A Parallel Packed Memory Array to Store Dynamic Graphs

Brian Wheatman* Helen Xu†

Abstract

The ideal data structure for storing dynamic graphs would support fast updates as well as fast range queries which underlie graph traversals such as breadth-first search. The Packed Memory Array (PMA) seems like a good candidate for this setting because it supports fast updates as well as cache-efficient range queries. Concurrently updating a PMA raises challenges, however, because an update may require rewriting the entire structure.

This paper introduces a parallel PMA with intra- and inter-operation parallelism and deadlock-free polylogarithmic-span operations. Our main observation is that the PMA is well-suited to concurrent updates despite occasionally requiring a rewrite of the entire structure because 1) most of the updates only write to a small part of the structure and 2) the worst case is highly parallel and cache-efficient.

To evaluate our data structure, we implemented Parallel Packed Compressed Sparse Row (PPCSR), a dynamic-graph processing framework that extends the Ligra interface with graph updates. We show that PPCSR is on average about 1.6x faster on graph kernels than Aspen, a state-of-the-art graph-streaming system. PPCSR achieves up to 80 million updates per second and is $2 - 5$x faster than Aspen on most batch sizes. Finally, PPCSR is competitive with Ligra and Ligra+, two state-of-the-art static graph-processing frameworks.

1 Introduction

Since many real-world sparse graphs change in real-time, there has been significant research effort devoted to systems for storing and processing dynamic graphs [19, 28, 33, 18, 10, 21, 22]. These systems must process a stream of updates (e.g. edge-weight update, or edge insertions and deletions) and a stream of queries quickly. That is, both update latency and query processing time must be fast. In this paper, we focus on parallel data structure design optimized specifically for fast cache-efficient range queries[1] while still maintaining fast updates.

A suitable data structure for dynamic graphs must support efficient vertex neighbor queries in order to gather a vertex's neighbors for the next phase of the algorithm. Many graph algorithms, such as breadth-first search and betweenness centrality, can be expressed by iteratively processing a set of active vertices and their neighbors [42]. Therefore, efficient data structures for graph processing should store neighbors as close as possible for locality during range queries.

There is a tradeoff between update and range query performance in data structure design. For example, a hash table can achieve $O(1)$ amortized update cost [14, Chapter 11], but a range query $r(u, v)$ must take $O(v-u)$ work. At the other extreme, a range query in a sorted array with n elements takes $O(\log n + k)$ work, where k is the number of elements in the range, but updating a sorted array takes $O(n)$ work.

In the static setting, Compressed Sparse Row (CSR) [47], a canonical storage format for sparse graphs, achieves optimal performance for range queries by storing edges in a contiguous sorted array. Unfortunately, CSR is a static storage format: adding an edge to CSR may require shifting the entire edge array. Inspired by the cache-friendliness of CSR, Packed Compressed Sparse Row (PCSR) [50] replaced the edge array in CSR with a Packed Memory Array (PMA) [24, 5] for (amortized) $O(\log^2 |E|)$ update cost and asymptotically optimal range queries.

There are a couple of factors that make PCSR a good candidate for processing dynamic graphs beyond its theoretical guarantees. First, the observed update cost of PCSR is much better in practice than its theoretical bound might suggest because the worst-case rewrites are cache-efficient [50]. Additionally, PCSR avoids pointer indirections in contrast with non-contiguous data structures such as search trees (e.g. B-trees), which require pointer chasing. Finally, PCSR supports efficient scans and has good cache locality because the elements are laid out contiguously in memory.

1.1 Parallelization Strategies In this paper, we propose parallel modifications to augment the PMA with both intra- and inter-operation parallelism to improve the performance. ***Intra-operation parallelism*** exploits logically parallel work present in the operations themselves, while ***inter-operation parallelism***

*John Hopkins University. Email: wheatman@cs.jhu.edu.

†Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology. Email: hjxu@mit.edu.

[1]A range query $r(u, v)$ in a data structure takes two indices u, v and returns all elements in the range $[u, v]$.

enables multiple threads to update or query the data structure at the same time.

For example, a PMA could support inter-operation parallelism without intra-operation parallelism by running the operations at the same time, but doing the work of each sequentially [6, 15].

We shall analyze the costs of intra-operation parallelism using the ***work-span model*** [14, Chapter 27]. The ***work*** is the total time to execute the entire algorithm on one processor. The ***span***[2] is the longest serial chain of dependencies in the computation (or the runtime in instructions on an infinite number of processors). The ***parallelism*** of an algorithm is the work divided by the span. In the work-span model, a parallel for loop on N iterations with $O(1)$ work per iteration has $O(N)$ work and $O(\log N)$ span because loops can be implemented with divide-and-conquer fork-join parallelism.

The PMA is well-suited to intra-operation parallelization because the expensive operations are highly parallel. In the worst case, an update in a PMA with n elements may require rewriting the entire structure, which takes $O(n)$ work. This work can be parallelized, however. As we will see, updating a PMA has $O(\log^2 n)$ span in the worst case.

Furthermore, we will use the ***shared-memory multiple-writer / multiple-reader model*** for inter-operation parallelism for generality.

There are several challenges in supporting inter-operation parallelism in a PMA when compared to search trees. In parallel search trees with locking, updates or queries may only need to acquire a few locks at a time (e.g. in hand-over-hand locking) to do an update or a query. Furthermore, purely functional trees may not even require locking because they can take a snapshot without traversing the entire structure [18]. These tree-based locking or snapshotting schemes do not directly translate to a PMA. Furthermore, an update to a search tree requires updating only a few nodes and pointers, while an update to a PMA (in the worst case) may require table doubling and rewriting the entire structure [50], which would seem to put the PMA at a disadvantage in terms of the fraction of the structure that needs to be locked. Previous work confirms this intuition: a PMA with locking and multiple writers achieves much lower update throughput when compared to search-tree variants optimized for writes [15].

We will show that a parallel PMA with locking can simultaneously achieve high update throughput and fast queries. The worst case of rewriting the entire structure during an update not only happens extremely rarely, but also is fast in practice because it is cache-efficient.

[2]Sometimes called **critical-path length** or **computational depth**.

1.2 Contributions

We describe ***Parallel Packed Compressed Sparse Row*** (PPCSR), a graph storage format based on a PMA with parallel modifications to support both inter- and intra-operation parallelism. Along the way, we show how to parallelize a PMA with polylogarithmic span for each operation. Furthermore, we introduce a deadlock-free locking scheme with polylogarithmic span[3].

We implemented PPCSR and found that it enables fast serializable phased updates and queries. That is, multiple writers can update concurrently, or multiple readers can read concurrently, but not both. To enable queries PPCSR extends the interface from Ligra [42], a static graph-processing framework. Therefore, all algorithms implemented with Ligra, such as graph-traversal algorithms, local graph algorithms [44], and others [17, 16] can be run on top of PPCSR with minor cosmetic changes.

We evaluate PPCSR and compare it to Aspen [18], Ligra [42], and Ligra+ [43], three state-of-the-art graph processing frameworks. Aspen is a graph-streaming framework, while Ligra and Ligra+ are static graph-processing frameworks. Although we expect the static graph-processing frameworks to outperform dynamic systems, we compare them on query cost to evaluate the cost of updatability. Therefore, we compare Aspen and PPCSR on update throughput, and all systems on graph kernel performance.

PPCSR achieves up to 80 million updates per second. As shown in Figure 2, PPCSR is $2-5$x faster than Aspen on small-batch updates but between $2-5$x slower on batch sizes of at least 10 million.

Furthermore, PPCSR supports efficient queries. As shown in Figure 1, PPCSR outperforms Aspen by about 1.6x on average on the four tested graph kernels. PPCSR is competitive with Ligra and Ligra+. On average, PPCSR is 1.25x slower than Ligra.

