Low-latency graph streaming using compressed purely-functional trees

As Published: 10.1145/3314221.3314598

Publisher: Association for Computing Machinery (ACM)

Persistent URL: https://hdl.handle.net/1721.1/136173

Version: Original manuscript: author's manuscript prior to formal peer review

Low-Latency Graph Streaming Using Compressed Purely-Functional Trees*

Laxman Dhulipala
CMU
ldhulipa@cs.cmu.edu

Guy E. Blelloch
CMU
guyb@cs.cmu.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract

Due to the dynamic nature of real-world graphs, there has been a growing interest in the graph-streaming setting where a continuous stream of graph updates is mixed with arbitrary graph queries. In principle, purely-functional trees are an ideal choice for this setting due as they enable safe parallelism, lightweight snapshots, and strict serializability for queries. However, directly using them for graph processing would lead to significant space overhead and poor cache locality.

This paper presents C-trees, a compressed purely-functional search tree data structure that significantly improves on the space usage and locality of purely-functional trees. The key idea is to use a chunking technique over trees in order to store multiple entries per tree-node. We design theoretically-efficient and practical algorithms for performing batch updates to C-trees, and also show that we can store massive dynamic real-world graphs using only a few bytes per edge, thereby achieving space usage close to that of the best static graph processing frameworks.

To study the efficiency and applicability of our data structure, we designed Aspen, a graph-streaming framework that extends the interface of Ligra with operations for updating graphs. We show that Aspen is faster than two state-of-the-art graph-streaming systems, Stinger and LLAMA, while requiring less memory, and is competitive in performance with the state-of-the-art static graph frameworks, Galois, GAP, and Ligra+. With Aspen, we are able to efficiently process the largest publicly-available graph with over two hundred billion edges in the graph-streaming setting using a single commodity multicore server with 1TB of memory.

*This is the full version of the paper appearing in the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), 2019.

1 Introduction

In recent years, there has been growing interest in programming frameworks for processing streaming graphs due to the fact that many real-world graphs change in real-time (e.g., [28, 29, 33, 21, 46, 82]). These graph-streaming systems receive a stream of queries and a stream of updates (e.g., edge and vertex insertions and deletions, as well as edge weight updates) and must process both updates and queries with low latency, both in terms of query processing time and the time it takes for updates to be reflected in new queries. There are several existing graph-streaming frameworks, such as STINGER, based on maintaining a single mutable copy of the graph in memory [28, 29, 33]. Unfortunately, these frameworks require either blocking queries or updates so that they are not concurrent, or giving up serializability [82]. Another approach is to use snapshots [21, 46]. Existing snapshot-based systems, however, are either very space-inefficient, or suffer from high latency on updates. Therefore, an important question is whether we can design a data structure that supports lightweight snapshots which can be used to concurrently process queries and updates, while ensuring that the data structure is safe for parallelism and achieves good asymptotic and empirical performance.

In principle, representing graphs using *purely-functional balanced search trees* [1, 56] can satisfy both criteria. Such a representation can use a search tree over the vertices (the vertex-tree), and for each vertex store a search tree of its incident edges (an edge-tree). Because the trees are purely-functional, acquiring an immutable snapshot is as simple as acquiring a pointer to the root of the vertex-tree. Updates can then happen concurrently without affecting the snapshot. In fact, any number of readers (queries) can concurrently acquire independent snapshots without being affected by a writer. A writer can make an individual or bulk update and then set the root to make the changes immediately and atomically visible to the next reader without affecting current active readers. A single update costs $O(\log n)$ work, and because the trees are purely-functional it is relatively easy and safe to parallelize a bulk update.

However, there are several challenges that arise when comparing purely-functional trees to compressed sparse row (CSR), the standard data structure for representing static graphs in shared-memory graph processing [63]. In CSR, the graph is stored as an array of vertices and an array of edges, where each vertex points to the start of its edges in the edge-array. Therefore, in the CSR format, accessing all edges incident to a vertex v takes $O(deg(v))$ work, instead of $O(\log n + deg(v))$ work for a graph represented using trees. Furthermore, the format requires only one pointer (or index) per vertex and edge, instead of a whole tree node. Additionally, as edges are stored contiguously, CSR has good cache locality when accessing the edges incident to a vertex, while tree nodes could be spread across memory. Finally, each set of edges can be compressed internally using graph compression techniques [70], allowing massive graphs to be stored using just a few bytes per edge [25]. This approach cannot be used directly on trees. This would all seem to put a search tree representation at a severe disadvantage.

In this paper, we describe a compressed purely-functional tree data structure that we call a C-tree, which addresses the poor space usage and locality of purely-functional trees. The C-tree data structure allows us to take advantage of graph compression techniques, and thereby store very large graphs on a single machine. The key idea of a C-tree is to chunk the elements represented by the tree and store each chunk contiguously in an array. Because elements in a chunk are stored contiguously, the structure achieves good locality. By ensuring that each chunk is large enough, we significantly reduce the space used for tree nodes. Although the idea of chunking is intuitive, designing a chunking scheme which admits asymptotically-efficient algorithms for batch-updates and also performs well in practice is challenging. We note that our chunking scheme is independent of the underlying balancing scheme used, and works for any type of element. In the context of

graphs, because each chunk in a C-tree stores a sorted set of integers, we can compress by applying difference coding within each block and integer code the differences. We compare to some other chunking schemes, including B-trees [4] and ropes [2, 30, 15, 9] in Section 3.3.

To address the asymptotic complexity issue, we observe that for many graph algorithms the $O(\log n)$ work overhead to access vertices can be handled in one of two ways. The first is for global graph algorithms, which process all vertices and edges. In this case, we can afford to compute a *flat snapshot*, which is an array of pointers to the edge-tree for each vertex. We show how to create a flat snapshot using $O(n)$ work, $O(\log n)$ depth, and $O(n)$ space. A flat snapshot can be created concurrently with updates and other reads since it copies from the persistent functional representation. Once a flat snapshot is created, the work for accessing the edges for a vertex v is only $O(deg(v))$, as with CSR. The second case is for local graph algorithms, where we cannot afford to create a flat snapshot. In this setting, we note that many local algorithms examine all edges incident to a vertex after retrieving it. Furthermore, although real-world graphs are sparse, their average degree is often in the same range or larger than $\log n$. Therefore, the cost of accessing a vertex in the vertex-tree can be amortized against the cost of processing its incident edges.

To evaluate our ideas, we describe a new graph-streaming framework called *Aspen* that enables concurrent, low-latency processing of queries and updates on graphs with billions of vertices and hundreds of billions of edges, all on a relatively modest shared-memory machine equipped with 1TB of RAM. Our system is fully serializable and achieves high throughput and performance comparable to state-of-the-art static graph processing systems. Aspen extends the interface proposed by Ligra [69] with operations for updating the graph. As a result, all of the algorithms implemented using Ligra, including graph traversal algorithms, local graph algorithms [71], algorithms using bucketing [24], and others [25], can be run using Aspen with minor modifications. To make it easy to build upon or compare with our work in the future, we have made Aspen publicly-available at https://github.com/ldhulipala/aspen.

Compared to state-of-the-art graph-streaming frameworks, Aspen provides significant improvements both in memory usage (8.5–11.4x more memory-efficient than Stinger [28] and 1.9–3.3x more memory-efficient than LLAMA [46]), and algorithm performance (1.8–10.2x faster than Stinger and 2.8–15.1x faster than LLAMA). Aspen is also comparable to the fastest static graph processing frameworks, including GAP [6] (Aspen is 1.4x faster on average), Ligra+ [70] (Aspen is 1.4x slower on average), and Galois [55] (Aspen is 12x faster on average). Compared to Ligra+, which is one of the fastest static compressed graph representations, Aspen only requires between 1.8–2.3x more space.

Our experiments show that adding a continuous stream of edges while running queries does not affect query performance by more than 3%. Furthermore, the latency is well under a millisecond, and the update throughput ranges from 11K–78K updates per second when performing one update at a time to *105M–442M updates per second* when performing batches of updates. We show that our update rates are an order of magnitude faster than the update rates achievable by Stinger, even when using very small batches.

The contributions of this paper are as follows:

(1) A practical compressed purely-functional data structure for search trees, called the C-tree, with operations that have strong theoretical bounds on work and depth.
(2) The approach of flat-snapshotting for C-trees to reduce the cost of random access to the vertices of a graph.
(3) Aspen, a multicore graph-streaming framework built using C-trees that enables concurrent, low-latency processing of queries and updates, along with several algorithms using the framework.
(4) An experimental evaluation of Aspen in a variety of regimes over graph datasets at different scales, including the largest publicly-available graphs (graphs with billions of vertices and hundreds

of billions of edges), showing significant improvements over state-of-the-art graph-streaming frameworks, and modest overhead over static graph processing frameworks.

2 Preliminaries

Notation and Primitives. We denote a graph by $G(V, E)$, where V is the set of vertices and E is the set of edges in the graph. For weighted graphs, the edges store real-valued weights. The number of vertices in a graph is $n = |V|$, and the number of edges is $m = |E|$. Vertices are assumed to be indexed from 0 to $n - 1$. For undirected graphs, we use $N(v)$ to denote the neighbors of vertex v and $deg(v)$ to denote its degree. We assume that we have access to a family of uniformly (purely) random hash functions which we can draw from in $O(1)$ work [22, 57]. In functions from such a family, each key is mapped to an element in the range with equal probability, independent of the values that other keys hash to, and the function can be evaluated for a given key in $O(1)$ work.

Work-Depth Model. We analyze algorithms in the work-depth model, where the ***work*** is the number of operations used by the algorithm and the ***depth*** is the length of the longest sequential dependence in the computation [38, 14].

Purely-Functional Trees. Purely-functional (mutation-free) data structures preserve previous versions of themselves when modified and yield a new structure reflecting the update [56]. The trees studied in this paper are binary search trees, which represent a set of ordered elements. In a purely-functional tree, each element is used as a *key*, and is stored in a separate tree node. The elements can be optionally associated with a value, which is stored in the node along with the key. Trees can also be augmented with an associative function f (e.g., $+$), allowing the sum with respect to f in a range of the tree be queried in $O(\log n)$ work and depth, where n is the number of elements in the tree.

Interfaces for Graphs. We will extend the interface defined by Ligra [69] and so we review its interface here. We use the ***vertexSubset*** data structure which represents subsets of vertices, and the EDGEMAP primitive which is used for mapping over edges incident to sets of vertices. EDGEMAP takes as input a subset of vertices and applies a function over the edges incident to the subset that satisfy a condition (e.g., edges to vertices that have not yet been visited by a breadth-first search). More precisely, **edgeMap** takes as input a graph $G(V, E)$, a vertexSubset U, and two boolean functions F and C; it applies F to $(u, v) \in E$ such that $u \in U$ and $C(v) = true$ (call this subset of edges E_a), and returns a vertexSubset U' where $u \in U'$ if and only if $(u, v) \in E_a$ and $F(u, v) = true$.

3 Compressed Purely-Functional Trees

In this section, we describe a compressed purely-functional search tree data structure which we refer to as a ***C-tree***. After describing the data structure in Section 3.1, we argue that our design improves locality and reduces space-usage relative to ordinary purely-functional trees (Section 3.2). Finally, we compare the C-tree data structure to other possible design choices, such as B-trees (Section 3.3).

3.1 C-tree Definition

The main idea of C-trees is to apply a chunking scheme over the tree to store multiple elements per tree-node. The chunking scheme takes the ordered set of elements to be represented and "promotes" certain elements to be heads, which are stored in a tree. The remaining elements are stored in tails associated with each tree node. To ensure that the same keys are promoted in different trees, a hash function is used to choose which elements are promoted. An important goal for C-trees is to maintain similar asymptotic cost bounds as for the uncompressed trees while improving space

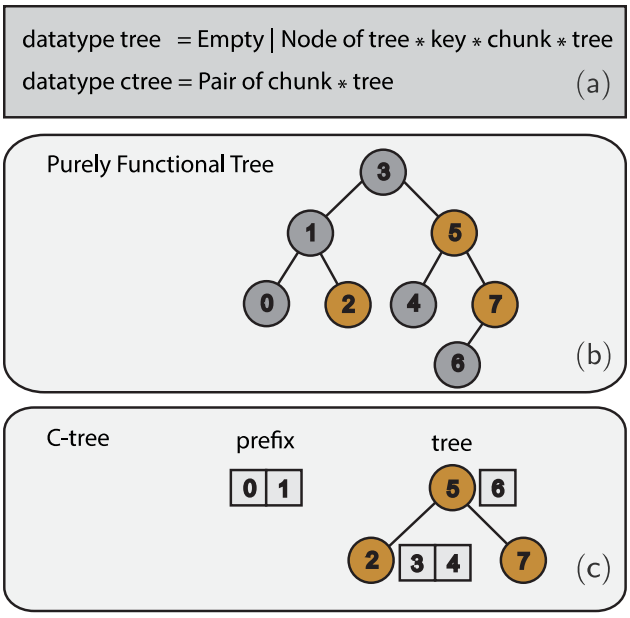

Figure 1: This figure gives the definition of the C-tree data structure in an ML-like language (subfigure (a)) and illustrates the difference between a purely-functional tree and a C-tree when representing a set of integers, S. Subfigure (b) shows a purely-functional tree where each element in S is stored in a separate tree node. We color the elements in S that are sampled as heads yellow, and color the non-head elements gray. Subfigure (c) illustrates how the C-tree stores S, given the heads. Notice that the C-tree has a chunk (the prefix) which contains non-head elements that are not associated with any head, and that each head stores a chunk (its tail) containing all non-head elements that follow it until the next head.

and cache performance, and to this end we describe theoretically efficient implementations of tree primitives in Section 4.

