

Fully Dynamic Spectral Sparsification for Directed Hypergraphs

Sebastian Forster¹, Gramoz Goranci², and Ali Momeni³

¹Department of Computer Science, University of Salzburg, Austria

²Faculty of Computer Science, University of Vienna, Austria

³Faculty of Computer Science, UniVie Doctoral School Computer Science DoCS,
University of Vienna, Austria

Abstract

There has been a surge of interest in spectral hypergraph sparsification, a natural generalization of spectral sparsification for graphs. In this paper, we present a simple fully dynamic algorithm for maintaining spectral hypergraph sparsifiers of *directed* hypergraphs. Our algorithm achieves a near-optimal size of $O(n^2/\varepsilon^2 \log^7 m)$ and amortized update time of $O(r^2 \log^3 m)$, where n is the number of vertices, and m and r respectively upper bound the number of hyperedges and the rank of the hypergraph at any time.

We also extend our approach to the parallel batch-dynamic setting, where a batch of any k hyperedge insertions or deletions can be processed with $O(kr^2 \log^3 m)$ amortized work and $O(\log^2 m)$ depth. This constitutes the first spectral-based sparsification algorithm in this setting.

1 Introduction

Sparsification—the process of approximating a graph with another that has fewer edges while preserving a key property—is a central paradigm in the design of efficient graph algorithms. A fundamental property of interest is graph cuts, which are not only foundational in graph theory and serve as duals to flows, but also have widespread applications in areas such as graph clustering [LGT14; PSZ17; Fen18] and image segmentation [BV06; KT10], to mention a few. In their seminal work, Benczúr and Karger [BK96] initiated the study of sparsifiers in the context of graph cuts. They showed that any graph admits a sparse reweighted cut-sparsifier whilst paying a small loss in the approximation. Spielman and Teng [ST11] introduced a variant of sparsification known as spectral sparsification, which generalizes cut sparsification and measures graph similarity via the spectrum of their Laplacian matrices. The development of such sparsification techniques has had a profound impact across algorithm design [Mad10; She13; ST14; Pen16], with the Laplacian paradigm standing out as a key example [BHV08; SS08; DS08; KM09; KMT11].

Motivated by the need to capture complex interdependencies in real-world data, beyond the pairwise relationships modeled by traditional graphs, there has been a surge of interest in recent years in developing spectral sparsifiers for hypergraphs. These efforts have led to algorithmic constructions that achieve near-optimal size guarantees [KKY21a; KKY21b; OST23; JLS23; Lee23]. However, a common limitation of these algorithms is the assumption that the input graph is static—an assumption that does not hold in many practical settings. For example, real-world graphs, such as those modeling social networks, are inherently dynamic and undergo continual structural changes. For undirected hypergraphs, this limitation has been addressed in two recent independent works [GM25; KLP25], which show that spectral hypergraph sparsifiers can be maintained dynamically, supporting both hyperedge insertions and deletions in polylogarithmic time with respect to input parameters. This naturally leads to the question of whether *directed* hypergraphs also admit similarly efficient dynamic sparsification algorithms.

In this paper, we study problems at the intersection of dynamic graph-based data structures and spectral sparsification for directed hypergraphs. Specifically, we consider the setting where a directed hypergraph undergoes hyperedge insertions and deletions, and the goal is to efficiently process these updates while maintaining a spectral sparsifier that approximates the input hypergraph. Our work builds upon variants of two key algorithmic constructions: the static spectral sparsification framework for directed hypergraphs developed by Oko, Sakaue, and Tanigawa [OST23], and the dynamic sparsifier maintenance techniques for ordinary graphs by Abraham, Durfee, Koutis, Krinner, and Peng [ADKKP16]. Leveraging insights from both lines of work and modifying them to our setting, we design efficient dynamic algorithms for maintaining spectral sparsifiers of directed hypergraphs, as formalized in the theorem below.

Theorem 1.1. *Given a directed hypergraph $H = (V, E, \mathbf{w})$ with n vertices, rank r , and at most m hyperedges (at any time), there is a fully dynamic data structure that, with high probability, maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of H of size $O(n^2/\varepsilon^2 \log^7 m)$ in $O(r^2 \log^3 m)$ amortized update time.*

The guarantees provided by the above theorem are nearly tight, for the following reasons: (1) even reading the hyperedges in a hypergraph of rank r requires $\Theta(r)$ time, which means any update time must inherently depend on r , and (2) in the setting of directed graphs, it is folklore that a complete bipartite graph on n vertices, with all edges directed from one partition to the

other, gives an $\Omega(n^2)$ lower bound on the size of any directed sparsifier. Moreover, Oko, Sakaue, and Tanigawa [OST23] established an even stronger lower bound, showing that any spectral sparsifier must also incur a $\Omega(\epsilon^{-1})$ dependence. The latter implies that the size of our dynamic sparsifier is optimal up to a factor of ϵ^{-1} and polylogarithmic terms.

Our algorithmic construction also extends naturally to the closely related batch-dynamic setting. In this model—similar to the fully dynamic setting—updates consist of hyperedge insertions and deletions, but are processed in batches, enabling the exploitation of parallelism. This approach is particularly well-suited for handling high-throughput update streams and may more accurately reflect how dynamic changes are managed in real-time systems. Our result in this model is formalized in the theorem below.

Theorem 1.2. *Given a directed hypergraph $H = (V, E, \mathbf{w})$ with n vertices, rank r , and at most m hyperedges (at any time) undergoing batches of k hyperedge additions or deletions, there is a parallel fully dynamic data structure that, with high probability, maintains a $(1 \pm \epsilon)$ -spectral hypersparsifier \tilde{H} of H of size $O(n^2/\epsilon^2 \log^7 m)$ in $O(kr^2 \log^3 m)$ amortized work and $O(\log^2 m)$ depth.*

1.1 Related Work

We briefly discuss the related work on graphs and hypergraphs for several computational models.

Static Algorithms. Starting with [ST11], spectral sparsification has been extensively studied on graphs [BSS12; KP12; BSST13; ZLO15; KLP16; LS17; LS18]. Recently, the concept has been extended to undirected and directed hypergraphs [SY19; BST19; KKTY21a; KKTY21b; RY22; OST23; Lee23; JLS23; KPS24].

Dynamic Algorithms. For undirected graphs, there have been several results for dynamic cut sparsifiers that use a similar approach to ours. This includes [ADKKP16], which achieves polylogarithmic update time using t -bundle spanners, and its extension against an adaptive adversary by [BBGNSSS22] and to directed graphs [Zha25]. More recently, [KLP25] achieved a fully dynamic spectral sparsifier with near-optimal size and update time for undirected hypergraphs. More broadly, spectral-based sparsification is closely related to dynamic vertex sparsifiers [GHP18; DGGP19; CGHPS20; GLP21; AMV21; BGJLLPS22; DGGLPSY22].

Distributed and Parallel Algorithms. [KX16] achieved simple yet powerful algorithms for spectral graph sparsification in both parallel and distributed settings. Very recently, [GK25] developed a parallel batch-dynamic algorithm for spanners. Other related works include distributed vertex sparsifiers [ZLB21; FGLPSY21].

Online and Streaming Algorithms. For graphs, [KL13] extended the sampling scheme based on effective resistances [SS08] to the semi-streaming setting. Moreover, [CMP20] obtained an online spectral sparsification algorithm for graphs. Recently, [STY24] proposed an online algorithm for spectral hypergraph sparsification. For graphs in dynamic streams, [AGM13] developed a spectral sparsifier. Also, there has been a series of work on spectral sparsification in dynamic streams for both graphs and hypergraphs [KLMMS17; KNST19; KMMMNST20].

1.2 Technical Overview

In this section, we present the main ideas behind our fully dynamic algorithm for maintaining a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of a directed hypergraph H . Our algorithm builds on the static algorithm of [OST23], which we briefly review.

The algorithm of [OST23] constructs \tilde{H} by computing a sequence of hypergraphs H_1, \dots, H_k , where H_1 is a spectral hypersparsifier of H , H_2 is a spectral hypersparsifier of H_1 , and so on, until $H_k = \tilde{H}$. Each H_i is obtained via simple sampling scheme (discussed shortly) that guarantees, with high probability, H_i is proportionally smaller than H_{i-1} . As a result, the number of iterations is bounded by $k = O(\log m)$. In constructing H_i from H_{i-1} , the algorithm obtains two sub-hypergraphs of H_{i-1} : the *coreset* hypergraph C_i and the *sampled* hypergraph S_i .

At a high level, C_i consists of a sufficient number of heavy-weight hyperedges of H_{i-1} (to be specified shortly). This ensures that, when hyperedges are sampled uniformly at random from the remaining hypergraph $H_{i-1} \setminus C_i$ to construct S_i , the resulting union $H_i := C_i \cup S_i$ forms a spectral hypersparsifier of H_{i-1} . More precisely, C_i is constructed as follows. For every pair $(u, v) \in V \times V$, the algorithm selects the $O(\log^3 m / \varepsilon^2)$ heaviest hyperedges whose tail¹ contains u and whose head contains v .² These hyperedges are then added to C_i .

To construct S_i , each hyperedge in $H_{i-1} \setminus C_{i-1}$ is sampled independently with probability $1/2$, and its weight is doubled. They prove that, with high probability, this simple sampling scheme produces a $(1 \pm \varepsilon)$ -spectral hypersparsifier $H_i = C_i \cup S_i$ of H_{i-1} . Moreover, with high probability, $|E(H_i)| \leq 3|E(H_{i-1})|/4$ and so $k = O(\log m)$. See [Figure 1a](#) for an illustration of their approach.

Unfortunately, the static algorithm cannot be directly converted into an efficient dynamic algorithm. This is mostly due to the way the hypergraphs in the sequence H_1, \dots, H_k are built on top of each other, as a single change in H can propagate into $O(m)$ changes across the sequence. As an example, we explain how the removal of a hyperedge e from H_i can cause at least two changes in H_{i+1} , which can eventually lead to $O(2^k) = O(m)$ changes throughout the sequence. Assume a hyperedge e is removed from H and that e also belongs to the coreset hypergraph C_i of H_{i-1} . As H_{i+1} is a sub-hypergraph of $H_i = C_i \cup S_i$, e may also belong to H_{i+1} , necessitating its removal from H_{i+1} as well. To maintain C_i as a coreset of H_{i-1} , the algorithm must replace e with a next heaviest hyperedge e' from $H_{i-1} \setminus C_i$. If e' is an unsampled hyperedge (i.e., belongs to $H_{i-1} \setminus S_i$), its addition to C_i (and so H_i) may require updating H_{i+1} as well: as H_{i+1} is a sub-hypergraph of H_i , the (newly added) hyperedge e' could be heavier than some hyperedge e'' in C_{i+1} , necessitating the replacement of e'' in C_{i+1} . Thus, the removal of a hyperedge e from H_i can trigger at least two removals (namely, e and e'') and one insertion (namely, e') in H_{i+1} . See [Subsection 3.1](#) for further details.