To be specific, our contributions are as follows:

- The design and theoretical analysis of a parallel PMA that supports intra- and inter-operation parallelism.
- An implementation of PPCSR on top of the parallel PMA using Cilk [23].
- An experimental study of PPCSR compared to Aspen, Ligra, and Ligra+ that demonstrates that PPCSR supports efficient updates and queries.

Map The rest of the paper is organized as follows. We present necessary preliminaries in Section 2. We describe modifications to the PMA to make parallelization easier

[3]Assuming grabbing a lock takes $O(1)$ work.

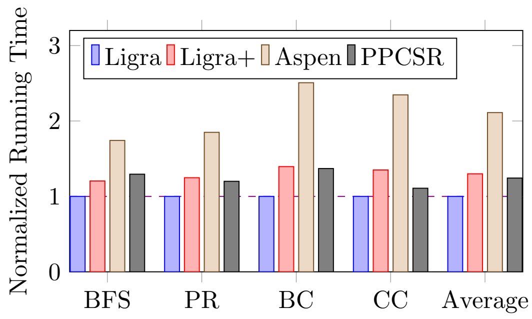

Figure 1: Time to run kernels normalized to Ligra averaged across all graphs. The four kernels tested were breadth-first search (BFS), PageRank (PR), betweenness centrality (BC), and connected components (CC).

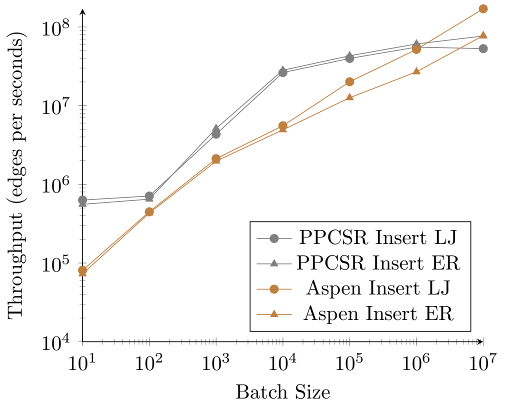

Figure 2: Insert throughput as a function of batch size on the LJ and ER graphs. The LJ graph is about 85 million edges, while the ER graph is about 1 billion edges.

in Section 3. We show how to exploit intra- and inter-operation parallelism in the PMA in Sections 4 and 5. We then review serial PCSR and describe how to augment it with locks to enable multiple writers in Section 6. We present the results from the experimental evaluation in Section 7. Finally, we review related work on graph processing systems in Section 8 and conclude in Section 9.

2 Preliminaries

In this section we will review necessary background to understand later sections. First, we describe a few parallel primitives that we will use to analyze and implement operations in PPCSR. Next, we review a serial Packed Memory Array. Finally, we introduce notation for describing graphs.

Parallel prefix sum. The `prefix_sum`(A, N) operation takes as input a list A of N numbers and outputs a list A' where $\forall i \in \{0, 1, \ldots, N-1\}$,

$$A'[i] = \sum_{j=0}^{i} A[i].$$

Parallel implementations of prefix sum [8] can be done in place in $O(N)$ work and $O(\log N)$ span.

Parallel memcpy. The `memcpy(src, dest, size)` copies `size` bytes of data from location `src` to location `dest`. It can be implemented in parallel using a single parallel for loop in $O(\texttt{size})$ work and $O(\log(\texttt{size}))$ span.

2.1 Packed Memory Array

A Packed Memory Array [5, 24] (PMA) maintains elements in order in an array with spaces between its elements. A PMA holds n elements in $N = O(n)$ cells and supports updates with amortized $O(\log^2 n)$ work. Point queries in a PMA take $O(\log n)$ work, and range queries $r(s, t)$ that return k elements have $O(\log n + k)$ work.

The PMA is composed of a contiguous implicit complete binary tree with leaves of size $\log N$. That is, the implict tree has $N/\log N$ leaves and height $\log(N/\log N)$. Each leaf $i \in \{0, \ldots, N/\log N - 1\}$ encompasses cells in the region $[i \log N, (i+1)\log N)$, and each internal node encompasses all of the cells of its descendants. The height of a node is the distance from that node to a leaf.

Each node of the PMA tree has an ***upper and lower density bound*** on its density, or the fraction of occupied cells in its region. When breached, these bounds are enforced by redistributing elements among its neighboring nodes, equalizing the densities between them.

Operations. A PMA implements three ***external operations***:

- `insert`: inserts an element into the PMA.
- `delete`: deletes an element from the PMA.
- `search`: finds an element in the PMA.

Range queries in a PMA can be implemented by searching for the start of the range and doing a forward scan until the end of the range.

In order to implement the external operations, a PMA also supports the following ***internal operations*** as subroutines:

- `count_non_nulls`: returns the number of elements of each PMA leaf in a specified region.
- `redistribute`: spreads elements from a node evenly among the leaves in the subtree rooted at that node.
- `double_pma`: doubles the size of the PMA.
- `halve_pma`: halves the size of the PMA.

The functions `count_non_nulls` and `redistribute` take start and end indices s, t that must be at the beginning and end of PMA nodes, respectively (i.e. $s, t \bmod \log N = 0$). Additionally, $(t - s)/\log N = 2^x$ for some non-negative integer x.

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

A graph $G = (V, E, w)$ is a set of vertices[4] V, a set of edges E, and an edge weight function w. We denote the number of vertices $|V|$, the number of edges $|E|$, and the degree of a vertex $v \in V$ is $\deg(v)$. Each vertex $v \in V$ is represented by a unique non-negative integer less than $|V|$ (i.e. $v \in \{0, 1, \ldots, |V| - 1\}$). Each edge is a 2-tuple (u, v) where $u, v \in V$. Finally, the weight function w maps each edge $e \in E$ to a non-zero real weight ($w(e) \in \mathbb{R}, w(e) \neq 0$).

3 PMA modifications

We give several modifications that can be made to a PMA which will aid in parallelizing it without impacting its theoretical guarantees.

Density bound. To ensure that parallel threads can always insert without waiting or blocking, we add a stricter upper density bound to the leaves of the PMA that ensures leaves are never completely full. Given an original upper density bound at the leaves d_{leaf}, the new upper density bound at the leaves of the PMA is $\min(d_{\text{leaf}}, (\log N - 1)/\log N)$. Since

$$\lim_{N\to\infty} (\log N - 1)/\log N = 1,$$

the additional density requirement does not impact the asymptotic behavior of the PMA. The extra bound ensures that a thread can always place an element immediately into the PMA and will only wait in the redistribute phase of an insert.

Packed-left property. To parallelize locking, we enforce a ***packed-left*** property of the nodes in the PMA so that inserts into one region do not spill over into others. Instead of evenly distributing elements in the PMA leaves, we put them all contiguously at the beginning of the leaf. The packed-left property along with the non-full density bound ensure that a thread will never shift elements into another node's region, which facilitates locking. Similarly, a delete would re-compress elements to the left at the beginning of each leaf.

Scanning over a PMA with the packed-left property asymptotically reduces the number of wasted accesses. When scanning over a standard PMA, each cell is checked

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

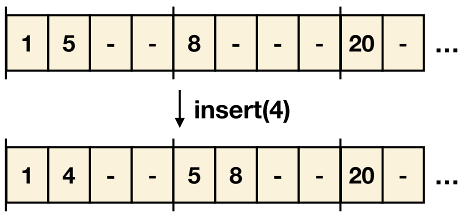

Figure 3: An example of inserting into a PMA with a leaf size of 4 and a leaf density bound of 0.5.

to see if it is null or not. The packed-left property reduces the number of empty cells evaluated to $O(N/\log N)$ from $O(N)$ because a pass through each leaf evaluates at most one empty cell.

