Batch-Parallel Compressed Sparse Row: A Locality-Optimized Dynamic-Graph Representation

Brian Wheatman
University of Chicago
Chicago, IL
wheatman@uchicago.edu

Randal Burns
Department of Computer Science
Johns Hopkins University
Baltimore, MD
randal@cs.jhu.edu

Helen Xu
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, GA
hxu615@gatech.edu

***Abstract*—The default data structure for storing sparse graphs is Compressed Sparse Row (CSR), which enables efficient algorithms but is not designed to accommodate changes to the graph. Since many real-world graphs are dynamic (i.e., they change over time), there has been significant work towards developing dynamic-graph data structures that can support fast algorithms as well as updates to the graph.**

This paper introduces Batch-Parallel Compressed Sparse Row (BP-CSR), a batch-parallel data structure optimized for storing and processing dynamic graphs based on the Packed Memory Array (PMA). At a high level, Batch-Parallel Compressed Sparse Row extends Packed Compressed Sparse Row (PCSR, HPEC '18), a serial dynamic-graph data structure built on a PMA. However, since the original PCSR runs only on one thread, it cannot take advantage of the parallelism available in multithreaded machines. In contrast, Batch-Parallel Compressed Sparse Row is built on the batch-parallel Packed Memory Array data structure (PPoPP '24) and can support fast parallel algorithms and updates.

The empirical evaluation demonstrates that Batch-Parallel Compressed Sparse Row supports fast parallel updates with minimal cost to algorithm performance. Specifically, Batch-Parallel Compressed Sparse Row performs up to 420 million inserts per second. Across a suite of 10 graph algorithms and 10 input graphs, Batch-Parallel Compressed Sparse Row incurs $1.05\times$ slowdown on average and about $1.5\times$ slowdown at most compared to Compressed Sparse Row (CSR), a classical static graph representation. Furthermore, the empirical results show that Batch-Parallel Compressed Sparse Row outperforms existing tree-based and PMA-based dynamic-graph data structures on both algorithms and updates.

***Index Terms*—Packed Memory Array, batch updates, dynamic graphs, graph data structures, storage formats.**

I. Introduction

There has been significant research effort devoted to developing dynamic-graph data structures (or containers) and their associated systems on a single large shared-memory machine [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. Systems built for storing and processing dynamic graphs must accommodate changes to the graph. Therefore, they have been designed to quickly process both updates (e.g., edge inserts/deletes) and queries (e.g., graph algorithms).

Packed Compressed Sparse Row (PCSR) [6] is an example of a dynamic-graph container built on the Packed Memory Array (PMA) [16] data structure. The PMA stores elements in one contiguous memory allocation, enabling fast traversals through the edges, which is a core primitive in graph algorithms [17]. Furthermore, it supports insertions much faster than the theoretical bounds suggest due to its cache-friendliness [12].

The original paper that introduced PCSR [6] used a sequential PMA to implement all the operations, leaving performance on the table by not taking advantage of thread-level parallelism. Modern dynamic-graph systems on multicores use multithreading to achieve good performance for both algorithms and updates.

Recent work makes progress towards addressing this issue with a batch-parallel PMA [12] and a PMA that supports concurrent updates [8]. Multithreaded PMAs support updates much faster than the serial PMA. On graph workloads, the batch-parallel PMA can support up to hundreds of millions of edge updates per second with multiple threads without sacrificing algorithm performance compared to other state-of-the-art dynamic-graph systems [12].

However, directly storing a graph in a single batch-parallel PMA without optimizing for graph-structured data adds overhead to algorithms because a standard PMA for graphs does not support accessing the start of a vertex's neighbor list in $O(1)$ time. To understand this issue, let us first consider how to process graphs in Compressed Sparse Row (CSR) format [18], a classical method for storing sparse graphs. CSR stores m neighbor IDs in an *edge array* of size m. The neighbors of each vertex v are stored contiguously in the edge array. Additionally, the CSR format stores the start of each vertex's neighbors in a *vertex array* of size n, where n is the number of vertices. Iterating through the neighbor list for a vertex v in CSR requires an $O(1)$ lookup in the vertex array for the start offset, and then $O(\text{degree}(v))$ steps to traverse the edges in the edge array. In contrast, finding the start of the neighbor list in a single PMA takes $O(\log(m))$ time, where m is the number of edges in the graph, to perform a binary search for

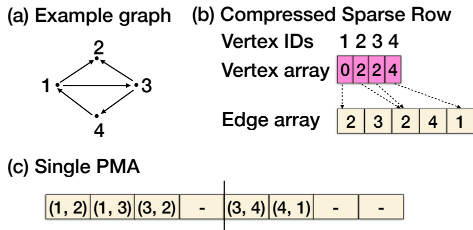

Fig. 1: A small example graph and how to store it in Compressed Sparse Row (CSR) and a single PMA. CSR, a classical graph storage format, represents the graph with two arrays: a vertex array and edge array. In contrast, the single PMA only needs one array, as it stores all the data in the edge array as edge pairs.

the correct position in the PMA. When the average degree of the graph is small, as is common in real-world graphs, the cost to find the start of the neighbor list in the PMA may be higher than the cost to traverse the edges. Figure 1 illustrates this distinction.

