NWHy: A Framework for Hypergraph Analytics: Representations, Data structures, and Algorithms

Xu T. Liu* Jesun Firoz† Assefaw H. Gebremedhin‡ Andrew Lumsdaine*†§
*University of Washington †Pacific Northwest National Lab ‡Washington State University §TileDB, inc.
*{x0, al75}@uw.edu †jesun.firoz@pnnl.gov ‡assefaw.gebremedhin@wsu.edu

***Abstract*—This paper presents NWHypergraph, (NWHy), a parallel high-performance C++ framework for both exact and approximate hypergraph analytics. NWHy provides data structures for various representations of hypergraphs and their associated graph projections (lower order approximations), including a new technique for hypergraph representation called adjoin graphs. We present a set of hypergraph algorithms for exact and approximate hypergraph analytics implemented in NWHy and demonstrate scalability and performance, operating on a variety of hypergraph representations, that is competitive with the state of the art. In addition, we propose two new queue-based algorithms for s-line graph construction, a lower-order approximation of hypergraphs, to demonstrate the effectiveness and versatility of work queue-based algorithm design. Our queue-based algorithms demonstrate similar performance to the non-queue-based algorithms for bipartite graphs.**

***Index Terms*—Hypergraph analytics, hypergraph representation, parallel hypergraph algorithms.**

I. Introduction

Hypergraphs have recently gained significant traction due to their robustness in modeling interactions that go beyond dyadic relationships [3], [10] and have emerged as a powerful alternative to modeling data with graphs. As a mathematical model, a graph represents pairwise relationships (edges) between entities (vertices or nodes). However, this abstraction is insufficient to model mutual relationships involving more than two entities. For example, modeling an author-paper relationship with graphs is challenging, when a three-authors paper needs to be represented. Here pairwise connections between the authors in an author-paper graph fail to capture the three-way relationships among the authors.

When modeling interactions beyond dyadic relationships, sets with cardinality ≥ 2 are better mathematical tools than edges (or pairs) for modeling datasets. As a generalized set-theoretic abstraction, hypergraphs are generalizations of graphs that allow mutual relationships among multiple entities to be expressed. That is, an edge in a hypergraph (hyperedge) may connect more than a pair of vertices (hypervertices or hypernodes). Note that, ”hypervertices” and ”hyperedges” represent two different types of entities (authors and papers for example), hence their representations (index or ID spaces) may be separate.

This paper presents NWHypergraph (NWHy), a high-performance, modern C++ library for hypergraph analytics. We discuss various intricacies and nuances related to implementing a hypergraph library that supports different hypergraph representations, hypergraph construction, generating approximations to hypergraphs, and algorithm design. In contrast to existing hypergraph libraries that only support exact hypergraph computations, NWHy provides support for computation on both exact hypergraphs and on their lower order approximations. These lower order approximations include clique expansion [1], and s-line graph [2], [17], [18] of hypergraphs. By providing interfaces to enable operations on both the original hypergraphs and their approximations, our library supports a wide variety of computations on hypergraphs. The user can choose one type of computation (exact or approximate) over another, based on the time and space requirements, as well as the availability of the algorithms for operating on the original hypergraph. For example, if a third party graph library, such as NWGraph [4] is available, one could create a lower order approximation of a given hypergraph in the form of a special graph (such as clique expansion, line graphs etc.) in NWHy and leverage the highly-tuned, parallel graph algorithms in the traditional graph library, without needing to devise new algorithm to operate on the original hypergraph.

Summary of contributions. NWHy[1] is a modern C++ parallel library for both exact and approximate hypergraph metrics computation. Its contributions include the following:

- Data structures supporting different representations of a hypergraph, namely bipartite graphs, adjoin graphs, clique-expansion graphs, and s-line graphs;
- A suite of important parallel algorithms for exact and approximate hypergraphs metric computations; and
- A Pybind-based Python API to make our high-performance C++-based library backend conveniently accessible to Python users.

The rest of the paper is organized as follows. In Section II, we give formal definitions of a graph, a bipartite graph and a hypergraph. The following section provides a brief introduction to our library and four hypergraph representations, data structures supported in our library, followed by our algorithms, parallelization effort, and Python APIs Section III. We report the scalability and performance results of our hypergraph algorithms in Section IV, discuss related work in Section V and draw conclusions in Section VI.

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

II. Preliminaries

Before proceeding to the details of our framework, we first present our terminology and notation.

A. Graphs

A *graph* $G = (V, E)$ is a finite set $V = \{v_1, v_2, ..., v_n\}$ of *nodes* (or *vertices*) and a finite set $E \subseteq \{(x, y)|x, y \in V, x \neq y\}$ of *edges*, which are pairs of vertices. When the underlying graph is not necessarily obvious, for clarity, we will denote the vertex set V by $V(G)$ and the edge set E denoted by $E(G)$. The edge is said to *join* x and y, and is said to be *incident* on x and y. The number of edges incident on x is called *degree* of x, and is denoted by $d(x)$. The vertices x and y are called *adjacent* if (x, y) is an edge, and x and y are said to be *neighbors*. The set of neighbors of a vertex x in graph G is denoted by $N_G(x)$.

One way to represent a graph is using an incidence matrix. A graph G has a $n \times m$ *incidence matrix* $B = (s_{ij}, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m)$ where n and m are the numbers of vertices and edges respectively, such that:

$$s_{ij} = \begin{cases} 1 & \text{if } v_i \text{ and } e_j \text{ are incident} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Another way to represent a graph is using an adjacency matrix. A graph G has a $n \times n$ square *adjacency matrix* $A = (a_{ij}, 1 \leqslant i, j \leqslant n)$ where:

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Another representation for a graph is an adjacency list, a combination of adjacency matrix with edge list, which we define as follows. The *adjacency list* of a graph G contains n lists, one for vertex $v_i, i = 1, 2, ..., n$. Each list contains the neighbors $(N_G(v_i) : v_i \in V)$ to which v_i is adjacent.

B. Bipartite Graphs

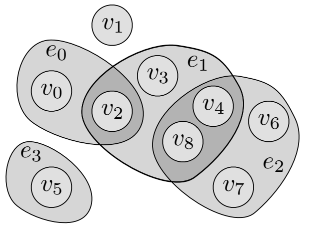
A *bipartite graph* $B = (U, V, E)$ is a graph whose vertices are divided into two disjoint sets $U = \{u_1, u_2, ..., u_r\}$ and $V = \{v_1, v_2, ..., v_s\}$ such that the edges set $E \subseteq \{\{x, y\}|x \in U, y \in V\}$ connects a vertex in U to a vertex in V. Such a partition (U, V) is called a *bipartition* of B, and U and V are its *parts*. A bipartite graph B has a $r \times s$ *bi-adjacency matrix* $A = (a_{ij}, 1 \leqslant i \leqslant r, 1 \leqslant j \leqslant s)$ where:

$$a_{ij} = \begin{cases} 1 & \text{if } u_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

C. Hypergraphs

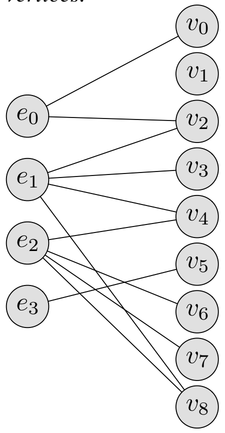
Hypergraphs are one form of a *set system*, which is an ordered pair (U, V), where U is a set of elements and V is a family of subsets of U. Adopting some of the terminology and notation from [2], [7], [8], we define a hypergraph and various related notations below.

