema for the vertex attribute and a schema for update values. The user program inherits the TBGPPApp class and implements a member function, Run, that initializes variables and launches the query. k represents the maximum length of walks. $voi[l]$ represents the vertices of interest for level l. Specifically, $voi[1]$ represents starting vertices of interest during the current superstep. $voi[l]$ represents source vertices in enumerated walks of length l starting from $voi[1]$.

Regarding the scatter phase, the users need to define different scatter functions, $adj\_scatter[l]$, for each level l. Here, $adj\_scatter$ corresponds to the COMPUTE in Figure 3 for each level. For level $l \in [1, k-1]$, $adj\_scatter[l]$ enumerates walks of length $l+1$ for W^{l+1}. Specifically, $adj\_scatter[l]$ takes a source vertex u of an ending edge of a walk of length l and marks the adjacent vertices of u in $voi[l+1]$ to enumerate walks of length $l+1$. Users can specify a partial order constraint to skip unnecessary walks to enumerate at the next level. For level $l = k$, $adj\_scatter[k]$ executes the computation. Regarding the gather phase, users need to define $vertex\_gather$, which is a gather function that aggregates updates on vertex attribute values. Regarding the apply phase, users need to define $vertex\_apply$, which is an apply function that recomputes the vertex attribute values using the aggregated values.

The users invoke Start to start processing the k-walk neighborhood query. The ProcessVertices function initializes the vertex attribute values before invoking Start.

GetParentList and GetCommonNbrList are provided as system primitives. GetParentList(Int l, VertexID v) returns a list of parents of v in the walks of length $l+1$ with v as the ending vertex. GetCommonNbrList(VertexID u, VertexID v) returns the list of the vertices that are common neighbors of u and v. We provide GetAdjList(VertexID v) that returns the adjacency list of v.

3 BALANCED BUFFER-AWARE PARTITIONING

We revisit the graph partitioning problem in a distributed graph analytics system considering two questions: 1) What are the most important partitioning objectives for improving overall processing performance? 2) How efficient should the graph partitioning be?

To answer the first question: Our objectives are threefold. 1) We balance the workloads across machines to maximize the overall efficiency of graph processing. In order to avoid imbalanced workloads, we need to balance the number of edges as well as the number of high-degree and low-degree vertices among machines. 2) We also

```
class TBGPPApp {
    // User has to implement
    def Run() = 0;

    // System-provided
    def ProcessVertices(Function function);
    def Start();

    def GetParentList(Int l, VertexID v);
    def GetCommonNbrList(VertexID u, VertexID v);
    def GetAdjList(VertexID v);

    // Member variables
    Int                 k;
    VertexIdSet[]       voi;
    ScatterFunction[]   adj_scatter;
    GatherFunction      vertex_gather;
    ApplyFunction       vertex_apply;
};
```

Figure 6: TurboGraph++ nested windowed streaming API.

balance the number of vertices across machines, so that each machine requires the same memory budget for ensuring in-memory access to the vertices. 3) We use range partitioning rather than hash partitioning, where the vertex IDs assigned to each machine are consecutively allocated. This way, we can access the state of any vertex by using cheap offset calculation. Otherwise, we would have to rely on costly hash maps.

In order to attain our objectives, we propose our graph partitioning scheme, Balanced Buffer-aware Partitioning (BBP), that balances workloads among machines being aware of the necessary memory budget for buffers of the vertex attribute values and the adjacency lists. Here, buffers refers to various in-memory data structures used to temporarily store data such as vertex window, adjacency list window, local gather buffer, and global gather buffer.

We explain how BBP achieves our objectives in Figure 7 (a) with an example of a graph with a skewed degree distribution. The grey-colored cell in the i_{th}-row and the j_{th}-column indicate an edge from vertex i to vertex j. BBP first sorts the vertices by their degrees and distributes them across machines in a round-robin manner to balance the number of edges as well as the number of high-degree and low-degree vertices among machines. In total, we have p-by-1 partitions, where p is the number of machines. Then, BBP renumbers vertex IDs to (1) assign consecutive vertex IDs to the vertices in each machine and (2) exploit the degree order information as partial order constraints that can significantly accelerate the set intersection computation [5, 20, 27, 28]. For vertices in each machine, new vertex IDs are assigned in descending order of their degrees. In this way, we allocate lower vertex ID for the vertices with higher degrees within each machine. Figure 7 (b) shows the result of the balanced partitioning with two machines. The renumbered vertices and their edges are partitioned into two ranges $[0, 3]$ and $[4, 7]$. Each partition has a well-balanced number of vertices and edges.

After renumbering, BBP further partitions the edges in each machine by the equally sized range of source and target vertex ID. The source vertices are range-partitioned into q ranges, and the destination vertices are range-partitioned into $p \times q$ ranges, where q is determined by the *available* memory budget in a machine for processing neighborhood-centric analytics, which we will discuss in Section 4.2. We call a partition of edges *edge chunk*. Each machine has q-by-pq edge chunks. Figure 7 (c) shows the resulting

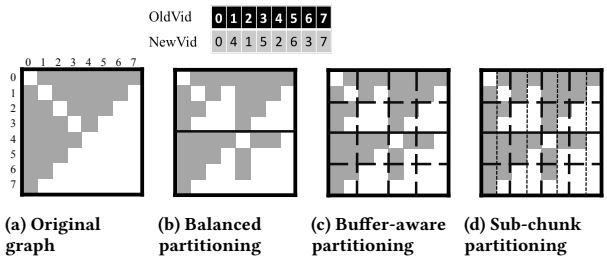


Figure 7: Balanced buffer-aware partitioning (p=2, q=2, r=2).

partitions with $q = 2$. For the vertices, we partition the vertex attribute data according to the ranges of the source of edge chunks. We refer to a partition of a vertex as a *vertex chunk*. If each vertex has j attributes, a vertex chunk is stored as j vector chunks in our columnar representation. Overall, there are $p * q$ vertex chunks.

Additionally, in order to reduce synchronization costs across NUMA nodes due to concurrent updates by Compare-And-Swap (CAS) for the in-memory local gather operations, we further range-partition each edge chunk by the destination vertices into r edge sub-chunks where r is the number of NUMA nodes. We balance the number of edges among the sub-chunks by exploiting the degree information of vertices. Figure 7 (d) shows the partitioning results for two NUMA nodes per machine.

Now, we address the second question: How efficient should the graph partitioning be? The analytics queries can be executed once the graph partitioning is done. Thus, the graph partitioning cost needs to be amortized over such queries. As long as the partitioning cost is less than typical neighborhood-centric analytics, such as triangle counting, the partitioning costs should be reasonable. The major cost of our partitioning method is sorting. The worst case sorting cost $O(|E| * log|E|)$ is clearly smaller than $O(|E| * |E|^{0.5})$, the worst case optimal cost for triangle counting [21]. For efficiently processing BBP, we fully overlap the computation and I/Os in order to fully utilize their bandwidths. The experimental results show that the processing time of BBP is comparable to the preprocessing time of the state-of-the-art systems.

4 EXECUTION OF NWSM

Here, we describe the execution of NWSM. We provide a detailed execution strategy for NWSM using three-level parallel and overlapped processing (3-LPO). Then, we explain our memory allocation policy. We provide implementation details in Appendix A.3

4.1 Three-level Parallel and Overlapped Processing

To increase the overall efficiency of graph processing, we must fully parallelize and overlap CPU computations and network/disk processing, so that underlying system hardware resources are fully utilized. For this, we present a novel execution strategy, three-level parallel and overlapped processing (3-LPO), that achieves full parallelism and full overlap of CPU processing, disk I/O processing, and network I/O processing.

