Practical Parallel Hypergraph Algorithms

Julian Shun
jshun@mit.edu
MIT CSAIL

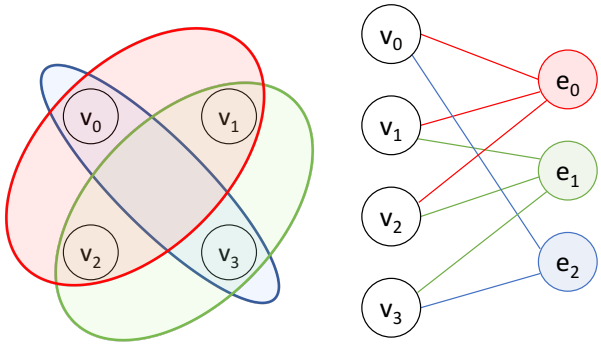
Abstract

While there has been significant work on parallel graph processing, there has been very surprisingly little work on high-performance hypergraph processing. This paper presents a collection of efficient parallel algorithms for hypergraph processing, including algorithms for betweenness centrality, maximal independent set, k-core decomposition, hypertrees, hyperpaths, connected components, PageRank, and single-source shortest paths. For these problems, we either provide new parallel algorithms or more efficient implementations than prior work. Furthermore, our algorithms are theoretically-efficient in terms of work and depth. To implement our algorithms, we extend the Ligra graph processing framework to support hypergraphs, and our implementations benefit from graph optimizations including switching between sparse and dense traversals based on the frontier size, edge-aware parallelization, using buckets to prioritize processing of vertices, and compression. Our experiments on a 72-core machine and show that our algorithms obtain excellent parallel speedups, and are significantly faster than algorithms in existing hypergraph processing frameworks.

CCS Concepts • **Computing methodologies** → **Parallel algorithms**; **Shared memory algorithms**.

1 Introduction

A graph contains vertices and edges, where a vertex represents an entity of interest, and an edge between two vertices represents an interaction between the two corresponding entities. There has been significant work on developing algorithms and programming frameworks for efficient graph processing due to their applications in various domains, such as social network and Web analysis, cyber-security, and scientific computations. One limitation of modeling data using graphs is that only binary relationships can be expressed, and can lead to loss of information from the original data. Hypergraphs are a generalization of graphs where the relationships, represented as *hyperedges*, can contain an arbitrary number of vertices. Hyperedges correspond to group relationships among vertices (e.g., a community in a social network). An example of a hypergraph is shown in Figure 1a.



(a) Hypergraph **(b)** Bipartite representation

Figure 1. An example hypergraph representing the groups $\{v_0, v_1, v_2\}$, $\{v_1, v_2, v_3\}$, and $\{v_0, v_3\}$, and its bipartite representation.

Hypergraphs have been shown to enable richer analysis of structured data in various domains, such as protein network analysis [76], machine learning [100], and image processing [15, 27]. Various graph algorithms have been extended to the hypergraph setting, and we list some examples of algorithms and their applications here. Betweenness centrality on hypergraphs has been used for hierarchical community detection [12] and measuring importance of hypergraphs vertices [77]. k-core decomposition in hypergraphs can be applied to invertible Bloom lookup tables, low-density parity-check codes, and set reconciliation [45]. PageRank and random walks on hypergraphs have been used for image segmentation and spectral clustering and shown to outperform graph-based methods [26, 27, 100]. Shortest paths, hyperpaths, and hypertrees have been used for solving optimization problems [30, 70], satisfiability problems and deriving functional dependencies in databases [30], and modeling information spread and finding important actors in social networks [31]. Independent sets on hypergraphs have been applied to routing problems [2] and determining satisfiability of boolean formulas [48].

Although there are many applications of hypergraphs, there has been little research on parallel hypergraph processing. The main contribution of this paper is a suite of efficient parallel hypergraph algorithms, including algorithms for betweenness centrality, maximal independent set, k-core

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374527

decomposition, hypertrees, hyperpaths, connected components, PageRank, and single-source shortest paths. For these problems, we provide either new parallel hypergraph algorithms (e.g., betweenness centrality and k-core decomposition) or more efficient implementations than prior work. Additionally, we show that most of our algorithms are theoretically-efficient in terms of their work and depth complexities.

We observe that our parallel hypergraph algorithms can be implemented efficiently by taking advantage of graph processing machinery. To implement our parallel hypergraph algorithms, we made relatively simple extensions to the Ligra graph processing framework [81] and we call the extended framework Hygra. As with Ligra, Hygra is well-suited for *frontier-based algorithms*, where small subsets of elements (referred to as *frontiers*) are processed in parallel on each iteration. We use a bipartite graph representation to store hypergraphs, and use Ligra's data structures for representing subsets of vertices and hyperedges as well as operators for mapping application-specific functions over these elements. The operators for processing subsets of vertices and hyperedges are theoretically-efficient, which enables us to implement parallel hypergraph algorithms with strong theoretical guarantees. Separating the operations on vertices from operations on hyperedges is crucial for efficiency and requires carefully defining functions for vertices and hyperedges to preserve correctness. Hygra inherits from Ligra various optimizations developed for graphs, including switching between different traversal strategies based on the size of the frontier (direction optimization), edge-aware parallelization, bucketing for prioritizing the processing of vertices, and compression.

Our experiments on a variety of real-world and synthetic hypergraphs show that our algorithms implemented in Hygra achieve good parallel speedup and scalability with respect to input size. On 72 cores with hyper-threading, we achieve a parallel speedup of between 8.5–76.5x. We also find that the direction optimization improves performance for hypergraphs algorithms compared to using a single traversal strategy. Compared to HyperX [46] and MESH [41], which are the only existing high-level programming frameworks for hypergraph processing that we are aware of, our results are significantly faster. For example, one iteration of PageRank on the Orkut community hypergraph with 2.3 million vertices and 15.3 million hyperedges [59] takes 0.083s on 72 cores and 3.31s on one thread in Hygra, while taking 1 minute on eight 12-core machines using MESH [41] and 10s using eight 4-core machines in HyperX [46]. Certain hypergraph algorithms (hypertrees, connected components, and single-source shortest paths) can be implemented correctly by expanding each hyperedge into a clique among its member vertices and running the corresponding graph algorithm on the resulting graph. We also compare with this alternative approach by using the original Ligra framework to process the clique-expanded graphs, and show the space usage and performance is significantly worse than that of Hygra (2.8x–30.6x slower while using 235x more space on the Friendster hypergraph).

Our work shows that high-performance hypergraph processing can be done using just a single multicore machine, on which we can process all existing publicly-available hypergraphs. Prior work has shown that graph processing can be done efficiently on just a single multicore machine (e.g., [24, 25, 67, 69, 81, 87, 96, 101]), and this work extends the observation to hypergraphs.

The rest of the paper is organized as follows. Section 2 discusses related work on graph and hypergraph processing. Section 3 describes hypergraph notation as well as the computational model and parallel primitives that we use in the paper. Section 4 introduces the Hygra framework. Section 5 describes our new parallel hypergraph algorithms implemented using Hygra. Section 6 presents our experimental evaluation of our algorithms, and comparisons with alternative approaches. Finally, we conclude in Section 7.

2 Related Work

Graph Processing. There has been significant work on developing graph libraries and frameworks to reduce programming effort by providing high-level operators that capture common algorithm design patterns (e.g., [19, 21, 22, 28, 33–35, 38–40, 42, 47, 51, 56, 60, 61, 63–65, 68, 69, 71–73, 75, 78, 81, 84–87, 91, 94, 96, 99, 101], among many others; see [66, 79, 93] for surveys). Many of these frameworks can process large graphs efficiently, but none of them directly support hypergraph processing.

Hypergraph Processing. As far as we know, HyperX [46] and MESH [41] are the only existing high-level programming frameworks for hypergraph processing, and they are built for distributed memory on top of Spark [95]. Algorithms are written using hyperedge programs and vertex programs that are iteratively applied on hyperedges and vertices, respectively. HyperX stores hypergraphs using fixed-length tuples containing vertex and hyperedge identifiers, and MESH stores the hypergraph as a bipartite graph. HyperX includes algorithms for random walks, label propagation, and spectral learning. MESH includes PageRank, PageRank-Entropy (a variant of PageRank that also computes the entropy of vertex ranks in each hyperedge), label propagation, and single-source shortest paths. Both HyperX and MESH do work proportional to the entire hypergraph on every iteration, even if few vertices/hyperedges are active, which makes them inefficient for frontier-based hypergraph algorithms. The Chapel HyperGraph Library [44] is a library for hypergraph processing that provides functions for accessing properties of hypergraphs, but the interface is much lower-level than the abstractions in HyperX, MESH, and Hygra. It has recently been used to analyze DNS data [1].

Hypergraphs are useful in modeling communication costs in parallel machines, and partitioning can be used to minimize communication. There has been significant work on both sequential and parallel hypergraph partitioning algorithms (see, e.g., [17, 18, 23, 49, 52, 54, 89]). While we do not consider the problem of hypergraph partitioning in this paper, these techniques could potentially be used to improve the locality of our algorithms.

Algorithms have been designed for a variety of problems on hypergraphs, including random walks [27], shortest hyperpaths [3, 70], betweenness centrality [74], hypertrees [30], connectivity [30], maximal independent sets [5, 9, 48, 50], and k-core decomposition [45]. However, there have been no efficient parallel implementations of hypergraph algorithms, with the exception of [45], which provides a GPU implementation for a special case of k-core decomposition.