More formally. For an element type K, fix a hash function, $h : K \to \{1, \ldots N\}$, drawn from a uniformly random family of hash functions (N is some sufficiently large range). Let b be a *chunking parameter*, a constant which controls the granularity of the chunks. Given a set E of n elements, we first compute the set of ***heads*** $H(E) = \{e \in E \mid h(e) \mod b = 0\}$. For each $e \in H(E)$ let its ***tail*** be $t(e) = \{x \in E \mid e < x < next(H(E), e)\}$, where $next(H(e), e)$ returns the next element in $H(E)$ greater than e. We then construct a purely-functional tree with keys $e \in H(E)$ and associated values $t(e)$.

Thus far, we have described the construction of a tree over the head elements, and their tails. However, there may be a "tail" at the beginning of E that has no associated head, and is therefore not part of the tree. We refer to this chunk of elements as the ***prefix***. We refer to either a tail or prefix as a ***chunk***. We represent each chunk as a (variable-length) array of elements. As described later, when the elements are integers we can use difference encoding to compress each of the chunks. The overall ***C-tree*** data structure consists of the tree over head keys and tail values, and a single (possibly empty) prefix. Figure 1 illustrates the C-tree data structure over a set of integer elements.

Properties of C-trees. The expected size of chunks in a C-tree is b as each element is independently selected as a head under h with probability $1/b$. Furthermore, the chunks are unlikely to be much larger than b—in particular, a simple calculation shows that the chunks have size at most $O(b \log n)$ with high probability (w.h.p.),[1] where n is the number of elements in the tree. Notice that an element chosen to be a head will be a head in any C-trees containing it, a property that simplifies the implementation of primitives on C-trees.

[1] We use ***with high probability (w.h.p.)*** to mean with probability $1 - 1/n^c$ for some constant $c > 0$.

Our chunking scheme has the following bounds, which we prove in Appendix 10.2.

Lemma 3.1. *The number of heads (keys) in a C-tree over a set E of n elements is $O(n/b)$ w.h.p. Furthermore, the maximum size of a tail (the non-head nodes associated with a head) or prefix is $O(b \log n)$ w.h.p.*

We also obtain the following corollary.

Corollary 3.1.1. *When using a balanced binary tree for the heads (one with $O(\log n)$ height for n keys), the height of a C-tree over a sequence E of n elements is $O(\log(n/b))$ w.h.p.*

3.2 C-tree Compression

In this section, we first discuss the improved space usage of C-trees relative to purely-functional trees without any assumption on the underlying type of elements. We then discuss how we can further reduce the space usage of the data structure in the case where the elements are integers.

Space Usage and Locality. Consider the layout of a C-tree compared to a purely-functional tree. By Lemma 3.1, the expected number of heads is $O(n/b)$. Therefore, compared to a purely-functional tree, which allocates n tree nodes, we reduce the number of tree nodes allocated by a factor of b. As each tree node is quite large (in our implementation, each tree node is at least 32 bytes), reducing the number of nodes by a factor of b can significantly reduce the size of the tree. Experimental results are given in Section 7.1.

In a purely-functional tree, in the worst case, accessing each element will incur a cache miss, even in the case where elements are smaller than the size of a cache line. In a C-tree, however, by choosing b, the chunking parameter, to be slightly larger than the cache line size (≈ 128), we can store multiple elements contiguously within a single chunk and amortize the cost of a cache miss across all elements read from the chunk. Furthermore, note that the data structure can provide locality benefits even in the case when the size of an element is larger than the cache line size, as a modest value of b will ensure that reading all but the heads, which constitute an $O(1/b)$ fraction of the elements, will be contiguous loads from the chunks.

Integer C-trees. In the case where the elements are integers, the C-tree data structure can exploit the fact that elements are stored in sorted order in the chunks to further compress the data structure. We apply a *difference encoding* scheme to each chunk. Given a chunk containing d integers, $\{I_1, \ldots, I_d\}$, we compute the differences $\{I_1, I_2 - I_1, \ldots, I_d - I_{d-1}\}$. The differences are then encoded using a byte-code [70, 80]. We applied byte-codes due to the fact that they are fast to decode while achieving most of the memory savings that are possible using a shorter code [12, 80].

Note that in the common case when b is a constant, the size of each chunk is small ($O(\log n)$ w.h.p.). Therefore, despite the fact that each chunk must be processed sequentially, the cost of the sequential decoding does not affect the overall work or depth of parallel tree methods. For example, mapping over all elements in the C-tree, or finding a particular element have the same asymptotic work as purely-functional trees and optimal ($O(\log n)$) depth. To make the data structure dynamic, chunks must also be recompressed when updating a C-tree, which has a similar cost to decompressing the chunks. In the context of graph processing, the fact that methods over a C-tree are easily parallelizable and have low depth lets us avoid designing and implementing a more complex parallel decoding scheme, like the parallel byte-code in Ligra+ [70].

3.3 Other Approaches

Our data structure is loosely based on a previous sequential approach to chunking [11]. That approach was designed to be a generic addition to any existing balanced tree scheme for a dictionary and has overheads due to this goal.

Another option is to use B-trees [4]. However, the objective of a B-tree is to reduce the height of a search tree to accelerate searching a tree in external memory, whereas our goal is to build a data structure that stores many contiguous segments in a single node to make compression possible.

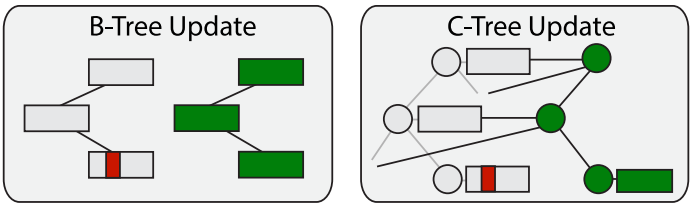

Figure 2: This figure shows the difference between performing a single update in a B-Tree versus an update in a C-tree. The data marked in green is newly allocated in the update. Observe that updating single element in a C-tree in the worst-case requires copying a path of nodes, and copying a single chunk if the element is not a head. Updating an element in a B-tree requires copying B pointers (potentially thousands of bytes) per level of the tree, which adds significant overhead in terms of memory and running time.

The problem with B-trees in our purely-functional setting is that we require path copying during functional updates, as illustrated in Figure 2. In our trees, this only requires copying a single binary node (32 or 40 bytes in our implementation) per level of the tree. For a B-tree, it would require copying B pointers (potentially thousands of bytes) per level of the tree, adding significant overhead in terms of memory and running time.

There is also work on chunking of functional trees for representing strings or (unordered) sequences [2, 30, 15, 9]. The motivation is similar (decrease space and increase locality), but the fact they are sequences rather than search trees makes the tradeoffs different. None of this work uses the idea of hashing or efficiently searching the trees. Using a hash function to select the heads has an important advantage in simplifying much of the code, and proving asymptotic bounds. Keeping the elements with internal nodes and using a prefix allows us to access the first b elements (or so) in constant work.

4 Operations on C-trees

In this section, we show how to support various tree operations over C-trees, such as building, searching and performing batch-updates to the data structure. These are operations that we will need for efficiently processing and updating graphs. We argue that the primitives are theoretically efficient by showing bounds on the work and depth of each operation. We also describe how to support augmentation in the data structure using an underlying augmented purely-functional tree. We note that the C-tree interfaces defined in this section operate over element-value pairs, whereas the C-trees defined in Section 3.1 only stored a set of elements for the sake of illustration. The algorithm descriptions elide the values associated with each element for the sake of clarity. We use operations on an underlying purely-functional tree data structure in our description, and state the bounds for operations on these trees as necessary (e.g., the trees described in Blelloch et al. [13] and Sun et al. [73]). The primitives in this section for a C-tree containing elements of type E and values of type V are defined as follows.

- ***Build***(S, f_V) takes a sequence of element-value pairs and returns a C-tree containing the elements in S with duplicate values combined using a function $f_V : V \times V \to V$.
- ***Find***(T, e) takes a C-tree T and an element e and returns the entry of the largest element $e' \leq e$.
- ***Map***(T, f) takes a C-tree T and a function $f : V \to ()$ and applies f to each element in T.
- ***MultiInsert***(T, f, S) and ***MultiDelete***(T, S) take a C-tree T, (possibly) a function $f : V \times V \to V$ that specifies how to combine values, and a sequence S of element-value pairs, and returns a C-tree containing the union or difference of T and S.

Our algorithms for Build, Find, and Map are straightforward, so due to space constraints, we give details about these implementations in Appendix 10.3.

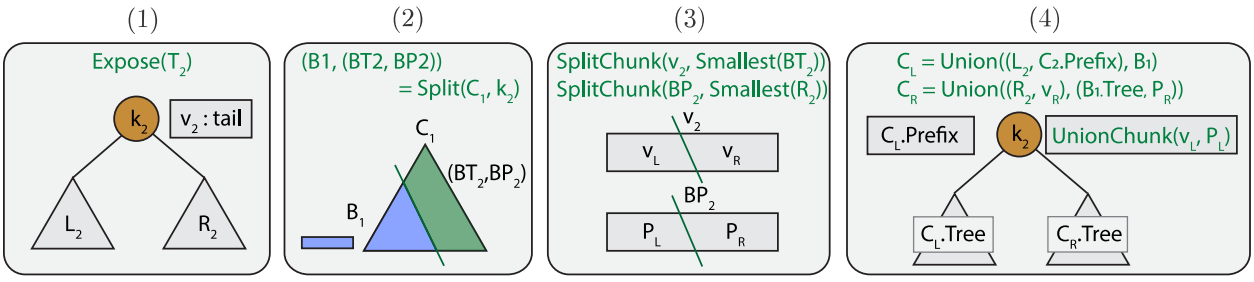

Figure 3: This figure illustrates how our UNION algorithm computes the union of two C-trees, T_1 and T_2. The text at the top of each figure (in green) denotes the sub-routine that is called, and the bottom portion of the figure illustrates the output of the call.

Algorithm 1 UNION

```
1:  function UNION(C1, C2)
2:    case (C1, C2) of
3:      ((null, _), _) → UNIONBC(C1, C2)
4:      (_, (null, _)) → UNIONBC(C2, C1)
5:      ((T1, P1), (T2, P2)) →
6:    let
7:      val (L2, k2, v2, R2) = EXPOSE(T2)
8:      val (B1, (BT2, BP2)) = SPLIT(C1, k2)
9:      val (vL, vR) = SPLITCHUNK(v2, SMALLEST(BT2))
10:     val (PL, PR) = SPLITCHUNK(BP2, SMALLEST(R2))
11:     val v2' = UNIONCHUNK(vL, PL)
12:     val (CL, CR) = UNION(B1, (L2, P2)) ||
                       UNION((BT2, PR), (R2, vR))
13:   in
14:     ctree(JOIN(CL.TREE, CR.TREE, k2, v2'), CL.PREFIX)
15:   end
```

4.1 Algorithms for Batch Insertions and Deletions

Our MULTIINSERT and MULTIDELETE algorithms are based on more fundamental algorithms for UNION, INTERSECTION, and DIFFERENCE on C-trees. Since we can simply build a tree over the input sequence to MULTIINSERT and call UNION (or DIFFERENCE for MULTIDELETE), we focus only on the set operations. Furthermore, because the algorithms for INTERSECTION and DIFFERENCE are conceptually very similar to the algorithm for UNION, we only describe in detail the UNION algorithm, and SPLIT, an important primitive used to implement UNION.

Union. Our UNION algorithm (Algorithm 1) is based on the recursive algorithm for UNION given by Blelloch et al. [13]. The main differences between the implementations are how to split a C-tree by a given element, and how to handle elements in the tails and prefixes. The algorithm takes as input two C-trees, C_1 and C_2, and returns a C-tree C containing the elements in the union of C_1 and C_2. Figure 3 provides an illustration of how our UNION algorithm computes the union of two C-trees. The algorithms use the following operations defined on C-trees and chunks. The **Expose** operation takes as input a tree and returns the left subtree, the element and prefix at the root of the tree, and the right subtree. The **Split** operation takes as input a C-tree B and an element k, and returns two C-trees B_1 and B_2, where B_1 (resp. B_2) are a C-tree containing all elements less than (resp. greater than) k. It can also optionally return a boolean indicating whether k was found in B, which is used when implementing DIFFERENCE and INTERSECTION. The **Smallest** operation returns the smallest head in a tree. The **UnionBC** algorithm merges a C-tree consisting of a prefix

and empty tree, and another C-tree. We also use the **SplitChunk** and **UnionChunk** operations, which are defined similarly to SPLIT and UNION for chunks.