The packed-left property maintains the work bounds of the original PMA because the original PMA evenly distributes elements in a leaf after inserting into that leaf [5, 7], which requires reading and writing to each cell in that leaf. In the worst case, inserts into a PMA with the packed-left property also require reading and writing to each cell in the associated leaf. Furthermore, a PMA with the packed-left property maintains the cache-efficiency of the original PMA.

4 Intra-operation parallelism

In this section we prove that all the PMA operations have polylogarithmic span. Section 2 describes the primitive parallel operations `memcpy` and prefix sum. Given an input of size n, parallel `memcpy` and prefix sum have $O(n)$ work and $O(\log n)$ span. All other omitted subroutines and proofs appear in Appendix A.

4.1 Internal Operations The `redistribute` function enforces the density bound of a region in the PMA. Specifically, the `redistribute(s, t)` function guarantees that all nodes in the region defined by s, t respect their density bounds.

THEOREM 4.1. *`redistribute(s, t)` has $O(t-s)$ work and $O(\log N)$ span.*

Proof. The pseudocode[5] for `redistribute(s, t)` can be found in Figure 4.

By Lemma A.1, the call to `count_non_nulls(s, t)` has $O(t-s)$ work and $O(\log N)$ span. The function then prefix sums all the leaves in the range in $O(t - s)$ work and $O(\log N)$ span.

The first `parallel_for` has $O(t - s)$ work and $O(\log N)$ span. The second `parallel_for` iterates over the number of leaves, which is $(t-s)/\log N$, so the span

[5]Unless otherwise specified, all divisions in pseudocode are integer division (rounded down).

```
def redistribute(s, t):
  counts = count_non_nulls(s, t)
  temp[t - s] # create array
  parallel_prefix_sum(counts)
  # copy and pack all edges to temp
  parallel_for k in [s, t); k += log(N):
    if i == s: start = 0
    else: start = counts[i-1]
    for j in [k*log(N), (k+1)*log(N)):
      if pma[j] is not null:
        temp[start] = pma[j], start++
      pma[j] = null

  num_leaves = (t - s) / log(N)
  end_idx = counts.size - 1
  leaf_avg = counts[end_idx] / num_leaves
  extra = counts[end_idx] % count_per_leaf

  parallel_for i in [0, num_leaves):
    # number of items for this leaf
    for_leaf = leaf_avg + (i < extra)
    # start of leaf in temp and in PMA
    tmp_start = leaf_avg*i + min(i, extra)
    leaf_start = s + (i * log(N))

    # copy edges into PMA
    memcpy(&pma[leaf_start],
           &temp[tmp_start], for_leaf)
```

Figure 4: Pseudocode for `redistribute(s, t)`.

of the second `parallel_for` is

$$O(\log((t-s)/\log N)) = O(\log(N/\log N)).$$

Therefore, the work and span of this `parallel_for` are $O(t-s)$ and $O(\log N)$, respectively.

The total work and span of `redistribute(s, t)` are therefore $O(t-s)$ and $O(\log N)$, respectively. □

Resizing the PMA. If the PMA becomes too dense or sparse, it may have to be resized with the `double_pma` and `halve_pma` functions. Given a PMA of N cells, both subroutines take $O(N)$ work and $O(\log N)$ span. At a high level, the functions densify the data, resize the PMA, and redistribute the data into the new size. The details can be found in Appendix A.

4.2 External Operations Next, we describe and analyze the `insert` function. The `insert(lo, hi, v)` function inserts an element v into a sorted region beginning at index `lo` and ending at index `hi`. Each insert in a PMA requires a search to find the location to insert the element, which has $O(\log N)$ span because it is a binary search on the PMA. Appendix A describes binary searching with null values in a PMA.

```
# inserts the element v in sorted order
def insert(lo, hi, v):
  depth = log(N / log(N)), height = depth
  index = search(lo, hi, v)
  # slide elements to the right until a
     null space is found
  slide_right(index)
  pma[index] = v
  # range of this leaf we inserted into
  start = (index / log(N)) * log(N)
  end = start + log(N)
  counts = count_non_nulls(start, end)
  # non-integer division
  density = float(counts[0]) / log(N)
  while density > density_bound(depth):
    # get start and end of parent nodes
    start = get_parent_start(start, depth)
    end = get_parent_end(end, depth)
    count = get_element_count(start, end)
    density = float(count) /
      (log(N) >> (height - depth))
    depth = depth - 1
    if depth < 0:
      double_pma()
      return
  redistribute(start, end)
```

Figure 5: Pseudocode for inserting into a PMA.

Inserting into a PMA takes amortized $O(\log^2 N)$ work [5] with the parallel modifications as described in Section 3.

THEOREM 4.2. *`insert(lo, hi, v)` has $O(\log^2 N)$ worst-case span.*

Proof. The pseudocode for the `insert(lo, hi, v)` function can be found in Figure 5. By Lemma A.3, the `search(lo, hi, v)` function has $O(\log N)$ span. The slide-right function touches at most $O(\log N)$ cells of the PMA, so it also has $O(\log N)$ span. There are at most $O(\log N)$ calls to `count_non_nulls(s, t)` and `parallel_sum`, which each have $O(\log N)$ span by Lemma A.1. Lastly, there is one call to either `double_pma` or `redistribute(s, t)`, which have $O(\log N)$ span by Lemma A.2 and Theorem 4.1. □

The bound in Theorem 4.2 is tight for the worst case when we must redistribute the entire PMA. The worst case is rare, however, and only happens once every $O(N)$ operations. We can also analyze the ***amortized span***, or the total span of a set of parallel operations performed one at a time.

THEOREM 4.3. *`insert(lo, hi, v)` has $O(\log N)$ amortized span.*

Proof. The pseudocode for the `insert(lo, hi, v)` function can be found in Figure 5. The bulk of this proof will focus on analyzing the costs of `count_non_nulls(s, t)` function, which vary over inserts.

As before, `search` and `slide_right` have $O(\log N)$ span.

Each insertion requires counting the elements in the corresponding leaf to check its density, which has $O(\log \log N)$ span. This is done by the helper routine `get_element_counts` which returns the number of elements in a region by counting them in parallel with logarithmic span. For every $N/\log N$ insertions, we will have to redistribute a larger section. Specifically, we will have to redistribute 2^i leaves every $N/(2^i \log N)$ insertions for positive integers i.

Let $H = \log(N/\log N)$, the height of the PMA. We calculate the "extra" span $T(N)$ of these redistributes over N insertions.

$$\begin{aligned}
T(N) &= \frac{N}{\log N}\sum_{j=1}^{H}\frac{1}{2^j}\Big(\sum_{i=1}^{j}\log(2^i\log N)\Big)\\
&= \frac{N}{\log N}\sum_{j=1}^{H}\frac{1}{2^j}\Big(j\log\log N + \sum_{i=1}^{j} i\Big)\\
&= \frac{N}{\log N}\Big(\log\log N\sum_{j=1}^{H}\frac{j}{2^j} + \sum_{j=1}^{H}\frac{j(j+1)}{2^{j+1}}\Big)\\
&\le \frac{N}{\lg N}\left(2\log\log N + 4\right) = O\Big(\frac{N\log\log N}{\log N}\Big).
\end{aligned}$$

The "total span" of counting the non-nulls over N insertions is therefore $O\left((N\log\log N)/\log N + N\log\log N\right)$ Dividing the "total span" by N yields $O(\log\log N)$ amortized span for the calls to `count_non_nulls(s, t)` over N insertions.