A *hypergraph* $H = (U, V)$ includes a finite set $U = \{u_1, u_2, ..., u_n\}$, called *hypernodes* or *hypervertices*, and a finite set of subsets of $V = \{e_1, e_2, ..., e_m\}$, called *hyperedges*, in which $e_i \subseteq U$ for $i = 1, 2, ..., m$. To avoid ambiguity, U will be denoted by $U(H)$, and V by $V(H)$ also.



$$\begin{aligned} H &= \{V, E\} \\ V &= \{v_0, v_1, \ldots, v_8\} \\ E &= \{e_0, e_1, e_2, e_3\} \\ e_0 &= \{v_0, v_2\} \\ e_1 &= \{v_2, v_3, v_4, v_8\} \\ e_2 &= \{v_4, v_6, v_7, v_8\} \\ e_3 &= \{v_5\} \end{aligned}$$

(a) *Hyperedges in a hypergraph are subsets of one or more vertices.*



$$\begin{aligned} B &= \{U', V', E'\} \\ U' &= \{e_0, e_1, e_2, e_3\} \\ V' &= \{v_0, v_1, \ldots, v_8\} \\ E' &= \{\{e_0, v_0\}, \{e_0, v_2\} \\ &\quad \{e_1, v_2\}, \{e_1, v_3\} \\ &\quad \{e_1, v_4\}, \{e_1, v_8\} \\ &\quad \{e_2, v_4\}, \{e_2, v_6\} \\ &\quad \{e_2, v_7\}, \{e_2, v_8\} \\ &\quad \{e_3, v_5\}\} \end{aligned}$$

(b) *Bipartite graph representation for the hypergraph above. Edges in the bipartite graph represent inclusion of a vertex in a hyperedge.*

Fig. 1: *Example hypergraph H and its equivalent bipartite representation B. The entities, and the relationships among them, are the same in either representation.*

We generalize the concept of incidence and the concept of adjacency in graphs to hypergraphs. A hyperedge may *join* one or more hypernodes, and is considered to be *incident* on these hypernodes. A hypernode may join arbitrary number of hyperedges, and is considered to be incident on these hyperedges. Hypernode u_i and hypernode u_j are called *adjacent* if u_i and u_j are incident on the same hyperedge, and u_i and u_j are *neighbors*. We denote a set of neighbors of a hypernode (or a hyperedge) x in H by $N_H(x)$.

A hypergraph can be represented with an incidence matrix, defined as follows. An *incidence matrix* of a hypergraph H is a $n \times m$ matrix, $B = (s_{ij}, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m)$ where n and m are the number of hypernodes and hyperedges respectively, such that:

$$s_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is incident to } e_j \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The transpose B^t of the incidence matrix is the *dual* of H, $H^* = (U^*, V^*)$. Here U^* is the hyperedge set $V(H)$ and V^* is the hypernode set $U(H)$. For instance, the incidence matrix of the dual H^* of the hypergraph H in Figure 1a is:

$$B_H^t = B_{H^*} = \begin{array}{c} \\ e_0 \\ e_1 \\ e_2 \\ e_3 \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \end{array} \\ \begin{pmatrix} 1 & & 1 & & & & & & \\ & & 1 & 1 & 1 & & & & 1 \\ & & & & 1 & & 1 & 1 & 1 \\ & & & & & 1 & & & \end{pmatrix} \end{array}$$

D. *s*-line Graphs and Clique-expansion Graphs of Hypergraphs

Two hyperedges $e, f \in E$ are ***s*-incident** if $|e \cap f| \geq s$ for $s \geq 1$. For integer $s \geq 1$, define the s-line graph of a

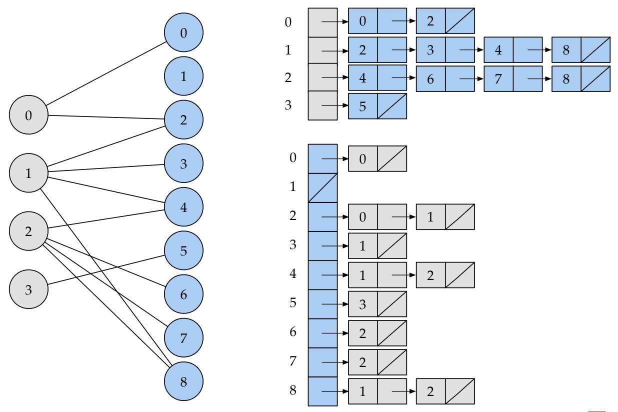


Fig. 2: *The bi-adjacency list of the hypergraph in Figure 1 can be indexed independently and represented as two mutually indexed incidence lists as the hyperedge incidence list and the hypernode incidence list.*

hypergraph $H = (V, E)$ as a graph $L_s(H) = (E_s, V)$ where $V \subseteq \binom{E}{2}$ and $\{e, f\} \in V$ iff e and f are s-incident.

Dually, given a hypergraph $H = (V, E)$ and its dual $H^* = (V^*, E^*)$, the s-clique graph is defined as a graph $C_s(H^*) = (E^*_s, V^*)$ where $V^* \subseteq \binom{E^*}{2}$ and $\{e, f\} \in V^*$ iff e and f are s-incident. 1-clique graph is also known as the clique-expansion graph of the hypergraph.

III. NWHy: Northwest Hypergraph Framework

This section discusses our Northwest Hypergraph (NWHy, short for NWHypergraph) framework in detail. NWHy is based on modern C++ and oneAPI Threading Building Blocks [21]. We describe different hypergraph representations that are available in NWHy, the corresponding APIs for various ways of constructing hypergraphs and their lower order approximations, the hypergraph algorithms implemented in NWHy, parallelization techniques and the Python interfaces to our library. For computation of graph metrics, our framework leverages our graph processing library, NWGraph.

A. Hypergraphs As Ranges

Recently the `std::range` concept has been added to C++20 [20]. Loosely speaking, `std::range` provides a compact, syntactic mechanism in C++20 to iterate over a collection of objects and compose iterations in various ways. Inspired by this abstraction and to build our library on top of modern, idiomatic C++20, in NWHypergraph (and NWGraph), we consider hypergraphs (graphs) as *range of ranges*, where the outer range iterates over the set of hyperedges (hypernodes) and the inner range iterates over the neighbor list (hypernodes or hyperedges). Two range concepts `std::ranges::forward_range` and `std::ranges::random_access_range` are particularly useful to us for hypergraphs. `std::ranges::forward_range` concept is a refinement of `std::range` for which it provides multi-pass access (only forward advancement) over the ranges of elements and guarantees that two iterators to the same range can be compared against each other. `std::ranges::random_access_range` concept is a refinement of `std::bidirectional_range` for which it provides constant time advancement (both forward and backward) with the +=, +, -=, and - operators, and array notation with subscripting. In NWHy, the data structures for hypergraphs model (for example `biadjacency`, cf. Section III-B1) the outer range as a `std::ranges::random_access_range` and the inner range as a `std::ranges::forward_range`.

