High-order Line Graphs of Non-uniform Hypergraphs: Algorithms, Applications, and Experimental Analysis

Xu T. Liu*† Jesun Firoz‡ Sinan Aksoy‡ Ilya Amburg‡
Andrew Lumsdaine†‡ Cliff Joslyn‡ Brenda Praggastis‡ Assefaw H. Gebremedhin†
*University of Washington †Washington State University ‡Pacific Northwest National Lab, USA
*{x0, al75}@uw.edu †{assefaw.gebremedhin}@wsu.edu ‡{{first name}.{last name}}@pnnl.gov

***Abstract*—Hypergraphs offer flexible and robust data representations for many applications, but methods that work directly on hypergraphs are not readily available and tend to be prohibitively expensive. Much of the current analysis of hypergraphs relies on first performing a graph expansion – either based on the nodes (clique expansion), or on the hyperedges (line graph) – and then running standard graph analytics on the resulting representative graph. However, this approach suffers from massive space complexity and high computational cost with increasing hypergraph size. Here, we present efficient, parallel algorithms to accelerate and reduce the memory footprint of higher-order graph expansions of hypergraphs. Our results focus on the hyperedge-based s-line graph expansion, but the methods we develop work for higher-order clique expansions as well. To the best of our knowledge, ours is the first framework to enable hypergraph spectral analysis of a large dataset on a single shared-memory machine. Our methods enable the analysis of datasets from many domains that previous graph-expansion-based models are unable to provide. The proposed s-line graph computation algorithms are orders of magnitude faster than state-of-the-art sparse general matrix-matrix multiplication methods, and obtain approximately $2-31\times$ speedup over a prior state-of-the-art heuristic-based algorithm for s-line graph computation.**

***Index Terms*—Hypergraphs, parallel hypergraph algorithms, line graphs, intersection graphs, clique expansion.**

I. Introduction

Hypergraph models are more natural representation than graphs for a broad range of systems—in biology, sociology, telecommunications, and physical infrastructures—involving *multi-way* relationships [3], [5], since graph models are limited to representing *pairwise* relationships. Mathematically, a **hypergraph** is a structure $\mathcal{H} = \langle V, E \rangle$, with a set $V = \{v_j\}_{j=1}^{n}$ of vertices, and an indexable family $E = \{e_i\}_{i=1}^{m}$ of hyperedges $e_i \subseteq V$. Hyperedges have different sizes $|e_i|$, possibly ranging from the singleton $\{v\} \subseteq V$ (distinct from the element $v \in V$) to the vertex set V. A hyperedge $e = \{u, v\}$ with $|e| = 2$ is the same as a graph edge. Indeed, all graphs $G = \langle V, E \rangle$ are hypergraphs: in particular, graphs are "2-uniform" hypergraphs, so that now $E \subseteq \binom{V}{2}$ and all $e \in E$ are unordered pairs with $|e| = 2$. An example hypergraph $\mathcal{H}$ is shown in Figure 1 on vertices $V = \{a, b, \ldots, f\}$ and edges $E = \{1 : \{a, b, c\}, 2 : \{b, c, d\}, 3 : \{a, b, c, d, e\}, 4 : \{e, f\}\}$.

A well-known method to study hypergraphs is to create a graph representation from the structure of the initial hypergraph using a graph expansion method such as the clique expansion [38]. The **clique expansion** replaces each hyperedge with a graph edge for each pair of vertices in the hyperedge. The information associated with hyperedges in the original hypergraph is lost in the new graph [23]. Moreover, the size of the newly-constructed graph with these expansion methods increases exponentially ([21], [14]), which can significantly limit the scalability and applicability of these techniques. For example, there are approx. 10.3 billion edges in the clique-expansion graph of the Friendster dataset and 54.5 billion edges in that of Orkut [14]. With billions of non-zero entries in the adjacency matrix of the clique-expansion graphs, processing these datasets is not possible on a single compute node.

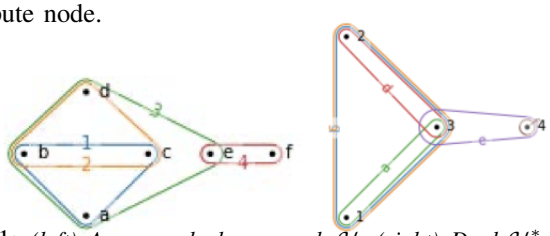


Fig. 1: *(left) An example hypergraph $\mathcal{H}$. (right) Dual $\mathcal{H}^*$ of the example hypergraph $\mathcal{H}$, defined later in Section II.*

In this work, we propose a scalable framework to study non-uniform hypergraphs with a lower-dimensional approximation of the original hypergraph called *s-line graphs* of a hypergraph. Our multi-stage, versatile framework starts from the original hypergraph, and consists of multiple stages, including pre-processing, s-line graph construction, squeezing the s-line graph, and s-measure (defined later) computation. An s-line graph construction considers the number of common (overlapping) vertices, denoted by s, between each pair of hyperedges to capture the strength of connections among hyperedges. Such a model can represent, for example, the strength of the collaboration in a collaboration network. Specifically, we are interested in this work with only high-order s-line graphs, where $s \geq 2$. Compared with the clique-expansion graphs, the s-line graph of Friendster only has 53 edges and that of Orkut has 4,289 edges for $s = 1024$. In an s-line graph, vertices (representing hyperedges of the original hypergraph) are connected when hyperedges intersect in at least s hypergraph vertices in the original hypergraph.

Dually, s-line graphs can also be constructed by considering the (hyper)vertices in the original hypergraph and their overlapping hyperedge sets. In this case, vertex s-line graph when $s = 1$ is the clique-expansion graph of a hypergraph.

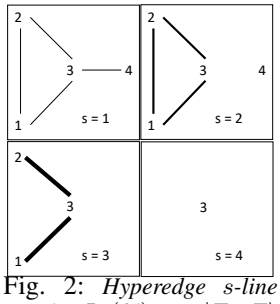


Fig. 2: *Hyperedge s-line graphs* $L_s(\mathcal{H}) = \langle E_s, F \rangle$ *for* $s = 1, 2, 3, 4$ *for the example in Figure 1. The width of the graph edges represents the strength of the connection in the original hypergraph.*

Figure 2 shows the hyperedge s-line graphs $L_s(\mathcal{H})$ for our example for $s = 1, 2, 3, 4$. Note the changing vertex sets E_s for each s value, decreasing to E_4={3} being the single hyperedge with $|e|$=5 ≥ 4. Throughout this paper, we refer s-line graphs as hyperedge s-line graphs.

The drastic difference in size between the clique-expansion graphs and the s-line graphs has implications in the adjacency matrix representations of the graphs. The size reduction entails drastic memory footprint reduction while computing a particular metric on the hypergraph (for example, when computing the Laplacian). Note that, in the s-line graph view of a hypergraph, as we vary the value of s, we can still retain the important connectivities in the original hypergraph.

A naive approach for the s-line graph construction is to find the intersection of the neighbor list of each pair of hyperedges in the original hypergraph. This is both compute- and memory-intensive. A recent parallel heuristic-based algorithm [30] significantly improves the performance over the naive approach by avoiding redundant set intersections. However, the approach is based on heuristics and can only compute one s-line graph at a time with one s value. Table I compares the performance of the algorithm presented in [30] with the method proposed in this work in terms of runtimes on LiveJournal dataset. As observed from the table, the s-line computation stage is the most time-consuming step in the pipeline. Hence, we propose two new (exact) parallel algorithms for s-line graph construction to reduce the overall execution time and improve the efficiency of the process. We apply our framework to different datasets and real-world problems to gain insights into its performance and utility.

We identify three additional motivations for computing s-line graphs of a hypergraph. First, once computed, **highly-tuned graph libraries** can be applied to the s-line graphs to measure different graph-theoretic metrics. The second motivation stems from **applications**, where hypergraphs and s-line graphs enable new insights based on s-line graph metrics. Third, s-line graphs enable **spectral graph analysis** of hypergraphs. To the best of our knowledge, there are no known method for directly computing the eigenvectors and eigenvalues of the rectangular incidence matrix of a hypergraph. The lack of a simple, eigenvalue-preserving algebraic relationship between the incidence matrix H of a hypergraph, and the adjacency matrices of s-line graphs suggests the existence of a method for implicitly determining the s-line graph spectrum without forming the s-line graph itself is highly unlikely. Eigenvalues can provide insight into, for example, how well each of the connected components in an s-line graph remains connected and consequently provide insight about the original hypergraph connectivity.