To address this issue, previous work proposed speeding up algorithms by building an ad-hoc copy of the vertex array on demand to avoid unnecessary PMA searches during the algorithms [12]. However, this approach requires entirely rebuilding the entire vertex array upon any update, which adds unnecessary overhead in the presence of dynamic updates. As we will detail in Section IV, pre-computing the vertex array takes between 1% and 50% of the algorithm time (averaged across all the graphs). In this work, we show how to avoid rebuilding the entire vertex array in the presence of updates.

This paper introduces *Batch-Parallel Compressed Sparse Row*, a dynamic-graph data structure built on the batch-parallel PMA that avoids the overhead of rebuilding entire the vertex array upon updates. Batch-Parallel Compressed Sparse Row combines the strengths of CSR and the batch-parallel PMA by maintaining a vertex array in addition to an edge PMA. The batch-parallel PMA enables Batch-Parallel Compressed Sparse Row to support fast multithreaded updates without giving up on algorithm performance.

Contributions

The main contributions of this paper are as follows:

- The design of Batch-Parallel Compressed Sparse Row (BP-CSR), a dynamic-graph system that extends PCSR using the batch-parallel PMA.
- An implementation of BP-CSR in C++.
- An evaluation of state-of-the-art dynamic-graph systems in terms of their algorithm performance, insert throughput, and space usage.

Results summary

We evaluate Batch-Parallel Compressed Sparse Row compared to other PMA-based graph representations: Packed Compressed Sparse Row (PCSR) [6], Parallel Packed Compressed Sparse Row (PPCSR) [8], F-Graph [12], as well as CPAM [5], a tree-based dynamic-graph representation. We compare all systems in terms of algorithm performance, update throughput, and space usage.

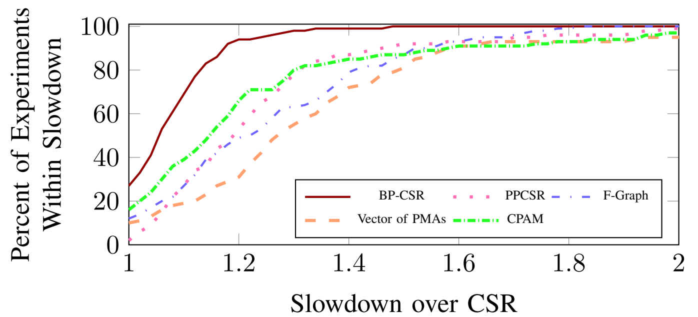

Fig. 2: Performance profile of each container compared to CSR. Each point (x, y) for a given graph container means that the container was at most x times slower than CSR on $y\%$ of problem settings (i.e., one configuration of algorithm and input graph).

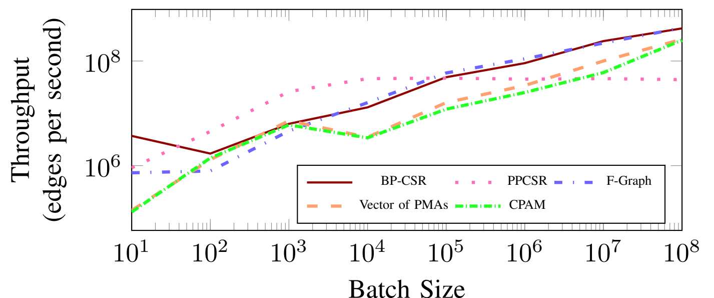

Fig. 3: Insert throughput as a function of batch size.

Figure 2 shows the slowdown that different dynamic-graph data structures incur relative to CSR in various experiment settings. Comparing dynamic data structures to CSR illustrates the cost to algorithm performance that they incur to support updates. Specifically, we evaluate data structures on a suite of 10 algorithms and 10 graphs (for a total of 100 problems). The results demonstrate that BP-CSR incurs the smallest average slowdown of $1.05\times$ compared to CSR. It achieves smaller slowdowns on more experiment settings (graphs and algorithms) compared to the other dynamic data structures.

Figure 3 evaluates dynamic-graph data structures in terms of batch-insert throughput and shows that BP-CSR does not give up update performance for algorithm performance: it achieves similar insertion performance compared to F-Graph [12] and about $2.4\times$ speedup over CPAM [5].

II. Preliminaries

This section describes the Packed Memory Array (PMA) [19], [16] data structure used to store edges in Batch-Parallel Compressed Sparse Row. Specifically, it will review the PMA's structure and operations. The discussion focuses on the batch-parallel insert algorithm for PMAs as the basis for efficient batch inserts in Batch-Parallel Compressed Sparse Row. Finally, this section will provide background about primitives in graph processing.

We will limit the discussions in this paper to inserts for simplicity, but deletes are implemented symmetrically.