TurboGraph++ starts to process a given query by executing PROCESSNWSM in Algorithm 1, which takes l, vs^l, $adjs^l$ as level, the l-th level vertex stream, and the l-th level adjacency list stream,

Algorithm 1: ProcessNWSM

```
   Input: Level l, VertexStream vs^l, AdjStream adjs^l
1: q_new = ComputeQValue(k)
2: if (q_new > q) then
3:     ExecuteBBP(q_new)
4: end
5: if (l = 1) then
6:     spawn Gather();
7: end
8: // overlap scatter phase with gather phase
9: while vw^l = Read (vs^l.NextWindowRange()) do
10:    foreach j ∈ [1, pq/⌊q/q_new⌋] do
11:        LGB^l.clear();
12:        while adjw^l = AsyncRead (adjs^l.NextWindowRange(),
           GetRange(vw^l), GetRange(j), adj_scatter^l) do
13:            wait until the above AsyncRead completes;
14:            if (l ≠ k) then
15:                ProcessNWSM (l + 1, VertexStream(voi^{l+1}),
                   AdjStream(voi^{l+1}));
16:            end
17:        end
18:        AsyncSend(LGB^l)
19:    end
20: end
21: if (l = 1) then
22:    GlobalBarrier();
23:    spawn GatherSpilledUpdates(UpdateStream(updates in disk));
24:    spawn Apply(VertexStream(all vertices in my machine));
25: end
```

respectively. ProcessNWSM consists of three phases: the scatter phase, gather phase, and apply phase.

The minimum memory requirement is closely related with a partitioning parameter q. Thus, before executing a query, TurboGraph++ first computes the minimum q value q_{new} for processing this query without an out-of-memory error (Line 1). Let q denote the current value used for the BBP. If $q_{new} > q$, we need more fine-grained partitions, so we execute BBP with q_{new} to repartition the graph. Otherwise, we use q (Lines 2 ~ 3).

Before executing the scatter phase, each machine launches a task to gather messages that will be generated from the scatter phase of other machines (Lines 5 ~ 6). This gather task is called *global gather*. For ease of explanation, assume that we use the partial adjacency list mode only. After that, we will explain how to support the full adjacency list mode.

The scatter phase at the l-th level reads vw^l from vs^l (Line 9) and $adjw^l$ from $adjs^l$ (Line 12). When $l = 1$, each machine reads the vs^1 containing active vertices and the $adjs^1$ from its local disk only. This is due to data parallelism; i.e., vertices along with their adjacency lists are range-partitioned across machines. In this situation, since there is no network I/O, we only need to overlap CPU processing with disk I/O processing in each machine. When $l > 1$, reading from the vs^l containing voi^l and the $adjs^l$ can involve network I/Os as well as remote disk I/Os. TurboGraph++ supports synchronous reads and asynchronous reads; both Read and AsyncRead support local disk read as well as remote disk read + network transfer. Whenever AsyncRead completes the I/O of some adjacency lists in $adjw^l$, TurboGraph++ iterates over the adjacency lists and invokes the callback function $adj\_scatter^l$ for each adjacency list. That is, TurboGraph++ overlaps the CPU processing with network I/O processing and remote disk I/O processing. Since all machines execute the scatter phase at the same time, we can fully utilize the cluster-wide CPU, network, and disk resources by full overlap.

Algorithm 2: Gather

```
   Input: UpdateStream us
1: mid = GetMyMachineID();
2: foreach Update u ∈ us do
3:     if u.vid ∈ [ |V|/p (mid − 1), |V|/p (mid − 1) + |V|/(p*q_new) ) then
4:         apply u to the gather buffer
5:     else
6:         spill u to disk with partitioning
7:     end
8: end
```

In order to support "in-memory local gather" in a machine that sends updates to other machines during the scatter phase, we allocate a local gather buffer at each level l, LGB^l. When a destination vertex is updated in $adj\_scatter^l$, we directly call the vertex gather function to perform the in-memory local gather in order to reduce the size of the update stream to send. This is how TurboGraph++ overlaps the scatter phase with the local gather phase.

Now, we explain how the global gather task works. Algorithm 2 shows how it works. For each update u, if u can be gathered in memory, we accumulate u to the global gather buffer (Line 4). If u cannot be gathered in memory, it is spilled to disk (Line 6). Here, we create $q_{new} - 1$ partitions in the disk. Note that this gathers messages generated by all scatters across machines.

After consuming all of ws^1 and as^1, TurboGraph++ needs to wait until all updates are gathered either in memory or disk by setting a global barrier (Line 22). After that, it needs to launch a task to gather remaining updates stored in disk (Line 23) and launch the apply task (Line 24). We model this as a producer-consumer problem, where the global gather task is a producer and the apply task is a consumer. For this purpose, we create another global gather buffer, so that we have a double buffer for the producer and the consumer. This is how we overlap the gather phase with the apply phase. Algorithm 3 shows how we gather the spilled updates, and Algorithm 4 shows how we perform the apply task. We can immediately start the apply task using the global gather buffer, while GatherSpilledUpdates loads the updates from disk and accumulates them to another global gather buffer. Algorithm Apply synchronizes the producer and consumer problem by waiting for the global gather task to finish merging spilled results for the current source vertex ID range (Line 2 of Apply). For each vertex in the vertex stream, we finally call *vertex_apply* to the corresponding vertex (Line 4 of Apply).

Now, we explain how to support the full adjacency list mode with two constraints: 1) Since the target vertex ID range of a full adjacency list spans the whole range of vertex IDs, the size of LGB^l could grow large. Even in this case, we allocate a fixed size memory for LGB^l. If LGB^l gets overflow while we append the generated updates during the scatter phase, we send the results accumulated so far in LGB^l to corresponding global gathers across machines.

Algorithm 3: GatherSpilledUpdates

Input: UpdateStream[] us
1: **foreach** *PartitionID* $pid \in [2, q_{new}]$ **do**
2: **foreach** $Update\ u$ in $us[i]$ **do**
3: apply u to the gather buffer for the current vw
4: **end**
5: **end**

Algorithm 4: Apply

Input: VertexStream vs
1: **while** vw = Read(vs.NextWindowRange()) **do**
2: **wait** until the gather task finishes merging spilled results for vw;
3: **foreach** $v \in vw$ **do**
4: $vertex\_apply(v)$;
5: **end**
6: **end**

2) When we store a large full adjacency list, $adjw^l$ can run out of space. Then, we keep a part of the adjacency list in $adjw^l$, while the remaining part is resident in the disk. TurboGraph++ also supports fast listing operations for adjacency lists stored in the disk.

4.2 Memory Allocation

In order to avoid the notorious out-of-memory error, we need to carefully analyze the memory requirement for a given k. The minimum memory requirement is closely related to partitioning parameters p and q, since p and q are the number of machines and the number of vertex chunks in a machine, respectively. Thus, we need to derive the minimum value of q, q_{min}. Based on q_{min}, we can derive a memory requirement for the input query. The following theorem states how we can compute q_{min}. Here, $|M|_{total}$ denotes the total memory budget. A fixed-sized memory is reserved for storing auxiliary data structures including network buffers. $|VA|$ denotes the total number of bytes of vertex attributes. α represents the relative size of the set containing the vertices of interest compared to the $|VA|$. PS is the size of page that stores the adjacency lists. The proof is in Appendix A.2.

Theorem 4.1. *Given a k-walk neighborhood query, the minimum value of q, q_{min} is as follows.*

$$q_{min} = \left\lceil \frac{1}{p} \frac{(4k+1)|VA|}{|M|_{total} - k(2PS + \alpha|VA|)} \right\rceil \quad (1)$$

Now, we explain our memory allocation policy. We first allocate a fixed memory budget for vw^l and LGB^l of each level and GGB according to our BBP with q_{min} as in Equation 3. We also allocate a fixed memory budget for voi^l of each level for storing the vertices of interest. Then, we allocate all the remaining memory budget to the adjacency list windows. We equally divide the remaining memory budget and assign it to $adjw^l$ at each level except for the one at the last level. Note that the $adjw^l$ at the last level only needs a small memory budget since there is no further nesting. We use this memory allocation policy for our evaluation.

