BYO: A Unified Framework for Benchmarking Large-Scale Graph Containers

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

Xiaojun Dong
UC Riverside
xdong038@ucr.edu

Zheqi Shen
UC Riverside
zheqi.shen@email.ucr.edu

Laxman Dhulipala
UMD
laxman@umd.edu

Jakub Łącki
Google Research
jlacki@google.com

Prashant Pandey
University of Utah
pandey@cs.utah.edu

Helen Xu
Georgia Tech
hxu615@gatech.edu

ABSTRACT

A fundamental building block in any graph algorithm is a *graph container* – a data structure used to represent the graph. Ideally, a graph container enables efficient access to the underlying graph, has low space usage, and supports updating the graph efficiently. In this paper, we conduct an extensive empirical evaluation of graph containers designed to support running algorithms on large graphs. To our knowledge, this is the first *apples-to-apples* comparison of graph containers rather than overall systems, which include confounding factors such as differences in algorithm implementations and infrastructure.

We measure the running time of 10 highly-optimized algorithms across over 20 different containers and 10 graphs. Somewhat surprisingly, we find that the average algorithm running time does not differ much across containers, especially those that support dynamic updates. Specifically, a simple container based on an off-the-shelf B-tree is only 1.22× slower on average than a highly optimized static one. Moreover, we observe that simplifying a graph-container Application Programming Interface (API) to only a few simple functions incurs a mere 1.16× slowdown compared to a complete API. Finally, we also measure batch-insert throughput in dynamic-graph containers for a full picture of their performance.

To perform the benchmarks, we introduce BYO, a unified framework that standardizes evaluations of graph-algorithm performance across different graph containers. BYO extends the Graph Based Benchmark Suite (Dhulipala et al. 18), a state-of-the-art graph algorithm benchmark, to easily plug into different dynamic graph containers and enable fair comparisons between them on a large suite of graph algorithms. While several graph algorithm benchmarks have been developed to date, to the best of our knowledge, BYO is the first system designed to benchmark graph containers.

PVLDB Reference Format:
Brian Wheatman, Xiaojun Dong, Zheqi Shen, Laxman Dhulipala, Jakub Łącki, Prashant Pandey, and Helen Xu. BYO: A Unified Framework for Benchmarking Large-Scale Graph Containers. PVLDB, 17(9): 2307 - 2320, 2024.
doi:10.14778/3665844.3665859

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/wheatman/BYO.

Proceedings of the VLDB Endowment, Vol. 17, No. 9 ISSN 2150-8097.
doi:10.14778/3665844.3665859

1 INTRODUCTION

A fundamental design decision in the process of developing any graph algorithm is the choice of the graph container, that is the data structure that represents the graph. This decision can greatly affect both the running time as well as the space usage of both the graph container and the entire algorithm. A classic example used in algorithms textbooks is the difference between an *adjacency matrix* and an *adjacency list*. The former is simply a matrix which uses $n \times n$ bits to store adjacency between all pairs of n graph vertices. While it supports very efficient queries about the existence of any edge, its space usage is often prohibitive, especially when applied to sparse graphs where the number of edges m is close to the number of vertices n. On the other hand, adjacency list, while much more space efficient, can require up to $\Theta(n)$ time to determine the existence of any edge.

In practice, it turns out that algorithms typically do not need to query for the existence of a given edge, and thus the adjacency list idea is more commonly used in practice. Specifically, the usual format of choice is the *Compressed Sparse Row (CSR)* format [78], which is an array-based version of the adjacency list format. CSR uses an array A of m neighbor ids to store m edges. The neighbors of each vertex v form a contiguous fragment of A, and so for each vertex v CSR format additionally stores where this fragment of A is located. As a result, CSR requires $O(m+n)$ space to represent a graph of n vertices and m edges. Due to its simplicity and good spatial locality, CSR allows accessing the graph very efficiently. However, updating CSR is prohibitively expensive, as inserting a single edge can require $\Theta(m)$ time due to the fact that all edges are stored in a single flat array.

To address this limitation, a significant research effort over the past decade has centered on efficient dynamic-graph containers and their corresponding systems [33, 35, 36, 40, 55, 58, 67, 79, 82, 84, 85]. A *dynamic-graph system* is made up of two parts: the *container* and the *programming framework*. The container stores the graph topology and handles changes to the graph, while the programming framework uses an Application Programming Interface (API), or a specification for how two system components communicate with each other, provided by the container to express and perform analytics.

Despite the impressive body of existing work on dynamic-graph systems and containers, at present it is essentially impossible to answer the very basic question of *which* container is appropriate for a given graph application. A major reason for this situation is that most if not all papers introducing new dynamic graph containers perform *end-to-end* comparisons with existing systems. As a concrete example, prior evaluations of the dynamic-graph systems SSTGraph [82] and CPAM [35] compare with earlier dynamic-graph systems such

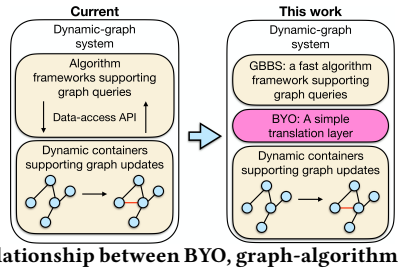

Figure 1: Relationship between BYO, graph-algorithm frameworks, and dynamic-graph containers.

as Aspen [36], but change not only the container but also important graph-algorithm details, making the source of any measured improvements unclear. A second question of no less importance that is unanswered by existing work is how much performance can be gained by using very simple "off-the-shelf" data structures (e.g., those from the standard library) to build dynamic graph systems.

In this paper, we introduce *Bring Your Own* (BYO), a unified programming framework for benchmarking and evaluating graph containers. We use BYO to perform a *comprehensive and fair* benchmark of 27 different graph containers, which include both state-of-the-art data structures such as CPAM [35] and SSTGraph [82], as well as off-the-shelf data structure libraries such as those from the `std` standard library and Abseil [2], an open-source standard library from Google. These generic data-structure libraries provide a reference implementation and demonstrate how much performance is left on the table with simple structures and minimal programming effort. Our benchmark involves running 10 fundamental graph algorithms on 10 large graph datasets with up to 4.2B edges.

Fairness of graph-container evaluation. An interesting, and perhaps surprising finding of our benchmark is the fact the algorithm performance does not vary drastically between the different containers when averaging over all graphs and algorithms (see Figure 2). This relatively small performance difference between different graph containers makes them particularly challenging to benchmark, because a reported performance gain in a proposed dynamic-graph system may be the result of many factors: the container might be better, the system may have better algorithm implementations, or the system may use a better language, compiler, or parallelization library (e.g., pthreads [66], OpenMP [29], Cilk [20], etc.). We note that varying all of these components can lead to performance variations which are at least as large in magnitude as the performance differences we observe between many of the graph containers that we compare. Hence, to truly evaluate two graph containers in an apples-to-apples way, BYO ensures that the framework and all other infrastructure (i.e., parallelization library, language, compiler) is consistent across all benchmarks. While this is a seemingly natural requirement, it was not fully met in existing papers evaluating graph systems.

Simplified graph-container evaluation. BYO is based on the Graph Based Benchmark Suite (GBBS) [37, 38], a high-performance graph-algorithm framework implemented on top of a CSR container. BYO provides a minimal translation layer between GBBS and graph containers (e.g., Aspen, SSTGraph, etc.). In other words, BYO introduces a simple and abstract container API, i.e., the API that the containers need to implement, and implements the popular Ligra/GBBS interface[1] using this API. This enables users to easily bring their own graph container and connect it to the programming framework (it suffices if the container implements BYO's container API), as well as to study their own new algorithms (as long as they are expressed in the Ligra/GBBS interface). BYO is able to represent directed, undirected, weighted and unweighted graphs. Figure 1 illustrates the relationship between BYO and other parts of a dynamic-graph system.

We note that the graph container API defined by BYO is very simple. Specifically, we find that to implement a wide variety of the primitives in GBBS, *all the graph container developer needs to implement is the map primitive* (excluding basic query functions such as `num_vertices` or `num_edges`). *Map* is a functional primitive that applies an arbitrary function `f` over a collection of elements. As we shall see, setting different functions in a map can express other functionality such as reduce and count. A map can easily be implemented with basic iterators such as those in the `C++` standard template library (STL) [51, 65] by applying the function `f` to each element in turn. This feature of BYO greatly simplifies the process of including a new graph container in the benchmark.

For comparison, the graph container API (that the container must support) from GBBS defines 10 primitive neighborhood operations (e.g., map, reduce, scan, etc.). Similarly, the GraphBLAS specification [32] includes 12 operations (e.g., mxm, assign, apply, etc.) for representing graph algorithms.

The main technical challenges in BYO were 1) identifying the correct minimal APIs that can generalize to large classes of graph containers and algorithms, 2) identifying all the code in the original GBBS implementation that makes assumptions about the underlying container and converting them to use modern C++ features that can determine which container functionality to use at compile-time to maximize performance, and 3) simplifying the design to make the translation smooth from the container-developer's point of view.

We built BYO based on GBBS because GBBS has been shown to support a wide variety of theoretically and practically efficient graph algorithms with better performance than alternatives. As we will show in Section 6, we verify these results and show that GBBS achieves 1.06−4.44× speedup on average compared to other frameworks (e.g., Ligra [73] and GraphBLAS [22, 30, 31, 53]).