There is one call to either `double_pma` or `redistribute(s, t)` on each insertion, which both have $O(\log N)$ span by Lemma A.2 and Theorem 4.1. ∎

Deleting an element from the PMA is symmetric to inserting an element and has $O(\log^2 N)$ work, $O(\log^2 N)$ worst-case span, and $O(\log N)$ amortized span. A delete requires a search to find the element, a slide left to overwrite it, and a `redistribute(s, t)` with lower density bounds to maintain the density requirements.

5 Inter-operation parallelism

To allow for multiple writers in parallel, we augment the PMA with one lock per leaf[6] of the PMA. To maintain the bounds from Section 4, we describe a locking scheme with polylogarithmic worst-case span assuming grabbing a lock takes $O(1)$ work.

Description of locks. We implemented reader-writer locks with a ranking system for prioritizing `redistribute`. When unlocking a lock, a thread can mark the lock so that the lock can only be taken by another thread with higher rank.

Grabbing locks in parallel. We show how to grab locks in parallel without deadlock and with polylogarithmic span. The only time we need to grab multiple locks at once is on a `redistribute`, where a thread will grab all the locks in the subtree rooted at the node it is redistributing.

We will now describe a scheme for grabbing contiguous sequences of locks on *leaves* in parallel called `lock_order` according to implicit priorities of each leaf in the PMA. The `lock_order` algorithm serially iterates over each priority in order and grabs the locks with that priority in parallel.

The `lock_order` algorithm first assigns implicit priorities to each leaf in the PMA depending on its index. The ***priority*** of a leaf with index i is `popcount`(i). The `popcount` function returns the number of ones in the bit representation of a number. For example, since $5 =$ `0b101`, `popcount`$(5) = 2$. We provide an example of how to assign priorities to nodes in Figure 6.

Remark 1. *The height of the root of any subtree defines the "pattern" of* `popcounts`*, but not the minimum popcount of the leaves in that subtree. For example, consider leaves 0-3 and 4-7 in Figure 6, which correspond to two subtrees with roots at the same level. The* `popcounts` *of consecutive leaves in each subtree have the same differences between them but have different minimums in the different subtrees. The unique minimum priority of any leaf in a subtree is the priority of the first leaf in that subtree. Consider leaves 4-7 in the second: their minimum* `popcount` *is 1 because the upper bit must be set (4 =* `0b100`*).*

We now prove that this parallel locking scheme is deadlock-free and has polylogarithmic span.

Theorem 5.1. *Grabbing locks for any two nodes in the PMA using* `lock_order` *is deadlock-free.*

Proof. We will show the theorem using case analysis. Suppose two threads are trying to grab locks for two nodes a and b. We denote the set of leaves in the subtree rooted at some node γ with leaves(γ).

Case 1: leaves$(a) \cap$ **leaves**$(b) = \emptyset$**.** Since the regions have no locks in common, grabbing them in parallel will not cause deadlock.

[6]Locking each leaf is equivalent to locking nodes at any set depth in the tree, which trades off between locking overhead and parallelism.

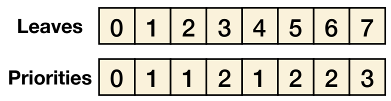

Figure 6: The indices of leaves in a PMA and the associated priorities.

Case 2: leaves(a) = leaves(b). If $a = b$, there will be a unique leaf with lowest priority according to Remark 1. The thread that grabs it first will grab the rest of the region while the other one waits for it, avoiding circular wait.

Case 3: leaves(a) ⊂ leaves(b) (w.l.o.g.). Let left$_a$ be the leftmost leaf in leaves(a). Since left$_a$ has smaller priority than all the other leaves in leaves(a), both threads will attempt to grab it before any other leaf in leaves(a). Therefore, whoever grabs left$_a$ will be able to grab leaves(a) first. There is no circular wait because the thread trying to grab the locks of a need no locks outside of leaves(a).

In all cases, there is no circular wait and therefore no deadlock. □

Lemma 5.1. *Grabbing all the locks for any node in the PMA according to* `lock_order` *has polylogarithmic span assuming $O(1)$ work to grab a lock.*

Proof. There are at most $\log N$ distinct priorities because there are at most $\log N$ bits required to represent the priority of a node. Furthermore, there are at most N locks with each priority, so grabbing all the locks with a given priority in parallel has $O(\log N)$ span. Therefore, the total span is $O(\log^2 N)$ in the worst case. □

Since most operations take locks for a small region of the PMA (e.g. inserts or small redistributes), it is rare to have to wait on another thread with a lock.

6 Parallel Packed Compressed Sparse Row

We review the serial Packed Compressed Sparse Row [50] (PCSR) data structure based on the PMA and describe a locking protocol for PPCSR in order to enable multiple parallel writers. Appendix B describes how to implement graph operations in parallel using the operations described in Section 4.

Compressed Sparse Row. Compressed Sparse Row (CSR) is a common storage format for sparse graphs [47, 36]. It stores a graph as a set of three dense arrays: a vertex array, an edge array, and a weights array. The edge array holds the edges first sorted by source, then by destination. The weights array stores the weights according to the order of edges in the edge array. The vertex array has one entry for each vertex

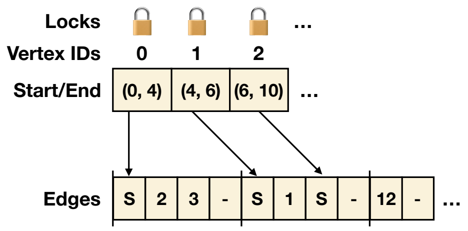

Figure 7: An example of a graph stored in PPCSR format. "S" denotes a sentinel at the beginning of a vertex's region in the edge PMA. The tall lines denote leaf boundaries and elements are packed left in leaves.

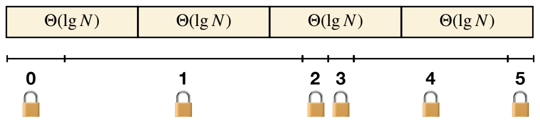

Figure 8: An example of the edge PMA in PPCSR with locks on vertices. The boxes represent the leaf boundaries of the PMA and the lines under the PMA represent regions associated with vertices in the graph (with their corresponding locks).

corresponding to the start of its region in the edge and weight array.

Packed Compressed Sparse Row. PCSR replaces the dense edge and weight array of CSR with a PMA. Each cell in the vertex list stores pointers to the beginning and end of the region in the edge PMA corresponding to the edges of that vertex.

PCSR also stores sentinels at the beginning of a vertex's region in the edge PMA. ***Sentinels*** are special elements that hold pointers to the region's source in the vertex array. These sentinels facilitate updates to the vertex list when elements are shifted in the edge PMA.

PCSR requires a constant factor more space than CSR. The vertex array takes twice as much space because it stores a pointer to the beginning and end of each region. The edge PMA takes $O(m + n)$ cells compared to the m cells in CSR.

Figure 7 contains an example of a graph stored in PPCSR.

In Section 5, we described how to lock a traditional PMA with one lock per node. In PPCSR, where there may be more than one lock per node from multiple vertices, grabbing all the associated vertex locks can be done sequentially. that lock to a higher level of the PMA tree. We present an example of how vertex regions might be distributed among PMA nodes in Figure 8.

7 Empirical evaluation

In this section, we empirically evaluate PPCSR and compare it with Aspen [18], a state-of-the-art graph-streaming system, and Ligra [42]/Ligra+ [43], two static graph-processing systems. We evaluate all systems

Name	Vertices	Edges	Avg. Deg.
LiveJournal (LJ)	$4,847,571$	$85,702,474$	17.8
Orkut	$3,072,627$	$234,370,166$	76.2
rMAT	$8,388,608$	$563,816,288$	60.4
Erdős-Rényi (ER)	$10,000,000$	$1,000,009,436$	100

Table 1: Sizes of (symmetrized) graphs used.

on algorithm speed and memory usage, and Aspen and PPCSR on update throughput. We implemented four algorithms in PPCSR: breadth-first search (***BFS***), single-source betweenness centrality (***BC***), PageRank (***PR***), and connected components (***CC***).