To overcome this issue, our static algorithm deviates from that of [OST23] in the following way. We ensure that each hyperedge is included in at most one coreset by recursing on the sampled hypergraph S_i rather than on $C_i \cup S_i$. At the same time, we add C_i to the sparsifier \tilde{H} . i.e., we set $\tilde{H} = C_1 \cup \dots \cup C_k \cup S_k$, which can be verified to remain a spectral sparsifier of H (as discussed in [Lemma 3.2](#)). See [Figure 1b](#) for an illustration. Using this approach, after the deletion of e from H_i , each hypergraph H_{i+1}, \dots, H_k in the sequence will undergo at most one

¹In a directed hypergraph, each hyperedge e is a pair $(t(e), h(e))$, with the tail $t(e) \subseteq V$ and the head $h(e) \subseteq V$ reflecting the direction of e . See [Section 2](#) for further details.

²If multiple hyperedges have equal weight, the algorithm picks them arbitrarily.

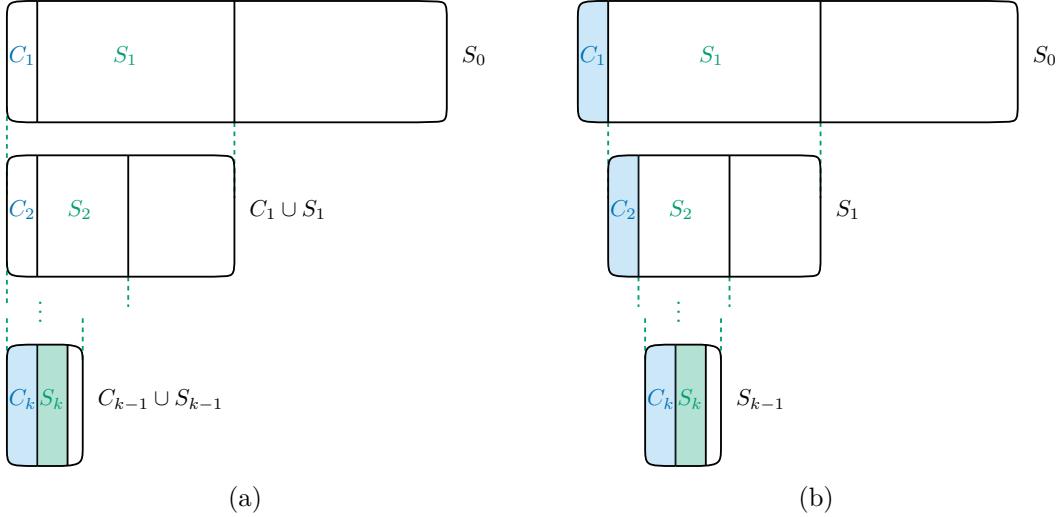


Figure 1: Comparison of (a) the algorithm of [OST23] and (b) our static algorithm. In each iteration i , their algorithm recurses on $H_{i-1} = C_{i-1} \cup S_{i-1}$ and computes $H_i = C_i \cup S_i$ for the next iteration. In contrast, our algorithm recurses solely on S_{i-1} , adds the coresets C_i to the sparsifier \tilde{H} , and computes the sampled hypergraph S_i for the next iteration. After $k = O(\log m)$ iterations, our algorithm terminates and returns $\tilde{H} = C_1 \cup \dots \cup C_k \cup S_k$, whereas the algorithm of [OST23] returns $\tilde{H} = C_k \cup S_k$ (the shaded parts in the figures). The increase in the size of \tilde{H} in our algorithm allows us to maintain \tilde{H} efficiently in the dynamic setting, as detailed in Subsection 3.2.

change. Specifically, in the case of replacing a hyperedge e' by e as in the high-recourse example above, there are two possibilities: either (1) e' does not belong to H_{i+1} (i.e., it is not in the sampled hypergraph S_i), in which case the replacement does not affect H_{i+1} , or (2) e' belongs to H_{i+1} , in which case the update is interpreted as the removal of e' from H_{i+1} (note that since C_i is excluded from H_{i+1} , the hyperedge e no longer belongs to H_{i+1}, \dots, H_k). The downside of this approach is that it increases the size of \tilde{H} from $O(n^2/\varepsilon^2 \log^3(n/\varepsilon))$ to $O(n^2/\varepsilon^2 \text{poly}(\log m))$. This increase becomes noticeable only when m is exponential in n . Even in that case, however, the size of \tilde{H} stays $\text{poly}(n)$, which is asymptotically much smaller than the exponential size of H .

To dynamize our static algorithm, we adapt an approach similar to that of [ADKKP16] for graphs. We first design a decremental data structure ([Algorithm 4](#) in [Subsection 3.2.1](#)), where the updates consist of only hyperedge deletions, and then use it to design a fully dynamic data structure ([Algorithm 5](#) in [Subsection 3.2.2](#)) via a reduction technique.

Our decremental algorithm leverages the fact that a hyperedge deletion in H causes at most one hyperedge deletion in each hypergraph in the sequence H_1, \dots, H_k . The removal of a hyperedge e from $H_i = C_i \cup S_i$ is handled using a straightforward replacement scheme. If e belongs to the sampled hypergraph S_i , then S_i remains *valid* after the removal of e , in the sense that it is still a set of hyperedges sampled uniformly at random from the updated $H_i \setminus C_i$. The update procedure is more involved if e belongs to the cores C_i . Recall that e was added to C_i through a pair $(u, v) \in V \times V$, where u belongs to the tail of e and v belongs to the head of e . The replacement of e is handled by removing it from all sets representing such

pairs and then selecting a hyperedge with high weight that is not already in C_i . Since there are $O(r^2)$ such pairs, this results in an $\tilde{O}(r^2)$ update time for maintaining H_i . See Subsection 3.2.1 for more details.

To convert our decremental data structure to a fully dynamic one, we leverage the decomposability property of spectral sparsifiers: the union of spectral sparsifiers of hyperedge partitions of H forms a spectral sparsifier of H (see Lemma 2.1 for a formal statement). The main idea is to maintain a hyperedge partition I_1, \dots, I_l of H where $|E(I_i)| \leq 2^i$ at any time. Deletions are handled by passing them to the respective I_i , whereas insertions are more difficult to handle: the data structure finds an integer j and moves all the hyperedges in I_1, \dots, I_{j-1} to I_j along with the inserted hyperedge. The choice of j depends on the number of insertions so far, with smaller values of j (corresponding to hypergraphs considerably smaller than H) being chosen more frequently. On each I_i , we run our decremental data structure to maintain a spectral sparsifier \tilde{I}_i of I_i , and upon an insertion, we reinitialize it for I_j . The algorithm sets $\tilde{H} = \tilde{I}_1 \cup \dots \cup \tilde{I}_l$, and since $k = O(\log m)$ the desired size of \tilde{H} follows. See Subsection 3.2.2 for further discussion.

Our approach in designing a decremental data structure and extending it to a fully dynamic one is similar to recent work on dynamic sparsification for undirected hypergraphs [GM25; KLP25]. Moreover, [GM25] also builds on the framework of [OST23]. The key difference is that we adapt the directed framework of [OST23] (specifically, λ -coresets), whereas [GM25] employs their undirected framework (specifically, t -bundle hyperspanners, a concept also used in [KLP25]). Both [GM25] and [KLP25] rely on spanner-based techniques to bound the effective resistances in the *associated graph* of the hypergraph as a crucial step in enabling their simple sampling scheme. For the directed case, however, [OST23] shows that this translates to using coresets, which are structurally simpler than hyperspanners. This structural simplicity allows us to design relatively simpler algorithms compared to those in [GM25; KLP25].

It is noteworthy to mention the recent work of [KPS24] that reduces directed hypergraph sparsification to undirected hypergraph sparsification. Combining this reduction with the fully dynamic undirected hypergraph sparsification result of [KLP25] yields a dynamic algorithm for directed hypergraph sparsification with guarantees similar to ours. However, the algorithm of [KLP25] is substantially more involved: (i) it relies on vertex-sampling steps, which our approach does not require, and (ii) it uses dynamic graph spanner constructions in a black-box manner. Moreover, it does not seem straightforward to extend their algorithm to the batch-parallel setting. In comparison, our coreset-based construction is significantly simpler and potentially more practically relevant. It readily extends to the batch-parallel setting and achieves a better and explicit polylogarithmic overhead in both sparsifier size and update time.

Lastly, in Section A, we show how to parallelize our data structures when H undergoes batches of k hyperedge deletions or additions as a single update. This discussion also explains how to adapt our fully dynamic data structure to support batch updates rather than single updates.

2 Preliminaries

Hypergraphs. A hypergraph $H = (V, E, \mathbf{w})$ consists of a set of vertices V , a set of hyperedges E , and a weight vector $\mathbf{w} \in \mathbb{R}_+^{|E|}$. The direction of a hyperedge $e \in E$ is defined by sets $t(e)$ and $h(e)$ as follows. Each hyperedge $e \in E$ is a pair $(t(e), h(e))$, where both $t(e)$ and $h(e)$ are

non-empty subsets of V . The sets $t(e)$ and $h(e)$ are called the *tail* and *head* of e , respectively; they indicate the direction of e . Note that $t(e)$ and $h(e)$ may overlap.

We use $E(H)$ to denote the set E of hyperedges of H whenever necessary, to avoid possible confusion. We define $n = |V|$ and $m = |E|$ and call H an m -edge n -vertex hypergraph. We say H is of rank r if, for every hyperedge $e \in E$, $|t(e) \cup h(e)| \leq r$.

Given a vector $\mathbf{x} \in \mathbb{R}^n$ defined on the set of vertices V , we define x_v to be the value of vector \mathbf{x} at the element associated with vertex $v \in V$. Similarly, for a vector $\mathbf{w} \in \mathbb{R}_+^m$ defined on the set of hyperedges E , we define w_e to be the value of vector \mathbf{w} at the element associated with hyperedge $e \in E$.

Spectral Sparsification of Directed Hypergraphs. We define the spectral property of a directed hypergraph $H = (V, E, \mathbf{w})$ using the energy function defined in the following. For a vector $\mathbf{x} \in \mathbb{R}^n$, we define the energy of \mathbf{x} with respect to H as

$$Q_H(\mathbf{x}) = \sum_{e \in E} w_e \max_{u \in t(e), v \in h(e)} (x_u - x_v)_+^2,$$

where $(x_u - x_v)_+ = \max\{x_u - x_v, 0\}$ and $(x_u - x_v)_+^2 = ((x_u - x_v)_+)^2$. Note that this definition generalizes a similar definition for graphs, given by $Q_G(\mathbf{x}) = \sum_{uv \in E} w_{uv} (x_u - x_v)^2$, which represents the total energy dissipated in G when viewed as an electrical network. In such electrical network, the endpoints of each edge uv have potentials x_u and x_v , respectively, and the edge itself has resistance $1/w_{uv}$. This interpretation is closely related to the notion of electrical flows, a concept that is deeply intertwined with spectral analysis.

The central object of this paper is the notion of a spectral hypersparsifier. A hypergraph $\tilde{H} = (V, \tilde{E}, \tilde{\mathbf{w}})$ is called a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H if for every vector $\mathbf{x} \in \mathbb{R}^n$,

$$(1 - \varepsilon)Q_{\tilde{H}}(\mathbf{x}) \leq Q_H(\mathbf{x}) \leq (1 + \varepsilon)Q_{\tilde{H}}(\mathbf{x}).$$

The following lemma will be useful later in proving the guarantees of our algorithm.