Benchmark results. We perform a cross-cutting evaluation of graph containers and frameworks along several distinct axes.

The first studies the impact of the graph API when fixing the underlying container. In particular, we study performance when moving from a very simple API (e.g., only supporting `map`) to a rich API supporting sophisticated traversal primitives. Our main finding is BYO using a few very basic primitives (e.g., map, degree, and the number of edges) is only 1.16× slower on average than BYO using the full API. However, more advanced primitives, e.g., parallel map, are necessary to achieve the best performance on specific instances such as on skewed graphs.

The second axis evaluates existing graph-algorithm frameworks compared to BYO to ensure that BYO is a good starting point for a large-scale evaluation and achieves high performance on a variety of

[1]GBBS is an iteration of Ligra with a richer interface and more algorithm implementations.

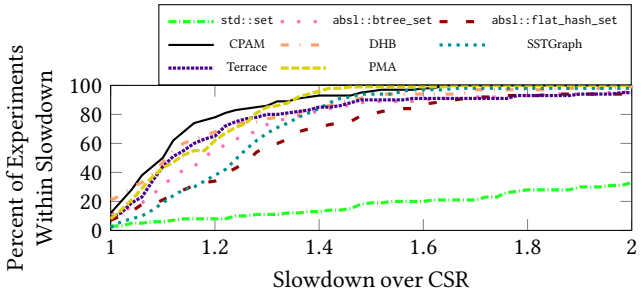

Figure 2: Slowdown of each container compared to CSR. Each point (x, y) for a given graph container[2] means that the container was at most x times slower than CSR on $y\%$ of experiments. A line going up faster implies that the container achieves closer performance relative to CSR on more experiments. We find that almost all structures are able to perform the majority of the experiments with at most a $1.4\times$ slowdown over CSR. `std::set` and absl::* are off-the-shelf containers, while the others are optimized. Details on the graph containers can be found in Section 5.

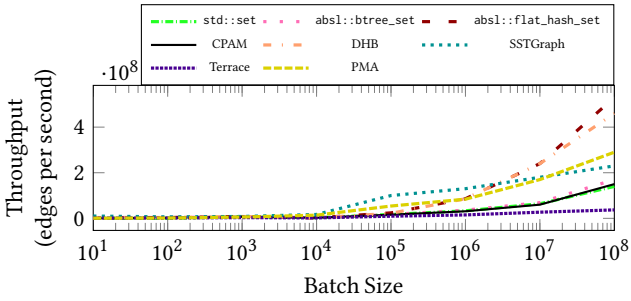

Figure 3: The throughput of inserts for different batch sizes. Of the data structures in this plot, `std::set` and absl::*, are off-the-shelf containers, while the others are optimized. Details on the different graph containers can be found in Section 5.

algorithms. We find that on average, BYO achieves competitive performance on graph algorithms when compared to Ligra [73], GraphBLAS [22, 30, 31, 53], and GBBS [37, 38], which are high-performance state-of-the-art frameworks.

The third axis studies the impact of different containers by fixing the algorithm using the BYO API and varying the container. Our results here have interesting ramifications for the future design of dynamic graph containers, as well as users of dynamic graph containers. To highlight just one example, we find that users wishing to use simple "off-the-shelf" (i.e., not tailor-made) data structures can build dynamic graph containers using Abseil B-trees while only incurring a $1.22\times$ slowdown on average across all algorithms and graphs over the best static graph container (CSR). However, off-the-shelf data structures suffer more than specialized data structures in the worst case on certain problem instances. For example, as shown in Figure 2, the Abseil B-tree incurs more slowdown on more problem settings relative to CPAM [35], a specialized data structure.

BYO addresses previous evaluation issues due to different framework implementations by making sensible optimizations for graph-algorithm performance accessible to all containers that use BYO. For example, the authors of the SSTGraph graph container implemented the Ligra framework on top of SSTGraph [82] to compare with Aspen [36], which also implements Ligra. However, the Ligra implementation in SSTGraph contains additional optimizations for certain algorithms that enable the overall system to achieve better performance on certain workloads. Specifically, SSTGraph found that one of these optimizations helped by 20% on Pagerank and 6% on Connected Components [82]. These optimizations are localized in the programming framework and could theoretically be applied to any dynamic-graph container; by incorporating them, we believe that BYO is the first system that can fairly and reliably isolate performance improvements to the graph container.

The fourth axis evaluates the performance of the dynamic graph containers when performing batch edge insertions and deletions. BYO also integrates numerous off-the-shelf containers (e.g., Abseil flat hash sets and B-trees), providing a more nuanced picture of dynamic graph containers built using standard data structures that to the best of our knowledge is absent in prior evaluations (see Figure 3).

[2]The set data structures (`std::set`, `absl::btree_set`, `absl::flat_hash_set`, and CPAM) use the inline optimization described in Section 4.

2 RELATED WORK

Graph-algorithm benchmarks and frameworks. Many graph-algorithm frameworks have appeared in the literature, but they have focused on graph-algorithm performance rather than containers. For example, several static-graph-algorithm frameworks such as Ligra [73], GraphBLAS [22, 30, 31, 53], Galois [3, 54], and GBBS [37, 38] deliver state-of-the-art running times, but are limited to using CSR as the underlying graph representation.

The GAP benchmark suite [16] (and other benchmark suites such as LDBC graphalytics [46]) has had great success in standardizing evaluations for graph algorithms. For example, it has been used to benchmark [11] several static-graph-algorithm frameworks including GraphBLAS as well as DSLs like GraphIt [88]. However, GAP was not designed to benchmark dynamic *containers* in a general way. Specifically, the GAP specification does not describe what a graph container API should look like. Moreover, the reference implementations in GAP are tightly knit with a CSR graph container.

Further, there are lines of research focused on developing incremental [27, 50, 62–64, 72] and dynamic [9, 25, 43, 49, 59, 60, 70, 81] algorithms, or designing highly efficient graph containers [35, 36, 40, 47, 67, 79, 82–85] with supports of dynamic updates to the graphs. However, all of these works either create an ad-hoc framework with specialized optimizations, or implement only a few algorithms and compare to a few of the recent works. This leads to comparisons between entire systems and not just between graph containers.

Importance of benchmarking graph containers alone.

Significant research effort has been devoted to developing and benchmarking graph containers and their corresponding systems. These works have reported significant speedups:

- SSTGraph finds a $1.6\times$ speedup over Aspen [82].
- Terrace finds a $1.7-2.6\times$ speedup over Aspen [67].
- Aspen finds $1.8-15\times$ speedup over prior dynamic data structures [36].
- VCSR finds as $1.2-2\times$ speedup over PCSR [10].
- PPCSR finds a $1.6\times$ speedup over Aspen [85].
- CompressGraph finds a $2\times$ speedup over Ligra+ [26]
- Teseo finds frequent speedups of at least $1.5\times$ over other graph containers [33]

However, it is likely that much of the improvements seen in these works are from factors other than the graph container itself.

Pros and cons of different graph containers. Aside from raw performance comparisons, graph containers may support different functionality and storage guarantees. A few major distinctions include:

- **Compressed vs uncompressed.** CPMA [83], CPAM [35], and Aspen [36] all support compression of the graph, which has been shown in prior work to help performance and space usage. In contrast, systems like Terrace [67] are uncompressed and use significantly more space, which may not work on much larger graphs.
- **Functional vs in-place.** Tree-based containers such as CPAM and Aspen may support functional updates, which enable concurrent queries and batch updates on the graph. Most containers (e.g., Terrace, PPCSR, etc.) store the graph in place, so queries and updates must happen in a phased manner.
- **Batch updates vs concurrent updates.** Some graph containers (e.g., CPMA, CPAM, and Aspen) provide native support for batch updates, which apply a set of updates to the container as one operation. Batch updates improve update throughput at the cost of latency compared to concurrent updates, where the updates may happen simultaneously, but each individual update is atomic.

The focus of this paper is on performance alone, so some of these differences (e.g., update type or functional storage) are not reflected in the results.

3 BACKGROUND

This section provides background on graphs and their representations necessary to understand the data structures studied in this paper. It also reviews the GBBS framework and describes how it uses different API components to express graph algorithms. Throughout this paper, we use the standard shared-memory model of parallelism in which we have a set of threads that access a shared memory.

Graphs. A *graph* is a way of storing objects as *vertices* and connections between those objects as *edges*. For simplicity we focus on *unweighted* graphs. Formally, an unweighted graph $G = (V, E)$ is a set of vertices V and a set of edges E. We denote the number of vertices $n = |V|$ and the number of edges $m = |E|$. Each vertex $v \in V$ is represented by a unique non-negative integer less than $|V|$ (i.e. $v \in \{0,1,...,|V|-1\}$). Each edge is a 2-tuple (u,v) where $u,v \in V$. For each edge, we refer to the vertex u as the *source* and the vertex v as the *destination*. For undirected graphs, for each edge (u,v) there exists an edge (v,u). In the undirected case, the *neighbors* of a vertex u are all vertices v such that there exists an edge (u,v). For a weighted graph each edge is a 3-tuple (u,v,w). Finally, the *degree* of a vertex in a graph is the number of neighbors it has.

3.1 Graph representations

A graph whose vertex set is $\{0,...,|V|-1\}$ can be thought of and represented as a sequence $s_1,...,s_{|V|-1}$ of *neighbor sets*, where a neighbor set s_i stores all of the neighbors of a vertex i.