Experimental setup. We implemented PPCSR as a `c++` library parallelized using `Cilk Plus` [23] and the Tapir [37][38] branch of the LLVM [29, 30] compiler. We compiled Aspen, Ligra, and Ligra+ with `g++` version 7.5.

All experiments were run on a 48-core 2-way hyper-threaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz with 189GB of memory from AWS [1].

Types of graphs. We tested on both real social network graphs and synthetic graphs. Social network graphs usually have a few very high-degree vertices while the rest of the vertices have low degree according to a power-law distribution [4]. We used the ***LiveJournal*** (LJ) and ***Orkut*** social network graphs from the SNAP dataset [31]. LiveJournal is a directed graph of the LiveJournal social network [9], and Orkut is an undirected graph of the Orkut social network. We also generated a random (***rMAT***) graph by sampling edges from an rMAT generator [12] with $a = 0.5; b = c = 0.1; d = 0.3$ to match the distribution from Aspen [18]. Finally, we generated a random Erdős-Rényi (***ER***) graph [20] with $n = 10,000,000$ and $p = 0.000005$ which was then symmetrized.

We used symmetrized versions of all the graphs for a fair comparison with the publicly available version of Aspen, which supports only unweighted undirected graphs. To store undirected unweighted graphs in PPCSR, we store directed edges both ways with weight 1. The sizes of all the graphs can be found in Table 1.

Since LiveJournal and Orkut are static graphs which may have been pre-processed with vertex reordering [49], we randomly relabel the vertices in all the input graphs to model the dynamic setting. Reordering is more difficult in streaming graphs because a good ordering may change with the stream of edges [3].

System descriptions. PPCSR and Aspen differ significantly in their underlying data structures and parallelization approaches. Aspen takes a purely functional approach with compressed trees, while PPCSR modifies a single parallel PMA with locks directly. Aspen is a compressed tree with difference encoding [45], whereas PPCSR is uncompressed. Aspen allows read-only operations (e.g. queries) during writing transactions, and vice versa (i.e. it does not use locks). It requires that the writer is sequentialized, however. In contrast, PPCSR supports concurrent readers or writers but uses locks, which prevents concurrent reading and writing in the same region of the data structure.

For simplicity, we implemented a locking scheme that grabs locks in a serial forward pass in PPCSR rather than according to the priority-based scheme described in Section 5. Since there is still an order to the locks, the forward-pass method is also deadlock-free. Although this method is not logarithmic in the worst case, almost all the operations only modify a small region of the PMA, so a thread usually only has to grab a constant number of locks.

Ligra is a static graph processing system that uses CSR as its underlying graph representation. Ligra+ adds data compression on top of the Ligra CSR representation.

7.1 Updates

We show that the batch insertions in PPCSR achieves up to 80 million edges per second for batch insertions and report our findings in Figure 2 and Table 2. To further optimize for large batches, PPCSR supports merging in a batch of edges. PPCSR outperforms Aspen on batches of up to $1,000,000$ edges, while Aspen is faster on batch sizes of at least $10,000,000$.

Setup. To generate our edges, we sample directed edges from the same rMAT generator that we used to generate the synthetic rMAT graph. To evaluate our insertion and deletion throughput, we add batches of directed edges to the LJ and ER graph in parallel (with potential duplicates). We report the average of 20 trials on small batches and the average of 5 trials on large batches.

Discussion. PPCSR is $2-5$x faster than Aspen on batch sizes up until $100,000$, competitive with Aspen on batches of $1,000,000$, but does not scale with larger batch sizes as Aspen does. However, most highly dynamic graphs require much less throughput for huge batches. For example, Twitter averages $5,700$ tweets per second, and peaked at $140,000$ tweets per second [35].

Aspen implements batch insertions as a per-vertex merge, while PPCSR implements batch insertions as concurrent point insertions. For a batch size of B edges and a PPCSR representation with $|E|$ edges, merging in the batch takes $O(B+|E|)$ work. Since it is theoretically better to perform a merge when the batch size is very large ($B \approx O(|E|/\log^2|E|)$), PPCSR supports merging in very large batches. Since insert and delete throughput in both systems were comparable, we illustrate only the insert throughput in Figure 2.

	Insert						Delete					
	LJ			ER			LJ			ER		
Batch Size	PPCSR	Aspen	P/A	PPCSR	Aspen	P/A	PPCSR	Aspen	P/A	PPCSR	Aspen	P/A
1.00E1	6.31E5	8.07E4	7.82	5.57E5	7.28E4	7.65	1.59E6	8.39E4	18.9	2.06E6	7.49E4	27.5
1.00E2	7.10E5	4.48E5	1.59	6.48E5	4.32E5	1.50	1.52E6	4.71E5	3.22	1.56E6	4.28E5	3.65
1.00E3	4.34E6	2.12E6	2.05	5.13E6	1.97E6	2.61	7.80E6	2.24E6	3.49	8.24E6	2.12E6	3.89
1.00E4	2.63E7	5.55E6	4.74	2.82E7	4.93E6	5.71	3.03E7	6.25E6	4.85	3.18E7	5.44E6	5.83
1.00E5	3.98E7	2.01E7	1.98	4.30E7	1.26E7	3.42	4.74E7	2.02E7	2.35	5.10E7	1.18E7	4.31
1.00E6	5.54E7	5.18E7	1.07	6.08E7	2.69E7	2.26	7.64E7	5.15E7	1.48	7.90E7	2.66E7	2.97
1.00E7	5.30E7	1.70E8	0.31	7.67E7	7.76E7	0.99	7.98E7	1.70E8	0.47	8.29E7	7.97E7	1.04
1.00E8	2.08E8	4.56E8	0.46	4.68E7	2.50E8	0.19	2.43E8	4.98E8	0.49	7.87E7	2.72E8	0.29

Table 2: Throughput for inserting and deleting edges with varying batch sizes in the LJ and ER graphs in PPCSR and Aspen. P/A denotes the ratio of the respective throughputs (PPCSR/Aspen).

In practice, memory bandwidth is the main bottleneck in insertions in PPCSR because every insert requires a cache-inefficient binary search. Although theoretically insertions into the ER graph should be slower than insertions into the LJ graph because the ER graph is much bigger, in practice they are similar because the size of the binary search each insertion requires is similar.

PPCSR supports insertions much faster than its worst-case theoretical bound of $O(\log^2 |E|)$ would suggest. The theoretical bound is given by an amortization of the rebalances, but in practice the rebalances are extremely cache-efficient.

7.2 Query performance We evaluate the performance of PPCSR, Aspen, Ligra, and Ligra+ on BFS, PR, (single-source) BC, CC and report the exact runtimes in Table 3.

Algorithm setup. In order to run all algorithms using the same API as the other systems, we implemented the `EdgeMap` / `VertexSubset` interface proposed by Ligra in PPCSR. The `VertexSubset` in PPCSR has an additional optimization for the case when the frontier is all the vertices, which improves PR and CC. We keep track of whether the frontier is full and skip membership queries if it is.

For PR, we removed the early exit and damping from the Ligra implementation, ported it into PPCSR, and verified the correctness of the translation into Aspen in private communication. For BFS and BC, we ported the Ligra implementation into PPCSR and ran the native Aspen implementations. For CC, we converted the Ligra implementation into PPCSR and Aspen. For BFS and BC, we ran all systems starting from the same vertex.