B. Hypergraph Representations in NWHy

In NWHy, there are four hypergraph representations. We discuss each of these representations in the following.

1) Hypergraphs as Bipartite Graphs: Indexed as Two Separate Index Sets

A hypergraph $H = (U, V)$ can be represented as a bipartite graph $B = (U, V, E)$, where $U(B)$ is the set of hypernodes of H, $V(B)$ is the set of hyperedges, and $E(B)$ contains edges which represent inclusion of a hypernode x in a hyperedge y when y is incident on x. The *bi-adjacency matrix* of B is the incidence matrix of H. Note that, since most of the real-world hypergraphs are sparsely connected, the neighborhood information encoded as adjacency/incidence matrices are generally implemented as an adjacency list or other well-known compact data structures such as Compressed Sparse Row (CSR) data structure. For instance, the bi-adjacency list of the bipartite graph in Figure 1 is shown in Figure 2. In NWHy, we implement the bi-adjacency representation of a hypergraph as two separate but mutually indexed CSR data structures. In such a representation, a hypergraph consists of two separate index sets, one for the hyperedges, another for the hypernodes. The exact hypergraph algorithms leveraging bi-adjacency representation iterates over these two mutually indexed lists for exact hypergraph metric computations. Due to the fact that two separate index sets are maintained (adjacencies), the biggest drawback of any hypergraph algorithms using bi-adjacency is that the algorithm has to maintain two independent algorithm-specific data structures (work queue, frontier, and resultant array, etc.), one for the hyperedges, another for the hypernodes.

The interface of `biadjacency` in NWHy is shown in Listing 1. The `outer_iterator` and `inner_iterator` of the `biadjacency` class models hypergraphs as range of ranges. Given the `biedgelist`, to represent a hypergraph, two `biadjacency` can be created, one for the hyperedges, and another one for the hypernodes. Both of these classes are inherited from the `bipartite_graph_base` class, which contains the vertex cardinality information of the two partitions in the bipartite graph. Note that, due to two separate index spaces, both the maximum No. of vertices ($n0$) and the maximum No. of hyperedges ($n1$) information may be required to construct the bipartite graphs. The NWHy-specific API calls to construct a hypergraph bi-adjacency is shown in Listing 2. The constructed bi-adjacencies (`hyperedges` and `hypernodes`) can then be iterated over as a C++20 range of ranges, as demonstrated in listing 3.

The bi-adjacency list of the hypergraph is the most commonly adopted hypergraph representations. Hypergraph libraries including Hygra [25], CHGL [13], MESH [11] and

Listing 1 *Hypergraph data structures*

```
class bipartite_graph_base {
public:
 bipartite_graph_base(size_t n0, size_t n1)
  : vertex_cardinality_(n0, n1) {}
protected:
 std::array<size_t, 2> vertex_cardinality_;
};
template<class... Attributes>
class biedgelist : public bipartite_graph_base {
public:
 biedgelist(size_t n0 = 0, size_t n1 = 0);
 auto num_vertices();
 auto num_edges();
 ...
private:
 std::tuple<std::vector<Attributes>...> base_;
};
template<class... Attributes>
class biadjacency : public bipartite_graph_base {
public:
 biadjacency(biedgelist& el);
 auto num_vertices();
 std::vector<size_t> degrees();
 iterator begin();
 iterator end();
 inner_range operator[](size_t i);
 ...
private:
 size_t N_;
 std::vector<size_t> indices_;
 std::tuple<std::vector<Attributes>...> indexed_;
};
```

Listing 2 *APIs for constructing different representations of a hypergraph*

```
//Hypergraph as a bipartite graph
biedgelist bi_el = graph_reader(mm_file);
biadjacency<0> hyperedges(bi_el);
biadjacency<1> hypernodes(bi_el);
//Adjoin (hyper)graph indexed in one index set
size_t nrealedges = 0, nrealnodes = 0;
edge_list adjoin_el =
 graph_reader_adjoin(mm_file, nrealedges, nrealnodes);
adjacency<0> adjoin_graph(adjoin_el);
//Clique expansion graph of hypergraph
edgelist onelinegraph_els =
 to_two_graph_hashmap_cyclic(hypernodes, hyperedges,
degrees(hypernodes), 1, num_threads, num_bins);
adjacency<0> clique_expansion_graph(onelinegraph_els);
//s-line graph of hypergraph for a given s
edgelist slinegraph_els =
 to_two_graph_hashmap_cyclic(hyperedges, hypernodes,
degrees(hyperedges), s, num_threads, num_bins);
adjacency<0> slinegraph(slinegraph_els);
```

Listing 3 *API for iterating over a hypergraph, represented as bi-adjacencies*

```
//Traverse hyperedges and their incident hypernodes
for (auto hyperE = 0;
     hyperE < num_vertices(hyperedges, 0);
     ++hyperE) {
    for (auto e : hyperedges[hyperE])
        auto hyperN = target(e);
}
//Traverse every neighborhood of hyperedges
for (auto e_neighbors : hyperedges) {
    for (auto e : e_neighbors)
        auto hyperN = target(e);
}
```

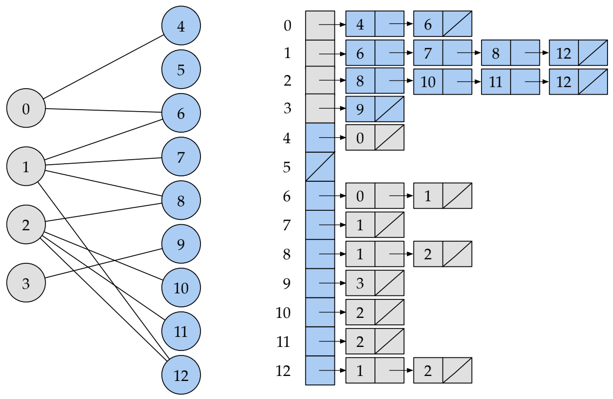


Fig. 3: *The bi-adjacency list of the hypergraph in Figure 1a can be indexed with a single index set. Both the hyperedge incidence and the hypernode incidence are represented with a single incidence list.*

HyperX [14] provide this representation for hypergraphs.

a) Rectangular Matrix Operation Support

Many of the hypergraph algorithms are operated on the incidence matrix of a hypergraph. This is generally different than graph algorithms based on adjacency matrices. When considering the matrix abstraction for the adjacency and incidence matrices, it is important to note that adjacency matrices are square matrices while incidence matrices are generally rectangular (due to the facts that the number of hyperedges and the number of hypernodes may be different, and they constitute two different index spaces because they represent different entities, such as author-paper or community-member relations). Hence hypergraph libraries need to support rectangular matrices efficiently. NWHy does not assume the matrix dimension to be the same and provides support for rectangular matrix computation.