Stage	Algorithm in [30]	our method
preprocessing	0.122s	0.152s
s-overlap	**313.864s**	**12.085s**
squeeze	3.845s	2.656s
s-connected components	22ms	11ms
total time	329.520s	14.904s
speedup	1×	26×
#set intersections	8.66×10^9	0

TABLE I: *Computational cost of each step of the high-order line graph framework with the LiveJournal dataset [37]. Clearly, s-overlap computation (in bold) is the dominant stage in the process. Note that our method does not perform any set intersection operation.*

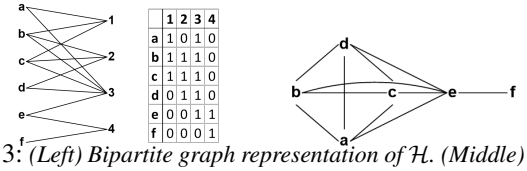


Fig. 3: *(Left) Bipartite graph representation of* $\mathcal{H}$*. (Middle) Incidence matrix* $(\mathcal{H})$*. (Right) 2-section* $\mathcal{H}_2$*.*

Summary of contributions. In this paper, we:

- Propose two new hashmap-based s-line graph computation algorithms that completely avoid set intersection operations and prove to be significantly faster than the state-of-the-art efficient algorithm (§III).
- Propose a (C++ based) high performance, scalable framework[1] for computing higher order line graph of hypergraphs (§IV).
- Apply our framework on three real-world problems: uncovering collaborations in co-authorship networks and in co-staring networks, and identifying important genes in transcriptomics data. We demonstrate both higher efficiency and practical usability (§V).
- Empirically analyze scalability of our framework on a variety of real-world datasets and show superior performance over the algorithm proposed in [30] (§VI). We also compare our approach with a state-of-the-art sparse matrix-matrix multiplication (SpGEMM) library-based implementation (§VI-G) and show superior performance.

II. Background

A. Hypergraph Representations

Hypergraphs may be represented in a number of equivalent forms. Given a hypergraph $\mathcal{H}$, one can construct the **bipartite graph** $B(\mathcal{H}) = \langle V \sqcup E, E' \rangle$ whose vertex set is the disjoint union of the hypergraph's vertices V and hyperedges E, and whose edge set is the undirected graph edges $E' \subseteq V \sqcup \binom{E}{2}$, where $\{v, e\} \in E'$ iff $v \in e$. Further, one can construct the Boolean **incidence matrix** $\mathbf{H}_{n \times m}$ where for $i \in [n], j \in [m]$, $b_{ij} = 1$ if $v_i \in e_j$, otherwise $b_{ij} = 0$. Note that $\mathbf{H}$ is not square. These two representations are illustrated in Figure 3 for the example hypergraph introduced in Figure 1.

The **dual hypergraph** $\mathcal{H}^* = \langle E^*, V^* \rangle$ of $\mathcal{H}$ has vertex set $E^* = \{e_i^*\}_{i=1}^m$ and family of hyperedges $V^* = \{v_j^*\}_{j=1}^n$, where $v_j^* := \{e_i^* : v_j \in e_i\}$. The dual $\mathcal{H}^*$ for our example is shown in Figure 1. $\mathcal{H}^*$ is just the hypergraph with the transposed incidence matrix H^T, and $(\mathcal{H}^*)^* = \mathcal{H}$.

In graphs, the structural relationship between two distinct

[1]Forthcoming code for our framework NWHypergraph will, pending institutional approval, be posted at https://github.com/pnnl/NWHypergraph

vertices u and v can *only* be whether they are adjacent in a *single* edge ($\{u, v\} \in E$) or not ($\{u, v\} \notin E$); and dually, that between two distinct edges e and f can *only* be whether they are incident at a *single* vertex ($e \cap f = \{v\} \neq \emptyset$) or not ($e \cap f = \emptyset$). In hypergraphs, both of these concepts are applicable to *sets* of vertices and edges, and additionally become *quantitative*. Define $\text{adj}\colon 2^V \to \mathbb{Z}_{\geq 0}$ and $\text{inc}\colon 2^E \to \mathbb{Z}_{\geq 0}$, in both set notation and (polymorphically) pairwise:

$$\text{adj}(U) = |\{e \supseteq U\}|, \quad \text{adj}(u, v) = |\{e \supseteq \{u, v\}\}|$$
$$\text{inc}(F) = |\cap_{e \in F} e|, \quad \text{inc}(e, f) = |e \cap f|$$

for $U \subseteq V, u, v \in V, F \subseteq E, e, f \in E$. These concepts are dual, in that adj on vertices in $\mathcal{H}$ maps to inc on edges in $\mathcal{H}^*$, and *vice versa*. And for singletons, $\text{adj}(\{v\}) = \deg(v) = |e \ni v|$ is the degree of the vertex v, while $\text{inc}(\{e\}) = |e|$ is the size of the edge e. In our example, we have $\text{adj}(b, c) = 3$, while $\text{inc}(\{1, 2, 3\}) = 2$.

B. Hypergraph Measures and s-Line Graphs

Two edges $e, f \in E$ are **s-incident** if $\text{inc}(e, f) = |e \cap f| \geq s$ for $s \geq 1$. An **s-walk** is a sequence of edges $\langle e_0, e_1, \ldots, e_n \rangle$ such that each e_{i-1}, e_i are s-incident for $1 \leq i \leq n$. An **s-path** is an s-walk where no edges are repeated.

Aksoy *et al.* have developed various s-line graph metrics on the basis of s-walks [2]. Here, we describe two of the metrics used in our paper. Let $E_s := \{e \in E : |e| \geq s\}$. The **$s$-betweenness centrality** of a hyperedge e is $\sum_{f \neq g \in E_s} \frac{\sigma^s_{fg}(e)}{\sigma^s_{fg}}$, where $\sigma^s_{fg}(e)$ is the total number of shortest s-walks from hyperedge f to g and σ^s_{fg} is the number of those shortest s-walks that contain hyperedge e. A subset of hyperedges $F \subseteq E_s$ is an **s-connected component** if there is an s-walk between all edges $e, f \in F$, and F is a maximal such subset. These measures have important applications in hypernetwork science. For example, Feng *et al.* apply s-betweenness centrality to analyze biological datasets [10].

Consider the **2-section** $\mathcal{H}_2 = \langle V, F \rangle$ of a hypergraph $\mathcal{H}$ as a graph on the same vertex set V, but now with edges $F \subseteq \binom{V}{2}$ such that $\{u, v\} \in F$ iff there is some hyperedge $e \in E$ with $\{u, v\} \subseteq e$ (see Figure 3). Thus $\mathcal{H}_2$ can be thought of as a kind of "underlying graph" of a hypergraph $\mathcal{H}$.

Also of key interest is the 2-section of the dual hypergraph $\mathcal{H}^*$, called the **line graph** $L(\mathcal{H}) = (\mathcal{H}^*)_2$. Note that the vertices in $L(\mathcal{H})$ are the hyperedges in E, and two such (now) vertices $e, f \in E$ are connected with a line graph edge iff $\text{inc}(e, f) > 0$. In general, for integer $s \geq 1$, define the **s-line graph** of a hypergraph $\mathcal{H}$ as a graph $L_s(\mathcal{H}) = \langle E_s, F \rangle$ where $F \subseteq \binom{E}{2}$ and $\{e, f\} \in F$ iff e and f are s-incident. It is known that in general, a hypergraph $\mathcal{H}$ cannot always be reconstructed from even *all* of the s-line graphs $L_s(\mathcal{H})$ *together with* the s-line graphs $L_s(\mathcal{H}^*)$ of the dual [23]. Nonetheless, Aksoy *et al.* have demonstrated that all of the above measures *can* be calculated from the s-line graphs $L_s(\mathcal{H})$. 1-line graphs are also known as *intersection graphs* or *one-mode projections*.

s-line graphs can be naively calculated from the incidence matrix $\mathbf{H}$, specifically, $\mathbf{L} := \mathbf{H}^\top \mathbf{H}$ is an $m \times m$ symmetric integer **weighted adjacency matrix**, where each cell $\mathbf{L}[i, j], i, j \in [m]$, records $\text{inc}(e_i, e_j)$, and the diagonal entries

Algorithm 1 Algorithm proposed in [30] to compute the edge list of an s-line graph for a given s.
Input: Hypergraph $\mathcal{H} = (V, E)$, s
Output: s-line graph edge list $L_s(\mathcal{H})$

```
1: L_s(H) ← ∅
2: L_t(H) ← ∅, for each thread t
3: for all hyperedge e_i ∈ E do in parallel
4:     for each vertex v_k of e_i do
5:         for each hyperedge e_j of v_k where (i < j) do
6:             count ← set_intersection(neighbor_list(e_i), neighbor_list(e_j))
7:             if count ≥ s then
8:                 L_t(H) ← L_t(H) ∪ {e_i, e_j}
9: L_s(H) ← L_s(H) ∪ every L_t(H)
10: return L_s(H)
```

$\mathbf{L}[i, i]$ record edge size $\text{inc}(\{e_i\}) = |e_i|$. For integer $s \geq 1$, define a Boolean filtration matrix $\mathbf{L}_s$ where $\mathbf{L}_s[i, j] = 1$ if $\mathbf{L}[i, j] \geq s$, and 0 otherwise. Then $\mathbf{L}_s - I$ is the adjacency matrix of L_{s+1}.

