Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs

Prashant Pandey
ppandey@berkeley.edu
Lawrence Berkeley National Lab and
University of California Berkeley

Brian Wheatman
wheatman@cs.jhu.edu
Johns Hopkins University

Helen Xu
hjxu@mit.edu
Massachusetts Institute of Technology

Aydin Buluc
abuluc@lbl.gov
Lawrence Berkeley National Lab and
University of California Berkeley

ABSTRACT

Various applications model problems as streaming graphs and need to quickly apply a stream of updates and run algorithms on the updated graph. Furthermore, many dynamic real-world graphs, such as social networks, follow a skewed distribution of vertex degrees, where there are a few high-degree vertices and many low-degree vertices.

Existing static graph-processing systems achieve high performance and low space usage by exploiting graph skewness via preprocessing a cache-efficient graph partitioning based on vertex degree. In the streaming setting, the whole graph is not available upfront, however, so finding an optimal partitioning is not feasible in the presence of updates. As a result, existing streaming graph processing systems take a "one-size-fits-all" approach, leaving performance on the table.

We present Terrace, a system for streaming graphs that uses a hierarchical data structure design to store a vertex's neighbors in different data structures depending on the degree of the vertex. This multi-level structure enables Terrace to dynamically partition vertices based on their degrees and adapt to skewness in the underlying graph.

Our experiments show that Terrace supports faster batch insertions for batch sizes up to 1M when compared to Aspen, a state-of-the-art graph streaming system. On graph query algorithms, Terrace is between 1.7×–2.6× faster than Aspen and between 0.5×–1.3× as fast as Ligra, a state-of-the-art static graph-processing system.

CCS CONCEPTS

• **Theory of computation** → **Dynamic graph algorithms**; **Data structures design and analysis**.

KEYWORDS

Graph data structures; Streaming; Indexing

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457313

ACM Reference Format:
Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457313

1 INTRODUCTION

Many real-world sparse graphs, such as social networks or road networks, change over time. Therefore, systems for storing and processing dynamic (i.e. streaming) graphs [17, 28, 32, 34, 38, 47, 55] have been designed to process a stream of updates (e.g., edge weight update, or edge insertions and deletions) and a stream of queries quickly. That is, both query processing time and the time that it takes for updates to appear in the graph representation must be fast.

The ability to quickly apply a batch of updates is critical for efficient streaming graph processing. For example, in incremental triangle counting, insertion (or deletion) time accounted between 25% – 90% of the overall time [56]. Similarly, on 32 cores, updating the graph takes up to 90% of the overall running time in incremental connected components [58]. In this paper, we focus on data structure design for dynamic graph processing for both efficient updates and queries.

In practice, dynamic real-world graphs follow skewed vertex degree distributions as shown in Table 1. For example, real-world graphs, such as those from social networks [33, 62] or computational biology, contain a few very high-degree vertices and many low-degree vertices. This skewness presents unique challenges for efficiently representing dynamic graphs. However, these diverse distributions also present an opportunity to build cache-efficient graph representations via adaptive data structures that take advantage of degree distributions.

Existing static graph-processing systems that optimize for skewness demonstrate the potential for improved cache locality. For example, PowerLyra [21] partitions vertices based on their degree to improve locality of vertex computations. Other frameworks preprocess the graph into cache-friendly formats to improve locality. For example, Cagra [91] uses segmenting to divide the graph into cache-friendly subgraphs. Similarly, Gridgraph [92] partitions vertices and edges into blocks for locality. These techniques greatly improve locality in computations on static graphs, but do not easily translate to graphs that evolve over time. GPU-based static graph

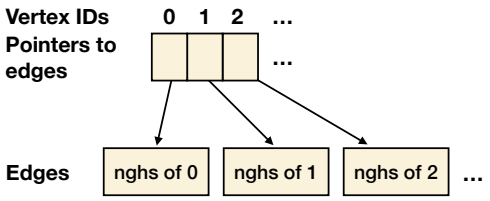


Figure 1: A high-level design for graph storage formats. There is a vertex structure that keeps track of where the neighbors (nghs) for each vertex are stored, and a structure for each vertex's edges.

processing systems also exploit the skewness to support fast graph algorithms and use the vertex's degree to decide which scheduler to use to run iterations [36, 52, 65].

In contrast, many existing dynamic graph-processing systems take a "one-size-fits-all" approach to data structure design, leaving performance on the table when processing and updating skewed graphs. Figure 1 illustrates a classical design for a graph storage format: a list of pointers (one for each vertex) to preselected data structures holding each vertex's neighbors (nghs). For example, the canonical static Compressed Sparse Row [81] (CSR) format stores a list of offsets into an edge list. Dynamic graph systems adopt a similar two-level design: Stinger [32] stores neighbors in a variant of a blocked adjacency list, while Aspen [28] stores each vertex's neighbors in a separate probabilistic balanced tree (C-tree). Since the neighbor data structures can only be accessed after a memory indirection, these dynamic systems must incur at least two cache misses per vertex during a graph traversal. Moreover, in tree-based representations such as Aspen, traversing a vertex's neighbors requires non-sequential memory accesses, which are slower than sequential memory accesses in array-based representations such as CSR.

The ideal structure for storing a vertex's neighbors in a dynamic graph framework depends on the access pattern of graph algorithms and the cost of doing updates. If a vertex has low degree, a simple data structure such as an array incurs minimal indirection and supports efficient traversal and updates. If a vertex has high degree, however, a more complex data structure such as a tree with better asymptotic search and update performance may be more suitable. Even though a balanced tree may have asymptotically better performance than an array, in the context of storing a vertex's neighbors, these data structures have crossover points in their performance depending on the degree of the vertex.

Characterizing graph skewness. Table 1 presents the distribution of vertex degrees in three different real-world graphs that exhibit skewness. These graphs are picked from three different domains.

Graph	% <10 Nghs	% <100 Nghs	% <1000 Nghs
LiveJournal	65	97.2	99.98
Twitter	64.56	95.39	99.51
Protein	30.47	61.49	98.80

Table 1: Distribution of degree of vertices in three different real-world graphs. Columns show the % of vertices that have less than 10, 100, 1000 neighbors (nghs). The maximum degree in the graphs are: LiveJournal (20333), Twitter (2997487), and Protein (3779).

A major fraction of all the vertices in these graphs have less than 10 neighbors which can be easily packed in a single cache line along with other meta information about the vertex, e.g., the vertex degree. However, there is also high variance between degree of vertices: the maximum degree in these graphs goes up to 2.99 Million (Twitter graph). Therefore, the high-degree vertices must be stored in a sophisticated structure to enable efficient updates and queries. These data about real-world graphs show that we must treat low- and high-degree vertices differently to achieve better cache locality and good performance.

Exploiting skewness in streaming graphs. We introduce Terrace, a dynamic graph-processing framework, that exploits skewness present in real-world graphs to build a cache-optimized representation. The main idea behind Terrace is a hierarchical data structure design that stores a vertex's incident edges in different data structures based on its degree. That is, a vertex's degree determines what type of data structure its edges will be stored in. The hierarchical design and degree cutoffs can be adapted to the distribution of a particular graph for improved performance and space usage.

A key insight behind Terrace is that neighbors of low-degree vertices can be stored *in place* rather than in a separate data structure, reducing latency and improving locality. That is, a few neighbors of each vertex can be stored directly in the vertex structure. Storing neighbors in-place in the vertex structure avoids cache misses for low-degree vertices during a graph traversal because it avoids following pointers for low-degree vertices.

At a high level, Terrace stores edges in three main types of data structures: a sorted array that stores a few neighbors per vertex in place, a shared Packed Memory Array [11, 41] (PMA) that compactly stores neighbors of medium-degree vertices, and per-vertex B-trees [23, Chapter 18] for high-degree vertices. The PMA and B-tree are cache-efficient structures with asymptotically better update and query costs than traditional packed lists.

Cache miss analysis. Existing static and dynamic graph-processing systems incur a high number of cache misses during graph kernels because they use a uniform out-of-place per-vertex structure regardless of vertex degree. To verify our hypothesis, we measured[1] the number of cache misses during graph kernels in Ligra [75] and Aspen [28], two state-of-the-art graph processing systems, as well as in Terrace, and report the results in Table 2. We picked breadth-first search and PageRank [89] as these two kernels have distinct access patterns and can be used as representatives for access patterns in other graph kernels. Ligra is a static graph framework that stores its edges in CSR format, while Aspen supports dynamic graphs using compressed trees. Both Ligra and Aspen incur more cache misses than Terrace because they require indirection to access neighbors for all vertices, while Terrace stores neighbors of low-degree vertices in place. The improved locality in Terrace translates into graph kernel performance: Figure 3 summarizes the results of our evaluation.