Lemma 2.1 (Decomposability). *Let H_1, \dots, H_k partition the hyperedges of a hypergraph H . For each $1 \leq i \leq k$, let \tilde{H}_i be a $(1 \pm \varepsilon)$ -spectral hypersparsifiers of H_i . Then, the union $\bigcup_{i=1}^k \tilde{H}_i$ is a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H .*

Proof. By definition, for every vector $\mathbf{x} \in \mathbb{R}^n$, we have

$$(1 - \varepsilon)Q_{\tilde{H}_i}(\mathbf{x}) \leq Q_{H_i}(\mathbf{x}) \leq (1 + \varepsilon)Q_{\tilde{H}_i}(\mathbf{x}).$$

Summing over all $1 \leq i \leq k$, results in

$$(1 - \varepsilon) \sum_{i=1}^k Q_{\tilde{H}_i}(\mathbf{x}) \leq \sum_{i=1}^k Q_{H_i}(\mathbf{x}) \leq (1 + \varepsilon) \sum_{i=1}^k Q_{\tilde{H}_i}(\mathbf{x}),$$

which means

$$(1 - \varepsilon)Q_{\tilde{H}}(\mathbf{x}) \leq Q_H(\mathbf{x}) \leq (1 + \varepsilon)Q_{\tilde{H}}(\mathbf{x})$$

as H_1, \dots, H_k partition H . \square

Chernoff Bound [Che52; MU17]. Let X_1, \dots, X_k be independent random variables, where each X_i equals 1 with probability p_i , and 0 otherwise. Let $X = \sum_{i=1}^k X_i$ and $\mu = \mathbb{E}[X] = \sum_{i=1}^k p_i$. Then, for all $\delta \geq 0$,

$$\mathbb{P}[X \geq (1 + \delta)\mu] \leq \exp\left(-\frac{\delta^2\mu}{2 + \delta}\right). \quad (1)$$

Parallel Batch-Dynamic Model. We use the work-depth model to analyze our parallel algorithm. Work is defined as the total number of operations done by the algorithm, and depth is the length of the longest chain of dependencies. Intuitively, work measures the time required for the algorithm to run on a single processor, whereas depth measures the optimal time assuming the algorithm has access to an unlimited number of processors.

In the batch-dynamic setting, each update consists of a batch of k insertions or deletions, and the goal is to take advantage of performing several hyperedge insertions or deletions as a single update to improve the work and depth of the parallel algorithm. Note that this model is equivalent to the one with mixed updates (i.e., when the batch consists of both insertions and deletions), as this can be transformed into two steps, each consisting of insertions and deletions separately, without asymptotically increasing the work or depth.

3 Dynamic Spectral Sparsification

In this section, we present our fully dynamic algorithm of [Theorem 1.1](#) for maintaining a $(1 \pm \varepsilon)$ -spectral hypersparsifier of a directed hypergraph $H = (V, E, \mathbf{w})$. To achieve this goal, we use a static algorithm as the cornerstone for our dynamic algorithm. The static algorithm is explained in [Subsection 3.1](#) and is followed by the dynamic data structure in [Subsection 3.2](#).

3.1 The Static Algorithm

We start by briefly explaining the algorithm of [\[OST23\]](#), which serves as a foundation for our algorithm. Their algorithm constructs a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of H using an iterative approach: starting with $H_0 = H$, each iteration i computes a sub-hypergraph H_i of H_{i-1} , until the final iteration computes $H_{i_{\text{last}}}$, where $\tilde{H} = H_{i_{\text{last}}}$.

To compute H_i from H_{i-1} , the algorithm first obtains a λ -coreset C_i of H_{i-1} . Roughly speaking, C_i is a sub-hypergraph containing ‘heavyweight’ hyperedges of H_{i-1} and is defined as follows. The algorithm defines an ordering on hyperedges in H by their weights in decreasing order. Note that this ordering naturally extends to every hypergraph H_i as a sub-hypergraph of H . To construct C_i , the algorithm examines every pair $(u, v) \in V \times V$ and selects the first λ hyperedges in the ordering (if any) that are not already in C_i , whose tail contains u and whose head contains v . These hyperedges are then added to C_i . The second building-block of H_{i-1} is the sampled hypergraph S_i of H_{i-1} defined as follows. The algorithm samples the non-heavy hyperedges (i.e., the ones in $H_{i-1} \setminus C_i$) with probability $1/2$ and adds them to S_i while doubling their weight. Consequently, $H_i = C_i \cup S_i$. See [Algorithm 1](#) for a pseudocode.

Having heavyweight hyperedges in C_i ensures that the simple sampling scheme used for constructing S_i results in $H_i = C_i \cup S_i$, which, with high probability, is a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H_{i-1} [\[OST23, Lemma 4.3\]](#). Due to the sampling scheme, H_i is roughly half the size of H_{i-1} , which ensures the termination of the algorithm after $i_{\text{last}} = O(\log m)$ iterations.

Algorithm 1: CORESET-AND-SAMPLE(H, ε)

Input: an m -edge hypergraph $H = (V, E, \mathbf{w})$ and $0 < \varepsilon < 1$
Output: a coresset C of H and a sampled hypergraph S

```

1  $\lambda \leftarrow c_\lambda \log^3 m / \varepsilon^2$                                 /*  $c_\lambda$  is a sufficiently large constant */
2  $C, S \leftarrow (V, \emptyset, \mathbf{0})$ 
3 Order the hyperedges of  $H$  by their weights in decreasing order
4 foreach pair  $(u, v) \in V \times V$  do                                     /* compute  $C$  */
5   Construct the set  $E(u, v) \subseteq E$  such that  $e \in E(u, v)$  iff  $u \in t(e)$  and  $v \in h(e)$ 
6   Add the first  $\lambda$  hyperedges (if any) in the ordering from  $E(u, v) \setminus C$  to  $C$  while
      retaining their weights
7 foreach hyperedge  $e$  of  $H \setminus C$  do                                     /* compute  $S$  */
8   With probability  $1/2$ , add  $e$  to  $S$  and double its weight
9 return  $(C, S)$ 

```

Using this sequence $H_1, \dots, H_{i_{\text{last}}}$ of hypergraphs, they achieve a sparsifier \tilde{H} with an almost optimal size of $O(n^2/\varepsilon^2 \log^3(n/\varepsilon))$ [OST23, Theorem 1.1]. See Figure 1a for an illustration.

Unfortunately, employing the sequence of hypergraphs $H_1, \dots, H_{i_{\text{last}}}$ can result in $O(m)$ recourse (and consequently, update time) in the dynamic setting. For example, consider the removal of a hyperedge e from H that also belongs to the coresset C_i of H_{i-1} . In this scenario, to ensure C_i remains a λ -coreset and so $H_i = C_i \cup S_i$ remains a $(1 \pm \varepsilon_i)$ -spectral hypersparsifier of H_{i-1} , we need to replace e with another hyperedge e' from $H_{i-1} \setminus C_i$. i.e., if e was added to C_i through the pair (u, v) , hyperedge e' in $H_{i-1} \setminus C_i$ is the next hyperedge in the ordering whose tail contains u and whose head contains v . Since S_i is a set of sampled hyperedges uniformly at random, it may be the case that e' does not belong to S_i , and therefore to none of H_j for $j > i$. Since e' is added to H_i after the update (as it now belongs to C_i), it must be taken into account in the update of H_{i+1} as a sub-hypergraph of (newly updated) H_i . But now, e' may be heavier than another hyperedge e'' in C_{i+1} , which necessitates the replacement of e'' with e' . Since e can be present in C_{i+1} as well, this means that the deletion of e from H_i can result in at least 2 changes in H_{i+1} , and at least $2^{k-i} = O(m)$ changes in the sequence, which is too expensive to afford.

To alleviate this issue, our static algorithm deviates from that of [OST23] by recursing on S_i instead of $C_i \cup S_i$ as follows. At each iteration i , the algorithm recurses on S_{i-1} , adds the coresset C_i of S_{i-1} to \tilde{H} , and samples the rest, $S_{i-1} \setminus C_i$, to obtain a smaller hypergraph S_i for the next iteration. More precisely, the algorithm starts with $S_0 = H$ and an empty sparsifier \tilde{H} . In the first iteration, it computes the coresset C_1 of S_0 , and adds it to \tilde{H} . The algorithm then samples the remaining hypergraph $S_0 \setminus C_1$ to compute S_1 by sampling every hyperedge in $S_0 \setminus C_1$ with probability $1/2$ and doubling their weight. Similar to [OST23], with high probability, $C_i \cup S_i$ is a $(1 \pm \varepsilon_i)$ -spectral hypersparsifier of S_{i-1} (Lemma 3.1). The algorithm then recurses on S_1 , and so on. Thus, our algorithm adds the sequence of coresets $C_1, \dots, C_{i_{\text{last}}}$ to \tilde{H} while recursing on the sequence of hypergraphs $S_1, \dots, S_{i_{\text{last}}}$. In the last iteration i_{last} , our algorithm adds $C_{i_{\text{last}}}$ as well as $S_{i_{\text{last}}}$ to \tilde{H} . See Algorithm 2 for a pseudocode and Figure 1b for an illustration.

This technique, which is similar to the one used in [ADKKP16] for graphs, ensures that each hyperedge of \tilde{H} is associated with at most *one* coresset. This guarantees $O(\log m)$ hyperedge

Algorithm 2: SPECTRAL-SPARSIFY(H, ε)

Input: hypergraph $H = (V, E, \mathbf{w})$ and $0 < \varepsilon < 1$
Output: a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of H

- 1 $i \leftarrow 0$
- 2 $k \leftarrow \lceil \log_{3/4} m \rceil$
- 3 $m^* \leftarrow n^2/\varepsilon^2 \log^3 m$
- 4 $\tilde{H} \leftarrow (V, \emptyset, \mathbf{0})$
- 5 $S_0 \leftarrow H$
- 6 **while** $i \leq k$ and $|E(H_i)| \geq 32cm^*$ **do** */* c is from Lemma 3.1 */*
- 7 $(C_{i+1}, S_{i+1}) \leftarrow \text{CORESET-AND-SAMPLE}(S_i, \varepsilon/(2k))$
- 8 $\tilde{H} \leftarrow \tilde{H} \cup C_{i+1}$
- 9 $i \leftarrow i + 1$
- 10 $i_{\text{last}} \leftarrow i$
- 11 $\tilde{H} \leftarrow \tilde{H} \cup S_{i_{\text{last}}}$
- 12 **return** \tilde{H}

deletions from $S_1, \dots, S_{i_{\text{last}}}$ after a hyperedge deletion in H (Lemma 3.5), but in return, results in a $\text{poly}(\log m)$ overhead in the size of \tilde{H} as we include the coresets $C_1, \dots, C_{i_{\text{last}}}$ in \tilde{H} . i.e., our algorithm ensures that \tilde{H} is a desired sparsifier of size $O(n^2/\varepsilon^2 \text{poly}(\log m))$, which is asymptotically much smaller than H even when m is exponential in n .

