The Distributed Computing Column

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

This month, in the Distributed Computing Column, Helen Xu is discussing everything you need to know about how to design efficient parallel applications that operate on dynamic graphs! She focuses on three key aspects: (1) How do you design the *containers* (i.e., data structures) that encapsulate the dynamic graph? (2) What is the right framework (i.e., interface) for interacting with a dynamic graph? (3) How do you fairly benchmark the performance of your parallel application for dynamic graphs?

To answer these questions, she discusses two main results. First, she presents a new container for dynamic graphs called *F-Graph*. F-Graph is a multicore batch-parallel dynamic-graph system that is optimized for spatial locality. It is built on top of a batch-parallel packed-memory array, yielding fast performance for a variety of graph applications. Next, she presents BYO, a unified framework for large-scale graph containers designed to facilitate benchmarking. BYO provides a simple and abstract container API, along with a clean interface. She then uses BYO to evaluate 27 different graph containers on 10 different graph algorithms using 10 large graph datasets. The resulting data illuminates the issues and trade-offs involved in designing parallel applications for dynamic graphs.

Overall, then, this article by Helen Xu provides significant insight into both the theory and practice of efficient parallel computation for dynamic graphs.

The Distributed Computing Column is particularly interested in contributions that propose interesting new directions and summarize important open problems in areas of interest. If you would like to write such a column, please contact me.

Dynamic Graphs, End-to-End: Containers, Frameworks, and Benchmarks

Helen Xu
Georgia Institute of Technology

1 Introduction to dynamic-graph containers

Dynamic graphs, or graphs that change over time, underlie many applications such as social networks [7], protein interactions [6], and paper citation networks [45]. Systems for dynamic graphs need to efficiently 1) apply a stream of updates (e.g., edge insertions and deletions) and 2) run queries (i.e., algorithms) on the updated graph. Unfortunately, these two objectives often conflict with each other [86].

In this column, I will address two main questions: 1) how to develop efficient parallel dynamic-graph data structures, or ***containers***, that support both fast updates and fast queries and 2) how to comprehensively and fairly benchmark dynamic-graph containers.

Query-update tradeoff. Let us examine several canonical graph containers to concretely understand the tradeoff between updates and queries. These examples will illustrate the challenges to supporting both fast updates and fast queries without giving up performance along either axis.

Perhaps the most classical graph data structure is the ***adjacency matrix***. Given a graph with n vertices, the associated adjacency matrix is an $n \times n$ matrix A where element A_{uv} is 1 if there is an edge from vertex u to vertex v, and 0 otherwise. The adjacency matrix format is ideal for edge updates - adding an edge to a graph stored in this format takes only $O(1)$ time to find and set the correct location in the matrix. Finding the existence of any particular edge also takes $O(1)$ time. It would seem that if one had $O(n^2)$ space, the adjacency matrix would be an ideal graph container.

In practice, however, the main operation underlying graph algorithms is a ***scan***, or iteration, through a vertex's neighbors, rather than individual edge-existence queries [69]. One common pattern in graph algorithms is processing a source vertex and adding its neighbors to the "active set" for the next round of processing. Finding and adding all neighbors of a given vertex to a set requires a vertex scan. Figure 1 provides concrete examples of how vertex scans are the main workhorse of graph algorithms.

Furthermore, many real-world graphs exhibit ***sparsity***: i.e., they contain many fewer edges than the total possible number of edges [64]. That is, almost all vertices have degree much less than $O(n)$. For example, the LiveJournal graph [7] has about 4.8 million vertices, but its average degree is only about 18. Therefore, an ideal graph data structure would support a scan of a given vertex v's neighbors in $O(\text{degree}(v))$ time. However, an adjacency matrix requires $O(n)$ time to scan the neighbors of any vertex, regardless of degree.

Therefore, the most popular graph data structure in high-performance graph applications is not the adjacency matrix but rather the ***Compressed Sparse Row*** (CSR) [72] representation. CSR stores a graph with m edges and n vertices in two arrays: an edge array A to store m neighbor

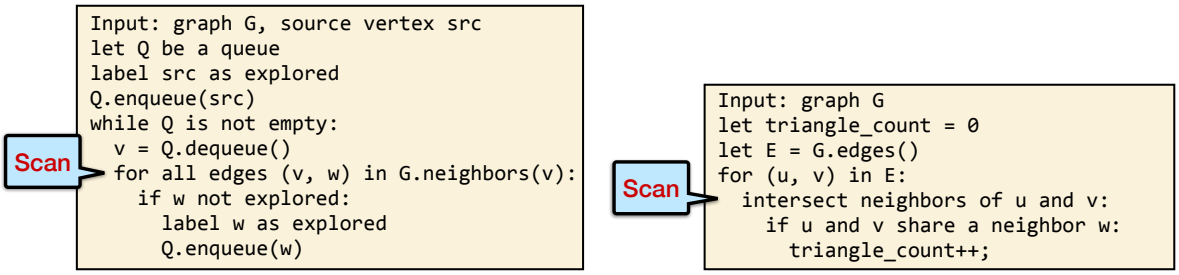

Figure 1: Example of how vertex scans underlie two fundamental graph algorithms: (a) breadth-first search and (b) triangle counting.

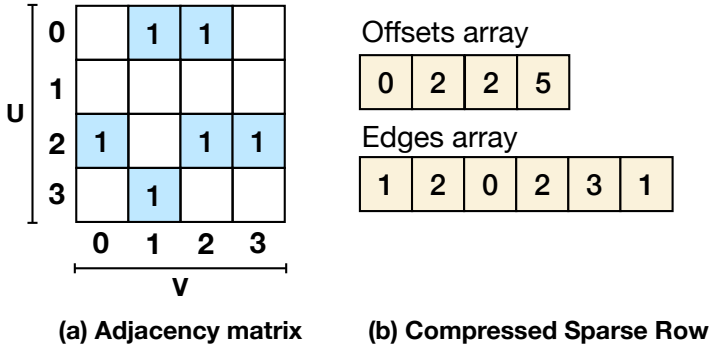

Figure 2: Example of the same graph stored in (a) adjacency matrix format and (b) Compressed Sparse Row format.

ids (for the edges) and an offsets array O to store n start indices (one per vertex). The neighbors of each vertex v form a contiguous fragment of A, so scanning over a given vertex v's neighbors only requires $O(\text{degree}(v))$ time. Furthermore, the neighbor ids of the edges are laid out contiguously in memory, so CSR exhibits good spatial locality and supports fast scans. However, CSR was not designed for dynamic graphs and is therefore prohibitively expensive to update: inserting a single edge may require $\Theta(m)$ time to move all of the other edges in the contiguous edge array. Figure 2 provides an example of a graph stored in adjacency matrix and CSR format.