III. Algorithms for Constructing s-line Graphs

In this section, we start by briefly discussing a previous state-of-the-art algorithm for the s-line graph computation [30] and derive the linear-algebraic equivalent formulation of the algorithm. We next transition to the linear algebraic formulation of our new algorithm and present our parallel s-line graph and ensemble s-line graph computation algorithms. Additionally, we discuss the design and implementation details of our parallel algorithms. We conclude the section with discussion about the distinctions between our algorithm and SpGEMM-based approach, the relationship of s-line graph with the weighted clique-expansion graph and the practicality of the s-line graph. Crucially, our methods also enable scalable analysis of higher-order clique expansions, but for the purpose of this work we mostly frame our language around, and present results for, s-line graph computations.

A. Previous Approaches

Recently, Liu *et al.* proposed an algorithm [30] (shown in Algorithm 1), where only the pair of hyperedges with at least one common neighbor is considered for the s-line graph computation. Additional heuristics have been applied to reduce the amount of redundant work. These heuristics include degree-based pruning, skipping already visited hyperedges, short-circuiting set intersection and considering either the upper or the lower triangular part of the adjacency matrix of the s-line graphs. The proposed algorithm, in conjunction with these heuristics, achieves notable performance benefit over the naive approach. While Algorithm 1 improved the execution time of the s-line graph computation, performing explicit all-pairs set intersections despite incorporating different heuristics may still be computationally inefficient.

B. Linear Algebraic Formulation of Our Algorithms

Our approach exploits the linear algebraic relationships present in the adjacency matrix $\mathbf{L} = \mathbf{H}^\top \mathbf{H}$. There are two basic variants to consider to construct $\mathbf{L}$, which differ based

on loop ordering. In the first case, we consider the "ijk" loop ordering, where the inner loop is essentially a dot product between column i and column j of $\mathbf{H}$, that is, an intersection between the non-zero locations of those two rows:

1: **for** $i = 0, 1, \ldots$ **do**
2: **for** $j = 0, 1, \ldots$ **do**
3: **for** $k = 0, 1, \ldots$ **do**
4: $\mathbf{L}[i, j] \leftarrow \mathbf{L}[i, j] + \mathbf{H}[k, i]\mathbf{H}[k, j]$
5: $\mathbf{L}_s \leftarrow$ Boolean filtration on $\mathbf{L}$ based on s

An alternative ordering of the loops in matrix multiply interchanges the two inner loops.

1: **for** $i = 0, 1, \ldots$ **do**
2: **for** $k = 0, 1, \ldots$ **do**
3: **for** $j = 0, 1, \ldots$ **do**
4: $\mathbf{L}[i, j] \leftarrow \mathbf{L}[i, j] + \mathbf{H}[k, i]\mathbf{H}[k, j]$
5: $\mathbf{L}_s \leftarrow$ Boolean filtration on $\mathbf{L}$ based on s

In this case, the intersection is not so obvious. The inner loop copies row k of $\mathbf{H}$, scaled by element $\mathbf{H}[k, i]$, to row i of $\mathbf{L}$. The "intersection" now is implicit in whether $\mathbf{H}[k, i]$ is zero or non-zero. (In numerical linear algebra terminology, the inner loop is an "axpy," or vector addition, operation.)

If we were to carry out this operation with actual matrices, the two forms would be computationally equivalent. However, we are carrying out this computation with graph structures, which are best represented as sparse matrices. A computation using the graph structure, corresponding to the "ijk" ordering is given as

1: **for** $i = 0, 1, \ldots$ **do**
2: **for** $j = 0, 1, \ldots$ **do**
3: $\mathbf{L}[i, j] \leftarrow \mathbf{L}[i, j] + |\mathbf{H}.Adj[i] \cap \mathbf{H}.Adj[j]|$
4: $\mathbf{L}_s \leftarrow$ Boolean filtration on $\mathbf{L}$ based on s

$\mathbf{H}.Adj[i]$ indicates all vertices k adjacent to vertex i in $\mathbf{H}$, so that adj$(v_i, v_k) > 0$. Note that this form compares all pairs of vertices, which may be highly redundant if $\mathbf{H}$ is sparse.

The alternative "ikj" formulation instead allows us to exploit the structure of the graph.

1: **for** $i = 0, 1, \ldots$ **do**
2: **for** $k \in \mathbf{H}^\top.Adj[i]$ **do**
3: **for** $j \in \mathbf{H}.Adj[k]$ **do**
4: $\mathbf{L}[i, j] \leftarrow \mathbf{L}[i, j] + 1$
5: $\mathbf{L}_s \leftarrow$ Boolean filtration on $\mathbf{L}$ based on s

Here, rather than computing intersections between all pairs, we accumulate intersecting edges as we traverse the hypergraph.

C. Our Hashmap-based Algorithm to Compute a s-line Graph

Based on the above observation, in contrast to performing an explicit set intersection between the full neighbor lists of both e_i and e_j (Line 6 in Algorithm 1), our new algorithm (Algorithm 2) only counts the common neighbor v_k (Line 9 in Algorithm 2). The new algorithm maintains a running count of the amount of overlaps between e_i and e_j observed so far. This is reminiscent of counting "confirmed" common members (v_k) between e_i and e_j, instead of "searching" for common memberships between two neighbor lists of e_i and e_j.

Algorithm 2 Our algorithm to compute the edge list of an s-line graph for a given s using a hashmap data structure.
Input: Hypergraph $\mathcal{H} = (V, E)$, s
Output: s-line graph edge list $L_s(\mathcal{H})$

1: $L_s(\mathcal{H}) \leftarrow \emptyset$
2: $L_t(\mathcal{H}) \leftarrow \emptyset$, for each thread t
3: **for all** hyperedge $e_i \in E$ **do in parallel**
4: **if** $degree[e_i] < s$ **then** ▷ Degree-based pruning
5: **continue**
6: overlap_count $\leftarrow []$
7: **for each** vertex v_k of e_i **do**
8: **for each** hyperedge e_j of v_k where $(i < j)$ **do**
9: overlap_count$[e_j]$++
10: **for each** $[e_j, n] \in$ overlap_count **do**
11: **if** $n \geq s$ **then**
12: $L_t(\mathcal{H}) \leftarrow L_t(\mathcal{H}) \cup \{e_i, e_j\}$
13: $L_s(\mathcal{H}) \leftarrow L_s(\mathcal{H}) \cup$ every $L_t(\mathcal{H})$
14: **return** $L_s(\mathcal{H})$

Algorithm 3 Our algorithm to compute the edge lists of an ensemble of s-line graphs using hashmap data structures.
Input: Hypergraph $\mathcal{H} = (V, E)$, $array\_s$
Output: s-line graph edge lists $L_{s_i}(\mathcal{H}), \forall s_i \in array\_s$

1: overlap_count $\leftarrow \{\}$
2: $s \leftarrow$ smallest $s \in array\_s$
3: **for all** hyperedge $e_i \in E$ **do in parallel**
4: **if** $degree[e_i] < s$ **then**
5: **continue**
6: overlap_count$[e_i] \leftarrow []$
7: **for each** vertex v_k of e_i **do**
8: **for each** hyperedge e_j of v_k where $(i < j)$ **do**
9: overlap_count$[e_i][e_j]$++
10: **for all** $s_i \in array\_s$ **do in parallel**
11: $L_{s_i}(\mathcal{H}) \leftarrow \emptyset$
12: **for each** hyperedge $e_i \in E$ **do**
13: **for each** $[e_j, n] \in$overlap_count$[e_i]$ **do**
14: **if** $n \geq s_i$ **then**
15: $L_{s_i}(\mathcal{H}) \leftarrow L_{s_i}(\mathcal{H}) \cup \{e_i, e_j\}$
16: **return** $L_{s_i}(\mathcal{H}), \forall s_i \in array\_s$

To keep track of the running count, the algorithm allocates a hashmap data structure for each hyperedge e_i (Line 6 in Algorithm 2) on the fly, with 2-hop neighbors e_j as keys and the current overlap count of (e_i, e_j) as the values. The algorithm still considers only the set of edge pairs (e_i, e_j) with at least one common neighbor (v_k) (Lines 3–8 in Algorithm 2) and these wedges are considered only from one direction $(i < j)$. We also apply degree-based pruning heuristic to filter out the set of hyperedges with degree $< s$ from the computation, as they are not members of E_s.