The rest of this section examines the correctness of Algorithms 1 and 2. Algorithm 1, used as a subroutine of Algorithm 2, computes a coresnet and a sampled hypergraph of the input hypergraph. The guarantees of Algorithm 1 are stated in the following lemma.

Lemma 3.1. *Let $0 < \varepsilon < 1$ and let $H = (V, E, \mathbf{w})$ be an m -edge n -vertex hypergraph. For any positive constant $c \geq 1$, if $m \geq c \log m$, then Algorithm 1 returns a coresnet C and a sampled hypergraph S such that $C \cup S$ is a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H of size $O(m/2 + (2cm \log m)^{1/2} + \lambda n^2)$ with probability at least $1 - 1/m^c$.*

Proof. The lemma is derived from [OST23, Lemma 4.3]. The only difference is that we guarantee a probability of success of at least $1 - 1/\text{poly}(m)$, instead of $1 - 1/\text{poly}(n)$. Thus, we only prove the claim about the probability using an argument similar to that in [OST23, Lemma 4.4].

High probability claim: For every hyperedge e in $H \setminus C$, we define the random variable X_e to be equal 1 if e is sampled to be in S , and 0 otherwise. Let $X = \sum_{e \in S} X_e$. Since each hyperedge in S is sampled independently with probability $1/2$, we can use Equation (1) and we have $\mu = \mathbb{E}[X] = 1/2|H \setminus C| \leq m/2$. By substituting $\delta = (2c \log m / m)^{1/2} \leq 2$ in Equation (1),

$$\mathbb{P}\left[X \geq \left(1 + \left(\frac{8c \log m}{m}\right)^{1/2}\right) \frac{m}{2}\right] \leq \exp\left(-\frac{1}{4} \left(\frac{8c \log m}{m}\right) \frac{m}{2}\right) = \exp(-c \log m) = \frac{1}{m^c},$$

or equivalently

$$\mathbb{P}\left[X \leq \frac{m}{2} + (2cm \log m)^{1/2}\right] \geq 1 - \frac{1}{m^c}. \quad (2)$$

Since each pair $(u, v) \in V \times V$ adds at most λ hyperedges to S , we have $|S| = O(\lambda n^2)$. Together with [Equation \(2\)](#), it follows that $\tilde{H} = C \cup S$ has size $O(m/2 + (2cm \log m)^{1/2} + \lambda n^2)$ with probability at least $1 - 1/m^c$. \square

The guarantees of [Algorithm 2](#) are stated in the following lemma.

Lemma 3.2. *Let $0 < \varepsilon < 1$ and let $H = (V, E, \mathbf{w})$ be an m -edge n -vertex hypergraph. Then, with high probability, [Algorithm 2](#) returns a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of H of size $O(n^2/\varepsilon^2 \log^6 m)$.*

Proof. We prove each guarantee separately below.

Approximation guarantee: We prove that $H'_l = \bigcup_{j=1}^l C_j \cup S_l$ is a $(1 \pm \varepsilon/(2k))^l$ -spectral hypersparsifier of H by induction on l .

If $l = 1$, by [Lemma 3.1](#), $H_1 = C_1 \cup S_1$ is a $(1 \pm \varepsilon/(2k))$ -spectral hypersparsifier of H .

Suppose that, for an integer $l > 1$, H'_l be a $(1 \pm \varepsilon/(2k))^l$ -spectral hypersparsifier of H . To compute H'_{l+1} , the algorithm computes a $(1 + \varepsilon/(2k))$ -spectral hypersparsifier $\tilde{S}_l = C_{l+1} \cup S_{l+1}$ of S_l . Since $\bigcup_{j=1}^l C_j \cup S_l$ partition H'_l , by [Lemma 2.1](#), $H'_{l+1} = \bigcup_{j=1}^{l+1} C_j \cup (C_{l+1} \cup S_{l+1})$ is a $(1 \pm \varepsilon/(2k))$ -spectral hypersparsifier of H'_l . For every vector $\mathbf{x} \in \mathbb{R}^n$, we have

$$Q_{H'_{l+1}}(\mathbf{x}) \leq (1 + \varepsilon/(2k)) Q_{H'_l}(\mathbf{x}) \leq (1 + \varepsilon/(2k)) (1 + \varepsilon/(2k))^l Q_H(\mathbf{x}) = (1 + \varepsilon/(2k))^{l+1} Q_H(\mathbf{x}).$$

Similarly, $(1 - \varepsilon/(2k))^{l+1} Q_H(\mathbf{x}) \leq Q_{H'_{l+1}}(\mathbf{x})$, and so H'_{l+1} is a $(1 \pm \varepsilon/(2k))^{l+1}$ -spectral hypersparsifier of H .

The desired bound follows from the fact that $i_{\text{last}} \leq k$, and

$$(1 + \varepsilon/(2k))^k \leq (1 + \varepsilon) \quad \text{and} \quad (1 - \varepsilon) \leq (1 - \varepsilon/(2k))^k.$$

Size of \tilde{H} : Let m_i be the number of hyperedges present in S_i , where $1 \leq i \leq i_{\text{last}}$. We first show that $m_i \leq 3m_{i-1}/4$. By [Equation \(2\)](#), S_i has size $O(m_{i-1}/2 + (2cm_{i-1} \log m_{i-1})^{1/2})$, so it suffices to show that $(2cm_{i-1} \log m_{i-1})^{1/2} \leq m_{i-1}/4$. By the assumption of the loop, we have

$$m_{i-1} \geq 32cm^* = 32cn^2/\varepsilon^2 \log^3 m \geq 32c \log m_{i-1},$$

and thus $(2cm_{i-1} \log m_{i-1})^{1/2} \leq m_{i-1}/4$. This means that the size of $S_{i_{\text{last}}}$ is

$$O(\max\{\log m, m^*\}) = O(n^2/\varepsilon^2 \log^3 m).$$

The rest of \tilde{H} consists of the coresets $C_1, \dots, C_{i_{\text{last}}}$. By [Lemma 3.1](#), each C_i has size $O(\lambda_i n^2) = n^2/(\varepsilon/2k)^2 \log^3 m$. Since $k = O(\log m)$, the total size of the coresets is

$$O\left(\sum_{l=1}^k k^2 n^2 / \varepsilon^2 \log^3 m\right) = O(n^2/\varepsilon^2 \log^6 m).$$

Therefore, the total size of \tilde{H} is

$$O(n^2/\varepsilon^2 \log^3 m + n^2/\varepsilon^2 \log^6 m) = O(n^2/\varepsilon^2 \log^6 m).$$

High probability claim: Since $i_{\text{last}} = O(\log m)$ and from [Lemma 3.1](#), \tilde{H} is a $(1 \pm \varepsilon)$ -spectral hypersparsifier with probability at least $1 - O(\log m/m^c)$. The claim follows by choosing $c \geq 2$. \square

3.2 The Dynamic Algorithm

In this section, we dynamize our static algorithm ([Algorithm 2](#)). We first design a decremental data structure in [Subsection 3.2.1](#), where H undergoes only hyperedge deletions. Then, we reduce it to a fully dynamic data structure in [Subsection 3.2.2](#) using the following lemma.

Lemma 3.3. *Assume that there is a decremental algorithm that with probability at least $1 - 1/\text{poly}(m')$, maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier of any hypergraph with m' initial hyperedges in $T(m', n, \varepsilon^{-1})$ amortized update time and of size $S(m', n, \varepsilon^{-1})$, where S and T are monotone non-decreasing functions. Then, the algorithm can be transferred into a fully dynamic algorithm that, with high probability, maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier of any hypergraph with m hyperedges (at any point) in $O(T(m, n, \varepsilon^{-1}) \log m)$ amortized update time of size $O(S(m, n, \varepsilon^{-1}) \log m)$.*

The fully dynamic to decremental reduction ([Lemma 3.3](#)) uses the batching technique [[ADKKP16](#); [GM25](#); [KLP25](#)] and is explained in [Subsection 3.2.2](#).

3.2.1 Decremental Spectral Sparsifier

In this section, we explain our data structure ([Algorithm 4](#)) to decrementally maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of H . As [Algorithm 1](#) is used as a subroutine of [Algorithm 2](#), we begin by explaining its decremental implementation (presented in [Algorithm 3](#)).

Decremental Implementation of [Algorithm 1](#). To decrementally maintain a coresset C and a sampled hypergraph S of H , we need to ensure that after each deletion, the maintained $C \cup S$ remains a valid sparsifier for H . i.e., it continues to be a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H ([Lemma 3.1](#)).

Recall that, for every pair $(u, v) \in V \times V$, [Algorithm 1](#) defines the set $E(u, v)$ of hyperedges where $e \in E(u, v)$ iff $u \in t(e)$ and $v \in h(e)$. It then constructs a coresset C by adding λ (defined in [Algorithm 1](#)) heaviest hyperedges of $E(u, v) \setminus C$ to C . The hypergraph S is then obtained by sampling each hyperedge in $H \setminus C$ with probability $1/2$ while doubling its weight.

To construct C , the algorithm greedily chooses a pair (u, v) , adds its heavyweight hyperedges to C , and continues with another pair (u', v') . Note that the order of choosing the pairs might affect the choice of hyperedges included in C . Nevertheless, the guarantee of the algorithm ([Lemma 3.1](#)) is independent of this order. For example, it does not matter whether (u, v) is chosen first or (u', v') is. We will use this fact later to maintain a valid C and S after each deletion.

We use the same procedure to initialize the decremental implementation. Since we would need to find the heaviest hyperedges in $E(u, v) \setminus C$ after a deletion, we also order each $E(u, v)$.

Suppose that hyperedge e has been removed from H . Then, e must belong to one of the three cases below, which together cover all hyperedges of H .

- If e belongs to neither C nor to S , then its removal does not affect $C \cup S$. This is because C still contains the heaviest hyperedges associated with each pair $(u, v) \in V \times V$, and each hyperedge in S has been independently sampled.
- If e belongs to S , similar to the previous case, the sampled hypergraph S after the removal of e is still a valid sample. Thus, no further changes are required to maintain $C \cup S$.

Algorithm 3: DECREMENTAL-CORESET-AND-SAMPLE(H, ε)

```

Input: an  $m$ -edge hypergraph  $H = (V, E, \mathbf{w})$  and  $0 < \varepsilon < 1$ 
Maintain: a coresset  $C$  and a sampled hypergraph  $S$  of  $H$ 
1  $C, S \leftarrow (V, \emptyset, \mathbf{0})$ 
2 Procedure INITIALIZE
3    $(C, S) \leftarrow \text{CORESET-AND-SAMPLE}(H, \varepsilon)$            /* initialize  $C$  and  $S$  */
4   foreach pair  $(u, v) \in V \times V$  do
5     └ Order hyperedges in  $E(u, v)$  from highest to lowest weight
6 Procedure DELETE( $e$ )
7   Remove  $e$  from  $H$ 
8   if  $C$  contains  $e$  then
9     Let  $E(u, v)$  be the set from which  $e$  was added to  $C$ 
10    Choose a hyperedge  $e'$  in  $E(u, v) \setminus C$  with highest weight and add it to  $C$ 
11    Remove  $e'$  from  $S$ 
12    Remove  $e$  from  $C$ 
13   foreach pair  $(u, v) \in V \times V$  with  $u \in t(e)$  and  $v \in h(e)$  do
14     └ Remove  $e$  from  $E(u, v)$ 
15   Remove  $e$  from  $S$ 

```

- If e belongs to C , then the deletion from C translates to undoing the addition of e to C from the specific set $E(u, v)$ that originally added e to C . In this case, since e no longer exists in $E(u, v)$, we add a heaviest hyperedge e' from $E(u, v) \setminus C$ to C . Since the order of hyperedge addition to C does not affect its guarantees (Lemma 3.1), the updated C remains valid. For bookkeeping reasons, if e' was previously sampled, we remove it from S . Based on the previous discussion, the updated S is also valid.