2) Hypergraphs as Adjoined Graphs: Indexed as One Index Set

In NWHy, we present an alternative way to represent a hypergraph, by consolidating the two separate index spaces into one single, shared index space for hypernodes and hyperedges. We call the resultant graph an *adjoined graph* of a hypergraph. For instance the adjoin graph of Figure 1a is shown in Figure 3. Here, the IDs of hyperedges are from 0 to 3, while the IDs of the hypernodes are from 4 to 12. Hypergraphs indexed in a single index set is essentially a general graph. This hypergraph representation requires the hypergraph algorithms processing it to be range-aware, i.e., the algorithms must be aware of which part of the index set corresponds to the hyperedges, and which part to the hypernodes.

To convert the bi-adjacency of a hypergraph H, specifically its bipartite graph form B, into an *adjoin graph* $G = (V, E)$, the disjoint vertex sets $U(B)$ and $V(B)$ are re-indexed to a single index set $V(G)$, as follows. The set $V(G)$ is the direct sum of $U(B)$ and $V(B)$, and $E(G) \subseteq V(G) \times V(G)$. The elements of $E(G)$ maintain a certain structure due to the origination of G as a bipartite graph (i.e., edges represent connections only between $U(B)$ and $V(B)$). Let $I_U : V(G) \rightarrow V(G)$ be the projection operator of $V(G)$ to $U(B)$ and let $I_V : V(G) \rightarrow$

$$\begin{array}{c|ccccccccccccc}
 & e_0 & e_1 & e_2 & e_3 & v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \\
\hline
e_0 & & & & & 1 & & 1 & & & & & & \\
e_1 & & & & & & & 1 & 1 & 1 & & & & 1 \\
e_2 & & & & & & & & & 1 & & 1 & 1 & 1 \\
e_3 & & & & & & & & & & 1 & & & \\
v_0 & 1 & & & & & & & & & & & & \\
v_1 & & & & & & & & & & & & & \\
v_2 & 1 & 1 & & & & & & & & & & & \\
v_3 & & 1 & & & & & & & & & & & \\
v_4 & & 1 & 1 & & & & & & & & & & \\
v_5 & & & & 1 & & & & & & & & & \\
v_6 & & & 1 & & & & & & & & & & \\
v_7 & & & 1 & & & & & & & & & & \\
v_8 & & 1 & 1 & & & & & & & & & &
\end{array}$$

Fig. 4: *The adjacency matrix of the adjoined graph G of the hypergraph H in Figure 1a*

$V(G)$ be the projection operator of $V(G)$ to $V(B)$. Then, any element $e = \{u, w\} \in E(G)$ will have the following property:

$$(I_U(u) = u \wedge I_V(w) = w) \vee (I_V(u) = u \wedge I_U(w) = w)$$

The graph G is called an *adjoined graph* of a hypergraph H or a bipartite graph B.

For example, a square adjacency matrix of the adjoined graph G of the hypergraph H in Figure 1a is shown in Figure 4. Observe that adjacency matrix A_G of G 1) is a sparse matrix; 2) is symmetric; 3) and has the following form:

$$A_G = \begin{pmatrix} 0 & B_H^t \\ B_H & 0 \end{pmatrix}$$

where B_H is the incidence matrix of H, B_H^t is the incidence matrix of its dual H^*. Because A_G is sparse, G better be stored in an adjacency list or CSR representation to be space-efficient. The NWHy-specific API calls to construct an adjoined graph is shown in Listing 2.

In such an adjoin graph representation of a hypergraph, due to the shared index set, any graph algorithms can be used to compute hypergraph metrics. However, this is under the assumption that the algorithms know the ranges of the shared index set of the hyperedges and the hypernodes. After the graph algorithms computed the hypergraph metrics, we split the resultant array of the graph algorithms into the hyperedge resultant array and the hypernodes resultant array respectively.

One of the drawbacks of the adjoin graph of a hypergraph is that this representation cannot adopt relabel-by-degree for performance optimization. Relabel-by-degree (also known as permute-by-row/column) is a trick to improve the workload distribution and memory access pattern in graph algorithms or matrix-matrix multiplication [9], [19]. Relabel-by-degree relabels the vertices of a graph based on their degrees in descending order, so that the high-degree vertices will be given smaller IDs and the low-degree vertices will obtain larger IDs; or vice versa in ascending order. For an adjoin graph of a hypergraph, after relabel-by-degree, the IDs of the hyperedges and the hypernodes are mixed together, hence indistinguishable. We propose a solution in Section III-C to address this issue.

3) *Clique-expansion Graphs of Hypergraphs*

The third representation of a hypergraph in NWHy is the clique expansion graph [29]. The clique expansion replaces each hyperedge with a graph edge between each pair of vertices incident on the hyperedge. Once a clique-expansion graph of a hypergraph is computed, any graph algorithm can be leveraged to compute different metrics. NWHy supports clique expansion construction. The NWHy-specific API calls to construct a clique-expansion graph is shown in Listing 2.

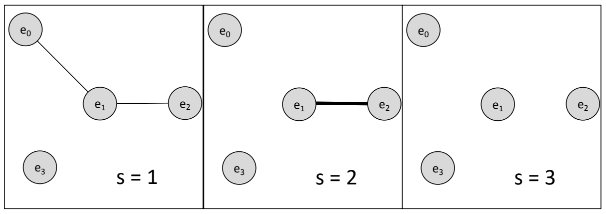


Fig. 5: *There are three s-line graphs of the hypergraph in Figure 1a in which the vertices are the hyperedges of the original hypergraph. The width of the graph edges represents the strength of the connection in the original hypergraph.*

However, there are several well-known drawbacks of clique expansion. First, the inclusion information of the hypernodes in the hyperedges of the original hypergraph is lost in the clique-expansion graph [15]. Moreover, the construction of the clique-expansion graph is both computation-heavy and memory-intensive. The size of the clique-expansion graph increases exponentially compared to its original hypergraph representation, which significantly limits the scalability and usability of clique expansion [11], [14]. Finally, the clique-expansion graph either retains information related to the hypernodes or the hyperedges. Hence only partial information is retained in each case. Computing hypergraph metrics in this way can be expensive too.

4) *s-line Graphs of Hypergraphs*

An important alternative representation of hypergraphs that is available in NWHy is the s-line graph of a hypergraph ([2], [17], [18]), a low order approximation of the original hypergraph. To construct an s-line graph for a particular value of s, each hyperedge in the original hypergraph is considered as a vertex of the newly-constructed line graph. Whenever there are at least s common neighbors between a hyperedge pair in the original hypergraph, a graph edge is created between the vertices in the s-line graph that represent the hyperedge pair in the original hypergraph. s-line graphs capture the strength of connections among hyperedges. The s-line graphs of the hypergraph in Figure 1a are shown in Figure 5.