High-performance dynamic-graph containers for multicores. To address this limitation, significant research effort over the past decade has been devoted to developing efficient dynamic-graph containers [38, 53, 32, 54, 34, 63, 75, 78, 82, 83, 33, 80, 52, 77, 43, 67, 24, 66, 84, 87].

The goal of this line of work is to introduce data structures to support both **efficient updates and algorithms** for dynamic graphs. Figure 3 provides a cartoon illustration of the query-update tradeoff and intuition for the data-structure design objectives.

When it comes to developing fast dynamic-graph data structures on modern multicores, codes must be optimized for the core features of modern multicore machines - many threads, large main memories, and a steep memory hierarchy [86]. Since today's multicores include large main memories (possibly 1TB or more), many of the dynamic-graph containers reside in memory. As we shall see, in-memory dynamic-graph containers must be optimized for locality to efficiently make use of both the memory subsystem as well as parallelism in multicores.

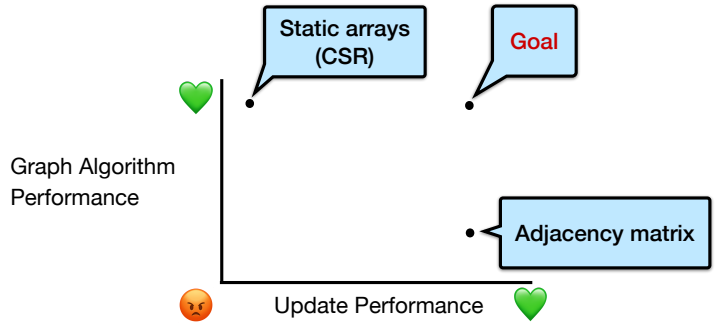

Figure 3: The query-update tradeoff and the goal of dynamic-graph containers.

2 F-Graph: A dynamic-graph system based on the Compressed Packed Memory Array

This section describes F-Graph, a multicore batch-parallel dynamic-graph system, as a case study for how to create efficient parallel dynamic-graph containers by optimizing for spatial locality [80][1]. As we will see, F-Graph overcomes the query-update tradeoff with a cache-optimized dynamic-array data structure. It supports both faster algorithms and faster updates compared to state-of-the-art dynamic-tree-based graph containers.

The associated artifact[2] was honored with the Best Artifact Award at PPoPP '24.

Batch-parallel dynamic-graph containers

Many modern data structure libraries, including dynamic-graph containers, parallelize ***batch updates*** that insert or delete multiple elements rather than point updates because each individual update is usually not worth parallelization due to their sublinear complexity [42, 9, 33, 34, 71, 39, 74, 8, 55]. Direct algorithmic support for batch updates simplifies parallelism and reduces the work per update by amortizing shared work between updates (e.g., searches for the target location in the data structure).

Batch-parallel data structures demonstrate the important role that the memory subsystem plays in good utilization of multithreaded parallelism and overall performance in practice. Almost all batch-parallel data structure implementations are based on pointer-based data structures such as trees [33, 34, 17, 42, 9, 71, 39, 74]. However, the main bottleneck in the parallel scalability of these implementations is not parallelism but memory bandwidth limitations due to pointer indirections during tree traversal [17, 42]. Dhulipala *et al.* [34, 33] address these issues by adding blocking and compression to improve spatial locality in trees. These batch-parallel cache-optimized trees form the basis for Aspen [34] and CPAM [33], two state-of-the-art dynamic-graph containers.

Despite these improvements, cache-optimized trees inherently leave performance on the table because they incur random memory accesses rather than reading memory contiguously [12, 63, 83, 85]. Theoretically, cache-friendly trees (e.g., B-trees [10]) are asymptotically optimal in the classical external-memory model [3] for both updates and scans. Empirically, however, tree-based data structures are over 2× slower to scan compared to array-based data

[1]The full version of the paper can be found on arXiv at `https://arxiv.org/abs/2305.05055`.

[2]The artifact and library are available at `https://github.com/wheatman/Packed-Memory-Array`.

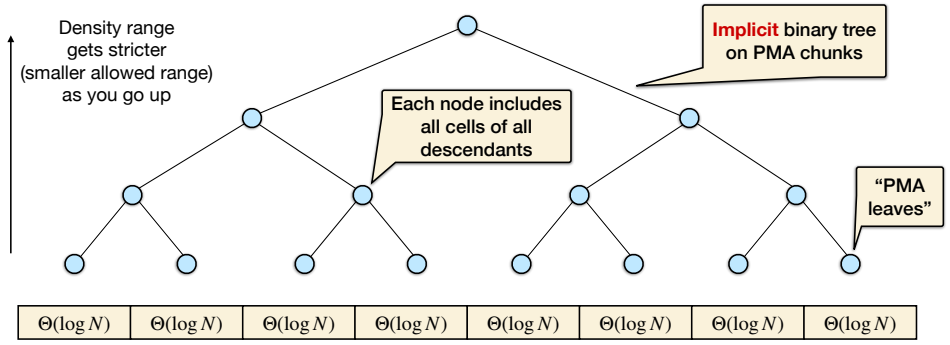

Figure 4: The high-level structure of a PMA and how the single flat array defines an implicit tree.

structures support scans because arrays avoid pointer chasing and therefore take advantage of sequential memory access [63, 79, 85].

Optimizing for sequential access with PMAs The Packed Memory Array (PMA) [47, 13, 14], a dynamic array-based data structure optimized for cache-friendliness (i.e., spatial locality), is a promising candidate for a batch-parallel dynamic-graph container. So far, the PMA has appeared in domains such as graph processing [66, 82, 30, 83, 32, 63, 78], particle simulations [37], and computer graphics [73].

Even though the PMA has appeared in several systems for dynamic graphs, past implementations exhibit low update throughput compared to batch-parallel trees because the PMAs did not include direct algorithmic support for parallel batch updates [83]. Previous work [31] introduced a serial batch-update algorithm for PMAs, but stopped short of parallelization.

Adding batch-parallelism to the Packed Memory Array

PMA structure. The primary feature of a PMA is that it stores its data in sorted order in one contiguous array, which enables fast cache-efficient scans through the elements [47, 13, 14]. To enable efficient updates, the PMA also stores (a constant factor of) empty spaces between its elements to reduce data movement upon updates. Specifically, a PMA with n elements uses $N = \Theta(n)$ cells.

The PMA defines an implicit binary tree which is used during insertions to maintain the proper amount of empty spaces throughout the array. The array is logically (not physically) divided up into leaves of size $\Theta(\log(N))$ cells, so the implicit tree has $\Theta(N/\log(N))$ leaves and height $\Theta(\log(N/\log(N)))$. Every node in the PMA tree corresponds to a ***region*** of cells. Each leaf $i \in \{0,\ldots,N/\log(N)-1\}$ has the region $[i\log(N),(i+1)\log(N))$, and each internal node's region encompasses all of the regions of its descendants. The ***density*** of a region in the PMA is the fraction of occupied cells in that region. Figure 4 illustrates a PMA and its corresponding implicit tree.