We implemented all kernels in PPCSR assuming undirected graphs to compare with Aspen. For each graph kernel, we took the average of ten trials.

PageRank. Figure 9 illustrates the relative speed on PR of all the systems. On all the graphs we tested, PPCSR achieves between 1.2 – 2x speedup over Aspen on PR because PPCSR supports fast ordered traversals. Furthermore, PPCSR is competitive with Ligra and Ligra+ on PR (between 1−1.6x slower). PR is essentially a linear scan through the PMA because it iterates through all vertices.

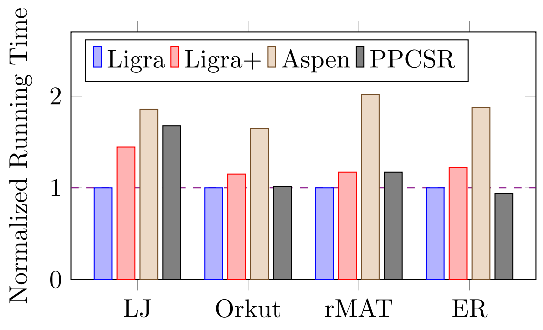

Figure 9: Time for all systems to calculate PR normalized to Ligra.

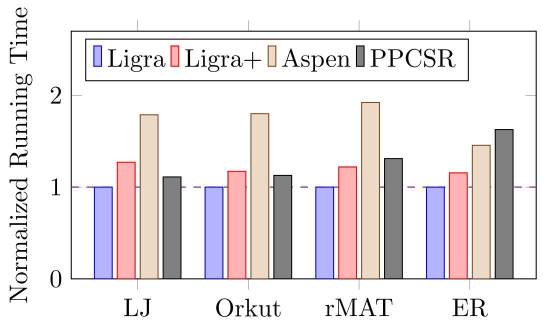

Figure 10: Time for all systems to calculate BFS normalized to Ligra.

Breadth-first search. Figure 10 illustrates the relative speed on BFS of all the systems. PPCSR is competitive (0.6 – 1.1x) with Aspen and is 1.1 – 1.6x slower than Ligra on BFS. We hypothesize that Aspen may experience extra work overheads due to compression. Furthermore, when the number of vertices in the BFS frontier is large, processing the frontier requires an efficient ordered scan through PPCSR.

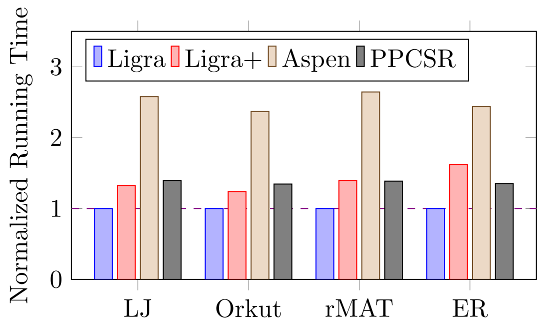

Figure 11: Time for all systems to calculate BC normalized to Ligra.

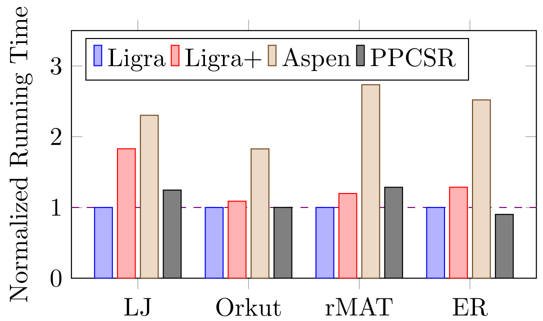

Figure 12: Time for all systems to calculate CC normalized to Ligra.

Betweenness centrality. Figure 11 illustrates the relative speed on BC of all the systems. On BC, PPCSR is about 2x faster than Aspen, and competitive (about 1.3x) with Ligra. Since BC is more computationally- and memory-intensive than BFS, it requires more passes through the structures. PPCSR and Ligra support efficient ordered passes.

Connected components. Figure 12 illustrates the relative speed on CC of all the systems. PPCSR exhibits about 2x speedup over Aspen and achieves similar performance with Ligra on CC. Since CC starts with all vertices in the frontier, it has more iterations with many vertices, which PPCSR can traverse efficiently.

7.3 Memory usage By design, PPCSR should use about 2x the space of an unoptimized CSR representation to store the empty spaces of the PMA. It can store the billion-edge ER graph in about 16 GB.

PPCSR uses between $1.3 - 2.3$x the space of Aspen, between $2 - 2.5$x the space of Ligra, and $2.3 - 3.2$x the space of Ligra+. We report the memory usage of all systems in Table 4. One reason for the space difference is that Aspen and Ligra+ use data compression techniques (e.g. delta compression), while PPCSR is uncompressed.

8 Related work

Many techniques for optimizing graph processing systems are independent of the underlying data structure.

Batching updates [19, 28, 33, 18] improves update throughput by amortizing the work of writing to the graph but may delay the time an update appears in the graph. If a graph processing system updates the graph only when it receives some number of updates, an update may have to wait for the batch to become sufficiently large.

Snapshotting [13, 27, 25, 26, 33] demonstrates the tradeoff between the overhead of taking snapshots and the freshness of the snapshot. More frequent snapshots are required for a more updated view of the graph, but taking snapshots requires extra processing. Snapshots can also convert a graph representation into a more traversal-friendly data structure [18]. Common traversal-based graph operations on dynamic graphs prefer the most up-to-date state of the graph [32].

Another technique is phasing the updates and queries separately [2, 10, 11, 19, 21, 22, 34, 41, 39, 40, 46, 48, 51]. This can improve the performance of queries since they do not need to worry about synchronization with insertions. However, all these approaches may delay queries. For example, a query may have to wait until after a snapshot is done or until a batch of edges was written to the graph.

9 Conclusions

Dynamic sparse graphs appear in applications from social networks to network routing and often see thousands of updates per second. We introduce Parallel Packed Compressed Sparse Row, a dynamic graph data structure which has parallel operations with polylogarithmic span and allows for concurrent updates and queries. In practice, PPCSR supports about 80 million updates per second while maintaining fast queries and traversals and performs updates much faster than its worst-case theoretical bounds would suggest. PPCSR is especially well-suited to graph traversals scan through all of the edges (e.g. PageRank).

Acknowledgments

We would like to thank Julian Shun and Laxman Dhulipala for their help in conducting the experimental evaluation. We would like to thank Tim Kaler, Tao Schardl, and Charles E. Leiserson for helpful comments and suggestions.

Research was sponsored by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the

	BFS								PR							
	PPCSR		Ligra		Ligra+		Aspen		PPCSR		Ligra		Ligra+		Aspen	
Graph	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}
LJ	0.57	0.025	0.74	0.022	1.14	0.028	1.60	0.040	4.78	0.19	4.25	0.11	6.26	0.16	10.03	0.21
Orkut	0.53	0.021	0.65	0.019	0.94	0.022	1.55	0.033	7.31	0.20	8.14	0.20	8.61	0.23	15.73	0.32
rMAT	0.65	0.048	1.36	0.037	1.95	0.045	3.32	0.071	34.64	1.05	38.21	0.90	45.70	1.05	95.63	1.81
ER	1.26	0.054	1.11	0.033	1.59	0.038	2.01	0.048	76.49	1.68	70.56	1.79	85.55	2.19	155.34	3.36
	BC								CC							
	PPCSR		Ligra		Ligra+		Aspen		PPCSR		Ligra		Ligra+		Aspen	
Graph	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}
LJ	2.59	0.09	2.26	0.07	3.54	0.09	7.72	0.17	2.65	0.07	1.97	0.06	3.12	0.10	5.60	0.13
Orkut	3.24	0.11	3.01	0.08	4.31	0.10	9.46	0.19	5.84	0.09	4.05	0.09	4.15	0.10	10.57	0.17
rMAT	7.58	0.25	7.36	0.18	12.01	0.25	25.88	0.48	11.93	0.41	14.10	0.32	18.20	0.38	45.08	0.86
ER	8.85	0.22	6.76	0.16	12.32	0.27	24.93	0.40	35.19	0.55	28.00	0.61	38.65	0.78	82.76	1.53