Contributions

To be specific, our contributions are as follows:

[1] We measured the number of cache misses using the `perf` utility in Linux. To compute the average number of cache misses we measure the total cache misses for 1, 10, 20, 100 rounds of kernel runs and then averaged it for a single run.

Kernel	Ligra	Aspen	Terrace
BFS	3.5M	6.3M	1.1M
PR	174M	197M	128M

Table 2: Average cache misses in breadth-first search (BFS) and PageRank (PR) on the LiveJournal graph over 100 rounds. Cache misses are higher in PR as it was run for 10 iterations compared to a single iteration in BFS.

- The design of a dynamic graph-processing system using hierarchical data structures for improved locality.
- An implementation of Terrace, a graph-processing system using the hierarchical design in Cilk [40].
- An experimental study of Terrace compared to Aspen [28] and Ligra [75], two state-of-the-art graph processing frameworks, that demonstrates that Terrace supports faster updates and queries.

The goal of this paper is to demonstrate how to organize vertex neighbors dynamically in a hierarchical way rather than in a "one-size-fits-all" framework. Although the idea of handling low- and high-degree vertices separately has been introduced in the static setting, this work takes the first step in hierarchical processing for the dynamic setting. Therefore, one of the main contributions is the multilevel design of Terrace and the characterization of desirable data structure properties at each level rather than a new data structure. The simplicity of the design of Terrace is its strength.

In terms of evaluation, we compare Aspen and Terrace on update throughput, and all systems on graph kernel performance. There is an extension of Ligra, called Ligra+ [76], that adds compression on top of the regular graph representation in Ligra. On the graphs that we tested, Ligra+ was slower than Ligra although more space-efficient, so we only include the results for Ligra.

Our implementation of Terrace extends the interface proposed by Ligra [75] with functionality for updating the graph. Therefore, all algorithms implemented with Ligra and Aspen, such as graph-traversal algorithms, local graph algorithms [77], and others [26, 27] can be run on top of Terrace with minor cosmetic changes.

Figure 2 shows that Terrace achieves up to 48 million updates per second and supports faster batch insertions (between 1.1×–3.1×) for batch sizes up to 1M when compared to Aspen. Table 6 contains the full results of batch insertions and deletions in Terrace and Aspen. Figure 3 shows that Terrace performs the shared graph kernels 1.7×–2.6× faster than Aspen and up to 1.3× faster than Ligra. On the kernels that do not have implementations in Aspen, Terrace is about 1.6× slower than Ligra.

Terrace overcomes traditional tradeoffs between fast updates and locality of graph computations. Existing state-of-the-art systems lie on either end of the spectrum: for example, Ligra is a static system and faster for vertex computations, while Aspen is dynamic but slower for vertex computations. Terrace shows how to support updates as fast as Aspen while being faster or similar to Ligra for vertex computations.

2 PRELIMINARIES

In this section, we formally define graphs and introduce the external-memory model for cache analysis that we will use to analyze the theoretical performance of different graph-processing

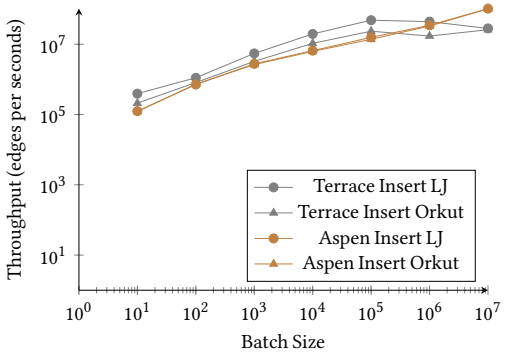


Figure 2: Batch insert throughput in Aspen and Terrace as a function of batch size on the LJ and Orkut graphs. The LJ graph has about 85 million edges, while the Orkut graph has about 234 million edges.

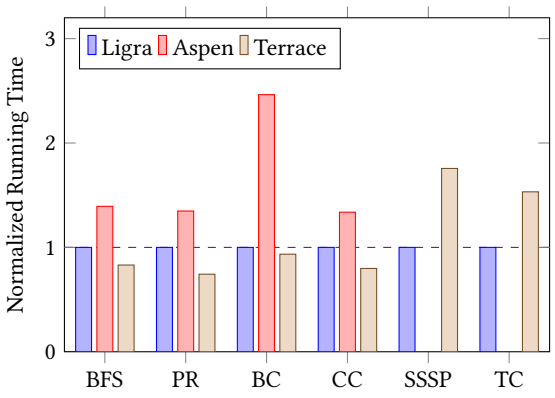


Figure 3: Average time to run kernels across all graphs in Ligra, Aspen, and Terrace normalized to Ligra. The four kernels tested for all systems were breadth-first search (BFS), PageRank (PR), single-source betweenness centrality (BC), and connected components (CC). Aspen does not have publicly available implementations of single-source shortest paths (SSSP) or triangle counting (TC), so we omit it from SSSP and TC.

systems. Finally, we will review the Packed Memory Array (PMA) data structure that underlies Terrace.

Graph preliminaries. A graph is a way of storing objects as *vertices* and connections between those objects as *edges*.

Definition 1 (Graph). *A graph $G=(V,E,w)$ is a set of vertices V, a set of edges E, and an edge weight function w. We denote the number of vertices with $|V|$, the number of edges with $|E|$, and the degree[2] of a vertex $v \in V$ with $deg(v)$. Each vertex $v \in V$ is represented by a unique non-negative integer less than $|V|$ (i.e. $v \in \{0,1,\ldots,|V|-1\}$). Each edge is a 2-tuple (u,v) where $u,v \in V$. Finally, the weight function w maps each edge $e \in E$ to a non-zero real weight ($w(e) \in \mathbb{R}, w(e) \neq 0$).*

External-memory model. The *external-memory model* given by Aggarwal and Vitter [3] represents two levels of memory: a small

[2]In this paper we focus only on directed graphs and use degree to mean out-degree. An undirected graph can be represented by a directed graph with edges in both directions.

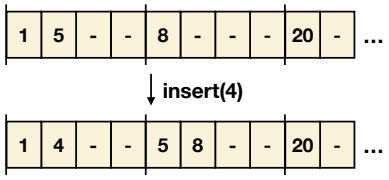


Figure 4: An example of inserting into a PMA with a leaf size of 4 and a leaf upper density bound of 0.5.

fast cache of bounded size and an arbitrarily large slow memory. Performance is measured in terms of the number of cache-line transfers, which bring B consecutive elements to cache from memory. For example, scanning a list of N elements takes $O(N/B)$ transfers.

B-trees generalize balanced binary trees to work well in the external-memory model and are widely used in databases [23, Chapter 18]. In this paper, we consider B-trees with node size (fanout) $\Theta(B)$, where B is the cache-line size from the external-memory model. A B-tree on N elements takes $O(N)$ space and supports updates and point queries in $O(\log_B N)$ transfers. Furthermore, a B-tree on N elements supports range queries in $O(\log_B N + k/B)$ transfers, where k is the number of elements in the query range. B-trees support fast updates but are slower to traverse than array-based structures because their nodes are not contiguous in memory.

2.1 Packed Memory Array

The Packed Memory Array [11, 41] (PMA) is an array-based order-maintenance data structure that keeps spaces between elements. A PMA on N elements takes $O(N)$ space and supports updates in amortized $O(\log^2(N/B))$ transfers in the external-memory model. Point queries in a PMA take $O(\log N)$ transfers, and range queries that return k elements take $O(\log N + k/B)$ transfers. Although, B-trees asymptotically dominate PMAs in terms of updates and queries, in practice PMAs are faster to scan because their elements are stored contiguously in memory. Due to these properties, PMAs are used to efficiently represent sparse graphs [50, 86, 87].

The PMA maintains an implicit complete binary tree on its cells with leaves of $\log N$ cells each. Each leaf i in the PMA where $i \in \{0,\ldots,N/\log N - 1\}$ includes cells in the region $[i\log N, (i+1)\log N)$, and each internal node[3] encompasses all of the cells of its descendants. The height of a node is the distance from that node to a leaf.

Each node of the PMA tree has an upper and lower density bound that defines the number of empty cells allowed in that node. The density of a node is the fraction of non-empty cells in its region. The upper and lower density bounds in each node are related to the height of that node. The PMA enforces its density bounds by redistributing elements to neighbor nodes whenever a node violates its density bound so that the densities of both siblings are equal. Figure 4 illustrates an example of an insert and a redistribution in a PMA. In practice, PMAs support updates much faster than their amortized bounds might suggest because the amortization comes from cache-efficient redistributes [86].

[3]In other works, graph vertices are sometimes called nodes. For clarity, in this work, we will always call graph elements "vertices" and use "nodes" to refer to implicit PMA tree or the nodes in an explicit tree.