In addition to the changes explained above, the maintenance involves the removal of e from every set $E(u, v)$ containing e , which, as explained in the lemma below, adds $O(\log m)$ overhead to the update time. Putting it all together, Algorithm 3 is our decremental data structure for maintaining $C \cup S$. We have the following.

Lemma 3.4. *Given a constant $c \geq 2$ and an m -edge n -vertex hypergraph $H = (V, E, \mathbf{w})$ of rank r undergoing hyperedge deletions. If $m \geq c \log m$, then Algorithm 3 maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier $C \cup S$ of H of size $O(m/2 + (2cm \log m)^{1/2} + n^2/\varepsilon^2 \log^3 m)$ in $O(r^2 \log m)$ amortized update time with probability at least $1 - 1/m^{c-1}$.*

Proof. Suppose the hyperedge e is removed from H .

Correctness: From the discussion above, it follows immediately that $C \cup S$, after the update, is a valid sparsifier for H ; the correctness of its spectral property is addressed in the high probability claim below.

Size of $C \cup S$: Since after each hyperedge deletion from $C \cup S$, the algorithm substitutes it with at most one other hyperedge from H , the size of $C \cup S$ is monotonically decreasing. Thus, $C \cup S$ has the same size guarantee as of Algorithm 1 explained in Lemma 3.1, which is

$$O(m/2 + (2cm \log m)^{1/2} + \lambda n^2) = O(m/2 + (2cm \log m)^{1/2} + n^2/\varepsilon^2 \log^3 m).$$

Update time: Since each hyperedge e contains $O(r)$ vertices, the total number of $E(\cdot, \cdot)$'s containing e is $O(r^2)$. Thus, constructing $E(\cdot, \cdot)$'s, takes $O(mr^2)$ time, while ordering them adds $O(mr^2 \log m)$ time to the initialization time as well. As explained before, the deletion of e from H translates to its deletion from at most r^2 sets of $E(\cdot, \cdot)$'s, each of which takes $O(\log m)$ time, i.e., $O(mr^2 \log m)$ total time. Other changes, such as substituting another hyperedge in S , can be done by probing $E(\cdot, \cdot)$'s only once in total. We conclude that the total update time is $O(mr^2 \log m)$, resulting in $O(r^2 \log m)$ amortized update time.

High probability claim: By choosing $c \geq 2$ in [Lemma 3.1](#), each update is guaranteed to succeed with probability at least $1 - 1/m^c$. Since there are at most m updates, it follows that [Algorithm 3](#) succeeds with probability at least $1 - 1/m^{c-1}$. \square

Decremental Implementation of Algorithm 2. Our decremental data structure for maintaining a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of H ([Algorithm 4](#)) is a decremental implementation of [Algorithm 2](#). Recall that, in [Algorithm 2](#), we recurse on the sampled hypergraphs $S_1, \dots, S_{i_{\text{last}}}$ and add the coresets $C_1, \dots, C_{i_{\text{last}}}$ to \tilde{H} . By [Lemma 3.2](#), $\tilde{H} = \bigcup_{j=1}^{i_{\text{last}}} C_j \cup S_{i_{\text{last}}}$ is a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H .

In [Algorithm 4](#), we decrementally maintain the hypergraphs $S_1, \dots, S_{i_{\text{last}}}$ and the coresets $C_1, \dots, C_{i_{\text{last}}}$. Suppose that hyperedge e has been deleted from H as the most recent update. Below, we discuss how the data structure handles the deletion in all possible scenarios.

- If e does not belong to \tilde{H} , then e can only exist in the sets of sampled hyperedges $S_1, \dots, S_{i_{\text{last}}-1}$. In this case, removing e from all S_i 's does not affect \tilde{H} since the set of sampled hyperedges remains valid, as each hyperedge is sampled independently.
- If e belongs to $S_{i_{\text{last}}}$, then e is present in all sets of sampled hyperedges, i.e., in $S_1, \dots, S_{i_{\text{last}}}$. Note that, although e belongs to \tilde{H} in this case, since it is only present in sampled hypergraphs, its removal does not affect the coresets and can be handled similarly to the previous case.
- If e belongs to a coreset C_i , then it belongs to the sets of sampled hyperedges S_1, \dots, S_{i-1} , and since it is in the coreset C_i of S_{i-1} , it cannot be present in $S_i, \dots, S_{i_{\text{last}}}$. Similar to the previous cases, we can simply remove e from S_1, \dots, S_{i-1} to maintain $C_1 \cup S_1, \dots, C_{i-1} \cup S_{i-1}$. For $C_i \cup S_i$, we need to maintain C_i , for which we use [Algorithm 3](#) as a subroutine to find a hypergraph e' to be added to C_i as the substitution for e . By [Algorithm 3](#), e' belongs to S_{i-1} and thus its addition to C_i does not affect $C_1 \cup S_1, \dots, C_{i-1} \cup S_{i-1}$. On the other hand, e' might be present in $C_{i+1} \cup S_{i+1}, \dots, C_{i_{\text{last}}} \cup S_{i_{\text{last}}}$, and we need to ensure they are still valid after adding e' to C_i . If e' appears in the sets of sampled hyperedges, then similar to the previous cases, simply removing e' from the sparsifiers makes them valid. Therefore, we pass the deletion of e' to the next sparsifiers until we reach $C_j \cup S_j$ containing e' in its coreset C_j . Again, by construction, e' cannot be present in $C_{j+1} \cup S_{j+1}, \dots, C_{i_{\text{last}}} \cup S_{i_{\text{last}}}$, and we need to remove e' from C_j . This procedure is similar to the one we just explained for the removal of e from C_i , except that we now use it to remove e' from C_j .

As explained above, to handle hyperedge deletions in each sparsifier $C_i \cup S_i$, the data structure utilizes [Algorithm 3](#) and then transmits the changes in $C_i \cup S_i$ to $C_{i+1} \cup S_{i+1}$. The key point of our data structure is that it guarantees the transfer of at most *one* hyperedge deletion from one level to the next, thereby ensuring low recourse at each level. This results in at most $i_{\text{last}} = O(\log m)$ hyperedge deletions across all levels following a hyperedge deletion in H . The guarantees of the data structure are stated below.

Algorithm 4: DECREMENTAL-SPECTRAL-SPARSIFY(H, ε)

```

Input: an  $m$ -edge hypergraph  $H = (V, E, \mathbf{w})$  and  $0 < \varepsilon < 1$ 
Maintain: a  $(1 \pm \varepsilon)$ -spectral hypersparsifier  $\tilde{H}$  of  $H$ 

1  $i \leftarrow 0$ 
2  $k \leftarrow \lceil \log_{3/4} m \rceil$ 
3  $m^* \leftarrow n^2/\varepsilon^2 \log^3 m$ 
4  $S_0 \leftarrow H$ 
5 Procedure INITIALIZE
6   while  $i \leq k$  and  $|E(H_i)| \geq 32cm^*$  do           /*  $c$  is from Lemma 3.1 */
7      $\mathcal{A}_{i+1} \leftarrow$  initialize DECREMENTAL-CORESET-AND-SAMPLE( $S_i, \varepsilon/(2k)$ )
                           /*  $\mathcal{A}_{i+1}$  decrementally maintains  $C_{i+1}$  and  $S_{i+1}$  */
8      $\tilde{H} \leftarrow \tilde{H} \cup C_{i+1}$ 
9      $i \leftarrow i + 1$ 
10     $i_{\text{last}} \leftarrow i$ 
11     $\tilde{H} \leftarrow \tilde{H} \cup S_{i_{\text{last}}}$ 
12 Procedure DELETE( $e$ )
13   Remove  $e$  from  $H$  and  $\tilde{H}$ 
14    $i \leftarrow 1$ 
15   while  $i \leq i_{\text{last}}$  do
16     Pass the deletion of  $e$  to  $\mathcal{A}_i$ 
17     if  $e'$  has been added to  $C_i$  due to the deletion of  $e$  then
18       Add  $e'$  to  $\tilde{H}$ 
19       Remove  $e'$  from  $S_i$ 
20        $e \leftarrow e'$ 
21      $i \leftarrow i + 1$ 

```

Lemma 3.5. Given a constant $c \geq 3$ and an m -edge n -vertex hypergraph $H = (V, E, \mathbf{w})$ of rank r undergoing hyperedge deletions, Algorithm 4 maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of H of size $O(n^2/\varepsilon^2 \log^6 m)$ in $O(r^2 \log^2 m)$ amortized update time with probability at least $1 - 1/m^{c-2}$. Additionally, each deletion in H results in $O(\log m)$ recourse in \tilde{H} .

Proof. Suppose that a deletion has happened in H .

Correctness: From Lemma 3.4, each \mathcal{A}_i correctly handles the deletions. The fact that the data structure correctly transfers hyperedge deletions between \mathcal{A}_i 's and thus correctly maintains \tilde{H} follows from the discussion above.

Size of \tilde{H} : The data structure includes $i_{\text{last}} = O(\log m)$ coresets $C_1, \dots, C_{i_{\text{last}}}$ in \tilde{H} . By Lemma 3.4, for each $1 \leq i \leq i_{\text{last}}$, C_i has a size of $O(n^2/\varepsilon_i^2 \log^3 m)$, where $\varepsilon_i = \varepsilon/(2k)$ and $k = O(\log m)$. As shown in the proof of Lemma 3.2, the size of $S_{i_{\text{last}}}$ is $O(n^2/\varepsilon^2 \log^3 m)$. It follows that the size of \tilde{H} is

$$O\left(\sum_{i=1}^k n^2/\varepsilon_i^2 \log^3 m + n^2/\varepsilon^2 \log^3 m\right) = O(n^2/\varepsilon^2 \log^6 m).$$

Update time: The data structure initializes and decrementally maintains $i_{\text{last}} = O(\log m)$ data structures of [Algorithm 3](#). By [Lemma 3.4](#), each data structure takes $O(mr^2 \log m)$ total update time. Also, at each update, the data structure transmits at most one hyperedge from one level to the next, which results in an $O(m \log m)$ total transmission. Thus, the algorithm takes $O(mr^2 \log^2 m)$ total update time, or equivalently, $O(r^2 \log^2 m)$ amortized update time.