Storing graphs with set containers. The sequence of sets abstraction leads to a classical design for a graph storage format: a list of pointers (one for each vertex) to pre-selected data structures holding each vertex's neighbors. Graph-container developers can trade off performance properties (e.g., insert vs scan) based on the choice of per-vertex data structures.

A *set container* is any data structure that stores a unique collection of elements and can be used to store vertex neighbors. Examples of set containers include red-black trees [28], B-trees [14], hash maps [28], and arrays. For the purposes of this paper, we define a set container to support the following operations:

- `insert(e)/remove(e)`: Insert/delete element `e` into/from the set.
- `map(f)`: Apply the function `f` to all elements of the set (can be implemented with `C++` style iterators).
- `size()`: Return the number of elements in the set.

These functionalities are naturally expected from any set container data structure. For example, the `C++` standard template library (STL) [51, 65] includes these functions (as well as others) in their container specification.

Compressed graphs. To handle increasingly large graphs, several graph-processing systems [35–37, 74] include support for compressed graph formats. Specifically, they provide support for graphs where neighbor lists are encoded using byte codes [18, 19] and a parallel generalization [74] of byte codes. Byte codes store a vertex's neighbor list by difference encoding [76] consecutive vertices, with the first vertex difference encoded with respect to the source. Compression enables larger graphs to fit in memory and reduces memory traffic, which may help during parallel processing [35–37, 74].

All of the *compressed* graph representations in this paper use difference encoding and byte codes to compress the graphs.

3.2 GBBS API

GBBS [37] is a shared-memory graph processing framework based on Ligra [73] that provides a benchmark suite of over 20 non-trivial graph problems. GBBS uses a shared-memory approach to parallel graph processing in which the entire graph is stored in the main memory of a single multicore machine. Graphs in GBBS are assumed to be stored in the compressed sparse row (CSR) format described earlier. The underlying neighbors stored can be stored either uncompressed, or using a compressed format.

Vertex datatypes and primitives. The vertex datatype interface provides functional primitives over vertex neighborhoods, such as map, reduce, scan, count (a special case of reduce where the map function is a boolean function), as well as primitives to extract a subset of the neighborhood satisfying a predicate (filter), among other primitives.

VertexSubsets. GBBS uses the vertexSubset datatype from Ligra, which represents a subset of vertices in the graph. A subset can either be *sparse* (represented as a collection of vertex IDs) or *dense* (represented as a boolean array or bit-vector of length n, the number of vertices in the graph).

EdgeMap. edgeMap is a basic graph processing primitive useful for performing graph traversal. The edgeMap primitive takes as input a frontier, or vertexSubset. It then applies a user-defined function to generate a new frontier consisting of neighbors of the input frontier. For example, in a breadth-first search, the user-defined primitive emits a neighbor in the output frontier if it has not yet been visited. GBBS includes several generalizations of edgeMap that aggregate the results of the edgeMap at the source vertex, as well as generalizations that return a subset of the neighbors of the input vertexSubset.

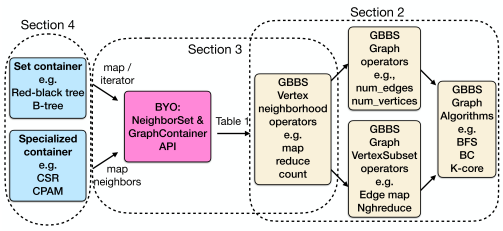

Figure 4: Relationship between BYO framework, graph containers, and graph algorithms (via GBBS).

4 BYO API DESIGN AND IMPLEMENTATION

The goal of BYO is to make it as easy as possible for a graph-container developer to use any data structure in a high-performance and general graph programming framework. This section details the "set" and "graph container" APIs that BYO exposes to connect with arbitrary graph data structures. It also describes the changes that we made to the GBBS implementation to be agnostic to the underlying data structure. Finally, we will discuss framework-level optimizations that have appeared in various places throughout the literature that we have collected in BYO. For simplicity, we will describe all of the API components in terms of unweighted graphs.

BYO provides a translation layer between graph containers and GBBS. Figure 4 illustrates the relationship between data structures, BYO, and GBBS components. Specifically, BYO translates between the data structure API and the read-only GBBS neighborhood operators such as map, reduce, and scan (Section 3.2). This paper focuses on these operators to strike a balance between simplicity and expressiveness.

4.1 NeighborSet API

As described in Section 3, a graph can be represented as a sequence of sets where each set contains the edges incident to a single vertex, that is a neighbor set. Many graph representations, such as the adjacency list in Stinger [40], the tree of trees in Aspen [36] and CPAM [35], directly implement this two-level structure.

We now describe the *NeighborSet API*, a high-level description of the necessary functionality for *a single* vertex neighbor set. The NeighborSet API enables easy parallelization over the vertex set, since all of the neighbor sets are independent.

Figure 5 illustrates the relationship between the vertex level (maintained by BYO) and the set data structures (implemented by the developer). BYO abstracts away the details of choosing a data structure for both the vertex sequence and neighbor sets and enables the user to just implement the neighbor set. Currently, BYO implements the vertex sequence as an `std::vector` for simplicity, but could theoretically use any set data structure.

We find that the minimal API necessary for a neighbor set data structure to implement the GBBS operators which do not change the neighbor sets is to expose a `size` function and an `iterator` which supports sequentially iterating through the elements one by one, i.e., a *forward* iterator. These are two basic functionalities are naturally expected from set implementations. For example, the C++ standard template library (STL) [51, 65], a widely-used standard library of basic utilities, includes both of these (among others).

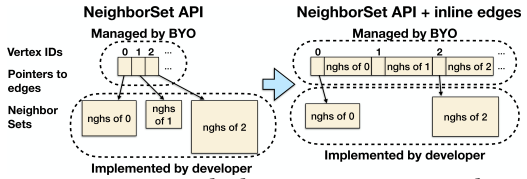

Figure 5: Data structure and inline optimization using the set API.

Required functionality for algorithms.

- `iterator` or `map(f)`: Apply the function `f` to all elements in the set. As mentioned in Section 1, an iterator can be used to implement `map` by simply iterating through all elements in the set and applying the function `f`.
- `size()`: Return the number of elements in the set.

Optional functionality for algorithms.

- `map_early_exit(f)` (if no iterator): Apply the function `f` to the elements in the set until `f` returns true. We can implement it with an iterator by exiting the iteration when `f` returns true.
- `parallel_map(f)`: Apply the function `f` in parallel to all elements in the set.
- `parallel_map_early_exit(f)`: Apply the function `f` in parallel to the elements in the set until `f` returns true. Because the iterations are running in parallel, other threads may continue even after one returns true.

Required functionality for updates. In addition to providing read access to the graph, dynamic-graph representations must also support updates (inserts / deletes). Therefore, BYO requires the following standard API if the data structure supports dynamicity:

- `insert(e)` : Insert a single element `e` into the set.
- `delete(e)`: Delete a single element `e` from the set.

Optional functionality for updates. Some data structures (e.g., Aspen [36] and CPAM [35]) may natively support batch updates at the set level. That is, in addition to the parallelization over the vertices, the data structure itself may support batch updates for work sharing and potentially additional parallelism. Therefore, BYO also provides an interface for batch insertions/deletions as part of the NeighborSet API:

- `insert_batch(batch)`: Insert a batch of elements into the set.
- `delete_batch(batch)`: Delete a batch of elements from the set.

Translating graph batch-update API into NeighborSet API. Modern dynamic-graph systems support inserting and deleting a *batch* (i.e., a set) of edges rather than one at a time. Different data structures require different input forms for efficient batch updates to be applied. BYO supports three different forms for batches. The first is to simply globally sort the batch. This is good for data structures that perform global merges. The second is to semi sort [39, 44, 80], which groups equal keys (sources) together but does not necessarily globally order the keys across the whole list, to partition the batch into edges destined for different vertices the batch by source. Just semi sorting is sufficient for data structures that use the NeighborSet API, but do not derive any benefit from sorting such as hash tables. The third is to first semi sort by source, and then group and integer sort each individual set of edges. This is good for ordered set containers.

Advantages of the NeighborSet API. The NeighborSet API is designed to make it as easy as possible for a data-structure developer to integrate their container with BYO, as long as they implement the basic contract specified in the STL container API. Notably, if

a developer wants to integrate a set library that implements `size` and `iterator` functionality with BYO, they *do not need to write any additional code*. To improve ease of use, we implemented BYO to automatically translate from the STL container API to the BYO API. That is, integrating a data structure that implements the STL container API just requires importing it at the top of the test driver and specifying its type as the graph container under test.

Furthermore, we incorporate the *inline optimization* from Terrace [67] into the vertex set in BYO to benefit arbitrary data structures and enable faster systems overall. This optimization stores a few (about 10) edges inline in the vertex level next to the pointer to the neighbor set for each vertex. The goal is to avoid indirections for low-degree vertices. The idea was originally introduced in Terrace but can be generally applied to any graph container with the sequence of sets structure. Figure 5 illustrates the inline optimization in an arbitrary sequence of sets graph representation. On average, we find that the inline optimization speeds up set containers by 1.06× on average, which we will detail in Section 6.