3 HIERARCHICAL DATA STRUCTURE DESIGN

In this section, we will describe the high-level motivation behind the hierarchical data structure design in Terrace. Specifically, we will propose a three-level data structure design to take advantage of skewness in graphs, in contrast to the classical "one-size-fits-all" design. The location of each vertex's neighbors in Terrace's hierarchical design depends on the degree of that vertex.

Balancing locality and updatability. The first principle in the design of Terrace is that order-maintenance array-based and tree-based data structures provide different guarantees and exhibit crossover points in terms of updatability and traversal cost. Trees designed for the external-memory model (e.g. B-trees) are quick to update and achieve asymptotically optimal cost to list all elements, but access memory out-of-order. In contrast, ordered array-like structures have asymptotically worse insertion cost than trees, but support fast traversals because they are stored contiguously in memory. In practice, there is a crossover point in the update performance of tree-like and array-like structures based on the number of elements in the structures. Therefore, the choice of structure for a vertex's neighbors should depend on that vertex's degree.

Separating vertices based on degree. The next principle in the design of Terrace is that vertices should share contiguous array-based structures for locality, but only if their degree is not too high. Sharing an array-like structure between vertices avoids cache misses while switching vertices during a traversal through the edges. If the vertices have high degree, however, the effect of saving a single cache miss per vertex is negligible because the cost to traverse all the edges dominates. Furthermore, sharing the data structure between vertices trades improved locality for slower updatability because the update cost depends on the total size of the structure. Storing high-degree vertices in an array-like structure will slow down updates for all vertices in the structure regardless of their degree. Therefore, high-degree vertices should store their neighbors in separate per-vertex data structures so they do not affect the cost of updating smaller-degree vertices. High-degree vertices are more suited to tree-based structures, because they require better asymptotic updatability guarantees.

One size does not fit all. Since the benefit of a contiguous data structure depends on the degree of vertices that use it, we propose storing vertex neighbors in either array-like or tree-like structures based on vertex degree. Specifically, we propose a hierarchical design that stores the neighbors of *medium-degree* vertices in a shared array-based structure and the neighbors of *high-degree* vertices in per-vertex trees.

Storing the neighbors of medium-degree vertices in an array-based structure improves cache locality during traversals. We bound the maximum degree that any vertex in the array-based structure can have, so the total size of the array-based structure is bounded. In contrast, storing the neighbors of high-degree vertices in per-vertex trees ensures that updating those vertices does not bottleneck the update throughput of the entire system.

Storing neighbors in place. In addition to storing neighbors in different data structures based on vertex degree, one natural optimization is to store some neighbors *in place* because accessing neighbors requires accessing at least one cache line to look up the pointer to the next data structure. Storing each vertex's neighbors in an out-of-place data structure disrupts locality during graph queries and updates. In contrast, storing some neighbors in place in the same cache line can save a cache miss from accessing a separate data structure.

Therefore, the three-level design that we propose is as follows:

(1) A list of in-place neighbors and any necessary metadata for each vertex,
(2) a shared array-based data structure containing neighbors of medium-degree vertices, and
(3) individual tree-based data structures for each high-degree vertex.

4 DATA STRUCTURE CHOICES

In this section, we describe how we choose data structures for different levels in Terrace and theoretically analyze the hierarchical design discussed in Section 3.

In the first level, we use an array of *vertex blocks* containing metadata and in-place neighbors for each vertex. In the second level, we use a packed-memory array (PMA) [11, 41] as an associative structure to store the neighbors of medium-degree vertices. In the third level, we use individual B-trees [23, Chapter 18] for each high-degree vertex.

We denote the maximum number of in-place neighbors per vertex with the parameter S and the maximum number of neighbors per vertex in the PMA with the parameter L. A vertex can have all its neighbors stored in place if its degree is less than S or spread across in place and PMA or in place and B-tree depending upon whether its degree is greater or smaller than $S+L$. That is, if a vertex v has neighbors only in place, $\deg(v) \leq S$. If a vertex v has neighbors in the in-place and PMA level, $S < \deg(v) \leq S+L$. Similarly, if a vertex v has neighbors in the in-place and B-tree level, $\deg(v) > S+L$.

In-place level. The first level in Terrace consists of a list of vertex blocks designed to store a few neighbors of each vertex in place and avoid a cache miss for accessing the neighbors of in-place vertices. Each vertex has a corresponding vertex block, and vertex blocks are ordered by vertex index. The vertex block corresponding to vertex v stores the degree of v, up to S neighbors of v sorted in place, and a pointer to the root of the corresponding B-tree in the third level (if v has high degree). The number of in-place neighbors S is a configurable parameter and is adjusted so that each vertex block can fit in a cache line or two, if Terrace must store extra attributes (e.g., weights) per edge.

Array-like level. The second level in Terrace stores up to L neighbors per medium-degree vertex in a single shared PMA to support cache-efficient traversals when edges are accessed in order. Storing neighbors in the PMA provides good cache locality as all neighbors of a given vertex are stored in consecutive memory locations, like in the edge list of CSR. The cost of performing an update or query operation in a PMA is asymptotically higher than in a B-tree, however. Since the cost to update the PMA in Terrace depends on the total PMA size, we limit the degree of each vertex that stores its neighbors in the PMA.

The maximum number of neighbors per vertex in the PMA level, L, is a configurable parameter that balances update throughput and cache locality in Terrace. That is, the parameter L exploits the crossover point between PMA and B-tree insertions in practice: when neighbors of a vertex are stored in a few consecutive pages, insertions in a PMA are competitive with insertions in a B-tree even though B-tree insertions asymptotically dominate PMA insertions.

Tree-like level. The third level in Terrace consists of individual B-trees (one for each vertex with degree $> S+L$). B-trees are a good candidate for storing high-degree vertices because they are quick to modify, have minimal space overhead, and good scan performance.

Putting it all together. As illustrated in Figure 5b, the neighbors of any vertex may be stored in at most two levels in Terrace. Each vertex has a vertex block in the first level. However, each vertex block can only store a small number of neighbors. If a vertex's neighbors do not fit in its vertex block, its remaining neighbors are stored in either the PMA or B-tree level. Terrace maintains a global order of neighbors for each vertex across different levels, i.e., the in-place neighbors are always in sorted order and the biggest in-place neighbor is smaller than the smallest neighbor in the PMA or B-tree.

Figure 5b illustrates how Terrace stores four vertices with different degrees when $S=2, L=3$. Vertex 0 has only two neighbors, so all of its neighbors fit in the first level. Vertices 1 and 3 have 5 neighbors each, so their neighbors are distributed between the in-place and PMA level. The first two neighbors are stored in the vertex block and the next three neighbors are stored in the PMA. Vertex 2 has 10 neighbors, so its first two neighbors are stored in its vertex block and the last field in the vertex block contains the pointer to the root of the corresponding B-tree where rest of the neighbors are stored.

4.1 Theoretical analysis

Table 3 shows the asymptotic runtime of operations in Ligra, Aspen, and Terrace in the external-memory model (Section 2). Given an edge (u,v) or vertex u, the operations in Table 3 are as follows:

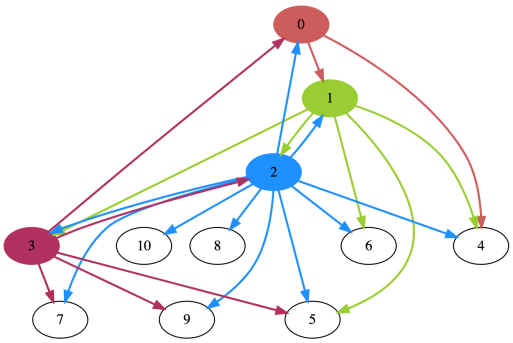
- `add_edge`(u,v) adds an edge from vertex u to v.
- `find_edge`(u,v) returns whether the edge (u,v) exists in the graph.
- `get_neighbors`(u) returns all neighbors of vertex u.

The runtime of operations in Terrace depends on the degree of the vertex in question. For an in-place vertex, adding, querying, or listing all neighbors incurs only $O(S/B)$ cache misses. For a medium-degree vertex, adding an edge requires inserting a new item in the PMA or moving an item from the in-place neighbors and adding the new item in the in-place list. Therefore, the number of cache misses is dominated by the insert operation in the PMA, which in turn depends on the overall size of the PMA. Querying a vertex requires a binary search on that vertex's neighbors, which only depends on the degree of the vertex. Listing all neighbors of a vertex requires a sequential scan through that vertex's neighbors in the PMA, which again only depends on the degree of the vertex. For a high-degree vertex, adding, querying, or listing is dominated by inserting/searching through the B-tree consisting of all the neighbors of the vertex and hence depends only on the degree of the vertex.