The idea of the algorithm is to call EXPOSE on the tree of one of the two C-trees (C_2), and split the other C-tree (C_1) based on the element exposed at the root of C_2's tree (Line 7). The split on C_1 returns the trees B_1 and B_2 (Line 8). The algorithm then recursively calls UNION on the C-trees constructed from L_2 and R_2, the left and right subtrees exposed in C_2's tree with the C-trees returned by SPLIT, B_1, and B_2.

However, some care must be taken, since elements in k_2's tail, v_2, may come after some heads in B_2. Similarly, elements in B_2's prefix may come after some heads of R_2. In both cases, we should merge these elements with their corresponding heads' tails. We handle these cases by splitting v_2 by the leftmost element of B_2 (producing v_L and v_R), and splitting B_2's prefix by the leftmost element of R_2 (producing P_L and P_R). The left recursive call to UNION just takes the C-trees B_1 and (L_2, P_2). The right recursive call takes the C-trees $(B_2.\text{TREE}, P_R)$, and (R_2, v_R). Note that all elements in the prefixes P_R and v_R are larger than the smallest head in B_2 and R_2. Therefore, the C-tree returned from the right recursive call has an empty prefix. The output of UNION is the C-tree formed by joining the left and right trees from the recursive calls, k_2, and the tail v_2' formed by unioning v_L and P_L, with the prefix from C_L.

UnionBC. Recall that the UNIONBC algorithm merges a C-tree consisting of a prefix and empty tree, and another C-tree. We give a detailed description and pseudocode of the algorithm in Appendix 10.3. The idea of UNIONBC is to split the prefix based on the leftmost element of P's tree into two pieces, P_L and P_R containing elements less than and greater than the leftmost element respectively. P_L is merged with P's prefix to generate P'. The elements in P_R find the heads they correspond to by searching the tree for the largest head that is smaller than them. We then construct a sequence of head-tail pairs by inserting each element in P_R into its corresponding elements tail. Finally, we generate a new tree, T', by performing a MULTIINSERT into C's tree with the updated head-tail pairs. The return value is the C-tree (T', P').

Split. SPLIT takes a C-tree, $C = (T, P)$, and an element k and returns a triple consisting of a C-tree of all elements less than k, whether the element was found, and a C-tree of all elements greater than k. We provide a high-level description of the algorithm here and defer the pseudocode and details to Appendix 10.3.

The algorithm works by enumerating cases for how the split key can split C. If k is less than the first element in P, then we return an empty C-tree, false, indicating that k was not found, and C as the right C-tree. Similarly, if k splits P (it lies between the first and last elements of P) then we split P, and return the list of elements less than the split key as the left C-tree, with the boolean and right tree handled similarly. Otherwise, if the above cases did not match, and the tree is null, then we return C as the left C-tree. The recursive cases are similar to how SPLIT is implemented in Blelloch et al. [13], except for the case where k splits the tail at the root of the tree. Another important detail is how we compute the first and last elements of a chunk. Instead of scanning the chunk, which will cause us to do work proportional to the sum of chunks on a root-to-leaf path in the tree, we store the first and last elements at the head of each chunk to perform this operation in $O(1)$ work and depth. This modification is important to show that SPLIT can be done in $O(b \log n)$ work and depth w.h.p. on a C-tree.

4.2 Work and Depth Bounds

Due to space constraints, we provide the details, correctness proofs, and analysis for our C-tree primitives in Appendix 10.3, and state the work and depth bounds below.

Building. Building (Build(S, f_V)) a C-tree can be done in $O(n \log n)$ work and $O(b \log n)$ depth w.h.p. for a sequence of length n.

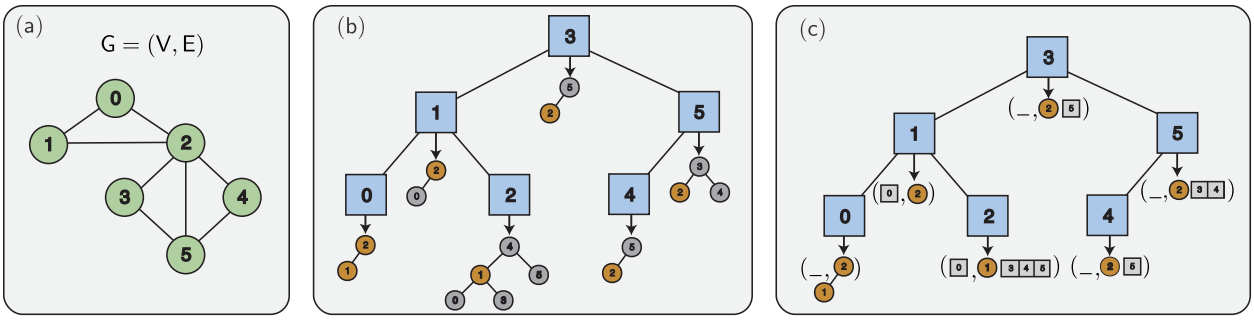

Figure 4: We illustrate how the graph (shown in subfigure (a)) is represented as a simple tree of trees (subfigure (b)) and as a tree of C-trees (subfigure (c)). As in Figure 1, we color elements (in this case vertex IDs) that are sampled as heads yellow. The prefix and tree in each C-tree are drawn as a tuple, following the datatype definition in Figure 1.

Searching. Searching ($\textsc{Find}(T, e)$) for an element e in a C-tree can be implemented in $O(b \log n)$ work and depth w.h.p., and $O(b + \log n)$ work and depth in expectation.

Mapping. Mapping ($\textsc{Map}(T, f)$) over a C-tree containing n elements with a constant-work function f can be done in $O(n)$ work and $O(b \log n)$ depth w.h.p.

Batch Updates. Batch updates ($\textsc{MultiInsert}(T, f, S)$ and $\textsc{MultiDelete}(T, S)$) can be performed in $O(b^2(k \log((n/k) + 1)))$ expected work and $O(b \log k \log n)$ depth w.h.p. where $k = \min(|T|, |S|)$ and $n = \max(|T|, |S|)$.

5 Representing Graphs as Trees

Representation. An undirected graph can be implemented using purely functional tree-based data structures by representing the set of vertices as a tree, which we call the ***vertex-tree***. Each vertex in the vertex-tree represents its adjacency information by storing a tree of identifiers of its adjacent neighbors, which we call the ***edge-tree***. Directed graphs can be represented in the same way by simply storing two edge-trees per vertex, one for the out-neighbors, and one for the in-neighbors. The resulting graph data structure is a tree-of-trees that has $O(\log n)$ overall depth using any balanced tree implementation (w.h.p. using a treap). Figure 4 illustrates the vertex-tree and the edge-trees for an example graph (subfigure (a)). Subfigure (b) illustrates how the graph is represented using simple trees for both the vertex-tree and edge-tree. Subfigure (c) illustrates using a simple tree for the vertex-tree and a C-tree for the edge-tree. We augment the vertex-tree to store the number of edges contained in its subtrees, which is needed to compute the total number of edges in the graph in $O(1)$ work. Weighted graphs can be represented using the same structure, with the only difference being that the elements in the edge-trees are modified to store an associated edge weight. Note that computing associative functions over the weights (e.g., aggregating the sum of all edge-weights) could be easily done by augmenting the edge and vertex-trees. We also note that the vertex-tree could also be compressed using a C-tree but defer evaluating this idea for future work.

Basic Graph Operations. We can compute the number of vertices and number of edges in the graph by querying the size (number of keys) in the vertex-tree and the augmented value of the vertex-tree respectively, which can both be done in $O(1)$ work. Finding a particular vertex just searches the vertex-tree, which takes $O(\log n)$ work and depth.

edgeMap. We implement $\textsc{edgeMap}$ (defined in Section 2) by mapping over the vertices in the input vertexSubset U in parallel and for each vertex $u \in U$ searching the vertex-tree for its edge-tree, and then mapping over u's incident neighbors, again in parallel. For each of u's neighbors v, we apply the map function $F(u, v)$ if the filter function $C(v)$ returns *true*. Other than finding vertices in the input vertexSubset in G and traversing edges via the tree instead of an array, the implementation

is effectively the same as in Ligra [69]. The direction optimization [5, 69] can also be implemented, and we describe more details later in this section. Assuming the functions F and C take constant work, EdgeMap takes $O(\sum_{u \in U} deg(u) + |U| \log n)$ work and $O(\log n)$ depth.

Batch Updates. Inserting and deleting edges are defined similarly, and so we only provide details for InsertEdges. Note that updates (e.g., to the weight) of existing edges can be done within this interface. Let A be the sequence (batch) of edge updates and let $k = |A|$.

We first sort the batch of edge pairs using a comparison sort. Next, we compute an array of source vertex IDs that are being updated and for each ID, in parallel, build a tree over its updated edges. We can combine duplicate updates in the batch by using the duplicate-combining function provided by the C-tree constructor. As the sequence is sorted, the build costs $O(k)$ work and $O(\log k)$ depth. Next, in the update step, we call MultiInsert over the vertex-tree with each $(source, tree)$ pair in the previous sequence. The combine function for MultiInsert combines existing values (edge-trees) with the new edge-trees by calling Union on the old edge-tree and new edge-tree.

We give a simple worst-case analysis of the algorithm and show that the algorithm performs $O(k \log n)$ work overall, and has $O(\log^3 n)$ depth. All steps before the MultiInsert cost $O(k \log k)$ work, and $O(\log k)$ depth in total, as they sort and apply parallel sequence operations to sequences of length k [38]. As the depth of both the vertex-tree and edge-tree is $O(\log n)$, the overall work of updating both the vertex-tree and each affected edge-tree can be upper bounded by $O(k \log n)$. The depth of MultiInsert is $O(\log n(\log m + D_{\text{Union}}))$, where $D_{\text{Union}})$ is the depth of union. This simplifies to $O(\log^3 n)$ by upper-bounding D_{Union} on any two trees as $O(\log^2 n)$, as shown in Appendix 10.3.

5.1 Efficiently Implementing Graph Algorithms

We now address how to efficiently implement graph algorithms using a tree of C-trees, mitigating the increase in access times due to using trees. We first describe a technique for handling the asymptotic increase in work for global graph algorithms due to using trees. We then consider local algorithms, and argue that for many local algorithms, the extra cost of searching the vertex-tree can be amortized. Finally, we describe how direction optimization [5] can be easily implemented over the C-tree data structure.

Flat Snapshots. Notice that algorithms in our framework that use edgeMap incur an extra $O(K \log n)$ factor in their work, where K is the total number of vertices accessed by edgeMap over the course of the algorithm. For an algorithm like breadth-first search, which runs in $O(m+n)$ work and $O(D \log n)$ depth for a graph with diameter D using a static-graph processing framework [25], a naive implementation using our framework will require performing $O(m + n \log n)$ work (the depth is the same, assuming that b is a constant).

Instead, for *global graph algorithms*, which we loosely define as performing $\Omega(n)$ work, we can afford to take a *flat snapshot* of the graph, which reduces the $O(K \log n)$ term to $O(K)$. The idea of a flat snapshot is very simple—instead of accessing vertices through the vertex-tree, and calling Find for each v supplied to edgeMap, we just precompute the pointers to the edge-trees for all $v \in V$ and store them in an array of size n. This can be done in linear work and $O(\log n)$ depth by traversing the vertex-tree once to fetch the pointers. By providing this array, which we call a ***flat snapshot*** to each call to edgeMap, we can directly access the edges tree in $O(1)$ work and reduce the work of edgeMap on a vertexSubset, U, to $O(\sum_{u \in U} deg(u) + |U|)$. In practice, using a flat snapshot speeds up BFS queries on our input graphs by an average of 1.26x (see Table 6).

Local Algorithms. In the case of local graph algorithms, we often cannot afford to create a flat snapshot without a significant increase in the work. We observe, however, that after retrieving a vertex many local algorithms will process all edges incident to it. Because the average degree

in real-world graphs is often in the same range or larger than $\log n$ (see Table 1), the logarithmic overhead of accessing a vertex in the vertex-tree in these graphs can be amortized against the cost of processing the edges incident to the vertex, on average.