3 Preliminaries

Hypergraph Notation. We denote an unweighted hypergraph by $H(V, E)$, where V is the set of vertices and E is the set of hyperedges. A weighted hypergraph is denoted by $H = (V, E, w)$, where w is a function that maps a hyperedge to a real value (its weight). The number of vertices in a hypergraph is $n_v = |V|$, and the number of hyperedges is $n_e = |E|$. Vertices are assumed to be labeled from 0 to $n_v - 1$, and hyperedges from 0 to $n_e - 1$. For undirected hypergraphs, we use $deg(e)$ to denote the number of vertices a hyperedge $e \in E$ contains (i.e., its cardinality), and $deg(v)$ to denote the number of hyperedges that a vertex $v \in V$ belongs to. In directed hypergraphs, hyperedges contain incoming vertices and outgoing vertices. For a hyperedge $e \in E$, we use $N^-(e)$ and $N^+(e)$ to denote its incoming vertices and outgoing vertices, respectively, and $deg^-(e) = |N^-(e)|$ and $deg^+(e) = |N^+(e)|$. We use $N^-(v)$ and $N^+(v)$ to denote the hyperedges that $v \in V$ is an outgoing vertex and incoming vertex for, respectively, and $deg^-(v) = |N^-(v)|$ and $deg^+(v) = |N^+(v)|$. We denote the *size* $|H|$ of a hypergraph to be $n_v + \sum_{e \in E}(deg^-(e) + deg^+(e))$, i.e., the number of vertices plus the sum of the hyperedge cardinalities.

Computational Model. We use the work-depth model [43] to analyze the theoretical efficiency of algorithms. The *work* of an algorithm is the number of operations used, and the *depth* is the length of the longest sequence dependence. We assume that concurrent reads and writes are supported.

By Brent's scheduling theorem [14], an algorithm with work W and depth D has overall running time $W/P + D$, where P is the number of processors available. A parallel algorithm is *work-efficient* if its work asymptotically matches that of the best sequential algorithm for the problem, which is important since in practice the W/P term in the running time often dominates.

Parallel Primitives. *Scan* takes an array A of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus x = x$ for any x, and returns the array $(\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus_{i=0}^{n-2} A[i])$ as well as the overall sum, $\perp \oplus_{i=0}^{n-1} A[i]$. *Filter* takes an array A and a predicate f and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order as in A. Scan and filter take $O(n)$ work and $O(\log n)$ depth (assuming $\oplus$ and f take $O(1)$ work) [43].

A *compare-and-swap* CAS($\&x, o, n$) takes a memory location x and atomically updates the value at location x to n if the value is currently o, returning *true* if it succeeds and *false* otherwise. A *fetch-and-add* FAA($\&x, n$) takes a memory location x, atomically returns the current value at x and then increments the value at x by n. A writeMin($\&x, n$) takes a memory location x, and a value n, and atomically updates x to be the minimum of the value at x and n; it returns *true* if the update was successful and *false* otherwise. We assume that these operations take $O(1)$ work and depth in our model, but note that these operations can be simulated work-efficiently in logarithmic depth on weaker models [37].

4 Hygra Framework

This section presents the interface and implementation of the Hygra framework, which extends Ligra [81] to hypergraphs. The Hygra interface is summarized in Table 1.

4.1 Interface

Hygra contains the basic ***VertexSet*** and ***HyperedgeSet*** data structures, which are used to represent subsets of vertices and hyperedges, respectively. ***VertexMap*** takes as input a boolean function F and a VertexSet U, applies F to all vertices in U, and returns an output VertexSet containing all elements from $u \in U$ such that $F(u) = true$. ***HyperedgeMap*** is an analogous function for HyperedgeSets.

VertexProp takes as input a hypergraph H, a VertexSet U, and two boolean functions F and C. It applies F to all pairs (v, e) such that $v \in U$, $e \in N^+(v)$, and $C(e) = true$ (call this subset of pairs P), and returns a HyperedgeSet U' where $e \in U'$ if and only if $(v, e) \in P$ and $F(v, e) = true$. ***HyperedgeProp*** takes as input a hypergraph H, a HyperedgeSet U, and two boolean functions F and C. It applies F to all pairs (e, v) such that $e \in U$, $v \in N^+(e)$, and $C(v) = true$ (call this subset of pairs P), and returns a VertexSet U' where $v \in U'$ if and only if $(e, v) \in P$ and $F(e, v) = true$. For weighted hypergraphs, the F function takes the weight as the third argument.

We provide a function ***HyperedgeFilterNgh*** that takes as input a hypergraph H, a HyperedgeSet U, and a boolean function C, and filters out the incident vertices v for each hyperedge $e \in U$ such that $C(v) = false$. This mutates the hypergraph, so that future computations will not inspect these vertices. ***HyperedgePropCount*** takes as input a hypergraph H, a HyperedgeSet U, and a function F, and applies F to each neighbor of U in parallel. The function F takes two arguments, a vertex v and the number of hyperedges in U

Interface	Description
VertexSet HyperedgeSet	Represents a subset of vertices $V' \subseteq V$. Represents a subset of hyperedges $E' \subseteq E$.
VertexMap(U : VertexSet, F : $vertex \rightarrow bool$) : VertexSet	Applies $F(u)$ for each $u \in U$; returns a VertexSet $\{u \in U \mid F(u) = true\}$.
HyperedgeMap(U : HyperedgeSet, F : $hyperedge \rightarrow bool$) : HyperedgeSet	Applies $F(u)$ for each $u \in U$; returns a HyperedgeSet $\{u \in U \mid F(u) = true\}$.
VertexProp(H : $hypergraph$, U : VertexSet, F : $(vertex \times hyperedge) \rightarrow bool$, C : $hyperedge \rightarrow bool$) : HyperedgeSet	Applies $F(v, e)$ for each $v \in U$, $e \in N^+(v)$ where $C(e) = true$; returns a HyperedgeSet $\{e \mid v \in U, e \in N^+(v), C(e) = true, F(v, e) = true\}$.
HyperedgeProp(H : $hypergraph$, U : HyperedgeSet, F : $(hyperedge \times vertex) \rightarrow bool$, C : $vertex \rightarrow bool$) : VertexSet	Applies $F(e, v)$ for each $e \in U$, $v \in N^+(e)$ where $C(v) = true$; returns a VertexSet $\{v \mid e \in U, v \in N^+(e), C(v) = true, F(e, v) = true\}$.
HyperedgeFilterNgh(H : $hypergraph$, U : HyperedgeSet, C : $vertex \rightarrow bool$)	For each $e \in U$, removes all $v \in N^+(e)$ where $C(v) = false$ from the hypergraph.
HyperedgePropCount(H : $hypergraph$, U : HyperedgeSet, F : $vertex \times int \rightarrow vertex \times int$) : $(vertex \times int)$ $array$	Applies F to pairs (v, cnt), where $v \in N^+(U)$ and cnt is the number of hyperedges in U that v is an outgoing vertex for; returns an array of $(vertex \times int)$ pairs containing the non-null return values of applications of F.
MakeBuckets(n : int, A : int $array$, I : $ordering$) : $buckets$	Creates and returns a bucketing structure that iterates in order I storing n vertices, where vertex v is stored in bucket $A[v]$.
NextBucket(B : $buckets$) : (int, VertexSet)	Returns the bucket number of the next bucket in B and a VertexSet containing the vertices in that bucket.
UpdateBuckets(B : $buckets$, A : $(vertex \times int)$ $array$)	For each $(v, bkt) \in A$, moves vertex v to bucket bkt in B.

Table 1. Summary of the Hygra interface.

that v is an outgoing vertex for, and returns a pair containing a vertex and an integer, either of which can be null ($\perp$). The output of HyperedgePropCount is an array of pairs containing the non-null return values from applications of F.

Hygra also supports the bucketing interface developed in the Julienne framework [24]. Vertices are stored in buckets associated with bucket IDs, and algorithms can process buckets in increasing or decreasing order. Vertices can be moved to different buckets during the computation. The ***MakeBuckets*** function takes a size, an integer array A, and an ordering, and creates a bucketing structure B that stores each vertex v in bucket $A[v]$. The ***NextBucket*** function takes a bucketing structure B and returns the next non-empty bucket in the specified ordering. The ***UpdateBuckets*** function takes a bucketing structure B and an array of pairs (v, bkt), and moves each vertex v from its original bucket to the bucket with ID bkt. Julienne actually groups multiple buckets together into an overflow bucket and thus has a GetBucket function determines the physical bucket from a logical bucket ID, but for simplicity we will not use it in our discussion.

4.2 Implementation

A method for representing hypergraphs is to create a clique among all pairs of vertices in each hyperedge and store the result as a graph [41, 46]. However, this leads to a loss of information compared to the original hypergraph as the groups of vertices in hyperedges are no longer distinguished. Furthermore, the space required to store the resulting graph can be significantly higher than that of the original hypergraph [41, 46]. Another approach is to use a bipartite graph representation with vertices in one partition and hyperedges in the other, where each hyperedge connects to all vertices belonging to it [41] (an example is shown in Figure 1b). This is the approach that we adopt in this paper.

In the bipartite representation, there is an edge (u, ngh) if $u \in V$ and $ngh \in N^+(u)$, or $u \in E$ and $ngh \in N^+(u)$. The edges for each element are stored in an adjacency array. We also store the incoming edges for each vertex and hyperedge to enable the direction optimization that we discuss in Section 4.3. For weighted hypergraphs, we store the weights interleaved with the edges in the bipartite representation for cache locality. The hypergraph can be transposed using the Transpose function, which swaps the roles of the incoming and outgoing edges for all elements.