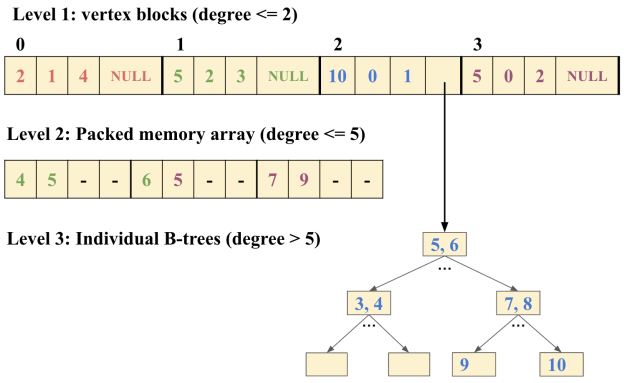
Ligra uses CSR as its underlying representation, which is a static graph format designed for queries but not updates. Therefore,

Operation	Ligra [75]	Aspen [28]		Terrace	
`add_edge`(u,v)	$O((\|E\|+\|V\|)/B)$	$O(\log\|V\|+c^2\log(\deg(u))/B)$	in exp.	$O(S/B)$ $O(S/B+\log^2(\texttt{PMA\_SIZE}-S))$ $O(S/B+\log_B(\deg(u)-S))$	when $\deg(u)\le S$ when $S<\deg(u)\le S+L$ when $\deg(u)>S+L$
`find_edge`(u,v)	$O(\log(\deg(u)))$	$O(\log\|V\|+c/B)$ $O(\log\|V\|+c\log(\deg(u))/B)$	in exp. w.h.p.	$O(S/B)$ $O(S/B+\log(\deg(u)-S))$ $O(S/B+\log_B(\deg(u)-S))$	when $\deg(u)\le S$ when $S<\deg(u)\le S+L$ when $\deg(u)>S+L$
`get_neighbors`(u)	$O(\deg(u)/B)$	$O(\log\|V\|+\deg(u)/B+deg(u)/c)$		$O(\deg(u)/B)$	

Table 3: The table lists the theoretical runtime performance of graph representations storing a graph $G(V,E)$. All bounds are $\Omega(1)$, but we omit the added 1 for ease of notation. The parameter c is expected size of nodes in Aspen (called b in the Aspen paper [28] and set to 2^8). Furthermore, S and L denote the cutoffs for the medium-degree and high-degree structures in Terrace, and `PMA_SIZE` denotes the size of the middle-level PMA in Terrace. The theoretical performance is measured in the external-memory model discussed in Section 2. The node size in the B-tree is $\Theta(B)$ where B is the cache-line size from the external-memory model.



(a) A sample directed graph.



(b) An example showing how vertices of different degrees are stored in Terrace. Specifically, we show how Terrace stores vertices 0-3 from the graph in Figure 5a when $S=2, L=3$. The first level is an array of vertex blocks. The second level is a shared PMA, and the last level consists of individual per-vertex B-trees.

Figure 5: An example of a graph stored in Terrace. If a vertex has reasonably high degree, its edges may be stored across multiple data structure levels.

adding an edge in Ligra depends on the total number of vertices and edges in the graph. Querying or listing in Ligra only depends on the degree of the vertex.

Aspen is a dynamic representation based on probabilistic balanced C-trees that supports fast concurrent updates and queries. Specifically, it stores a tree per vertex to hold its neighbors as well as a tree of pointers to each of the per-vertex trees. Since Aspen stores the

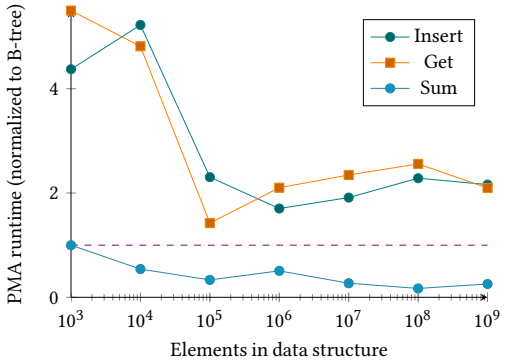


Figure 6: Normalized running time of insert, get, and sum in a PMA (normalized to a B-tree).

vertex array as a tree, it requires least $O(\log|V|)$ work per operation [4]. Its insertion cost may improve upon Terrace for medium- and high-degree nodes depending on the expected size of C-tree nodes.

4.2 Data structure microbenchmarks

Although the PMA and B-tree have the same (optimal) asymptotic scan cost in the external-memory model, they exhibit significant differences in scan performance in practice due to differences in their structure. The PMA stores all data contiguously for efficient sequential scans, while the B-tree stores its data in cache-line sized blocks connected by pointers for asymptotically faster searches and updates. The external-memory model does not capture the relative performance benefit of accessing sequential cache lines (in the PMA) compared to pointer chasing (in the B-tree) [10].

To illustrate tradeoffs between the PMA and B-tree and guide when to prefer each data structure, we ran a micro-benchmark to test insertion, point query, and sum (aggregating all values) time in both a PMA and B-tree and report the results in Figure 6. We find that the PMA supports scanning over all elements (in the sum benchmark) $2\times-5\times$ than the B-tree, but $1.5\times-5.2\times$ slower for inserts and gets.

We exploit this tradeoff between the PMA and B-tree in the hierarchical design of Terrace. When the number of neighbors is relatively small, we use the PMA for faster scans. On the other hand, when the

[4] Aspen may perform an additional optimization called a flat snapshot [28] to flatten the node tree into an array, but we omit it from the analysis because it relies on amortization of the cost across multiple queries.

number of neighbors is large, we use the B-tree to balance insertion and scan cost.

5 IMPLEMENTATION OF TERRACE

In this section, we discuss how to tune degree cutoff parameters between levels for cache locality. We then explain how we implement batch updates and multi-threading in Terrace. Finally, we give a brief description of how we extend the VertexSubset/EdgeMap API in Ligra [75] to implement graph kernels in Terrace.

Optimizing Terrace for cache locality. For unweighted graphs, vertex blocks in the first level are sized to fit in a single cache line so that accessing in-place neighbors only requires a single cache miss. Since the metadata in each vertex block takes 12 bytes (4 bytes for the degree and 8 bytes for the B-tree pointer), a cache line of B bytes can hold up to $(B-12)/4$ in-place neighbors. Since a cache line is typically 64 bytes on most x86 machines, we set the maximum number of in-place neighbors $S=(64-12)/4=13$.

When the graph is weighted, we use two consecutive cache lines per vertex block to pack metadata, neighbors, and weights. After accounting for metadata, there is space for 14 neighbors with weights in two cache lines, so we set $S=14$ in the weighted case.

Finally, we restrict the maximum number of neighbors for a vertex that can be stored in the second level (PMA) to $S+L=1024$ throughout our evaluation so that all of the neighbors of a single vertex can fit in a small number of consecutive 4 KB pages. S and L are configurable parameters and the performance of Terrace is not sensitive to slight changes to these parameters. We perform a detailed evaluation to understand the performance sensitivity to these parameters in Section 6.4.

Batch updates. Given the hierarchical design in Terrace, we perform batch updates in phases. In the first phase, we sort all the edges in the batch based on the destination vertex and then based on the source vertex. In the second phase, for each vertex, we merge in-place neighbors and the new incoming neighbors in a new sorted list of neighbors. Finally, we store the first S neighbors from the merged list in place and insert the rest either in the PMA or the B-tree depending on the degree of the vertex. If the degree of a vertex becomes greater than $S+L$ during a batch insertion, we remove that vertex's neighbors from the PMA and insert them in a B-tree along with the new incoming neighbors.

Deletes are implemented symmetrically to insertions. Given a sorted batch of edges to delete, we first remove all of those edges that were stored in-place and then delete the rest either in the PMA or B-tree. After deletion, if a vertex degree drops from the B-tree to the PMA level, we delete the B-tree and put all of its edges into the PMA level. To fill the new empty spaces in the vertex block, we move the smallest edges from the corresponding vertex's PMA or B-tree to the vertex block.

Multi-threading. Terrace supports updating multiple vertices at once, but only a single thread may update a given vertex at a time.

Since the vertex blocks and B-trees in Terrace are not shared between vertices, multiple threads can concurrently update individual vertices in those levels without contention. We use lightweight spin locks to synchronize threads trying to update neighbors in the same vertex.

Since the PMA in the second level of Terrace is shared between vertices, multi-threaded updates in the PMA require additional locks. In Terrace, we implemented a locking-based thread-safe PMA [11, 25] based on the single-threaded PMA implementation given by Wheatman and Xu for Packed Compressed Sparse Row (PCSR) [86, 87].

VertexSubset and EdgeMap API. We implement the interface proposed by Ligra [75] to define graph kernels in Terrace. The *VertexSubset* data structure represents a set of active vertices, and the *EdgeMap* primitive applies a function to edges incident to a set of vertices.