Direction Optimization. Direction optimization is a technique first described for breadth-first search in Beamer et al. [5], and later generalized as part of Ligra in its EDGEMAP implementation [69]. It combines a sparse traversal, which applies the F function in EDGEMAP to the outgoing neighbors of the input vertexSubset U, with a dense traversal, which applies F to the incoming neighbors u of all vertices v in the graph where $C(v) = true$ and $u \in U$. The dense traversal improves locality for large input vertexSubsets, and reduces edge traversals in some algorithms, such as breadth-first search. The traversal mode on each iteration is selected based on the size of U and its out-degrees. We implemented the optimization by implementing a sparse traversal and a dense traversal that traverses the underlying C-trees.

6 Aspen Graph-Streaming Framework

In this section, we outline the Aspen interface and implementation for processing streaming graphs, and provide the full interface in Appendix 10.4. The Aspen interface is an extension of Ligra's interface. It includes the full Ligra interface—vertexSubsets, EDGEMAP, and various other functionality on a fixed graph. On top of Ligra, we add a set of functions for updating the graph—in particular, for inserting or deleting sets of edges or sets of vertices. We also add a flat-snapshot function. Aspen currently does not support weighted edges, but we plan to add this functionality using a similar compression scheme for weights as used in Ligra+ in the future. All of the functions for processing and updating the graph work on a *fixed and immutable version (snapshot)* of the graph. The updates are functional, and therefore instead of mutating the version, return a handle to a new graph. The implementation of these operations follow the description given in the previous sections.

The Aspen interface supports three functions, ACQUIRE, SET, and RELEASE, for acquiring the current version of a graph, setting a new version, and releasing a the version. The interface is based on the recently defined *version maintenance problem* and implemented with the corresponding lock-free algorithm to solve it [8]. RELEASE returns whether it is the last copy on that version, and if so we garbage collect it. The three functions each act atomically. The framework allows any number of concurrent readers (i.e., transactions that ACQUIRE and RELEASE but do not set) and a single writer (ACQUIREs, SETs, and then RELEASEs). Multiple concurrent readers can acquire the same version, or different versions depending on how the writer is interleaved with them. The implementation of this interface is non-trivial due to race conditions between the three operations. Importantly, however, no reader or writer is ever blocked or delayed by other readers or writers. The Aspen implementation guarantees strict serializability, which means that the state of the graph and outputs of queries are consistent with some serial execution of the updates and queries corresponding to real time.

Aspen is implemented in C++ and uses PAM [73] as the underlying purely-functional tree data structure for storing the heads. Our C-tree implementation requires about 1400 lines of C++, most of which are for implementing UNION, DIFFERENCE, and INTERSECT. Our graph data structure uses an augmented purely-functional tree from PAM to store the vertex-tree. Each node in the vertex tree stores an integer C-tree storing the edges incident to each vertex as its value. We note that the vertex-tree could also be compressed using a C-tree, but we did not explore this direction in the present work. To handle memory management, our implementations use a parallel reference counting garbage collector along with a custom pool-based memory allocator. The pool-allocation is critical for achieving good performance due to the large number of small memory allocations in the the functional setting. Although C++ might seem like an odd choice for implementing a functional

Graph	Num. Vertices	Num. Edges	Avg. Deg.
LiveJournal	4,847,571	85,702,474	17.8
com-Orkut	3,072,627	234,370,166	76.2
Twitter	41,652,231	2,405,026,092	57.7
ClueWeb	978,408,098	74,744,358,622	76.4
Hyperlink2014	1,724,573,718	124,141,874,032	72.0
Hyperlink2012	3,563,602,789	225,840,663,232	63.3

Table 1: Statistics about our input graphs.

Graph	Flat Snap.	Aspen Uncomp.	Aspen (No DE)	Aspen (DE)	Savings
LiveJournal	0.0722	2.77	0.748	0.582	4.75x
com-Orkut	0.0457	7.12	1.47	0.893	7.98x
Twitter	0.620	73.5	15.6	9.42	7.80x
ClueWeb	14.5	2271	468	200	11.3x
Hyperlink2014	25.6	3776	782	363	10.4x
Hyperlink2012	53.1	6889	1449	702	9.81x

Table 2: Statistics about the memory usage using different formats in Aspen. **Flat Snap.** shows the amount of memory in GBs required to represent a flat snapshot of the graph. **Aspen Uncomp.**, **Aspen (No DE)**, and **Aspen (DE)** show the amount of memory in GBs required to represent the graph using uncompressed trees, Aspen without difference encoding of chunks, and Aspen with difference encoding of chunks, respectively. **Savings** shows the factor of memory saved by using Aspen (DE) over the uncompressed representation.

interface, it allows us to easily integrate with PAM and Ligra. We also note that although our graph interface is purely-functional (immutable), our global and local graph algorithms are not. They can mutate local state within their transaction, but can only access the shared graph through an immutable interface.

7 Experiments

Algorithms. We implemented five algorithms in Aspen, consisting of three global algorithms and two local algorithms. Our global algorithms are breadth-first search (***BFS***), single-source betweenness centrality (***BC***), and maximal independent set (***MIS***). Our BC implementation computes the contributions to betweenness scores for shortest paths emanating from a single vertex. The algorithms are similar to the algorithms in [25] and required only minor changes to acquire a flat snapshot and include it as an argument to EDGEMAP. As argued in Section 5.1, the cost of creating the snapshot does not asymptotically affect the work or depth of our implementations. The work and depth of our implementations of BFS, BC, and MIS are identical to the implementations in [25]. Our local algorithms are 2-***hop*** and ***Local-Cluster***. 2-hop computes the set of vertices that are at most 2 hops away from the vertex using EDGEMAP. The worst-case work is $O(m + n \log n)$ and the depth is $O(\log n)$. Local-Cluster is a sequential implementation of the Nibble-Serial graph clustering algorithm (see [71, 72]), run using $\epsilon = 10^{-6}$ and $T = 10$.

In our experiments, we run the global queries one at a time due to their large memory usage and significant internal parallelism, and run the local queries concurrently (many at the same time).

Experimental Setup. Our experiments are performed on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4×2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use a work-stealing scheduler that we implemented. The scheduler is implemented similarly to Cilk for parallelism. Our programs are compiled with the `g++` compiler (version 7.3.0) with the `-O3` flag. All experiments involving balanced-binary trees use weight-balanced trees as the underlying balanced tree implementation [13, 73]. We use Aspen to refer to the system using C-trees and difference encoding within each chunk and explicitly specify other configurations of the system if necessary.

Application	LiveJournal			com-Orkut			Twitter		
	(1)	**(72h)**	**(SU)**	**(1)**	**(72h)**	**(SU)**	**(1)**	**(72h)**	**(SU)**
BFS	0.981	0.021	46.7	0.690	0.015	46.0	7.26	0.138	52.6
BC	4.66	0.075	62.1	4.58	0.078	58.7	81.2	1.18	68.8
MIS	3.38	0.054	62.5	4.19	0.069	60.7	71.5	0.99	72.2
2-hop	4.36e-3	1.06e-4	41.1	2.95e-3	6.82e-5	43.2	0.036	8.70e-4	41.3
Local-Cluster	0.075	1.64e-3	45.7	0.122	2.50e-3	48.8	0.127	2.59e-3	49.0

Table 3: Running times (in seconds) of our algorithms over symmetric graph inputs where **(1)** is the single threaded time **(72h)** is the 72-core time (with hyper-threading, i.e., 144 threads), and **(SU)** is the self-relative speedup.

Application	ClueWeb			Hyperlink2014			Hyperlink2012		
	(1)	**(72h)**	**(SU)**	**(1)**	**(72h)**	**(SU)**	**(1)**	**(72h)**	**(SU)**
BFS	186	3.69	50.4	362	6.19	58.4	1001	14.1	70.9
BC	1111	21.8	50.9	1725	24.5	70.4	4581	58.1	78.8
MIS	955	12.1	78.9	1622	22.2	73.0	3923	50.8	77.2
2-hop	0.883	0.021	42.0	1.61	0.038	42.3	3.24	0.0755	42.9
Local-Cluster	0.016	4.45e-4	35.9	0.022	6.75e-4	32.5	0.028	6.82e-4	41.0

Table 4: Running times (in seconds) of our algorithms over symmetric graph inputs where **(1)** is the single threaded time **(72h)** is the 72-core time (with hyper-threading, i.e., 144 threads), and **(SU)** is the self-relative speedup.

Graph Data. Table 1 lists the graphs we use. ***LiveJournal*** is a directed graph of the LiveJournal social network [16]. ***com-Orkut*** is an undirected graph of the Orkut social network. ***Twitter*** is a directed graph of the Twitter network, where edges represent the follower relationship [44]. ***ClueWeb*** is a Web graph from the Lemur project at CMU [16]. ***Hyperlink2012*** and ***Hyperlink2014*** are directed hyperlink graphs obtained from the WebDataCommons dataset where nodes represent web pages [50]. Hyperlink2012 is the *largest publicly-available graph*, and we show that *Aspen is able to process it on a single multicore machine*. We symmetrized the graphs in our experiments, as the running times for queries like BFS and BC are more consistent on undirected graphs due to the majority of vertices being in a single large component.

Overview of Results. We show the following experimental results in this section.

- The most memory-efficient representation of C-trees saves between 4–11x memory over using uncompressed trees, and improves performance by 2.5–2.8x compared to using uncompressed trees (Section 7.1).
- Algorithms implemented using Aspen are scalable, achieving between 32–78x speedup across inputs (Section 7.2).
- Updates and queries can be run concurrently in Aspen with only a slight increase in latency (Section 7.3).
- Parallel batch updates in Aspen are efficient, achieving between 105–442M updates/sec for large batches (Section 7.4).
- Aspen outperforms Stinger by 1.8–10.2x while using 8.5–11.4x less memory (Section 7.5).
- Aspen outperforms LLAMA by 2.8–7.8x while using 1.9–3.5x less memory (Section 7.6).
- Aspen is competitive with state-of-the-art static graph processing systems, ranging from being 1.4x slower to 30x faster (Section 7.7).

7.1 Chunking and Compression in Aspen

Memory Usage. Table 2 shows the amount of memory required to represent real-world graphs in Aspen without compression, using C-trees, and finally using C-trees with difference encoding. In the uncompressed representation, the size of a vertex-tree node is 48 bytes, and the size of an edge-tree

b (Exp. Chunk Size)	Memory	BFS (72h)	BC (72h)	MIS (72h)
2^1	68.83	0.309	2.72	2.17
2^2	41.72	0.245	2.09	1.71
2^3	26.0	0.217	1.68	1.41
2^4	17.7	0.172	1.45	1.24
2^5	13.3	0.162	1.32	1.14
2^6	11.1	0.152	1.25	1.07
2^7	9.97	0.142	1.22	1.01
2^8	9.42	**0.138**	**1.18**	0.99
2^9	9.17	0.141	1.20	0.99
2^{10}	9.03	0.152	1.19	0.98
2^{11}	8.96	0.163	1.20	0.98
2^{12}	**8.89**	0.170	1.21	**0.98**

Table 5: Memory usage (gigabytes) and performance (seconds) for the Twitter graph as a function of the (expected) chunk size. All times are measured on 72 cores using hyper-threading. Bold-text marks the best value in each column. We use 2^8 in the other experiments.

.

node is 32 bytes. On the other hand, in the compressed representation, the size of a vertex-tree node is 56 bytes (due to padding and extra pointers for the prefix) and the size of an edge-tree node is 48 bytes. We calculated the memory footprint of graphs that require more than 1TB of memory in the uncompressed format by hand, using the sizes of nodes in the uncompressed format.

We observe that by using C-trees and difference encoding to represent the edge trees, we reduce the memory footprint of the dynamic graph representation by 4.7–11.3x compared to the uncompressed format. Using difference encoding provides between 1.2–2.3x reduction in memory usage compared to storing the chunks in an uncompressed format. We observe that both using C-trees and compressing within the chunks is crucial for storing and processing our largest graphs in a reasonable amount of memory.

Comparison with Uncompressed Trees. Next, we study the performance improvement gained by the improved locality of the C-tree data structure. Due to the memory overheads of representing large graphs using the uncompressed format (see Table 2), we are only able to report results for our three smallest graphs, LiveJournal, com-Orkut, and Twitter, as we cannot store the larger graphs even with 1TB of RAM in the uncompressed format. We ran BFS on both the uncompressed and C-tree formats (using difference encoding) and show the results in the Appendix (Table 13). The results show that using the compressed representation improves the running times of these applications from between 2.5–2.8x across these graphs.

Choice of Chunk Size. Next, we consider how Aspen performs as a function of the expected chunk size, b. Table 5 reports the amount of memory used, and the BFS, BC, and MIS running times as a function of b. In the rest of the paper, we fixed $b = 2^8$, which we found gave the best tradeoff between the amount of memory consumed (it requires 5% more memory than the most memory-efficient configuration) while enabling good parallelism across different applications.

