Parallel Algorithms for Efficient Computation of High-Order Line Graphs of Hypergraphs

Xu T. Liu *†, Jesun Firoz†, Andrew Lumsdaine†‡, Cliff Joslyn†,
Sinan Aksoy†, Brenda Praggastis†, Assefaw H. Gebremedhin*
*Washington State University, †Pacific Northwest National Lab, ‡University of Washington, USA
*{xu.liu2, assefaw.gebremedhin}@wsu.edu, †{{first name}.{last name}}@pnnl.gov

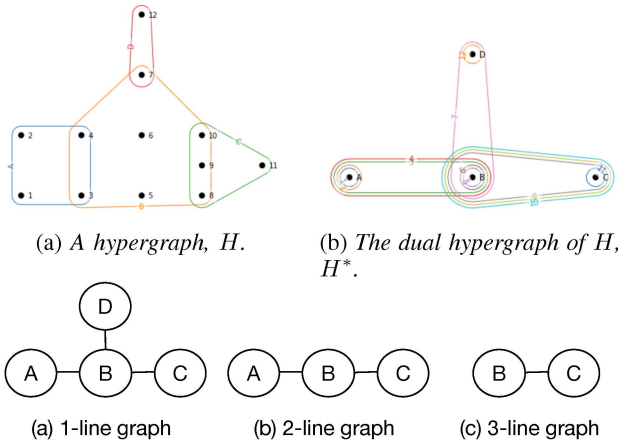
2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00045

***Abstract*—This paper considers structures of systems beyond dyadic (pairwise) interactions and investigates mathematical modeling of multi-way interactions and connections as hypergraphs, where captured relationships among system entities are set-valued. To date, in most situations, entities in a hypergraph are considered connected if there is at least one common "neighbor". However, minimal commonality sometimes discards the "strength" of connections and interactions among groups. To this end, considering the "width" of a connection, referred to as the *s-overlap* of neighbors, provides more meaningful insights into how closely the communities or entities interact with each other. In addition, s-overlap computation is the fundamental kernel to construct the line graph of a hypergraph, a low-order approximation of the hypergraph which can carry significant information about the original hypergraph. Subsequent stages of a data analytics pipeline then can apply highly tuned graph algorithms on the line graph to reveal important features. Given a hypergraph, computing the s-overlaps by exhaustively considering all pairwise entities can be computationally prohibitive. To tackle this challenge, we develop efficient algorithms to compute s-overlaps and the corresponding line graph of a hypergraph. We propose several heuristics to avoid execution of redundant work and improve performance of the s-overlap computation. Our parallel algorithm, combined with these heuristics, is orders of magnitude (more than $10\times$) faster than the naive algorithm in all cases and the SpGEMM algorithm with filtration in most cases (especially with large s value).**

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

I. Introduction

Graph-theoretical mathematical abstractions represent entities of interest as *vertices* connected by *edges* between pairs of them. Classical graph models benefit from simplicity and a degree of universality. But as abstract mathematical objects, graphs are limited to representing *pairwise* relationships between entities, whereas real-world phenomena in these systems can be rich in *multi-way* relationships involving interactions among more than two entities, dependencies between more than two variables, or properties of collections of more than two objects. Representing *group* interactions is not possible in graphs natively, but rather requires either more complex mathematical objects, or coding schemes like "reification" or semantic labeling in bipartite graphs. Lacking multi-dimensional relations, it is hard to address questions of "community interaction" in graphs: e.g., how is a collection of entities A connected to another collection B through chains of other communities? Where does a particular community stand in relation to other communities in its neighborhood?



(a) *A hypergraph, H.* (b) *The dual hypergraph of H, H^*.*

(c) *The line graphs $L_s(H)$ of hypergraph H for $s = 1, 2, 3$.*

Fig. 1: *Euler diagram of (a) a hypergraph H and (b) its dual H^*. H has hyperedges $E = \{A, B, C, D\}$ on the vertex set $V = \{1, 2, \ldots, 12\}$. Dually, H^* has edge set $E^* = \{1, 2, \ldots, 12\}$ with vertex set $V^* = \{A, B, C, D\}$. The diagram shows each of the four hyperedges as a "lasso" around its vertices. The visualization was done with the HyperNetX library [32]. (c) The s-line graphs for H for $s = 1, 2, 3$*

The mathematical object that *natively* represents multi-way interactions in networks is called a "hypergraph" [7]. In contrast to a graph, in a hypergraph $H = (V, E)$ those same vertices are now connected generally in a family E of hyperedges, where now a hyperedge $e \in E$ is an arbitrary subset $e \subseteq V$ of k vertices, thereby representing a k-way relationship for any integer $k > 0$. Hypergraphs are thus the natural representation of a broad range of systems, including those with the kinds of multi-way relationships mentioned above. Indeed, hypergraph-structured data (i.e. hypernetworks) are ubiquitous, occurring whenever information presents naturally as set-valued, tabular, or bipartite data.

An example of a hypergraph $H = (V, E)$ is shown in Figure 1. Here the vertex set is $V = \{1, 2, \ldots, 12\}$, and there are four hyperedges $e \in E = \{A, B, C, D\}$, where each $e \subseteq V$. For example, $B = \{3, 4, \ldots, 10\} \in E$. Note that edges vary in size, with e.g. $|A| = 4, |B| = 8$. Note also that $|D| = 2$. Thus D, and D alone, is a proper graph edge. In fact, every graph is a 2-uniform hypergraph, where every edge has size 2. Every hypergraph H has a dual H^* constructed by considering the set E as a new vertex set V^* and the set V

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00045

Stage	Naive	Our method
s-overlap	**314.757s**	**0.684s**
Squeeze	0.005s	0.005s
s-connected component	<1ms	<1ms
Total time	314.762s	0.689s
Speedup	1×	460×
#set intersections	3.52×10^{10}	3.26×10^{7}

TABLE I: *Computational cost of each step of the pipeline to compute the s-line graph from the email-EuAll dataset [15]. Clearly, s-overlap computation (in bold) time is the dominant stage in the process.*

as hyperedge set, E^* in H^*. In a graph, incident edges share a common vertex, while in a hypergraph, incident hyperedges can vary in intersection size. For example, $|A \cap B| = 2$, while $|B \cap C| = 3$ and $|B \cap D| = 1$. We use the variable s to indicate edge intersection size, and call that intersection the *s-overlap* between those edges.

Computation of s-overlaps of a hypergraph to find all pairs of connecting hyperedges with minimal s common neighbors is the most fundamental kernel in the general framework of hypergraph analytics for connectivity and traversal [4]. As we will show, hypergraph structures naturally stratify into substructures parameterized by s-overlap, each of which lends itself to being exploited by traditional, highly efficient graph algorithms. In particular, the s-line graph of a hypergraph is a form of dual graph (not a hypergraph) now on the set of hyperedges as vertices, where each pair of hyperedges are connected if they have an s-overlap. Figure 1c shows the s-line graphs for $s = 1, 2, 3$ for H from Figure 1a.