Serial inserts in PMAs PMA inserts use the implicit tree to maintain the overall structure. As shown in Figure 5(a), a PMA insert first ***searches*** for the target leaf that the element should go in the sorted order. It then ***places*** the element at the correct location in that leaf. The density bounds guarantee that there is always at least one free cell to place an element in each leaf. Next, it ***counts*** the cells in all necessary nodes in the PMA implicit tree, traversing up until it finds a node that does not violate its density bound. Finally, based on the results from the count, the

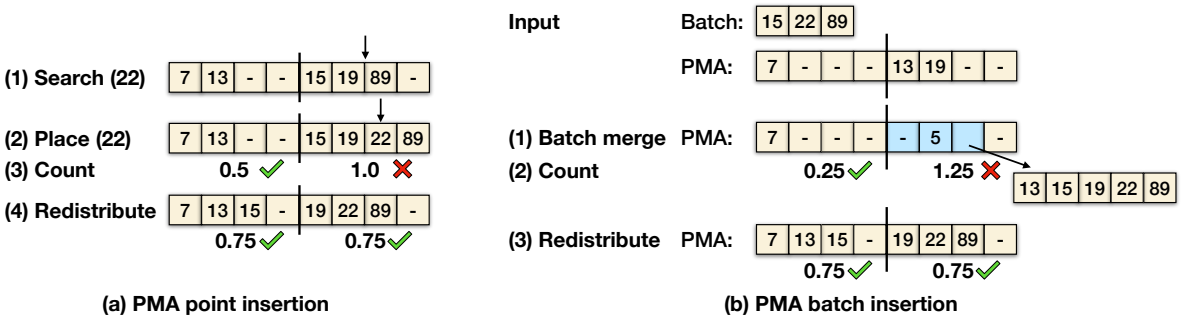

Figure 5: Example of how to perform (a) point inserts and (b) batch inserts in a PMA. The numbers in the count step are the densities of each leaf. In this example, the density bound is 0.9. As shown in step (1), the batch-merge step may overflow a leaf if there are not enough free cells. In that case, elements are stored out of place until the redistribute phase. The full paper contains more details about all steps of the batch-insert algorithm.

PMA ***redistributes*** elements equally among leaves in the node it counted up to, resolving the density bounds in all of its descendants.

Parallel batch-update algorithm for PMAs. The ***batch-insert***[3] ***algorithm*** for PMAs takes as input a PMA with n elements and a batch with k sorted elements to insert. An unsorted batch can be converted into a sorted batch in $O(k\log(k))$ work.

The batch-insert algorithm for PMAs is optimized for the case when the batch is neither too small nor too large. At one extreme, if k is very small (e.g., $k < 100$), the overheads from the batch-insert algorithm outweigh the benefits, so point updates are more efficient than batch updates. At the other extreme, if k is large (e.g., $k \geq n/10$), the optimal algorithm is to rebuild the entire data structure with a linear two-finger merge. The batch-insert algorithm for PMAs performs local merges to address the intermediate case between these two extremes.

At a high level, the batch-insert algorithm for PMAs relies on recursive local merges of the batch elements to the correct PMA leaves in the data structure. In addition to the implicit tree for densities, the PMA also defines another implicit binary search tree on the PMA leaves, where each node is one leaf in the tree (as opposed to the internal nodes encompassing multiple leaves as in the density tree). The recursion begins in the middle leaf of this PMA binary search tree, merges in the appropriate elements from the batch, and in parallel, recurses on the two halves. There are some similarities to batch-insert algorithms for trees, which are implemented with unions/differences [17]. Figure 5(b) provides a worked example of a batch insert in a PMA.

There are two unique challenges to parallel batch updates in PMAs: 1) potentially overflowing leaves with local merges, and 2) efficiently determining which regions to redistribute after the batch merges (via counting). Since multiple leaves may be written to in parallel during the batch merge step, the algorithm cannot overwrite neighboring leaves, even if there are not enough cells to accommodate all elements destined for a given leaf. To address this issue, the batch-insert algorithm for PMAs may store some elements out-of-place temporarily. Furthermore, naively parallelizing the counting step to determine densities (and therefore the regions to redistribute) over the elements in the batch is technically correct because the counting

[3]The batch-parallel PMA supports both inserts and deletes, but we focus on the insert case for ease of presentation. Deletes are implemented symmetrically to inserts, and the full version of the paper contains experiments for both.

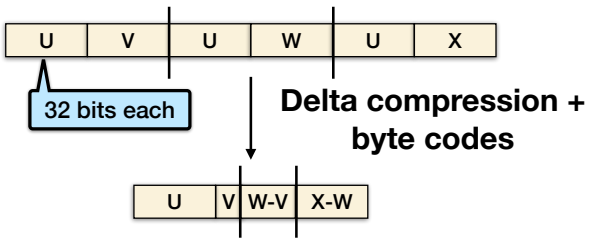

Figure 6: An example of how to store the edges (u,v), (u,w), and (u,x) (assuming $v < w < x$) in F-Graph.

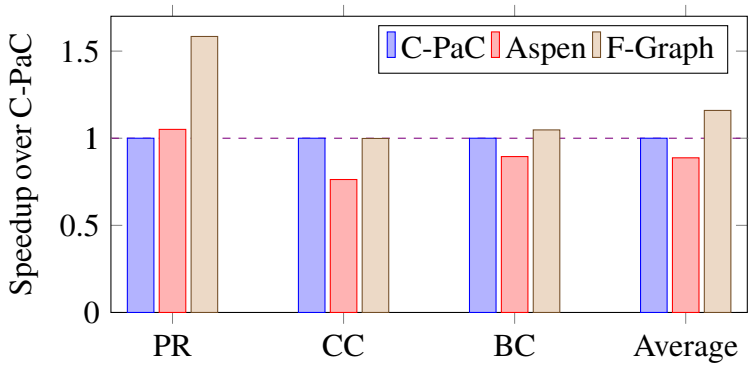

Figure 7: Relative speedup of graph algorithms over C-PaC. The algorithms tested are PageRank (PR), Connected Components (CC), and single-source Betweenness Centrality (BC). The algorithms were tested on a suite of graphs ranging in size from about 86 million edges to about 3.6 billion edges.

is a read-only operation, but may perform a great deal of redundant work. Please see the full paper for details about the batch-insert algorithm, which relies on an efficient parallel counting algorithm to achieve the desired asymptotic bounds.