High probability claim: From [Lemma 3.4](#), for any constant $c \geq 2$, each \mathcal{A}_i correctly maintains its coresets and the set of sampled hyperedges with probability at least $1 - 1/m^{c-1}$. Since $1 \leq i \leq \log m$, by choosing $c \geq 3$, the claim follows.

Recourse bound: As explained before, every hyperedge deletion from H translates to at most one hyperedge deletion or addition in \mathcal{A}_i for $1 \leq i \leq i_{\text{last}}$. Since $i_{\text{last}} = O(\log m)$, it follows that the total number of hyperedge changes in \tilde{H} is $O(\log m)$. \square

3.2.2 Fully Dynamic Spectral Sparsifier

With a decremental data structure ([Algorithm 4](#)) in hand, we now obtain the fully dynamic data structure. We begin by explaining the reduction from the fully dynamic to the decremental data structure and by proving [Lemma 3.3](#). We then prove [Theorem 1.1](#).

Reduction from Fully Dynamic to Incremental. We explain how to use the decremental data structure to design a fully dynamic data structure, as described in [Algorithm 5](#). In a nutshell, the data structure uses the batching technique; it maintains a batch of $(1 \pm \varepsilon)$ -spectral hypersparsifiers $\tilde{H}_1, \dots, \tilde{H}_k$, each of which decrementally maintains sub-hypergraphs H_1, \dots, H_k , respectively, such that these sub-hypergraphs partition H . The union $\tilde{H} = \bigcup_{i=1}^k \tilde{H}_i$, by decomposability ([Lemma 2.1](#)), is a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H .

More specifically, the data structure starts with an empty H and empty H_1, \dots, H_k and ensures that each H_i contains at most 2^i hyperedges at all time (which we call the size constraint of H_i). Intuitively, a hyperedge e inserted in H is placed in H_1 as long as the size constraint of H_1 is not violated. If there are already two hyperedges in H_1 , the data structure moves *all* hyperedges of H_1 (along with e) to the empty H_2 . Note that H_2 can contain at most four hyperedges, and so its size constraint is not violated. However, if there were already two hyperedges in H_2 , then the addition of three more hyperedges (from H_1 plus e) would violate its size constraint. In this case, the data structure would move the hyperedges of H_1 and H_2 plus e to H_3 , and so on.

To regularize the insertion process, we initialize a counter t as a binary sequence of $\log k$ zeros. After each insertion, the data structure increments t (in the binary format) by one and finds the highest index i whose bit flipped due to the increment. The data structure then moves the hyperedges in H_1, \dots, H_{i-1} , along with the inserted hyperedge e , to H_i and reinitializes the decremental data structure with the new H_i .

The deletion process can be done as before; since each hyperedge is associated with exactly one sub-hypergraph H_i , its deletion would be easily handled by passing it to the decremental data structure maintaining \tilde{H}_i .

Proof of Lemma 3.3. Suppose that the decremental data structure maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier of any hypergraph with m' initial hyperedges in $T(m', n, \varepsilon^{-1})$ amortized update time and of size $S(m', n, \varepsilon^{-1})$ with probability at least $1 - 1/\text{poly}(m')$.

Correctness: From the discussion above, the sub-hypergraphs H_1, \dots, H_k correctly partition the hyperedges of H . By [Lemma 3.5](#), the sparsifiers $\tilde{H}_1, \dots, \tilde{H}_k$ are correctly maintained. It

Algorithm 5: FULLY-DYNAMIC-SPECTRAL-SPARSIFY(H, ε)

Input: an empty n -vertex hypergraph $H = (V, E, \mathbf{w})$ with $|E| \leq m$ at any time
Maintain: a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of H

```

1 Procedure INITIALIZE
2   foreach  $1 \leq i \leq \log m$  do
3      $H_i, \tilde{H}_i \leftarrow (V, \emptyset, \mathbf{0})$ 
4      $t \leftarrow 0$ 
5      $i_{\text{last}} \leftarrow 1$ 
6 Procedure ADD( $e$ )
7    $t \leftarrow t + 1$ 
8    $j \leftarrow \max\{i \mid t \text{ is divisible by } 2^{i-1}\}$ 
9    $i_{\text{last}} \leftarrow \max\{j, i_{\text{last}}\}$ 
10   $H_j \leftarrow H_j \cup H_{j-1} \cup \dots \cup H_1$ 
11  Add  $e$  to  $H_j$ 
12  foreach  $1 \leq i \leq j - 1$  do
13     $H_i, \tilde{H}_i \leftarrow (V, \emptyset, \mathbf{0})$ 
14   $\mathcal{A}_j \leftarrow (\text{re})\text{initialize DECREMENTAL-SPECTRAL-SPARSIFY}(H_j, \varepsilon)$ 
     /*  $\mathcal{A}_j$  decrementally maintains the newly computed  $\tilde{H}_j$  */
15 Procedure DELETE( $e$ )
16   $j \leftarrow \text{index of the specific hypergraph among } H_1, \dots, H_n \text{ that contains } e$ 
17  Pass the deletion of  $e$  to  $\mathcal{A}_j$ 
18 Procedure OUTPUTSPARSIFIER()
19  return  $\tilde{H} = \bigcup_{i=1}^{i_{\text{last}}} \tilde{H}_i$ 

```

follows from [Lemma 2.1](#) that $\tilde{H} = \bigcup_{i=1}^k \tilde{H}_i$ is a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H .

Size of \tilde{H} : By assumption, each \tilde{H}_i has size $S(m_i, n, \varepsilon^{-1})$, which is upper bounded by $S(m, n, \varepsilon^{-1})$ since $m_i \leq m$ and S is a monotone non-decreasing function. Since $k = O(\log m)$, the size of \tilde{H} is bounded by $\sum_{i=1}^{\log m} S(m_i, n, \varepsilon^{-1}) = O(S(m, n, \varepsilon^{-1}) \log m)$.

Update time: After l insertions, H_i has been reinitialized for at most $l/2^i$ times. This is because H_i can contain at most $m_i = 2^i$ hyperedges. By assumption, the total time for the initialization and maintenance of \mathcal{A}_i is bounded by $m_i T(m_i, n, \varepsilon^{-1}) = 2^i T(m_i, n, \varepsilon^{-1})$, which is bounded by $2^i T(m, n, \varepsilon^{-1})$ since $m_i \leq m$ and S is a monotone non-decreasing function. Thus, the total time for the initialization and maintenance of \mathcal{A}_i throughout l insertions is $O(\frac{l}{2^i} 2^i T(m, n, \varepsilon^{-1})) = O(l T(m, n, \varepsilon^{-1}))$. Therefore, the total update time for maintaining $\tilde{H}_1, \dots, \tilde{H}_k$ is

$$O\left(\sum_{i=1}^{\log m} l T(m, n, \varepsilon^{-1})\right) = O(l T(m, n, \varepsilon^{-1}) \log m),$$

resulting in an $O(T(m, n, \varepsilon^{-1}) \log m)$ amortized update time.

High probability claim: After l insertions, each H_i is reinitialized $O(l)$ times. By [Lemma 3.5](#), the probability of failure is at most l/m^{c-2} , where $c \geq 3$. Since $k = \log m$, it follows that the

probability of failure for maintaining $\tilde{H}_1, \dots, \tilde{H}_k$ is at most $l \log m / m^{c-2}$. Since $l = \text{poly}(m)$, the high probability claim follows by choosing c to be large enough. \square

Proof of Theorem 1.1. Given an m -edge n -vertex hypergraph H of rank r , by Lemma 3.5, the decremental data structure (Algorithm 4) maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H of size $S(m', n, \varepsilon^{-1}) = O(n^2/\varepsilon^2 \log^6 m)$ in $T(m, n, \varepsilon^{-1}) = O(r^2 \log^2 m)$ amortized update time.

Since S and T are monotone non-decreasing functions, it follows from Lemma 3.3 that, for any hypergraph H containing at most m hyperedges at any point, the fully dynamic data structure (Algorithm 5) maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier of H of size $O(n^2/\varepsilon^2 \log^7 m)$ in $O(r^2 \log^3 m)$ amortized update time. \square

Acknowledgement

We thank the anonymous reviewers for their helpful suggestions.

SF: This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 947702).

References

- [ADKKP16] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. *On Fully Dynamic Graph Sparsifiers*. In: *57th FOCS*, pages 335–344, 2016. [1, 2, 4, 8, 11]
- [AGM13] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. *Spectral Sparsification in Dynamic Graph Streams*. In: *16th Workshop on Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX)*. Vol. 8096, pages 1–10, 2013. [2]
- [AMV21] Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. *Faster Sparse Minimum Cost Flow by Electrical Flow Localization*. In: *62nd FOCS*, pages 528–539, 2021. [2]
- [BBGNSSS22] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. *Fully-Dynamic Graph Sparsifiers Against an Adaptive Adversary*. In: *49th ICALP*. Vol. 229, 20:1–20:20, 2022. [2]
- [BGJLLPS22] Jan van den Brand, Yu Gao, Arun Jambulapati, Yin Tat Lee, Yang P. Liu, Richard Peng, and Aaron Sidford. *Faster maxflow via improved dynamic spectral vertex sparsifiers*. In: *54th STOC*, pages 543–556, 2022. [2]
- [BHV08] Erik G. Boman, Bruce Hendrickson, and Stephen A. Vavasis. *Solving Elliptic Finite Element Systems in Near-Linear Time with Support Preconditioners*. *SIAM J. Numer. Anal.* 46.6, pages 3264–3284, 2008. [1]
- [BK96] András A. Benczúr and David R. Karger. *Approximating s-t Minimum Cuts in $O(n^2)$ Time*. In: *28th STOC*, pages 47–55, 1996. [1]
- [BSS12] Joshua Batson, Daniel A. Spielman, and Nikhil Srivastava. *Twice-Ramanujan Sparsifiers*. *SIAM Journal on Computing* 41.6, pages 1704–1721, 2012. [2]
- [BSST13] Joshua D. Batson, Daniel A. Spielman, Nikhil Srivastava, and Shang-Hua Teng. *Spectral sparsification of graphs: theory and algorithms*. *Commun. ACM* 56.8, pages 87–94, 2013. [2]
- [BST19] Nikhil Bansal, Ola Svensson, and Luca Trevisan. *New Notions and Constructions of Sparsification for Graphs and Hypergraphs*. In: *60th FOCS*, pages 910–928, 2019. [2]
- [BV06] Yuri Boykov and Olga Veksler. *Graph Cuts in Vision and Graphics: Theories and Applications*. In: *Handbook of Mathematical Models in Computer Vision*, pages 79–96, 2006. [1]
- [CGHPS20] Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. *Fast Dynamic Cuts, Distances and Effective Resistances via Vertex Sparsifiers*. In: *61st FOCS*, pages 1135–1146, 2020. [2]
- [Che52] Herman Chernoff. *A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations*. *The Annals of Mathematical Statistics* 23.4, pages 493–507, 1952. [7]
- [CMP20] Michael B. Cohen, Cameron Musco, and Jakub Pachocki. *Online Row Sampling*. *Theory Comput.* 16, pages 1–25, 2020. [2]
- [DGGLPSY22] Sally Dong, Yu Gao, Gramoz Goranci, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Guanghao Ye. *Nested Dissection Meets IPMs: Planar Min-Cost Flow in Nearly-Linear Time*. In: *33rd SODA*, pages 124–153, 2022. [2]
- [DGGP19] David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. *Fully dynamic spectral vertex sparsifiers and applications*. In: *51st STOC*, pages 914–925, 2019. [2]
- [DS08] Samuel I. Daitch and Daniel A. Spielman. *Faster approximate lossy generalized flow via interior point algorithms*. In: *40th STOC*, pages 451–460, 2008. [1]
- [Fen18] Zhuo Feng. *Similarity-aware spectral sparsification by edge filtering*. In: *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018. [1]
- [FGLPSY21] Sebastian Forster, Gramoz Goranci, Yang P. Liu, Richard Peng, Xiaorui Sun, and Mingquan Ye. *Minor Sparsifiers and the Distributed Laplacian Paradigm*. In: *62nd FOCS*, pages 989–999, 2021. [2]