A. Packed Memory Array

The PMA maintains elements in sorted order in a contiguous array with a constant factor of empty spaces between its elements. The empty cells facilitate sublinear dynamic updates by reducing data movement during inserts. Furthermore, since the PMA stores all elements in one single memory allocation, it supports fast cache-efficient iteration through the data.

A PMA exposes three operations:

- `batch_insert(S)`: inserts elements in a batch `S` into the PMA. The batch `S` is a set of elements of the same type of the elements in the PMA.
- `search(x)`: returns a pointer to the smallest element that is at least `x` in the PMA.
- `range_map(start, end, f)`: applies the function `f` to all elements in the range [`start`, `end`). One can think of `range_map` as an iterator that processes all elements in a given range.

To maintain the appropriate number of empty spaces to support efficient updates, the PMA array with N elements defines an implicit binary tree with leaves of size $\Theta(\log(N))$ cells. That is, the implicit tree has $\Theta(N/\log(N))$ leaves and height $\Theta(\log(N/\log(N)))$. Each node corresponds to a *region* of cells that encompasses all of its descendants. Each level of the tree has *density bounds* on the number of empty spaces that must be present in each of the nodes at that level.

Point inserts: PMA inserts use the implicit tree to maintain the overall structure. An 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 PMA *redistributes* elements equally among leaves in the node it counted up to, resolving the density bounds in all of its descendants.

Batch inserts: PMAs can also algorithmically support *parallel batch inserts*, which applies an entire batch of elements to a PMA [12]. Formally, the `batch_insert` function takes as input a PMA with N elements and a batch of k elements, and inserts all elements in the batch into the PMA. Batch inserts are more efficient than point inserts because they can share work and more efficiently achieve parallelization. The batch-insert algorithm for PMAs has three phases (similar to the steps of a point insert):

Merge: Examine the midpoint of the batch, search for the corresponding target leaf in the PMA, and merge all relevant elements from the batch into that leaf. If some elements overflow, the algorithm modifies the PMA to store those elements temporarily out-of-place. For the next level of the recursion, split the PMA and batch into halves, excluding the elements merged in at this level.

Count: Just like in the point-insert case, count the cells in PMA nodes, traversing from leaves that violate their density bounds, until we find nodes that respect their density bounds.

Redistribute: Evenly distribute elements in the nodes found from the count phase so that their descendants have the correct spaces to respect their density bounds.

In practice, batch updates in PMAs are often faster than the worst-case theoretical bounds suggest because they can also reduce redundant work in searching and counting if multiple elements are destined for the same leaf in the PMA.

Growing factor: When the PMA becomes too dense, the underlying array must grow by a constant factor, called the *growing factor*, to accommodate the new elements and respect the density bounds. Any constant realizes the theoretical bounds, but the growing factor is often implemented as be $2\times$ for simplicity in the literature [6], [8].

B. Graph Processing

A graph is an abstraction that represents entities as *vertices* and connections between those entities as *edges*. In the unweighted case, edges can be represented as pairs (u, v) for given vertices u, v. In the weighted case, the edges may have an additional value w and be represented as tuples (u, v, w).

Sparse graph representations (e.g., CSR) save space and computation over implicit representations such as adjacency matrix by explicitly storing the edge tuples. They take space proportional to $O(n + m)$, where n is the number of vertices, and m is the number of edges. Given a vertex v, they support iteration over its neighbors in $O(\texttt{degree}(v))$.

Prior work showed that the `range_map` primitive is sufficient for graph data structures to express a wide range of graph algorithms [17]. For example, Ligra [20], a popular graph-algorithm framework, can run many algorithms including breadth-first search, connected components, PageRank, and others, with abstractions based on the `range_map` primitive. Therefore, how fast a graph container can support algorithms is directly related to how fast it can support `range_map`.

III. Batch-PCSR Structure and Algorithms

This section describes 1) the high-level structure of the Batch-Parallel Compressed Sparse Row (BP-CSR) graph data structure and 2) how it supports batch updates and the map primitive. It details how BP-CSR avoids rebuilding the vertex array with a modification of the batch-parallel insert algorithm for PMAs [12].

A. Structure

At a high level, the structure of Batch-Parallel Compressed Sparse Row is very similar to PCSR [6] (and by extension, CSR). There are two arrays - the *vertex array* and the *edge PMA*. The vertex array stores two entries per vertex: the *offset* pointing to the start of the vertex's edges in the edge PMA, and the degree of each vertex. Just as in CSR, the edge PMA stores the destinations of the edges explicitly. Furthermore, just like in PCSR, the edge PMA also stores one *sentinel* per vertex. A sentinel is a special value at the beginning of each vertex's