Using the batch-parallel PMA as a dynamic-graph container

Storing a graph in a compressed flat array F-Graph[4] is built on a single batch-parallel PMA that stores the entire graph as a list of edges in sorted order. It differs from traditional graph representations because it uses only a single array to store both the vertex and edge data. To understand the difference, recall the traditional CSR representation, which stores the graph in two arrays. The offsets array saves space: the edges array then only needs to store destinations and not sources.

As an additional optimization, the PMA underneath F-Graph is compressed with delta compression and byte codes [16, 15, 70]. With delta compression, the Compressed PMA (CPMA) stores differences (deltas) between elements rather than the full elements in all elements except the first in each PMA leaf. Figure 6 illustrates how delta compression saves space by eliding out the source vertex in almost all of the edges (except the start of each leaf and and the first edge of each vertex).

[4]F-Graph uses only one compressed PMA (a flat array) to store the graph. The F in F-Graph comes from the musical key of F, which has one flat.

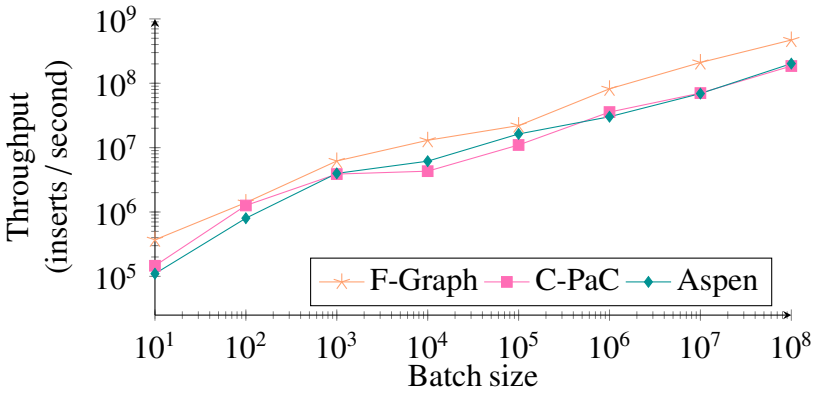

Figure 8: Insert throughput as a function of batch size on the Friendster graph (num. vertices ≈ 125 million, num. edges ≈ 3.6 billion). Edges to add were generated according to an rMAT distribution [20].

Results The empirical evaluation demonstrates that F-Graph is on average 1.2× faster on a suite of graph algorithms, achieves 2× faster throughput for batch updates, and uses marginally less space to store the graphs compared to C-PaC, a state-of-the-art dynamic-graph system based on blocked compressed trees [33]. Furthermore, F-Graph is on average 1.3× faster on graph algorithms, achieves 2× faster throughput for batch updates, and uses 0.6× space to store the graphs compared to Aspen, another high-performance tree-based dynamic-graph system [34].

The full paper includes a much more thorough evaluation, but this article includes a couple of representative plots in Figures 7 and 8. All experiments were run on a 64-core 2-way hyper-threaded Intel machine with 256 GB of memory from AWS [5].

The empirical results demonstrate the potential for dynamic-graph containers to overcome the query-update tradeoff by optimizing for spatial locality. Although cache-optimized trees theoretically dominate[5] PMAs on inserts and match PMAs on scans, in practice, trees are slower to scan because of pointer indirections. The basic theoretical models do not capture the cost of these random accesses, but they have a significant effect on empirical performance. Furthermore, the PMA's cache-friendliness enables F-Graph to supports batch updates much faster than the theoretical bound suggests and even faster than cache-optimized trees because of its locality[6].

Discussion

The empirical advantage of F-Graph, a PMA-based dynamic-graph container, over C-PaC and Aspen, two tree-based dynamic-graph containers, demonstrates the importance of optimizing parallel data structures for the memory subsystem. Specifically, the CPMA's array-based layout enables it to take advantage of the speed of contiguous memory accesses. Despite the theoretical prediction, the batch-parallel CPMA empirically overcomes the query-update tradeoff due to its locality.

[5]Given a cache-line size B and n elements, PMAs support inserts in $\Omega(\log(n)+\log^2(n)/B)$ cache-line transfers, while B-trees support inserts in $O(\log_B(n))$ cache-line transfers in the external-memory model [3].

[6]In microbenchmarks, the PMA was shown to have many fewer L1 and L3 misses when compared to cache-optimized trees.

3 Fair and comprehensive benchmarking of graph containers

The previous sections gloss over an important question in end-to-end system development - *how does the overall dynamic-graph system implement and run the algorithms?* Let us take a step back to examine overall dynamic-graph system performance, since the goal is to efficiently perform both algorithms and updates on the graph. The choice of container is a key decision for holistic system performance, but it is not the only factor.

Structure of dynamic-graph systems. General graph-processing systems have two main components: the ***programming framework*** and ***graph container***. 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.

So far, we have focused on optimizing the graph container in the previous sections, but the programming framework is an equally important factor in graph-algorithm performance. Since both components are necessary for good performance, significant research effort has been devoted to developing and optimizing both sides. On the graph framework side, researchers have developed many high-performance abstractions, such as Ligra [69], the Graph Based Benchmark Suite (GBBS) [35], and the GraphBLAS [28, 27, 51]. Previous work has shown that these abstractions can achieve competitive performance with hand-optimized implementations such as those from the GAP benchmark suite [11].

Issues with current methods for creating dynamic-graph systems

Although there has been great progress on developing and optimizing both the programming framework and graph container, these two directions have mostly been independent lines of work. An ideal dynamic-graph system would combine advancements in both dynamic-graph programming frameworks and containers, as shown in Figure 9. However, integrating different components is challenging because often the framework and container implementations are tightly coupled.

Comprehensiveness of system. The separation between framework and container development results in systems that are limited in terms of performance, capabilities, and generality.

On the frameworks side, for example, the Ligra/GBBS/GraphBLAS abstractions express algorithms in terms of basic data-access primitives to build algorithms that could be implemented by any data structure, but the current codebases use CSR as their representation for simplicity. Although CSR enables good graph-algorithm performance, as we have discussed, it does not support updatability. Furthermore, there are lines of research focused on developing incremental [23, 60, 49, 59, 68, 58] and dynamic [65, 76, 57, 56, 2, 21, 48, 41] algorithm frameworks, but these also implement ad-hoc data structures underneath the framework, leaving performance on the table.