One implementation choice that we considered was to directly pass bipartite graphs to the Ligra framework. However, this would either require hyperedges to have distinct identifiers from vertices, making it unnatural to index arrays in the application code, or require mapping the hyperedge identifiers to the range $[0, \ldots, n_e - 1]$ on every array access, leading to additional overhead on hyperedge accesses and

more complicated application code. Instead, we modified the Ligra code to distinguish between vertices and hyperedges and represent them using identifiers in the ranges $[0, \ldots, n_v - 1]$ and $[0, \ldots, n_e - 1]$, respectively. We borrow existing data structures and functions from Ligra, which we describe here for completeness.

VertexSets (HyperedgeSets) have two underlying implementations: a sparse integer array storing the IDs of the elements in the set, and a dense boolean array of length $|V|$ ($|E|$) storing 1's in the locations corresponding to the IDs of the elements in the set, and 0's everywhere else.

Implementing VertexMap and HyperedgeMap simply requires mapping the function over the input VertexSet or HyperedgeSet, and applying a parallel filter on the result. Assuming that the function takes $O(1)$ work (which is true in all of our applications), the overall work is $O(|U|)$ and depth is $O(\log |U|)$ for an input set U.

VertexProp and HyperedgeProp map the C function over the outgoing edges of the input set and for the edges that return *true*, applies the F function in parallel. A parallel scan is applied over the degrees of elements in the input to determine offsets into an array storing the neighbors. A parallel filter is applied over the neighbors of F to obtain the output set. For an input set U, and functions F and C that take $O(1)$ work (which is true in all of our applications), this takes $O(|U| + \sum_{u \in U} deg^+(u))$ work and $O(\log |H|)$ depth. We can remove duplicates from the output in the same bounds.

HyperedgeFilterNgh can be implemented by inspecting all neighbors of each hyperedge in the input HyperedgeSet in parallel and using a parallel filter to remove the vertices not satisfying C. This takes the same work and depth as HyperedgeProp. HyperedgePropCount requires the same work and depth bounds as HyperedgeProp as the counts can be implemented using fetch-and-adds or a semisort [36].

We refer the reader to [24] for implementation details of the bucketing structure. For the complexity of bucketing, we will use the following lemma from [24]:

Lemma 1 ([24]). *For n identifiers, T total buckets, K calls to* UpdateBuckets*, each of which updates a set S_i of identifiers, and L calls to* NextBucket*, bucketing takes $O(n + T + \sum_{i=0}^{K} |S_i|)$ expected work and $O((K + L) \log n)$ depth with high probability.*

4.3 Optimizations

VertexProp and HyperedgeProp uses the direction optimization [6, 81] to switch between a sparse traversal (described in Section 4.2) and a dense traversal based on the size of the input VertexSet or HyperedgeSet and the sum of its out-degrees. For VertexProp, the dense traversal loops over all hyperedges e in parallel, checking if they satisfy the C function, and if so applying F on its incoming edges serially, stopping once $C(e)$ returns *false*. We use the dense traversal when the input set and sum of its out-degrees is a constant fraction (1/20 in our experiments) of the sum of in-degrees of hyperedges (which preserves work-efficiency), and the sparse traversal otherwise. The sum of out-degrees is computed using a parallel scan. We have an analogous implementation for HyperedgeProp. The sparse traversals use the sparse set representation, and the dense traversals uses the dense set representation. The input set is converted between the representations based on the traversal type.

For the dense traversals, instead of simply mapping over the vertices with a parallel-for loop, we added an edge-aware parallelization scheme that creates tasks containing a roughly equal number of edges that are managed by the work-stealing scheduler [98]. We found this optimization to significantly improve load balancing for hypergraphs with highly-skewed degree distributions.

As in Ligra, we also provide a push-based dense traversal that densely represents the input set but loops over their outgoing edges, instead of over incoming edges of all vertices.

For VertexProp and HyperedgeProp, we use optimized versions that do not remove duplicates that can be used if the program guarantees that no duplicates will be generated in the output. When the output of VertexMap, HyperedgeMap, VertexProp, and HyperedgeProp is not needed, we use optimized implementations that do not call filter.

To reduce memory usage, Hygra supports compression of the underlying bipartite graph using the compression code from Ligra [83]. The neighbors of vertices and hyperedges are compressed using variable-length codes, and decoded on-the-fly when accessed in VertexProp and HyperedgeProp.

5 Parallel Hypergraph Algorithms

We have designed a collection of parallel hypergraph algorithms using Hygra: betweenness centrality (BC), maximal independent set (MIS), k-core decomposition, hypertrees, hyperpaths, connected components (CC), PageRank, and single-source shortest paths (SSSP). Our algorithms for betweenness centrality and k-core decomposition are new, while the connected components, PageRank, and single-source shortest paths algorithms are more efficient variants of previously described hypergraph algorithms [41, 46] and are similar to the corresponding graph algorithms in Ligra. The hypertrees and hyperpaths algorithms are similar to parallel breadth-first search on graphs. The maximal independent set algorithm is the first practical implementation for finding maximal independent sets in hypergraphs. We provide pseudocode for several of the algorithms and the pseudocode uses partially evaluated functions, i.e., invoking a function with fewer than all of its arguments gives a function that takes the remaining arguments as input. The reader may skip any of the algorithms in this section without loss of continuity.

5.1 Betweenness Centrality

The *betweenness centrality (BC)* [29] of a vertex v measures the fraction of shortest paths between all pairs of vertices that pass through v. In this paper, we consider BC on unweighted hypergraphs, although the definition extends to weighted hypergraphs. More formally, let $\sigma_{s,t}$ be the number of shortest paths between vertices s and t, $\sigma_{s,t}(v)$ be the number of shortest paths between s and t that pass through v, and $\delta_{s,t}(v) = \sigma_{s,t}(v)/\sigma_{s,t}$. The betweenness centrality of vertex v is defined to be $\sum_{s \neq v \neq t \in V} \delta_{s,t}(v)$. Brandes [13] presents a sequential algorithm for computing BC on graphs that takes $O(|V||E|)$ work, where each vertex v does a forward traversal to compute the number of shortest paths from v to every other vertex, and a backward traversal to compute the betweenness centrality contributions for all vertices from shortest paths starting at v. Each traversal takes $O(|E|)$ work. Brandes defines the dependency of a vertex s on v as $\delta_{s\bullet}(v) = \sum_{t \in V} \delta_{s,t}(v)$, and the traversals from s compute $\delta_{s\bullet}$ values for all other vertices. The betweenness centrality of a vertex v will then be $\sum_{s \in V} \delta_{s\bullet}(v)$. This algorithm has been parallelized in the literature (see, e.g., [69, 81, 88, 90]).

Puzis et al. [74] present a sequential algorithm for computing betweenness centrality in hypergraphs based on Brandes' algorithm. In the forward phase, a breadth-first search-like procedure is run, generating a predecessor set for each vertex and hyperedge containing all elements in the previous level of the search. Let $P_V(v)$ be the predecessor hyperedges of vertex v and $P_E(e)$ be the predecessor vertices of hyperedge e for the search from source s. $\sigma_{s,v}$ will be computed as $\sum_{u \in P_E(e) \,:\, e \in P_V(v)} \sigma_{s,u}$. This phase takes $O(n_v + \sum_{e \in E} (deg^+(e) \cdot deg^-(e)))$ work as each hyperedge is expanded once per incoming vertex. Note that this work complexity can be super-linear in the size of the hypergraph. The backward phase computes the dependency scores by iteratively propagating values from vertices to their predecessor hyperedges, and from hyperedges to their predecessor vertices starting from the furthest elements from the source. The update equation for a hyperedge e is shown in Equation 1 and for a vertex v is shown in Equation 2.

$$\hat{\delta}_s(e) = \sum_{v \,:\, e \in P_V(v)} \frac{\delta_{s\bullet}(v)}{\sigma_{s,v}} \tag{1}$$

$$\delta_{s\bullet}(v) = 1 + \sum_{e \,:\, v \in P_E(e)} (\sigma_{s,v} \cdot \hat{\delta}_s(e)) \tag{2}$$

By separating the vertex and hyperedge updates, each hyperedge and vertex only needs to be processed once, and the total work of the backward phase is $O(|H|)$. Puzis et al. [74] also propose a heuristic for merging vertices belonging to only a single hyperedge together, but the theoretical complexity remains the same.

In this section, we present a new parallel BC algorithm on hypergraphs that takes linear work per source vertex. We represent vertices and hyperedges at equal distance from the source as frontiers using VertexSets and HyperedgeSets, and process each frontier in parallel. We split the updates in the forward phase into separate update steps for vertices and hyperedges, so that each hyperedge only needs to be expanded once, giving linear work. The backward phase processes the frontiers in decreasing distance from the source, using fetch-and-adds to update the $\hat{\delta}_s$ and $\delta_{s\bullet}$ values. Computing exact BC scores would require running the algorithm from all sources, although in practice a subset of sources are used to compute approximate BC scores [4, 32]. As far as we know, this is the first parallel BC algorithm on hypergraphs. Our algorithm also uses direction optimization, in contrast to the original sequential algorithm of Puzis et al. [74].