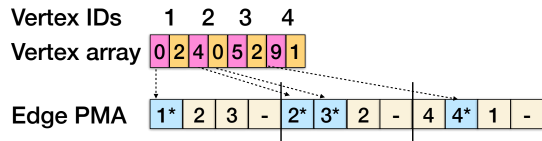

Fig. 4: An example of how BPCSR can store the graph from Figure 1(a). The high bit denotes whether or not an element in the PMA is a sentinel (denoted by *). 0 is a special value for empty spaces, so we 1-index the graph vertices. In the vertex array, the start offsets are pink, and the degrees are yellow. In the PMA, the sentinels are colored blue, while the edges and PMA empty spaces are tan.

neighbor list that denotes the corresponding source vertex. The implementation sets the top bit to denote that an entry in the edge PMA is a sentinel and stores the source vertex ID in the lower bits. As we will explain later in this section, the sentinels are used to keep the relevant entries in the vertex array up-to-date during insertions to the edge PMA. Figure 4 illustrates an example of how Batch-Parallel Compressed Sparse Row stores a graph in a vertex array and a PMA.

Weighted graphs: For simplicity, we focus on describing how to store the graph topology (edge information), but BP-CSR supports edge weights with an additional *weights PMA* that mirrors the structure and layout of the edge PMA. The sentinels in BP-CSR differ from those in other PMA-based graph data structures such as PCSR and PPCSR because previous systems store the sentinels in the weights PMA. In contrast, BP-CSR uses the top bit to denote sentinels in the edge PMA. This design change enables BP-CSR to save space in the unweighted case, because previous data structures must always store the weight PMA (even in the unweighted case) due to the sentinel structure.

B. Algorithms

Map in BP-CSR: To perform the `range_map` primitive, we first use the vertex array to determine the offset of the start of the given vertex's neighbors in the edge PMA. We then iterate forward until the next vertex's offset, skipping over the sentinel and any empty spaces. BP-CSR supports a map over a vertex v's neighbors in $O(1+\texttt{degree}(v))$ time. It takes $O(1)$ time to find the start of a vertex's neighbors in the edge PMA using the vertex array, and $O(\texttt{degree}(v))$ time to iterate over the neighbors.

Batch updates in BP-CSR: Next, we will describe how BP-CSR adapts the batch-insert algorithm for a single PMA [12] to a two-level structure. The batch-insert algorithm for BP-CSR takes as input a set of edges sorted by source and destination vertex ID and adds the set of edges to the graph. All edges not currently in the graph will be added after the algorithm is completed. We will describe the algorithm for the unweighted case for simplicity, but the algorithm can also accommodate weights with the same layout duplicated in the weights PMA.

At a high level, the batch-insert algorithm for BP-CSR follows almost the same structure described in Section II. The three phases of the batch-insert algorithm in BP-CSR are the same as in the single PMA: 1) batch merge, 2) count, and 3) redistribute. The main changes are that 1) the vertex array enables the algorithm to skip some searches with lookups for the start offset, and 2) the vertex array may need to be updated after the batch has been added to the edge PMA. Figure 5 illustrates an example of a batch insert in BP-CSR.

In the standard PMA case, the batch-merge phase first performs a search for the midpoint of the batch in the PMA and merges elements into the relevant PMA leaf. Similarly, the first step in batch inserts for BP-CSR is to choose the midpoint of the batch and find the correct leaf corresponding to the selected edge. Since every edge in the batch already has a source vertex ID, we can bypass the search into the PMA for the target leaf and instead use the vertex array to find the offset (and therefore leaf) of the midpoint edge. We use the vertex array to determine the graph vertex that corresponds to the first sentinel in the target leaf. Finally, we find the first edge in the batch corresponding to the target leaf (which may be different from the originally chosen midpoint) and perform a merge just like in the original batch algorithm. During the merge, we can update the vertex array with any changed degrees. We then recurse on the remaining left and right "halves" of the batch and edge PMA.

The counting phase is exactly the same as in the original batch-insert algorithm, since it only depends on the positions of elements in the edge PMA. The output of the counting phase is the ranges of the edge PMA to redistribute.

Finally, the redistribute phase is almost the same as in the original algorithm, but with a slight change of updating the vertex array with any changed sentinel locations. To determine which sentinel locations have changed, we can check every element being written during the redistribute (i.e., all elements that are changing locations) for whether or not it is a sentinel by just checking whether the top bit is set. If a sentinel is moving locations, the algorithm simply updates the vertex array with its new location.

These changes do not affect the worst-case asymptotic bounds for the batch-insert algorithm for PMAs. In the best case, the bounds may improve because some of the searches become array lookups.

IV. Evaluation

This section empirically evaluates Batch-Parallel Compressed Sparse Row (BP-CSR) compared to state-of-the-art dynamic-graph data structures in terms of algorithm performance, update throughput, and space usage. We find that BP-CSR achieves the best of all worlds: it supports algorithms and updates faster than other optimized dynamic-graph systems. Furthermore, BP-CSR uses less space to store the graphs compared to other data structures.