There are three main motivations for computing the s-line graph of a hypergraph. First, optimized graph kernels can be computed on the s-line graph directly to compute important metrics such as s-connected components, s-centralities etc. Second, s-line graph computation enables structural analyses of hypergraphs that are otherwise challenging or impossible to determine without their formation. As we illustrate in Section II, one such example is spectral analyses of hypergraphs. Third, s-line graphs best capture applications in many important domains [4], [17]. For example, in [17], s-overlap walks are used to define hypergraph s-centralities applied to gene expression levels in virology data sets.

Computing the s-line graph of a hypergraph on large datasets can become computationally challenging when considering each possible combination of pairs of hyperedges (Table I). To the best of our knowledge, the only available, sequential implementation of the s-overlap computation is found in the Python-based HyperNetX [32] library with NetworkX [19] as the backend. However, the algorithm implemented in HyperNetX (naive algorithm discussed in Section IV) is quite inefficient and fails to execute on the large datasets that we consider in Section V. Hence devising efficient, parallel algorithm to compute s-line graph is vital. Our paper aims to address these two aspects. We propose efficient algorithms for computing s-line graphs, apply different heuristics to eliminate redundant work, and discuss the parallelization strategies for our algorithms. To the best of our knowledge, we are the first to take such endeavor and propose efficient parallel algorithms for s-line graph computation. Considering the potentials for applicability of s-overlap computation in many application domains, we hope our algorithms will be useful to the data analytics and network science community.

The contributions of this paper are:

- Identifying the core kernel for hypergraph-related connectivity and traversal algorithms, namely s-overlap computation. Efficient s-overlap and corresponding s-line graph computation algorithms and parallelization techniques for the s-overlap computation algorithms.
- Implementation of our own customized cyclic range partitioner to distribute workload cyclically among the threads to avoid load imbalance. We also consider Intel's Threading Building Block's built-in blocked range partitioning strategy and evaluate both cyclic and blocked distribution strategies for our parallel algorithms.
- Experimental results demonstrating the efficacy of our algorithm and heuristics.

The rest of the paper is organized as follows. Section II provides a motivating example of an application of s-line that offers valuable insight about a dataset. In Section III, we provide formal mathematical definitions and concepts related to hypergraphs and our current work. Next, we present our algorithm and heuristics in Section IV. We report experimental results in Section V. We discuss related work in Section VI and draw conclusions in Section VII.

II. A Motivating Example

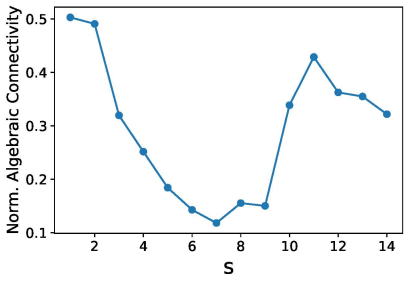


Fig. 2: *Normalized algebraic connectivity for dataset in [3] with various s values.*

For certain hypergraph analytics, the formation of s-line graphs is strictly necessary. To illustrate a particular type of analysis that necessitates s-line graph construction, we compute the normalized algebraic connectivity of the s-line graphs of a Disease-Gene dataset [3]. Normalized algebraic connectivity is the second smallest eigenvalue of the normalized Laplacian matrix [11]; larger values imply stronger connectivity properties of the s-line graph and hence the hypergraph. As plotted in Figure 2, the fluctuations in algebraic connectivity across values of s reveals information about the network structure: the sharp dip from $s = 1$ until $s = 7$ suggests most diseases are linked to one another via sparse gene overlaps. The subsequent increase until $s = 11$ suggests that diseases which are associated with at least 11 genes are notable in being more well-connected amongst each other. 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. These observations are a small example of the insights afforded by spectral methods; see [11] for more.

III. Background

There are disparities among different definitions of hypergraphs in the literature (in terms of allowing duplicates, isolated vertex etc.). Here, we adopt the most general definitions from [4].

A *hypergraph* $H = (V, E)$ has a set $V = \{v_1, \ldots, v_n\}$ of elements called *vertices*, and an indexed family of sets $E = (e_1, \ldots, e_m)$ called *hyperedges* in which $e_i \subseteq V$ for $i = 1, \ldots, m$. The *degree* $d(v) = |e \ni v|$ of a vertex v is the number of hyperedges it belongs to. The *size* (degree) $|e|$ of a hyperedge e is the number of vertices it contains. A hypergraph for which the size of all the hyperedges are the same, let's say k, is called a *k-uniform* hypergraph. If the size of the hyperedges varies, the hypergraph is *non-uniform*. In case of graphs, $|e_i| = 2$, i.e. an edge connects exactly two vertices. Hence, graphs are also called *2-uniform hypergraph*.

Defining E as an indexed family of sets (as opposed to a set system or a multi-set) allows for multiple copies of edges that can be differentiated by index. This definition also allows for isolated vertices not included in any hyperedge, empty edges, and singleton edges. In a hypergraph two vertices are considered *adjacent* if there is a hyperedge e_i that contains both of these vertices. We do not allow self-loops in this paper.

For each hypergraph $H = (V, E)$, there exists a *dual* hypergraph H^* whose vertices are the edges e_i of H and whose hyperedges are the vertices v_i of H. The incidence matrix, I of a hypergraph $H = (V, E)$ is an $n \times m$ matrix where

$$I(i,j) = \begin{cases} 1 & \text{if } v_i \in e_j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

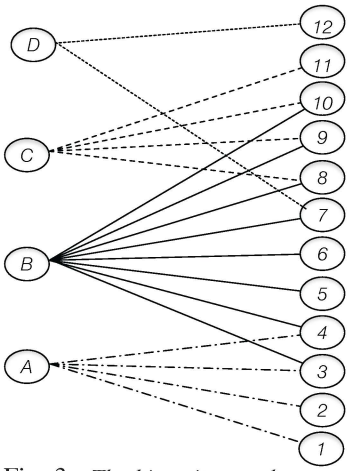


Fig. 3: *The bipartite graph representation for H in Figure 1a.*

The *representative* graph, *intersection graph* or *line* graph $L(H)$ of a hypergraph H is a simple graph whose vertices $x_1, \ldots, x_m$ represent the hyperedges $e_1, \ldots, e_m$ of H, and vertex x_i is connected with vertex x_j if $e_i \cap e_j \neq \emptyset$, for $i \neq j$. The edges $\{x_i, x_j\}$ of $L(H)$ can be assigned weights $w_{i,j} = |e_i \cap e_j|$ indicating the size of the hyperedge intersection. For two such nodes x_i, x_j, for any s for which $s \leq w_{i,j}$, we say that e_i and e_j have an *$s-$overlap* or *$s-$width*. For some $s \geq 1$, restricting $L(H)$ to those edges with weight $w_{i,j} \geq s$ yields the *s-line graph* denoted $L_s(H)$. Note that thereby the line graph $L(H) = L_1(H)$ is the 1-line graph.

Hypergraphs are one-to-one with *bipartite* graphs by the following construction. A *bipartite* graph consists of two disjoint sets of vertices such that no two vertices within the same set are connected by an edge. The bipartite graph of a hypergraph H has one set of vertices, V_1 consisting of the vertices V and another set V_2 consisting of the hyperedges E. An edge exists between two vertices of the bipartite graph if the corresponding vertex is included in the incidence of the hyperedge. Figure 3 shows the bipartite graph for H from Figure 1.