More formally, an EdgeMap takes as input a graph $G=(V,E,w)$, a VertexSubset U, and two boolean functions F and C. A call to EdgeMap applies function F to a set of edges E' such that an edge (u,v) is in E' if and only if $u\in U$ and $C(v)=true$. It returns a VertexSubset U' such that vertex $u\in U'$ if and only if $(u,v)\in E'$ and $F(u,v)=true$.

We have one optimization in our VertexSubset which can help with some algorithms. The VertexSubset has a boolean flag which specifies if the subset includes all of the vertices. If the flag is set, then membership queries into the VertexSubset simply return true instead of performing a lookup.

6 EVALUATION

In this section, we empirically evaluate Terrace and compare it with Aspen [28], a state-of-the-art graph-streaming system. We also include Ligra [75], a static graph-processing system, as a baseline for running graph algorithms in our evaluation. Ligra is static and supports faster graph algorithms compared to streaming systems. We compare all systems in terms of running time for different graph algorithms and memory footprint, and Terrace and Aspen on edge update (insert/delete) throughput. Finally, we test different Terrace configurations to investigate the performance effects of the level cutoff parameters and the three-level structure.

Experimental setup. We implemented Terrace as a `C++` library parallelized using `Cilk Plus` [40] and the Tapir [70] branch of the LLVM [48, 49] compiler. We compiled Aspen and Ligra with `g++` version 7.5 as recommended by the respective authors. All experiments were run on a 48-core 2-way hyper-threaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz with 189 GB of memory from AWS [4]. However, to perform a fair evaluation and avoid non-uniform memory access (NUMA) issues across sockets we ran all experiments on a single socket with 24 physical cores and 48 hyper-threads.

Graph kernels. Table 4 details the algorithms we implemented in Terrace: breadth-first search (BFS), PageRank (PR), connected components (CC), single-source betweenness centrality (BC), triangle counting (TC), and single-source shortest paths (SSSP). The algorithms are almost exactly the same as in Ligra [75] with minor cosmetic changes. The CC implementation does not have a shortcut, and the PR implementation runs for a fixed number (10) of iterations (i.e. it does not early-exit). Finally, the SSSP algorithm implements Bellman-Ford [23, Chapter 24].

Datasets. Table 5 lists the graphs used in the evaluation and their sizes. We tested on real social network graphs, a graph from computational biology, and a synthetic graph. Social network graphs usually have a few very high-degree vertices while the rest of the vertices have low degree according to a power-law distribution [8].

Graph kernel	Input	Output	Notes
Breadth-first search (BFS)	Source vertex	$\|V\|$-sized array of parent IDs	
PageRank (PR)		$\|V\|$-sized array of ranks	No early exit
Connected components (CC)		$\|V\|$-sized array of component labels	No shortcut
Triangle counting (TC)		Number of triangles	
Betweenness centrality (BC)	Source vertex	$\|V\|$-sized array of centrality scores	Single source
Single-Source shortest paths (SSSP)	Source vertex	$\|V\|$-sized array of distances	Bellman-Ford

Table 4: A list of graph kernels and inputs and outputs used to evaluate graph representation systems.

Dataset	Vertices	Edges	Avg. Degree
LiveJournal	4,847,571	85,702,474	17.8
Orkut	3,072,627	234,370,166	76.2
rMAT	8,388,608	563,816,288	60.4
Protein	8,745,543	1,309,240,502	149.7
Twitter	61,578,415	2,405,026,092	39.1

Table 5: A list of (symmetrized) graph datasets, number of vertices, number of edges, and average degree of those graphs used to evaluate graph representation systems.

We used the *LiveJournal* (LJ) and *Orkut* social network graphs from the SNAP dataset [51]. LiveJournal is a directed graph of the LiveJournal social network [15], and Orkut is an undirected graph of the Orkut social network. Additionally, we used the *Twitter* social network graph, which is a directed graph of the Twitter network of follower relationships [9].

We also use the *Protein* network graph [7]. The protein network graph is an induced subgraph and is available in the data repository of the HipMCL algorithm 4 [7]. It contains 1/8-th of the original vertices, of the sequence similarity network that contained all the isolate genomes from the IMG database at the time. Unlike social network graphs, the protein network graph is not heavily skewed and most (98.8%) vertices have degree less than 1000. We also generated an arbitrary graph by sampling edges from an rMAT generator [20] with $a = 0.5; b = c = 0.1; d = 0.3$ to match the distribution from Aspen [28] (we will refer to this graph as the *rMAT* graph).

To evaluate SSSP, we generated weighted graphs from unweighted graphs by assigning random integer weights in the range $[0,256)$.

We used symmetrized versions of all of the graphs for a fair comparison with the publicly available version of Aspen, which supports only unweighted undirected graphs.

Since LiveJournal, Orkut, and Twitter are static graphs which may have been preprocessed with vertex reordering [85], we randomly relabeled the vertices in all of the input graphs to model the dynamic streaming graph setting. Reordering is more difficult in streaming graphs because a good ordering may change with the stream of edges [6].

System descriptions. Terrace and Aspen differ significantly in their underlying data structures and parallelization approaches. Aspen takes a purely functional approach with compressed trees, while Terrace modifies a single hierarchical data structure with locks directly. Aspen allows read-only operations (e.g. queries) during writing transactions, and vice versa (i.e. it does not use locks). It requires that the writer is sequentialized, however. In contrast, Terrace uses locks and allows for concurrent reading and writing in different regions of the data structure.

In this evaluation, we performed updates and queries in a phased manner, so queries did not need to acquire locks. The space overhead of locking still remains and impacts the cache-behavior during graph computations, however. This behavior is the same in Aspen as it uses functional trees and there is no overhead of locking if there are no updates. We further discuss mixing concurrent updates and queries in graph streaming systems in Section 7.

Ligra is a static graph processing system that uses CSR as its underlying graph representation.

6.1 Update throughput

Setup. To evaluate insertion and deletion throughput, we first insert edges from an existing graph in Terrace. We then add a new batch of directed edges (with potential duplicates) to the existing graph and delete the same batch of edges from the graph. The batch insertion and deletion are performed using multiple threads. The graph layout remains the same at the start of every batch of insertions and deletions because the set of edges during insertion and deletion for each batch size. We perform update evaluation on the LJ and Orkut graphs. To generate edges for updates, we sample directed edges from the same rMAT generator that we used to generate the synthetic rMAT graph. We report the average of 10 trials.

Results. We show that Terrace achieves throughput up to 48 million edges per second for batch insertions and up to 9 million edges per second for batch deletions. We report our findings in Table 6. On LJ, Terrace outperforms Aspen on batches of up to 1,000,000 edges, while Aspen is faster on a batch size of 10,000,000. On Orkut, Terrace is faster on batch sizes up to 100,000, while Aspen is faster on batch sizes of at least $1,000,000$. For edge deletion, Aspen outperforms Terrace for batch sizes greater than 1000 on LJ and 100 on Orkut.

Discussion. Terrace is up to 3× faster than Aspen on batch sizes up until 1,000,000 on LJ and up to 1.75× faster than Aspen on batch sizes up until 100,000, but does not scale with larger batch sizes as Aspen does. Aspen scales with large batches because it implements insertions as a per-vertex tree merge. As the batch size increases, the number of edges per vertex increases and the overhead of the merge operation is amortized over a larger number of edges. In contrast, Terrace implements batch updates in phases at each level of the structure and performs updates at the granularity of each vertex. For larger batches, the vertex with the most updates dominates the running time.

Most highly dynamic graphs do not require the throughput that Aspen achieves on huge batches, however. For example, Twitter averages 9,346 tweets per second [1] and peaked at 140,000 tweets

	Insert						Delete					
	LJ			Orkut			LJ			Orkut		
Batch Size	Terrace	Aspen	T/A	Terrace	Aspen	T/A	Terrace	Aspen	T/A	Terrace	Aspen	T/A
1E1	3.93E5	1.25E5	3.14	2.11E5	7.28E4	1.75	1.42E6	1.31E5	10.86	7.49E5	1.28E5	5.86
1E2	1.11E6	7.11E5	1.56	8.12E5	4.32E5	1.11	2.41E6	7.62E5	3.16	1.37E6	7.55E5	1.82
1E3	5.48E6	2.77E6	1.98	3.25E6	1.97E6	1.23	4.72E6	2.98E6	1.59	1.97E6	2.83E6	0.69
1E4	1.96E7	6.56E6	2.99	1.06E7	4.93E6	1.70	5.55E6	7.38E6	0.75	2.52E6	7.05E6	0.36
1E5	4.83E7	1.57E7	3.09	2.35E7	1.26E7	1.70	8.68E6	1.61E7	0.54	3.62E6	1.46E7	0.25
1E6	4.40E7	3.46E7	1.27	1.71E7	2.69E7	0.52	9.23E6	3.43E7	0.27	4.36E6	3.32E7	0.13
1E7	2.82E7	1.03E8	0.27	2.59E7	7.76E7	0.25	6.61E6	1.05E8	0.06	4.62E6	1.05E8	0.04

Table 6: Throughput for inserting and deleting edges with varying batch sizes in the LJ and Orkut graphs in Terrace and Aspen. T/A denotes the ratio of the respective throughputs (Terrace/Aspen).