5 EXPERIMENTS

In this section, we evaluate the performance of TurboGraph++. The main goals of this experimental study are as follows.

- We evaluate our BBP scheme and 3-LPO strategy. We show that BBP can be performed with a reasonable pre-processing cost to achieve balanced workloads across machines. Then, we show that our 3-LPO strategy achieves full parallelism and full overlap of CPU computation, disk I/O, and network I/O for varying hardware configurations. (Section 5.2)
- We compare the performance of TurboGraph++ against the state-of-the-art distributed graph analytics systems using real-world graph data sets. (Section 5.3)
- We show scalability and efficiency of TurboGraph++ compared to the state-of-the-art distributed graph analytics systems for varying graph size and the number of machines. (Section 5.4)

5.1 Experimental Setup

Datasets: We use synthetic graphs of various sizes and four real graphs (Twitter [18] (TWT), YahooWeb [1] (YH), Clueweb09 [2] (CW09), Clueweb12 [3] (CW12)). Synthetic graphs are generated according to the RMAT model [8] using the TrillionG graph generator [26]. We denote by RMAT_X the RMAT graph with 2^{X-4} vertices and 2^X edges. Table 1 summarizes the datasets.

Table 1: The statistics of datasets.

	Dataset	$\|V\|$	$\|E\|$	Size
Real-world Graph Datasets	TWT	41.6 M	1.37 B	21.9 GB
	YH	1.4 B	6.18 B	98.9 GB
	CW09	4.8 B	7.39 B	118.3 GB
	CW12	6.3 B	66.8 B	1.06 TB
Synthetic Graph Datasets	RMAT_X ($24 \le X \le 38$)	2^{X-4}	2^X	256 MB ~ 4 TB

Queries: We use total five queries: PageRank (PR), Single Source Shortest Path (SSSP), Weakly Connected Components (WCC), Triangle Counting (TC) and Local Clustering Coefficient (LCC). For their analysis, we group them into two categories based on their time complexities. PR, SSSP, and WCC belong to the first group, while TC and LCC belong to the second group. We run PR for 3 iterations and report the average elapsed time per iteration. We run SSSP and WCC until convergence and report the total elapsed times. For SSSP, we use a vertex with the largest number of neighbors in the graph as the source vertex.

Competitors: We compare TurboGraph++ against the five state-of-the-art distributed graph analytics systems including both external-memory and in-memory systems to prove the scalability and efficiency of TurboGraph++. We use the two external-memory systems (Chaos [30] and HybridGraph [37]) and three in-memory systems (Gemini [44], Pregel+ [41], and GraphX [13]). Pregel+ is known to be the fastest in-memory graph analytics system [23]. GraphX is a graph analytics interface on top of Spark, a general distributed processing framework. For GraphX, we set its RDD persistence level to MEMORY_AND_DISK, so that when the system memory is full, it can spill RDDs to disks for processing large graphs.

For TC, due to the serious scalability problem of the aforementioned systems, we additionally compare against PTE [27]. Note that PTE achieves the worst case optimal cost for TC. We implement TC on Pregel+ and HybridGraph according to [12].

Table 2: Parameters used in the experiments.

Parameters	Description	Values used
d	data graph	RMAT_X (**RMAT_{35}**), TWT, YH, CW09, CW12
q	query	Group1: **PR**, SSSP, WCC Group2: **TC**, LCC
m	the number of machines	5, 10, 15, 20, **25**
disk	storage device	**PCIe SSD**, HDD

For a fair comparison of all systems including TurboGraph++, we modify HybridGraph, Pregel+, and Gemini so that they use the 64-bit vertex id representation. We select the best performing parameters for all competitors in our cluster.

Measure: We measure the query execution time and the total amount of bytes for disk I/O and network I/O aggregated over all machines. All in-memory systems load graph data into memory before the query processing, but we exclude their loading time in the elapsed time measurement. We use *dstat* for monitoring resource usage in each machine in the cluster. For disk I/O, we report the aggregated sum of bytes read and written. In order to measure the exact disk I/O during the query execution, we drop the OS page cache after the pre-processing phase of each system. For network I/O, we report the aggregated sum of bytes sent from a machine and received by another machine. We measure the maximum bandwidths of disk and network by using *fio* and *iperf*, respectively. We measure the CPU time by *clock_gettime()*. The timeout is set to 8 hours.

Running environments: We conduct experiments with 25 machines, interconnected by an InfiniBand QDR 4x network switch. Each machine has two Intel Xeon E5-2450 CPUs, each with 8 cores, and 32GB RAM (16GB per a NUMA node). We use two different clusters where all machines have the same hardware configurations except for storage devices. Every machine in one cluster is equipped with a PCIe SSD, while every machine in the other is equipped with 4 HDDs configured as a RAID-0 array with 256 KB stripe size. The maximum bandwidths of PCIe SSD and HDD are 1.5 GB/s and 300 MB/s, respectively.

Table 2 summarizes the parameter values used for the experiments. Unless otherwise noted, bold values are used as default parameter values in the experiments.

5.2 Evaluation of BBP and 3-LPO

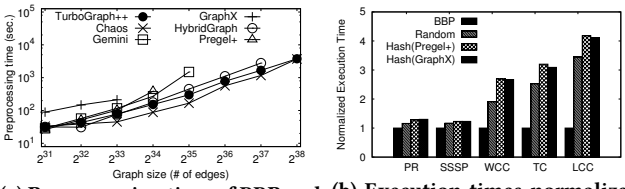
We first show that the BBP scheme achieves a reasonable preprocessing cost. We then evaluate our 3-LPO strategy to show that it achieves full parallelism and full overlap of CPU computation, disk I/O, and network I/O for varying hardware configurations.

5.2.1 Efficiency of BBP Scheme. We evaluate the preprocessing cost of BBP and the other systems. We measure the elapsed time of BBP in TurboGraph++ and compare it to the preprocessing time of the competitors for varying graph size (Figure 8 (a)). Note that the preprocessing times of external-memory systems include the time for writing the partitioned graph data back to disk. BBP takes an average of 1.39 times longer than the preprocessing time of Chaos. However, as the graph size increases, BBP takes a similar time as Chaos's preprocessing time at the largest graph. We observe that Chaos poorly overlaps CPU, disk, and network and fails to fully utilize the given hardware resources.

5.2.2 Effectiveness of BBP Scheme. We evaluate the effectiveness of the BBP scheme against a uniformly random partitioning (Random) and two hash partitioning schemes used in GraphX and Pregel+, denoted as Hash(GraphX) and Hash(Pregel+), respectively. We use the group1 and group2 queries. Figure 8 (b) shows relative execution times when we use these partitioning schemes, compared to the execution time when we use BBP.

For the group1 queries, BBP outperforms the Random, Hash(Pregel+), and Hash(GraphX) by 1.41, 1.73, and 1.73 times on average, respectively. The performance improvement for WCC is the most significant in the group1 queries, due to the imbalanced workloads.

For the group2 queries, BBP outperforms Random, Hash(Pregel+), and Hash(GraphX) by 3, 3.69, and 3.6 times on average, respectively. This is due to 1) the well-balanced workloads across machines and 2) faster set intersections using degree-order vertex IDs.



(a) Preprocessing time of BBP and other systems. **(b) Execution times normalized to the execution time with BBP.**

Figure 8: Evaluation of BBP.

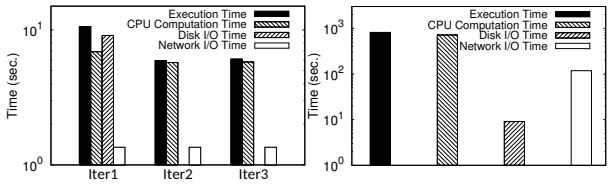
5.2.3 Evaluation of the 3-LPO Strategy. We show that 3-LPO achieves full parallelism and full overlap of CPU computation, disk I/O and network I/O. We first calculate CPU, disk I/O, and network I/O times for PR and TC. The CPU time is the average CPU clock time of all computation threads of all machines in a cluster. The disk I/O time and network I/O time are computed by dividing the total I/O bytes by the aggregated bandwidth of disk and network in the cluster, respectively. Then, we compare these times to the query execution time of TurboGraph++. We show that the query execution time is almost determined by the time taken for the most bounded resource with 3-LPO.