A. Experimental setup

All experiments were run across all cores of a machine with 2 48-core 2-way hyper-threaded Intel® Xeon® Platinum 8488C CPU @ 3.20GHz, for a total of 192 threads, with 384 GB of

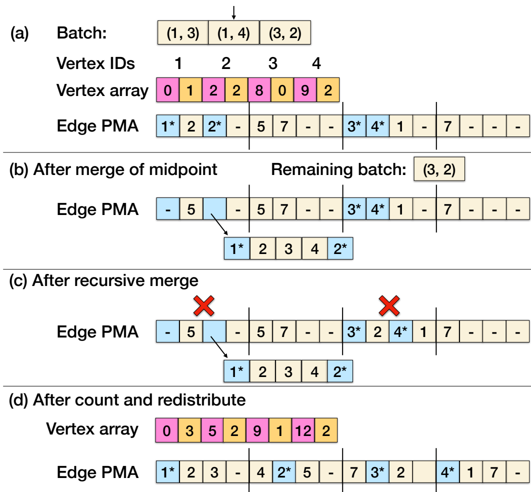

Fig. 5: An example of a batch insert in BP-CSR. In this example, the density bound in the leaves is 0.9. Just as in the regular batch-parallel PMA, the batch insert first has a merge phase where it recursively merges the batch into the PMA, then a count and redistribute phase to resolve the density bounds. In the vertex array, the start offsets are pink, and the degrees are yellow. In the PMA, the sentinels are colored in blue, while the edges and PMA empty spaces are tan. After the second merge, the batch has been fully merged in. The red X over some leaves denotes that leaves violate their density bound and therefore must be redistributed with neighboring leaves.

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)

TABLE I: Graph algorithms tested in this paper via BYO [17].

memory from AWS [21]. Across all the cores, the machine has 4.5 MiB of L1 data cache, 192 MiB of L2 cache, and 210 MiB of L3 cache. The code is all implemented in `c++` and is compiled with `g++-13`. Each experiment (algorithm, graph, data structure) is run 3 times and the times are averaged.

B. Systems

The evaluation includes the proposed system, Batch-Parallel Compressed Sparse Row (BP-CSR), as well as several PMA-based graph representations: Packed Compressed Sparse Row (PCSR) [6], Parallel Packed Compressed Sparse Row (PPCSR) [8], F-Graph [12], and CPAM [5], a tree-based dynamic-graph representation. All data structures were run in uncompressed mode for a fair comparison.

Graph	*Vertices*	*Edges*	*Avg. Degree*
Road (RD) [24]	23,947,347	57,708,624	2
LiveJournal (LJ) [25]	4,847,571	85,702,474	18
Com-Orkut (CO) [26]	3,072,627	234,370,166	76
rMAT (RM) [27]	8,388,608	563,816,288	67
Erdős-Rényi (ER) [28]	10,000,000	1,000,009,380	100
Protein (PT) [29]	8,745,543	1,309,240,502	150
Twitter (TW) [22]	61,578,415	2,405,026,092	39
papers100M (PA) [30]	111,059,956	3,228,124,712	29
Friendster (FS) [23]	124,836,180	3,612,134,270	29
Kron (KR) [31]	134,217,728	4,223,264,644	31

TABLE II: Sizes of (symmetrized) graphs used.

For simplicity, all data structures were run in unweighted mode if they support it (BP-CSR, F-Graph, and CPAM). Some of the systems (PCSR, PPCSR) only support weighted storage of graphs, so we ran those in weighted mode. This will impact the size but should have a negligible impact on algorithm performance and a small impact on insert performance.

To run graph algorithms, we integrated all the data structures with BYO [17], a general graph-algorithm framework based on the Ligra abstraction [20]. BYO enables apples-to-apples comparisons of data structures by standardizing the algorithm implementation and infrastructure (e.g., programming language, parallelization library, compiler, etc.). All systems can support algorithms in parallel via BYO, and all systems besides PCSR can support multithreaded updates. Table I lists the algorithms tested in the evaluation.

All algorithms were run in unweighted mode, that is, they only read over the edge information in the graph.

C. Graphs

Table II details the graphs used in the evaluation. The graphs come from different domains including social networks, road networks, and computational biology. They range in size from tens of millions up to billions of edges. The inputs come from the popular GAP benchmark suite [22] and SNAP dataset [23], as well as others. We refer the interested reader to the BYO paper on graph benchmarking for more details about the datasets [17].

D. Algorithm Performance

Table III reports the average, 95th percentile, and maximum slowdown over CSR for each data structure across all 100 experiment settings (10 algorithms $\times$ 10 graphs). The results show that BP-CSR incurs only $1.05\times$ slowdown on average and at most $1.5\times$ slowdown compared to CSR, a cache-optimized static-graph representation.

BP-CSR supports algorithms faster than all the tested data structures in almost all cases. Specifically, we found that it was the fastest data structure on 72 out of 100 experiment settings. Figure 2 shows that BP-CSR achieves within $1.2\times$ slowdown over CSR on 94 out of 100 experiment settings.