IV. Algorithms for Computing s-overlaps and s-line Graphs

In this section, we discuss in detail our efficient algorithm and heuristics to compute the s-overlaps of a hypergraph. Given a dataset represented as a hypergraph H, by computing the s-overlaps, we can infer interesting relationships among different entities in the dataset and subsequently apply any optimized graph algorithm to the line graph, $L(H)$ by executing the following steps.

Step 1: Compute the number of common neighbors between each pair of hyperedges in H and discard any pair(s) for which the common neighbor count is less than s. This is the s-overlap computation step.

Step 2: Construct the s-line graph $L_s(H)$ based on the s-overlap computation. This involves only considering the hyperedge pairs for which there is at least s common neighbors. Each of these edge pairs will constitute an edge in the edgelist of $L_s(H)$. Once we compute all such pairs of hyperedges, we construct the s-line graph $L_s(H)$. In $L_s(H)$, each hyperedge e_i is considered as a vertex vl_i and there exists an edge between vertex vl_i and vl_j if there is at least s common neighbors between the corresponding hyperedges e_i and e_j in H. For example, Figure 1c shows the line graphs of the hypergraph in Figure 1a when we consider $s = 1, 2, 3$. For constructing a 2-line graph, here in the original hypergraph H in Figure 1a, the hyperedge pairs $\{B, C\}$ and $\{A, B\}$ shares at least 2 vertices: ($\{8, 9, 10\}$ for the first pair $\{B, C\}$ and $\{3, 4\}$ for the other). Hence, in $L_2(H)$ (Figure 1c), we include an edge between B and C and another edge between A and B. Note that, there is no edge between A and C, since they do not share any vertices ($s < 3$ between A and C).

Step 3: Once the s-line graph is constructed from the s-overlaps, we can execute any graph algorithm of interest on the s-line graph.

A. A Naive Algorithm to Compute the s-overlaps

A naive algorithm, which computes the s-overlaps and constructs the edge list of the line graph based on the computed s-overlaps, considers each pair of hyperedges in the hypergraph to see whether there are at least s common vertices shared by the pair. If this is the case, the hyperedge pair (e_i, e_j) is added to the edge list of the line graph. However, many hyperedges may not have any common neighbor (vertex) to connect them. Hence, this algorithm performs a lot of redundant work when exhaustively checking each pair of hyperedges.

Algorithm 1 An alternative algorithm to compute the s-overlaps of a hypergraph and construct the edge list of the line graph based on s-overlaps.
Input: Hypergraph $H = (V, E)$, s
Output: s-line graph edge list $L_s(H)$

1: $L_s(H) \leftarrow \emptyset$
2: **for each** hyperedge $e_i \in E$ **do**
3: **for each** incident vertex v_k of e_i **do**
4: **for each** incident hyperedge e_j of v_k such that $(j \neq i)$ **do**
5: count $\leftarrow set\_intersection(e_i, e_j)$
6: **if** count $\geq s$ **then**
7: $L_s(H) \leftarrow L_s(H) \cup \{e_i, e_j\}$
8: **return** $L_s(H)$

Algorithm 2 An efficient algorithm with heuristics to compute the s-overlaps of a hypergraph and construct the edge list of the line graph based on s-overlaps.
Input: Hypergraph $H = (V, E)$, $size[e_i]$ $\forall e_i \in E$, s
Output: s-line graph edge list $L_s(H)$

1: $L_s(H) \leftarrow \emptyset$
2: **for each** hyperedge $e_i \in E$ **do**
3: **if** $size[e_i] < s$ **then** ▷ Degree-based pruning
4: **continue**
5: **for each** $e_j \in E \setminus \{e_i\}$ **do**
6: $visited[e_j] \leftarrow$ false
7: **for each** incident vertex v_k of e_i **do**
8: **for each** incident hyperedge e_j of v_k **do**
9: **if** $e_i \geqslant e_j$ **then** ▷ Consider strictly upper triangular part of the hyperedge adjacency matrix
10: **continue**
11: **if** $size[e_j] < s$ **then**
12: **continue**
13: **if** $visited[e_j] ==$ true **then**
14: **continue** ▷ The current hyperedge e_j has already been considered for the set intersection with e_i
15: **else**
16: $visited[e_j]$ = true
17: s_overlapped $\leftarrow set\_intersection(e_i, e_j, s)$
18: **if** s_overlapped == true **then**
19: $L_s(H) \leftarrow L_s(H) \cup \{e_i, e_j\}$
20: **return** $L_s(H)$

B. An Alternative Algorithm to Compute the s-overlaps

To avoid considering redundant pairs of hyperedges when computing the s-overlaps, we propose an alternative algorithm in Algorithm 1. In this algorithm, instead of considering each pair of hyperedges in the hypergraph $\{(e_i, e_j) |\ e_i \in E,\ e_j \in E,\ i \neq j\}$ for the s-overlap computation, we only consider those hyperedge pairs (e_i, e_j) that are incident to a common vertex. Mathematically, we only consider hyperedge pairs $\{(e_i, e_j) |\ e_i \in E,\ e_j \in E,\ i \neq j\ \wedge \exists\ n_k |\ n_k \in V,\ n_k \in I(e_i),\ n_k \in I(e_j)\}$.

C. Pruning Strategies for Optimization

To avoid redundant work, we apply the following pruning techniques in different stages of Algorithm 1, which results in Algorithm 2. The heuristics for pruning redundant work are:

- **Degree-based pruning:** When considering a pair of hyperedges (e_i, e_j) for the s-overlap computation, since it is required that there are at least s vertices in the incidence list of each of these hyperedges, if the size of either edge $|e_i|$ or $|e_j|$ is smaller than s, we can exclude the hyperedge from consideration when computing the s-overlap (Line 3 and Line 11 in Algorithm 2).
- **Skip already visited hyperedges:** Consider a connected hyperedge pair (e_i, e_j). Here e_i and e_j denote hyperedge IDs (associated with each set of vertices in e_i and e_j). It is possible that, e_i's one-hop hyperedge neighbor, in this case e_j, can be reached via multiple different vertices v_k, v_l, v_p etc., where $k \neq l \neq p$ (Line 7 in Algorithm 2). This means there exist "wedges" (e_i, v_k, e_j), (e_i, v_l, e_j), (e_i, v_p, e_j) etc. in the bipartite graph representation of the hypergraph. Since we only need to perform set intersection between potential hyperedge pairs once, we can keep track of whether a one-hop hyperedge e_j has already been considered for intersection by maintaining a `visited` array. This boolean `visited` array of size m is allocated outside of the outer loop in Algorithm 2. At the beginning of each iteration of the outer loop, the array is cleared first. During the execution of the innermost loop, we check whether the current hyperedge e_j has already been considered for set intersection. If this hyperedge is already visited, we can skip the set intersection. Otherwise we set the $visited$ entry to true for e_j and proceed with the set intersection (Lines 13–16 in Algorithm 2).
- **Short-circuiting in set intersection:** During the set intersection of the vertex neighbor lists of hyperedges e_i and e_j, as soon as we detect s overlaps of vertices between e_i and e_j, we can return immediately, without iterating through the complete list of neighbors, and return a boolean value of `true`, instead of intersection `count`, indicating that the edge pair has at least s common neighbors. Otherwise the `set_intersection` returns false. The modified set intersection takes s as an input for comparison.
- **For set intersection, consider strictly upper triangular part of the hyperedge adjacency:** Consider a *connected* hyperedge pair (e_i, e_j). Since each edge e_i and e_j will be considered independently (on Line 2 of Algorithm 2), and since the connected pair has at least one neighboring vertex in common, set intersection will be performed twice with the same edges (e_i, e_j) individually on Line 17: by considering the wedges (e_i, n_k, e_j) and (e_j, n_k, e_i), where n_k is a common neighboring vertex. We can eliminate one of these redundant set intersections by only considering wedge (e_i, n_k, e_j) when $e_i > e_j$. In other words, we only consider the upper triangular matrix of edge adjacency (here the edge adjacency refers to the $m \times m$ matrix where a non-zero entry exists whenever the edge-pair $((e_i, e_j))$ is at least 1-connected. Note that we are not constructing