The pseudocode for our BC algorithm from a single source is shown in Algorithm 1. We initialize auxiliary arrays as well as the *DependenciesV* array storing the final dependency scores on Lines 1–6. The forward phase of the algorithm is shown on Lines 22–37. We first set the number of paths for the source vertex to 1, mark it as visited, and place it on the initial frontier, represented as a VertexSet (Lines 22–23). While there are still reachable hyperedges and vertices, we repeatedly propagate the number of paths from vertices to hyperedges via VertexProp on Line 27 and from hyperedges to vertices via HyperedgeProp on Line 31. The function PathUpdate (Lines 8–9) passed to VertexProp and HyperedgeProp increments the number of paths of a successor element using a fetch-and-add. In contrast to [74], we first gather the number of paths at a hyperedge from all of its predecessor vertices before passing it to its successor vertices. In this way, a hyperedge only needs to visit each of its successor vertices once, passing the sum of all contributions from predecessor vertices. Duplicates in the output do not need to be removed as PathUpdate returns *true* only for the first update on the target. The Check function (Lines 10–11) passed to VertexProp and HyperedgeProp guarantees that only unexplored vertices and hyperedges are visited. We mark visited hyperedges and vertices on Lines 29 and 33, respectively, to ensure that each hyperedge and vertex is visited at most once. Each frontier that is explored is placed in the *Levels* array, so that we can explore them in a backward fashion in the second phase of the algorithm.

The backward phase of the algorithm is shown on Lines 35–44. We reuse the arrays *VisitedV* and *VisitedE* (Line 35). We transpose the hypergraph (Line 36) and explore the frontiers from the first phase in a backward fashion. Line 40 uses a VertexMap with the VisitVertexBack function (Lines 14–16) to mark vertices on the frontier as visited and add 1 to their dependency score, as required in Equation 2. Line 41 uses a VertexProp with the VtoE function (Lines 17–18) on predecessors (obtained by considering unexplored vertices via the Check function), which implements Equation 1. Line 43 marks hyperedges on the frontier as visited with

Algorithm 1 Pseudocode for BC in Hygra

```
1:  NumPathsV = {0, ..., 0}
2:  NumPathsE = {0, ..., 0}
3:  VisitedV = {0, ..., 0}
4:  VisitedE = {0, ..., 0}
5:  DependenciesV = {0, ..., 0}
6:  DependenciesE = {0, ..., 0}
7:  Levels = []
8:  procedure PathUpdate(NumPathsSrc, NumPathsDst, s, d)
9:      return (FAA(&NumPathsDst[d], NumPathsSrc[s]) == 0)
10: procedure Check(Visited, i)
11:     return (Visited[i] == 0)
12: procedure Visit(Visited, i)
13:     Visited[i] = 1
14: procedure VisitVertexBack(v)
15:     Visited[v] = 1
16:     DependenciesV[v]+=1
17: procedure VtoE(v, e)
18:     FAA(&DependenciesE[e], DependenciesV[v]/NumPathsV[v])
19: procedure EtoV(e, v)
20:     FAA(&DependenciesV[v], DependenciesE[e] × NumPathsV[v])
21: procedure BC(H, src)                          ▷ src is the source vertex
22:     NumPathsV[src] = 1, VisitedV[src] = 1
23:     VertexSet FrontierV = {src}
24:     HyperedgeSet FrontierE = {}
25:     currLevel = 0
26:     while (true) do
27:         FrontierE = VertexProp(H, FrontierV,
              PathUpdate(NumPathsV, NumPathsE), Check(VisitedE))
28:         if |FrontierE| == 0 then break
29:         HyperedgeMap(FrontierE, Visit(VisitedE))
30:         Levels[currLevel++] = FrontierE
31:         FrontierV = HyperedgeProp(H, FrontierE,
              PathUpdate(NumPathsE, NumPathsV), Check(VisitedV))
32:         if |FrontierV| == 0 then break
33:         VertexMap(FrontierV, Visit(VisitedV))
34:         Levels[currLevel++] = FrontierV
35:     VisitedV = {0, ..., 0}, VisitedE = {0, ..., 0}
36:     Transpose(H)
37:     currLevel = currLevel − 1
38:     while currLevel ≥ 0 do
39:         FrontierV = Levels[currLevel--]
40:         VertexMap(FrontierV, VisitVertexBack)
41:         VertexProp(H, FrontierV, VtoE, Check(VisitedE))
42:         FrontierE = Levels[currLevel--]
43:         HyperedgeMap(FrontierE, Visit(VisitedE))
44:         HyperedgeProp(H, FrontierE, EtoV, Check(VisitedV))
45:     return DependenciesV
```

a HyperedgeMap. Finally, Line 44 implements the sum in Equation 2 with the EtoV function (Lines 19–20) on predecessors.

Analysis. We analyze the complexity for a single source vertex. In the forward phase of BC, each vertex and hyperedge will appear in at most one frontier because once a vertex or hyperedge has been visited, its *VisitedV* or *VisitedE* entry will be marked, and it will fail the check by the Check function in subsequent iterations. Therefore the sum of the sizes of all frontiers plus their out-degrees will be $O(|H|)$. As VertexProp and HyperedgeProp do work proportional to the size of the input set plus the sum of its out-degrees, the overall work performed by the algorithm is $O(|H|)$, which is work-efficient. Each call to VertexProp and HyperedgeProp takes $O(\log |H|)$ depth, and so the overall depth is $O(D \log |H|)$ where D is the diameter of the hypergraph. The backward phase processes each frontier exactly once, giving the same work and depth bounds. Thus, the overall work is $O(|H|)$ and depth is $O(D \log |H|)$.

5.2 Maximal Independent Set

Given an undirected, unweighted hypergraph, an *independent set* is a subset of vertices $U \subseteq V$ such that no hyperedge has all of its incident vertices in U. In a graph, this definition is equivalent to the condition that no two vertices in an independent set are neighbors, although this does not hold for hypergraphs (a hyperedge may have multiple incident vertices included in an independent set as long as not all of its incident vertices are included). A *maximal independent set (MIS)* is an independent set that is not contained in a larger independent set. Finding maximal independent sets in parallel has been widely studied for graphs, and there exists linear-work parallel algorithms for the problem [10, 62]. However, the problem is much harder to solve on hypergraphs, and the total work of known parallel algorithms is super-linear [5, 9, 48, 50]. These algorithms have only been described in theory, and as far as we know, there have been no implementations of parallel MIS algorithms on hypergraphs.

This paper implements a variant of the Beame-Luby MIS algorithm [5], which is a core component of a more recent algorithm by Bercea et al. [9]. The algorithm is iterative and performs the following steps in each iteration:

(**1**) Generate a sample of vertices I, each sampled with probability $p = 1/(2^{d+1}\Delta)$, where $d = \max_{e \in E} deg(e)$ and Δ is the normalized degree as defined in [5, 9].
(**2**) For any hyperedge e that has all of its vertices in I, remove all vertices in e from I.
(**3**) Add the remaining vertices in I to the MIS and delete them from V.
(**4**) Remove the vertices in I from all remaining hyperedges.
(**5**) Remove hyperedges whose vertices is a subset of another hyperedge's vertices.
(**6**) Remove hyperedges that contain only one vertex, and remove those vertices from V.

Our implementation picks vertices with a constant probability $p = 1/3$ as we found that it performs better in practice, and does not perform Step (5), which is not needed for correctness. The pseudocode is shown in Algorithm 2.

Our implementation uses a *Flags* array to represent the status of vertices, with a value of $Flags[v] = 0$ meaning that v is undecided, $Flags[v] = 1$ indicating that v is not

Algorithm 2 Pseudocode for MIS in Hygra

```
1: Flags = {0, ..., 0}
2: Counts = {0, ..., 0}
3: procedure Sample(round, v)
4:     With probability p, set Flags[v] = round
5: procedure Count(e, v)
6:     FAA(&Counts[e], 1)
7: procedure ResetNgh(e, v)
8:     Flags[v] = 0
9: procedure FilterV(v)
10:    return (Flags[v] == 0)
11: procedure FilterE(e)
12:    if (deg(e) == 1 and Flags[ngh_0(e)] == 0) then
13:        Flags[ngh_0(e)] = 1
14:    return (deg(e) > 1)
15: procedure Ind(e)
16:    return (Counts[e] == deg(e))
17: procedure Reset(e)
18:    Counts[e] = 0
19: procedure CheckF(round , v)
20:    return (Flags[v] == round)
21: procedure MIS(H)
22:    VertexSet FrontierV = {0, ..., n_v − 1}          ▷ all vertices
23:    HyperedgeSet FrontierE = {0, ..., n_e − 1}       ▷ all hyperedges
24:    round = 1
25:    while (|FrontierV| > 0) do
26:        round++
27:        VertexMap(FrontierV, Sample(round))
28:        HyperedgeMap(FrontierE, Reset)
29:        HyperedgeProp(H, FrontierE, Count, CheckF(round))
30:        HyperedgeSet FullEdges = HyperedgeMap(FrontierE, Ind)
31:        HyperedgeProp(H, FullEdges, ResetNgh, CheckF(round))
32:        HyperedgeFilterNgh(H, FrontierE, FilterV)
33:        FrontierE = HyperedgeMap(FrontierE, FilterE)
34:        FrontierV = VertexMap(FrontierV, FilterV)
35:    return Flags
```

in the MIS, and any other value indicating that v is in the MIS. *Flags* is initialized to all 0's on Line 1. We also initialize an auxiliary array *Counts*, which will be used to count the number of incident vertices of hyperedges selected in the random sample (Line 2). We create initial frontiers containing all vertices and hyperedges (*FrontierV* and *FrontierE* on Lines 22–23). We also keep track of the round number (Lines 24 and 26). Line 27 uses a VertexMap with the function Sample (Lines 3–4) to sample vertices by marking their *Flags* value with the round number with probability p. We reset the *Counts* values for hyperedges on the frontier on Line 28. On Line 29, we count for each hyperedge the number of its vertices that were selected in the sample for this round using HyperedgeProp with the Count (Lines 5–6) and CheckF (Lines 19–20) functions. We then check which hyperedges had all of their vertices selected in the sample on Line 30 with a HyperedgeMap with the Ind function that checks if the count is equal to the hyperedge's cardinality (Lines 15–16). The HyperedgeSet *FullEdges* contains the hyperedges where this is true, and we unmark the *Flags* values of their vertices on Line 31 using HyperedgeProp with the ResetNgh function (Lines 7–8). On Line 32, we remove vertices that have been selected in the MIS from the hyperedges using HyperedgeFilterNgh with the FilterV function (Lines 9–10), so that we do not need to process them in future rounds. Line 33 updates the hyperedge frontier by filtering out hyperedges with cardinality 0 and 1 using the FilterE function (Lines 11–14). For hyperedges with cardinality 1 we mark their only vertex (ngh_0) as not being in the MIS. Line 34 updates the vertex frontier with the FilterV function by filtering out vertices whose status has already been decided. The algorithm terminates when the status of all vertices have been decided, at which point *FrontierV* will be empty.