BP-CSR resolves two issues in F-Graph, a state-of-the-art PMA-based dynamic-graph system: 1) the cost of scans, and 2) inefficient space usage (in the uncompressed case). F-Graph is a dynamic-graph data structure that directly stores the edge list

in a single PMA in sorted order. Since there is no vertex array, F-Graph supports finding the start of a vertex's neighbor list in $O(\log(m))$ time, where m is the number of edges. In contrast, BP-CSR reduces the time to find the start of a neighbor list to $O(1)$ using the vertex array. As an optimization to avoid this cost, F-Graph builds a vertex array in an ad-hoc way upon the start of algorithms. We do not count this time in the reported times for F-Graph, but, averaged across all the graphs, we find that it takes up to 50% of the time for cheaper algorithms (BFS and LDD) and down to 1% of the time for more expensive algorithms (kcore, BC, PR, and coloring). Additionally, F-Graph stores the graph as a list of (source, destination) pairs. Without compression, the sources are repeated in every edge incident to a given vertex. In contrast, BP-CSR uses the vertex array to avoid storing the source vertices. As a result, BP-CSR supports faster algorithms on average compared to F-Graph (improvement from $1.23\times$ to $1.05\times$ average slowdown over CSR) by avoiding redundant space usage.

BP-CSR outperforms PCSR and PPCSR, two other PMA-based dynamic-graph systems, by reducing the growing factor and space usage. Both PCSR and PPCSR use a growing factor of $2\times$, while BP-CSR uses a growing factor of $1.2\times$ by default. We found that PCSR ran into implementation limits on all but the smallest graphs, we report the results for those graphs in Table IV. For comparison, we include BP-CSR2, a variant of BP-CSR with a growing factor of $2\times$. As expected, BP-CSR2, PCSR, and PPCSR experience similar algorithm times because they all have similar memory layouts. When limiting to the small graphs, PCSR incurs $1.36\times$ slowdown on average over CSR, while BP-CSR incurs $1.05\times$ slowdown.

Finally, BP-CSR outperforms CPAM, a dynamic-graph data structure based on cache-optimized trees, because the PMA stores edges contiguously in memory, while CPAM stores one tree per vertex separately in memory. CPAM incurs $1.18\times$ slowdown over CSR on average and about $2\times$ at most.

To demonstrate the benefit of storing all edges in a single PMA, we compare BP-CSR with a similar toy system that stores a graph as a vector of PMAs (one per vertex). We call this system "PMAs (V)" in Table III. On average, the vector of PMAs incurs $1.29\times$ slowdown compared to CSR and at most about $2\times$ slowdown because it incurs additional cache misses between vertex neighbor lists.

Container	*Slowdown over CSR*			*Bytes per edge*			*Insert TP*
	Avg	*95%*	*Max*	*Min*	*Avg*	*Max*	*Max*
CSR	1.00	1.00	1.00	4.05	4.44	7.32	—
BP-CSR	1.05	1.23	1.46	5.21	6.25	12.54	4.2E8
BP-CSR2	1.21	1.57	1.77	8.71	12.99	23.59	4.2E8
PPCSR	1.22	1.69	2.22	8.99	14.22	35.21	0.5E8
F-Graph	1.23	1.63	1.81	10.00	11.59	24.16	4.3E8
CPAM	1.18	1.94	2.51	4.12	5.53	21.58	2.5E8
PMAs (V)	1.29	1.94	2.38	7.58	12.12	64.74	2.6E8

TABLE III: Algorithm performance, space usage, and maximum insert throughput (TP, in inserts/s) of the different containers. PMAs (V) is a vector with a PMA per vertex. Each container's algorithm time is normalized to CSR's time averaged over all 100 settings of 10 algorithms $\times$ 10 graphs (closer to 1 is good).

Container	*Slowdown over CSR*			*Bytes per edge*		
	Average	*95%*	*Max*	*Min*	*Average*	*Max*
CSR	1.00	1.00	1.00	4.10	5.29	7.32
PCSR	1.36	2.04	2.47	13.70	19.28	25.58
BP-CSR	1.05	1.20	1.46	5.71	8.01	12.54
BP-CSR2	1.17	1.55	1.59	9.32	15.37	23.59
PPCSR	1.19	1.52	1.92	9.69	19.89	35.21

TABLE IV: Algorithm performance (normalized to CSR) and space usage on the graphs (RD, LJ, CO) that the original PCSR can run without errors on.

E. Batch-Insert Performance

To evaluate insertion throughput, we first insert all edges from the Twitter graph. We then add a new batch of directed edges (with potential duplicates) to the existing graph in both systems. To generate edges for inserts, we sample directed edges from an rMAT generator [27] (with $a = 0.5$; $b = c = 0.1$; $d = 0.3$ to match the distribution from prior work [5]). We do not include PCSR because it is single threaded. All other systems support multithreaded insertions.

Figure 3 and Table III show that BP-CSR does not sacrifice insertion throughput for improved algorithm performance: it matches F-Graph and achieves about $2.4\times$ speedup over CPAM on average across batch sizes. The batch insert algorithm in BP-CSR is very similar to the one in F-Graph, and we find their performance to be similar as well. BP-CSR outperforms CPAM in batch inserts because of the PMA's cache-friendliness.