the edge adjacency concretely, it is just for conceptual reference. Additionally, to avoid self-loops in $L_s(H)$ (when $e_i = e_j$), we only consider strictly upper triangular part of the hyperedge adjacency.

D. Linear Algebraic Formulation of the s-line Graph Computation

Conceptually, considering the incidence matrix H, and identity matrix I, s-line graph computation can be expressed as the multiplication of two sparse matrices H and H^T (where H^T is the transpose of matrix H), then subtracting $2I$ from it (to avoid self loops in the resultant matrix), and finally applying a *filtration* operation to each entry of the $HH^T - 2I$ matrix, such that, in the resultant matrix R, $R_{i,j} = 0$ if $\{HH^T - 2I\}_{i,j} < s$, otherwise $R_{i,j} = 1$.

Computing the sparse general matrix-matrix multiplications (SpGEMM) [18] and then applying the filter operation to find the edgelist of a s-line graph is closely-related (NOT equivalent) to the alternative approach Algorithm 1. However, SpGEMM is time-consuming (results shown later) and there are three reasons why it is not efficient for s-line graph computation. First, it considers both the upper triangular and lower triangular parts of the hyperedge adjacency matrix, even though the matrix is symmetric. Second, since SpGEMM is more general, SpGEMM has to compute and materialize the product matrix before applying filtration upon it. This requires extra space to store the intermediate results (while the alternative approach does not). This is the only difference between the alternative approach and SpGEMM plus filtration. Third, it cannot apply heuristics to speedup the computation, such as pruning or short circuiting the set intersection.

E. Hypersparsity and ID Squeezing

The edge list of the line graph based on the s-overlap computation, $L_s(H)$, contains the original hyperedge IDs. However, as we increase the value of s, where s is bounded by the maximum size of the hyperedges, many hyperedges may not be included in the edge list of the line graph, simply because there may not be enough overlaps between any such pairs of hyperedges. In terms of adjacency/incidence matrix representation, *hypersparsity* refers to the cases where many rows of the matrix do not have any non-zero element (or zero-columns for the transposed matrix).

Allocating memory and constructing the line graph based on the original IDs of the hyperedges is infeasible due to this observed hypersparsity phenomena. To avoid this problem, we construct a new set of contiguous IDs for the hyperedges (i.e. for the vertices in the line graph) included in the line graph's edge list and maintain a mapping between original IDs and the new IDs. Remapping the IDs is termed as *squeezing*. ID squeezing is an essential step before materializing the line graph adjacency from the edge list.

F. Parallelization

We implemented our algorithms in C++17 and utilize Intel's Threading Building Block (TBB) [1] to parallelize our algorithms. For example, the outer `for` loop (Line 2 in Algorithm 2) is parallelized with TBB's `parallel_for`. We invoke TBB's `parallel_for` in the form of `(range, body, partitioner)`. Considering this form enables us to provide custom range to TBB. We discuss one of the custom ranges (cyclic range) in the context of workload balancing techniques later in this section. `parallel_for` breaks the hyperedge iteration space (range) in chunks and schedules each chunk on a separate thread. Each thread executes the body (a lambda function, containing the logic for the s-line graph computation) on each of these chunks.

1) Workload distribution strategies for load balancing: The first argument (range) of TBB's `parallel_for` API, `(range, body, partitioner)`, provides provision to supply different strategies to partition and distribute the iteration space (range) among the threads. As long as the provided range adheres to the C++ `Range` concept [31], it enables the user to experiment with different workload balancing techniques.

Blocked range. When considering one-dimensional iteration space of hyperedges for partitioning, one possibility is to specify the range of hyperedges as a `blocked_range` (other options include 2D, 3D partitioning etc.). A `blocked_range` represents a recursively splittable range and is provided as a built-in range in TBB. Each thread is assigned a contiguous chunk of iteration space to work on. Additionally we specify TBB's `auto_partitioner` as the range partitioning strategy to enable optimization of the parallel loop by range subdivision based on work-stealing events. The `auto_partitioner` adjusts the number of chunks depending on the available execution resources and load balancing needs.

However, applying the blocked range strategy to distribute the workload can be problematic for applications with irregular work pattern such as hypergraph algorithms. This is because the degree distributions of the hyperedges of the hypergraphs we experimented with are mostly non-uniform. Splitting the hyperedge range (ID range of the hyperedges) with blocked distribution can create uneven workload if hyperedges with consecutive IDs have high degrees. In this case, some threads will be assigned larger workload compared to others.

Cyclic range. An alternative to the blocked_range strategy is a `cyclic range`, where, assuming the stride size is equal to the number of threads nt, thread 0 processes hyperedges $e_0, e_{0+nt}, e_{0+2*nt}, e_{0+3*nt}$ etc., thread 1 processes hyperedges $e_1, e_{1+nt}, e_{1+2*nt}, e_{1+3*nt}$ and so on. Here e_i denotes hyperedge ID. If a hypergraph contains high-degree hyperedges with consecutive IDs, the cyclic range will distribute the workload more evenly among the threads than the blocked range by reducing the effect of consecutive high-degree hyperedge IDs (induced locality) on the workload.

Relabel by degree. When applying the cyclic workload distribution strategy, it may be useful to relabel the hyperedge IDs according to their degrees, so that the high-degree hyperedges will be given smaller IDs and the low-degree hyperedges will obtain larger IDs. The high-degree hyperedges will be adjacent to each other (since their new IDs will be consecutive). In this