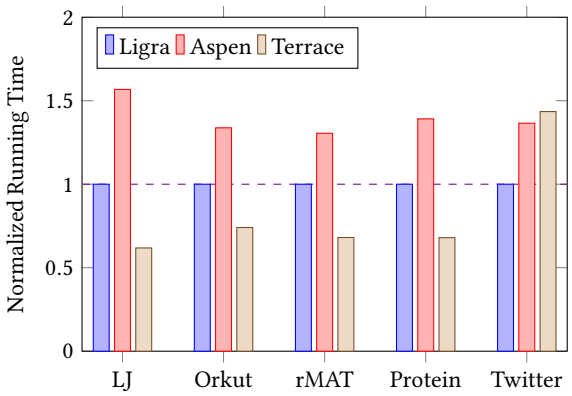


Figure 7: Time to run BFS normalized to Ligra.

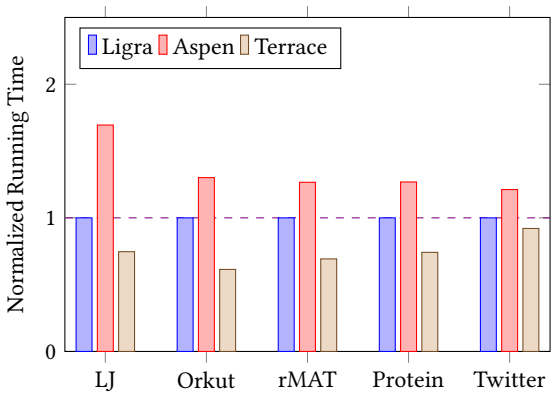


Figure 8: Time to run PR normalized to Ligra.

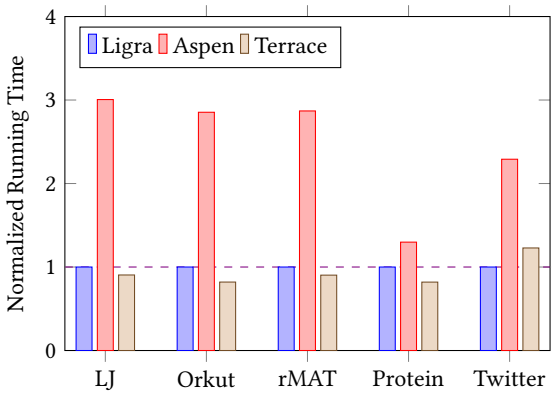


Figure 9: Time to run BC normalized to Ligra.

per second [69]. At its peak, Facebook is estimated to process about 13 million transactions per second [19]. Snapchat, another social network, saw around 210 million snaps per day in 2019 (about 2,500 per second) [79]. Applications in cybersecurity process about 10–15 million edges per second [12].

Terrace is not yet optimized for batch deletions which makes Terrace slower for deletions than Aspen for most batch sizes. Batch deletions are not as straightforward as insertions and require a careful engineering effort. Supporting batch deletions is not an inherent limitation of Terrace's design, however.

6.2 Query performance

We evaluate the performance of Terrace, Aspen and Ligra on BFS, PR, (single-source) BC, and CC, and report the results in Table 7. We plot the normalized time to Ligra using data from Table 7 of the various kernels in Figures 7, 8, 9, and 10. Since the publicly available version of Aspen is unweighted, we compare Terrace and Ligra on SSSP in Table 8. Finally, we compare Terrace and Ligra on TC, since the intersection primitive that the TC algorithm is based on is not yet optimized in Aspen and performs poorly. Table 9 presents the performance of Terrace and Ligra on TC. For each graph kernel, we took the average of 10 trials.

Traversals in graph kernels can be divided into two main categories. Vertices may be accessed in an arbitrary order as in PR, or in an order defined by the graph topology as in BFS. CC follows a similar traversal to PR, and BC follows a similar traversal to BFS. In arbitrary order, systems with more locality such as Ligra and Terrace can iterate over the edges with fewer cache misses than systems that store edges out of place. In topology-defined order, all of the systems are likely to incur a cache miss when accessing the neighbors of an arbitrary vertex.

Breadth-first search. Figure 7 illustrates the relative speed on BFS of all the systems. On average, Terrace outperforms Ligra

	BFS						PR					
	Terrace		Ligra		Aspen		Terrace		Ligra		Aspen	
Graph	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}
LJ	0.44	0.02	0.85	0.03	1.20	0.05	8.35	0.31	11.90	0.42	21.41	0.71
Orkut	0.44	0.02	0.71	0.03	0.97	0.04	23.65	0.42	26.08	0.80	41.55	1.05
rMAT	0.68	0.04	1.53	0.05	1.91	0.07	63.18	2.16	100.98	3.12	153.18	3.95
Protein	0.57	0.03	0.61	0.04	1.16	0.05	137.28	4.97	278.00	6.70	242.30	8.50
Twitter	X	0.33	X	0.23	X	0.32	X	18.26	X	19.83	X	24.03
	BC						CC					
	Terrace		Ligra		Aspen		Terrace		Ligra		Aspen	
Graph	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}
LJ	2.18	0.09	2.42	0.10	6.38	0.29	2.31	0.09	2.33	0.11	3.63	0.15
Orkut	2.86	0.10	3.29	0.12	5.61	0.34	3.34	0.12	4.16	0.17	6.08	0.22
rMAT	7.74	0.28	7.98	0.31	17.60	0.88	16.34	0.41	15.25	0.59	21.87	0.82
Protein	1.5	0.09	1.43	0.12	2.31	0.15	42.16	1.27	45.17	1.58	62.64	2.11
Twitter	X	2.53	X	2.06	X	4.72	X	4.32	X	4.32	X	5.23

Table 7: Running times (in seconds) of Terrace, Ligra, and Aspen on BFS, PR, BC, and CC. T_1 denotes the time on one thread, and T_{48} denotes the time on all (48) threads. Single thread numbers for Twitter graph are omitted due to time constraints.

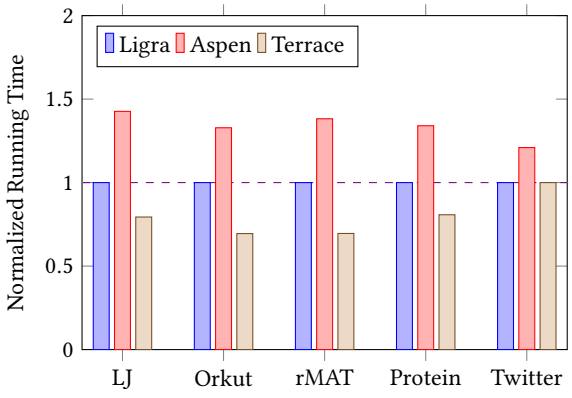


Figure 10: Time to run CC normalized to Ligra.

and Aspen by 1.2× and 1.6×, respectively. Terrace performs better since it saves cache misses with its in-place level. All of the graphs tested exhibit skewness, so most of their vertices can be stored in place. Terrace performs worse on Twitter than the other graphs because Twitter has much higher maximum degree, so many edges are stored in the relatively unoptimized B-tree level of Terrace. Future optimizations include replacing the B-tree with an optimized balanced tree representation, such as Aspen's C-trees [28].

PageRank. Figure 8 illustrates the relative speed on PR of all the systems and shows that Terrace achieves between 1.2×–2× speedup over Aspen and outperforms Ligra by 1.3× on average. Terrace shows better performance on PR because it supports faster ordered access of in-place neighbors and neighbors stored in the second level PMA. For most input graphs, a considerable fraction of all edges reside in the in-place and PMA level (see Table 11). Moreover, the VertexSubset optimization described in Section 5 also helps to improve the PR algorithm running time in Terrace.

Betweenness centrality. Figure 9 illustrates the relative speed on BC of all the systems. Terrace achieves similar (.8 × −1.1×) performance compared to Ligra and outperforms Aspen by 1.6×−3×. BC is similar to BFS in that it follows a topology-defined order and is computationally- and memory-intensive. Therefore, Aspen and Ligra diverge further than in BFS because Aspen incurs relatively more cache misses.

Connected components. Figure 10 illustrates the relative speed on CC of all the systems. On average, Terrace achieves 1.2× speedup over Ligra and 1.7× speedup over Aspen. CC starts with all vertices in the frontier, so more in-place neighbors are accessed during larger frontiers in Terrace which helps to avoid unnecessary cache misses.