Finally, we evaluate PPCSR, a concurrent dynamic-graph data structure, and find that BP-CSR ranges from $4\times$ slower on small batches (1k) to $10\times$ faster on the largest batch size (10M). This disparity comes from the two systems' fundamentally different parallelization schemes - PPCSR uses locks for concurrency control, while BP-CSR applies the entire batch as one operation in a lock-free manner. The lock-based parallel PMA in PPCSR does not scale as well with the batch size (i.e., the number of insertions) because of increased contention. On the other hand, the batch-insert algorithm is well-suited for large batches due to increased opportunities for shared work between updates.

F. Scalability

a) Experimental Setup: To measure the strong scaling with increasing thread counts, we tested breadth-first search (BFS), PageRank (PR), and batch inserts on the Twitter graph. Since the machine has two sockets each with 96 threads (for a total of 192 threads), we measured the times with all power-of-two numbers of threads from $1, 2, \ldots, 64$, then all threads on one socket (96 threads), then the entire machine (192 threads). We omit the original PCSR because it does not support multithreaded insert. For batch inserts, we set the batch size to 1 million and generated the inserts in the same way described earlier. All times are the average of three trials.

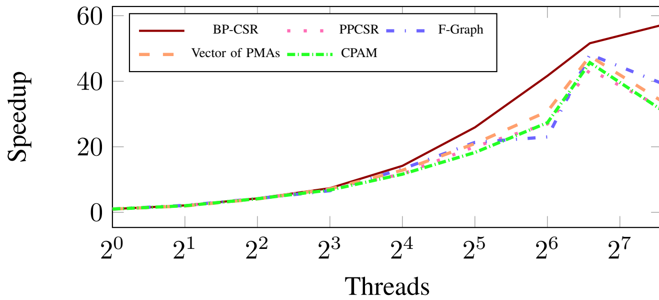

Fig. 6: Strong scaling of BFS on the TW graph.

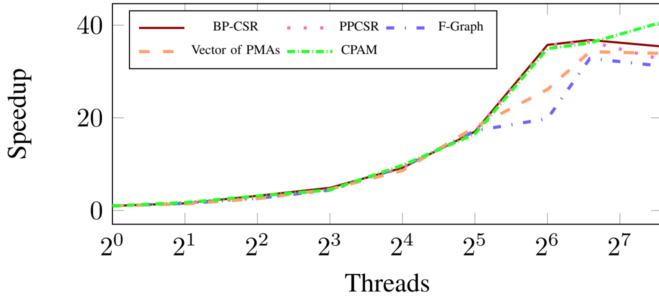

Fig. 7: Strong scaling of PR on the TW graph.

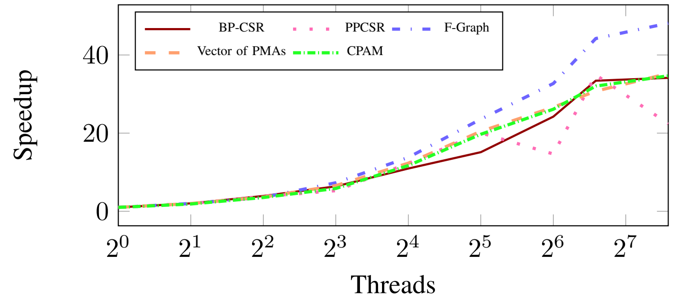

Fig. 8: Strong scaling of batch inserts with `batch_size` = 1M on the TW graph.

Container	*BFS*			*PageRank*			*Batch Insert*		
	T_1	T_{192}	$\frac{T_1}{T_{192}}$	T_1	T_{192}	$\frac{T_1}{T_{192}}$	T_1	T_{192}	$\frac{T_1}{T_{192}}$
BP-CSR	5.56	0.10	57.2	315	8.9	35.4	0.22	0.006	34.2
PPCSR	4.97	0.16	31.8	353	10.8	32.6	0.47	0.021	22.4
F-Graph	5.50	0.14	39.3	316	10.2	31.2	0.25	0.005	48.1
CPAM	3.80	0.12	31.1	340	8.4	40.7	0.57	0.017	34.7
PMAs (V)	8.37	0.16	51.4	343	12.0	28.6	0.56	0.016	35.9

TABLE V: Sequential time (T_1), parallel time on all 192 threads (T_{192}), and parallel speedup ($\frac{T_1}{T_{192}}$) of BFS, PR, and batch inserts (`batch_size` = 1M) on the TW graph.

b) Discussion: Figures 6, 7, and 8 illustrate the strong scaling of BFS, PR, and batch inserts (resp.) of the tested systems. We chose BFS and PR to illustrate two extremes of graph algorithms - BFS is a lightweight algorithm that traverses each edge at most once, while PR is more work-intensive and traverses all of the edges multiple times throughout the algorithm. Across the workloads, BP-CSR achieves competitive parallel speedup when compared to the other systems. In many cases across workloads, systems may incur slowdown when going from one to two NUMA nodes despite additional threads because of additional data movement.