7.2 Parallel Scalability of Aspen

Algorithm Performance. Tables 3 and 4 report experimental results including the single-threaded time and 72-core time (with hyper-threading) for Aspen using compressed C-trees. For BFS, we achieve between 46–70x speedup across all inputs. For BC, our implementations achieve between 50–78x speedup across all inputs. Finally, for MIS, our implementations achieve between 60x–78x speedup across all inputs. We observe that the experiments in [25] report similar speedups for the same graphs. For local algorithms, we report the average running time for performing 2048 queries sequentially and in parallel. We achieve between 41–43x speedup for 2-hop, and between 35–49x speedup for Local-Cluster.

Graph	Without FS	With FS	Speedup	FS Time
LiveJournal	0.028	0.021	1.33	3.8e-3
com-Orkut	0.018	0.015	1.12	2.3e-3
Twitter	0.184	0.138	1.33	0.034
ClueWeb	4.98	3.69	1.34	0.779
Hyperlink2014	7.51	6.19	1.21	1.45
Hyperlink2012	18.3	14.1	1.29	3.03

Table 6: 72-core with hyper-threading running times (in seconds) comparing the performance of BFS without flat snapshots (**Without FS**) and with flat snapshots (**With FS**), as well as the running time for computing the flat snapshot (**FS Time**).

Graph	Update		Query (BFS)	
	Edges/sec	Latency	Latency (C)	Latency (I)
LiveJournal	7.86e4	1.27e-5	0.0190	0.0185
com-Orkut	6.02e4	1.66e-5	0.0179	0.0176
Twitter	4.44e4	1.73e-5	0.155	0.155
ClueWeb	2.06e4	4.83e-5	4.83	4.82
Hyperlink2014	1.42e4	7.04e-5	6.17	6.15
Hyperlink2012	1.16e4	8.57e-5	15.8	15.5

Table 7: Throughput and average latency achieved by Aspen when concurrently processing a sequential stream of edge updates along with a sequential stream of breadth-first search queries (each BFS is internally parallel). **Latency (C)** reports the average latency of the query when running the updates and queries concurrently, while **Latency (I)** reports the average latency when running queries in isolation on the modified graph.

Flat Snapshots. Table 6 shows the running times of BFS with and without the use of a flat snapshot. Our BFS implementation is between 1.12–1.34x faster using a flat snapshot, including the time to compute a flat snapshot. The table also reports the time to acquire a flat snapshot, which is between 15–24% of the overall BFS time across all graphs. We observe that acquiring a flat snapshot is already an improvement for a single run of an algorithm, and quickly becomes more profitable as multiple algorithms are run over a single snapshot of the graph (e.g., multiple BFS's or betweenness centrality computations).

7.3 Simultaneous Updates and Queries

In this sub-section, we experimentally verify that Aspen can support low-latency queries and updates running concurrently. In these experiments, we generate an update stream by randomly sampling 2 million edges from the input graph to use as updates. We sub-sample 90% of the sample to use as edge insertions, and immediately delete them from the input graph. The remaining 10% are kept in the graph, as we will delete them over the course of the update stream. The update stream is a random permutation of these insertions and deletions. We believe that sampling edges from the input graph better preserves the properties of the graph and ensures that edge deletions perform non-trivial work, compared to using random edge updates.

After constructing the update stream, we spawn two parallel jobs, one which performs the updates sequentially and one which performs global queries. We maintain the undirectedness of the graph by inserting each edge as two directed edge updates, within a single batch. For global queries, we run a stream of BFS's from random sources one after the other and measure the average latency. We note that for the BFS queries, as our inputs are symmetrized, a random vertex is likely to fall in the giant connected component which exists in all of our input graphs. The global queries therefore process nearly all of the vertices and edges.

Table 7 shows the throughput in terms of directed edge updates per second, the average latency to make an undirected edge visible, and the latency of global queries both when running concurrently with updates and when running in isolation. We note that when running global queries in isolation,

Graph	Batch Size					
	10	10^3	10^5	10^7	10^9	$2 \cdot 10^9$
LiveJournal	8.26e4	2.88e6	2.29e7	1.56e8	4.13e8	4.31e8
com-Orkut	7.14e4	2.79e6	2.22e7	1.51e8	4.21e8	4.42e8
Twitter	6.32e4	2.63e6	1.23e7	5.68e7	3.04e8	3.15e8
ClueWeb	6.57e4	2.38e6	7.19e6	2.64e7	1.33e8	1.69e8
Hyperlink2014	6.17e4	2.12e6	6.66e6	2.28e7	9.90e7	1.39e8
Hyperlink2012	6.45e4	2.04e6	4.97e6	1.84e7	8.26e7	1.05e8

Table 8: Throughput (directed edges/second) obtained when performing parallel batch edge insertions on different graphs with varying batch sizes, where inserted edges are sampled from an rMAT graph generator. We note that the times for batch deletions are similar to the time for insertions. All times are on 72 cores with hyper-threading.

we use all of the threads in the system (72-cores with hyper-threading). We observe that our data structure achieves between 22–157 thousand directed edge updates per second, which is achieved while concurrently running a parallel query on all remaining threads. We obtain higher update rates on smaller graphs, where the small size of the graph enables it to utilize the caches better. In all cases, the average latency for making an edge visible is at most 86 microseconds, and is as low as 12.7 microseconds on the smallest graph.

The last two columns in Table 7 show the average latency of BFS queries from random sources when running queries concurrently with updates, and when running queries in isolation. We see that the performance impact of running updates concurrently with queries is less than 3%, which could be due to having one fewer thread. We ran a similar experiment, where we ran updates on 1 core and ran multiple concurrent local queries (Local-Cluster) on the remaining cores, and found that the difference in average query times is even lower than for BFS.

7.4 Performance of Batch Updates

In this sub-section, we show that the batch versions of our primitives achieve high throughput when updating the graph, even on very large graphs and for very large batches. As there are insufficient edges on our smaller graphs for applying the methodology from Section 7.3, we sample directed edges from an rMAT generator [20] with $a = 0.5, b = c = 0.1, d = 0.3$ to perform the updates. To evaluate our performance on a batch of size B, we generate B directed edge updates from the stream (note that there can be duplicates), repeatedly call InsertEdges and DeleteEdges on the batch, and report the median of three such trials. The costs that we report *include* the time to sort the batch and combine duplicates.

Table 8 shows the throughput (the number of edges processed per second) of performing batch edge insertions in parallel on varying batch sizes. The throughput for edge deletions are within 10% of the edge insertion times, and are usually faster (see Figure 5). The running time can be calculated by dividing the batch size by the throughput. We illustrate the throughput obtained for both insertions and deletions in Figure 5 for the largest and smallest graph, and note that the lines for other graphs are sandwiched between these two lines. The only exception of com-Orkut, where batch insertions achieve about 2% higher throughput than soc-LiveJournal at the two largest batch sizes.

We observe that Aspen's throughput seems to vary depending on the graph size. We achieve a maximum throughput of 442M updates per second on com-Orkut when processing batches of 2B updates. On the other hand, on the Hyperlink2012 graph, the largest graph that we tested on, we achieve 105M updates per second for this batch size. We believe that the primary reason that small graphs achieve much better throughput at the largest batch size is that nearly all of the vertices in the tree are updated for the small graphs. In this case, due to the asymptotic work bound for the update algorithm, the work for our updates become essentially linear in the tree size.

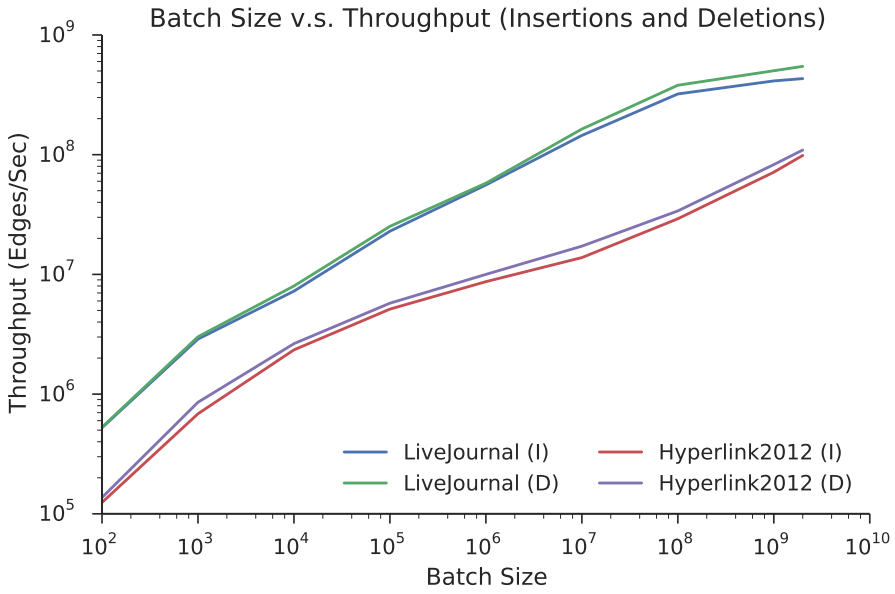

Figure 5: Throughput (edges/sec) when performing batches of insertions (I) and deletions (D) with varying batch sizes on Hyperlink2012 and LiveJournal in a log-log scale. All times are on 72 cores with hyper-threading.

7.5 Comparison with Stinger

In this sub-section, we compare Aspen to Stinger [28], a state-of-the-art graph-streaming system.

Stinger Design. Stinger's data structure for processing streaming graphs is based on adapting the CSR format to support dynamic updates. Instead of storing all edges of a vertex contiguously, it chunks the edges into a number of blocks, which are chained together as a linked list. Updates traverse the list to find an empty slot for a new edge, or to determine whether an edge exists. Therefore, updates take $O(deg(v))$ work and depth for a vertex v that is updated. Furthermore, updates use fine-grained locking to perform edge insertions, which may result in contention when updating very high degree vertices. As Stinger does not support compressed graph inputs, we were unable to run the system on our input graphs that are larger than Twitter.

Memory Usage. We list the sizes of the three graphs that Stinger was able to process in Table 9. The Stinger interface supports a function which returns the size of its in-memory representation in bytes, which is what we use to report the numbers in this paper.

We found that Stinger has a high memory usage, even in the memory-efficient settings used in our experiments. The memory usage we observed appears to be consistent with [28], which reports that the system requires 313GB of memory to store a scale-free (RMAT) graph with 268 million vertices and 2.15 billion edges, making the cost 145 bytes per edge. This number is on the same order of magnitude as the numbers we report in Table 9. We found that Aspen is between 8.5–11.4x more memory efficient than Stinger.

Batch Update Performance. We measure the batch update performance of Stinger by using an rMAT generator provided in Stinger to generate the directed updates. We set $n = 2^{30}$ for updates in the stream. The largest batch size supported by Stinger is 2M directed updates. The update times for Stinger were fastest when inserting into nearly-empty graphs. For each batch size, we insert 10 batches of edges of that size into the graph, and report the median time.

The results in Table 10 show the update rates for inserting directed edge updates in Stinger and Aspen. We observe that the running time for Stinger is reasonably high, even on very small batches, and grows linearly with the size of the batch. The Aspen update times also grow linearly,

Graph	ST	LL	Ligra+	Aspen	ST/Asp.	LL/Asp.	L+/Asp.
LiveJournal	4.98	1.12	0.246	0.582	8.55x	1.92x	0.422x
com-Orkut	10.2	3.13	0.497	0.893	11.4x	3.5x	0.55x
Twitter	81.8	31.4	5.1	9.42	8.6x	3.3x	0.54x
ClueWeb	–	–	100	200	–	–	0.50x
Hyperlink2014	–	–	184	363	–	–	0.50x
Hyperlink2012	–	–	351	702	–	–	0.50x

Table 9: The first four columns show the memory in gigabytes required to represent the graph using Stinger (**ST**), LLAMA (**LL**), Ligra+, and Aspen respectively. **ST/A**, **LL/A**, and **L+/A** is the amount of memory used by Stinger, LLAMA, and Ligra+ divided by the memory used by Aspen respectively. Stinger and LLAMA do not support compression and were not able to store the largest graphs used in our experiments.

Batch Size	Stinger	Updates/sec	Aspen	Updates/sec
10	0.0232	431	9.74e-5	102,669
10^2	0.0262	3,816	2.49e-4	401,606
10^3	0.0363	27,548	6.98e-4	1.43M
10^4	0.171	58,479	2.01e-3	4.97M
10^5	0.497	201,207	9.53e-3	10.4M
10^6	3.31	302,114	0.0226	44.2M
$2 \cdot 10^6$	6.27	318,979	0.0279	71.6M

Table 10: Running times and update rates (directed edges/second) for Stinger and Aspen when performing batch edge updates on an empty graph with varying batch sizes. Inserted edges are sampled from the RMAT graph generator. All times are on 72 cores with hyper-threading.