D. Computing Ensemble of s-line Graphs

Occasionally, we need to compute an ensemble of s-line graphs, instead of a single one, for different values of s. In this scenario, running algorithm 2 multiple times to generate s-line graphs separately may be inefficient. Hence, to compute an ensemble of s-line graphs, we modify algorithm 2 to first

accumulate and store the overlap counts, and then filter out edge-pairs based on a particular s value. The modified algorithm is shown in Algorithm 3. Since multiple s-line graphs will be constructed, instead of the in-place insertion of edges (e_i, e_j) with s overlapping neighbors in the s-line graph's edge list (Line 12 in algorithm 2), we decouple this insertion step from the counting step. The algorithm maintains a running count of overlaps for each pair of hyperedges (e_i, e_j) (Line 9 in algorithm 3). Once the counting step is completed, for each value of s, the algorithm loops through the hashmap containing all $(e_j, counts)$ pairs for each e_i and construct the edge list of the s-line graph (Lines 10–15 in Algorithm 3). Degree-based pruning can be applied to filter out the hyperedges with degree smaller than the smallest s in $array\_s$. To avoid duplicate counting for a pair of edges (e_i, e_j), we prune redundant computation related to edge (e_j, e_i).

E. Parallel Time Complexity Analysis

We analyze the complexity of Algorithm 2 and Algorithm 3 in the work-depth model [18]. The work W is equal to the total number of independent computations. The depth D is equal to the time required for the critical path computation (in the computation DAG, the longest chain of dependency). If P processors are available, with a randomized work-stealing scheduler, Brent's scheduling principle dictates that the running time is $O(W/P + D)$. Each hyperedge is visited once on the outermost loop ($|E|$). Without considering any heuristics, the second inner loop visits $\overline{d}_v$ number of incident hypernodes on average. The innermost loop visits $\overline{d}_e$ incident hyperedges on average. Because lookup and insertion of elements in a hashmap is constant on average, therefore, Algorithm 2 takes $O(|E|\overline{d}_v\overline{d}_e)$ on average, and $O(|V||E|^2)$ time in the worst case. The overall work is $O(|V||E|^2)$, and overall depth is $O(log|H|)$. Here $|H|$ denotes the number of non-zero entries in the hypergraph incidence matrix. Algorithm 3 has the same time complexity as Algorithm 2. Next we consider degree-based pruning and considering only the upper triangular part of the adjacency matrix $\mathbf{L}_s(e_i, e_j$ pairs with $i < j)$. The degree-based pruning trims the work in outermost loop to E_s. Considering only the upper triangle of the adjacency matrix $\mathbf{L}_s$ essentially cuts the overall work by half.

F. Parallel Implementation Design Considerations

We implement our framework in `C++20`. Since s-line graph computation is the most compute-intensive stage in the pipeline, we parallelize our algorithms to compute the s-line graphs in Stage 3. For this purpose, we leverage the parallel constructs available in Intel oneAPI Threading Building Blocks (oneTBB) [17]. In particular, the outermost for loops iterating over the hyperedges in Algorithm 2 and Algorithm 3 are parallelized with the `parallel_for` construct in oneTBB. `parallel_for`, in the form of `(range, body, partitioner)`, allows different ranges to be passed in to enable partitioning the range (hyperedges) in different ways so that different workload distribution strategies among the threads can be tested, as long as the provided range meets the `C++` range requirements.

Ranges and Partitioning strategies. oneTBB provides a built-in range, namely *blocked range*, where the hyperedges (IDs) can be divided into blocks (chunks) and each chunk of contiguous hyperedges (IDs) can be assigned to one thread. Additionally, we adopt an alternative, customized range, namely *cyclic range*. Here, given the stride size equal to the number of total threads nt, thread 0 processes hyperedges $e_0, e_{0+nt}, e_{0+2*nt}, e_{0+3*nt}$ and so on, thread 1 processes hyperedges $e_1, e_{1+nt}, e_{1+2*nt}, e_{1+3*nt}$ and so on. Here e_i denotes a hyperedge ID. oneTBB is based on work-stealing runtime scheduler. Work stealing scheduler is particularly beneficial in our context, since this enables idle threads to steal work from other straggler threads, which are currently processing, for example, high-degree hyperedges.

Granularity Control. To accommodate flexibility for load balancing, oneTBB also provides provision for specifying the *granularity* of work done by each thread, while reducing the overheads of work stealing and task scheduling. We leverage this fine-grained control to specify the block size of the chunk of work (i.e. the number of hyperedges assigned to each thread). We notice that chunk size up to 256 achieves similar performance. With larger chunk sizes, the scheduling overhead noticeably impacts algorithm performance.

Data Structures for the Main Performance Criterion (Overlap Count). The hashmap data structures for maintaining the overlap_counts in our algorithms are thread-local data structures, implemented with the C++ `std::unordered_map`. In Algorithm 3, for example, each hyperedge is associated with a hashmap that maintains a list of neighbors with at least one overlapping vertex. Before applying filtering (s), the size of each of these individual hashmap is equal to the degree of each hyperedge. With hypergraphs with skewed-degree distribution, s-line computation may have hashmaps for which the sizes vary significantly.

Consideration of dynamic vs pre-allocated thread-local storage: We have observed that pre-allocated thread-local storage (TLS) (i.e. per-thread hashmap allocated outside of the outermost for loop and resetting it after each iteration) may be beneficial for computing s-line graphs with hypergraphs with denser overlapping neighbor sets for each pair of hyperedges. Web dataset, discussed in Section VI, is one such example. For a particular s value, Web generates denser s-line graph. Dynamically allocating and deallocating a hashmap in each iteration on-the-fly inside the outermost for loop is costlier in this case. All other datasets, however, prefer dynamically-allocated hashmap for each thread in each iteration.

G. Relationship among Our Hashmap-based s-line Graph Algorithm, Algorithm 1 and Sparse Matrix-Matrix Multiplication (SpGEMM).

When constructing a single s-line graph for a particular s value, considering the pairs of hyperedges sharing at least one common node is equivalent to computing the sparse general matrix-matrix multiplications (SpGEMM) [13] followed by a filter operation to find the edgelist of an s-line graph. However, the SpGEMM-based approach is both time-consuming and memory-intensive. There are three reasons why it is not efficient for computing s-line graphs. First, it considers both

the upper triangular and the lower triangular of hyperedge adjacency matrix $\mathbf{L}_s$ even though the matrix is symmetric. In contrast, our algorithm can exploit this symmetry to consider either the upper or the lower triangular part of the matrix. Second, since SpGEMM is more general, it has to compute and store the product matrix before applying filtration upon the matrix. This requires extra space to store the intermediate results (i.e., the product matrix). Our algorithm, on the other hand, can apply the filtration operation on-the-fly and does not require to materialize the product matrix due to the known s value. Third, the SpGEMM-based approach cannot apply other heuristics to speedup the computation, such as degree-based pruning (prune all the hyperedges with degree $< s$) or short circuit the set intersection as applied in Algorithm 1. We report the performance comparison of our algorithms with a state-of-the-art parallel SpGEMM library in Section VI-G.

H. Relation to the (Weighted) Clique-expansion Graph

Given a hypergraph $\mathcal{H}$ with incidence matrix $\mathbf{H}$, we can compute the weighted clique-expansion adjacency matrix as $\mathbf{W} = \mathbf{H}\mathbf{H}^T - \mathbf{D}_V$ where $\mathbf{D}_V$ is a diagonal matrix with node degrees as its diagonal entries. It is easy to see that $\mathbf{W}[i,j]$ is the number of hyperedges nodes i and j appear together in. Note that we can use $\mathbf{W}$ to obtain $L_s(H^*)$ for every integer $s \geq 1$ through its adjacency matrix $\mathbf{L}_s^*$. We set $\mathbf{L}_s^*[i,j] = 1$ if $\mathbf{W}[i,j] \geq s$ and 0 otherwise. However, the above procedure would be very memory-intensive as $\mathbf{W}$ can be very dense.

This observation implies that we could use our approach to efficiently compute s-sections, or "s-clique" graphs, where a graph edge connects two nodes if the nodes appear together in a hyperedge at least s times, bypassing memory limitation issues by not having to explicitly compute $\mathbf{W}$. In particular, this could be accomplished by running our algorithm to directly compute $L_s(\mathbf{H}^*)$ for a given s. So in other words, the s-line graph problem is dual to the s-clique problem. Although we frame our paper through the s-line graph perspective, it is crucial to note that the tools we develop apply equally well to the s-clique graph problem. The choice of which perspective to take depends on whether one wants to investigate edge- (s-line graph) or node- (s-clique graph) centric properties, and on the particular application.

I. Motivation for using higher-order graph expansions

A widespread approach to hypergraph analysis is to focus instead on associated graph projections, such as the clique expansion. As discussed in Section III-H, our framework actually includes the clique expansion as a special case: the s-line graph of the dual hypergraph (i.e. s-clique graph) is the graph obtained by linking vertices in the hypergraph whenever they belong to s or more shared hyperedges. In this way, the 1-line graph of the dual hypergraph is the clique expansion. Compared to the clique expansion approach, there are significant, practical benefits afforded by the s-clique approach, for $s > 1$.