	Terrace		Ligra		T/L	
Graph	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}
LJ	9.10	0.39	7.42	0.22	1.22	1.77
Orkut	13.60	0.53	8.00	0.28	1.70	1.89
rMAT	45.95	1.61	35.85	1.01	1.28	1.59

Table 8: Running times (in seconds) of Terrace and Ligra on SSSP. T_1 denotes the time on one thread, and T_{48} denotes the time on all (48) threads. T/L denotes the ratio of the respective throughputs (Terrace/Ligra).

	Terrace		Ligra		T/L	
Graph	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}
LJ	34.06	1.30	21.18	0.60	1.60	2.17
Orkut	191.79	6.52	111.00	3.08	1.72	2.12
rMAT	54.30	1.54	257.80	5.04	0.21	0.31

Table 9: Running times (in seconds) of Terrace and Ligra on TC. T_1 denotes the time on one thread, and T_{48} denotes the time on all (48) threads. T/L denotes the ratio of the respective throughputs (Terrace/Ligra).

Single-source shortest paths. Table 8 shows that Terrace is between 1.6×−1.9× slower than Ligra on SSSP. The graph traversal in SSSP is similar to that of BFS, so Terrace can take advantage of in-place neighbors. However, Terrace incurs extra overhead for storing

Graph	Terrace	Ligra	Aspen	T/A
LJ	1.43	.34	1.18	1.2
Orkut	2.41	.91	1.77	1.3
rMAT	8.73	2.13	4.32	2.02
Protein	19.05	5.27	9.08	2.09
Twitter	43.78	9.87	20.85	2.09

Table 10: Memory footprint (in GB) of relabeled and original graphs on the different systems. T/A denotes the ratio of the respective memory footprints (Terrace/Aspen).

Graph	% In-place	% PMA	% B-tree
LJ	20.12	77.20	2.66
Orkut	7.44	84.06	8.49
rMAT	5.72	93.72	0.54
Protein	2.67	83.00	14.32
Twitter	8.38	39.68	51.92

Table 11: Percentage space distribution of three layers in Terrace for different graphs.

weights compared to Ligra because it must store additional empty spaces in the PMA to store the weights array. In the weighted case, accessing neighbors in the PMA level is more expensive than in CSR.

Triangle counting. Table 9 shows that Terrace is up to 2.2× slower than Ligra on TC. TC is a computationally intensive kernel that repeatedly loops over vertices and edges, so the smaller representation in Ligra has better locality. Terrace performs well on TC on rMAT because rMAT is more skewed than the other graphs, so almost all vertices can be stored in place. However, for other graphs whenever neighbors are spread across the PMA or the B-tree, looping over neighbors to compute intersections is inefficient and incurs multiple cache misses.

6.3 Memory usage

Table 10 reports the memory footprint of the different systems. The space usage of Terrace is up to 2.1× higher than Aspen because Aspen uses data compression techniques, while Terrace uses uncompressed data structures with extra space overhead. Adding data compression to Terrace would decrease space usage and add a small amount of computational overhead.

We present the distribution of the memory in the three levels of Terrace in Table 11. For every graph in our evaluation besides Twitter, most of the edges (between 77% – 94%) are stored in the PMA level. The PMA data structure maintains extra space to support fast update operations. Specifically, the PMA in Terrace has lower density bounds in the range (0.125,0.25) and upper density bounds in the range $(0.75, (\log N - 1)/\log N)$, where N is the number of cells in the PMA. The exact density bound in the PMA depends on the number of edges stored in the PMA. For more details about PMA density bounds, see Section 2.1. There is an inherent tradeoff between the amount of empty space and the speed of updates, however. We plan to investigate the potential tradeoff between space utilization and update speed in future work.

6.4 Terrace configurations

We perform two categories of Terrace micro-benchmarks: we evaluate the performance impact of the 1) cutoffs between levels (S and L), and 2) data structures in different levels of the hierarchy.

Setup. To test the level cutoffs, we vary the values of S (number of in-place neighbors) and L (maximum degree to stay in the PMA). Specifically, we set $S = 29$ (default is 13) to fit the vertex block in two cache lines instead of one. To test the medium-degree cutoff, we fix $S = 13$ and vary L between 2^8 to 2^{12} (default is 2^{10}).

To verify the effects of each level of Terrace, we omit one out of the three levels in Terrace and measure the performance. Specifically, we use three different configurations: Inplace+PMA, Inplace+Btree, and PMA+Btree.

We evaluate the performance on four graph kernels BFS, PR, CC, and BC. We use three datasets (*LiveJournal*, *Orkut*, and *rMAT*) for both sets of experiments. We also use the *Twitter* graph when omitting Terrace levels to evaluate the impact of B-trees on the performance, since B-trees contain a significant fraction of edges in Twitter (see Table 11). We report the results by averaging the running times over all datasets and normalized the running time of the modified Terrace with the default configuration.

Discussion. Figure 11 illustrates the effect of varying the level cutoff parameters S and L. Terrace is not sensitive to changes in the configuration: the variance in the performance for different graph kernels varies between 1% – 16%. The highest variance is seen in PR and CC, since these both require traversals in an arbitrary order which slightly increases the sensitivity to the change in configuration compared to BFS and BC.

Figure 12 presents the results of omitting levels in Terrace. Using only the in-place and PMA levels improves the performance by 15% – 20% for PR and CC because the PMA allows fast sequential access. However, removing the B-tree (and only keeping the in-place and PMA levels) reduces the update throughput by 40.18% which aligns with the update-query tradeoff described in Section 4.2. Using only the in-place and B-tree reduces the performance by 14% – 88% as the B-tree has poor cache locality compared to the PMA. Therefore, the three-level Terrace design strikes a balance between updatability and graph kernel performance.

7 RELATED WORK

In this paper we focused on dynamic graphs in the streaming setting [13], but there has been significant research effort devoted to processing graphs in the static setting [24, 37, 53, 57, 63, 66, 68, 75, 84]. A survey of static frameworks can be found in [60, 90].

Many streaming graph systems apply updates in batches [28, 32, 47, 55] to amortize the work of writing to the graph. Batching updates improves update throughput but may delay the time an update appears in the graph because an update may have to wait for a batch to become sufficiently large.

There are two main approaches to applying updates in streaming graph systems. The first and the more popular approach, which we adopt in this paper, phases updates and queries separately [5, 17, 18, 32, 34, 38, 61, 71–73, 80, 83, 88]. Separating updates and queries can improve the performance of queries because it removes the need to synchronize writing and reading to the graph data structure.

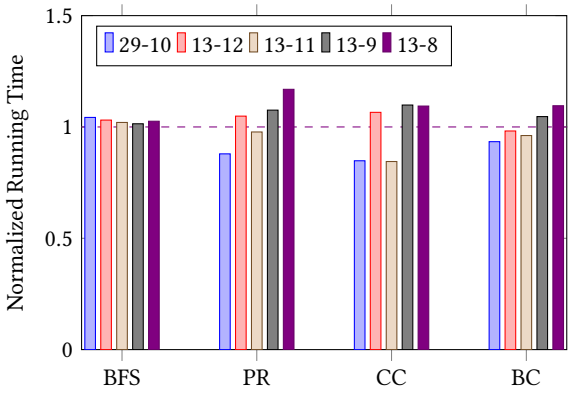


Figure 11: Normalized time of Terrace with different level cutoffs. The cutoffs are in the format $S - L$ where $S, 2^L$ are the in-place and PMA cutoffs, respectively. For example, the original Terrace configuration can be denoted $13-10$.

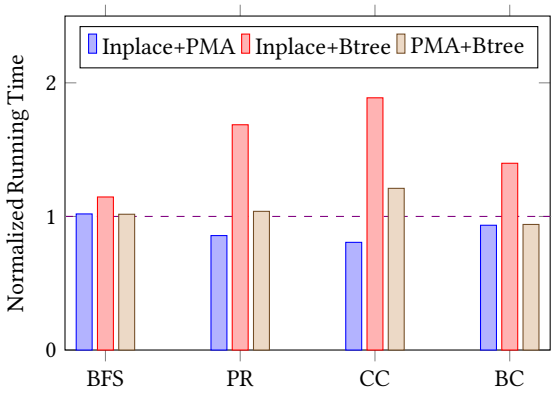


Figure 12: Normalized time of Terrace with different hierarchical configurations.