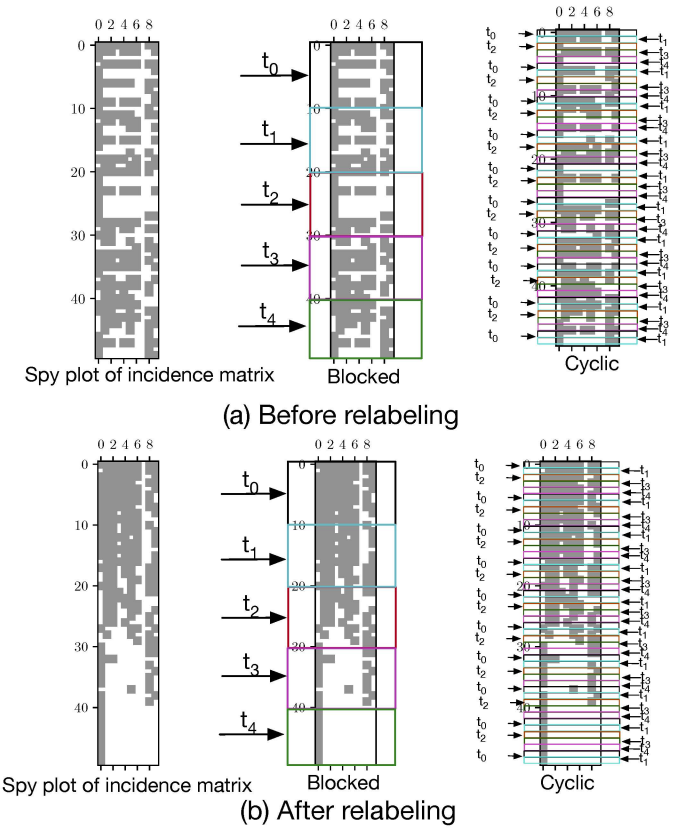


Fig. 4: *Spy plot of the incidence matrix of a hypergraph before and after relabeling. We also show how workloads are distributed among threads when using blocked vs cyclic range in each case (assuming that workloads are distributed among 5 threads).*

way, when processing the hyperedges, each thread will have a better chance of load-balanced workload assignment when applying the cyclic distribution.

Figure 4 shows the spyplots of a hypergraph incidence matrix before and after relabeling the IDs of the hyperedges. In addition, the figure also illustrates how workload is distributed among threads when considering the blocked and cyclic partitioning. As can be observed from the figure, before relabeling the IDs of the hyperedges, the blocked range-based partitioning suffers from uneven workload distribution among the threads (color coding in the figure shows the range of hyperedges assigned to each thread for processing). On the other hand, the cyclic range circumvents this problem by trying to scatter the high-degree hyperedges among all the threads. To make this approach even more effective, relabel-by-degree ensures that all high-degree hyperedges ends up with consecutive IDs so that each thread will choose one high-degree hyperedge ID at a time for processing when considering the cyclic distribution of the hyperedges among the threads.

2) Thread-local data structures: To avoid any duplicate set intersection, each thread also maintains a `boolean visited` array of size $m = |E|$ to track the set of hyperedges that have already participated in the set intersection with a hyperedge. In addition, each thread also maintains a thread-local `vector` for inserting edge pairs, if such pair meets the s-overlap requirement. Once all the threads finish processing their assigned edge range, the thread-local edgelists are consolidated into one array to construct $L_s(H)$.

Network type	hypergraph	$\|V\|$	$\|E\|$	$\overline{d}_e$	Δ_v	Δ_e
Social	Friendster	7.94M	1.62M	14	1700	9299
	LiveJournal	3.2M	7.49M	15	300	1.05M
Webgraph	Web	27.7M	12.8M	11	1.1M	11.6M
Cyber	activeDNS (256 files)	4.5M	43M	8	1M	10M
Collaboration	IMDB	896k	3.8M	8	1.6k	1334
Miscellaneous	email-EuAll	265.2k	265.2k	1.6	7.6k	930

TABLE II: *Input hypergraph characteristics. Total number of vertices ($|V|$) and hyperedges ($|E|$) along with the average size of each edge ($\overline{d}_e$), max degree of a vertex (Δ_v), and maximum size of a hyperedge (Δ_e) for the hypergraph inputs are tabulated here. All the hypergraphs have a skewed hyperedge degree distribution.*

G. Time Complexity Analysis

Given a hypergraph $H = (V, E)$, where $|V| = n$, $|E| = m$, let us denote the average degree of $v_i \in V$ by $\overline{d}$, and average size of $e_i \in E$ by $\overline{k}$. In the worst case scenario, each set intersection will perform $O(m)$ comparisons. Hence, Algorithm 1 takes $O(nm^3)$ time in the worst case.

In the worst case, without any heuristic, our efficient algorithm (Algorithm 2) takes $O(nm^3)$ time. To factor in the effect of the heuristics on the execution time, let us assume that, on average, each set intersection operation performs $\overline{k}$ number of comparisons. Considering this, in the average case, our algorithm takes $O(m \cdot \overline{k} \cdot \overline{d} \cdot \overline{k})$ time. The amount of work that the degree-based pruning saves ($O(\overline{d} \cdot \overline{k})$) is based on the value of s. Since this heuristic is encoded in both the outer loop and inner loop, it saves most of the redundant work by avoiding execution of two innermost loops. With the upper triangular heuristic, it saves half of the work by skipping one endpoint of every hyperedge pair. Additionally, with short-circuiting in set intersection, we only need s successful comparisons for early termination. Our approach takes $O(m \cdot \overline{d} \cdot \overline{k} \cdot s)$ time after all heuristics kick in.

V. Evaluation

In this section, we report and analyze our experimental results for computing s-overlaps and line graphs.

A. Experimental setup

For experimental evaluation, we ran our experiments on a machine with a two-socket Intel Xeon Gold 6230 processor, with 20 physical cores per socket, each running at 2.1 GHz, has 28 MB L3 cache. The system has 188 GB of main memory. We implemented our code in C++17, compiled the library with Intel TBB 2020.2.217, GCC 10.1.0 compiler and `-Ofast` compilation flag.

B. Datasets

We conducted our experiments with datasets mentioned in Table II. These datasets represent a set of hypergraphs constructed from diverse domains: ranging from social networks to cyber data to web, each having different structural properties. Basic statistics of these hypergraphs are presented in Table II.

The activeDNS (ADNS) dataset from Georgia Institute of Technology [24] contains mappings from domains to IP addresses. When constructing hypergraphs with ADNS dataset, we consider the domains as the hyperedges and IPs as vertices. We followed the same procedure as described in [21] to curate this dataset.

We also ran our experiments with datasets curated in [33]. For these curated datasets, in particular, each hypergraph, constructed from the social network datasets such as Friendster [26] in Table II, are materialized by running a community detection algorithm on the original dataset from Stanford Large Network Dataset Collection (SNAP) [26]. 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) [23] as bi-partite graphs. The IMDB dataset [15] represents a bi-partite graph, where the hyperedges are movies and vertices are actors/actresses.

Notation	Algorithm
$f0$	All heuristics included (Algorithm 2)
$f1$	Only degree-based pruning
$f2$	Only skipping of the visited hyperedges heuristic
$f3$	Only short-circuiting in the set intersection
$f4$	Baseline efficient with no heuristic (Algorithm 1)

TABLE III: *Notation for different algorithms and heuristics.*

C. Experimental Results

In this section, we discuss the experimental results in detail. First we report the performance improvement of our algorithm over the naive one, as we add different heuristics, one at a time, as well as combining them altogether. Next, we assess the effect of our proposed heuristics as we vary the size of s. We also report the scalability results (strong and weak scaling) of our algorithms with different workload distribution strategies. We discuss the inputs for which relabel-by-degree with the cyclic workload distribution is helpful. Finally we show how the workload is distributed among threads with different workload distribution techniques. For the subsequent discussion, we refer to Table III for the shorthand notations of our different algorithms and heuristics. To be consistent and fair, we applied the upper triangular heuristic across all the algorithms, so that only upper triangular part of the hyperedge adjacency is considered when computing the s-overlaps (thus avoiding inclusion of self-loops and duplicate edges in the line graph edge list).