On the container side, systems that include a new dynamic-graph container often run algorithms with either 1) direct implementations of kernels on top of the container [32, 4, 46, 38, 82] or 2) ad-hoc implementations of frameworks [63, 80, 83, 34, 33]. Direct implementations can achieve good performance but limit system generality, since adding new algorithms requires

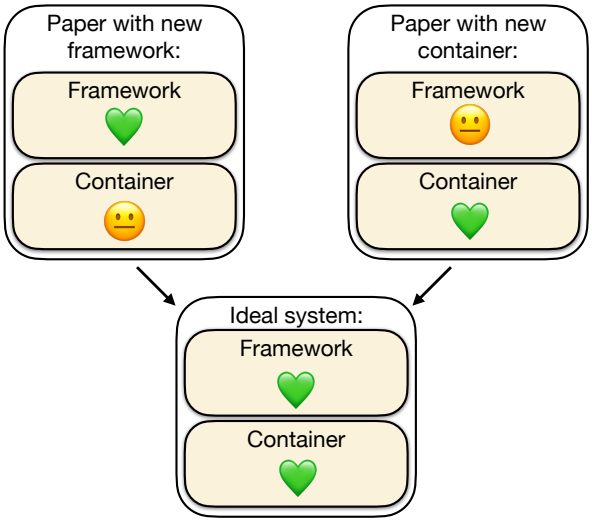

Figure 9: A cartoon to illustrate an ideal-dynamic graph system that combines advances in both dynamic-graph frameworks and containers.

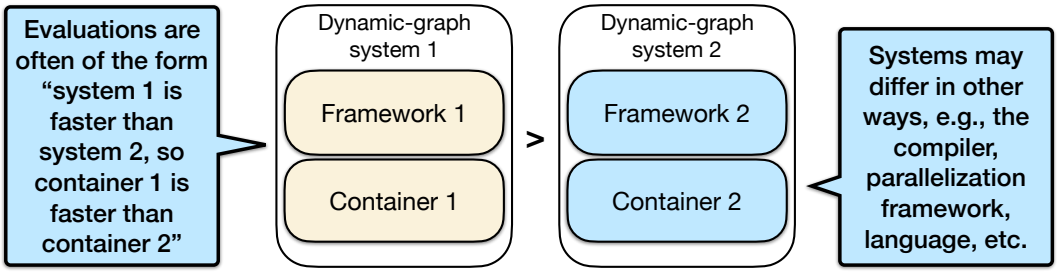

Figure 10: A cartoon to illustrate the current practice of evaluating end-to-end dynamic-graph systems. Even if a paper claims to introduce a new framework or container, they still must implement the other component, which can affect the overall system performance.

integrating the implementation with each container individually. As a result, systems that use direct implementations usually compare containers on a relatively small set of algorithms. Ad-hoc frameworks enable more general evaluations, but they are not as optimized or expressive compared to implementations that focus on the framework.

Benchmarking graph containers alone. Since the graph-algorithm implementation (either direct or via a framework) and container are usually highly intertwined, most if not all papers introducing new dynamic-graph containers perform *end-to-end* comparisons with existing systems. Although these comparisons aim to compare the containers because the only novelty is in the container, there may be counfounding factors from the rest of the system. As a concrete example, prior evaluations of the dynamic-graph systems SSTGraph [78] and CPAM [33] compare with earlier dynamic-graph systems such as Aspen [34], but change not only the container but also important graph-algorithm details, making the source of any measured improvements unclear. Figure 10 illustrates a high-level view of the difficulty when making container comparisons from end-to-end systems evaluations.

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× speedup over Aspen [78].

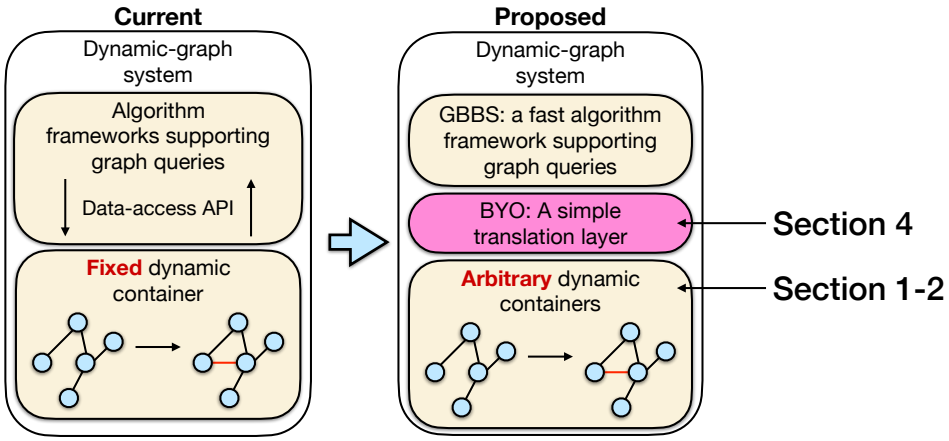

Figure 11: Relationship between graph-algorithm frameworks and dynamic-graph containers in graph-processing systems. BYO provides a simple translation layer between frameworks and containers to enable arbitrary containers to easily plug into the framework.

- Terrace finds a 1.7−2.6× speedup over Aspen [63].
- Aspen finds 1.8−15× speedup over prior dynamic data structures [34].
- VCSR finds as 1.2−2× speedup over PCSR [4].
- PPCSR finds a 1.6× speedup over Aspen [83].
- CompressGraph finds a 2× speedup over Ligra+ [22]
- Teseo finds frequent speedups of at least 1.5× over other graph containers [32]

However, it is likely that much of the improvements seen in these works are from factors other than the graph container itself. 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 [62], OpenMP [26], Cilk [18], etc.).

As a result, 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.

4 BYO: A unified framework for benchmarking large-scale graph containers

To address these issues, this section describes Bring Your Own (BYO)[7], a unified programming framework for benchmarking and evaluating graph containers [81]. BYO is based on the Graph Based Benchmark Suite (GBBS) [36, 35], a high-performance graph-algorithm framework implemented on top of a CSR container.

Summary

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

[7]The full paper is available at `https://arxiv.org/abs/2405.11671`.

API that the containers need to implement, and implements the popular Ligra/GBBS interface[8] 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). Figure 11 shows the relationship between BYO, the programming framework, and graph container.

The paper uses 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 [33] and SSTGraph [78], as well as off-the-shelf data structure libraries such as those from the `std` standard library and Abseil [1], 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. The resulting evaluation involves running 10 fundamental graph algorithms on 10 large graph datasets with up to 4.2B edges.

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. The graph container API defined by BYO is very simple. Specifically, 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) [61, 50] 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 [29] 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. The full paper verifies these results and shows that BYO achieves $1.06-4.44\times$ speedup on average compared to other frameworks (e.g., Ligra [69] and GraphBLAS [51, 19, 27, 28]).