Phasing may delay queries, however, because they must wait until an update phase is finished. The second approach uses snapshotting [22, 42, 43, 46, 55] to enable concurrent updates and queries. Snapshots may even improve query performance by converting the graph storage format into one more amenable to queries [28]. More frequent snapshots are required for a more updated view of the graph, but taking snapshots requires extra processing. Common traversal-based graph operations on dynamic graphs prefer the most up-to-date state of the graph [54]. For a survey of streaming graph-processing systems, we refer the reader to [14]. Future work includes extending Terrace to take snapshots to handle mixed concurrent updates and queries.

Although both update approaches theoretically support incremental graph workloads, many recent works on dynamic graph algorithms model the first approach of applying updates in atomic batches. Specifically, the *batch-parallel* model has emerged as the primary theoretical model for design and analysis of incremental graph algorithms [2, 14, 29, 30, 35, 64, 82].

Finally, previous work has also focused on graph databases [16, 31, 44, 45, 67, 74] that support transactions while processing a streaming graph. Unfortunately, support for transactions in graph databases induces significant overhead when compared to state-of-the-art graph-streaming systems such as Stinger [59]. Therefore, our focus is on data structure design, which is independent of support for transactions.

8 CONCLUSION

In this paper, we improved the performance of dynamic graph processing via hierarchical data structure design by taking advantage of the inherent skewness in the degree distribution of real-world graphs. Terrace dynamically adapts to the skewness in the underlying graph. It stores a vertex's incident edges in different data structures based on its degree and support cache-efficient updates and traversals.

We believe Terrace strikes an appropriate balance between batch update speed and graph algorithm performance. It is faster than or competitive with Aspen, a state-of-the-art streaming graph processing system, on batch updates of practical batch sizes. At the same time, Terrace is 2× faster than Aspen on average, and is competitive with or outperforms Ligra, a fast static graph-processing system, on most tested graph kernels.

The hierarchical design approach offers promise for building high-performant streaming graph representations. In future work, it would be interesting to combine it with incremental graph algorithms that optimize for dynamic graphs [12, 30, 39, 56, 58, 78] to build highly-optimized streaming graph systems.

Future work includes reducing the memory footprint of Terrace using a compressed B-tree implementation and lowering the upper density bound in the PMA to reduce the space overhead to perform a comparison with Aspen with similar memory overheads. By design, the PMA uses a constant fraction of extra slots to support fast inserts. The PMA implementation in Terrace uses twice the space of a packed array, but could easily be changed to use a smaller constant to reduce the space usage at the cost of slightly more expensive insertions. Future work also includes implementing optimized batch deletion in Terrace.

ACKNOWLEDGMENTS

This research is funded in part by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DEAC02-05CH11231.

Research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Internet live stats. https://www.internetlivestats.com/one-second/#tweets-band, Feb 2021.

[2] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 381–392, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[4] Amazon. Amazon web services. https://aws.amazon.com/, 2019.

[5] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, 2018.

[6] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31. IEEE, 2016.

[7] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic acids research*, 46(6):e33–e33, 2018.

[8] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[9] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[10] Michael A Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, et al. Small refinements to the dam can have big consequences for data-structure design. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, 2019.

[11] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 399–409. IEEE, 2000.

[12] Jonathan Berry, Matthew Oster, Cynthia A Phillips, Steven Plimpton, and Timothy M Shead. Maintaining connected components for infinite graph streams. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 95–102, 2013.

[13] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. *arXiv preprint arXiv:1912.12740*, 2019.

[14] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. *Computing Research Repository (CoRR)*, 2020.

[15] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[16] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}: Facebook's distributed data store for the social graph. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 49–60, 2013.

[17] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[18] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management*, pages 1–8. ACM, 2012.

[19] Salvatore Catanese, Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Extraction and analysis of facebook friendship relations. In *Computational social networks*, pages 291–324. Springer, 2012.

[20] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[21] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[22] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98. ACM, 2012.

[23] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 3 edition, 2009.

[24] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 752–768, 2018.

[25] Dean De Leo and Peter Boncz. Fast concurrent reads and updates with pmas. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA'19, pages 8:1–8:8, New York, NY, USA, 2019. ACM.

[26] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 293–304, 2017.

[27] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 393–404, 2018.

[28] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 918–934. ACM, 2019.

[29] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. *Parallel Batch-Dynamic Graphs: Algorithms and Lower Bounds*, pages 1300–1319.

[30] Laxman Dhulipala, Quanquan C Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 129–143. SIAM, 2021.

[31] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. Weaver: A high-performance, transactional graph database based on refinable timestamps. *Proc. VLDB Endow.*, 9:852–863, 2016.

[32] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[33] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review*, 29(4):251–262, 1999.

[34] Guoyao Feng, Xiao Meng, and Khaled Ammar. Distinger: A distributed graph data structure for massive dynamic graph processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1814–1822. IEEE.

[35] Paolo Ferragina and Fabrizio Luccio. Batch dynamic algorithms for two graph problems. In *International Conference on Parallel Architectures and Languages Europe*, pages 713–724. Springer, 1994.

[36] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6, 2014.

[37] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[38] Oded Green and David A Bader. custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[39] Oded Green, Robert McColl, and David A Bader. A fast algorithm for streaming betweenness centrality. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, pages 11–20. IEEE, 2012.

[40] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf.

[41] Alon Itai, Alan G Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *International Colloquium on Automata, Languages, and Programming*, pages 417–431. Springer, 1981.

[42] Anand Iyer, Li Erran Li, and Ion Stoica. Celliq: Real-time cellular network analytics at scale. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 309–322, 2015.

[43] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 5. ACM, 2016.

[44] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1149–1164, 2017.

[45] Pradeep Kumar and H Howie Huang. G-store: high-performance graph store for trillion-edge processing. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841. IEEE, 2016.

[46] Pradeep Kumar and H Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 249–263, 2019.

[47] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

[48] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002. *See* `http://llvm.cs.uiuc.edu`.

[49] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International*

Symposium on Code Generation and Optimization (CGO'04), page 75, Palo Alto, California, March 2004.

[50] Dean De Leo and Peter A. Boncz. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endow.*, 14(6):1053–1066, 2021.

[51] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[52] Hang Liu and H Howie Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[53] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: a new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 340–349, 2010.

[54] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[55] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 363–374. IEEE, 2015.

[56] Devavret Makkar, David A Bader, and Oded Green. Exact and parallel triangle counting in dynamic graphs. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 2–12. IEEE, 2017.

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

[58] Robert McColl, Oded Green, and David A Bader. A new parallel algorithm for connected components in dynamic graphs. In *20th Annual International Conference on High Performance Computing*, pages 246–255. IEEE, 2013.

[59] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications*, pages 11–18, 2014.

[60] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[61] Derek G Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, iterative data processing with timely dataflow. *Communications of the ACM*, 59(10):75–83, 2016.

[62] Mark EJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[63] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471, 2013.

[64] Krzysztof Nowicki and Krzysztof Onak. Dynamic graph algorithms with batch updates in the massively parallel computation model. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2939–2958. SIAM, 2021.

[65] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, 2016.

[66] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, 2016.

[67] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. Managing large graphs on multi-cores with graph awareness. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 41–52, 2012.

[68] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Synthesizing parallel graph programs via automated planning. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 533–544, 2015.

[69] Raffi. New tweets per second record, and how! https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html, 2013.

[70] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *ACM SIGPLAN Notices*, volume 52, pages 249–265. ACM, 2017.

[71] Dipanjan Sengupta and Shuaiwen Leon Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *International Supercomputing Conference*, pages 97–119. Springer, 2017.

[72] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*, pages 319–333. Springer, 2016.

[73] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proceedings of the VLDB Endowment*, 11(1):107–120, 2017.

[74] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.

[75] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[76] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412. IEEE, 2015.

[77] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W Mahoney. Parallel local graph clustering. *Proceedings of the VLDB Endowment*, 9(12), 2016.

[78] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find with applications to incremental graph connectivity. In *European Conference on Parallel Processing*, pages 561–573. Springer, 2016.

[79] Snapchat. Snap kit. https://kit.snapchat.com/news/snap-kit-sdk-1-4, Jan 2020.

[80] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards large-scale graph stream processing platform. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 1321–1326. ACM, 2014.

[81] William F Tinney and John W Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

[82] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. Batch-parallel euler tour trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106. SIAM, 2019.

[83] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. *ACM SIGOPS Operating Systems Review*, 51(2):237–251, 2017.

[84] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single {PC}. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 387–401, 2015.

[85] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828, 2016.

[86] Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE Conference on High Performance Extreme Computing (HPEC)*, 2018.

[87] Brian Wheatman and Helen Xu. A parallel packed memory array to store dynamic graphs. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 31–45. SIAM, 2021.

[88] Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[89] Wenpu Xing and Ali Ghorbani. Weighted pagerank algorithm. In *Communication Networks and Services Research, 2004. Proceedings. Second Annual Conference on*, pages 305–314. IEEE, 2004.

[90] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.

[91] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.

[92] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 375–386, 2015.