App.	Graph	ST	LL	A	A(1)	$A^\dagger$	ST/A	LL/A
BFS	LiveJournal	0.478	0.161	0.047	–	0.021	10.2	3.42
	com-Orkut	0.548	0.192	0.067	–	0.015	8.18	2.86
	Twitter	6.99	8.09	1.03	–	0.138	6.79	7.85
BC	LiveJournal	18.7	0.408	0.105	5.45	0.075	3.43	3.88
	com-Orkut	32.8	1.32	0.160	7.74	0.078	4.23	8.25
	Twitter	223	53.1	3.52	122	1.18	1.82	15.1

Table 11: Running times (in seconds) comparing the performance of algorithms implemented in Stinger (**ST**), LLAMA (**LL**), and Aspen. **A** is the parallel time using Aspen *without direction-optimization*. **(A(1))** is the one-thread time of Aspen, which is only relevant for comparing with Stinger's BC implementation. $\mathbf{A}^\dagger$ is the parallel time using Aspen *with direction-optimization*. **(ST/A)** is Aspen's speedup over Stinger and **(LL/A)** is Aspen's speedup over LLAMA.

but are very fast for small batches. Perhaps surprisingly, our update time on a batch of 1M updates is faster than the update time of Stinger on a batch of 10 edges.

Algorithm Performance. Lastly, we show the performance of graph algorithms implemented using the Stinger data structures. We use the BFS implementation for Stinger developed in McColl et al. [48]. We used a BC implementation that is available in the Stinger code base. Unfortunately, this implementation is entirely sequential, and so we compare Stinger's BC time to our single-threaded time. Neither of the Stinger implementations perform direction-optimization, so to perform a fair comparison, we used an implementation of BFS and BC in Aspen that disables direction-optimization. Table 11 shows the parallel running times of of Stinger and Aspen for these problems. For BFS, which is run in parallel, we achieve between 6.7–10.2x speedup over Stinger. For BC, which is run sequentially, we achieve between 1.8–4.2x speedup over Stinger. A likely reason that Aspen's BFS is significantly faster than Stinger's is that it can process edges incident to high-degree vertices in parallel, whereas traversing a vertex's neighbors in Stinger requires sequentially traversing a linked list of blocks.

7.6 Comparison with LLAMA

In this sub-section, we compare Aspen to LLAMA [46], another state-of-the-art graph-streaming system.

LLAMA Design. Like Stinger, LLAMA's streaming graph data structure is motivated by the CSR format. However, like Aspen, LLAMA is designed for batch-processing in the single-writer multi-reader setting and can provide serializable snapshots. In LLAMA, a batch of size k generates a new snapshot which uses $O(n)$ space to store a vertex array, and $O(k)$ space to store edge updates in a dynamic CSR structure. The structure creates a linked list over the edges incident to a vertex that is linked over multiple snapshots. This design can cause the depth of iterating over the neighbors of a vertex to be large if the edges are spread over multiple snapshots.

Unfortunately, the publicly-available code for LLAMA does not provide support for evaluating streaming graph algorithms or batch updates. However, we we were able to load static graphs and run several implementations of algorithms in LLAMA for which we report times in this section. As LLAMA does not support compressed graph inputs, we were unable to run the system on our input graphs that are larger than Twitter.

Memory Usage. Unfortunately, we were not able to get LLAMA's internal allocator to report correct memory usage statistics for its internal allocations. Instead, we measured the lifetime memory usage of the process and use this as an estimate for the size of the in-memory data structure built by LLAMA. The memory usage in bytes for the three graphs that LLAMA was able to process is shown in Table 9. The cost in terms of bytes/edge for LLAMA appears to be consistent, which matches the fact that the internal representation is a flat CSR, since there is a single snapshot. Overall, Aspen is between 1.9–3.5x more memory efficient than LLAMA.

Algorithm Performance. We measured the performance of a parallel breadth-first search (BFS) and single-source betweenness centrality (BC) algorithms in LLAMA. The same source is used for both LLAMA and Aspen for both BFS and BC. BFS and BC in LLAMA do not use direction-optimization, and so we report our times for these algorithms without using direction-optimization to ensure a fair comparison.

Table 11 shows the running times for BFS and BC. We achieve between 2.8–7.8x speedup over LLAMA for BFS and between 3.8–15.1x speedup over LLAMA for BC. LLAMA's poor performance on these graphs, especially Twitter, is likely due to sequentially exploring the out-edges of a vertex in the search, which is slow on graphs with high degrees.

7.7 Static Graph Processing Systems

We compared Aspen to Ligra+, a state-of-the-art shared-memory graph processing system, GAP, a state-of-the-art graph processing benchmark [6], and Galois, a shared-memory parallel programming library for C++ [55].

Ligra+. Table 12 the parallel running times of our three global algorithms expressed using Aspen and Ligra+. The results show that Ligra is 1.43x faster than Aspen for global algorithms on our small inputs. We also performed a more extensive experimental comparison between Aspen and Ligra+, comparing the parallel running times of all of our algorithms on all of our inputs (Tables 14 and 15). Compared to Ligra+, across all inputs, algorithms in Aspen are 1.51x slower on average (between 1.2x–1.7x) for the global algorithms, and 1.45x slower on average (between 1.0–2.1x) for the local algorithms. We report the local times in Tables 14 and 15. The local algorithms have a modest slowdown compared to their Ligra+ counterparts, due to logarithmic work vertex accesses being amortized against the relative high average degrees (see Table 1).

GAP. Table 12 shows the parallel running times of the BFS and BC implementations from GAP. On average, our implementations in Aspen are 1.4x faster than the implementations from GAP over

App.	Graph	GAP	Galois	Ligra+	Aspen	$\frac{\text{GAP}}{\text{A}}$	$\frac{\text{GAL}}{\text{A}}$	$\frac{\text{L+}}{\text{A}}$
BFS	LiveJ	0.0238	0.0761	0.015	0.021	1.1x	3.6x	0.71x
	Orkut	0.0180	0.0661	0.012	0.015	1.2x	4.4x	0.80x
	Twitter	0.139	0.461	0.081	0.138	1.0x	3.3x	0.58x
BC	LiveJ	0.0930	–	0.052	0.075	1.24x	–	0.69x
	Orkut	0.107	–	0.062	0.078	1.72x	–	0.79x
	Twitter	2.62	–	0.937	1.18	2.22x	–	0.79x
MIS	LiveJ	–	1.65	0.032	0.054	–	30x	0.59x
	Orkut	–	1.52	0.044	0.069	–	22x	0.63x
	Twitter	–	8.92	0.704	0.99	–	9.0x	0.71x

Table 12: Running times (in seconds) comparing the performance of algorithms implemented in GAP, Galois, Ligra+, and Aspen. $\frac{\text{GAP}}{\text{A}}$, $\frac{\text{GAL}}{\text{A}}$, and $\frac{\text{L+}}{\text{A}}$ are Aspen's speedups over GAP, Galois, and Ligra+ respectively.

all problems and graphs. We note that the code in GAP has been hand-optimized using OpenMP scheduling primitives. As a result, the GAP code is significantly more complex than our code, which only uses the high-level primitives defined by Ligra+.

Galois. Table 12 shows the running times of using Galois, a shared-memory parallel programming library that provides support for graph processing [55]. Galois' algorithms (e.g., for BFS and MIS) come with several versions. In our experiments, we tried all versions of their algorithms, and report times for the fastest one. On average, our implementations in Aspen are 12x faster than Galois. For BFS, Aspen is between 3.3–4.4x faster than Galois. We note that the Galois BFS implementation is synchronous, and does not appear to use Beamer's direction-optimization. We omit BC as we were not able to obtain reasonable numbers on our inputs using their publicly-available code (the numbers we obtained were much worse than the ones reported in [55]). For MIS, our implementations are between 9–30x faster than Galois.

8 Related Work

We have mentioned some other schemes for chunking in Section 3.3. Although we use functional trees to support snapshots, many other systems for supporting persistence and snapshots use version lists [7, 61, 26]. The idea is for each mutable value or pointer to keep a timestamped list of versions, and reading a structure to go through the list to find the right one (typically the most current is kept first). LLAMA [46] uses a variation of this idea. However, it seems challenging to achieve the low space that we achieve using such systems since the space for such a list is large.

8.1 Graph Processing Frameworks

Many processing frameworks have been designed to process static graphs (e.g. [23, 60, 58, 78, 47, 32, 45, 55, 69], among many others). We refer the reader to [49, 81] for surveys of existing frameworks. Similar to Ligra+ [70], Log(Graph) [10] supports running parallel algorithms on compressed graphs. Their experiments show that they have a moderate performance slowdown on real-world graphs, but sometimes get improved performance on synthetic graphs [10].

Existing dynamic graph streaming frameworks can be divided into two categories based on their approach to ingesting updates. The first category processes updates and queries in phases, i.e., updates wait for queries to finish before updating the graph, and queries wait for updates to finish before viewing the graph. Most existing systems take this approach, as it allows updates to mutate the underlying graph without worrying about the consistency of queries [28, 29, 33, 79, 3, 66, 65, 64, 53, 19, 77, 74, 68, 18]. Hornet [18], one of the most recent systems in this category, reports a throughput of up to 800 million edges per second on a GPU with 3,840 cores (about twice our throughput using 72 CPU cores for similarly-sized graphs); however the graphs used in Hornet are much smaller that what Aspen can handle due to memory limitations of GPUs. The second

category enables queries and updates to run concurrently by isolating queries to run on snapshots and periodically have updates generate new snapshots [21, 46, 37, 36].

GraphOne [43] is a system developed concurrently with our work that can handle queries running on the most recent version of the graph while updates are running concurrently by using a combination of an adjacency list and an edge list. They report an update rate of about 66.4 million edges per second on a Twitter graph with 2B edges using 28 cores; Aspen is able to ingest 94.5 million edges per second on a larger Twitter graph using 28 cores. However, GraphOne also backs up the update data to disk for durability.

There are also many systems that have been built for analyzing graphs over time [40, 34, 41, 51, 52, 35, 31, 62, 75, 76]. These systems are similar to processing dynamic graph streams in that updates to the graph must become visible to new queries, but are different in that queries can performed on the graph as it appeared at any point in time. Although we do not explore historical queries in this paper, functional data structures are particularly well-suited for this scenario since it is easy to keep any number of persistent versions simply by keeping their roots.

8.2 Graph Databases

There has been significant research on graph databases (e.g., [17, 39, 67, 59, 42, 27, 54]). The main difference between processing dynamic graph-streams and graph databases is that graph databases support transactions, i.e., multi-writer concurrency. A graph database running with snapshot isolation could be used to solve the same problem we solve. However, due to their need to support transactions, graph databases have significant overhead even for graph analytic queries such as PageRank and shortest paths. McColl et al. [48] show that Stinger is orders of magnitude faster than state-of-the-art graph databases.

9 Conclusion

We have presented a compressed fully-functional tree data structured called the C-tree that has theoretically-efficient operations, low space usage, and good cache locality. We use C-trees to represent graphs, and design a graph-streaming framework called Aspen that is able to support concurrent queries and updates to the graph with low latency. Experiments show that Aspen outperforms state-of-the-art graph-streaming frameworks, Stinger and LLAMA, and only incurs a modest overhead over state-of-the-art static graph processing frameworks. Future work includes designing incremental graph algorithms and historical queries using Aspen, and using C-trees in other applications. Although our original motivation for designing C-trees was for representing compressed graphs, we believe that they are of independent interest and can be used in applications where ordered sets of integers are dynamically maintained, such as compressed inverted indices in search engines.

Acknowledgements

This research was supported in part by NSF grants #CCF-1408940, #CF-1533858, #CCF-1629444, and #CCF-1845763, and DOE grant #DE-SC0018947. We thank the reviewers for their helpful comments.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[2] U. A. Acar, A. Charguéraud, and M. Rainey. Theory and practice of chunked sequences. In *European Symposium on Algorithms (ESA)*, pages 25–36, 2014.

[3] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, Feb. 2018.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Sep 1972.

[5] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 12:1–12:10, 2012.

[6] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[7] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[8] N. Ben-David, G. E. Blelloch, Y. Sun, and Y. Wei. Multiversion concurrency with bounded delay and precise garbage collection. 2019.

[9] J.-P. Bernardy. The haskell Yi package, 2008.

[10] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler. Log(graph): A near-optimal high-performance graph representation. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 7:1–7:13, 2018.

[11] D. K. Blandford and G. E. Blelloch. Compact representations of ordered sets. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–19, 2004.

[12] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 49–61, 2004.

[13] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2016.

[14] G. E. Blelloch and B. M. Maggs. Algorithms and theory of computation handbook. chapter Parallel Algorithms. Chapman & Hall/CRC, 2010.

[15] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: An alternative to strings. *Softw. Pract. Exper.*, 25(12), 1995.

[16] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *International World Wide Web Conference (WWW)*, pages 595–602, 2004.

[17] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

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

[19] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *International Workshop on Cloud Data Management (CloudDB)*, pages 1–8, 2012.

[20] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining (SDM)*, pages 442–446, 2004.

[21] 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 *European Conference on Computer Systems (EuroSys)*, pages 85–98, 2012.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[23] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 752–768, 2018.