Standardized graph-container evaluation. Furthermore, BYO addresses previous evaluation issues due to different framework implementations by making sensible optimizations for

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

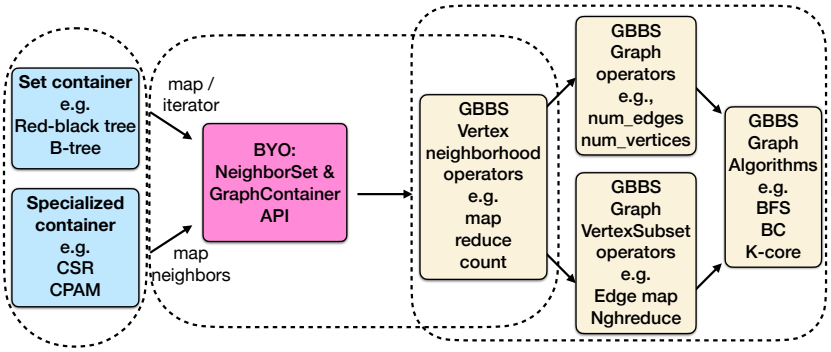

Figure 12: Relationship between BYO framework, graph containers, and graph algorithms (via GBBS).

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 [78] to compare with Aspen [34], 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 [78]. 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.

BYO API Description

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. I will summarize the "set" and "graph container" APIs that BYO exposes to connect with arbitrary graph data structures as well as framework-level optimizations that have appeared in various places throughout the literature that we have collected in BYO. The full paper contains further details about the API components (including discussion of how updates are included in BYO's API) and BYO's implementation.

Figure 12 illustrates the relationship between data structures, BYO, and GBBS components. The GBBS framework uses the VertexSubset abstraction from Ligra [69] for maintaining the active vertex set.

NeighborSet API description. 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 [38], the tree of trees in Aspen [34] and CPAM [33][9], directly implement this two-level structure.

BYO provides 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.

[9]The previous section referred to CPAM as C-PaC for clarity compared to CPMA. In this section, we will use CPAM to refer to the library that implements PaC-trees.

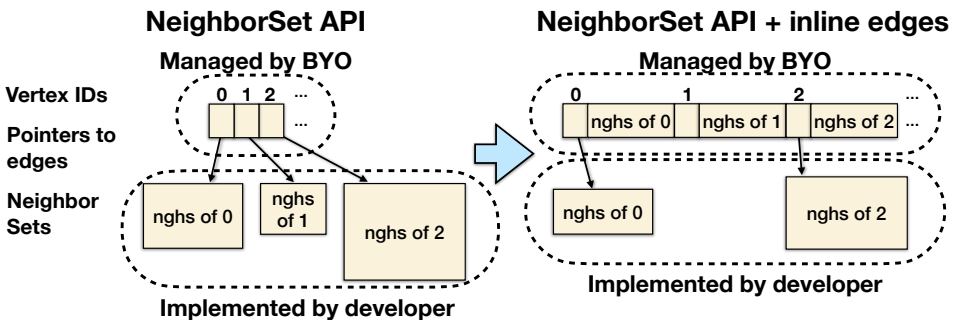

Figure 13: Data structure and inline optimization using the set API.

Figure 13 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) [61, 50], a widely-used standard library of basic utilities, includes both of these (among others).

More formally, the required functionality for algorithms for the NeighborSet API is as follows:

- `iterator` or `map(f)`: Apply the function `f` to all elements in the set. 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.

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 [63] 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 13 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.

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) })

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

GraphContainer API. Furthermore, BYO provide an alternative ***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) [72] representation stores all of the neighbor sets contiguously in one array. Furthermore, some optimized dynamic-graph containers like Terrace [63] and SSTGraph [78] 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`:

- `map_neighbors(i, f)`: Apply the function `f` to all neighbors of vertex `i`.
- `num_vertices()`: Return the number of vertices in 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 [63], which stores some neighbor sets contiguously in a dynamic array-like data structure. Additionally, SSTGraph [78] 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.

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, the BYO implementation also modifies the implementation of some 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.

Finally, to further standardize the evaluation between graph containers, BYO includes several framework-level optimizations that can benefit all systems, since they are independent of the container. The paper contains the full details of these optimizations and their performance benefits.

Results

The full paper includes a cross-cutting evaluation of graph containers and frameworks along several distinct axes. I will focus on the graph-container evaluation that BYO enables, but the paper includes benchmarks regarding the BYO framework itself and its comparison to other state-of-the-art frameworks such as Ligra and GraphBLAS. All experimtns were run on an Intel machine with 64 physical cores (128 hyperthreads) and 1024 GiB of main memory.

Using BYO[10], compare over 20 different containers in an apples-to-apples way on graph-algorithm performance without external factors from the algorithm implementation such as the specific algorithm for a problem or systems-level factors like the language and parallelization method. The evaluation includes not only specialized dynamic-graph containers, but also "off-the-shelf" data structures (e.g., those from the standard library) to determine how much performance can be gained just by using simple existing data structure implementations.

The algorithms included in the evaluation 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 [35].

The evaluation also measures the performance of 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.

Summary. In terms of graph algorithm performance (Figure 14), 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 (Figure 15), 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,

[10]The code is available at `https://github.com/wheatman/BYO`.

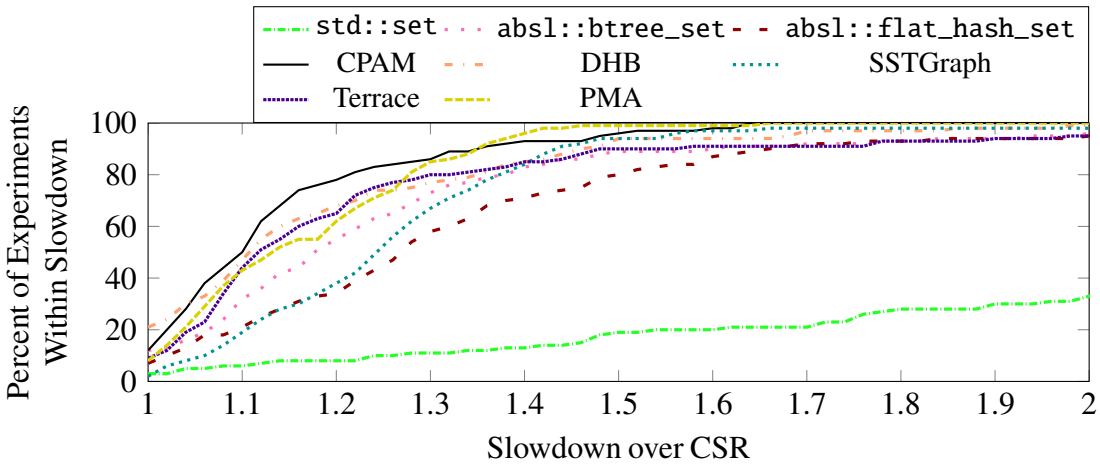

Figure 14: Slowdown of each container compared to CSR. Each point (x,y) for a given graph container means that the container was at most x times slower than CSR on y% of experiments. A line going up faster implies that the container achieves closer performance relative to CSR on more experiments. We find that almost all structures are able to perform the majority of the experiments with at most a 1.4× slowdown over CSR. `std::set` and absl::* are off-the-shelf containers, while the others are specialized for dynamic graphs.