5.3 k-core Decomposition

For an undirected, unweighted hypergraph, a *k-core* is a maximal connected sub-hypergraph where every vertex has induced degree at least k. The *coreness* problem is to compute for each vertex the largest value of k for which it is part of the k-core. A simple parallel algorithm for coreness iteratively removes all vertices with degree at most k along with their incident hyperedges starting with $k = 0$, assigning removed vertices a coreness value of k, and incrementing k when all remaining vertices have induced degree greater than k [45]. Since each iteration requires scanning over all remaining vertices, this algorithm requires a total of $O(|H| + \rho|V|)$ work, where ρ is the number of iterations required by the algorithm, also known as the peeling complexity [24].

This section presents a new linear-work algorithm for computing coreness on hypergraphs based on the linear-work algorithm for graphs by Dhulipala et al. [24]. We have also implemented the $O(|H| + \rho|V|)$ work coreness algorithm in Hygra, and compare the performance of the $O(\rho|H|)$ work algorithm and the work-efficient algorithm in Section 6. To obtain work-efficiency, our algorithm uses the bucketing data structure described in Section 4. The pseudocode of our algorithm is shown in Algorithm 3.

An array D is initialized with the degrees of the vertices (Line 1). This array will keep track of the induced degrees of the vertices, and also store the final coreness value of the vertices. An array *Flags* (Line 2) is used to keep track of whether a hyperedge has been deleted (0 means not deleted and 1 means deleted). Line 13 initializes the bucketing structure, specifying that they should be processed in increasing order. Line 15 gets the next non-empty bucket in increasing order, and returns k, which corresponds to the current k-core being processed, as well as vertices with degree at most k in a VertexSet *FrontierV*. Line 17 marks the neighboring vertices of *FrontierV* as deleted using VertexProp with the functions RemoveHyperedge (Lines 3–4) and CheckRemoved (Lines 5–6). Hyperedges that are deleted will be returned in the HyperedgeSet *FrontierE*, and duplicates do not need to

Algorithm 3 Pseudocode for Coreness in Hygra

```
1:  D = {deg(v_0), ..., deg(v_{n_v-1})}          ▷ initialized to vertex degrees
2:  Flags = {0, ..., 0}                           ▷ initialized to all 0
3:  procedure RemoveHyperedge(v, e)
4:      return CAS(&Flags[e], 0, 1)
5:  procedure CheckRemoved(e)
6:      return (Flags[e] == 0)
7:  procedure UpdateD(k, v, numNghs)
8:      if D[v] > k then
9:          D[v] = max(D[v] − numNghs, k)
10:         return (v, D[v])
11:     else return (⊥, ⊥)
12: procedure Coreness(H)
13:     B = MakeBuckets(n_v, D, Increasing), finished = 0
14:     while (finished < n_v) do
15:         (k, VertexSet FrontierV) = NextBucket(B)
16:         finished+=|FrontierV|
17:         HyperedgeSet FrontierE = VertexProp(H, FrontierV,
              RemoveHyperedge, CheckRemoved)
18:         Moved = HyperedgePropCount(H, FrontierE, UpdateD(k))
19:         UpdateBuckets(B, Moved)
20:     return D
```

be removed, since the CAS on Line 4 will return *true* exactly once per hyperedge. On Line 18, we update the induced degrees of the vertices due to the removal of hyperedges using a HyperedgePropCount. The UpdateD function (Lines 7–11) will decrement the induced degree of each vertex v by its number of neighbors in *FrontierE* (*numNghs*), and set it to k if it falls below k, since this means v will have a coreness value of k. UpdateD returns a pair indicating the target bucket of the vertex v, which is its new induced degree $D[v]$ (Line 10). For vertices whose coreness value have already been determined, the null pair $(\perp, \perp)$ is returned (Line 11). The non-null pairs are stored in the *Moved* array output by HyperedgePropCount. Line 19 moves the vertices to new buckets using UpdateBuckets with the *Moved* array as input. The algorithm terminates when all vertices have been extracted from the bucket structure and processed.

Analysis. Each hyperedge will place each of its incident vertices in the *Moved* array only when it is deleted. Therefore the total size of the sets passed to UpdateBuckets is $O(\sum_{e \in E} deg(e))$. The number of identifiers in the bucket structure is n_v and the number of buckets is at most the maximum vertex degree, which is $O(n_e)$. The total number of calls to UpdateBuckets and NextBucket is the peeling complexity ρ. Using Lemma 1, we obtain an expected work of $O(|H|)$ and depth of $O(\rho \log |H|)$ with high probability.

5.4 Hypertrees

Given an unweighted hypergraph and a source vertex *src*, a *hypertree* contains all vertices and hyperedges reachable from *src* [30]. An algorithm that computes a hypertree outputs predecessor arrays for vertices and hyperedges, which specify one of its predecessors in a shortest path from *src*. The predecessor of a hyperedge is a vertex, and vice versa. The sequential algorithm for generating hypertrees is similar to a breadth-first search, and takes linear work in the size of the hypergraph [30]. Vertices are visited in order of their distance from the source, and each hyperedge is processed only the first time that a vertex visits it.

A parallel algorithm can be obtained by processing all vertices or hyperedges at the same distance from the source in parallel. This algorithm can be naturally implemented in Hygra using the VertexProp and HyperedgeProp functions. The frontier of vertices or hyperedges at the same distance from *src* are maintained using VertexSets and HyperedgeSets. The algorithm is similar to Ligra's parallel breadth-first search implementation.

Each vertex and hyperedge will appear in at most one frontier and therefore the sum of the sizes of all frontiers plus their out-degrees is $O(|H|)$. As VertexProp and HyperedgeProp do work proportional to the size of the input set plus the sum of its out-degrees, the overall work performed by the algorithm is $O(|H|)$, which is work-efficient. Each call to VertexProp and HyperedgeProp takes $O(\log |H|)$ depth, and so the overall depth is $O(D \log |H|)$ where D is the diameter of the hypergraph.

5.5 Hyperpaths

Given an unweighted hypergraph and a source vertex *src*, a *hyperpath tree* is a maximal hypergraph containing all vertices reachable from *src* via cycle-free paths (i.e., no vertex appears in more than one hyperedge along any particular path) [30]. The sequential algorithm for computing hyperpath trees [30] visits a hyperedge only when all incoming vertices of the hyperedge have visited it (instead of the first time an incoming vertex visits it). The algorithm takes linear work in the size of the hypergraph.

We implement a parallel algorithm for computing a hyperpath tree in Hygra, which requires minor changes to our hypertree algorithm so that a hyperedge is added to a frontier only when all of its incoming vertices have visited it. The overall work of the algorithm is $O(|H|)$ and depth is $O(L \log |H|)$, where L is the length of the longest simple path in the resulting hyperpath tree.

5.6 Connected Components

Given an undirected, unweighted hypergraph, a *connected component* is a maximal set of vertices that can all reach one another via incident hyperedges. The label propagation technique can be used to compute the connected components of a hypergraph [41, 46]. The idea is to initialize vertices with unique IDs and iteratively propagate IDs of vertices to their neighbors, having each vertex store the minimum ID among the IDs that it receives and its own. At convergence, the IDs on the vertices partition them into connected components.

We implement the label propagation algorithm in Hygra, but we note that there are more efficient parallel algorithms

for connected components for graphs (e.g., [80, 82]) that could be applied to hypergraphs. Our implementation iteratively propagates vertex IDs to hyperedges and hyperedge IDs to vertices using VertexProp and HyperedgeProp, respectively, with the writeMin function until the frontier becomes empty. The output frontier of VertexProp and HyperedgeProp contain only the elements whose IDs have changed. The vertex IDs are initialized to be unique integers, and the hyperedge IDs are initialized to ∞. Each iteration of the algorithm takes $O(|H|)$ work and $O(\log |H|)$ depth as the calls to VertexProp and HyperedgeProp could potentially process all vertices and hyperedges. For a hypergraph with diameter D, the overall work is $O(D|H|)$ and overall depth is $O(D \log |H|)$.

5.7 PageRank

PageRank is an algorithm for computing the importance of vertices in a graph [16], and can be extended to hypergraphs [8, 41, 46]. We consider PageRank on unweighted, connected hypergraphs. The following update equations defines the algorithm for a damping factor $0 \leq \alpha \leq 1$:

$$PR[v] = \frac{1-\alpha}{n_v} + \alpha \sum_{e \in N^-(v)} \frac{PR[e]}{deg^+(e)} \tag{3}$$

$$PR[e] = \sum_{v \in N^-(e)} \frac{PR[v]}{deg^+(v)} \tag{4}$$

Vertices spread their ranks equally to hyperedges for which they are incoming vertices for in Equation 4, and hyperedges spread their ranks equally to outgoing vertices in Equation 3. The update equations are applied iteratively until some convergence criterion is met (e.g., a maximum number of iterations is reached or the error falls below some threshold).