1) Performance comparison of our algorithm with different heuristics and the naive algorithm: Figure 5 reports speedup of our algorithm with different heuristics compared to the naive algorithm, discussed in Section IV. Here we only consider $s = 8$. With other s values, the algorithms exhibit similar trend. In these experiments, we consider dns-008, LiveJournal and Friendster datasets. We have chosen these datasets to represent different application domains and varying hypergraph sizes (ranging from medium to large).

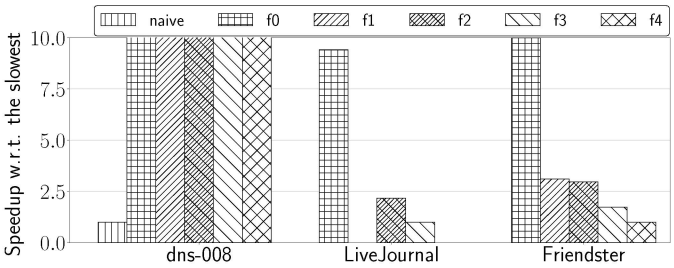


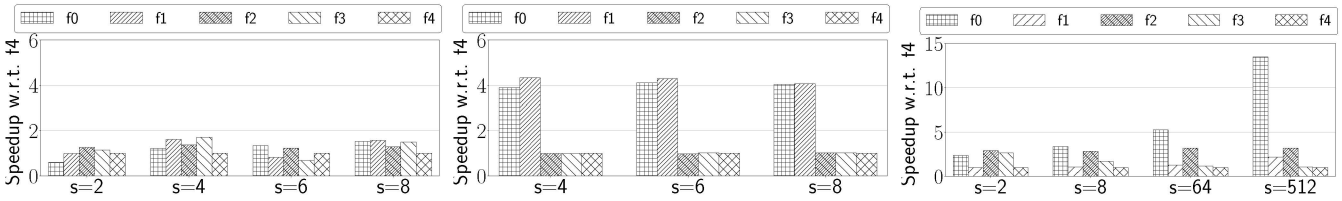
Fig. 5: *Performance comparison of different s-overlap computation algorithms, including the naive algorithm. Here, for each dataset, we report the speedup relative to the slowest execution time that have ran to completion. In two cases (LiveJournal and Friendster), naive algorithm did not finish execution in a reasonable time. In addition, for LiveJournal, f1 did not finish in a reasonable time. Hence these three data points are missing in the plot. The results are reported with 32 threads and for $s = 8$.*

From the figure, we observe that, with dns-008 dataset, our efficient algorithm is significantly faster compared to the naive algorithm. With LiveJournal and Friendster datasets, the naive algorithm could not finish within a reasonable time limit. With dns-008 dataset, our algorithm is more than 10x faster compared to the naive algorithm. Although not visible in Figure 5, we also observed that, for these datasets, the most important heuristic for s-overlap computation is the degree-based pruning heuristic ($f1$). In fact, with dns-008 dataset, by only using degree-based pruning heuristic, our algorithm is slightly faster than $f0$. This is because dns-008 dataset has less than 1% hyperedges with degrees greater than 8. In this case, adding other heuristics introduced more overhead.

With LiveJournal and Friendster social network-based hypergraph inputs, $f2$ (skipping visited hyperedges) heuristic is one of the most impactful heuristics. With the LiveJournal input, $f1$ (degree) heuristic based algorithm ran out of memory. Hence this data point is missing in the figure. Compared to other inputs, the hyperedges in Livejournal have higher maximum degree-count (Table II). Hence, compared to other same-scale hypergraphs, more set-intersections need to be performed when applying only degree-based heuristic. With Friendster and LiveJournal inputs, the naive algorithm either did not finish in reasonable time or ran out of memory.

2) Ablation study - effect of different heuristics on different datasets: We consider the effects of different heuristics on the s-overlap computation of hypergraphs that are drawn from different domains. In addition, we also vary the value of s to see which (set of) heuristic(s) is the most impactful in improving performance. Figure 6 shows the result of our experiments with IMDB, dns-128, and Friendster datasets with $s = 2, 4, 6, 8$. Here the speedup is reported by normalizing with respect to the baseline algorithm with no heuristic (f4).

With the IMDB dataset, the performance of the algorithms varies significantly, based on the s values. When computing the s-overlaps with cyber datasets, dns-128, degree-based pruning, $f1$, significantly improves performance. As we increase the value of s from 4 to 8, the effect of this heuristic remains impactful. This implies that pruning the hyperedges based on their sizes eliminated a lot of redundant set intersections. Other



(a) *IMDB.*

(b) *dns-128. Degree-based pruning is the most impactful heuristic in this case.*

(c) *Friendster. Skipping visited hyperedges heuristic is the most impactful heuristic as well as degree-based pruning for $s >= 8$.*

Fig. 6: *Effect of different heuristics on different datasets and with different s values. The speedup is reported by normalizing w.r.t the baseline algorithm (f4). Here we report execution time running with 32 threads.*

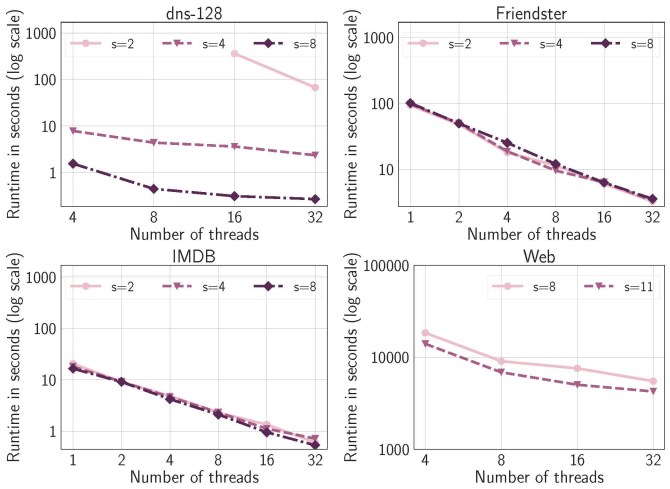


Fig. 7: *Strong scaling results with cyclic distribution for Algorithm 2, $f0$*

heuristics also positively effect the execution time, however less so compared to the degree-based pruning. With $s = 2$ and dns-128 dataset, only our algorithm with all heuristics included ($f0$) ran successfully. Other variations of the algorithm either did not finish in reasonable time or ran out of memory.

On the other hand, with Friendster social network dataset, we observe that $f2$ heuristic, i.e. skipping already visited hyperedges for set intersection is the most impactful heuristic.

Other datasets such as with Web dataset and various s values, only $f0$ finished in a reasonable time limit. Other heuristics are not very helpful in isolation for this dataset.