Another benefit of the NeighborSet API is flexibility in the choice of outer set data structure with a fixed inner set type. For example, BYO can store directed graphs with two dense outer vectors (one for incoming and one for outgoing neighbors). Similarly, BYO could store the outer set sparsely for a static or dynamic form of Doubly-Compressed Sparse Rows (DCSR) [21].

4.2 GraphContainer API

Next, we introduce the *GraphContainer API* to connect BYO to graph data structures that do not represent the neighbor sets as separate independent data structures. For example, the classical Compressed Sparse Row (CSR) [78] representation stores all of the neighbor sets contiguously in one array. Furthermore, some optimized dynamic-graph containers like Terrace [67] and SSTGraph [82] collocate some neighbor sets for locality. These graph data structures internally manage both the vertex and neighbor sets.

We find that the minimal API necessary for a data structure to support a diverse set of graph algorithms via BYO is just `map_neighbors` and `num_vertices`.

Required functionality for algorithms.
- `map_neighbors(i, f)`: Apply the function `f` to all neighbors of vertex `i`.
- `num_vertices()`: Return the number of vertices in the graph.

Optional functionality for algorithms.
- `num_edges()`: Return the number of edges in the graph.
- `degree(i)` : Return the degree of vertex `i`.
- `map_neighbors_early_exit(i, f)`: Apply the function `f` to the neighbors of vertex `i` until `f` returns true.
- `parallel_map_neighbors(i, f)`: Apply the function `f` in parallel to all neighbors of vertex `i`.
- `parallel_map_neighbors_early_exit(i, f)`: Apply the function `f` in parallel to the neighbors of vertex `i` until `f` returns true. Because the iterations are running in parallel, some threads may continue even after one returns true.

Many of these functions such as `num_vertices`, `num_edges`, and `degree` are commonly expected from any graph container as a way to query the graph structure. The `map_neighbors` functionality is also commonly implemented in graph containers to support graph algorithms. The optimized variants of map (early exit and parallel) can be more difficult to implement than serial map. They have been studied in the literature [15, 36, 74] and have been reported to help on some algorithms and graphs. We perform a comprehensive study of the performance benefits of the individual API components in Section 6.

Table 1: GBBS primitives implemented using just the map primitive.

GBBS Vertex Operator	*B.Y.O. Lambda*
Map	Pass through provided function
Reduce	auto value = identity map([&](auto ...args) { value.combine(f(args...)) })
Count	int cnt = 0 map([&](auto ...args) { cnt += f(args...) })
Degree	int cnt = 0 map([&](auto ...args) { cnt += 1) })
getNeighbors	Set ngh = {} map([&](auto ...args) { ngh.add(args) })
Filter	Set ngh = {} map([&](auto ...args) { if (pred(args) ngh.add(args) })

Required functionality for updates. The GraphContainer API directly translates the batch-update API at the BYO level to the underlying container. Since the batch given to BYO may not be sorted, BYO sorts it because many batch-update algorithms require sorted batches [35, 36, 67, 82, 83]. Data structures using the GraphContainer API in BYO must implement the following functions:
- `insert_sorted_batch(batch)`: Insert a sorted batch of edges into the graph.
- `delete_sorted_batch(batch)`: Delete a sorted batch of edges from the graph.

Advantages of the GraphContainer API. The GraphContainer API enables cross-set optimizations that cannot be expressed in the NeighborSet API at the cost of programming effort. For example, in the classical CSR, the edges are stored contiguously in one array for locality rather than in separate per-vertex arrays, which is not easily captured by the set of sets abstraction. Another example is the hierarchical structure in Terrace [67], which stores some neighbor sets contiguously in a dynamic array-like data structure. Additionally, SSTGraph [82] shares some metadata between the different neighbor sets for space savings, which cannot be captured with the independent sets abstraction. However, the GraphContainer API cannot access the general inline optimization supported by the NeighborSet API.

4.3 Maintaining graph metadata in BYO

Furthermore, BYO reduces the burden on the programmer by internally maintaining information about graph structure at the framework level when it is not done at the container level. Specifically, it stores the the degree of each vertex in the vertex set as well as the total number of edges if needed. If a set container does not implement `size` or a graph container does not implement `degree` and/or `num_edges`, BYO defaults to its internal metadata.

4.4 Connecting BYO to GBBS

BYO simplifies the list of original read-only GBBS neighborhood operators such as map, reduce, count, etc. by implementing several of them with `map`. The original GBBS specification required the data-structure developer to implement several neighborhood operators. In contrast, BYO requires them to implement **only one**. Table 1

demonstrates how to implement the original GBBS neighborhood operators using different `map` lambdas.

In addition to providing the translation layer from the GBBS vertex neighborhood operators, we also needed to modify the implementation of some vertexSubset operators in GBBS because they assume that the underlying graph is stored in CSR format. This is not inherent in the high-level GBBS specification, but was a prevalent assumption in the codebase. Specifically, several EdgeMap functions assumed that they could directly perform array access into the container to access relevant parts of the graph, which does not hold for arbitrary data structures.

VertexSubset optimizations in BYO. We also include several optimizations to the higher-level vertexSubset abstraction in BYO that can benefit all systems, since they are independent of the container. Specifically, we converted the boolean array that the vertexSubset uses in dense mode to a *bitarray*, which has been shown in prior work [82] to improve overall algorithm performance by about 1.05×. We confirm these results with our own experiments and find that on average, BYO with a boolean array is 1.04× slower than BYO with a bit array. We also removed some unnecessary work (e.g., copies and sorts) at the vertexSubset level.

GBBS algorithms supported in BYO. In addition to algorithms that only require read-only neighborhood operators (e.g., map, reduce, degree, etc.), BYO can also support whole-graph algorithms that at first seem to require vertex-vertex operators such as intersection by first applying an out-of-place *filter* operation (see Table 1) to convert the data structure into a more amenable format for algorithms. For example, efficient implementations of triangle counting (TC), a classical example of an algorithm based on vertex-vertex intersections, first perform a filter to reduce the number of edges and eliminate enumerating duplicate triangles [75]. Applying this filtering optimization is the standard technique for running other whole-graph algorithms involving vertex-vertex operations, e.g., k-truss, butterfly counting, structural similarities, etc [24, 42, 52, 68, 71, 86].

Notably, this filtering operation is not an algorithmic change made in BYO. The triangle counting (TC) implementation in base GBBS first performs a filter before intersection, even when the graph is stored in CSR. To demonstrate BYO's functionality, we used filter to implement TC on top of a variety of dynamic containers. We found that they all support TC in similar time because the only variation is the time spent doing the filter, which is relatively inexpensive compared to the time performing intersections on the filtered graph.

5 DATA STRUCTURES EVALUATED

This section details the many graph containers that we evaluate using BYO in this paper. We include general-purpose off-the-shelf data structure libraries that were not designed or optimized for graph processing as easy-to-use baselines. We also include state-of-the-art special-purpose graph containers from the literature to demonstrate the advantage of further development and optimization.

5.1 Off-the-shelf data structures

We evaluate off-the-shelf data structures both from the `C++` standard template library (STL) [51, 65], and from Abseil [2], a collection of open-source data structure implementations in `C++` designed to augment the STL. These containers serve as baselines to demonstrate how much performance is left on the table with simple easy-to-use data structures. The details about the included structures are as follows:

- `std::set` [6]: A standard container library that comes with any `C++` distribution. It maintains a sorted set of unique elements and is usually implemented with a red-black tree [28].
- `std::unordered_set` [7]: A standard container library. The elements are stored unsorted in a hash table [28].
- `absl::btree_set` [2]: An ordered set data structure that generally conforms to the STL container API. It is implemented as a B-tree [14], a classical cache-friendly tree data structure.
- `absl::flat_hash_set` [2]: An unordered set data structure that generally conforms to the STL container API. It is implemented using a hash table.
- `std::vector` [8]: A basic array implementation that comes with any `C++` distribution. It stores elements continguously in memory.

Integration with BYO. These general-purpose data structures are all sets and therefore use the NeighborSet API described in Section 4.1. Using the common STL container API functions of `size` (returns the number of elements in the container) and `iterator` (enables access to the elements in the container), the general-purpose data structures naturally support the `num_edges`, `degree`, `map`, and `map_with_early_exit` functionality through translation via BYO.

Since these set data structures implement the NeighborSet API, they do not need any additional code to integrate with BYO, as described in Section 4.1. However, vector is still implemented in full manually to give users of the system an example of each function being implemented and to include the parallel mapping functions which can not be generated automatically.

5.2 Optimized data structures

We also evaluate optimized general set data structures that have previously been used for dynamic graphs as well as special-purpose graph containers designed specifically for graphs. These containers demonstrate what kind of performance is achievable with tailor-made data structures. Some of these optimized data structures have both uncompressed and compressed versions (Section 3), which we will denote with †. The details are as follows:

- Compressed Sparse Row† [78] (CSR): A classical *static* representation for graphs.
- Terrace [67]: A dynamic-graph container optimized for skewed graphs. It uses a hierarchical structure built on arrays, a Packed Memory Array (PMA) for graphs [84, 85], and B-trees [14] to organize the vertices by degree.
- SSTGraph [82]: A dynamic-graph container built on a shallow hierarchy of sorted PMAs [17, 48].
- PMA† [83]: A cache-oblivious updateable array.
- Dynamic Hashed Blocks (DHB) [79]: A dynamic-graph container based on block-based hashing.
- Aspen† [36]: A randomized blocked tree.
- CPAM† [35]: A deterministic blocked tree.