We implement PageRank in Hygra by iteratively calling VertexProp to pass PageRank values from vertices to hyperedges and HyperedgeProp to pass PageRank values from hyperedges to vertices. We also use VertexMap to normalize the PageRank scores as required in Equation 3, and use VertexMap and HyperedgeMap to reset arrays. We can also implement the PageRank-Entropy algorithm from MESH [41], which computes the entropy of the ranks of vertices in each hyperedge. This can be done with a VertexProp call that passes the entropy contribution of each vertex's rank to each hyperedge that it is an incoming vertex for.

Each iteration of PageRank (and PageRank-Entropy) processes all vertices and hyperedges using VertexProp and HyperedgeProp. Therefore, the per-iteration work is $O(|H|)$ and depth is $O(\log |H|)$.

5.8 Single-Source Shortest Paths

Given a weighted hypergraph and a source vertex *src*, the goal of *single-source shortest paths (SSSP)* is to compute the distance of the shortest path from *src* to every other reachable vertex in the hypergraph. We implement a parallel SSSP algorithm for hypergraphs in Hygra based on the Bellman-Ford algorithm for SSSP on graphs [20].

The algorithm initializes tentative shortest path distances (SP) of all vertices and hyperedges to ∞, except for the source vertex which has a distance of 0. Each iteration processes the active vertices, which are the vertices whose SP value changed in the previous iteration. Initially, only the source vertex is active. On each iteration, the algorithm calls VertexProp with a Relax function, which uses writeMin to update the SP values of all hyperedges with active incoming vertices to the minimum of their original SP value and the SP value of the incoming vertex plus the weight of the hyperedge. It then calls HyperedgeProp to update the SP values of outgoing vertices of hyperedges that were just updated using the same Relax procedure. If no SP values change in an iteration then the shortest path distances have been found, and the algorithm terminates. If the algorithm hasn't terminated after $n_v - 1$ iterations, then that means there is a negative weight cycle, and the algorithm reports this and terminates. The work of this algorithm is $O(n_v|H|)$ as each iteration can process all vertices and hyperedges, and the depth is $O(n_v \log |H|)$.

6 Experiments

Experimental Setup. We run all of our experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with four 2.4GHz 18-core E7-8867 v4 Xeon processors, each with a 45MB cache. The machine has a total of 1TB of RAM. Our programs use Cilk Plus [58] for parallelism and are compiled with the g++ compiler (version 5.5.0) with the `-O3` flag. By using Cilk's work-stealing scheduler we are able obtain an expected running time of $W/P + O(D)$ for an algorithm with W work and D depth on P processors [11]. Hygra also supports compilation with OpenMP.

For the parallel experiments, we use the command `numactl -i all` to balance the memory allocations across the sockets. All of the parallel speedup numbers that we report are based on the running time on 72-cores with hyper-threading compared to the running time on a single thread.

Data Sets. Our input hypergraphs are shown in Table 2. *com-Orkut* and *Friendster* are constructed using the community data from the Stanford Large Network Dataset Collection (SNAP) [59], where each community is a hyperedge containing its members as vertices. These are the largest real-world datasets used by prior work on hypergraph processing [41, 46]. We also include three larger real-world datasets, *orkut-groups*, *Web*, and *LiveJournal*, which are constructed from bipartite graphs from the Koblenz Network Collection (KONECT) [55]. To test on larger inputs, we also constructed synthetic random hypergraphs. *Rand1* and *Rand2* have 10^8 and 10^9 vertices/hyperedges, respectively, where the cardinality of each hyperedge is 10 and its member vertices are chosen uniformly at random. *Rand3* has 10^7

Hypergraph	$\|V\|$	$\|E\|$	$\sum_{e\in E} deg(e)$	$\max_{v\in V} deg(v)$	$\max_{e\in E} deg(e)$	Num. peeling rounds (ρ)	Num. clique-expanded edges
com-Orkut	2.32×10^6	1.53×10^7	1.07×10^8	2958	9120	1698	3.87×10^{10}
Friendster	7.94×10^6	1.62×10^6	2.35×10^7	1700	9299	351	5.53×10^9
Orkut-group	2.78×10^6	8.73×10^6	3.27×10^8	40425	3.18×10^5	2923	2.45×10^{12}
Web	2.77×10^7	1.28×10^7	1.41×10^8	1.1×10^6	1.16×10^7	3.18×10^5	1.06×10^{14}
LiveJournal	3.20×10^6	7.49×10^6	1.12×10^8	300	1.05×10^6	820	2.7×10^{12}
Rand1	10^8	10^8	10^9	34	10	30	4.45×10^9
Rand2	10^9	10^9	10^{10}	35	10	33	4.5×10^{10}
Rand3	10^7	10^7	10^9	153	100	109	4.95×10^{10}

Table 2. Hypergraph inputs.

vertices/hyperedges, and the cardinality of each hyperedge is 100 with its member vertices chosen uniformly at random. For input size scalability experiments, we also generated random hypergraphs with varying sizes and hyperedge cardinalities. For SSSP, we use weighted versions of the hypergraphs with random hyperedge weights from 1 to $\lfloor \log_2(\max(n_v, n_h)) \rfloor$. The inputs are all undirected.

Results. Table 3 shows the sequential and parallel running times of our algorithms, as well as their parallel speedup. The BC times are for a single source, and PageRank times are for 1 iteration. For k-core, we include times for both the work-efficient (WE) version and the work-inefficient (WI) version. We did not include hyperpaths in our experiments because the hyperpaths found in the inputs are too small to give meaningful running times. For the Orkut-group, Web, and LiveJournal inputs, we used the edge-aware parallelization scheme due to their highly skewed degree distributions.

Overall, the algorithms get good parallel speedup, ranging from 8.5–76.5x, and the parallel times on the real-world inputs are usually under 1 second. The random hypergraphs are larger than hypergraphs used in prior work, and we are able to achieve parallel running times on the order of seconds for Rand1 and Rand3 and tens of seconds for Rand2. The lower speedups for k-core on the real-world inputs are due to the large number of peeling rounds (see Table 2), many of which have few active vertices and hyperedges.

We see that our work-efficient k-core algorithm is usually much faster than the work-inefficient version, by a factor of up to 733x in parallel, as it does less work. The benefit is higher for the inputs with more peeling rounds (e.g., the Web hypergraph).

Figure 2 shows the running time vs. number of threads for all of the algorithms on Rand1. We see good parallel scalability for all of the algorithms, with speedups ranging from 31–53x on 72 cores with hyper-threading.

Figure 3 shows the running time vs. hyperedge count for all of the algorithms on random hypergraphs with 10^7 vertices and cardinality-10 hyperedges (we also tried fixing the vertex and hyperedge count and varying the cardinality, and found similar trends). We see a near-linear increase in running time on all of the algorithms except hypertree and k-core, which have a sub-linear increase. For hypertree, the

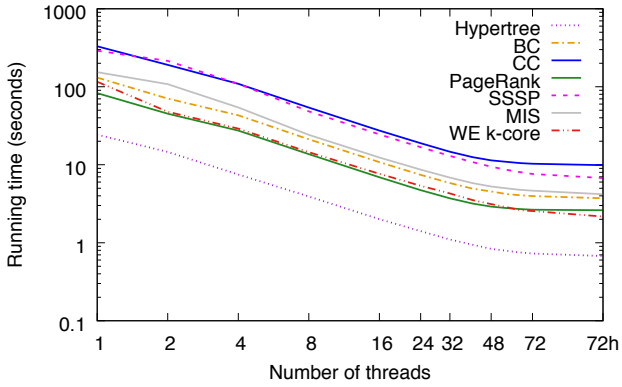


Figure 2. Running time vs. number of threads on Rand1. "72h" refers to 144 hyper-threads.

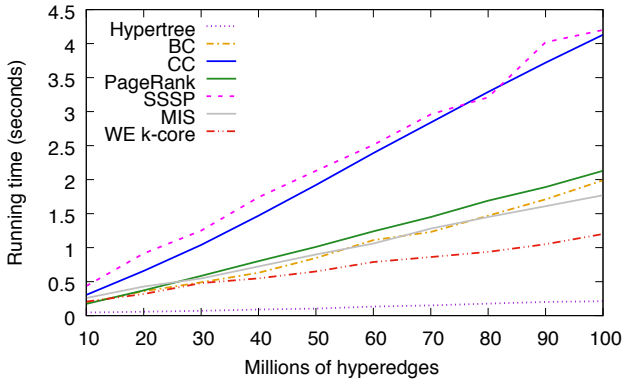


Figure 3. Running time vs. number of hyperedges on 72 cores with hyper-threading.

number of edges traversed increases sub-linearly due to the direction optimization that avoids many edge traversals. For k-core, the peeling complexity, and hence running time, does not increase linearly with the number of hyperedges.