Figure 9 shows the elapsed times for CPU, disk, and network, respectively, when TurboGraph++ executes queries. For PR, we provide the results of three iterations. In the first iteration, the disk I/O time dominates the execution time. The disk is the most bounded resource since it needs to read the edge pages from disk. However, for the next two iterations, the CPU becomes the bounded resource since most of the edge pages are already loaded in memory. For TC, the CPU time dominates the query execution time, which shows that TC is computationally expensive and the CPU is the bounded resource. Here, regarding the overhead of enumerating a k-reachable walk set, we measure the CPU time spent for iterating the vertices in a one level window and identifying the vertices for a two level window. It is measured as 5.83 secs which is only 0.7% of the query time. Most of the CPU computations are used for the set intersections.

We conduct the same experiments with HDDs instead of a PCIe SSD. The results are shown in Figure 10. The overall tendencies for bounded resources are the same: the disk for the first iteration

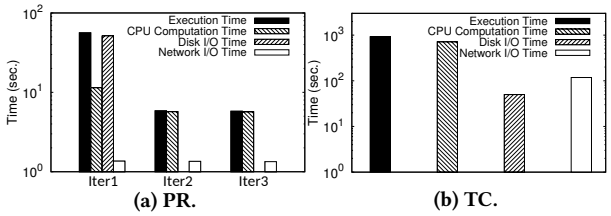
of PR and the CPU for the remaining two iterations of PR and TC. Figure 11 shows the resource usage over time while processing PR by varying disk device. It is clear that the bounded resource is the disk for the first iteration and the CPU for the next two iterations.

With our 3-LPO strategy, CPU computation, disk I/O, and network I/O are effectively parallelized and overlapped. We can almost determine the query execution time by the time taken for the most bounded resource. Note that the CPU time for enumerating a k-reachable walk set can easily be hidden by the compulsory disk and network I/O times thanks to our 3-LPO strategy.



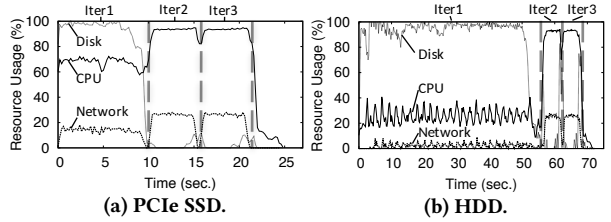
(a) PR. (b) TC.

Figure 9: Decomposed execution time with PCIe SSD.



(a) PR. (b) TC.

Figure 10: Decomposed execution time with HDD.



(a) PCIe SSD. (b) HDD.

Figure 11: Resource usage during processing PR on RMAT$_{35}$.

5.3 Overall Performance

We evaluate the overall performance of TurboGraph++ for all queries with four real graph data sets (TWT, YH, CW09, and CW12). Figures 12, 13, and 14 show the query execution times, disk I/O times, and network I/O times, respectively. We mark T, O, or F if the system fails due to time-out, out-of-memory, or for other reasons.

Figures 12 and 14 together show that the network is no longer bottlenecked. The utilized I/O bandwidths for network are much lower than the maximum bandwidth. Specifically, when executing PR, the average utilization is 5.4%, 2.7%, 7.2%, 0.4%, 8.0%, and 0.3% for TurboGraph++, Chaos, Gemini, HybridGraph, Pregel+, and GraphX, respectively. When executing TC, the average utilization is 13.5% and 2.9% for TurboGraph++ and PTE, respectively.

Group1 queries: Figures 12 (a), (b), and (c) show the query execution times for PR, SSSP, and WCC, respectively. Against the external-memory systems, TurboGraph++ consistently and significantly outperforms HybridGraph and Chaos. Specifically, TurboGraph++ is 44.6 and 163.9 times faster, on average, than HybridGraph and Chaos, respectively. The outstanding performance of TurboGraph++ comes from two key factors: (1) low disk and network I/O costs and (2) full overlap of computation and I/O by 3-LPO.

TurboGraph++ has the lowest disk I/O and network I/O for all cases. The lowest disk I/O is thanks to the effective buffer utilization of TurboGraph++. HybridGraph and Chaos rely on the OS page cache whose page replacement policy is not useful for graph access patterns, leading to the poor buffer utilization. The lowest network I/O is thanks to the in-memory local gather operations of the updates during the scatter phase.

Regarding the second factor, consider PR with CW12. Compared to HybridGraph and Chaos, TurboGraph++ has lower I/O cost by 4.68 and 8.24 times for disk and by 1.57 and 22.89 times for network, respectively. Beyond the lower I/O costs, TurboGraph++ achieves 50.38 and 42.18 times faster execution times by fully parallelizing and overlapping computation and I/O with 3-LPO.

For SSSP and WCC, TurboGraph++ achieves higher performance gaps against HybridGraph and Chaos than for PR. Even though SSSP and WCC need to access only a portion of vertices in a superstep, HybridGraph and Chaos have to access almost all vertices/edges, leading to high I/O cost and poor efficiency. Besides, for YH and CW12, SSSP requires 4373 and 954 supersteps, and WCC requires 1552 and 4117 supersteps, respectively. As a result, for YH and CW12, HybridGraph and Chaos take very long execution times or fail to process before the timeout.

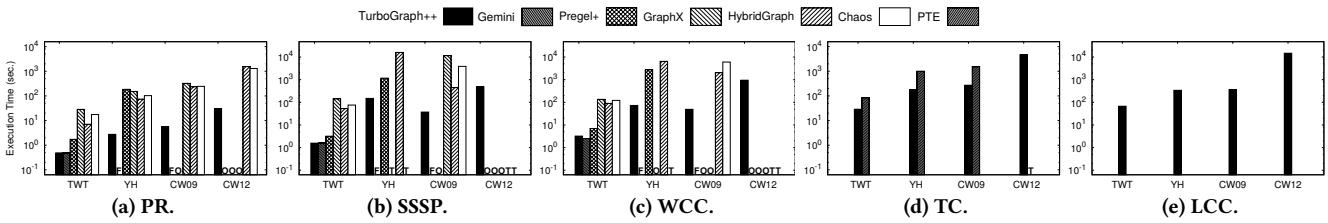
Against the in-memory systems, TurboGraph++ consistently outperforms Pregel+ and GraphX. Although TurboGraph++ requires disk I/O during processing, it is 20.1 and 102.7 times faster, on average, than Pregel+ and GraphX, respectively. TurboGraph++ even performs comparable to Gemini for all queries at TWT by fully overlapping the disk I/O with the CPU computation and network I/O. Gemini and Pregel+ fail to process graphs larger than TWT and YH, due to the crash during partitioning and the out-of-memory error, respectively. GraphX can process larger graphs than Pregel+ and Gemini due to its capability to persist RDDs. However, it still fails to process CW09 or CW12 due to the out-of-memory error.

Group2 queries: Figures 12 (d) and (e) show the query execution times for TC and LCC, respectively. Only TurboGraph++ among the graph analytics systems can process TC and LCC without crashing. All in-memory systems fail for all graphs due to the out-of-memory error. This out-of-memory error of the vertex-centric systems confirms the reports in [27, 29, 32, 43]. Note that LCC requires per-vertex triangle counting, which has a higher space and time complexity than TC. Only PTE can process up to CW09. However, PTE fails to process CW12 due to timeout.