In particular, s-clique graphs can reduce the density of graph projections while preserving – or even amplifying – essential features of the network. Line graphs (or clique expansions) of hypergraph-structured data tend to be prohibitively dense because a single high degree vertex (resp., large hyperedge) yields quadratically many edges. For instance, in an author-paper hypergraph, a single paper with many authors (i.e. large hyperedge) links all pairs of those authors, whereas for $s > 1$, the s-clique graph approach requires *more than one* joint paper to link those authors in the collaboration graph.

Disease	Rank & Score Percentile		
	$s = 1$	$s = 10$	$s = 100$
Malignant neoplasm of breast	1 (100%)	1 (100%)	1 (100%)
Breast carcinoma	2 (99.99%)	2 (99.99%)	2 (99.99)
Malignant neoplasm of prostate	3 (99.97%)	4 (99.96%)	4 (99.96%)
Liver carcinoma	4 (99.96%)	3 (99.97%)	3 (99.98%)
Colorectal cancer	5 (99.95%)	5 (99.95%)	6 (99.94%)

TABLE II: *Ordinal rank and score percentile of the top 5 diseases by PageRank score in the clique expansion (i.e. $s = 1$), as well as the s-line graphs of the dual hypergraph (s-clique expansion), for $s = 10, 100$.*

In practice, we find the density of s-clique graphs drops off exponentially in s in data sets from far-ranging domains. In log-log scale, Figure 4 plots the number of edges in s-clique graphs against s for disGeNet (a disease-gene dataset [36]), condMat (an author-paper network from the condensed matter section of the arXiv [35]), compBoard (a board member-company network from [2]), and lesMis (a character-scene network derived in [24] from Victor Hugo's Les Miserables). While the rates of decrease differ across datasets, s-clique graphs rapidly sparsify as s increases. For larger datasets, the formation of the clique expansion is intractable; s-clique graphs provide an alternative in these cases.

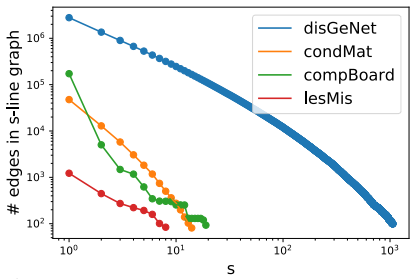


Fig. 4: *The number of edges in the s-clique graph of four datasets*

Even when s-clique graph formation is feasible for $s = 1$, focusing on $s > 1$ may be sufficient or preferable for a number of basic analytic tasks. While this of course is data and question dependent, we illustrate the potential effectiveness of this approach for one common analytical task: centrality and ranking. In biology, hypergraphs have been utilized to identify structurally critical genes and diseases in interactome networks [12]. Returning to the disease-gene network, we construct the clique expansion (linking diseases associated with common genes), compute the PageRank score of the diseases, and compare this to the PageRank rankings of diseases in the s-clique graphs, for $s = 10$ and $s = 100$. Table II presents how the top 5 ranked diseases in the clique expansion ($s = 1$) are ranked in the $s = 10$ and $s = 100$ higher-order clique expansions. These three graphs are of vastly different densities, having 2.7M, 246K, 12K edges, respectively. Nonetheless, the ordinal rankings and score percentiles for the top 5 rated diseases are nearly identical across all three graphs. Extending to the top 400 diseases – which constitute those above 95% percentile of scores – shows that 92% and 88% of these diseases remain in

the top 400 for $s = 10$ and $s = 100$, respectively. In this case, the higher-order s-clique graph approach identifies essentially the same critical diseases according to their PageRank using a network with 231 times fewer edges than the clique expansion.

IV. Our s-line Graph Computation Framework

We now discuss our s-line graph framework for non-uniform hypergraphs in detail. The framework has five major stages, two of which are at least partially optional, depending on the needs of a particular data set and problem.

Stage-1 Pre-processing. Pre-processing hypergraph includes removing isolated vertices, empty edges, and relabeling.

Relabeling. Large hypergraphs with highly-skewed, non-uniform degree distributions generally benefit from relabeling the hyperedge IDs according to their degrees (henceforth referred to as *relabel-by-degree*). Let's consider a "wedge" motif (e_i, v_k, e_j) in the bipartite graph hypergraph form $B(\mathcal{H})$. When counting the common neighbor v_k, to avoid considering v_k twice: once in view of (e_i, v_k, e_j) and another as (e_j, v_k, e_i), all s-line computation algorithms include a comparison $(i < j)$, so that the "wedge" is traversed only once (Line 5 in Algorithm 1, Line 8 in Algorithm 2 and Line 8 in Algorithm 3). This is equivalent to considering only the upper triangular part of the adjacency matrix $\mathbf{L}_s$.

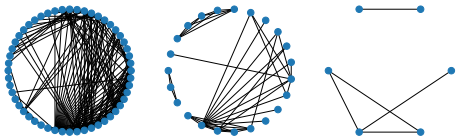
Relabel-by-degree in ascending order, in conjunction with considering the upper triangular part of $\mathbf{L}_s$, may improve the performance of the algorithm. Additionally, this helps achieve better load balancing among threads while executing a parallel s-line graph computation algorithm in the later stage. Equivalently, relabel-by-degree in *descending* order, in conjunction with considering the *lower triangular part* of $\mathbf{L}_s$, may provide similar performance improvement.

Stage-2 (optional) Computing toplexes. We calculate the toplexes $\check{E}$, and thereby the simplified hypergraph $\check{\mathcal{H}}$. A **toplex** is a maximal hyperedge e such that there exits no hyperedge f where $\nexists f \supseteq e$. Let $\check{E} \subseteq E$ be the set of all toplexes. For a hypergraph $\mathcal{H}$, $\check{\mathcal{H}} = \langle V, \check{E} \rangle$ is the **simplification** of $\mathcal{H}$, and $\mathcal{H}$ is **simple** when $\mathcal{H} = \check{\mathcal{H}}$, so that all hyperedges are toplexes. A simplification may result in significantly smaller $\check{\mathcal{H}}$, which, in turn, reduce the memory footprint of subsequent stages. Efficient algorithms for computing toplexes [31] are available.

Stage-3 Computation of the edge list of the s-line graph of a given hypergraph. The most important and compute-intensive stage of the s-line graph framework involves construction of the s-line graph itself. Depending on the requirement, the objective of this stage can be two-fold: the computation of *only one* s-line graph for a particular s value or an *ensemble* of s-line graphs for different values of s. Computation of an ensemble of s-line graphs is more memory-intensive in comparison to just computing a single s-line graph. We discuss in detail two algorithms for computing individual and ensemble of line graphs in the next section.

Stage-4 ID squeezing (optional) and s-line graph construction. After we finish computing the edge list of the s-line graphs, many hyperedge pairs may not be included in the newly-constructed s-line graph due to insufficient overlap between their vertex sets. Hence, the adjacency matrix of the s-line graph may be *hypersparse* (many rows will be empty when considering s-overlap). Retaining the original IDs of the hyperedges to construct the new s-line graph will thus be wasteful in terms of memory. Hence, optionally, we may remap the IDs to a contiguous space to eliminate the "holes" in the ID space of the s-line graph. This stage is called *ID squeezing*. The s-line graph is constructed based on the generated edge list.

Stage-5 s-metric computation. Once the s-line graph is constructed, different s-line graph metrics are computed, including s-connected components, s-centrality, s-distance, etc. When computing these metrics, any standard, relevant graph algorithm can be applied to compute such metrics.



(a) $s = 1$ (b) $s = 3$ (c) $s = 5$

Fig. 5: *Line graphs computed from the virology genomics data [10]. They are plotted using NetworkX in Shell layout. The six most important genes in the original hypergraph are identified by the 5-line graph, which are ISG15, IL6, AFT3, RSAD2, USP18 and IFIT1.*

V. Real-world Applications

In this section, we illustrate the utility of our framework using three real-world applications: identifying the most important genes in a transcriptomics data, revealing strong co-authorships among authors, and uncovering collaboration networks among actors on Internet Movie Database (IMDB).

A. Identifying Genes Critical to Pathogenic Viral Response

Though graph models are quite successful in biological data modeling, they have limitations in representing complex relationships amongst entities. In biology, hypergraphs can be used to model gene and protein interaction networks. Here we construct a hypergraph from the virology genomics data [10], where there are 9760 hyperedges representing genes, and 201 vertices representing individual biological samples with specific experimental "conditions" (e.g., mouse lung cells treated with a strain of Influenza virus and sampled at 8 hours). We omit the details of extracting the hypergraphs from the dataset due to space constraint.