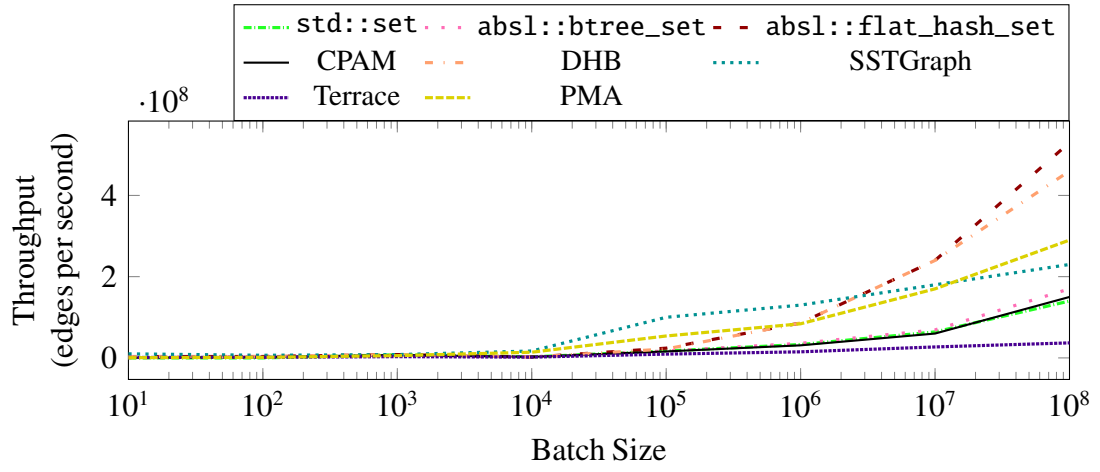

Figure 15: The throughput of inserts for different batch sizes. Of the data structures in this plot, `std::set` and absl::*, are off-the-shelf containers, while the others are specialized for dynamic graphs.

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.

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 2 reports the average, 95th percentile, and maximum slowdown over CSR for each data structure across all 100 problem settings (10 algorithms × 10 graphs).

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

Table 2: Data structure algorithm performance and space usage. All data structures are uncompressed unless otherwise specified. 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.

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 [1] B-tree 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 [33] + 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 [33], Terrace [63], DHB [75], CPMA [80], SSTGraph [78] and Aspen [34]) incur between $1.11-1.44\times$ slowdown on average relative to CSR.

These results do not invalidate previous evaluations because BYO enables direct comparisons of *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 performance differences were the result of variations in the framework as well as the container.

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 14 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 $1.25\times$ of CSR's performance on 63 experiments, while PaC-trees with the inline optimization achieved within $1.25\times$ of CSR's performance on 83 experiments.

We also measure the space usage of all containers which support getting their memory usage in Table 2. We find the bytes per edge varies significantly between graphs even when the container is fixed - by at least $2\times$ and sometimes up to $10\times$. 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 $2\times$ compared their uncompressed counterparts.

Guidance for choosing graph containers. Table 3 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. Please see the paper for details on the algorithms and graphs.

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 tested Erdos-Renyi (ER) graph [40] 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*. 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 (Approximate Densest Subgraph (ADS), Maximal Independent Set (MIS), and PageRank (PR)) when run on the RMAT graph (RM). 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 if a user has a specific algorithm or graph class that they are optimizing for, the trends noted above can help them select a different container.

	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

Table 3: The fastest container for every graph × algorithm combination. The graphs are sorted left to right by size (in number of edges). * next to a container denotes the NeighborSet API version with the inline optimization. CPAM/C-CPAM refers to the uncompressed/compressed version of CPAM, respectively. PMA(V) refers to the vector of PMAs using the NeighborSet API, and PMA refers to the single PMA using the GraphContainer API.

Discussion

This section summarizes 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.

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, the results demonstrate that while on average off-the-shelf data structures achieve highly competitive performance with specialized data structures.

However, the results indicate two promising directions for graph-container developers for optimizing specialized containers. First, the results demonstrate that off-the-shelf structures leave significant performance on the table for certain algorithms/graphs. Designing specialized containers for hard instances can mitigate worst-case performance. Second, specialized graph containers can overcome the folklore query-update tradeoff with efficient parallelization of batch updates.

5 Conclusion

Due to their ubiquity, dynamic graphs have attracted significant research attention towards creating fast algorithms, frameworks, and data structures for processing them. Dynamic-graph algorithms and data structures present an exciting opportunity for practical algorithm engineering that will be required to scale them to large graphs.

This column focuses on developing and benchmarking dynamic-graph containers and their associated systems on modern multicores, which are characterized by their large main memories, many parallel threads, and steep cache hierarchies. Specifically, I discussed two directions:

- Developing dynamic-graph containers that support both fast algorithms and fast updates without giving up performance in either direction. I used F-Graph, a container based on the Packed Memory Array data structure, as a case study for how cache-optimized data structures

can overcome traditional performance tradeoffs.
- Comprehensive and fair benchmarking of dynamic-graph containers without confounding factors from the overall dynamic-graph system, which includes both algorithm framework and the container. I show how BYO, a high-performance graph-algorithm framework designed for simplicity and ease of use, can standardize evaluations of graph containers and enable developers to easily build expressive and fast dynamic-graph systems.

Remarks and future directions

I believe that the two concrete results I mentioned in the column are steps in the right direction, but there is potential for the high-level ideas to play an even wider role in future development and benchmarking of dynamic-graph systems.

The results in this column demonstrate the Packed Memory Array's potential as a replacement for dynamic trees, but the PMA is not yet comparable to trees in terms of functionality and generality. For example, tree implementations often target more complex use cases such as functional updates [34] and transactional updates [25]. Furthermore, trees are common in other settings such as out-of-core and disk-based storage systems, but to my knowledge, there is not yet a high-performance disk-based PMA implementation.

Furthermore, BYO takes the first step towards apples-to-apples comparisons of graph containers, but is built on GBBS, which currently runs static graph algorithms, or algorithms that must recompute the entire answer from scratch in the presence of changes to the graph. There has been a great deal of work from the theory community on developing dynamic-graph algorithms, or algorithms that maintain some intermediate state to take updates into account and avoid full recomputation. I refer the interested reader to an excellent survey on the topic [44]. However, it is non-trivial to implement dynamic-graph algorithms that are actually faster than static algorithms in practice due to parallelism and locality issues. Independently of the theory community, the systems community has proposed many programming frameworks for dynamic algorithms on graphs [23, 60, 49, 59, 68, 58, 65, 76, 57, 56, 2, 21, 48, 41]. However, these framework implementations are also often tightly coupled with their underlying graph container, so they suffer from the same issues of generality.