Although TurboGraph++ has in some cases higher disk and network I/Os than PTE (Figure 13 (d) and 14 (d)), it consistently outperforms PTE in terms of execution time by up to 5.67 times by fully overlapping computation and I/O with the 3-LPO strategy. We observe that the computation of PTE is frequently blocked by the I/O, leading to poor resource utilization.

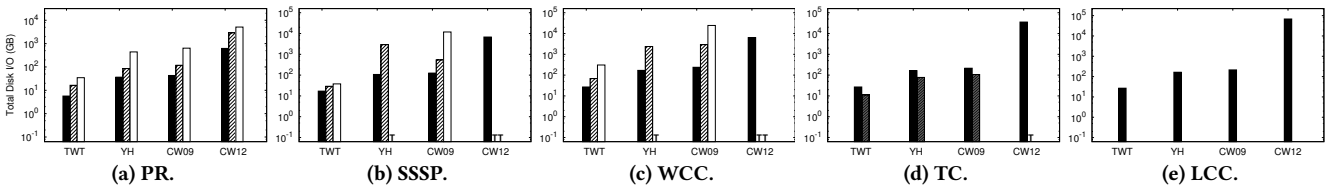
5.4 Scalability

We evaluate the scalability for processing large graphs with the given system resources by two varying scenarios. We first vary the size of RMAT graphs while fixing the number of machines. We then vary the number of machines while fixing the size of graphs.



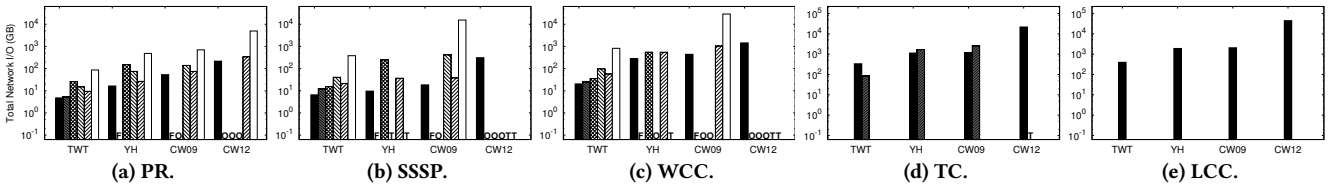
(a) PR. (b) SSSP. (c) WCC. (d) TC. (e) LCC.

Figure 12: Execution time on real graphs (PR, SSSP, WCC, TC, and LCC).



(a) PR. (b) SSSP. (c) WCC. (d) TC. (e) LCC.

Figure 13: Total disk I/O cost on real graphs (PR, SSSP, WCC, TC, and LCC).



(a) PR. (b) SSSP. (c) WCC. (d) TC. (e) LCC.

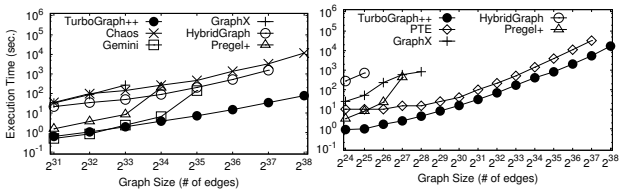
Figure 14: Total network I/O cost on real graphs (PR, SSSP, WCC, TC, and LCC).

5.4.1 Varying Graph Size. We show the scalability of TurboGraph++ for large graphs. Figure 15 shows query execution times for varying the size of graphs while fixing the number of machines.

Regarding PR, TurboGraph++ almost always outperforms all competitors. Specifically, it is faster than Pregel+, HybridGraph, GraphX, and Chaos at $RMAT_{31}$ by 3.70, 33.73, 55.84, and 55.29 times, respectively. As the graph size increases, TurboGraph++ starts to outperform even Gemini from $RMAT_{33}$. Note that Pregel+ and Gemini incur the disk swap at $RMAT_{34}$ and $RMAT_{35}$, respectively. HybridGraph fails to load $RMAT_{38}$ due to the out-of-memory error. The heap analysis with the *Eclipse memory analyzer* shows that its internal data structure, GraphDataServerDisk, takes up huge memory space holding adjacency lists while loading. Although Chaos can process up to the largest graph, it is significantly slower than the others due to the high I/O cost and poor overlap.

Regarding TC, TurboGraph++ consistently and significantly outperforms all competitor graph analytics systems. Specifically, TurboGraph++ is faster than Pregel+, GraphX, and HybridGraph at $RMAT_{24}$ by 3.60, 27.67, and 304.81 times, respectively. The execution time of Pregel+ increases rapidly from $RMAT_{24}$ due to the increasing network I/O for messaging. A further rapid increase occurs at $RMAT_{27}$ due to disk swap. In the vertex-centric model, TC requires the vertices to hold the messages that encode the two-level neighborhoods around them, leading to high I/O cost and memory usage. HybridGraph fails to process a small $RMAT_{26}$ with the out-of-memory error. This is because it keeps in memory the internal data structure called MessagePack, which packs the messages for batching the two-hop neighborhoods of vertices.

TurboGraph++ consistently outperforms PTE. The larger the graph size, the larger the performance gap becomes. Specifically, TurboGraph++ is 3.14 times faster than PTE at $RMAT_{31}$ but at $RMAT_{37}$, it outperforms PTE by 5.96 times. It indicates that fully parallelizing and overlapping the computation and I/Os is challenging but important to achieve both scalable and efficient processing of large graphs. PTE fails to process $RMAT_{38}$ due to timeout.



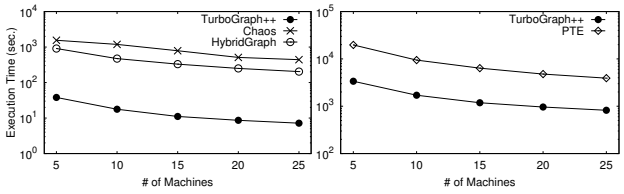
(a) PR. (b) TC.

Figure 15: Execution time of PR and TC for varying graph size.

5.4.2 Varying the Number of Machines. Now, we show the scalability of TurboGraph++ for varying the number of machines while fixing the size of graphs. We use two graphs: $RMAT_{35}$ and $RMAT_{33}$. For $RMAT_{35}$, we compare against the external-memory systems since none of the in-memory systems can process $RMAT_{35}$ with less than 25 machines. For $RMAT_{33}$, we compare against both external-memory and in-memory systems because it is the largest graph that both systems can process.

Figure 16 shows the query execution times on $RMAT_{35}$ for PR and TC, respectively. Regarding PR, TurboGraph++ consistently and significantly outperforms all the external-memory systems. TurboGraph++ also achieves near-linear speedup with a slope of 0.97. Chaos shows the longest execution time and poor speedup due to the high I/O cost and poor overlap. On average, TurboGraph++ outperforms HybridGraph and Chaos by 27.54 and 59.86 times, respectively.

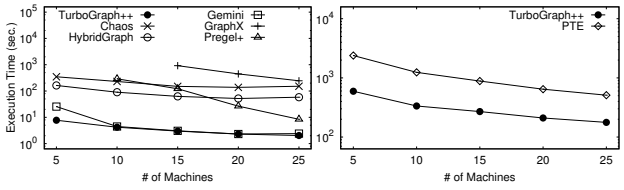
Regarding TC, TurboGraph++ always outperforms PTE. TurboGraph++ achieves scalability similar to that of PTE, showing near linear speedup as the number of machines increases. However, with its superior efficiency, TurboGraph++ with only 5 machines outperforms PTE with 25 machines. Even if we assume a perfect speedup of PTE, it may need more than 125 machines to perform comparable to TurboGraph++ with 25 machines, which illustrates the importance of efficiency as well as scalability.



(a) PR. (b) TC.

Figure 16: Execution time of PR and TC varying the number of machines ($RMAT_{35}$).