To identify important genes in this hypergraph, we compute the s-connected components and the s-betweenness centrality scores of the vertices within each s-connected component. Figure 5 shows these s-line graphs. As s increases, the important genes are clearly identifiable in the visualization. In particular, gene IFIT1 and USP18 have the highest centrality scores, implying that they are the two most important genes. They share more than 100 vertices between them. This indicates that IFIT1 and USP18 are both perturbed in over 100 experimental conditions at the same time. Our s-line graphs clearly reveal the strength of the connections of those two genes that previous graph-based models are unable to deduce.

B. Revealing Relationships Among Authors

For certain hypergraph analytics, the formation of an ensemble of s-line graphs is strictly necessary. To illustrate a particular type of analysis that necessitates s-line graph construction, we construct a hypergraph from the condensed matter author-paper network in Los Alamos e-Print Archive [35]. This hypergraph contains 16,726 authors as vertices, 22,016 papers as hyperedges, and 58,595 author-paper inclusions.

To reveal the relationships among authors in this network, we compute an ensemble of s-line graphs where s ranges from 1 to 16 (16 is the max s that produces non-singleton components). We compute the normalized algebraic connectivity of the s-line graphs of author-paper dataset. Normalized algebraic connectivity is the second-smallest eigenvalue of the normalized Laplacian matrix[11], [7]; larger values imply stronger connectivity properties of the s-line graph and hence the hypergraph.

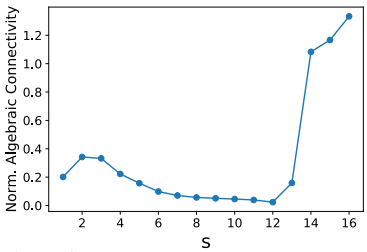


Fig. 6: *Normalized algebraic connectivity for condensed matter author-paper network.*

As observed from Figure 6, decreasing values of algebraic connectivity from s=3 to s=12 reveals that many authors collaborate on papers only sparsely, meaning the vertices (authors) within a connected component are sparsely connected with each other. However, the sharp increase in algebraic connectivity starting from s=13 demonstrates the fact that authors who have co-authored at least in 13 papers are more likely to collaborate with each other (signified by the denser connections within a connected component of an s-line graph). In this way, eigenvalues can provide insight into how well each of the connected components in an s-line graph remains connected and consequently provide insight about the original hypergraph connectivity. In addition, as the s value grows, these techniques can assist in understanding how well the connectivity is preserved.

C. Uncovering Collaborations Among Actors

Consider uncovering groupings of actors who have collaborated on at least s movies. We can query this information from Internet Movie Database (IMDB) by constructing a hypergraph (where the movies are vertices, and actors are hyperedges), and computing the s-line graphs. We compute s-connected components and s-betweenness centrality on these s-line graphs. We start by working on three database tables from the database: `title.basic`, `name.basic` and `title.principals` [16]. These tables contain approx. 11 million titles, approx. 8 million actor names, and approx. 18 million principal cast/crew for titles respectively.

The three collaboration networks that we uncovered within IMDB are reported below. Only the actors having a non-zero centrality scores are shown. These actors collaborated in more than 100 movies together:

```
(compute s-connected components) 4 us
Here are the 100-connected components:
[Adoor Bhasi, Bahadur, Paravoor Bharathan, Jayabharati,
Prem Nazir], [Matsunosuke Onoe, Suminojo],
[Kijaku Ôtani, Kitsuraku Arashi],[Panchito, Dolphy].
(compute s-betweenness centrality) 15 us
Adoor Bhasi(0.1111), Matsunosuke Onoe(0.0111),
Kijaku Ôtani(0.0111) //normalized score
```

We observe that, for the network in which Adoor Bhasi is a member, he has a centrality score of 0.11, while others have a score of 0. This means that Adoor Bhasi is the most important actor. Specifically, this network is a star graph where Adoor is the center vertex because all the other actors have a zero centrality score. Previous multigraph-formulation approach implemented in Python to compute betweenness centrality along took 10 hours on a Windows 10 machine (a 3.2 GHz CPU with 8 GB RAM) [29]. On the other hand, our implementation took a total of 80ms to execute on a Mac Mini (M1 chip, with 16GB RAM) to compute the 100-line graph, 100-connected components and 100-betweenness centrality.

VI. Experimental Analysis

In this section, we evaluate the performance of our s-line graph algorithms in comparison with the algorithms proposed in [30] and an efficient SpGEMM algorithm. We also discuss the scalability, workload characteristics and evaluation of the workload balancing techniques of our proposed algorithms. Table III summarizes the shorthand notations we use for different algorithms with different workload distribution strategies.

A. Experimental Setup

Our experiments are run on a machine with a two-socket Intel Xeon Gold 6230 processor, having 20 physical cores per socket, each running at 2.1 GHz, and 28 MB L3 cache. The system has 188 GB of main memory. Our code is implemented in `C++20`, parallelized with Intel oneTBB 2020.3, and compiled with GCC 10.2 compiler and `-Ofast -march=native` compilation flags.

B. Dataset

We conducted experiments with real-world hypergraphs (Table IV) from various domains, ranging from social to cyber to web. The activeDNS (ADNS) dataset from Georgia Institute of Technology contains mappings from domains to IP addresses [26]. When constructing hypergraphs with ADNS dataset, we consider the domains as the hyperedges and IPs as vertices. Additionally, we ran our experiments with datasets curated in [37]. For these curated datasets, in particular, each hypergraph, constructed from the social network datasets such as com-Orkut and Friendster in Table IV, are materialized by running a community detection algorithm on the original dataset obtained from Stanford Large Network Dataset Collection (SNAP) [28]. In the resultant hypergraphs, each community is considered as a hyperedge and each member of a community as a vertex. Other larger datasets include Web, and LiveJournal, collected from Koblenz Network Collection (KONECT) [25] as bipartite graphs.

Additionally, we selected two large datasets: Amazon-reviews [1] (where hyperedges are sets of product reviews on Amazon, and nodes are product categories) and Stackoverflow-answers [1] (where hyperedges are sets of questions and

nodes are the tags for questions answered by users on Stack Overflow).

Notation	Algo.	Partitioning	Relabel-by-degree
$1BA$	Algo. 1	Blocked	Ascending
$1BD$	Algo. 1	Blocked	Descending
$1BN$	Algo. 1	Blocked	No
$1CA$	Algo. 1	Cyclic	Ascending
$1CD$	Algo. 1	Cyclic	Descending
$1CN$	Algo. 1	Cyclic	No
$2BA$	Algo. 2	Blocked	Ascending
$2BD$	Algo. 2	Blocked	Descending
$2BN$	Algo. 2	Blocked	No
$2CA$	Algo. 2	Cyclic	Ascending
$2CD$	Algo. 2	Cyclic	Descending
$2CN$	Algo. 2	Cyclic	No

TABLE III: *Notation for different algorithms with different partitioning techniques and relabel-by-degree ordering.*

Type	hypergraph	$\|V\|$	$\|E\|$	$\overline{d}_v$	$\overline{d}_e$	Δ_v	Δ_e
Social	com-Orkut	2.3M	15.3M	46	7	3k	9.1k
	Friendster	7.9M	1.6M	3	14	1.7k	9.3k
	LiveJournal	3.2M	7.5M	35	15	300	1.1M
Web	Web	27.7M	12.8M	5	11	1.1M	11.6M
	Amazon-reviews	2.3M	4.3M	32	17	29k	9.4k
	Stackoverflow-answers	1.1M	15.2M	2	24	356	61.3k
Cyber	activeDNS	4.5M	43.9M	11	1	714.6k	1.3k
Email	email-EuAll	265.2k	265.2k	2	2	7.6k	930

TABLE IV: *Input characteristics. The number of vertices ($|V|$) and hyperedges ($|E|$) along with the average degree ($\overline{d}$), and maximum degree (Δ) for the hypergraph inputs are tabulated here. All the hypergraphs have a skewed hyperedge degree distribution.*

C. Performance Analysis

In Figure 7, we report the performance of different algorithms listed in Table III. The execution time for each algorithm is normalized w.r.t. 1CN (Algorithm 1 with cyclic distribution and no relabeling). Here, we do not report results of Algorithm 3, as it fails on most of the datasets (except for email-EuAll) due to its memory limitation.

As observed from Figure 7, our algorithm (Algorithm 2), in conjunction with the right combination of workload distribution strategy and relabel-by-degree, performs best and achieves $\approx 2\times - 31\times$ speedup for Web, and LiveJournal datasets. Larger inputs with skewed degree distribution (containing a handful of high-degree hyperedges) perform best when run with 2BA (Algorithm 2 with blocked distribution and hyperedges relabeled by degrees in ascending order). Interestingly, relabeling the hyperedges based on their degrees (both ascending and descending) does not provide drastic performance benefit for Friendster, Amazon-reviews and Stackoverflow-answers. These 3 datasets have smaller maximum degrees (Δ_e). Hence, relabel-by-degree does not provide significant benefit in improving the performance. In this case, the additional overhead of relabeling the hyperedges based on degrees heavily penalizes the execution time (we included the pre-processing time to relabel by degree in the total execution time).