Integration with BYO. We integrate CSR [78], Terrace [67], SSTGraph [82], and DHB [79] with the GraphContainer API in BYO. All of these data structures already natively include the `degree` functionality, and most include `num_edges`. For the systems that implemented the Ligra interface (Terrace [67], SSTGraph [82], and PMA [83]), we

adapted the map-based functionality from their original integration with Ligra to integrate with BYO. In DHB [79], we used the provided iterator to implement the `map_neighbors` and `map_neighbors_early_exit` functionality.

Additionally, we incorporate Aspen [36] and CPAM [35] with the NeighborSet API in BYO. These tree-based containers natively implement `size` and all of the map variants. We incorporate the PMA [83] into both the NeighborSet and GraphContainer API in BYO since it was originally presented as a single PMA for the entire graph.

6 EXPERIMENTAL EVALUATION

We employ BYO to evaluate 27 graph containers (and variants), 10 graph algorithms, and 10 graph datasets.

We first summarize the high-level takeaways from our large-scale evaluation. Next, we study the performance effects of the different functionalities in the BYO API described in Section 4, which helps explain the impact of missing functions in later evaluations. We then evaluate BYO compared to other state-of-the-art graph-algorithm frameworks to ensure that BYO's extra generality does not come at the cost of performance. Finally, we perform a comprehensive evaluation of 27 graph data structures on a suite of 10 graph algorithms. We also evaluate the dynamic structures on their update throughput. We publicize the raw data for all experiments as well as the code in the Github repo[3]. We evaluate all systems in an unweighted, undirected mode to enable the widest compatibility.

6.1 Summary

First, we evaluate different configurations of the GraphContainer API (Section 4.2) in BYO with CSR as the underlying graph representation to determine the importance of the different functions (e.g., the types of maps). On average, we find that the *minimal efficient* API configuration implements the required functionality, `degree`, and `num_edges`. This configuration is only 1.16× slower compared to the *full configuration*, or the BYO configuration with all of the (required and optional) functionality listed in Section 4.2. However, in the worst case, the minimal efficient configuration incurs up to 3.1× slowdown over the full API. Adding the more advanced maps (parallel map and early exit) mitigates the worst case, which is important for difficult problem instances e.g., graphs with high degree. The full details are in Section 6.3.

Next, we evaluate BYO compared to other state-of-the-art graph-algorithms frameworks in Section 6.4 and find that BYO is competitive with GraphBLAS, Ligra, and GBBS, which are state-of-the-art high-performance frameworks for graph algorithms. Specifically, BYO achieves between 1.06−4.44× speedup on average compared to other frameworks. These results indicate that BYO is a good candidate for integrating with various graph containers because the resulting systems are both expressive and achieve high performance. We also compare several of the original systems that introduced dynamic-graph containers (e.g., PMA, CPAM) with their original frameworks to their implementations using BYO and find that BYO's implementation is faster.

Finally, we perform a comprehensive study of dynamic-graph containers on both graph algorithm and batch-insert performance in Sections 6.5 and 6.6.

[3]https://github.com/wheatman/BYO

Table 2: Graph problems supported in BYO and their categories [37].

Category	*Problem*
Shortest-path problems	Breadth-First Search (BFS) Single-Source Betweenness Centrality (BC) $O(k)$-Spanner (Spanner)
Connectivity	Low-Diameter Decomposition (LDD) Connectivity (CC)
Substructure	Approximate Densest Subgraph (ADS) k-core
Covering	Graph Coloring (Coloring) Maximal Independent Set (MIS)
Eigenvector	PageRank (PR)

In terms of graph algorithm performance, our findings show that graph data structures are very similar on average, but that developing specialized graph data structures is worthwhile because additional optimization effort can improve holistic performance on more challenging instances, e.g., high-degree graphs. All of the data structures tested besides the unoptimized `std::set` and `std::unordered_set` incur most about 1.5× slowdown compared to CSR when averaging across all algorithms and graphs. Furthermore, the best specialized container (CPAM with inline) is only about 1.1× faster than the best off-the-shelf data structure (`absl::btree_set` with inline) on average. However, the worst-case slowdown for the `absl::btree_set` is 2.6×, while CPAM achieved a maximum slowdown of 1.9× over CSR. These results suggest that specialized data structures can improve upon off-the-shelf data structures on more difficult problem settings.

BYO cuts through combinatorial explosion in terms of programming effort to enable large-scale comparisons of graph containers on a diverse suite of algorithms to provide a complete view of how fast a graph container can support algorithms in a variety of cases.

In terms of batch inserts, we find that off-the-shelf structures exhibit a folklore query-update tradeoff: the Abseil B-tree, which is best off-the-shelf structure for algorithms, experienced around a 3× slowdown on larger batch inserts compared to the Abseil flat hash set. However, the hash set was worse on algorithms compared to the B-tree. However, specialized containers can overcome the query-update tradeoff on the largest batches: the single PMA is better on algorithms on average compared to the B-tree as well as on the largest batch size.

6.2 Experimental setup

Algorithms evaluated. Table 2 lists the 10 graph problems that BYO provides parallel algorithms for based on the data-structure API and GBBS abstractions. BYO does not change the algorithm implementations from GBBS (just the translation from the data structure to the lower-level primitives). Therefore, BYO inherits the strong theoretical bounds on algorithm work and depth (and therefore parallelism) from GBBS. These algorithms cover a wide range of problems, including shortest-path, connectivity, substructure, covering, and eigenvector problems. We refer the interested reader to the GBBS paper for full details on the algorithms and their implementations [37].

Systems setup. We ran all experiments on an Intel®Xeon®Gold 6338 CPU @ 2.00GHz dual socket machine with 64 physical cores (128 hyperthreads) and 1024 GiB of main memory running across 16 channels at 3200MT/s. The machine has 5 MiB of L1 cache, 80 MiB of L2 cache, and 96 MiB of L3 cache.

Table 3: Sizes of (symmetrized) graphs used (ordered by size).

Graph	*Vertices*	*Edges*	*Avg. Degree*
Road (RD)	23,947,347	57,708,624	2
LiveJournal (LJ)	4,847,571	85,702,474	18
Com-Orkut (CO)	3,072,627	234,370,166	76
rMAT (RM)	8,388,608	563,816,288	67
Erdős-Rényi (ER)	10,000,000	1,000,009,380	100
Protein (PR)	8,745,543	1,309,240,502	150
Twitter (TW)	61,578,415	2,405,026,092	39
papers100M (PA)	111,059,956	3,228,124,712	29
Friendster (FS)	124,836,180	3,612,134,270	29
Kron (KR)	134,217,728	4,223,264,644	31

To validate the choice of the BYO API, and show that it achieves similar end-to-end running times as existing systems, we first compare BYO with Ligra, GBBS, GraphBLAS, and GAP on the 4 algorithms common to all of the systems: BFS, BC, CC, and PR (Table 2). Ligra/GBBS/GraphBLAS are state-of-the-art graph-algorithm frameworks, while GAP is a suite of direct algorithm implementations.

We compiled all codes besides Ligra [73] with `g++ 11.4.0`. GraphBLAS and GAP [16] are parallelized natively with OpenMP [29]. Since Ligra [73] was designed and tested with Cilk [20] but Cilk is no longer supported in g++, we compiled Ligra using `clang++ 14.0.6` with OpenCilk 2.0 [69], the modern iteration of Cilk. We ran GBBS with its default custom parallelization framework and scheduler. BYO uses the same custom parallel scheduler since it uses GBBS as a starting point for its implementation.

To test the GraphBLAS programming framework [22, 30, 31, 53], we include LAGraph [5, 61], a collection of algorithms implemented using GraphBLAS.

We kept the number of trials unchanged from each system's distribution and took the average over all trials. By default, Ligra/GBBS/BYO perform 3 trials (with one extra warmup trial) per experiment, GAP performs 16, and GraphBLAS performs 64.

Furthermore, we evaluate the systems resulting from integrating BYO with existing containers compared to their original systems. Specifically, we ran PMA, SSTGraph and CPAM with their original driver code, compiled with g++-11, which use their own implementation of the Ligra API to run algorithms. The original systems use the same parallelization framework as BYO. We ran the PMA variants and SSTGraph on BFS, BC, PR, and CC, and CPAM on BFS and BC because those were the algorithm implementations provided in the original codebases that intersected with the ones provided via BYO. The algorithm implementations vary slightly between BYO and the existing systems, but the high-level abstractions are the same.

We ran all graph containers in unweighted mode (and by extension, all algorithms) for simplicity.

Datasets. Table 3 lists the 10 graphs used in the evaluation and their sizes. All of the graphs included in the evaluation are undirected and unweighted for simplicity. We started with a selection of graphs from the GAP benchmark suite [16], a widely-used graph-evaluation specification. These include the the Road (RD) [1] network, the *Twitter* (TW) [16] graph, and the *Kron* graph [56] with the same parameters as Graph 500 (A=0.57, B=C=0.19, D=0.05) [4]. Much like in GAP, we include a uniform random graph *Erdős-Rényi* (ER) graph [41] generated with $n = 10^7$ and $p = 5 \cdot 10^{-6}$. We also generated an *rMAT* (RM) graph by sampling edges from an rMAT generator [23] with $a = 0.5; b = c = 0.1; d = 0.3$ to match the distribution commonly used in evaluating graph containers [35, 36, 67, 83]. Additionally, we include several other social network graphs: the *LiveJournal* (LJ) [13], *Community Orkut* (CO) [87], and *Friendster* (FS) graphs from the SNAP dataset [57]. For coverage from other application domains, we include the *papers100M* (PA) dataset from the Open Graph Benchmark [45]. Finally, we also use the *Protein* (PR) network graph, an induced subgraph available in the data repository of the HipMCL [12] algorithm. Unlike social network graphs, the protein network graph is not heavily skewed in terms of degree distribution. These inputs represent a wide range of inputs both in skewness and in size.