Figure 17 shows the query execution times on $RMAT_{33}$ for PR and TC, respectively. Regarding PR, TurboGraph++ consistently outperforms all the competitors. Gemini has a long execution time when the number of machines is 5 due to the disk swap. Pregel+ and GraphX involve the disk swap or I/O for all cases due to their high memory requirements. Note that the total size of the input graph is only about 128 GB while the aggregated memory capacity of 5 and 25 machines are 160 GB and 800 GB, respectively. Regarding TC, the overall tendency is similar to that of $RMAT_{35}$. TurboGraph++ always outperforms PTE, which depicts the poor efficiency of PTE.



(a) PR. (b) TC.

Figure 17: Execution time of PR and TC varying the number of machines ($RMAT_{33}$).

6 RELATED WORK

Graph Processing Model: Graph processing model directly affects efficiency of a system and the programmability for various graph algorithms. Both the vertex-centric model [22, 24, 37, 41, 44] and the edge-centric model [12, 30, 31] are widely studied. Some improvements [35, 38, 40] have also been made over the vertex-centric processing model. For example, Giraph++ [35] and Blogel [40] extend the vertex-centric model via an efficient block-centric approach that exposes the partitioning information for further optimizations, such as avoiding communication for computation within a block.

However, the existing processing models ignore neighborhood information around vertices. Thus, they are unsuited for graph analytics that require computations against the neighborhoods [16, 29]. NScale [29] presents a new neighborhood-centric processing model which provides users with the neighborhood information around vertices for ease of programming and higher efficiency. However, it exhibits inefficiencies in extracting subgraphs before processing [39] and high memory usage due to in-memory indexing on subgraphs during processing.

In this paper, we have presented a new graph processing abstraction called nested windowed streaming for scalable and efficient processing of neighborhood-centric analytics.

Graph Partitioning: Various graph partitioning approaches have been proposed [9, 12, 30, 44]. The edge-cut approach balances vertices and distributes their edges correspondingly. The vertex-cut [12] balances edges and distributes the corresponding source and destination vertices. The hybrid-cut [9] applies edge-cut for low-degree vertices and vertex-cut for high-degree vertices. One of the main objectives of the aforementioned methods is to minimize network communication costs. However, as [6] points out, in a modern cluster with a high-speed network, the network is no longer a bottleneck.

In this paper, we have proposed a balanced buffer-aware partitioning scheme in order to balance the workloads among machines and bound memory usage for processing neighborhood-centric analytics with reasonable preprocessing time.

Graph Analytics Systems: Many in-memory or external-memory graph analytics systems have been developed [9, 10, 13, 14, 19, 30, 35, 37, 38, 40, 41, 44, 45]. The single machine external-memory systems [14, 19, 45] have inherently limited scalability due to their computational power and storage capacity. The distributed in-memory systems cannot scale to large graphs, especially for neighborhood-centric analytics since the graphs containing the neighborhoods can grow to orders of magnitude size larger than the input graph. The distributed external-memory systems [30, 37] perform significantly slower than the state-of-the-art in-memory systems [41, 44] due to their poor hardware utilization and poor overlap of CPU computation, disk I/O, and network I/O.

In this paper, we have proposed a 3-LPO processing strategy for fully parallelized and overlapped CPU computation, disk I/O, and network I/O tasks in order to achieve both scalability and efficiency.

7 CONCLUSION

In this paper, we have presented TurboGraph++, a distributed graph analytics system without compromising scalability and efficiency. TurboGraph++ provides the nested windowed streaming model to efficiently process neighborhood-centric analytics with a fixed memory budget. Through balanced, buffer-aware partitioning, TurboGraph++ achieves both a reasonable partitioning cost and balanced workloads. The 3-LPO of TurboGraph++ has demonstrated three-level parallel and overlapped processing of CPU computation, disk I/O, and network I/O to fully utilize all hardware resources in a cluster.

Extensive experiments have shown that TurboGraph++ scales well to large graphs with a fixed memory budget, while it is also even faster than the-state-of-the-art distributed in-memory graph analytics systems. We have also shown that TurboGraph++ outperforms even PTE, the-state-of-the-art distributed method for triangle counting, with our nested windowed streaming processing.

ACKNOWLEDGMENT

We are very grateful to TaeSung Lee, Inhyuk Na, and Hyeonji Kim for many stimulating discussions and parts of implementation of the network communication layer and graph partitioning method. We also appreciate the anonymous reviewers for their valuable and insightful comments. This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1401-04.

REFERENCES

[1] 2002. *The Yahoo Webscope Program: yahoo! altavista web page hyperlink connectivity graph.* https://webscope.sandbox.yahoo.com/.
[2] 2009. *The lemur project: Clueweb09 web graph.* http://www.lemurproject.org/clueweb09.
[3] 2012. *The lemur project: Clueweb12 web graph.* http://www.lemurproject.org/clueweb12.
[4] 2012. *MPI: A Message-Passing Interface Standard Version 3.0.* http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.
[5] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. 2013. Enumerating subgraph instances using map-reduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 62–73.
[6] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: It's time for a redesign. *Proceedings of the VLDB Endowment* 9, 7 (2016), 528–539.
[7] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. 1976. *Graph theory with applications*. Vol. 290. Citeseer.
[8] 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, 442–446.
[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 1.
[10] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.
[11] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 495–510.
[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] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI*, Vol. 14. 599–613.
[14] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 77–85.
[15] Nilesh Jain, Guangdeng Liao, and Theodore L Willke. 2013. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 4.
[16] Arijit Khan. 2017. Vertex-Centric Graph Processing: The Good, the Bad, and the Ugly. *Proceedings of the 20th International Conference on Extending Database Technology* (2017).
[17] Ajay D Kshemkalyani and Mukesh Singhal. 2011. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press.
[18] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. ACM, 591–600.
[19] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. USENIX.
[20] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.
[21] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science* 407, 1-3 (2008), 458–473.
[22] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.
[23] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment* 8, 3 (2014), 281–292.
[24] 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.
[25] R Meusel, O Lehmberg, C Bizer, and S Vigna. 2014. *Web data commons-hyperlink graphs.* http://webdatacommons.org/hyperlinkgraph/.
[26] Himchan Park and Min-Soo Kim. 2017. TrillionG: A Trillion-scale Synthetic Graph Generator using a Recursive Vector Model. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 913–928.
[27] Ha-Myung Park, Sung-Hyon Myaeng, and U Kang. 2016. Pte: Enumerating trillion triangles on distributed systems. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1115–1124.
[28] Todd Plantenga. 2013. Inexact subgraph isomorphism in MapReduce. *J. Parallel and Distrib. Comput.* 73, 2 (2013), 164–175.
[29] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2014. NScale: neighborhood-centric analytics on large graphs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1673–1676.
[30] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 410–424.
[31] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.
[32] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. 2013. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1906–1917.
[33] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 625–636.
[34] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1222–1230.
[35] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.
[36] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*. ACM, 333–342.
[37] Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, and Jeffrey Xu Yu. 2016. Hybrid Pulling/Pushing for I/O-Efficient Distributed and Iterative Graph Computing. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 479–494.
[38] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. 2013. Fast iterative graph computation with block updates. *Proceedings of the VLDB Endowment* 6, 14 (2013), 2014–2025.
[39] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, and James Cheng. 2016. Big graph analytics systems. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2241–2243.
[40] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.
[41] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*. ACM, 1307–1317.
[42] Makoto Yui, Jun Miyazaki, Shunsuke Uemura, and Hayato Yamana. 2010. Nb-GCLOCK: A non-blocking buffer management based on the generalized CLOCK. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 745–756.
[43] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. 2014. MOCgraph: Scalable distributed graph processing using message online computing. *Proceedings of the VLDB Endowment* 8, 4 (2014), 377–388.
[44] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA*.
[45] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning.. In *USENIX Annual Technical Conference*. 375–386.