[24] L. Dhulipala, G. E. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[25] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 393–404, 2018.

[26] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[27] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: a high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment*, 9(11):852–863, 2016.

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

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

[30] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in manticore. *J. Funct. Program.*, 20(5-6):537–576, Nov. 2010.

[31] F. Fouquet, T. Hartmann, S. Mosser, and M. Cordy. Enabling lock-free concurrent workers over temporal graphs composed of multiple time-series. In *ACM Symposium on Applied Computing (SAC)*, volume 8, pages 1054–1061, 2018.

[32] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USNIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

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

[34] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *European Conference on Computer Systems (EuroSys)*, pages 1:1–1:14, 2014.

[35] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. Le Traon. Analyzing complex data in motion at scale with temporal graphs. In *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 596–601, 2017.

[36] A. Iyer, L. E. Li, and I. Stoica. Celliq : Real-time cellular network analytics at scale. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 309–322, 2015.

[37] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *International Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 5:1–5:6, 2016.

[38] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[39] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. ZipG: A memory-efficient graph store for interactive queries. In *ACM SIGMOD International Conference on Management of Data*, pages 1149–1164, 2017.

[40] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *International Conference on Data Engineering (ICDE)*, pages 997–1008, 2013.

[41] U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. In *International Conference on Extending Database Technology (EDBT)*, pages 65–76, 2016.

[42] P. Kumar and H. H. Huang. G-Store: High-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 830–841, 2016.

[43] P. Kumar and H. H. Huang. GraphOne: A data store for real-time analytics on evolving graphs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 249–263, 2019.

[44] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*, pages 591–600, 2010.

[45] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010.

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

[47] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.

[48] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A performance evaluation of open source graph databases. In *Workshop on Parallel Programming for Analytics Applications*, pages 11–18, 2014.

[49] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, Oct. 2015.

[50] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science*, 1(1):33–47, 2015.

[51] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. ImmortalGraph: A system for storage and analysis of temporal graphs. *ACM TOS*, pages 14:1–14:34, 2015.

[52] O. Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[53] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, Sept. 2016.

[54] Neo4j.

[55] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.

[56] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[57] A. Pagh and R. Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, 2008.

[58] S. Pai and K. Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 2016.

[59] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX Conference on Annual Technical Conference (ATC)*, pages 41–52, 2012.

[60] D. Prountzos, R. Manevich, and K. Pingali. Synthesizing parallel graph programs via automated planning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 533–544, 2015.

[61] D. P. Reed. Naming and synchronization in a decentralized computer system, 1978.

[62] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

[63] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. SIAM, 2003.

[64] D. Sengupta and S. L. Song. EvoGraph: On-the-fly efficient mining of evolving graphs on GPU. In *International Supercomputing Conference (ISC)*, pages 97–119, 2017.

[65] 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 *Euro-Par*, pages 319–333, 2016.

[66] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on GPUs. *Proc. VLDB Endow.*, 11(1):107–120, Sept. 2017.

[67] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.

[68] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A system for real-time iterative analysis over evolving data. In *ACM SIGMOD International Conference on Management of Data*, pages 417–430, 2016.

[69] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 135–146, 2013.

[70] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *IEEE Data Compression Conference (DCC)*, pages 403–412, 2015.

[71] J. Shun, F. Roosta-Khorasani, K. Fountoulakis, and M. W. Mahoney. Parallel local graph clustering. *Proc. VLDB Endow.*, 9(12):1041–1052, Aug. 2016.

[72] D. A. Spielman and S.-H. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *ACM Symposium on Theory of Computing (STOC)*, pages 81–90, 2004.

[73] Y. Sun, D. Ferizovic, and G. E. Belloch. Pam: Parallel augmented maps. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 290–304, 2018.

[74] T. Suzumura, S. Nishii, and M. Ganse. Towards large-scale graph stream processing platform. In *International World Wide Web Conference (WWW)*, pages 1321–1326, 2014.

[75] M. Then, T. Kersten, S. Günnemann, A. Kemper, and T. Neumann. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *Proc. VLDB Endow.*, 10(8):877–888, Apr. 2017.

[76] K. Vora, R. Gupta, and G. Xu. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim.*, 13(4):32:1–32:27, Oct. 2016.

[77] K. Vora, R. Gupta, and G. Xu. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 237–251, 2017.

[78] K. Wang, G. H. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph query processing with abstraction refinement - scalable and programmable analytics over very large graphs on a single PC. In *USENIX Annual Technical Conference (ATC)*, pages 387–401, 2015.

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

[80] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes (2nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., 1999.

[81] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Found. Trends databases*, 7(1-2):1–195, Jan. 2017.

[82] C. Yin, J. Riedy, and D. A. Bader. A new algorithmic model for graph analysis of streaming data. In *International Workshop on Mining and Learning with Graphs*, 2018.

10 Appendix

10.1 Parallel Primitives

The following parallel procedures are used to describe our algorithms. ***Scan*** takes as input an array A of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus x = x$ for any x, and returns the array $(\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus_{i=0}^{n-2} A[i])$ as well as the overall sum, $\perp \oplus_{i=0}^{n-1} A[i]$. Scan can be done in $O(n)$ work and $O(\log n)$ depth (assuming $\oplus$ takes $O(1)$ work) [38]. ***Filter*** takes an array A and a predicate f and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order as in A. Filter can be done in $O(n)$ work and $O(\log n)$ depth (assuming f takes $O(1)$ work).

10.2 Details on C-tree Properties

We now provide the proof for Lemma 3.1.

Proof. Each element is selected as a head with probability $1/b$, and so by linearity of expectations, the expected number of heads is n/b. Define X_i to be the independent random variable that is 1 if E_i is a head and 0 otherwise. Let X be their sum, and $E[X] = n/b$. Applying a Chernoff bound proves that the number of heads is $O(n/b)$ w.h.p.

We now show that each tail is not too large w.h.p. Consider a subsequence of length $t = b \cdot (c \ln n)$ for a constant $c > 1$. The probability that none of the t elements in the subsequence are selected as a head is $(1 - 1/b)^t \leq (1/e)^{c \ln n} = 1/n^c$. Therefore, a subsequence of E of length t has a head w.h.p. We complete the proof by applying a union bound over all length t subsequences of E. □

10.3 Details on C-tree Primitives

This section provides details missing from the main body of the paper on how to build, search, and map over the C-tree data structure.

Building. Building (Build(S, f_V)) the data structure can be done in $O(n \log n)$ work and $O(b \log n)$ depth w.h.p. for a sequence of length n Given an unsorted sequence of elements, we first sort the sequence using a comparison sort which costs $O(n \log n)$ work and $O(\log n)$ depth [38]. Duplicate values in S can now be combined by applying a scan with f_V, propagating the sum with respect to f_V rightward, and keeping only the rightmost value in the resulting sequence using a filter.

Next, we hash each element to compute the set of heads and their indices, which can be done using a parallel map and filter in $O(n)$ work and $O(\log n)$ depth. Constructing the tails for each head can be done in $O(n)$ work and $O(b \log n)$ depth w.h.p. by mapping over all heads in parallel and sequentially scanning for the tail, and applying Lemma 3.1. The prefix is generated similarly. Finally, we build a purely-functional tree over the sequence of head and tail pairs, with the heads as the keys, and the tails as the values, which takes $O(n)$ work and $O(\log n)$ depth.

Searching. Searching (FIND(T, e)) for a given element e can be implemented in $O(b \log n)$ work and depth w.h.p and $O(b + \log n)$ work and depth in expectation. The idea is to simply search the keys in the C-tree for the first head $\leq e$. If the head e' that we find is equal to e we return TRUE, otherwise we check whether e lies in the tail associated with e' sequentially and return TRUE if and

Algorithm 2 UnionBC

```
1: function UnionBC(C1, C2)
2:   case (C1, C2) of
3:     ((null, null), _) → C2
4:   | ((_, P1), (T2, P2)) →
5:   let
6:     val (PL, PR) = SplitChunk(P1, Smallest(T2))
7:     val keys = map(λe.(FindHead(T2, e), e), PR)
8:     val ranges = UniqueKeyRanges(keys)
9:     val updates = map(λ(k, s, e).UnionRange(T2, k, s, e), ranges)
10:    val T2' = MultiInsert(updates, T2)
11:  in
12:    ctree(T2', UnionLists(PL, P2))
13:  end
```

only if e is in the tail. The depth of the tree is $O(\log n)$ and the size of the tail is $O(b \log n)$ w.h.p. ($O(b)$ in expectation) by Lemma 3.1, giving the bounds.

Mapping. Mapping (Map(T, f)) over a C-tree containing n elements with a constant-work function f can be done in $O(n)$ work and $O(b \log n)$ depth w.h.p. We simply apply a parallel map over the underlying purely-functional tree, which runs in $O(n)$ work and $O(\log n)$ depth [73]. The map operation for each node in the tree simply calls f on the key (a head), and then sequentially processes the tail, applying f to each element in it. We then apply f to each element in the prefix. The work is $O(n)$ as each element is processed once. As each chunk has size $O(b \log n)$ w.h.p. by Lemma 3.1, the overall depth is $O(b \log n)$ w.h.p.

Union Implementation. Algorithm 1 first checks whether Union is applicable by checking that both trees are present, and calls UnionBC, which computes the union of a prefix and a C-tree, if either tree is null (Lines 3–4). Next, the algorithms calls Expose on T_2 to bind k_2 and v_2 (the head and its tail) as the root of T_2, and L_2 and R_2 as T_2's left and right subtrees, respectively (Line 7). We then split C_1 based on k_2 (Line 8), which returns two C-trees, B_1 and (BT_2, BP_2) which contain all elements less than k_2 and all elements greater than k_2, respectively.

Notice that some elements in v_2, the tail from the root of T_2, may need to be sent to the recursive call involving B_2 as R_2's prefix if a head in B_2 has a value that comes before elements in v_2. To capture these elements that should join heads in B_2, we split v_2 based on the smallest element in B_2's tree (Line 9), binding v_L and v_R to the lists containing elements less than and greater than the smallest element, respectively. Similarly, some of the elements in BP_2 (non-head elements that are less than all elements in BT_2) may also be less than the heads in R_2, and should therefore be merged with k_2's new tail. We similarly split BP_2 based on the smallest element in R_2 to compute P_L and P_R (Line 10). The new tail for the root is computed on Line 11. Finally, on Line 12 we recursively call union in parallel to obtain the C-trees C_L and C_R respectively. Observe that C_R's prefix must be empty because we split the prefixes of the two C-trees participating in the right call to only contain elements larger than the smallest head in B_2 and the smallest head in R_2. Therefore, all elements in both prefixes of the right recursive call will ultimately end up joining some head's tail, implying that the prefix of C_R is empty. The tree in the output C-tree is obtained by calling the Join function for purely functional trees on k_2, v_2', C_L.Tree, and C_R.Tree, and the prefix is just the prefix from C_L (Line 14).

UnionBC. Algorithm 2 implements UnionBC, the base-case of Union, which computes the union of a prefix and a C-tree. If P_1 is null, we return C_2 (Line 3). Otherwise, P_1 is non-empty, and some of its elements may need to be unioned with P_2, while others may belong in tails in T_2. We split P_1 by the first key in T_2 (Line 6), returning the keys in P_1 less than (P_L) and greater than (P_R) the

first key in T_2. We first deal with P_R, which contains elements that should be sent to T_2. First, we find the head for each element in P_R in parallel by applying a map over the elements $e \in P_R$ (Line 7). Next, we compute the unique ranges for each key by calling UNIQUEKEYRANGES, which packs out the keys into a sequence of key, start index, and end index triples containing the index of the first and last element that found the key. This step can be implemented by a map followed by a scan operation to propagate the indices of boundary elements, and a pack (Line 8). Next, in parallel for each unique key, we call UNIONRANGE, which unions the elements sent to k with its current tail in T_2 and constructs *updates*, a sequence of head-tail pairs that are to be updated in T_2 (Line 9). Finally, we call MULTIINSERT with T_2 and *updates*, which returns the tree that we will output (Line 10). Note that the MULTIINSERT call here operates on the underlying purely-functional tree. We return a C-tree containing this tree, and the union of P_L and P_2 (Line 12). Using the fact that the expected size of P_1 is b, the overall work of UNIONBC is $O(b \log |C_2| + b \cdot b) = O(b^2 + b \log |C_2|)$ in expectation to perform the finds and merge the elements in P_1 with a corresponding tail. The depth is $O(\log b \log |C_2|)$ due to the MULTIINSERT.

Split Implementation. The SPLIT algorithm (Algorithm 3) takes a C-tree (C) and a split element (k), and returns a pair of C-trees where the first contains all elements less than the split element, and the second contains all elements larger than it. It first checks to see if C is empty, and returns two empty C-trees if so (Line 3).