Table 4: The average performance of different GraphContainer API configurations with CSR as the underlying container. "Min" refers BYO with just the required functionality and "Full" refers to BYO with both required and optional functions described in Section 4.2. The remaining configurations are described with either what they add to min, or what they remove from full. The 95% and max columns show the 95th percentile and maximum slowdown over the full API across all algorithms and all graphs.

API configuration	*Slowdown over full API*		
	Average	*95%*	*Max*
Min (just `map_neighbors` and `num_vertices`)	10.69	231	1379
Min + `degree`	1.43	4.1	22.8
Min efficient (Min + `degree` + `num_edges`)	1.16	2.5	3.1
Full minus `num_edges`	1.31	2.78	22.9
Full minus `degree`	2.18	7.4	14.5
No early exit (Full minus both map early exit)	1.12	2.5	3.1
No parallel map (Full minus both parallel map)	1.01	1.3	1.9
Full minus `parallel_map_neighbors_early_exit`	0.98	1.03	1.1
Full (All required and optional functionality)	1.00	1.00	1.00

6.3 API Microbenchmarks

We start by performing a study to understand what are the important ways in which the graph processing system needs to interact with the underlying graph. We perform a set of experiments where we keep the graph data structure consistent (CSR) and only vary the API we use to run the algorithms. Table 4 reports the average slowdown relative to the full API (all functions implemented) of the different API configurations (missing some functions).

The *minimal efficient* API configuration with only `num_edges` and `degree` on top of the required functionality (`map` and `num_vertices`), BYO incurs 1.16× slowdown on average compared to the full API with all map variants. Furthermore, In terms of algorithms, the minimal efficient API incurs over 1.2× slowdown (on average across graphs) on BFS, Spanner, and LDD, which can be explained due to the lack of early exit, as these algorithms can all benefit from direction-optimization and early termination during edgeMap [15]. These results suggest that a data-structure developer that implements only a few basic functions such as `num_edges` and `degree` can achieve close to the best-possible performance from BYO on a majority of cases. The `num_edges` and `degree` functions are necessary for performance because they are used frequently in the edgeMap/vertexSubset graph-algorithm abstraction from Ligra/GBBS to determine the cutoff between sparse and dense mode for the vertexSubset [73].

Furthermore, Table 4 shows that implementing the more advanced maps does not significantly help: on average, omitting map with early exit incurs 1.12× slowdown and omitting parallel map incurs only 1.01× slowdown. However, there are specific algorithms and graphs on which these advanced maps are critical for performance. For example, in the worst case (LDD on KR), omitting early exit may incur up to 3× slowdown compared to the full API. Omitting early exit does not significantly affect performance in most cases: 65 cases incurred negligible (less than 1.03×) slowdown, 27 cases incurred more than 1.1×

slowdown, and 7 cases incurred more 2× slowdown compared to the full API. On the other hand, we found that the worst case (k-core on TW) for omitting parallel maps resulted in 1.8× slowdown compared to the full API; parallel map is especially important in high-degree graphs such as TW. However, the overall effect on performance from omitting parallel map was even less than from omitting early exit: without parallel maps, 84 cases incurred negligible (less than 1.03×) slowdown, 9 cases incurred more than 1.1× slowdown, and 0 cases incurred more than 2× slowdown compared to the full API.

Based on these results, we use all available API functionality besides `parallel_map_early_exit` in Sections 6.4 and 6.5.

6.4 Comparison with other frameworks

The goal of this section is to demonstrate that BYO is a good basis for our large-scale graph container evaluation by comparing it to several other graph frameworks, including some frameworks that introduced graph containers which we study in our benchmark. Our objective in this section is to show that graph containers that run algorithms with BYO do not lose out on performance compared to running with other frameworks. Many of the comparisons in this section are *not apples-to-apples* due to minor variations in algorithm implementations. However, it does show that BYO is competitive with existing work for end-to-end performance, and grounds the results and lessons we obtain through BYO in a high-performance starting point.

Figure 6 shows that BYO achieves competitive performance overall when compared with direct algorithm implementations from the GAP benchmark suite [16] as well as other high-performance frameworks including Ligra, GraphBLAS, and GBBS.

On average, GAP supports algorithms about 1.6× faster than BYO, but the majority of the performance gap comes from Connectivity (CC) because the two systems implement very different algorithms for the problem. GAP implements the Afforest algorithm [77], which is based on an idea similar to direction-optimization that enables the algorithm to potentially examine far fewer than all m edges. On the other hand, the GBBS algorithm we use is a concurrent version of union-find which examines every edge, and is the state-of-the-art for incremental connected components. For the problems that GAP and BYO implement the same algorithms for, the average performance gap narrows: GAP achieves about 1.15× speedup over BYO if we exclude CC from consideration. These results demonstrate that the cost of abstraction in BYO is relatively small when compared to the direct implementations from GAP.

The focus of this paper is on frameworks because they enable graph containers to easily express a diverse set of algorithms. Direct implementations are infeasible for large-scale evaluations because every algorithm must be integrated with every data structure, combinatorially increasing the amount of programming effort needed with every new data structure and algorithm.

In terms of frameworks, BYO achieves very similar performance (within about 1.05×) compared to GBBS, the starting point for BYO's implementation. Out of the frameworks we evaluated, Ligra/GBBS use similar abstractions based on vertexSubset/edgeMap [73]. GBBS builds upon Ligra with additional optimizations and functionality (e.g., bucketing [34]). BYO inherits these advancements from GBBS, and both GBBS/BYO achieve on average 1.7× speedup over Ligra. Finally, we also evaluate GraphBLAS [22, 30, 31, 53], a state-of-the-art

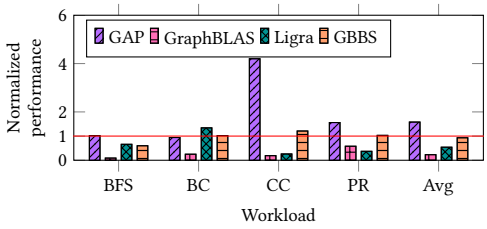

Figure 6: Relative performance normalized to BYO (up is good). A bar above 1 means there was speedup over BYO. Unlike the rest of this paper, this is not an apples-to-apples comparison because each system implements different algorithms and has been tuned for different graphs. This shows that BYO is competitive with the existing landscape of work, while still enabling flexibility in both container and algorithm choice. All systems besides GAP are frameworks, while GAP is a library of direct algorithm implementations. All systems use CSR as the graph representation.

graph-algorithm framework based on sparse linear algebra, and find that BYO achieves about 4× speedup over GraphBLAS. Our findings about GraphBLAS are consistent with a recent comparison of GAP and GraphBLAS [31] that shows that GraphBLAS is slower than GAP because it cannot access optimizations such as kernel fusion.

Comparing to the original systems. Furthermore, we verified that containers that run algorithms using BYO do not give up performance compared to their performance in the original systems (with different frameworks for algorithms). Specifically, when averaging across algorithms and graphs, BYO was between 1.1×−1.8× faster than the original frameworks when integrated with PMA (both uncompressed and compressed), SSTGraph, and CPAM.

6.5 Comprehensive container evaluation

At a high level, the tested graph data structures are very similar on average, but specialized data structures have an advantage over off-the-shelf structures in terms of worst-case performance across problem instances. Table 5 reports the average, 95th percentile, and maximum slowdown over CSR for each data structure across all 100 problem settings (10 algorithms × 10 graphs).

On average, we find that the overall difference between the best off-the-shelf dynamic structure and the best specialized dynamic structure is within about 1.1×. Specifically, the Abseil [2] B-tree combined with the inline optimization described in Section 4 incurs 1.22× slowdown compared to CSR. Furthermore, we find that the best specialized graph data structure on average is a vector of uncompressed PaC-trees [35] + inline, which incurred 1.11× slowdown relative to CSR.

The average differences between specialized structures are much smaller than previously reported in other papers because BYO standardizes the evaluation and makes optimizations previously available in one system accessible to all data structures. Specifically, we find that the specialized containers (PaC-trees, Terrace, DHB, CPMA, SSTGraph and Aspen) incur between 1.11−1.44× slowdown on average relative to CSR.

These results do not invalidate previous evaluations because this paper compares *containers* directly rather than overall *systems*. Previously, papers that introduced containers were only able to compare their systems (both the container and framework) because of the lack of a unified easy-to-use framework. Therefore, previously-reported

Table 5: Data structure algorithm performance and space usage. Each container's time is normalized to CSR's time averaged over all 100 settings of 10 algorithms × 10 graphs. A number closer to 1 means better performance (higher is worse). The 95% and max columns show the 95th percentile and maximum slowdown over CSR across all algorithms and all graphs. We also show the space usage of the different graph data structures in terms of bytes per edge.