3) Strong scaling results: We also conduct the strong scaling experiments with the hypergraph inputs from different domains, both with the blocked partitioning and the cyclic partitioning strategies. Here we increase the number of threads while keeping the dataset (input size) constant. We ran our efficient algorithm ($f0$) with all the heuristics turned on. We report the experimental results with the cyclic distribution in Figure 7. Here we consider dns-128 (representing cyber dataset), Friendster (representing social network), IMDB (representing collaboration network), and Web (webgraph) datasets as inputs. As can be seen from the figure, as we increase the number of threads, the performance of the algorithm improves significantly. Blocked distribution demonstrates similar strong scaling behavior for most datasets, except for dns-128 and Web. With the dns-128 dataset, even with smaller number of threads, the cyclic workload successfully distributes workload evenly among the threads to achieve better performance. With large-scale hypergraphs such as Web dataset, for which the maximum degree of a hyperedge is 11M, $f0$ with cyclic partitioning achieves $3\times$ speedup over $f0$ with blocked distribution (the absolute execution time $\approx$ 8000 seconds for cyclic vs $\approx$ 21000 seconds for blocked range).

4) Weak scaling results: For weak scaling experiments, we approximately double the size of the hypergraph (workload) as we double the number of threads (computing resources). We perform the weak scaling experiments of Algorithm 2 ($f0$) with the activeDNS dataset and considering the cyclic workload distribution strategy. Blocked partition demonstrates similar trend. To construct the hypergraphs, we start with 4 input files worth of data (dns-004) and progressively increase the hypergraph size by adding data from more files, as we increase the resources (dns-004 with 1 thread, dns-008 with 2 threads, dns-016 with 4 threads, and so on, up to dns-128).

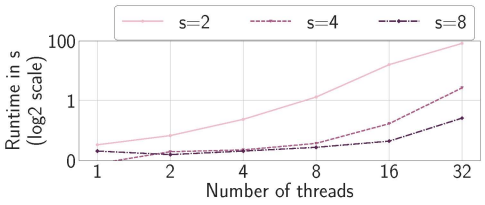
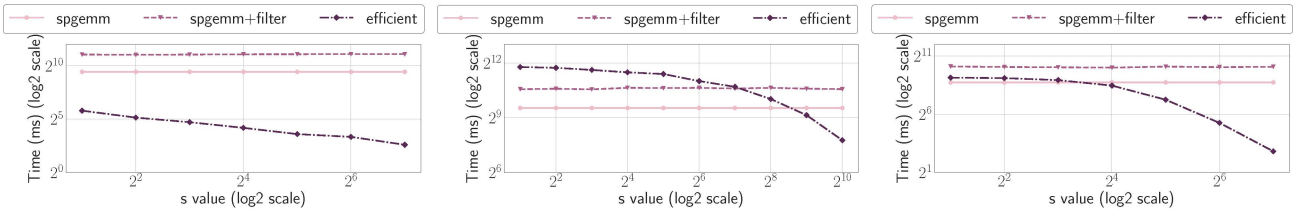


Fig. 8: *Weak scaling results of Algorithm 2 ($f0$) with cyclic distribution with the activeDNS dataset.*

We report our weak scaling results in Figure 8. As can be seen from the figure, with larger s values, $f0$ with cyclic distribution exhibit better scaling (the execution time remains almost constant as we increase the problem size and proportionately increase the computing resources). For higher s values, the heuristics are more effective to reduce the amount of redundant work.

5) Evaluation of the relabel-by-degree technique: We also evaluate how relabeling the hyperedge IDs based on degrees influences the execution time. We have not observed any performance benefit of relabeling when it is considered with blocked partitioning. However, when relabeling is considered in conjunction with cyclic distribution, the performance of the s-overlap computation improves. For smaller hypergraphs, the pre-processing time for relabeling can overshadow the benefit of faster s-overlap execution time. For example, Friendster takes $\approx$ 8 seconds to relabel the hyperedge IDs based on de-



(a) email-EuAll. (b) Friendster. (c) IMDB.

Fig. 9: *Comparison of SpGEMM-based approach with Algorithm 2 applying cyclic partitioning, and relabeling by degree (ascending), $f0$*

grees and ≈ 2.7 seconds to compute $f0$ with cyclic distribution ($s = 8$). Without relabeling, the s-overlap computation takes ≈ 3 seconds. In this case, the overhead is quite significant.

The most interesting application of relabel-by-degree are for computing s-overlap with hypergraphs that are larger in size and have a small set of extremely high degree hyperedges. One such example is the Web dataset. For computing the s-overlap with $s = 11$, $f0$ with only cyclic distribution takes 4230 seconds. On the other hand, when relabel-by-degree is combined with the cyclic workload distribution, it takes $\approx$ 3397 seconds to execute the s-overlap computation step and ≈ 27 seconds for the pre-processing step, for a total of ≈ 3424 seconds. In this case, we obtain a speedup of $1.23\times$ over the version with only the cyclic distribution. We also observe that relabel-by-degree in ascending order is considerably faster due to higher cache utility rate according to the profiling done with Intel VTune profiler. Considering the upper triangular part of the hyperedge adjacency in conjunction with relabel-by-degree in ascending order makes our algorithm cache-friendly.

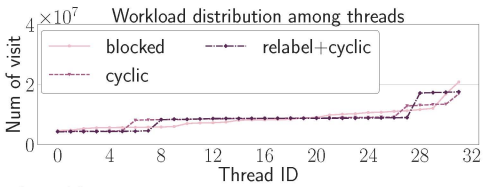


Fig. 10: *Workload distribution of Friendster among 32 threads with different partitioning strategies.*

6) Analysis of workload distribution strategies: To visually assess the effectiveness of different workload distribution techniques among the threads when considering the cyclic and blocked partitioning strategies, we count the total number of hyperedges considered (number of visits) in the innermost loop of our efficient algorithm by each thread. We collect these statistics for the Friendster dataset and report these workloads in Figure 10. As can be observed from the figure, with cyclic distribution, most of the threads have similar workload profile (the longest running horizontal lines). In contrast, with blocked distribution, the workload varies among the threads.

7) Comparison with an SpGEMM-based approach: We also benchmark the performance of our s-line graph computation algorithm against a state-of-the-art SpGEMM-based library [2], [29]. We add the filtration step to their code. The SpGEMM library first computes HH^T and then a filter is applied to extract the edgelist which satisfies the s-overlap constraints. The results are reported in Figure 9. We report the execution time for the SpGEMM step and both SpGEMM plus filtration step separately. With email-EuAll and IMDB datasets, our algorithm is faster with different s values. With the Friendster dataset, computing the s line graph with smaller s value is faster with the SpGEMM library. However, as we increase the s value, our algorithm runs faster than the SpGEMM-based approach. The improvement can be attributed to the degree-based pruning and skipping already visited edges heuristics. Computation of s line graphs with higher s values (for example $s = 1024$) is still relevant for Friendster because, even with such higher s overlap requirement, we found 20 connected components in the constructed s line graph. "Friends of friends" are mostly part of the same communities, hence $f2$ heuristic helps improving the performance of our algorithm over off-the-shelf SpGEMM by pruning redundant work.

VI. Related Work

While graph theory dominates network science modeling, hypergraph analytic has also been successfully applied to many real-world datasets and applications, and the value of hypergraphs and related structures are increasingly being recognized, e.g. [6], [20], [21], [25].