Table 3: Running times of PPCSR, Ligra, Ligra+, and Aspen on BFS, PR, BC, and CC. T_1 denotes the time on one thread, and T_{96} denotes the time on all (96) threads

Name	PPCSR	Ligra	Ligra+	Aspen
LJ	1.3	.34 (.66)	.23 (.55)	0.66 (.98)
Orkut	4.4	.91 (1.78)	.53 (1.4)	1.04 (1.91)
rMAT	8.79	2.13 (4.23)	1.51 (3.61)	3.93 (6.03)
ER	16.2	3.76 (7.49)	2.82 (6.55)	7.06 (9.16)

Table 4: Memory footprint (in GB) of the graphs on the different systems. PPCSR was run with weights, and all other systems were run without weights. To compare weighted and unweighted, we add the ideal $4 \times |E|$ (each weight is 4 bytes) to all structures which do not store weights, shown in parentheses.

U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] Amazon, *Amazon web services*. `https://aws.amazon.com/`, 2020.

[2] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, *Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows*, Proceedings of the VLDB Endowment, 11 (2018), pp. 691–704.

[3] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, *Rabbit order: Just-in-time parallel reordering for fast graph analysis*, in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 22–31.

[4] A.-L. Barabási and R. Albert, *Emergence of scaling in random networks*, Science, 286 (1999), pp. 509–512.

[5] 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.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, *Concurrent cache-oblivious b-trees*, in Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures, ACM, 2005, pp. 228–237.

[7] M. A. Bender and H. Hu, *An adaptive packed-memory array*, ACM Transactions on Database Systems (TODS), 32 (2007), p. 26.

[8] G. E. Blelloch, *Prefix sums and their applications*, tech. rep., Citeseer, 1990.

[9] P. Boldi and S. Vigna, *The webgraph framework i: compression techniques*, in Proceedings of the 13th international conference on World Wide Web, 2004, pp. 595–602.

[10] 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.

[11] Z. Cai, D. Logothetis, and G. Siganos, *Facilitating real-time graph mining*, in Proceedings of the fourth international workshop on Cloud data management, ACM, 2012, pp. 1–8.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-mat: A recursive model for graph mining*, in Proceedings of the 2004 SIAM International Conference on Data Mining, SIAM, 2004, pp. 442–446.

[13] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, *Kineograph: taking the pulse of a fast-changing and connected world*, in Proceedings of the 7th ACM european conference on Computer Systems, ACM, 2012, pp. 85–98.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT Press, 3 ed., 2009.

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

[16] L. Dhulipala, G. Blelloch, and J. Shun, *Julienne: A framework for parallel graph algorithms using work-efficient bucketing*, in Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, 2017, pp. 293–304.

[17] L. Dhulipala, G. E. Blelloch, and J. Shun, *Theoretically efficient parallel graph algorithms can be fast and scalable*, in Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, 2018, pp. 393–404.

[18] ———, *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.

[19] 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.

[20] P. Erdös and A. Rényi, *On random graphs i*, Publicationes Mathematicae Debrecen, 6 (1959), p. 290.

[21] G. Feng, X. Meng, and K. Ammar, *Distinger: A distributed graph data structure for massive dynamic graph processing*, in 2015 IEEE International Conference on Big Data (Big Data), IEEE, pp. 1814–1822.

[22] O. Green and D. A. Bader, *custinger: Supporting dynamic graph algorithms for gpus*, in 2016 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2016, pp. 1–6.

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

[24] A. Itai, A. G. Konheim, and M. Rodeh, *A sparse table implementation of priority queues*, in International Colloquium on Automata, Languages, and Programming, Springer, 1981, pp. 417–431.

[25] A. Iyer, L. E. Li, and I. Stoica, *Celliq: Real-time cellular network analytics at scale*, in 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), 2015, pp. 309–322.

[26] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, *Time-evolving graph processing at scale*, in Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, ACM, 2016, p. 5.

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

[28] A. Kyrola, G. E. Blelloch, and C. Guestrin, *Graphchi: Large-scale graph computation on just a pc*, USENIX, 2012.

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

[30] C. Lattner and V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004, p. 75.

[31] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection.* `http://snap.stanford.edu/data`, June 2014.

[32] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, *Challenges in parallel graph processing*, Parallel Processing Letters, 17 (2007), pp. 5–20.

[33] 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.

[34] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi, *Incremental, iterative data processing with timely dataflow*, Communications of the ACM, 59 (2016), pp. 75–83.

[35] @raffi, *New tweets per second record, and how!*, Aug 2013.

[36] Y. Saad, *Iterative methods for sparse linear systems*, SIAM, Philadelphia, 2nd ed ed., 2003.

[37] T. B. Schardl, W. S. Moses, and C. E. Leiserson, *Tapir: Embedding fork-join parallelism into llvm's intermediate representation*, in ACM SIGPLAN Notices, vol. 52, ACM, 2017, pp. 249–265.

[38] ———, *Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation*, ACM Transactions on Parallel Computing (TOPC), 6 (2019), pp. 1–33.

[39] D. Sengupta and S. L. Song, *Evograph: On-the-fly efficient mining of evolving graphs on gpu*, in International Supercomputing Conference, Springer, 2017, pp. 97–119.

[40] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, *Graphin: An online high performance incremental graph processing framework*, in European Conference on Parallel Processing, Springer, 2016, pp. 319–333.

[41] M. Sha, Y. Li, B. He, and K.-L. Tan, *Accelerating dynamic graph analytics on gpus*, Proceedings of the VLDB Endowment, 11 (2017), pp. 107–120.

[42] J. Shun and G. E. Blelloch, *Ligra: a lightweight graph processing framework for shared memory*, in Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2013, pp. 135–146.

[43] J. Shun, L. Dhulipala, and G. E. Blelloch, *Smaller and faster: Parallel processing of compressed graphs with ligra+*, in 2015 Data Compression Conference, IEEE, 2015, pp. 403–412.

[44] J. Shun, F. Roosta-Khorasani, K. Fountoulakis, and M. W. Mahoney, *Parallel local graph clustering*, Proceedings of the VLDB Endowment, 9 (2016).

[45] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, USA, 1997.

[46] T. Suzumura, S. Nishii, and M. Ganse, *Towards large-scale graph stream processing platform*, in Proceedings of the 23rd International Conference on World Wide Web, ACM, 2014, pp. 1321–1326.

[47] W. F. Tinney and J. W. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proceedings of the IEEE, 55 (1967), pp. 1801–1809.

[48] K. Vora, R. Gupta, and G. Xu, *Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations*, ACM SIGOPS Operating Systems Review, 51 (2017), pp. 237–251.

[49] H. Wei, J. X. Yu, C. Lu, and X. Lin, *Speedup graph processing by graph ordering*, in Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 1813–1828.

[50] B. Wheatman and H. Xu, *Packed compressed sparse row: A dynamic graph representation*, in 2018 IEEE Conference on High Performance Extreme Computing (HPEC), 2018.

[51] M. Winter, R. Zayer, and M. Steinberger, *Autonomous, independent management of dynamic graphs on gpus*, in 2017 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2017, pp. 1–7.