A APPENDIX

A.1 Programming examples of NWSM

Figure 18 shows the implementation of PageRank. For a vertex u, $u.pr$ and $u.degree$ are the current pagerank and degree values, respectively. $u.pr\_tmp$ stores the accumulated updates. For an update upd, $upd.value$ is the pagerank score that a source vertex contributes to a destination vertex over each edge. The program first initializes k as one and allocates voi accordingly (Line 5-6). Then, the user-defined functions are registered (Line 8-11). The PR_Scatter (Line 24-27), the scatter function at the 1-st stream, generates the updates from a vertex u to its neighbors. The PR_Gather (Line 29-31), the gather function, gathers the produced updates and accumulates the $upd.value$ onto pr_tmp of each target vertex. The PR_Apply (Line 33-36), the apply function, recomputes the pagerank value $u.pr$ based on the accumulated update on $u.pr\_tmp$. The ProcessVertices (Line 14-16) iterates over vertices and initializes their pagerank values. After that, Start is called to execute the nested windowed streaming processing (Line 18-21).

Figure 19 shows the implementation of triangle counting. The implementation uses the two-hop neighborhood processing. It loads the two-hop neighborhood of vertices into the two nested windows and counts the number of triangles by finding the common neighbors of the two connected vertices. The program first initializes k as two and allocates voi accordingly (Line 2-4). Then, the user-defined functions are registered (Line 6-8). The EnumOneHopNbr (Line 17-20), the scatter function for the 1-st stream, identifies the one-hop neighbors of u for enumeration. The FindTriangles (Line 22-28), the scatter function for the 2-nd stream, counts the number of the common neighbors of a vertex u in the 2-nd window and a vertex v in the 1-st window connected to u. We provide the GetCommonNbrList(u,v) iterator to filter the common neighbors of u and v as a system primitive that performs the fast intersection of their adjacency lists.

A.2 Proof of Theorem 1

Proof. At each level of TBGPPApp, we need to have memory spaces for vw^l, LGB^l, $adjw^l$, and voi^l. In addition, we have a global gather buffer, GGB. Their memory sizes are denoted by $|vw^l|$, $|LGB^l|$, $|adjw^l|$, $|voi^l|$, and $|GGB|$, respectively. Then, the total memory usage is given as follows:

$$\sum_{l=1}^{k} \left(\left|vw^l\right| + \left|LGB^l\right| + \left|adjw^l\right| + \left|voi^l\right| \right) + |GGB| \qquad (2)$$

The minimum memory requirement for each data structure can be computed as in Equations 3. $|VA|$ stands for the total number of bytes of vertex attributes. $|V|$ is the total number of vertices in the graph. $|voi^l|$ is the size of the vertex Id set which we implement using a bitmap data structure.

```
1  // Schema
2  // - vertex: {degree:int, pr:double, pr_tmp: double}
3  // - update: {dst:int, value:double}
4  PageRank::Run() {
5     k = 1;
6     voi = new VertexIdSet[k];
7
8     adj_scatter = new ScatterFunction[k];
9     adj_scatter[1] = &PR_Scatter;
10    vertex_gather = &PR_Gather;
11    vertex_apply = &PR_Apply;
12
13    // initialize PageRank values
14    ProcessVertices([&] (Vertex u) {
15         u.pr = (u.degree>0) ? (1/u.degree) : 1;
16    });
17
18    foreach(superstep: [1, MaxSuperstep]) {
19         voi[1].MarkAll(); // visit all vertices
20         Start();
21    }
22 }
23
24 PageRank::PR_Scatter(Vertex u, Iter iter) {
25     foreach (Vertex v: iter)
26        v.Update(u.pr / u.degree); // generate updates
27 }
28
29 PageRank::PR_Gather(Vertex u, Update upd) {
30     u.pr_tmp += upd.value; //aggregates updates on u.
           pr_tmp
31 }
32
33 PageRank::PR_Apply(Vertex u) {
34     u.pr = 0.85 + 0.15 * u.pr_tmp; //recomputes u.pr
           using u.pr_tmp
35     u.pr_tmp = 0;
36 }
```

Figure 18: Programming example (PageRank).

```
1  TriangleCounting::Run () {
2     k = 2;
3     voi = new VertexIdSet[k];
4     tri_cnts = 0;
5
6     adj_scatter = new ScatterFunction[k];
7     adj_scatter[1] = &EnumOneHopNbr;
8     adj_scatter[2] = &FindTriangles;
9
10    voi[1].MarkAll();
11    Start();
12
13    ReduceSum(tri_cnts);
14    Print("Triangles: %ld\n", tri_cnts);
15 }
16
17 TriangleCounting::EnumOneHopNbr(Vertex u, Iter iter) {
18     foreach (Vertex v: iter)
19         if (CheckPartialOrder(u,v)) voi[2].Mark(v);
20 }
21
22 TriangleCounting::FindTriangles(Vertex v, Iter iter) {
23     foreach (Vertex u: GetParentList(1, v)) {
24         if (!CheckPartialOrder(u,v)) continue;
25         foreach (Vertex w: GetCommonNbrList(u,v)))
26             if (CheckPartialOrder(v,w)) tri_cnts++;
27     }
28 }
```

Figure 19: Programing example (Triangle Counting).

$$\left|vw^l\right| = 2 * \frac{|VA|}{p*q} \quad for\ all\ l \in [1,k]$$
$$\left|LGB^l\right| = 2 * \frac{|VA|}{p*q} \quad for\ all\ l \in [1,k]$$
$$\left|adjw^l\right| = 2 * PS \quad \text{where PS means page size } for\ all\ l \in [1,k]$$
$$\left|voi^l\right| = \alpha * |VA| \quad \text{where } \alpha = \frac{1}{8 * |VA|/|V|} \ for\ all\ l \in [1,k]$$
$$|GGB| = \frac{|VA|}{p*q} \tag{3}$$

Now, we can compute the total minimum memory requirement from Equation 2 and Equation 3 as in Equation 4. In order to avoid the out-of-memory error during processing, the total memory budget $|M|_{total}$ must be larger than the total memory requirement, $|M|_{min}$. We compute the minimum number of partitions for q in BBP from an inequality $|M|_{total} \geq |M|_{min}$. Now, we have q_{min} as in Equation 1.

$$|M|_{min} = \sum_{l=1}^{k} \left(\frac{4|VA|}{p*q} + 2PS + \alpha|VA| \right) + \frac{|VA|}{p*q} \tag{4}$$

□

Depending on applications, some memory spaces are not used during execution. In this situation, we do not allocate memory for those spaces as an optimization; 1) when the number of vertex chunks is one and there is no need for double buffering, and 2) there are no output vectors.

A.3 Implementation

We explain selected details on implementation of TurboGraph++. TurboGraph++ is implemented in C++ with more than 10,000 lines of code.

Graph Representation in Disk: We store a vertex attribute as a binary array in the disk. The edges in each edge sub-chunk are stored in a list of slotted pages where the page size is 64 KB by default. Each page consists of forward growing *records* and backward growing *slots*. The record stores the edges of a vertex in adjacency list format. The slot is a pair of a source vertex IDs and the offset to its record. For indexing the disk pages containing the edges, we exploit a two-level index on the edge pages by the range of destination vertices at the first level and by the range of source vertices at the second level.

Adjacency List Materialization: We provide two different processing modes of adjacency lists. For the partial list mode, we can directly use the adjacency lists stored in each edge page. However, for the full list mode, we need to materialize the full adjacency lists. Given the vertices of interest, the adjacency lists of the vertices in a consecutive range are always materialized in batch to maximize the sequential disk I/O. We first identify the necessary edge pages and issue page I/Os in ascending order of page IDs for sequential reads. We merge the loaded partial adjacency lists in the edge pages with their source vertex IDs as keys to the full adjacency lists.