- [GHP18] Gramoz Goranci, Monika Henzinger, and Pan Peng. *Dynamic Effective Resistances and Approximate Schur Complement on Separable Graphs*. In: *26th ESA*. Vol. 112, 40:1–40:15, 2018. [2]
- [GK25] Mohsen Ghaffari and Jaehyun Koo. *Parallel Batch-Dynamic Algorithms for Spanners, and Extensions*. In: *37th SPAA*, pages 299–313, 2025. [2]
- [GLP21] Yu Gao, Yang P. Liu, and Richard Peng. *Fully Dynamic Electrical Flows: Sparse Maxflow Faster Than Goldberg-Rao*. In: *62nd FOCS*, pages 516–527, 2021. [2]
- [GM25] Gramoz Goranci and Ali Momeni. *Fully Dynamic Spectral Sparsification of Hypergraphs*. *CoRR* abs/2502.01421, 2025. [1, 5, 11]
- [JLS23] Arun Jambulapati, Yang P. Liu, and Aaron Sidford. *Chaining, Group Leverage Score Overestimates, and Fast Spectral Hypergraph Sparsification*. In: *55th STOC*, pages 196–206, 2023. [1, 2]
- [KKTY21a] Michael Kapralov, Robert Krauthgamer, Jakab Tardos, and Yuichi Yoshida. *Spectral Hypergraph Sparsifiers of Nearly Linear Size*. In: *62nd FOCS*, pages 1159–1170, 2021. [1, 2]
- [KKTY21b] Michael Kapralov, Robert Krauthgamer, Jakab Tardos, and Yuichi Yoshida. *Towards tight bounds for spectral sparsification of hypergraphs*. In: *53rd STOC*, pages 598–611, 2021. [1, 2]
- [KL13] Jonathan A. Kelner and Alex Levin. *Spectral Sparsification in the Semi-streaming Setting*. *Theory Comput. Syst.* 53.2, pages 243–262, 2013. [2]
- [KLMMS17] Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. *Single Pass Spectral Sparsification in Dynamic Streams*. *SIAM J. Comput.* 46.1, pages 456–477, 2017. [2]
- [KLP16] Ioannis Koutis, Alex Levin, and Richard Peng. *Faster Spectral Sparsification and Numerical Algorithms for SDD Matrices*. *ACM Trans. Algorithms* 12.2, 17:1–17:16, 2016. [2]
- [KLP25] Sanjeev Khanna, Huan Li, and Aaron Puterman. *Near-Optimal Linear Sketches and Fully-Dynamic Algorithms for Hypergraph Spectral Sparsification*. In: *57th STOC*, pages 1190–1200, 2025. [1, 2, 5, 11]
- [KM09] Jonathan A. Kelner and Aleksander Madry. *Faster Generation of Random Spanning Trees*. In: *50th FOCS*, pages 13–21, 2009. [1]
- [KMMMNST20] Michael Kapralov, Aida Mousavifar, Cameron Musco, Christopher Musco, Navid Nouri, Aaron Sidford, and Jakab Tardos. *Fast and Space Efficient Spectral Sparsification in Dynamic Streams*. In: *SODA*, pages 1814–1833, 2020. [2]
- [KMT11] Ioannis Koutis, Gary L. Miller, and David Tolliver. *Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing*. *Comput. Vis. Image Underst.* 115.12, pages 1638–1646, 2011. [1]
- [KNST19] Michael Kapralov, Navid Nouri, Aaron Sidford, and Jakab Tardos. *Dynamic Streaming Spectral Sparsification in Nearly Linear Time and Space*. *CoRR* abs/1903.12150, 2019. [2]
- [KP12] Michael Kapralov and Rina Panigrahy. *Spectral sparsification via random spanners*. In: *ITCS*, pages 393–398, 2012. [2]
- [KPS24] Sanjeev Khanna, Aaron (Louie) Puterman, and Madhu Sudan. *Almost-Tight Bounds on Preserving Cuts in Classes of Submodular Hypergraphs*. In: *51st ICALP*. Vol. 297, 98:1–98:17, 2024. [2, 5]
- [KT10] Pushmeet Kohli and Philip H. S. Torr. *Dynamic Graph Cuts and Their Applications in Computer Vision*. In: *Computer Vision: Detection, Recognition and Reconstruction*. Vol. 285. Studies in Computational Intelligence, pages 51–108, 2010. [1]
- [KX16] Ioannis Koutis and Shen Chen Xu. *Simple Parallel and Distributed Algorithms for Spectral Graph Sparsification*. *ACM Trans. Parallel Comput.* 3.2, 14:1–14:14, 2016. [2]
- [Lee23] James R. Lee. *Spectral Hypergraph Sparsification via Chaining*. In: *55th STOC*, pages 207–218, 2023. [1, 2]

- [LGT14] James R. Lee, Shayan Oveis Gharan, and Luca Trevisan. *Multiway Spectral Partitioning and Higher-Order Cheeger Inequalities*. *Journal of the ACM* 61.6, pages 1–30, 2014. [1]
- [LS17] Yin Tat Lee and He Sun. *An SDP-based algorithm for linear-sized spectral sparsification*. In: *49th STOC*, pages 678–687, 2017. [2]
- [LS18] Huan Li and Aaron Schild. *Spectral Subspace Sparsification*. In: *59th FOCS*, pages 385–396, 2018. [2]
- [Mad10] Aleksander Madry. *Fast Approximation Algorithms for Cut-Based Problems in Undirected Graphs*. In: *51st FOCS*, pages 245–254, 2010. [1]
- [MU17] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017. [7]
- [OST23] Kazusato Oko, Shinsaku Sakaue, and Shin-ichi Tanigawa. *Nearly Tight Spectral Sparsification of Directed Hypergraphs*. In: *50th ICALP*, 94:1–94:19, 2023. [1, 2, 3, 4, 5, 7, 8, 9]
- [Pen16] Richard Peng. *Approximate Undirected Maximum Flows in $O(m\text{polylog}(n))$ Time*. In: *27th SODA*, pages 1862–1867, 2016. [1]
- [PSZ17] Richard Peng, He Sun, and Luca Zanetti. *Partitioning Well-Clustered Graphs: Spectral Clustering Works!* *SIAM Journal on Computing* 46.2, pages 710–743, 2017. [1]
- [RY22] Akbar Rafiey and Yuichi Yoshida. *Sparsification of Decomposable Submodular Functions*. In: *36th Conference on Artificial Intelligence (AAAI)*, pages 10336–10344, 2022. [2]
- [She13] Jonah Sherman. *Nearly Maximum Flows in Nearly Linear Time*. In: *54th FOCS*, pages 263–269, 2013. [1]
- [SS08] Daniel A. Spielman and Nikhil Srivastava. *Graph sparsification by effective resistances*. In: *40th STOC*, 2008. [1, 2]
- [ST11] Daniel A. Spielman and Shang-Hua Teng. *Spectral Sparsification of Graphs*. *SIAM J. Comput.* 40.4, pages 981–1025, 2011. [1, 2]
- [ST14] Daniel A. Spielman and Shang-Hua Teng. *Nearly Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems*. *SIAM J. Matrix Anal. Appl.* 35.3, pages 835–885, 2014. [1]
- [STY24] Tasuku Soma, Kam Chuen Tung, and Yuichi Yoshida. *Online Algorithms for Spectral Hypergraph Sparsification*. In: *25th Conference on Integer Programming and Combinatorial Optimization (IPCO)*. Vol. 14679, pages 405–417, 2024. [2]
- [SY19] Tasuku Soma and Yuichi Yoshida. *Spectral Sparsification of Hypergraphs*. In: *30th SODA*, pages 2570–2581, 2019. [2]
- [Zha25] Yibin Zhao. *Fully Dynamic Spectral and Cut Sparsifiers for Directed Graphs*, 2025. [2]
- [ZLB21] Chunjiang Zhu, Qinqing Liu, and Jinbo Bi. *Spectral vertex sparsifiers and pair-wise spanners over distributed graphs*. In: *38th ICML*. Vol. 139, pages 12890–12900, 2021. [2]
- [ZLO15] Zeyuan Allen Zhu, Zhenyu Liao, and Lorenzo Orecchia. *Spectral Sparsification and Regret Minimization Beyond Matrix Multiplicative Updates*. In: *47th STOC*, pages 237–245, 2015. [2]

A Parallel Batch-Dynamic Spectral Sparsification

In this section, we present our batch parallel algorithm for maintaining a $(1 \pm \varepsilon)$ -spectral hypersparsifier \tilde{H} of a directed hypergraph $H = (V, E, \mathbf{w})$ undergoing hyperedge deletions and additions.

The algorithm is an extension of our fully dynamic sequential data structure ([Algorithm 5](#)). Recall that to maintain \tilde{H} in the fully dynamic setting, [Algorithm 5](#) decrementally maintains a sequence of sub-hypergraphs $H_1, \dots, H_{i_{\text{last}}}$ using [Algorithm 4](#), which itself builds upon [Algorithm 3](#). Therefore, to explain our fully dynamic extension, we begin by explaining how we implement [Algorithms 3](#) and [4](#) in the batch parallel setting.

A.1 Parallel Batch-Dynamic Implementation of [Algorithm 3](#).

We first analyze each step of the algorithm individually. In the end, we will combine them to obtain the guarantees of the algorithm.

The initialization process of [Algorithm 3](#) consists of finding a coresset C and a sampled hypergraph S . To handle this process, for every pair $u, v \in V$, we defined $E(u, v)$ to contain a hyperedge e iff $u \in t(e)$ and $v \in h(e)$. We explain in the claim below how to construct $E(\cdot, \cdot)$'s in parallel.

Claim A.1.1. *All sets $E(\cdot, \cdot)$ can be constructed in $O(mr^2 \log m)$ total work and $O(\log r)$ depth.*

Proof. Since each hyperedge e consists of $O(r)$ vertices, it can appear in at most r^2 of the sets $E(\cdot, \cdot)$'s. For every hyperedge e , we identify the corresponding sets $E(\cdot, \cdot)$'s and add e to them. Since there are initially m hyperedges and adding them to each $E(\cdot, \cdot)$ takes $O(\log m)$ work, the bound on the total work follows.