Container	*Slowdown over CSR*			*Bytes per edge*		
	Average	*95%*	*Max*	*Min*	*Average*	*Max*
NeighborSet API (Vector of...)						
`absl::btree_set`	1.26	1.9	2.3			
`absl::btree_set` (inline)	1.22	2	2.6			
`absl::flat_hash_set`	1.40	2.3	3.4			
`absl::flat_hash_set` (inline)	1.29	2.1	2.6			
`std::set`	2.59	5.0	5.8			
`std::set` (inline)	2.37	4.9	5.6			
`std::unordered_set`	2.01	3.7	6.0			
`std::unordered_set` (inline)	1.90	3.5	5.9			
Aspen	1.22	2	2.5	5.7	12.0	53.4
Aspen (inline)	1.14	1.7	2.0	5.8	7.4	14.9
Compressed Aspen	1.44	2.1	2.6	3.4	5.0	12.1
Compressed Aspen (inline)	1.34	1.9	2.6	3.4	5.5	14.9
CPAM	1.16	1.4	1.5	4.1	4.9	9.0
CPAM (inline)	1.11	1.5	1.6	4.1	6.6	21.6
Compressed CPAM	1.37	1.7	1.9	3.4	4.5	8.9
Compressed CPAM (inline)	1.30	1.8	2.1	3.5	6.2	21.6
PMA	1.25	1.9	3.2	8.1	13.9	46.5
Compressed PMA	1.35	1.9	3.3	4.9	11.2	46.5
Tinyset	1.27	1.9	5.1	5.5	8.6	26.5
Vector	1.07	1.4	1.9	4.1	5.0	10.2
GraphContainer API						
CSR	1.00	1.0	1.0	4.1	5.1	10.6
Compressed CSR	1.23	1.5	1.6	2.3	3.8	10.6
DHB	1.15	1.7	2.4			
PMA	1.15	1.4	1.6	10.0	12.3	24.2
Compressed PMA	1.31	2.0	2.2	3.1	5.6	17.7
SSTGraph	1.25	1.5	2.4	4.0	6.4	19.9
Terrace	1.20	2.0	3.3	9.3	17.7	47.8

performance differences were the result of variations in the framework as well as the container. Additionally some existing work is designed for specific types of graphs or algorithms and thus tested on situations that they are expected to perform well.

Although the off-the-shelf and specialized data structures achieve similar performance on average, the specialized data structures have better overall performance when looking at the holistic set of experiments. Figure 2 shows for how many experiment settings a given data structure achieved within some slowdown relative to CSR. For example, Abseil's B-tree with the inline optimization achieved within 1.25× of CSR's performance on 63 experiments, while PaC-trees with the inline optimization achieved within 1.25× of CSR's performance on 83 experiments. In the worst case, Abseil's B-tree is 2.6× slower than CSR on the k-core algorithm on the KR graph due to a lack of parallel maps. In contrast, the vector of PaC-trees with the inline optimization, which is carefully designed to be space-efficient and cache-friendly and can leverage parallel maps, incurred only 1.16× slowdown for the same problem and graph. These results suggest that specialized data structures can mitigate performance variations on challenging instances such as high-degree graphs.

Space usage. We also measure the space usage of all containers which support getting their memory usage in Table 5. We find the bytes per edge varies significantly between graphs even when the container is fixed - by at least 2× and sometimes up to 10×. In all cases, the worst-case bytes per edge is on the road graph due to its low degree. Finally, compressed data structures can reduce the space usage by 2× compared their uncompressed counterparts.

Guidance for choosing graph containers

We first analyze how different specialized graph containers perform on different problem settings. Next, we compare the performance of different container configurations on the whole. Specifically, we analyze the effect of compression, the inline optimization (via the NeighborSet API) and the effect of collocated data (via the GraphContainer API).

Relationship between containers and problem settings. Table 6 shows the fastest container for each combination of graph and algorithm tested. These results provide guidance for choosing among containers for different graph and algorithm types.

Overall, we find that the optimized tree-based containers (CPAM and Aspen) achieve the best performance most frequently on different problem settings. CPAM performs especially well on the ER graph - it is the fastest on 7/10 algorithms. We conjecture that its performance is due to the uniform degree distribution and relatively high average degree in ER.

Several other containers exhibit strengths in specific algorithm or graph categories:

- The PMA is the fastest container on the *Coloring* algorithm for all graphs. Coloring is a covering-type algorithm that requires iterating over the entire graph in any order, which the PMA is well-suited due to being optimized for contiguous memory access.
- DHB achieves the best performance more often on *large graphs* (i.e., on PA, FS, and KR). We conjecture that DHB is well-suited to large graphs because it uses custom memory allocations that enable it to store more data contiguously.
- Terrace has the best performance on some algorithms (ADS, MIS, and PR) when run on the RMAT graph. Terrace is optimized for skewed graphs, and RMAT is a synthetic skewed graph.
- On very sparse graphs, e.g., RD, data structures with fewer pointers and co-located memory such as PMA, CPMA, and SSTGraph are the best choice due to the improved locality of these data structures when the average degree of the graph is extremely low.

To summarize, CPAM and Aspen are solid choices for overall performance but other systems are better for specific cases.

Next, we compare classes of data structures and the optimizations possible through the different APIs in BYO.

Effect of compression on containers. These results demonstrate that compression does not help performance on the tested data structures and graphs. Specifically, the compressed versions of CSR, Aspen, PaC-trees, and PMA incurred between 1.1−1.23× additional slowdown over CSR compared to the uncompressed versions.

These results stand in contrast to previous work that demonstrated speedups due to compression [35–37, 74, 83]. Although the algorithms are still memory bound, the computational overhead from decompression impacts performance in algorithms in BYO because margins for optimization and overall improvement are smaller, so any additional work has a more pronounced effect on performance.

We focus our evaluation in this paper on graphs with up to billions of edges to match commonly-used dataset sizes specified by graph benchmark suites such as GAP [16] and LDBC Graphalytics [46]. However, compression is important for feasibility as well as performance as graphs scale even larger to hundreds of billions of edges. Without compression, these graphs cannot fit in memory.

Table 6: The fastest container for every graph × algorithm combination. * next to a container denotes the NeighborSet API version with the inline optimization. C-CPAM refers to the compressed version of CPAM. PMA(V) refers to the vector of PMAs using the NeighborSet API, and PMA refers to the single PMA using the GraphContainer API.

	RD	*LJ*	*CO*	*RM*	*ER*	*PR*	*TW*	*PA*	*FS*	*KR*
BFS	CPMA	CPAM*	Aspen*	CPAM*	CPAM*	TinySet	CPAM*	Aspen*	CPAM	CPMA
BC	absl::FHS*	CPAM*	Aspen*	CPAM*	CPAM*	DHB	Aspen*	Aspen*	Aspen*	CPAM*
Spanner	PMA	CPAM*	CPAM*	Aspen*	CPAM*	Aspen*	DHB	Aspen*	DHB	DHB
LDD	PMA	CPAM*	CPMA	CPAM*	CPAM*	DHB	CPMA	CPAM*	CPMA	CPMA
CC	absl::FHS*	Aspen*	Aspen	Aspen	Aspen	Aspen	Aspen	DHB	DHB	DHB
ADS	SSTGraph	TinySet	SSTGraph	Terrace	CPAM	absl::btree	PMA	DHB	DHB	DHB
KCore	C-CPAM*	C-CPAM*	CPAM	SSTGraph	SSTGraph	TinySet	CPAM	CPAM*	CPAM*	Aspen*
Coloring	PMA	PMA	PMA	PMA	PMA	PMA	PMA	PMA	PMA	PMA(V)
MIS	PMA	CPAM*	TinySet	Terrace	CPAM	TinySet	Aspen*	CPAM*	Aspen*	Aspen*
PR	DHB	Aspen*	Aspen	Terrace	CPAM*	Aspen	Aspen*	TinySet	PMA (V)	DHB

Effects of inline optimization in NeighborSet API. Table 5 demonstrates that the inline optimization improves performance of set data structures on average by 1.06× (between 1.03−1.09× averaged across both graphs and algorithms). The inline optimization was introduced in the Terrace [67] graph system as part of its hierarchical data structure design. However, we incorporate this optimization into the NeighborSet API in BYO to benefit all data structures.

Advantages of GraphContainer API. Finally, we find that collocating data between sets (enabled by the GraphContainer API described in Section 4.2) improves performance by between 1.02−1.1× because it reduces indirections between neighbor lists. For example, as shown in Table 5, the vector of vectors (the NeighborSet API version of CSR) incurs 1.07× slowdown over CSR. Similarly, the single PMA is faster on average compared to the vector of PMAs (one per vertex), and SSTGraph is faster on average compared to the vector of tinyset (the NeighborSet API version of SSTGraph). These results demonstrate that collocating the data improves performance on a diverse set of algorithms and graphs.

6.6 Batch-insert evaluation

We measure batch-insert throughput, an important feature of dynamic-graph containers, and report the results in Figure 3.

Setup. To evaluate update throughput, we first insert all edges from the TW graph into the system under test. We then insert a new batch of directed edges (with potential duplicates) to the existing graph in the system under test and then delete that batch of edges from the graph. We repeat this procedure 3 times per batch. To generate edges for inserts/deletes, we sample directed edges from an RMAT generator [23] (with $a = 0.5; b = c = 0.1; d = 0.3$, matching the distribution from prior work on dynamic-graph systems [35, 67, 83]).