Buffer Management: In order to maximize the overall performance in the external-memory system, effective buffer management is crucial. Regarding edge pages, we maintain an edge page buffer that consists of the page-sized memory frames. When a page I/O is issued and the page is already in the buffer, it simply pins the page in memory and does not incur disk I/O. Otherwise, it first finds an available frame and issues the I/O. For efficient look-up of pages in the buffer frames, we use an array-based page table as in the operating system. For efficient page replacement, we use a variant of the non-blocking clock replacement algorithm in [42]. In order to avoid the sequential flooding problem, at the beginning of each superstep, we pre-pin those edge pages that are already in memory, process them first, and then unpin them. Since the size of the edge buffer is constant, when we calculate q, we subtract the edge buffer size from the total memory size.

NUMA-aware Sub-chunk Scheduling: In order to reduce the CAS operations across NUMA nodes, we perform NUMA-aware scheduling in the partial list mode by exploiting the edge sub-chunk partitioning of BBP. Note that the edge sub-chunks store the edges with the disjoint range of destination vertices. By scheduling the threads on the same NUMA node to the same edge sub-chunk, we reduce the synchronization cost across NUMA nodes for performing the in-memory local gather on the local gather buffer. For load balancing between NUMA nodes, each thread can steal the work of the other NUMA nodes rather than being idle after it finishes the work of its own NUMA node.

Fault Tolerance: We support fault-tolerance by checkpointing [24]. At the end of each superstep, the vertex attribute data are made permanent to disk. A failure can be recovered by rolling back to the latest available checkpoint and restarting the computation. Thus, the cost for failure handling is the disk I/O for flushing the vertex attribute data to disks.

Reliable communication layer: Reliable communication is an important issue in a distributed environment. We use both a library of Message Passing Interface (MPI) [4] and TCP/IP socket communication for reliable communication [17]. The MPI has a reliable communication interface as its design objective [4]. More efficient and reliable communication is beyond our scope.

A.4 Usability Case Study

We have hired three undergraduate research interns and two graduate students who do not have any experience in distributed graph engines and asked them to develop the triangle counting application on Apache Giraph and TurboGraph++. The students found that our model is intuitive and easy to understand.

Regarding the easiness of use of the programming APIs, they were asked to score each system on a scale 1 to 10 (10 being the highest score). The average score for TurboGraph++ is 8, while the average score for Giraph is 5.8. The participants found that 1) they could not directly apply well-known algorithms to triangle counting, but instead had to come up with a new algorithm using the message passing, 2) the vertex-centric model is simple but it is difficult to program complex queries including triangle counting. 3) the participants have to consider the memory usage of the system during the processing in order to avoid the out-of-memory error.

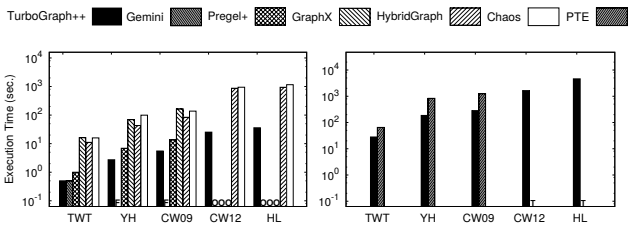
A.5 Additional Experimental Results

A.5.1 Experimental results with larger memory machines. The purpose of this additional experiment set is to compare TurboGraph++ against competitors on larger memory machines. Each machine in this experiment is equipped with 64GB RAM. We use PR and TC queries for evaluation. The overall performance trends were similar to the previous experiments with 32GB RAM machines. We highlight the differences.

Real Graphs: In order to show performance gains on a real large graph with more than 100B edges, we add a hyperlink graph (HL) [25] with 3.3B vertices and 119B edges. Figure 20 shows the query execution times of PR and TC.

For PR, Gemini still fails to process YH and CW09 due to crashing during partitioning even with 64GB RAM machines. Pregel+ processes CW09 without the out-of-memory error. Even with 64GB RAM (two times larger RAM size), GraphX still fails to process large real graphs, CW12 and HL. HybridGraph and Chaos can process HL without the out-of-memory error but TurboGraph++ significantly outperforms HybridGraph and Chaos by 29.53 and 36.5 times, respectively.

For TC, TurboGraph++ process all graphs while all in-memory systems still fail for all graphs due to the out-of-memory error. PTE still fails to process CW12 due to timeout.



(a) PR. (b) TC.

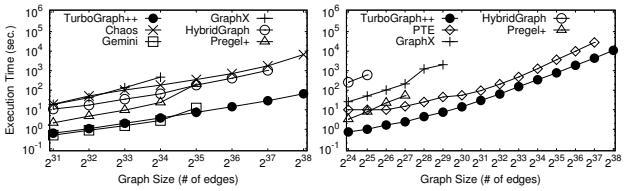
Figure 20: Execution time of PR and TC on real graphs with 64GB RAM machines.

Data Scalability: We compare the data scalability of the systems with the RMAT graphs. Figure 21 shows the query execution times for PR and TC.

For PR, Gemini still fails to process $RMAT_{36}$ but it incurs a significantly smaller number of disk swaps at $RMAT_{35}$ with 64GB RAM machines. Pregel+ and GraphX process two times larger graphs due to 64GB RAM, but TurboGraph++ outperforms them by up to 28.3 and 123.3 times. HybridGraph still fails to process $RMAT_{38}$.

For TC, PTE still fails to process $RMAT_{38}$ due to the timeout. GraphX can now process a two times larger graph up to $RMAT_{29}$. Pregel+ becomes much faster in $RMAT_{27}$. Now, it is 21.6 times slower than TurboGraph++.

A.5.2 Comparisons against Giraph. The purpose of this experiment set is to compare TurboGraph++ against out-of-core Giraph, since [10] claims to process trillion-scale graphs with the out-of-core Graph. We also tune the out-of-core Giraph by varying the number of partitions in memory and selecting the best parameter for each graph. Figure 22 shows the execution times of the TurboGraph++ and Giraph.



(a) PR. (b) TC.

Figure 21: Execution time of PR and TC for varying graph size with 64GB RAM machines.

Despite the out-of-core capability, when executing PR, Giraph incurs the out-of-memory error for graphs larger than $RMAT_{35}$. Note that [10] uses 200 machines rather than 25 machine, and the memory size of each machine is unknown. Furthermore, when executing TC, out-of-core Giraph incurs the out-of-memory error for all graphs. For those data sets that Giraph successfully processes, TurboGraph++ outperforms Giraph by up to 22.2 times for PR.

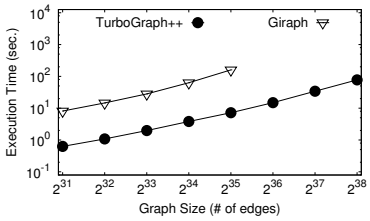


Figure 22: Execution time of PR for varying graph size.

A.6 General Subgraph Matching

In TurboGraph++, general subgraph matching can be readily extended from triangle counting by considering the following three factors only. Specifically, in terms of lines of pseudo code, the number of lines for triangle counting is 29, while the number of lines for general subgraph matching is 55.

First, we relax the condition for defining the adjacency list stream at level $(l + 1)$. Until now, for ease of understanding, we assume that the adjacency list stream at level $(l + 1)$ is derived from the adjacency list stream at level l. We relax this condition so that the adjacency list stream at level $l + 1$ can be defined from streams at level j (where $j < l + 1$).

Second, k is set to the number of vertices in the complement of a maximal independent set for a query graph. An independent set is a set of vertices in a graph such that no two of which are adjacent [7]. A maximal independent set is an independent set that is not a subset of any other independent set. For example, in triangle counting, k is set to 2 since any independent set for a triangle-shaped query has a single vertex, and thus the complement set has two vertices.

Lastly, users need to execute n-way set intersection supported by TurboGraph++. When programming the triangle counting query in Figure 19, 2-way set intersection is used.

On the other hand, in the vertex-centric programming model, users need to implement a complicated mechanism to pass messages which encode multi-hop neighbors of a vertex. Even if the users manage to implement a general subgraph matching method using the vertex-centric programming model, it is bound to crash due to the excessive amount of memory required for representing multi-hop neighbors in each vertex.