Dually, an s-clique graph construction considers the hypervertices whenever they join s or more shared hyperedges. Note that the clique-expansion graph is a special case of the s-clique graph for s=1. The 1-line graph of the dual hypergraph is the clique-expansion graph of the original hypergraph. One of the APIs available in NWHy to construct an s-line graph is shown in Listing 2. Once the s-line graph is computed, any graph algorithm can be applied to compute any metric of interest (such as breadth-first search, connected component, etc.).

Aksoy *et al.* have developed various s-line graph metrics on the basis of s-walks [2]. An s-walk is a random walk on the s-line graph. However, s-line graph construction algorithm was not outlined in [2]. In previous works [17], [18], we

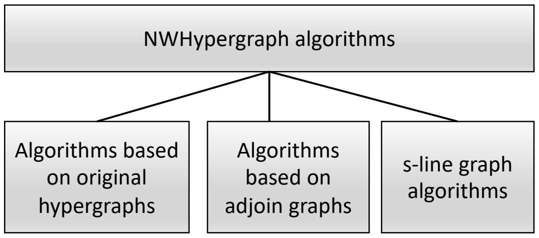

NWHypergraph algorithms
Algorithms based on original hypergraphs
Algorithms based on adjoin graphs
s-line graph algorithms

Fig. 6: *Different categories of hypergraph algorithms in NWHy dealing with different representations of hypergraphs.*

have proposed multiple efficient s-line graph construction algorithms and have incorporated them in NWHy. We have also demonstrated the applicability and effectiveness of using s-line graphs and s-metrics to approximate the hypergraph metrics, even though information loss is existent. The detail definition of those metrics is omitted here due to space limit. We refer our readers to [2], [17], [18] for detailed discussion. In addition to the algorithms presented in [17], [18], two new, alternative algorithms for computing an s-line graph will be presented in Section III-C. NWHy also includes several approximate hypergraph algorithms, including s-connected component, s-betweenness centrality, s-single source shortest path, etc., based on s-line graphs.

C. Algorithms

Algorithms implemented in NWHy can be divided into three categories: 1) algorithms that directly operate on the bipartite representation of the original hypergraph, 2) algorithms that execute on the adjoin representation, and 3) algorithms computing different metrics on the s-line graphs and clique expansion graphs, which are approximations of the original hypergraph (Figure 6). The first two categories of the algorithms can compute the exact results (since they operate on the original hypergraphs). The third category of algorithms computes the approximate results for different metrics, based on graph algorithms available in our graph library, NWGraph.

1) Hypergraph Algorithms based on Original Hypergraphs

In NWHy, we implemented Breadth-first Search (BFS) and Connected Component (CC) algorithms for hypergraphs represented as bipartite graphs, namely, HyperBFS and HyperCC. Our HyperBFS implementations include a Top-down BFS and a Bottom-up BFS implementation [5]. Our HyperCC implementation is based on Label Propagation ([22], [28]).

2) Hypergraph Algorithms for Adjoin Graphs of Hypergraphs

NWHy also includes BFS and CC algorithms for adjoin graph representation of hypergraphs, namely, AdjoinBFS and AdjoinCC. AdjoinBFS processes adjoin graphs of hypergraphs with a graph algorithm implemented with Direction-optimizing BFS. AdjoinCC implements the well-known Afforest algorithm [27], and the Label Propagation Algorithm.

3) s-line Graph Construction Algorithms

Compared to other hypergraph libraries, a unique feature of NWHy is that it includes s-line graph construction and s-metric computation algorithms. Several s-line graph construction algorithms are available in NWHy. These include a naive approach that considers all possible pairs of hyperedges to compute overlaps, the heuristic based set intersection algorithm [17], the hashmap-based counting algorithm [18], and the ensemble algorithm [18] for computing an ensemble of s-line graphs. In addition to these four s-line graph construction algorithms, in this subsection we propose two additional ways to compute s-line graphs.

Algorithm 1 A single-phase queue-based algorithm to compute the edge list of an s-line graph of a hypergraph for a given s using a hashmap data structure. The hypergraph can be either a bipartite graph or an adjoin graph.
Input: Hypergraph $H = (V, E)$, s
Output: s-line graph edge list $L_s(H)$

```
1: for all hyperedge $e_i \in E$ do in parallel
2:     $queue \leftarrow \{e_i\}$        ▷ ID can be original or permuted
3: $L_s(H) \leftarrow \emptyset$
4: $L_t(H) \leftarrow \emptyset$, for each thread $t$
5: for all $e_i \in queue$ do in parallel
6:     if $degree[e_i] < s$ then
7:         continue
8:     overlap_count ← []
9:     for each vertex $v_k$ of $e_i$ do
10:        for each hyperedge $e_j$ of $v_k$ where $(i < j)$ do
11:            overlap_count[$e_j$]++
12:    for each $[e_j, n] \in$ overlap_count do
13:        if $n \geq s$ then
14:            $L_t(H) \leftarrow L_t(H) \cup \{e_i, e_j\}$
15: $L_s(H) \leftarrow L_s(H) \cup$ every $L_t(H)$
16: return $L_s(H)$
```

Algorithms proposed in [17], [18] iterate over the entire hyperedge set in the outermost for loop of the algorithm. This loop iterates over contiguous $[0, ..., n_e - 1]$, and assumes that the ID spaces of hyperedges and hypervertices are separate. Both the set-intersection based and hashmap-based algorithms are three nested for loops and the amount of work in the innermost loop is dictated by the pattern: for each hyperedge $e_i \in E$, for each incident hypernode v_j of e_i, for each hyperedge e_j that is incident to v_j. This "indirection" pattern is the determinant of the available workload per thread. Once the workload partition is decided for the outermost parallel for loop, the amount of work that will be executed by the threads becomes fixed. Hence, these one-phase, non-queue-based algorithm may still suffer from uneven workload distribution. Relabeling the hyperedges by degree may improve the workload distribution and memory access pattern. However, since all of these algorithms execute under the assumption that the hyperedges and hypernodes ID spaces are disjoint, i.e., the hyperedges IDs in the ranges of $[0, ..., n_e - 1]$ and the hypernodes IDs in the range of $[0, ..., n_v - 1]$ denote two different ID spaces, they are not directly applicable to the adjoin graph representation of hypergraphs, indexed as a single index set, namely the hyperedges and hypernodes uses IDs in the ranges of $[0, ..., n_e - 1, n_e, ..., n_e + n_v - 1]$. Moreover, techniques such as relabel-by-degree can't be applied on the adjoin graphs, since in doing so the IDs will be intermingled and IDs of