My main motivation for writing this column is to share my excitement for developing dynamic-graph data structures and their associated systems. Although there have already been many papers on this topic, I believe that it will continue to be an area for future development as dynamic graphs continue to grow. I look forward to seeing progress in these exciting directions.

Acknowledgements

Special thanks to all my collaborators who made this work possible - Brian Wheatman, Prashant Pandey, Aydın Buluç, Randal Burns, Xiaojun Dong, Zheqi Shen, Laxman Dhulipala, and Jakub Łącki.

References

[1] Abseil. `https://abseil.io/`. Accessed: 2023-09-23.

[2] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 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*, ASPLOS 2023, page 133–145, New York, NY, USA, 2023. Association for Computing Machinery.

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

[4] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. VCSR: Mutable CSR graph format using vertex-centric Packed Memory Array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 71–80, 2022.

[5] Amazon. Amazon web services. `https://aws.amazon.com/`, 2022.

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

[7] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 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*, pages 44–54, 2006.

[8] David A. Bader and K. Madduri. High-performance combinatorial techniques for analyzing massive dynamic interaction networks. In *DIMACS Workshop on Computational Methods for Dynamic Interaction Networks, DIMACS Center, Rutgers University, Piscataway, NJ, September 24-25, 2007.*, 2007.

[9] Antonio Barbuzzi, Pietro Michiardi, Ernst Biersack, and Gennaro Boggia. Parallel bulk insertion for large-scale analytics applications. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, pages 27–31, 2010.

[10] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

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

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

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

[14] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

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

[16] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. An experimental analysis of a compact graph representation. In *ALENEX*, 2004.

[17] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. SPAA '16, pages 253–264, New York, NY, USA, 2016. Association for Computing Machinery.

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

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

[20] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

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

[22] Zheng Chen, Feng Zhang, JiaWei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. CompressGraph: Efficient parallel graph analytics with rule-based compression. *Proc. ACM Manag. Data*, 1(1), may 2023.

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

[24] Yongli Cheng, Yan Ma, Hong Jiang, Lingfang Zeng, Fang Wang, Xianghao Xu, and Yuhang Wu. TgStore: An efficient storage system for large time-evolving graphs. *IEEE Transactions on Big Data*, (01):1–16, 2024.

[25] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

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

[27] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), dec 2019.

[28] Timothy A. Davis. Algorithm 10xx: SuiteSparse:GraphBLAS: Parallel graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, jan 2023. Just Accepted.

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

[30] Dean De Leo and Peter Boncz. Fast concurrent reads and updates with PMAs. GRADES-NDA'19, pages 8:1–8:8, New York, NY, USA, 2019. ACM.

[31] Dean De Leo and Peter Boncz. Packed memory arrays-rewired. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 830–841. IEEE, 2019.

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

[33] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. PaC-Trees: Supporting parallel and compressed purely-functional collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 108–121, New York, NY, USA, 2022. Association for Computing Machinery.

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

[35] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Trans. Parallel Comput.*, 8(1), apr 2021.

[36] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 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)*, GRADES-NDA'20, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Marie Durand, Bruno Raffin, and François Faure. A packed memory array to keep moving particles sorted. In *9th Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)*, pages 69–77. The Eurographics Association, 2012.

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

[39] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. In *International Symposium on Experimental Algorithms*, pages 111–122. Springer, 2014.

[40] Paul Erdös and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[41] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 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*, SIGMOD '21, page 513–527, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Leonor Frias and Johannes Singler. Parallelization of bulk operations for stl dictionaries. In *European Conference on Parallel Processing*, pages 49–58. Springer, 2007.

[43] Per Fuchs, Domagoj Margan, and Jana Giceva. Sortledton: a universal, transactional graph data structure. *Proceedings of the VLDB Endowment*, 15(6):1173–1186, 2022.

[44] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms–a quick reference guide. *ACM Journal of Experimental Algorithmics*, 27:1–45, 2022.

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

[46] Abdullah Al Raqibul Islam and Dong Dai. DGAP: Efficient dynamic graph analysis on persistent memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

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

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

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

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

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

[52] Pradeep Kumar and H Howie Huang. GraphOne: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)*, 15(4):1–40, 2020.

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

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

[55] Kamesh Madduri and David A. Bader. Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11. IEEE, 2009.

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

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

[58] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: Practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, jun 2020.

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

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

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

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

[63] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydın Buluç. Terrace: A hierarchical graph container for skewed dynamic graphs. In *SIGMOD*, page 1372–1385, 2021.

[64] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

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

[66] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on GPUs. *Proc. VLDB Endow.*, 11(1):107–120, September 2017.

[67] Jifan Shi, Biao Wang, and Yun Xu. Spruce: a fast yet space-saving structure for dynamic graph storage. *Proceedings of the ACM on Management of Data*, 2(1):1–26, 2024.

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

[69] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

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

[71] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. PAM: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 290–304, 2018.

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

[73] Julio Toss, Cicero AL Pahins, Bruno Raffin, and João LD Comba. Packed-memory quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. *Computers & Graphics*, 76:117–128, 2018.

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

[75] Alexander van der Grinten, Maria Predari, and Florian Willich. 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, 2022.

[76] Keval Vora, Rajiv Gupta, and Guoqing Xu. 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*, pages 237–251, 2017.

[77] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. XPGraph: XPline-friendly persistent memory graph stores for large-scale evolving graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1308–1325. IEEE, 2022.

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

[79] Brian Wheatman, Randal Burns, Aydın Buluç, and Helen Xu. Optimizing search layouts in Packed Memory Arrays. In *ALENEX*, 2023.

[80] Brian Wheatman, Randal Burns, Aydın Buluç, and Helen Xu. 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*, pages 348–363, 2024.

[81] 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. *Proceedings of the VLDB Endowment (to appear)*, 2024.

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

[83] Brian Wheatman and Helen Xu. A parallel Packed Memory Array to store dynamic graphs. In *ALENEX*, pages 31–45, 2021.

[84] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. faimGraph: High performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 754–766. IEEE, 2018.

[85] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. BP-Tree: Overcoming the point-range operation tradeoff for in-memory B-trees. *Proc. VLDB Endow.*, 16(11):2976–2989, jul 2023.

[86] Helen Jiang Xu. *The Locality-First Strategy for Developing Efficient Multicore Algorithm*. PhD thesis, Massachusetts Institute of Technology, 2022.

[87] Lei Zou, Fan Zhang, Yinnian Lin, and Yanpeng Yu. An efficient data structure for dynamic graph on GPUs. *IEEE Transactions on Knowledge and Data Engineering*, 2023.