Discussion. The off-the-shelf data structures exhibit the classical tradeoff between algorithm and update performance. The fastest off-the-shelf container for inserts is Abseil's flat hash set, which achieved between 3× speedup over Abseil's B-tree on inserts for large batches. On the other hand, Abseil's flat hash set incurred more slowdown on algorithms over CSR compared to Abseil's B-tree, as shown in Figure 2 and Table 5. We plot the set container variants with the inline optimization because we found they did not impact the insertion throughput greatly.

Specialized structures can overcome the algorithm-update tradeoff on the largest batches with special support for batch inserts. For example, the batch-parallel PMA (using the GraphContainer API) achieved 2−5× speedup over the Abseil B-tree on the larger batches due to the PMA's native support for parallel batch inserts. Furthermore, we found that DHB, a hash-based data structure that uses the GraphContainer API, achieves 1 − 3× speedup over the B-tree on the batch insertions. Both the PMA and DHB achieve better overall performance on algorithms compared to the B-tree (Table 5).

Other specialized structures may trade update speed for algorithm performance compared to the B-tree. For example, CPAM (with inline) incurred 1.15× slowdown compared to the B-tree (with inline) on batch inserts when averaging across batch sizes. Furthermore, we found that Terrace was about 10× slower than the B-tree for batch inserts on average across batches because Terrace maintains many edges in a concurrent contiguous PMA and therefore cannot trivially parallelize across vertices.

These results suggest there is potential in developing specialized batch-parallel data structures that overcome the tradeoff for batch inserts without giving up on algorithm performance. The results also show the importance of how the data needs to be processed before it can actually be inserted into the data structures. The fastest approaches for large batches are those that only require a semi-sort. The fastest for mid sized batches and second fastest for large batches require a full sort, but that perform merges for the data.

7 CONCLUSION

This paper introduces BYO, an easy-to-use, high-performance, and expressive graph-algorithm framework. BYO enables apples-to-apples comparisons between dynamic-graph containers by decoupling the graph containers from algorithm implementations. The BYO interface is simple, enabling comprehensive comparisons of new containers on a diverse set of applications with minimal programming effort.

We have conducted a large-scale evaluation of 27 graph containers using BYO to express algorithms. The results demonstrate that the differences between graph containers are smaller than what is commonly reported in papers introducing new graph containers. We attribute this discrepancy to the fact that these papers often perform end-to-end comparisons between graph systems, which vary both the framework and the container. Moreover, our results demonstrate that while on average off-the-shelf data structures achieve highly competitive performance with specialized data structures, they leave significant performance on the table for certain algorithms/graphs.

We believe that BYO will spark exploration into specialized data structures for specific cases where off-the-shelf data structures do not do well, as well as on data structures that support fast updates without giving up on algorithm performance.

ACKNOWLEDGMENTS

This work is supported by NSF grants CCF-2103483, CCF-2238358, CCF-2339310, CNS-2317194, IIS-2227669, and OAC-2339521. We thank the anonymous reviewers for their useful comments.

REFERENCES

[1] [n.d.]. 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/challenge9/. Accessed: 2023-09-25.

[2] [n.d.]. Abseil. https://abseil.io/. Accessed: 2023-09-23.

[3] [n.d.]. Galois. Available at https://iss.oden.utexas.edu/?p=projects/galois. Accessed 10-20-2023.

[4] [n.d.]. Graph500 benchmark. https://graph500.org/.

[5] [n.d.]. LAGraph. https://github.com/GraphBLAS/LAGraph. Accessed: 2023-07-06.

[6] [n.d.]. std::set. https://en.cppreference.com/w/cpp/container/set. Accessed 10-25-2023.

[7] [n.d.]. std::unordered_set. https://en.cppreference.com/w/cpp/container/unordered_set. Accessed 10-25-2023.

[8] [n.d.]. std::vector. https://en.cppreference.com/w/cpp/container/vector. Accessed 10-25-2023.

[9] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph Analytics on Evolving Data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 133–145. https://doi.org/10.1145/3575693.3575713

[10] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 71–80. https://doi.org/10.1109/CCGrid54584.2022.00016

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

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

[13] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 44–54.

[14] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (1972), 173–189.

[15] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[16] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

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

[18] D. K. Blandford, G. E. Blelloch, and I. A. Kash. 2003. Compact representations of separable graphs. In *SODA*.

[19] D. K. Blandford, G. E. Blelloch, and I. A. Kash. 2004. An experimental analysis of a compact graph representation. In *ALENEX*.

[20] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices* 30, 8 (1995), 207–216.

[21] Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.

[22] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017. Design of the GraphBLAS API for C. In *2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 643–652.

[23] 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.

[24] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. 2020. Accelerating truss decomposition on heterogeneous processors. *Proceedings of the VLDB Endowment* 13, 10 (2020), 1751–1764.

[25] Dan Chen, Chuangyi Gui, Yi Zhang, Hai Jin, Long Zheng, Yu Huang, and Xiaofei Liao. 2022. Graphfly: efficient asynchronous streaming graphs processing via dependency-flow. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[26] Zheng Chen, Feng Zhang, JiaWei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. 2023. CompressGraph: Efficient Parallel Graph Analytics with Rule-Based Compression. *Proc. ACM Manag. Data* 1, 1, Article 4 (may 2023), 31 pages. https://doi.org/10.1145/3588684

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

[28] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[29] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[30] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (dec 2019), 25 pages. https://doi.org/10.1145/3322125

[31] Timothy A. Davis. 2023. Algorithm 10xx: SuiteSparse:GraphBLAS: Parallel Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* (jan 2023). https://doi.org/10.1145/3577195 Just Accepted.

[32] Timothy A. Davis. 2023. User Guide for SuiteSparse:GraphBLAS. https://github.com/DrTimothyAldenDavis/GraphBLAS/blob/stable/Doc/GraphBLAS_UserGuide.pdf.

[33] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1053–1066.

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

[35] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-Trees: Supporting Parallel and Compressed Purely-Functional Collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3519939.3523733

[36] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *PLDI*. 918–934.

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

[38] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Portland, OR, USA) *(GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/3398682.3399168

[39] Xiaojun Dong, Yunshu Wu, Zhongqi Wang, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2023. High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*.

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

[41] Paul Erdös and Alfréd Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae Debrecen* 6 (1959), 290–297.

[42] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1719–1732.

[43] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-Millisecond Per-Update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 513–527. https://doi.org/10.1145/3448016.3457263

[44] Yan Gu, Julian Shun, Yihan Sun, and Guy E Blelloch. 2015. A top-down parallel semisort. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. 24–34.

[45] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[46] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. 2016. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1317–1328.

[47] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. Supercomputing.

[48] Alon Itai, Alan G. Konheim, and Michael Rodeh. 1981. A sparse table implementation of priority queues. In *ICALP*. 417–431.

[49] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 337–355.

[50] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 17–32.

[51] Nicolai M Josuttis. 2012. The C++ standard library: a tutorial and reference. (2012).

[52] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-truss decomposition on multicore systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[53] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[54] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 65–76.

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

[56] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. 2005. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *European conference on principles of data mining and knowledge discovery*. Springer, 133–145.

[57] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Available at http://snap.stanford.edu/data.

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

[59] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: Sparsity-aware incremental processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 83–98.

[60] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[61] Tim Mattson, Timothy A Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. 2019. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 276–284.

[62] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. 2020. Shared Arrangements: Practical Inter-Query Sharing for Streaming Dataflows. *Proc. VLDB Endow.* 13, 10 (jun 2020), 1793–1806. https://doi.org/10.14778/3401960.3401974

[63] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow.. In *CIDR*.

[64] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[65] David R Musser, Gilmer J Derge, and Atul Saini. 2001. *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Longman Publishing Co., Inc.

[66] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. 1996. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.".

[67] 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*. 1372–1385.

[68] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2150–2159.

[69] Tao B Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 189–203.

[70] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. Graphin: An online high performance incremental graph processing framework. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22*. Springer, 319–333.

[71] Jessica Shi and Julian Shun. 2022. Parallel algorithms for butterfly computations. In *Massive Graph Analytics*. Chapman and Hall/CRC, 287–330.

[72] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*. 417–430.

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

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

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

[76] Steven W Smith et al. 1997. The scientist and engineer's guide to digital signal processing. (1997).

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

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

[79] Alexander van der Grinten, Maria Predari, and Florian Willich. 2022. A fast data structure for dynamic graphs based on hash-indexed adjacency blocks. In *20th International Symposium on Experimental Algorithms (SEA 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[80] Jan Van Leeuwen. 1991. *Handbook of theoretical computer science (vol. A) algorithms and complexity*. Mit Press.

[81] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[82] Brian Wheatman and Randal Burns. 2021. Streaming sparse graphs using efficient dynamic sets. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 284–294.

[83] Brian Wheatman, Randal Burns, Aydin Buluc, and Helen Xu. 2024. CPMA: An efficient batch-parallel compressed set without pointers. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 348–363.

[84] Brian Wheatman and Helen Xu. 2018. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

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

[86] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 824–833.

[87] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* abs/1205.6233 (2012). arXiv:1205.6233 http://arxiv.org/abs/1205.6233

[88] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.