Algorithm 2 A two-phase queue-based algorithm to compute the edge list of an s-line graph of a hypergraph for a given s. The hypergraph can be either a bipartite graph or an adjoin graph.
Input: Hypergraph $H = (V, E)$, s
Output: s-line graph edge list $L_s(H)$

```
1: $queue_t \leftarrow \emptyset$, for each thread $t$
2: for all hyperedge $e_i \in E$ do in parallel
3:     for each vertex $v_k$ of $e_i$ do
4:         for each hyperedge $e_j$ of $v_k$ where $(i < j)$ do
5:             $queue_t \leftarrow \{e_i, e_j\}$      ▷ ID can be original or permuted
6: $queue \leftarrow queue \cup$ every $queue_t$
7: $L_s(H) \leftarrow \emptyset$
8: $L_t(H) \leftarrow \emptyset$, for each thread $t$
9: for all $\{e_i, e_j\} \in queue$ do in parallel
10:     count $\leftarrow set\_intersection(neighbor\_list(e_i), neighbor\_list(e_j))$
11:     if count $\geq s$ then
12:         $L_t(H) \leftarrow L_t(H) \cup \{e_i, e_j\}$
13: $L_s(H) \leftarrow L_s(H) \cup$ every $L_t(H)$
14: return $L_s(H)$
```

hyperedges and hypernodes will be thus indistinguishable.

To address these problems when hyperedge ID spaces are not necessarily within $[0, ..., n_e - 1]$ range, we propose to add a work queue variant of the existing algorithms in our framework. This solution is simple yet versatile. Independent of the hypergraph representation and application of degree-based relabeling to the IDs, a queue with all the potential hyperedge IDs are constructed at the beginning of the s-line graph construction algorithms. Based on this idea, we introduce two new queue-based algorithms to compute an s-line graph of a hypergraph for a given s, Algorithm 1 and Algorithm 2.

Algorithm 1 is based on counting using a hashmap data structure. Instead of processing the hyperedges from $[0, ..., n_e - 1]$ in [18] (aka instead of assuming contiguous IDs for hyperedges starting from 0 for hyperedges), we enqueue the hyperedge IDs into a work queue at Line 2 of Algorithm 1 (thus eliminating the fixed iteration structure of the for loop starting from 0 and iterating up to $n_e - 1$). Next the algorithm processes each hyperedge ID fetched from the queue. Since enqueuing the hyperedges into a work queue is linear to the number of the hyperedges, hence the time complexity remains the same as the original hashmap-based counting algorithm.

Algorithm 2 is based on set intersection of the hyperedge pairs and can be considered as a two-phase algorithm. In the first phase, instead of processing the hyperedges from $[0, ..., n_e - 1]$ in one go, Algorithm 2 considers the eligible (degree $\geq s$) hyperedge pairs and enqueues them into a work queue. In the second phase, it counts the number of common hypernodes by performing set intersection between the neighbor lists of each hyperedge pairs in the queue. Note that, the second phase has only one for loop (barring the set intersection), and may lend itself to better load balancing among the threads compared to its non-queue-based intersection version, since the control of granularity for workload per thread is more fine-grained.

Algorithm 3 An algorithm to find the toplexes of a hypergraph.
Input: Hypergraph $H = (V, E)$
Output: Toplexes $\check{E} = \{\nexists f \supseteq e, e \in E\}$

```
1: $\check{E} \leftarrow \emptyset$
2: for all $e_i \in E$ do in parallel
3:     flag $\leftarrow True$
4:     for each $e_j \in \check{E}$ such that $(i < j)$ do
5:         if $e_i \subseteq e_j$ then
6:             flag $\leftarrow False$; break
7:         if $e_j \subseteq e_i$ then
8:             $\check{E} \leftarrow \check{E} \setminus \{e_j\}$
9:     if flag $= True$ then
10:         $\check{E} \leftarrow \check{E} \cup \{e_i\}$
11: return $\check{E}$
```

Listing 4 *Different ways of iterating over a hypergraph in parallel, represented as bi-adjacencies*

```
//Traverse every hyperedge and its incident hypernodes
//in parallel using std::for_each and execution policy
std::for_each(std::execution::par_unseq,
 hyperedges.begin(),
 hyperedges.end(), [&](auto& e_neighbors) {
    std::for_each(std::execution::par_unseq,
     e_neighbors.begin(),
     e_neighbors.end(), [&](auto& e) {
        auto hyperN = target(e);
    });});
//Using tbb::parallel_for with built-in
//tbb::blocked_range adaptor and tbb::auto_partitioner
tbb::parallel_for(
 tbb::blocked_range(0, num_vertices(hyperedges, 0)),
 [&](auto r) {
    for (auto hyperE = r.begin(); hyperE != r.end();
     ++hyperE) {
        for (auto e : hyperedges[hyperE]) {
            auto hyperN = target(e);
        } }
}, tbb::auto_partitioner());
//Using tbb::parallel_for with our customized
//cyclic_neighbor_range adaptor and tbb::auto_partitioner
tbb::parallel_for(cyclic_neighbor_range(hyperedges,
 num_bins), [&](auto r) {
    for (auto j = r.begin(); j != r.end(); ++j) {
        auto&& [hyperE, e_neighbors] = *j;
        for (auto e : e_neighbors) {
            auto hyperN = target(e);
        } }
}, tbb::auto_partitioner());
```

4) *Toplex Computation*

In hypergraphs, there exists an inclusion relationship between two hyperedges $e \subseteq f$, or $f \subseteq e$. A *toplex* is a maximal hyperedge e such that $\nexists f \supseteq e$. We include an algorithm to compute the toplexes of hypergraphs in NWHy (Algorithm 3).

D. *Parallelization*

To parallelize different algorithms and to iterate over the ranges in parallel, NWHy can leverage C++ `std::for_each` with parallel execution policies (`std::execution::par_unseq`). However, real-world hypergraphs have skewed-degree distribution and C++ standard library currently does not provide any mechanism for controlling workload distribution among threads. Hence, as an alternative, we use oneAPI Threading Building Blocks (oneTBB) [21] to parallelize our algorithms. oneTBB is based on a work-stealing scheduler and is better suited for load balancing, since it provides the user with ways to specify workload distribution strategies (blocked, cyclic or any customized partitioning of work among the threads) and granularity of tasks. The built-in blocked range, for example, divides the hyperedges (IDs) into blocks/chunks of contiguous IDs and each chunk of contiguous hyperedges (IDs) can be assigned to one thread. Work stealing scheduling strategy is particularly beneficial for our library, since this enables idle threads to steal work from other straggler threads processing a hypergraph with skewed-degree distribution. We show how users can iterate over a hypergraph using these different approaches in Listing 4.

However, the built-in blocked range partitioning may be problematic for skewed-degree distributed hypergraphs, especially if the hyperedges/hypernodes are sorted according to their degrees. Here some of the threads will have highly-unbalanced workload due to assignment of high-degree hyperedges to first few threads. To circumvent this problem, We provide several custom partitioning strategies as range adaptors in our library.

In one of these custom range adaptor, named *cyclic range*, 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.