A Parallel PMA operations

We first specify a `count_non_nulls(s, t)` function that returns the number of non-nulls in each leaf in a region in the PMA defined by start and end indices s, t.

LEMMA A.1. *`count_non_nulls(s, t)` has work $O(t - s)$ and span $O(\log N)$.*

Proof. We can count the number of non-empty cells in each leaf in parallel using the parallel prefix as defined in Section 2. There are $t - s$ cells in the range, for work $O(t - s)$ and span $O(\log(t - s)) = O(\log N)$. □

The `double_pma` function doubles the number of cells in the PMA.

LEMMA A.2. *The `double_pma` procedure has work $O(N)$ and span $O(\log N)$.*

Proof. PMA doubling requires initializing a new PMA of size $2N$, copying over the old PMA into the new one, and redistributing in the new PMA. Initializing the new PMA of size $2N$ and copying over the old data has work $O(N)$ and span $O(\log N)$ since these operations take $O(1)$ work per cell. As shown in Theorem 4.1, redistribute also has work $O(N)$ and span $O(\log N)$. Therefore, `double_pma` has work $O(N)$ and span $O(\log N)$. □

Finally, the `halve_pma` function is the inverse of the `double_pma` function and requires initializing a new PMA of half the size and copying over the elements into the new PMA. It has the same asymptotic behavior as `double_pma` with $O(N)$ work and $O(\log N)$ span.

A.1 External operations The search function `search(lo, hi, v)` checks a sorted region of the PMA bounded by `lo, hi` (the beginning and end of the region, respectively) and returns the location of the smallest element that is at least v.

LEMMA A.3. *`search(lo, hi, v)` has $O(\log(hi - lo))$ work and span.*

Proof. The pseudocode for the `search` function can be found in Figure 13. We modify a traditional binary search to deal with null values. If the midpoint `pma[mid]` is null, we set the midpoint to the beginning of the next *PMA leaf* in $O(1)$ instructions. Since we enforce the *packed-left* property in PMA leaves, the beginning of each leaf is guaranteed to be non-null. Checking whether a cell is null and computing the beginning of the next leaf take constant time. Suppose that at some level of the binary search $\texttt{hi} - \texttt{lo} = \ell$. The maximum size of the next step is $\ell/2 + \log N$. If $\log N \approx \ell/2$, meaning that we do not decrease the size of the next binary search step by a constant fraction, then we can just look at all the cells serially with work and span $O(\log N)$. Otherwise, $\ell/2 + \log N = O(\ell/2)$ so we decrease the size of the search space by a constant fraction so we expect to take at most $\log N$ binary search steps. □

B Parallel graph operations

In this section, we show how to implement graph operations in PPCSR using the parallel PMA operations from Section 4. The read operations have logarithmic worst-case span and the write operations have polylogarithmic span.

B.1 Operations A graph storage format supports the following operations:

- `find_weight` returns the weight of an edge or 0 if it is not in the graph.
- `find_neighbors` returns the neighbors of a vertex.
- `add_edge` sets the weight of an edge if it is already in the graph, or adds the edge and its weight it if it is not yet in the graph.
- `delete_edge` removes an edge from the graph.
- `delete_vertex` removes a vertex from the graph.

B.2 Read Operations We can implement `find_weight(u,v)` directly with `search(lo, hi, v)` in the PMA. From Lemma A.3, `find_weight(u, v)` has $O(\log(deg(u))$ work and span.

Next, we describe the `find_neighbors` function, which finds the neighbors of a vertex in the graph. More formally, given a vertex $u \in V$, `find_neighbors(u)` returns a new set S_u of vertices such that for all $v \in V$, $v \in S_u$ if and only if $(u, v) \in E$. The pseudocode for `find_neighbors` can be found in Figure 14.

```
# returns the index of the first element
    with value at least v
def search(lo, hi, v):
  while (lo < hi):
    mid = (hi - lo) / 2
    if pma[mid] is null:
      # gets beginning of next leaf
      mid = ((mid / log(N)) + 1) * log(N)
      # do a linear scan of size O(log N)
      if mid > hi:
        for i in [lo, hi):
          if pma[i] >= v: return i
    # pma[mid] guaranteed to be non-null
    if pma[mid] is v: return mid
    elif pma[mid] > v: hi = mid
    else: lo = mid
  return lo
```

Figure 13: Pseudocode for `search(lo, hi, v)`.

```
def find_neighbors(u):
  start = vertices[u].start
  end = vertices[u].end
  counts[end - start]
  # end - start = O(deg(u))
  parallel_for i in [start, end):
    if edges[i] is not null:
      counts[i - start] = 1
    else:
      counts[i - start] = 0

  parallel_prefix_sum(counts)
  output[counts[end - start - 1]]
  parallel_for i in [start, end):
    if counts[i] > counts[i-1]:
      output[counts[i-1]] = edges[i]
  return output
```

Figure 14: Pseudocode for `find_neighbors` in PPCSR.

LEMMA B.1. *`find_neighbors(u)` has $O(deg(u))$ work and $O(\log(deg(u)))$ span.*

Proof. Each `parallel_for` loop that iterates over $O(deg(u))$ cells has work $O(deg(u))$ and span $O(\log(deg(u))$ because it iterates through $O(deg(u))$ cells in parallel. The `parallel_prefix_sum` on an array of length N can be implemented with span $O(\log N)$ [8]. The rest of the function takes $O(1)$ work. □

B.3 Write Operations We begin by describing `add_edge` and showing how to implement it with parallel PMA operations. `add_edge(u, v, w(u,v))` sets the value of the edge $(u, v) = w(u, v)$. If the edge $(u, v) \notin E$, `add_edge` adds it to the graph with weight $w(u, v)$.

THEOREM B.1. *`add_edge(u, v, w(u,v))` has amortized $O(\log^2(m + n))$ work, $O(\log^2(m + n))$ worst-case span, and $O(\log(m + n))$ amortized span.*

Proof. The `add_edge(u, v, w(u,v))` function updates the edge structure in PPCSR. First, we do a search to check if the edge already exists: if so, we update its weight. This takes $O(\log(deg(u))$ work and span by Lemma A.3.

Otherwise, we need to insert a new edge using `insert`. We modify `insert` to handle moving sentinels (in `slide_right` and `redistribute`). This modification takes $O(1)$ work per edge because it checks if each cell contains a sentinel and if so, modifies the pointer to that sentinel in the vertex array. The `insert` function takes amortized work and worst-case span $O(\log^2(m + n))$ by Theorem 4.2 and amortized span $O(\log(m + n))$ by Theorem 4.3. □

The `delete_edge` procedure is just the inverse of `add_edge` and has amortized $O(\log^2(m + n))$ work, $O(\log^2(m + n))$ worst-case span, and $O(\log(m + n))$ amortized span.

Next, we describe how to implement `add_vertex` with `add_edge`. The `add_vertex` function adds a new vertex with index n to a graph with n vertices and updates the edge structure with a sentinel.

LEMMA B.2. *`add_vertex` has amortized $O(\log^2(m+n))$ work, $O(\log^2(m+n))$ worst-case span, and $O(\log(m+n))$ amortized span.*

Proof. The `add_vertex` function updates both the vertex and the edge structure in PPCSR. First, `add_vertex` appends a new vertex to the end of the vertex array in amortized $O(1)$. If adding a new vertex triggers an $O(n)$ work copy, the copy has $O(\log n)$ span. We then insert the sentinel in the same way we inserted an edge using a call to `add_edge`. □

The `delete_vertex` function can be implemented with amortized $O(\log^2(m + n))$ work, $O(\log^2(m + n))$ worst-case span, and $O(\log(m + n))$ amortized span by keeping track of which vertices are deleted and rewriting the entire structure once half of them have been deleted.