Figures 4 and 5 show the impact of the direction optimization on com-Orkut and LiveJournal. We plot the running time using all sparse traversals, all dense traversals, and hybrid traversals with the default threshold of 1/20 fraction of the sum of in-degrees of the hyperedges for VertexProp and sum of in-degrees of vertices for HyperedgeProp. For

	com-Orkut			Friendster			Rand1			Rand2			Rand3			Orkut-group			Web			LiveJournal		
Algorithm	T_1	T_{72h}	SU	T_1	T_{72h}	SU	T_1	T_{72h}	SU	T_1	T_{72h}	SU	T_1	T_{72h}	SU	T_1	T_{72h}	SU	T_1	T_{72h}	SU	T_1	T_{72h}	SU
Hypertree	1.04	0.031	33.5	0.803	0.022	36.5	24.3	0.676	35.9	321	8.97	35.8	2.18	0.047	46.4	0.551	0.021	26.2	2.71	0.068	39.9	0.754	0.022	34.3
BC	5.31	0.12	44.3	2.7	0.07	38.6	131.0	3.72	35.2	1890	39.4	48.0	40.8	0.82	49.8	7.58	0.141	53.8	10.7	0.517	20.7	3.66	0.099	37.0
CC	7.87	0.162	48.6	3.34	0.082	40.5	330.0	9.88	33.4	4190	121	34.6	70.5	1.07	65.9	11.6	0.18	64.4	11.0	0.478	23.0	4.01	0.081	49.5
PageRank	3.31	0.083	39.9	0.941	0.026	36.2	84.1	2.61	32.2	955	28.6	33.4	57.3	1.6	35.8	6.88	0.119	57.8	5.27	0.27	19.5	2.66	0.062	42.9
SSSP	8.81	0.157	56.1	3.54	0.107	76.5	290.0	6.76	43.3	5730	79.8	71.8	54.0	1.0	54.0	14.7	0.261	56.3	7.04	0.245	28.7	5.56	0.118	47.1
MIS	7.73	0.227	34.1	3.26	0.11	29.6	154.0	4.19	36.8	1680	44.5	37.8	68.0	1.09	62.4	9.86	0.411	24.0	17.8	2.09	8.52	4.92	0.434	11.3
WE k-core	7.09	0.738	9.61	2.09	0.081	25.8	116.0	2.17	53.5	2210	31.8	69.5	41.0	0.866	47.3	13.7	0.831	16.5	12.5	0.965	13.0	6.13	0.325	18.9
WI k-core	33.9	1.88	18.0	9.72	0.421	23.1	133.0	3.4	39.1	1150	33.6	34.2	87.3	1.18	74.0	96.6	3.34	28.9	23500	707.0	33.2	16.9	0.905	18.7

Table 3. Sequential times (T_1) and 72-core with hyper-threading (T_{72h}) times (seconds), as well as the parallel speedup (SU).

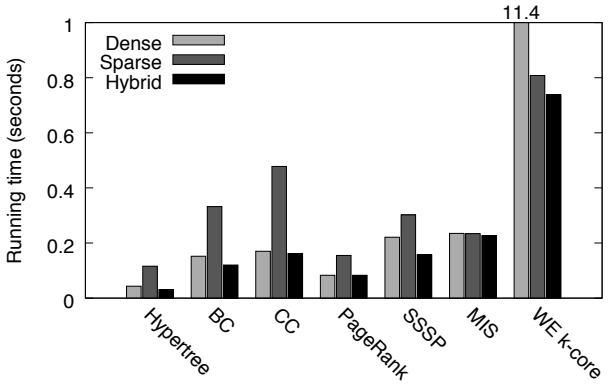


Figure 4. Running times of dense, sparse, and hybrid traversals on com-Orkut using 72 cores with hyper-threading.

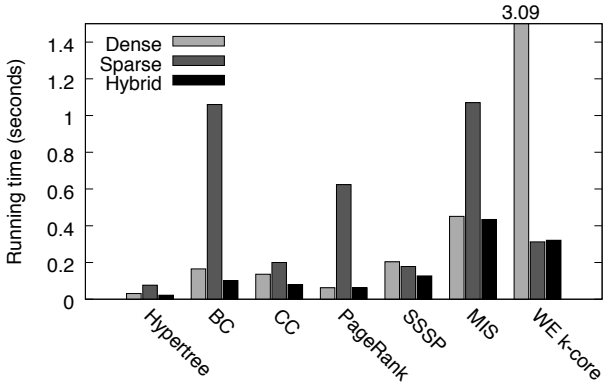


Figure 5. Running times of dense, sparse, and hybrid traversals on LiveJournal using 72 cores with hyper-threading.

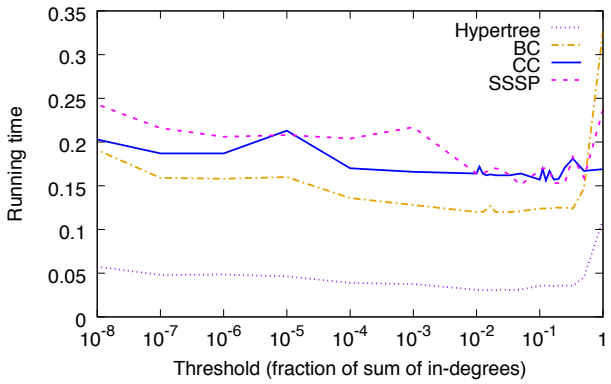


Figure 6. Running times as a function of threshold on com-Orkut using 72 cores with hyper-threading.

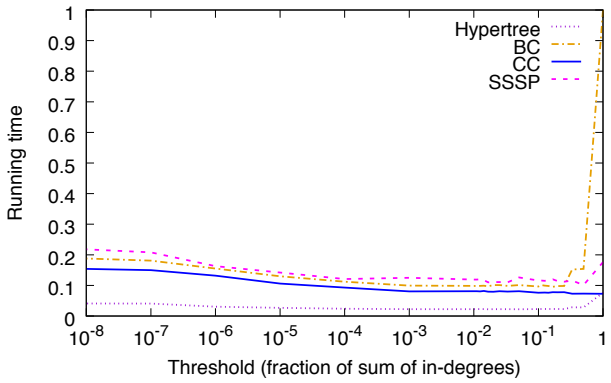


Figure 7. Running times as a function of threshold on LiveJournal using 72 cores with hyper-threading.

all of the algorithms, we see that the hybrid traversal is the the fastest or tied for the fastest among the three cases.

We found the default threshold to work reasonably well across all our applications and inputs. We show the running time as a function of threshold for several applications on com-Orkut and LiveJournal in Figures 6 and 7. We see that the performance is similar across a wide range of thresholds.

In Table 4, we report the memory, percentage of cycles stalled due to memory accesses, and LLC local miss rate for several algorithms on com-Orkut, Rand1, and LiveJournal. We see that the cache miss rate and memory bandwidth is the highest for the random hypergraph, Rand1, as the edges have very little locality. The memory bandwidth is close to the peak bandwidth of the machine, and the algorithms on Rand1 are memory bandwidth-bound. com-Orkut and LiveJournal exhibit locality in their structure, and thus have lower cache miss rates, and require fewer requests to DRAM, thereby lowering the memory bandwidth. However, a decent fraction of the cycles are still stalled waiting for memory accesses, making the algorithms memory latency-bound. All of the algorithms benefit from spatial locality when traversing the adjacency list in the bipartite representation.

	com-Orkut			Rand1			LiveJournal		
Algorithm	Fraction of Cycles Stalled	LLC Miss Rate	Memory Bandwidth	Fraction of Cycles Stalled	LLC Miss Rate	Memory Bandwidth	Fraction of Cycles Stalled	LLC Miss Rate	Memory Bandwidth
Hypertree	0.364	0.122	144.7	0.726	0.551	161.7	0.28	0.161	135.5
BC	0.462	0.111	131.1	0.804	0.808	161.8	0.35	0.097	120.1
CC	0.444	0.089	134.4	0.837	0.833	147.5	0.40	0.044	126.1
PageRank	0.79	0.24	123.1	0.927	0.933	146.6	0.69	0.163	104.0
SSSP	0.5	0.098	132.1	0.842	0.781	146.2	0.49	0.057	123.2
WE k-core	0.367	0.358	53.86	0.573	0.422	140.1	0.39	0.286	72.5

Table 4. Fraction of cycles stalled on memory requests, LLC local miss rate, and memory bandwidth (GB/s). All experiments use 72 cores with hyper-threading.

Comparison with Alternatives. While it is difficult to directly compare with HyperX and MESH as they are designed for distributed memory, we first perform a rough comparison in terms of the running times reported in their papers [41, 46]. MESH reports a running time of about 1 minute per iteration on com-Orkut using a cluster of eight 12-core machines, and HyperX reports a running time of about 10s using a cluster of eight 4-core machines (HyperX's algorithm is for random walks, which does less work than PageRank per iteration). In contrast, one iteration of Hygra's PageRank on com-Orkut takes 0.083s on 72 cores and 3.31s on one thread. Even adjusting for differences in processor specifications, we are significantly faster than their reported parallel numbers using just a single thread, and orders of magnitude faster in parallel. The large difference in performance of MESH and HyperX compared to Hygra is due to the higher communication costs of distributed memory and overheads of Spark.

We also ran MESH on our 72-core machine, and did a sweep of the parameter space (partition strategy and number of partitions), and the best running time we obtained for one iteration of PageRank on com-Orkut was over 2 minutes, which is much slower than Hygra's time. MESH reports competitive performance with HyperX [41], and so we expect the performance of HyperX to be in the same ballpark. For frontier-based algorithms our speedups would be even higher as HyperX and MESH require work proportional to the hypergraph size on every iteration whereas we only do work proportional to the frontier size plus the sum of its out-degrees. We ran the single-source shortest paths algorithm from MESH (which works on unit weights and so is similar to our hypertree algorithm) on our 72-core machine and observed that for com-Orkut just the first iteration takes over 1 minute. This is much slower than the Hygra time for running the algorithm to convergence.