D. Strong Scaling

We conducted strong scaling experiments for our algorithms with different hypergraph inputs and we report the results in Figure 8. Here we double the number of threads while keeping the input size constant. The performance of the algorithms improves up to 16 threads. Beyond 16 threads, performance does not improve significantly. For inputs with highly-skewed degree distribution (LiveJournal, com-Orkut, Web), 2CA demonstrates best scaling behaviour, as cyclic distribution enables better load balancing. Both block and cyclic distributions without relabeling achieve similar performance.

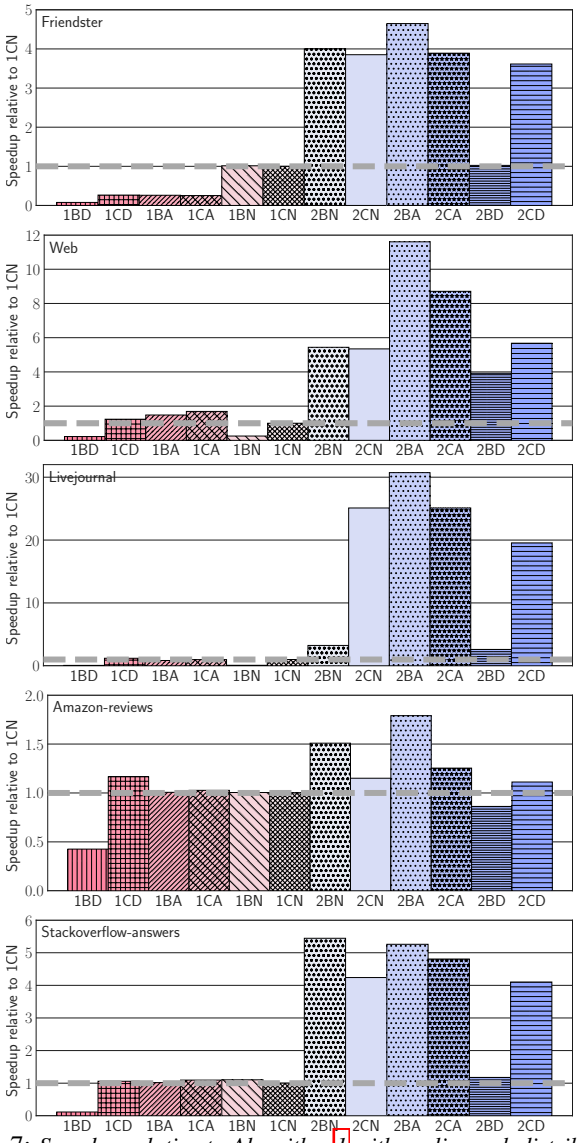


Fig. 7: *Speedup relative to Algorithm 1 with cyclic work distribution (1CN) where $s = 8$.*

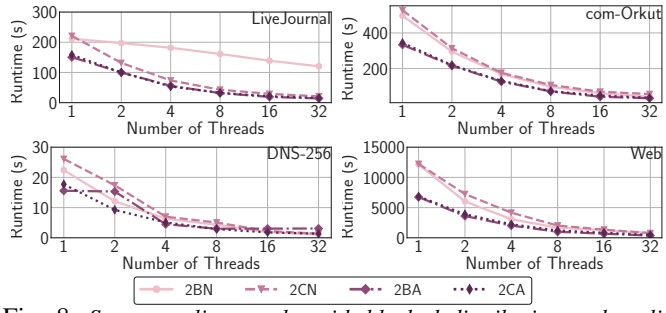


Fig. 8: *Strong scaling results with blocked distribution and cyclic distribution for Algorithm 2 when $s = 8$.*

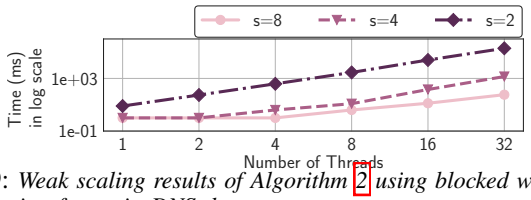


Fig. 9: *Weak scaling results of Algorithm 2 using blocked workload distribution for activeDNS dataset.*

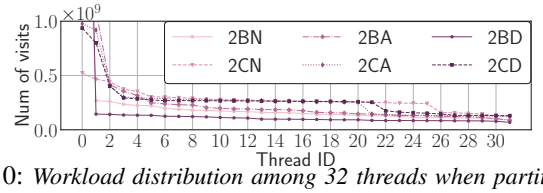


Fig. 10: *Workload distribution among 32 threads when partitioning the hyperedges (outermost loop of the s-line graph algorithms) in a blocked or cyclic manner in Algorithm 2 for LiveJournal input.*

E. Weak Scaling

We performed weak scaling experiments of Algorithm 2 with the activeDNS dataset using blocked workload distribution strategy. Here we approximately double the size of the hypergraph (workload) as we double the number of threads (computing resources). We start with 4 AVRO files worth of data (dns_4) and scale up to 128 files (dns_128). With larger s values, the performance of the algorithms improves (Figure 9).

F. Workload Characterization

Figure 10 shows the number of hyperedges visited by each thread in the innermost loop of Algorithm 2 with different partitioning strategies for LiveJournal dataset. As can be observed from Figure 10, without relabel-by-degree, cyclic distribution achieves better workload balance than blocked distribution. We also observe in Figure 7 that blocked or cyclic distribution, in conjunction with relabeling by degree in ascending order, performs best overall. We investigated this observation in details with Intel VTune Profiler and found out that relabel-by-degree in ascending order provides more favorable cache reuse (due to almost 0.5x less LLC cache misses) to Algorithm 2 than the descending order.

G. Comparison with an SpGEMM-based Approach

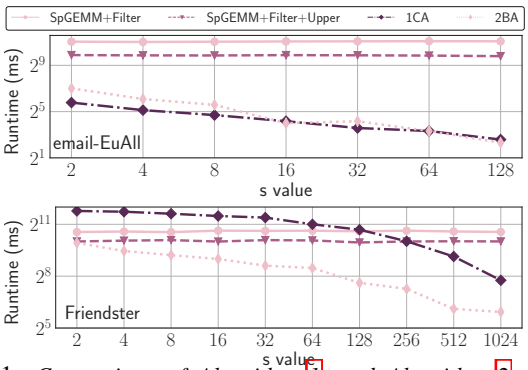


Fig. 11: *Comparison of Algorithm 1, and Algorithm 2 with an SpGEMM-based approach. Here SpGEMM+Filter+Upper refers to only consider the upper triangular part of the adjacency matrix.*

We also compare the performance of our hashmap-based algorithms and Algorithm 1 with a state-of-the-art SpGEMM-based library [33]. We modified the SpGEMM code to add the filtration step, and to only consider the upper triangular part of the matrix. Here, the SpGEMM library first computes HH^T, and then filters the edges with at least s overlaps. We report the results with email-EuAll and Friendster datasets. The SpGEMM library fails to run on other larger hypergraph datasets. The results are reported in Figure 11. With all datasets and for different s values, Algorithm 2 runs faster than the SpGEMM+Filter+Upper algorithm. The efficient algorithm (Algorithm 1) runs faster than the SpGEMM+Filter+Upper algorithm with the email-EuAll dataset, but slower than the SpGEMM+Filter+Upper algorithm with Friendster dataset (for smaller s values). With larger s values in all cases, our algorithm is orders of magnitude faster than the SpGEMM+Filter+Upper approach. The improvement can be attributed to the degree-based pruning. Note that computation of the s-line graphs with higher s values ($s = 1024$ for Friendster here) is still relevant, because, even with such a large s overlap constraint, we found 20 connected components in the constructed s-line graph. This reveals that these 20 communities which share at least 1024 common members are the core of Friendster dataset.

Both the efficient and our hashmap-based algorithm are more suitable than off-the-shelf SpGEMM algorithm for the s-line graph computation. The SpGEMM algorithm is too general since it has to compute and store the product matrix before applying filtration upon the matrix. In contrast, our algorithm performs an in-place filtration. In addition, the SpGEMM+Upper algorithm performs half of the total work by only considering the upper triangular part of the hyperedge adjacency matrix. However, it is still orders of magnitude slower than our algorithm (especially with larger s values).

H. Comparison with the Clique-expansion Approach

In Table V, we report the performance results of Algorithm 2 when s=1 (the clique expansion graph) and s=8 on larger datasets. We ran the Label Propagation-based Connected Components (LPCC) after computing the s-line graphs with Algorithm 2 (2CA). With s=1, only Friendster and Livejournal datasets completed execution on a 128GB-memory machine.