Due to the facts that some of the hypergraph algorithms require neighborhood access, we also provide another adaptor with a built-in partitioning strategy called *cyclic neighbor range* adaptor. Cyclic neighbor range is very similar to cyclic range, which partitions the hypergraph as a range in cyclic fashion. The main difference between a cyclic range adaptor and a cyclic neighbor range adaptor over a hypergraph is that a cyclic range adaptor returns one hyperedge at a time, while a cyclic neighbor range adaptor returns a tuple, which consists of one hyperedge and the hypernodes as a neighborhood that hyperedge is incident to.

E. Python APIs.

To provide data scientists with the capability to interact with our high-performance C++ backend easily, we have created a Python package called *nwhy* and provide sufficient Python APIs to our C++ code with pybind11 [23]. Pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa. Our Python package has been released on PyPI and can be installed through `pip install nwhy`, assuming oneTBB has been installed. A minimal working example using these APIs is shown in Listing 5.

Listing 5 *An example of using the Python APIs to interact with our C++ backend.*

```
import numpy as np
import nwhy
col = np.array([0, 0, 0, 1, 1, 1])
row = np.array([0, 1, 2, 0, 1, 2])
weight = np.array([1, 1, 1, 1, 1, 1])
#create a hypergraph hg
hg = nwhy.NWHypergraph(row, col, weight)
#compute the s-line graph of hg with s=2
s2lg = hg.s_linegraph(s=2, edges=True)
#query whether the 2-line graph is 2-connected
tmp = s2lg.is_s_connected()
#query the neighboring hyperedges of hyperedge 0
sn = s2lg.s_neighbors(v=0)
#query s-degree of hyperedge 0
sd = s2lg.s_degree(v=0)
#compute s-connected components
scc = s2lg.s_connected_components()
#compute s-distance between hyperedge 0 and 1
sdist = s2lg.s_distance(src=0, dest=1)
#compute s-path between hyperedge 0 and 1
sp = s2lg.s_path(src=0, dest=1)
#compute s-betweenness centrality
sbc = s2lg.s_betweenness_centrality(normalized=True)
#compute s-closeness centrality
sc = s2lg.s_closeness_centrality(v=None)
#compute s-harmonic closeness centrality
shc = s2lg.s_harmonic_closeness_centrality(v=None)
#compute s-eccentricity
se = s2lg.s_eccentricity(v=None)
```

IV. Experimental Analysis

In this section, we evaluate the performance of our hypergraph algorithms for bipartite graph representations (HyperCC, HyperBFS) and for adjoin graphs of hypergraph (AdjoinCC, AdjoinBFS). We compare the performance of our algorithms with the ones that are available in the Hygra framework [25], called HygraCC and HygraBFS. HygraCC is a Label Propagation-based connected component algorithm. HygraBFS is a Top-down BFS algorithm. We also evaluate and report the performance of our new s-line graph construction algorithms with the previous non-queue versions in [17], [18].

A. Experimental Setup

Our experiments are run on a machine with a two-socket Intel Xeon Gold 6230 processor, with 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 oneTBB 2021.4, and compiled with GCC 11 compiler and `-Ofast -march=native` compilation flags. Hygra is compiled with `-fopenmp -O3 -march=native` compilation flags.

Type	hypergraph	$\|V\|$	$\|E\|$	$\bar{d}_v$	$\bar{d}_e$	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
	Orkut-group	2.8M	8.7M	118	37	40k	318k
	LiveJournal	3.2M	7.5M	35	15	300	1.1M
Web	Web	27.7M	12.8M	5	11	1.1M	11.6M
Synthetic dataset	Rand1	100M	100M	10	10	34	10

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

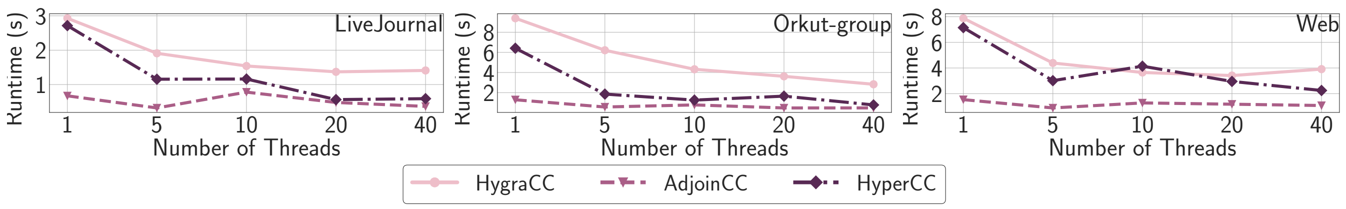


Fig. 7: *Strong scaling results of hypergraph connected component decomposition.*

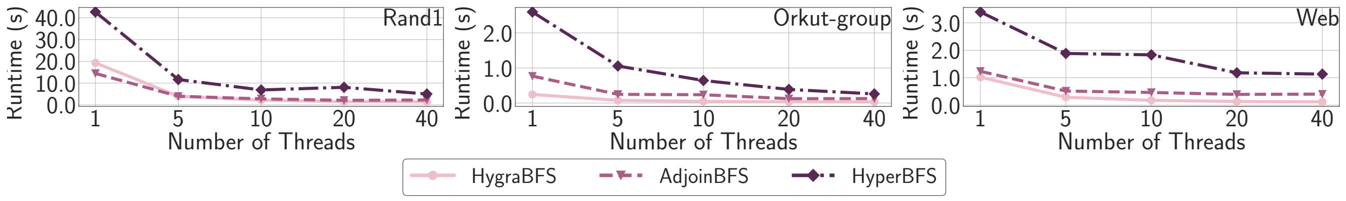


Fig. 8: *Strong scaling results of hypergraph breadth-first search.*

B. Dataset

We conducted experiments with real-world hypergraphs (Table I) curated in [25]. For these curated datasets, in particular, each hypergraph, constructed from the social network datasets such as com-Orkut and Friendster in Table I, are materialized by running a community detection algorithm on the original dataset obtained 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 hypernode. Other larger datasets include orkut-groups, Web, and LiveJournal, collected from Koblenz Network Collection (KONECT) [16] as bipartite graphs. The synthetic random hypergraph Rand1 is generated using Hygra [25]. For Rand1, the hypervertices for each of the hyperedge are chosen uniformly at random.

C. Strong Scaling Results of Hypergraph Algorithms with Bipartite Representation

We conduct strong scaling experiments for our hypergraph algorithms AdjoinCC, AdjoinBFS, HyperCC, and HyperBFS with different hypergraph inputs. Here we double the number of threads while keeping the input size constant. The results of CC is reported in Figure 7, and the results of BFS is reported in Figure 8. Our connected component algorithms overall demonstrate better performance and scalability. The performance of our BFS algorithm on adjoin graph is comparable to the BFS algorithm in Hygra for hypergraphs with uniform degree distribution (Rand1). The execution time of BFS with Orkut-group and Web are small due to the fact that there are a lot of connected components in the hypergraph and the traversals cover the connected components pretty fast. On the other hand, Rand1 only contains one single connected component, hence it takes significant amount of time to traverse the whole hypergraph.