The bound on depth follows from the following observation. By definition, to construct $E(\cdot, \cdot)$'s, we need to find the pairs (u, v) for every hyperedge e with $u \in t(e)$ and $v \in h(e)$ and add e to $E(u, v)$. Although $t(e)$ and $h(e)$ have size at most r , the process can be divided into similar tasks on subsets of $t(e)$ and $h(e)$ with size at most $r/2$. More precisely, we can partition $t(e) = A \cup B$ and $h(e) = C \cup D$ so that A, B, C, D all have size at most $r/2$. To find the pairs associated with e , we can find the pairs in the four subcases corresponding to $(A, C), (A, D), (B, C), (B, D)$. Thus, $D(r) = 4D(r/2)$, which solves to $O(\log r)$ depth. \square

Once the $E(\cdot, \cdot)$'s are computed, the algorithm chooses λ heavyweight hyperedges from each $E(\cdot, \cdot)$ to be included in coresset C . Since we look for the heaviest hyperedges in each set, we sort them from heaviest to lightest so that they can be accessed more efficiently after an update. This leads to the following claim.

Claim A.1.2. *All sets $E(\cdot, \cdot)$ can be sorted in $O(mr^2 \log m)$ total work and $O(\log^2 m)$ depth.*

Proof. For each $E(\cdot, \cdot)$, we use the parallelized version of MERGESORT, which has $O(|E(\cdot, \cdot)| \log |E(\cdot, \cdot)|)$ work and $O(\log^2 |E(\cdot, \cdot)|)$ depth. Thus, the depth is $O(\log^2 m)$, and the total work is

$$O\left(\sum_{u,v \in V} |E(u, v)| \log |E(u, v)|\right) = O\left(\sum_{u,v \in V} |E(u, v)| \log m\right) = O(mr^2 \log m),$$

following from the fact that each $E(\cdot, \cdot)$ contains $O(m)$ hyperedges, and that the total number of hyperedges in $E(\cdot, \cdot)$'s is $O(mr^2)$. \square

As mentioned above, the coresset C is computed by choosing λ (if available) hyperedges in each $E(\cdot, \cdot) \setminus C$ and adding them to C . We discuss this process in the following claim.

Claim A.1.3. *The coresset C can be computed in $O(\lambda n^2 \log m)$ total work and $O(\log m)$ depth.*

Proof. The algorithm chooses at most λ hyperedges from each $E(\cdot, \cdot) \setminus C$. Since each $E(\cdot, \cdot)$ is sorted, the algorithm only needs to probe the set and add a hyperedge if it has not already been added to C by another processor. However, the algorithm might probe $O(\lambda n^2)$ of the hyperedges in $E(\cdot, \cdot)$, which have already been added to C by other processors. Thus, it takes $O(\lambda n^2 \log m + \lambda \log m) = O(\lambda n^2 \log m)$ total work for each $E(\cdot, \cdot)$ to add enough edges to C , where the $O(\log m)$ overhead is due to standard techniques for avoiding memory conflicts. \square

After computing the coresset hypergraph C , the sampled hypergraph S is computed by sampling each hyperedge in $H \setminus C$ with probability $1/2$ and doubling its weight. This process can simply be parallelized as explained in the following claim.

Claim A.1.4. *The sampled hypergraph S can be computed in $O(m)$ total work and $O(1)$ depth.*

Proof. Since the sampling is independent for each hyperedge, the algorithm can sample each of them separately. The guarantees follow from the fact that the sampling of each hyperedge can be done in $O(1)$ time and $O(1)$ depth. \square

We now proceed to explain how the algorithm handles deletions. After the deletion of a hyperedge e from H , if e was included in C , the algorithm attempts to substitute it with another hyperedge. This process translates to finding the specific set $E(u, v)$ that added e to C , and then finding a heaviest hyperedge in $E(u, v) \setminus C$ and adding it to C . In the following claim, we discuss how this can be done in parallel when there is a batch of k hyperedge deletion.

Claim A.1.5. *Each batch of k hyperedge deletions can be handled in $O(kr^2 \log m)$ amortized work and $O(\log m)$ depth.*

Proof. As discussed in the sequential algorithm, each process for substituting a deleted hyperedge consists of $O(1)$ amortized changes in C and S , each of which costs $O(\log m)$ amortized work. To keep the depth short, the algorithm must not choose an already deleted hyperedge as the substitution, as this could result in $O(k)$ iterations to find the substitution. To alleviate this issue, we first remove the k hyperedges from $E(\cdot, \cdot)$'s. This results in $O(kr^2 \log m)$ extra work since each hyperedge could be present in $O(r^2)$ sets, and it takes $O(\log m)$ to remove an element from the sets.

Since updating $E(\cdot, \cdot)$'s dominates the update time, the amortized work of $O(kr^2 \log m)$ follows. The bound on depth follows accordingly, as there is no dependency between the deletions. \square

To obtain the guarantees of the entire algorithm, we use the claims above and derive the following lemma. The proof follows directly from the proofs of the claims, so we will not repeat them.

Lemma A.1. *Given an m -edge n -vertex hypergraph $H = (V, E, \mathbf{w})$ of rank r undergoing batches of k hyperedge deletions, the parallel batch-dynamic implementation of [Algorithm 3](#) initiates and maintains the coresset hypergraph C and the sampled hypergraph S of H , with high probability, in $O(kr^2 \log m)$ amortized work and $O(\log^2 m)$ depth.*

A.2 Parallel Batch-Dynamic Implementation of Algorithm 4.

The data structure of [Algorithm 4](#) recursively initializes [Algorithm 3](#) on smaller hypergraphs for $i_{\text{last}} = O(\log m)$ times to build in the sequence of coresets $C_1, \dots, C_{i_{\text{last}}}$ and the sequence of sampled hypergraphs $S_1, \dots, S_{i_{\text{last}}}$. The initialization guarantees are discussed in the following claim.

Claim A.2.1. *Algorithm 4 can be initialized in $O(mr^2 \log m)$ total work and $O(\log^2 m)$ depth.*

Proof. We separately analyze the two subroutines involved in the initialization. Combining their guarantees results in $O(mr^2 \log m)$ total work and $O(\log^2 m)$ depth.

Constructing and sorting $E(\cdot, \cdot)$'s: By [Claims A.1.1](#) and [A.1.2](#), it takes $O(mr^2 \log m)$ total work and $O(\log^2 m)$ depth to construct and sort all $E(\cdot, \cdot)$'s. The algorithm then uses these sets to find the coreset hypergraphs $C_1, \dots, C_{i_{\text{last}}}$ and the sampled hypergraphs $S_1, \dots, S_{i_{\text{last}}}$.

Constructing the coreset and sampled hypergraphs: By the proof of [Lemma 3.2](#), with high probability, each $C_i \cup S_i$ consists of at most $3/4$ hyperedges compared to $C_{i-1} \cup S_{i-1}$. Combining this fact with [Claims A.1.3](#) and [A.1.4](#), it follows that the total time to construct the whole sequence is $O(\sum_{i=1}^{i_{\text{last}}} (3m/4)^i) = O(m)$. Since $C_i \cup S_i$ is constructed on top of $C_{i-1} \cup S_{i-1}$, by [Claims A.1.3](#) and [A.1.4](#) and the fact that $i_{\text{last}} = O(\log m)$, we conclude that the depth of this process is $O(\log^2 m)$. \square

Recall that [Algorithm 4](#) handles each hyperedge deletion by maintaining C_i and S_i for every $1 \leq i \leq i_{\text{last}}$, and by passing at most one hyperedge deletion from level i to level $i + 1$. Unfortunately, the inter-level hyperedge is level-dependent, and inherently sequential. The following claim discusses how the algorithm handles a batch of k hyperedge deletions.

Claim A.2.2. *Each batch of k hyperedge deletions can be handled in $O(kr^2 \log^2 m)$ amortized work and $O(\log^2 m)$ depth.*

Proof. By [Claim A.1.5](#), each batch deletion at level i can be handled in $O(kr^2 \log m)$ amortized work and $O(\log m)$ depth. Since each level depends on the deletion from the previous level, and each level handles at most k hyperedge deletions, it follows that the total work is $O(kr^2 \log^2 m)$, and the depth is $O(\log^2 m)$. \square

We summarize the guarantees of the entire algorithm in the lemma below. The proof follows directly from the claims above and is therefore omitted.

Lemma A.2. *Given an m -edge n -vertex hypergraph $H = (V, E, \mathbf{w})$ of rank r undergoing batches of k hyperedge deletions, the parallel batch-dynamic implementation of [Algorithm 4](#) initiates and maintains a $(1 \pm \varepsilon)$ -spectral hypersparsifier, with high probability, in $O(kr^2 \log^2 m)$ amortized work and $O(\log^2 m)$ depth.*

A.3 Parallel Batch-Dynamic Implementation of Algorithm 5.

Our batch parallel data structure is based on [Algorithm 5](#), where its subroutine ([Algorithm 4](#)) is replaced with its batch parallel implementation (explained in [Subsection A.2](#)). For brevity, we do not repeat them here and only highlight the differences in our implementation compared to [Algorithm 5](#).

Our batch parallel implementation handles deletions similarly to [Algorithm 5](#), but now distributes up to k hyperedge deletions (one batch) across the sub-hypergraphs $H_1, \dots, H_{i_{\text{last}}}$, instead of distributing only a single hyperedge deletion.

The insertions are handled similarly to the insertion process in [Algorithm 5](#). The only difference is that after each batch of insertions, we increment the timer t by the number of inserted hyperedges (in the binary format), rather than just one. The rest of the algorithm remains the same: we reinitialize the sub-hypergraph H_j to contain all the hyperedges from H_1, \dots, H_j , along with the newly inserted batch of hyperedges, where j is chosen based on t (see [Algorithm 5](#) for more details).

We conclude this section by proving [Theorem 1.2](#) below.

Proof of Theorem 1.2. The proof of correctness, size of \tilde{H} , and high probability claim directly follows from the proof of [Lemma 3.3](#) and are not restated here. Below, we prove how the data structure achieves an $O(kr^2 \log^3 m)$ amortized work and $O(\log^2 m)$ depth by separately analyzing batch deletions and insertions.

Handling batch deletions: We pass the deletion of hyperedges to their associated sub-hypergraph. By [Lemma A.2](#), each sub-hypergraph can handle a batch of k hyperedge deletions in $O(kr^2 \log^2 m)$ amortized work and $O(\log^2 m)$ depth. Since there are $O(\log m)$ sub-hypergraphs, and that there is no dependency between sub-hypergraphs, it follows that the fully dynamic data structure can handle the deletions in $O(kr^2 \log^3 m)$ amortized work and $O(\log^2 m)$ depth.

Handling batch insertions: Using a similar approach to that used in the update time analysis of [Lemma 3.3](#), it follows that our implementation would have $O(\log m)$ overhead in work. Combined with [Lemma A.2](#), this results in an $O(kr^2 \log^3 m)$ amortized work. Since the sub-hypergraphs partition H and there is no dependency between them, it follows that the depth remains $O(\log^2 m)$. \square