	Friendster	LiveJournal	com-Orkut	Web
s=1	12s	76s	OOM	OOM
s=8	4s	31s	59s	1510s

TABLE V: *Execution time of s=1 (clique expansion)-based and s-line graph-based with s=8 Label-Propagation Connected Components (LPCC) with Algorithm 2 (2CA). With s=1, com-Orkut and Web ran out of memory on a 128GB machine. The reported time includes end-to-end execution time of our framework.*

VII. Related Work

Hypergraph methods are well known for their applications in computer science; for example, hypergraph partitioning enables parallel matrix computations [8] and application in VLSI [22]. In the network science literature, researchers have devised several path and motif-based hypergraph data analytics measures such as clustering coefficients and centrality metrics [9]. Although an expanding body of research attests to the utility of hypergraph-based analyses [4], [15] , and we are seeing increasingly wide adoption [19], [27], [32], many network science methods have been historically developed explicitly for graph-based analyses. Naik [34] wrote a survey

on theoretical developments on line graphs. Bermond *et al.* [6] studied the properties of the s-line graphs of hypergraphs.

Shared-memory C++-based framework Hygra [37], and distributed-memory frameworks such as Chapel-based CHGL [20], Apache Spark-based MESH [14] and HyperX [21] presented a collection of efficient parallel algorithms for hypergraphs in their frameworks. These frameworks either rely on the original hypergraph or the expansion graphs of hypergraphs. None of the works computes s-line graphs with $s > 1$ and therefore cannot compute the s-walk measures. Moreover, in MESH/HyperX, on 8 compute nodes, a Label-Propagation-based Connected Component algorithm with clique expansion takes more than 2000s. In contrast, our framework takes ≈6s for the same computation, on a single-node.

VIII. Conclusion

The notion of s-line graphs of a hypergraph is a novel way to interpret relationships among different entities in a given dataset. In this paper, we have presented a scalable s-line graph computation framework by identifying a core set of stages required for end-to-end s-metric computation. We proposed new parallel algorithms for s-line graph computations and explored different workload distribution strategies for our parallel algorithms in conjunction with considering relabel-by-degree and triangularization of the adjacency matrix as optimization techniques. We demonstrated that our algorithms outperform current state-of-the-art algorithms. In particular, hypergraphs with skewed-degree distribution can benefit from relabeling the hyperedge IDs by degrees. We showed that proper combination of algorithmic optimization and workload balancing technique can significantly improve the performance of the s-line graph computation stage, which is the most important and compute-intensive part of the framework.

Acknowledgement. This work was partially supported by the High Performance Data Analytics (HPDA) program at the Department of Energy's Pacific Northwest National Laboratory, and by the NSF awards IIS-1553528 and SI2-SSE 1716828. PNNL Information Release: PNNL-SA-167812. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

References

[1] "Austin R. Benson Datasets." [Online]. Available: https://www.cs.cornell.edu/~arb/data/
[2] S. G. Aksoy, C. Joslyn, C. O. Marrero, B. Praggastis, and E. Purvine, "Hypernetwork science via high-order hypergraph walks," *EPJ Data Science*, vol. 9, no. 1, p. 16, 2020.
[3] A.-L. Barabási, *Network Science*. Cambridge University Press, 2016.
[4] F. Battiston, G. Cencetti, I. Iacopini, V. Latora, M. Lucas, A. Patania, J.-G. Young, and G. Petri, "Networks beyond pairwise interactions: structure and dynamics," *Physics Reports*, 2020.
[5] C. Berge, *Graphs and hypergraphs*. North-Holland, 1973.
[6] J.-C. Bermond, M.-C. Heydemann, and D. Sotteau, "Line graphs of hypergraphs I," *Discrete Mathematics*, vol. 18, no. 3, pp. 235–241, 1977.
[7] F. Chung, *Spectral graph theory*. American Mathematical Soc., 1997.
[8] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Int'l Par. and Distri. Proc. Symp.* IEEE, 2006, p. 124.
[9] E. Estrada and J. A. Rodríguez-Velázquez, "Subgraph centrality and clustering in complex hyper-networks," *Physica A: Statistical Mechanics and its Applications*, vol. 364, pp. 581–594, 2006.
[10] S. Feng, E. Heath, B. Jefferson *et al.*, "Hypergraph models of biological networks to identify genes critical to pathogenic viral response," *BMC Bioinformatics*, vol. 22, no. 1, p. 287, 2021.
[11] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak Mathematical Journal*, vol. 23, no. 2, pp. 298–305, 1973.
[12] K.-I. Goh, M. E. Cusick, D. Valle, B. Childs, M. Vidal, and A.-L. Barabási, "The human disease network," *Proceedings of the National Academy of Sciences*, vol. 104, no. 21, pp. 8685–8690, 2007.
[13] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, p. 250–269, 1978.
[14] B. Heintz, R. Hong, S. Singh, G. Khandelwal, C. Tesdahl, and A. Chandra, "MESH: A flexible distributed hypergraph processing system," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 12–22.
[15] I. Iacopini, G. Petri, A. Barrat, and V. Latora, "Simplicial models of social contagion," *Nature Communications*, vol. 10, no. 1, p. 2485, 2019.
[16] IMDB Interfaces. [Online]. Available: https://www.imdb.com/interfaces/
[17] Intel Threading Building Blocks (TBB), 2021. [Online]. Available: https://github.com/oneapi-src/oneTBB
[18] J. Jaja, *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
[19] M. A. Javidian, Z. Wang, L. Lu, and M. Valtorta, "On a hypergraph probabilistic graphical model," *Annals of Mathematics and Artificial Intelligence*, vol. 88, no. 9, pp. 1003–1033, 2020.
[20] L. Jenkins, T. Bhuiyan, S. Harun, C. Lightsey, D. Mentgen *et al.*, "Chapel hypergraph library (chgl)," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.
[21] W. Jiang, J. Qi, J. X. Yu, J. Huang, and R. Zhang, "HyperX: A scalable hypergraph framework," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, pp. 909 – 922, 2019.
[22] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," *VLSI Design*, vol. 11, no. 3, pp. 285–300, 2000.
[23] S. Kirkland, "Two-mode networks exhibiting data loss," *J Complex Networks*, vol. 6:2, pp. 297–316, 2017.
[24] D. E. Knuth, *The Stanford GraphBase: a platform for combinatorial computing*. ACM, 1993, vol. 1.
[25] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd Intl. Conference on World Wide Web*, 2013, pp. 1343–1350.
[26] A. Lab, "Active DNS project," 2020. [Online]. Available: https://activednsproject.org/
[27] N. W. Landry and J. G. Restrepo, "The effect of heterogeneity on hypergraph contagion models," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 30, no. 10, p. 103117, 2020.
[28] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection; 2014," *http://snap. stanford. edu/data*, 2016.
[29] R. Lewis, "Who is the centre of the movie universe? using python and networkx to analyse the social network of movie stars," *CoRR*, vol. abs/2002.11103, 2020.
[30] X. T. Liu, J. Firoz, A. Lumsdaine, C. Joslyn, S. Aksoy, B. Praggastis, and A. H. Gebremedhin, "Parallel algorithms for efficient computation of high-order line graphs of hypergraphs," in *2021 IEEE 28th Int. Conf. on High Perf. Comput., Data, and Analytics (HiPC)*, 2021, pp. 312–321.
[31] M. Marinov, N. Nash, and D. Gregg, "Practical algorithms for finding extremal sets," *Journal of Experimental Algorithmics (JEA)*, vol. 21.
[32] M. Minas, "Hypergraphs as a uniform diagram representation model," ser. TAGT'98. Springer-Verlag, 1998, p. 281–295.
[33] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "Sparse General Matrix-Matrix Multiplication for multi-core CPU and Intel KNL ," 2020.
[34] R. N. Naik, "On intersection graphs of graphs and hypergraphs: A survey," *arXiv preprint arXiv:1809.08472*, 2018.
[35] M. E. J. Newman, "The structure of scientific collaboration networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 98, no. 2, pp. 404–409, 2001.
[36] J. Piñero, J. M. Ramírez-Anguita, J. Saüch-Pitarch, F. Ronzano *et al.*, "The disgenet knowledge platform for disease genomics: 2019 update," *Nucleic acids research*, vol. 48, no. D1, pp. D845–D855, 2020. [Online]. Available: https://www.disgenet.org
[37] J. Shun, "Practical parallel hypergraph algorithms," in *Proc. of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 232–249.
[38] J. Y. Zien, M. D. Schlag, and P. K. Chan, "Multilevel spectral hypergraph partitioning with arbitrary vertex sizes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 9, pp. 1389–1399, 1999.