D. Performance Comparison of s-line Graph Construction Algorithms

We compare our queue-based s-line graph algorithms with the Intersection algorithm in [17], and the Hashmap algorithm in [18]. We run experiments with both blocked range and cyclic range partitioning strategies along with ID relabel-by-degree in ascending/descending order. Among those runtime results, we only report the fastest among each algorithm. The normalized runtime results are reported in Figure 9, the execution time is normalized w.r.t. the execution time of the Hashmap-based algorithm. As we can see from the plot, our single-phase queue-based algorithm has similar performance as its best-performant non-queue-based version (Hashmap vs Algorithm 1, Intersection vs Algorithm 2). Yet our queue-based algorithms are more versatile about different hypergraph representations (either indexed in one index set, or in two indexed sets).

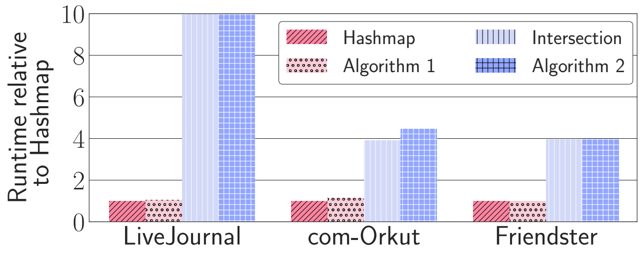


Fig. 9: *Runtime performance relative to the Hashmap algorithm for s-line graph computation.*

V. Related Work

For further background and discussion of the concepts of graphs, bipartite graphs and hypergraphs, we refer the reader to the books by Berge [6] and Bretto [8] for hypergraph theory; Bondy and Murty [7] for graph theory. In this section, we review related works focusing primarily on hypergraph frameworks, hypergraph representations, and algorithms.

Hypergraph frameworks. Jenkins *et al.* presented Chapel-based CHGL [13], a high-performance library for hypergraph computation. CHGL is only a prototype with abstract interfaces and it mainly focuses on random hypergraph generation. Shared-memory C++-based framework Hygra [25], and distributed-memory frameworks Apache Spark-based MESH [11] and HyperX [14] presented a collection of efficient parallel algorithms for hypergraphs in their frameworks, including algorithms for betweenness centrality, maximal independent set, k-core decomposition, hypertrees, hyperpaths, connected-components, PageRank, and single source shortest paths. HyperNetX [12] computes s-line graph naively in Python, alternatively it can use our NWHy Python APIs for s-line graph construction and s-metric computation.

Hypergraph representation in various frameworks. In Hygra and MESH, hypergraphs are represented as a bipartite graph, where one part comprises exclusively of hypernodes, the other exclusively of hyperedges. These two libraries, as well as CHGL, index hypergraphs as two separate sets. MESH and HyperX support hypergraph representation as a clique-expansion graph. None of the hypergraph frameworks support hypergraphs as a single index set. Nor do they support s-line graphs of hypergraphs.

Algorithms for graphs and hypergraphs. For hypergraph algorithms, Hygra has breadth-first search and a label-propagation-based connected component decomposition implemented. For graph algorithms, Shiloach and Vishkin [24] introduced the first parallel algorithm (SV CC) to find connected components in general graphs using the Parallel Random Access Machine (PRAM) model. Afforest [27] improved SV CC by skipping the largest component in the graph after a subgraph sampling step to reduce the number of edges visited. Minimum label propagation approach is another parallel method for CC [22], [28].

VI. Conclusion

We present a scalable framework for hypergraph analytics, called NWHypergraph (NWHy). NWHy supports four different representations and the corresponding data structures for these representations of hypergraphs: bipartite graphs, adjoin graphs, clique-expansion graphs, and s-line graphs. We implement a set of hypergraph algorithms in NWHy for exact and approximate hypergraph analytics. We propose a simple yet effective technique, called adjoin, to consolidate two separate ID spaces of hypergraphs to enable more flexible algorithm design. Experimentally, we demonstrate the scalability and performance of our CC and BFS algorithms operating on adjoin graphs of hypergraphs as well as algorithms operating on bipartite graphs, and show competitive performance as Hygra. To accommodate different representations of hypergraphs in constructing s-line graphs, we propose two new queue-based s-line graph construction algorithms to demonstrate the effectiveness of work-queue based algorithm design. We show that our new queue-based algorithms demonstrate similar performance to their non-queue-based algorithms.

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-171117. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

References

[1] S. Agarwal, K. Branson, and S. Belongie, "Higher order learning with graphs," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06. ACM, 2006, p. 17–24.

[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] I. Amburg, J. Kleinberg, and A. R. Benson, "Planted hitting set recovery in hypergraphs," *J. Phys. Complex.*, vol. 2, no. 3, p. 035004, 2021.

[4] A. Azad, M. M. Aznaveh, S. Beamer, M. Blanco *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.

[5] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[6] C. Berge, *Hypergraphs*, 1st ed. North Holland, 1989, vol. 45.

[7] J. A. Bondy and U. S. R. Murty, *Graph Theory*. Springer London, 2008, vol. 244.

[8] A. Bretto, *Hypergraph Theory: An Introduction*, ser. Mathematical Engineering. Springer International Publishing, 2013.

[9] 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.

[10] Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao, "Hypergraph neural networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 3558–3565, Jul. 2019.

[11] 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.

[12] "HyperNetX." [Online]. Available: https://github.com/pnnl/HyperNetX

[13] 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.

[14] 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.

[15] S. Kirkland, "Two-mode networks exhibiting data loss," *J Complex Networks*, vol. 6:2, pp. 297–316, 2017.

[16] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd Intl. Conference on World Wide Web*, 2013, pp. 1343–1350.

[17] X. T. Liu, J. Firoz, and et al., "Parallel algorithms for efficient computation of high-order line graphs of hypergraphs," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 312–321. [Online]. Available: https://arxiv.org/abs/2010.11448

[18] ——, "High-order line graphs of non-uniform hypergraphs: Algorithms, applications, and experimental analysis," in *Proc. of the 36th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2022, p. In press. [Online]. Available: https://arxiv.org/abs/2201.11326

[19] 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.

[20] E. Niebler, C. Carter, and C. Di Bella, "The one ranges proposal," Tech. Rep., 2018. [Online]. Available: https://wg21.link/p1035

[21] oneTBB. [Online]. Available: https://github.com/oneapi-src/oneTBB

[22] S. M. Orzan, "On distributed verification and verified distribution," Ph.D. thesis, VRIJE UNIVERSITEIT, Nov 2004.

[23] "pybind11." [Online]. Available: https://github.com/pybind/pybind11

[24] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[25] 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.

[26] "Stanford large network dataset collection." [Online]. Available: http://snap.stanford.edu/data

[27] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 12–21.

[28] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1821–1832, 2014.

[29] 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.