Otherwise, if C has a tree but not a prefix (Line 4), the algorithm proceeds into the recursive case which splits a tree. It first exposes T (Line 6), binding h to the head at the root of the tree, v to the head's tail, and L and R to its left and right subtrees, respectively. The algorithm then compares k to the head, h. There are three cases. If k is equal to h (the EQ case on Line 9), the algorithm returns a C-tree constructed from L and a null prefix as the left C-tree, and (R, v) as the right C-tree, since all elements in v are strictly greater than h. Otherwise, if k is less than h (the LT case on Line 10), the algorithm recursively splits the C-tree formed by the left tree with a null prefix, binding L_L as the left C-tree from the recursive call, and (LT_R, LT_P) as the right tree and prefix from the recursive call. It returns L_L as the left C-tree. The right C-tree is formed by joining LT_R with the right subtree (R), with h and v as the head and prefix, and taking the prefix as LT_P. The last case, when k is greater than h (the GT case on Line 16) is more complicated since k can split v, h's tail. The algorithm checks if k splits v (the case $k \leq$ LARGEST(v) on Line 17), and if so calls SPLITLIST on v based on k (Line 19) to produce v_L and v_R. The algorithm returns a C-tree constructed from L joined with h, and v_L as h's tail as the left C-tree, and a C-tree containing R and v_R as the prefix as the right C-tree. Finally, if $k >$ LARGEST(v), the algorithm recursively splits R, which is handled similarly to the case where it splits L.

The last case is if C has a non-null prefix, P. In this case, the algorithm tries to split the prefix, and recurses on the tree if the prefix was unsuccessfully split. The algorithm first binds e_l and e_r to the smallest and largest elements in P. It then checks whether $k \leq e_r$. If so, then it splits P based on k to produce P_L and P_R, which contain elements less than and greater than k, respectively. It then returns a C-tree containing an empty tree and P_L as the left C-tree, and T and P_R as the right C-tree. Otherwise, P is not split, but the tree, T may be, and so the algorithm recursively splits T by supplying the C-tree (T, null) to SPLIT. Since T has an empty prefix, splitting T cannot output a left C-tree with a non-empty prefix. We return the recursive result, with P included as the left C-tree's prefix.

Work and Depth Bounds.

Theorem 10.1. SPLIT(T, k) *performs* $O(b \log n)$ *work and depth w.h.p. for a* C*-tree* T *with* n *elements. The result holds for all balancing schemes described in [13].*

Algorithm 3 SPLIT

```
1: function SPLIT(C, k)
2:   case C of
3:     (null, null) → (empty, false, empty)
4:   | (T, null) →
5:     let
6:       val (L, h, v, R) = EXPOSE(T)
7:     in
8:       case COMPARE(k, h) of
9:         EQ → (ctree(L, null), ctree(R, v))
10:      | LT →
11:          let
12:            val (L_L, (LT_R, LP_R)) = SPLIT((L, null), k)
13:          in
14:            (L_L, ctree(JOIN(LT_R, R, h, v), LP_R))
15:          end
16:      | GT →
17:          if (k ≤ LARGEST(v)) then
18:            let
19:              val (v_L, v_R) = SPLITLIST(v, k)
20:            in
21:              (ctree(JOIN(L, null, h, v_L), ctree(R, v_R)))
22:            end
23:          else
24:            let
25:              val ((RT_L, RP_L), R_R) = SPLIT((R, null), k)
26:            in
27:              (ctree(JOIN(L, RT_L, h, v), RP_L)
28:            end
29:          end
30:  | (T, P) →
31:      let
32:        val (e_l, e_r) = (SMALLEST(P), LARGEST(P))
33:      in
34:        if k ≤ e_r then
35:          let
36:            val (P_L, P_R) = SPLITCHUNK(P, k)
37:          in
38:            (ctree(null, P_L), (T, P_R))
39:          end
40:        else
41:          let
42:            val ((T_L, _), C_R) = SPLIT(T, null)
43:          in
44:            (ctree(T_L, P), C_R)
45:          end
46:        end
47:      end
```

Proof Sketch. As SPLIT is a sequential algorithm, the depth is equal to the work. We observe that the SPLIT algorithm performs $O(1)$ work at each internal node except in a case where the recursion stops due to the split element, k, lying between LEFTMOST(P) and RIGHTMOST(P) (line 6), or before RIGHTMOST(v) (line 15). Naively checking whether k lies before RIGHTMOST(v) for each tail, v, on a root-to-leaf path could make us perform $\omega(b \log n)$ work, but recall that we can store

RIGHTMOST(P) at the start of P to make the check run in $O(1)$ work. Therefore, the algorithm performs $O(1)$ work for each internal node.

If the C-tree is represented using a weight-balanced tree, AVL tree, red-black tree, or treap then its height will be $O(\log n)$ (w.h.p. for a treap). In the worst-case, the algorithm must recurse until a leaf, and split the tail at the leaf, which has size $O(b \log n)$ w.h.p. by Lemma 3.1. Therefore the work and depth of SPLIT is $O(b \log n)$ w.h.p. The correctness proof follows by induction and case analysis.

□

Theorem 10.2. *For two C-trees T_1 and T_2, the* UNION *algorithm runs in $O(b^2(k \log((n/k) + 1)))$ work in expectation and $O(b \log k \log n)$ depth w.h.p. where $k = \min(|T_1|, |T_2|)$ and $n = \max(|T_1|, |T_2|)$.*

Proof sketch. The extra work performed in our algorithm is due to splitting and unioning tails at each recursive call, and the work performed in UNIONBC. Using the fact that the expected size of each tail is $O(b)$ we can modify the proof of the work of UNION given in Theorem 6 in [13] to bound our work. In particular, we perform $O(b)$ work in expectation for each node with non-zero splitting cost which pays at least 1 unit of cost in the proof in [13]. To account for the work of the UNIONBC, observe that the dominant cost in the algorithm are the calls to FIND on Line 6. Also notice that calls operate on a tree, T, generated by a SPLIT from the parent of this call, and the work of this step is $O(b(b + \log |T|))$ in expectation. We can therefore bound this work by charging each call to UNIONBC to the SPLIT call that generated it and applying linearity of expectations over all calls to UNIONBC. As we already pay $O(b \log |T|)$ for the call to SPLIT in the proof from [13] the overall work is affected by an extra factor b^2, resulting in the stated work bound. Note that for $b = O(1)$ the work is affected by a constant factor in expectation.

To bound the depth, observe that the depth of the call-tree (including the depth of splits) can be bounded as $O(b \log n \log k)$ using the recurrence as Theorem 8 in [13]. Furthermore, the depth due to splitting tails in recursive calls of UNION is $O(b \log n)$ w.h.p. per level, which is the same as the depth due to a call to SPLIT, and does not therefore increase the depth. Finally, although UNIONBC can potentially have $O((\log b + \log \log n) \log k)$ depth due to the MULTIINSERT, UNIONBC only appears as a leaf in the call-tree, and so its contribution to the depth is additive. Thus the overall depth is $O(b \log n \log k)$ w.h.p.

□

Intersection and Difference. Lastly, for INTERSECTION and DIFFERENCE, we note that the main difference between UNION and INTERSECTION and DIFFERENCE is that they may require removing the split key (which is always maintained and joined with using JOIN in UNION). The only extra work is an implementation of JOIN2 over C-trees which is similar to JOIN except it does not take a key in the middle (see [13] for details on JOIN2).

10.4 Aspen Interface

We start by defining a few types used by the interface. A **versioned_graph** is a data type that represents multiple snapshots of an evolving graph. A **version** is a purely-functional snapshot of a versioned_graph. A **T seq** is a sequence of values of type **T**. Finally, a **vertex** is a purely-functional vertex contained in some version.

Building and Update Primitives. The main functions in our interface are a method to construct the initial graph, methods to acquire and release versions, and methods to modify a graph. The remaining functions in the interface are for traversing and analyzing versions and are similar to the Ligra interface. Aspen's functions are listed below:

BuildGraph(n : int, m : int,
S : int seq seq) : versioned_graph
Creates a versioned graph containing n vertices and m edges. The edges incident to the i'th vertex are given by $S[i]$.

acquire() : (VG : versioned_graph) : version
Returns a valid version of a versioned_graph VG. Note that this version will be persisted until the user calls RELEASE.

release() : (VG : versioned_graph, G : version)
Releases a version of a versioned_graph VG.

InsertEdges() : (VG : versioned_graph, E': int $\times$ int seq)
Updates the latest version of the graph, $G = (V, E)$, by inserting the edges in E' into G. Makes a new version of the graph equal to $G[E \cup E']$ visible to readers.

DeleteEdges() : (VG : versioned_graph, E': int $\times$ int seq)
Updates the latest version of the graph, $G = (V, E)$, by deleting the edges in E' from G. Singleton vertices (those with degree 0 in the new version of the graph) can be optionally removed. Makes a new version of the graph equal to $G[E \setminus E']$ visible to readers.

InsertVertices() : (VG : versioned_graph, V': int seq)
Updates the latest version of the graph, $G = (V, E)$, by inserting the vertices in V' into G. Makes a new version of the graph equal to $G[V \cup V']$ visible to readers.

DeleteVertices() : (VG : versioned_graph, V': int seq)
Updates the latest version of the graph, $G = (V, E)$, by deleting the vertices in V' from G. Makes a new version of the graph equal to $G[V \setminus V']$ visible to readers.

Our framework also supports similarly-defined primitives for updating values associated with edges (e.g., edge weights) and updating values associated with vertices (e.g., vertex weights). The interface is similar to the basic primitives for updating edges and vertices.

Access Primitives. The functions for accessing a graph are defined similarly to Ligra. For completeness, we list them below. We also provide primitives over the vertex object, such as DEGREE, MAP, and INTERSECTION.

NumVertices (NumEdges)() : (G : version) : int
Returns the number of vertices (edges) in the graph.

FindVertex() : (G : version, v: int) : {vertex $\cup \square$}
Returns either the vertex corresponding to the vertex identifier v, or $\square$ if v is not present in G.

edgeMap() : (G : version, U : vertexSubset, F: int $\times$ int $\rightarrow$ bool, C: int $\rightarrow$ bool) : vertexSubset
Given a vertexSubset U, returns a vertexSubset U' containing all v such that $(u, v) \in E$ for $u \in U$ and $C(v) =$ true and $F(u, v) =$ true.

10.5 Additional Experimental Results

Application	Graph	Aspen Uncomp.	Aspen	(S)
BFS	LiveJournal	0.055	0.021	2.6x
	com-Orkut	0.042	0.015	2.8x
	Twitter	0.348	0.138	2.5x

Table 13: Aspen Uncomp. is the parallel time using Aspen with uncompressed trees, and **Aspen** is the parallel time of Aspen with C-trees and difference encoding. **(S)** is the speedup obtained by Aspen over the uncompressed format. All times are measured on 72 cores using hyper-threading.

Application	LiveJournal			com-Orkut			Twitter		
	L	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$	**L**	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$	**L**	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$
BFS	0.015	0.021	1.40x	0.012	0.015	1.25x	0.081	0.138	1.70x
BC	0.052	0.075	1.44x	0.062	0.078	1.25x	0.937	1.18	1.25x
MIS	0.032	0.054	1.68x	0.044	0.069	1.56x	0.704	0.99	1.40x
2-hop	3.06e-4	3.45e-4	1.13x	2.12e-4	2.52e-4	1.18x	2.79e-3	7.79e-3	2.79x
Local-Cluster	0.031	0.058	1.87x	0.046	0.097	2.10x	0.037	0.094	2.54x

Table 14: Running times (in seconds) of our algorithms over small symmetric graph inputs on a 72-core machine (with hyper-threading) where **L** is the parallel time using Ligra+, **A** is the parallel time using Aspen, and $\frac{\mathbf{A}}{\mathbf{L}}$ is the slowdown incurred by Aspen. All times are measured using 72 cores using hyper-threading.

Application	ClueWeb			Hyperlink2014			Hyperlink2012		
	textbfL	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$	**L**	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$	**L**	**A**	$\frac{\mathbf{A}}{\mathbf{L}}$
BFS	1.68	3.69	2.19x	3.44	6.19	1.79x	8.48	14.1	1.66x
BC	14.7	21.8	1.48x	17.8	24.5	1.37x	37.1	58.1	1.56x
MIS	8.14	12.1	1.48x	14.2	22.2	1.56x	32.2	50.8	1.57x
2-hop	0.024	0.028	1.16x	0.036	0.038	1.05x	0.072	0.075	1.04x
Local-Cluster	0.013	0.020	1.53x	0.013	0.021	1.61x	0.016	0.024	1.50x

Table 15: Running times (in seconds) of our algorithms over large symmetric graph inputs on a 72-core machine (with hyper-threading) where **L** is the parallel time using Ligra+, **A** is the parallel time using Aspen, and $\frac{\mathbf{A}}{\mathbf{L}}$ is the slowdown incurred by Aspen. All times are measured using 72 cores using hyper-threading.