The seminal books [7], [8], written by Berge, are the firsts of their kinds to provide methodological treatment of hypergraphs as a family of sets and their associated properties. Since then, a plethora of theoretical work has been published in the context of hypergraphs [10], [13], [16], [22], and line (intersection) graphs [9], [30]. For example, Bermond et al. [9] studied s-line graph of a hypergraph in details. They show that for any integer s and a graph G, there exists a partial hypergraph H of some complete h-partite hypergraph such that G is the s-line graph of H. A recent survey by Niak [30] on intersection graphs reports recent theoretical developments.

Only recently, Aksoy et al. [4] proposed a high-order hypergraph walk framework based on s-overlaps with *non-uniform hypergraphs*. The paper extends several notions of graph-based techniques to hypergraphs. Although the paper discussed in detail the higher-order analogs of hypergraph methods, the paper does not look into efficient computation of s-overlaps and larger dataset, which is the main objective of this paper. Previously, in the context of *uniform hypergraph*, Cooley et al. [12] studied the evolution of s-connected components in the k-uniform binomial random hypergraph. In contrast, we consider more general cases of hypergraphs: both uniform and non-uniform. Thakur and Tripathi studied the linear connectivity

problems in directed hyperpaths, where the hyperconnection between vertices are called L-hyperpaths [34].

Shun presented a collection of efficient parallel algorithms for hypergraphs in the Hygra framework [33]. However, there is no step involved in computing the s-line graph of a hypergraph and the algorithms are designed to run directly on the hypergraphs. Recent works presented a fast triangle counting implementation with cyclic distribution [5], [27]. As reported in [5], this is currently the best performing implementation compared to all other state-of-the-art graph frameworks. Our cyclic partitioning strategy is inspired by their work for workload balancing based on cyclic distribution.

Battiston et al. have provided a comprehensive survey of the state-of-the-art on the structure and dynamics of complex networks beyond dyadic interactions [6]. They discussed structure of systems with higher-order interactions, measures and properties of these systems, random models used for statistical inference in these systems, and dynamics of the systems with higher-order interactions.

The relabel-by-degree operation is equivalent to the row/-column permutation of a sparse matrix. Both can improve the workload distribution and memory access pattern [14], [28].

VII. Conclusion

In this work, we have proposed efficient, parallel algorithms and heuristics for computing the s-overlaps and the s-line graphs. Our algorithm is orders of magnitude (more than $10\times$) faster than the naive algorithm in all cases and the SpGEMM algorithm with filtration in most cases (especially with large s value) for computing the s-overlaps. To balance workload among the threads, we consider different workload partitioning techniques including blocked range; we also implement and apply a customized cyclic range to achieve better performance. Computing the s-overlaps is the fundamental step for creating high-order line graphs. Once we compute the s-line graphs, many important features of the original hypergraph are retained in the s-line graph and yet we can leverage highly tuned graph algorithms to analyze the data. In this way, analysts can leverage the multi-way relationships in hypergraph to find meaningful relationships.

VIII. 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-164086. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

References

[1] Intel Threading Building Blocks (TBB). [Online]. Available: https://github.com/oneapi-src/oneTBB

[2] (2020) Sparse General Matrix-Matrix Multiplication for multi-core CPU and Intel KNL. [Online]. Available: https://bitbucket.org/YusukeNagasaka/mtspgemmlib/src/master/

[3] (2021) DisGeNET. [Online]. Available: https://www.disgenet.org/

[4] 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.

[5] A. Azad and et. al, "Evaluation of graph analytics frameworks using the gap benchmark suite," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 216–227.

[6] 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.

[7] C. Berge, *Graphs and hypergraphs*. North-Holland, 1973.

[8] ——, *Hypergraphs: combinatorics of finite sets*. Elsevier, 1984.

[9] J.-C. Bermond, M.-C. Heydemann, and D. Sotteau, "Line graphs of hypergraphs I," *Discrete Mathematics*, vol. 18, no. 3, pp. 235–241, 1977.

[10] F. Chung, "The laplacian of a hypergraph," *Expanding graphs (DIMACS series)*, pp. 21–36, 1993.

[11] F. R. Chung, *Spectral graph theory*. American Math Soc., 1997, no. 92.

[12] O. Cooley, M. Kang, and C. Koch, "Evolution of high-order connected components in random hypergraphs," *Electronic Notes in Discrete Mathematics*, vol. 49, pp. 569–575, 2015.

[13] J. Cooper and A. Dutle, "Spectra of uniform hypergraphs," *Linear Algebra and its applications*, vol. 436, no. 9, pp. 3268–3292, 2012.

[14] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*. ACM, 1969, p. 157–172.

[15] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[16] W. Dörfler and D. Waller, "A category-theoretical approach to hypergraphs," *Archiv der Mathematik*, vol. 34, no. 1, pp. 185–192, 1980.

[17] S. Feng and et. al, "Hypergraph models of biological networks to identify genes critical to pathogenic viral response," *BMC Bioinformatics*, vol. 22, no. 1, p. 287, 2021.

[18] 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.

[19] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Tech. Rep., 2008.

[20] C. Joslyn and et. al, "Hypernetwork science: From multidimensional networks to computational topology," in *Unifying Themes in Complex Systems X*, 2021, pp. 377–392.

[21] C. A. Joslyn, S. Aksoy, D. Arendt, J. Firoz, L. Jenkins, B. Praggastis, E. Purvine, and M. Zalewski, "Hypergraph analytics of domain name system relationships," in *International Workshop on Algorithms and Models for the Web-Graph*. Springer, 2020, pp. 1–15.

[22] G. O. Katona, "Extremal problems for hypergraphs," in *Combinatorics*. Springer, 1975, pp. 215–244.

[23] J. Kunegis, "Konect: The koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, ser. WWW '13 Companion. New York, NY, USA: ACM, 2013, p. 1343–1350.

[24] A. Lab. (2020) Active DNS project. [Online]. Available: https://activednsproject.org/

[25] 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.

[26] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection; 2014," *http://snap. stanford. edu/data*, 2016.

[27] A. Lumsdaine, L. Dalessandro, K. Deweese, J. Firoz, and S. McMillan, "Triangle counting with cyclic distributions," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–8.

[28] C. Mueller, B. Martin, and A. Lumsdaine, "A comparison of vertex ordering algorithms for large graph visualization," in *2007 6th International Asia-Pacific Symposium on Visualization*, 2007, pp. 141–148.

[29] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors," *Parallel Computing*, vol. 90, p. 102545, 2019.

[30] R. N. Naik, "On intersection graphs of graphs and hypergraphs: A survey," *arXiv preprint arXiv:1809.08472*, 2018.

[31] E. Niebler, C. Carter, and C. Di Bella, "The one ranges proposal," Tech. Rep., 2018.

[32] B. Praggastis, D. Arendt, C. Joslyn, E. Purvine, S. Aksoy, and K. Monson, "PNNL HyperNetX," https://github.com/pnnl/HyperNetX, 2020.

[33] J. Shun, "Practical parallel hypergraph algorithms," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. ACM, 2020, pp. 232–249.

[34] M. Thakur and R. Tripathi, "Linear connectivity problems in directed hypergraphs," *Theoretical Computer Science*, vol. 410, no. 27, pp. 2592–2618, 2009.