Table V provides the data for sequential time, parallel time on all threads, and speedup on all threads. The workloads are memory-bound, so the maximum parallel speedup that any system achieves on 192 threads is about 57. On algorithms, BP-CSR improves the parallel speedup of PMA-based systems by reducing data movement with a two-level structure. However, on inserts, it achieves less parallel speedup compared to F-Graph because it is more sequentialized on inserts due to updating both the vertex and the edge level. However, their absolute parallel times are similar.

G. Space Usage

Tables III and IV report the space usage of the different graph data structures in terms of bytes per edge. We report the size with weights for systems that only have weighted mode (PCSR, PPCSR), and the unweighted size otherwise (BP-CSR, F-Graph, CPAM, vector of PMAs).

On average, BP-CSR uses about 6.3 bytes per edge, CSR uses 4.4 bytes per edge, and CPAM uses 5.5 bytes per edge. BP-CSR uses the smallest space of all the PMA-based structures because 1) it has the smallest growing factor and 2) it does not need to store weights or duplicated sources. Since the PMA memory layout is a simple array, most graph algorithms are translated into scans over this array, so reducing the memory usage maximizes the useful memory bandwidth.

V. Conclusion

This paper introduces BP-CSR, a dynamic-graph data structure built on the PMA. BP-CSR achieves the best of all worlds in terms of algorithm performance, update throughput, and space usage compared to other dynamic-graph data structures. It demonstrates the potential for optimizing cache-friendly data structures to target graph-structured data.

References

[1] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–5.

[2] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 363–374.

[3] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019, pp. 918–934.

[4] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[5] L. Dhulipala, G. E. Blelloch, Y. Gu, and Y. 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*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 108–121. [Online]. Available: https://doi.org/10.1145/3519939.3523733

[6] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[7] D. De Leo and P. Boncz, "Teseo and the analysis of structural dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 14, no. 6, pp. 1053–1066, 2021.

[8] B. Wheatman and H. Xu, *A Parallel Packed Memory Array to Store Dynamic Graphs*, 2021, pp. 31–45. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611976472.3

[9] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," *ACM Transactions on Storage (TOS)*, vol. 15, no. 4, pp. 1–40, 2020.

[10] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, "Livegraph: A transactional graph storage system with purely sequential adjacency list scans," *arXiv preprint arXiv:1910.05773*, 2019.

[11] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, "Terrace: A hierarchical graph container for skewed dynamic graphs," in *Proceedings of the 2021 international conference on management of data*, 2021, pp. 1372–1385.

[12] B. Wheatman, R. Burns, A. Buluc, and H. 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*, ser. PPoPP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 348–363. [Online]. Available: https://doi.org/10.1145/3627535.3638492

[13] A. van der Grinten, M. Predari, and F. 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.

[14] J. Shi, B. Wang, and Y. Xu, "Spruce: a fast yet space-saving structure for dynamic graph storage," *Proc. ACM Manag. Data*, vol. 2, no. 1, mar 2024. [Online]. Available: https://doi.org/10.1145/3639282

[15] P. Fuchs, D. Margan, and J. Giceva, "Sortledton: a universal, transactional graph data structure," *Proceedings of the VLDB Endowment*, vol. 15, no. 6, pp. 1173–1186, 2022.

[16] A. Itai, A. G. Konheim, and M. Rodeh, "A sparse table implementation of priority queues," in *ICALP*, 1981, pp. 417–431.

[17] B. Wheatman, X. Dong, Z. Shen, L. Dhulipala, J. Łącki, P. Pandey, and H. Xu, "Byo: A unified framework for benchmarking large-scale graph containers," *Proc. VLDB Endow.*, vol. 17, no. 9, p. 2307–2320, 8 2024. [Online]. Available: https://doi.org/10.14778/3665844.3665859

[18] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[19] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 399–409.

[20] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *PPoPP*, 2013, pp. 135–146.

[21] Amazon, "Amazon web services," https://aws.amazon.com/, 2024.

[22] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Available at http://snap.stanford.edu/data, Jun. 2014.

[24] "9th dimacs implementation challenge - shortest paths," http://www.dis.uniroma1.it/challenge9/, accessed: 2023-09-25.

[25] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. 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*, 2006, pp. 44–54.

[26] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012. [Online]. Available: http://arxiv.org/abs/1205.6233

[27] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SDM*, 2004, pp. 442–446.

[28] P. Erdös and A. Rényi, "On random graphs I," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.

[29] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç, "Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks," *Nucleic acids research*, vol. 46, no. 6, pp. e33–e33, 2018.

[30] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Advances in neural information processing systems*, vol. 33, pp. 22 118–22 133, 2020.

[31] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *European conference on principles of data mining and knowledge discovery*. Springer, 2005, pp. 133–145.