As mentioned in Section 4.2, another method for representing a hypergraph is to create a clique among all vertices for each hyperedge, store the result as a graph (known as the clique-expanded graph), and apply graph algorithms on it. This approach would work for algorithms that do not treat hyperedges differently from vertices, such as hypertree, connected components, and single-source shortest paths (for algorithms that treat the hyperedges specially, this approach would generate incorrect results). We show the number edges in the clique-expanded graph for each of our inputs in Table 2. We see that the sizes are several orders of magnitude greater than the corresponding hypergraph using the bipartite graph representation. As a baseline, we ran Ligra's breadth-first search, connected components, and SSSP implementations on the clique-expanded graph for Friendster (which is 235x larger than the bipartite representation) on 72 cores. Breadth-first search took 0.061s, which is 2.8x slower than Hygra's hypertree implementation (see Table 3). Connected components took 2.35s, which is 28.7x slower than Hygra. SSSP took 3.27s, which is 30.6x slower than Hygra. The overhead is due to additional edge traversals in the clique-expanded graph. However, the running time overhead is not as high as the space overhead, since the clique-expanded graph is much denser and has better locality. The overhead is only 2.8x for breadth-first search since the dense traversal optimization allows many edges to be skipped.

7 Conclusion

We have presented a suite of parallel hypergraph algorithms with strong theoretical guarantees. We implemented the algorithms by extending the Ligra graph processing framework to handle hypergraphs. Our experiments show that the algorithms achieve good parallel scalability and significantly better performance than prior work. Future work includes extending graph optimizations for locality and scalability (e.g., [7, 53, 57, 85, 92, 96, 97, 101]) to hypergraphs.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, MIT Research Support Committee Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Sinan Aksoy, Dustin Arendt, Louis S Jenkins, Brenda Praggastis, Emilie Purvine, and Marcin Zalewski. 2019. High Performance Hypergraph Analytics of Domain Name System Relationships. In *HICSS Symposium on Cybersecurity Big Data Analytics*.

[2] Noga Alon, Uri Arad, and Yossi Azar. 1999. Independent Sets in Hypergraphs with Applications to Routing via Fixed Paths. In *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems: Randomization, Approximation, and Combinatorial Algorithms and Techniques (RANDOM-APPROX)*. 16–27.

[3] Giorgio Ausiello, Giuseppe F. Italiano, and Umberto Nanni. 1998. Hypergraph traversal revisited: Cost measures and dynamic algorithms. In *Mathematical Foundations of Computer Science*. 1–16.

[4] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating betweenness centrality. In *Workshop on Algorithms and Models for the Web-Graph (WAW)*. 124–137.

[5] Paul Beame and Michael Luby. 1990. Parallel Search for Maximal Independence Given Minimal Dependence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 212–218.

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

[7] S. Beamer, K. Asanovic, and D. Patterson. 2017. Reducing Pagerank Communication via Propagation Blocking. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 820–831.

[8] Abdelghani Bellaachia and Mohammed Al-Dhelaan. 2013. Random Walks in Hypergraph. In *International Conference on Applied Mathematics and Computational Methods*. 187–194.

[9] Ioana O. Bercea, Navin Goyal, David G. Harris, and Aravind Srinivasan. 2017. On Computing Maximal Independent Sets of Hypergraphs in Parallel. *ACM Trans. Parallel Comput.* 3, 1, Article 5 (Jan. 2017), 5:1–5:13 pages.

[10] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 308–317.

[11] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[12] Cecile Bothorel and Mohamed Bouklit. 2011. An Algorithm for Detecting Communities in Folksonomy Hypergraphs. In *International Conference on Innovative Internet Community Services*. 159–168.

[13] Ulrik Brandes. 2001. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology* 25 (2001), 163–177.

[14] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206.

[15] Alain Bretto, Hocine Cherifi, and Driss Aboutajdine. 2002. Hypergraph imaging: an overview. *Pattern Recognition* 35, 3 (2002), 651–658.

[16] S. Brin and L. Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Computer Networks and ISDN Systems*. 107–117.

[17] U. V. Catalyurek and C. Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (Jul 1999), 673–693.

[18] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdaag, Robert T. Heaphy, and Lee Ann Riesen. 2009. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel and Distrib. Comput.* 69, 8 (2009), 711–724.

[19] L. Chen, X. Huo, B. Ren, S. Jain, and G. Agrawal. 2015. Efficient and Simplified Parallel Graph Processing over CPU and MIC. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 819–828.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3. ed.)*. MIT Press.

[21] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 752–768.

[22] Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2019. Phoenix: A Substrate for Resilient Distributed Graph Analytics. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 615–630.

[23] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. 2006. Parallel Hypergraph Partitioning for Scientific Computing. In *International Conference on Parallel and Distributed Processing (IPDPS)*. 124–124.

[24] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 293–304.

[25] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 393–404.

[26] L. Ding and A. Yilmaz. 2008. Image Segmentation as Learning on Hypergraphs. In *International Conference on Machine Learning and Applications*. 247–252.

[27] Aurelien Ducournau and Alain Bretto. 2014. Random walks in directed hypergraphs and application to semi-supervised image segmentation. *Computer Vision and Image Understanding* 120 (2014), 91–102.

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

[29] Linton Freeman. 1977. A set of measures of centrality based upon betweenness. *Sociometry* 40 (1977), 35–41.

[30] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed hypergraphs and applications. *Discrete Applied Mathematics* 42, 2 (1993), 177 – 201.

[31] J. Gao, Q. Zhao, W. Ren, A. Swami, R. Ramanathan, and A. Bar-Noy. 2012. Dynamic shortest path algorithms for hypergraphs. In *International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt)*. 238–245.

[32] Robert Geisberger, Peter Sanders, and Dominik Schultes. 2008. Better Approximation of Betweenness Centrality. In *Algorithms Engineering and Experiments (ALENEX)*. 90–100.

[33] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. 2018. Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms. In *Euro-Par*. 249–264.

[34] Joseph Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 17–30.

[35] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making Pull-based Graph Processing Performant. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 246–260.

[36] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.

[37] Torben Hagerup. 1992. Fast and optimal simulations between CRCW PRAMs. In *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*. 45–56.

[38] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2012. The STAPL Parallel Graph Library. In *Languages and*

Compilers for Parallel Computing (LCPC). 46–60.

[39] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2014. KLA: a new algorithmic paradigm for parallel graph computations. In *International Conference on Parallel Architectures and Compilation (PACT)*. 27–38.

[40] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2015. An Algorithmic Approach to Communication Reduction in Parallel Graph Algorithms. In *International Conference on Parallel Architecture and Compilation (PACT)*. 201–212.

[41] Benjamin Heintz, Rankyung Hong, Shivangi Singh, Gaurav Khandelwal, Corey Tesdahl, and Abhishek Chandra. 2019. MESH: A Flexible Distributed Hypergraph Processing System. In *IEEE International Conference on Cloud Engineering (IC2E)*. 12–22.

[42] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan. 2017. MultiGraph: Efficient Graph Processing on GPUs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 27–40.

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

[44] Louis Jenkins, Tanveer Hossain Bhuiyan, Sarah Harun, Christopher Lightsey, David Mentgen, Sinan G. Aksoy, Timothy Stavcnger, Marcin Zalewski, Hugh R. Medal, and Cliff Joslyn. 2018. Chapel HyperGraph Library (CHGL). In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

[45] Jiayang Jiang, Michael Mitzenmacher, and Justin Thaler. 2017. Parallel Peeling Algorithms. *ACM Trans. Parallel Comput.* 3, 1, Article 7 (Jan. 2017), 7:1–7:27 pages.

[46] W. Jiang, J. Qi, J. X. Yu, J. Huang, and R. Zhang. 2019. HyperX: A Scalable Hypergraph Framework. *IEEE Transactions on Knowledge and Data Engineering* 31, 5, 909–922.

[47] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2011. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.* 27, 2 (2011), 303–325.

[48] Richard M. Karp, Eli Upfal, and Avi Wigderson. 1988. The Complexity of Parallel Search. *J. Comput. Syst. Sci.* 36, 2 (April 1988), 225–253.

[49] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. 1999. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7, 1 (March 1999), 69–79.

[50] Pierre Kelsen. 1992. On the Parallel Complexity of Computing a Maximal Independent Set in a Hypergraph. In *ACM Symposium on Theory of Computing (STOC)*. 339–350.

[51] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9.

[52] Gaurav Khanna, Nagavijayalakshmi Vydyanathan, T. Kurc, U. Catalyurek, P. Wyckoff, J. Saltz, and P. Sadayappan. 2005. A hypergraph partitioning based approach for scheduling of tasks with batch-shared I/O. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. 792–799.

[53] Vladimir Kiriansky, Yunming Zhang, and Saman P. Amarasinghe. 2016. Optimizing Indirect Memory References with milk. In *International Conference on Parallel Architectures and Compilation (PACT)*. 299–312.

[54] Sriram Krishnamoorthy, Umit Catalyurek, Jarek Nieplocha, Atanas Rountev, and P. Sadayappan. 2006. Hypergraph Partitioning for Automatic Memory Hierarchy Management. In *ACM/IEEE Conference on Supercomputing (SC)*.

[55] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *International Conference on World Wide Web (WWW)*. 1343–1350.

[56] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph computation on Just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 31–46.

[57] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2018. Accelerating PageRank using Partition-Centric Processing. In *USENIX Annual Technical Conference (ATC)*. 427–440.

[58] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *J. Supercomputing* 51, 3 (2010).

[59] Jure Leskovec and Andrej Krevl. 2019. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[60] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. 2018. ShenTu: Processing Multi-trillion Edge Graphs on Millions of Cores in Seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 56:1–56:11.

[61] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. 340–349.

[62] Michael Luby. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.* 15, 4 (November 1986), 1036–1055.

[63] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *IEEE International Conference on Data Engineering (ICDE)*. 363–374.

[64] Saeed Maleki, G. Carl Evans, and David A. Padua. 2015. Tiled Linear Algebra a System for Parallel Graph Algorithms. In *Languages and Compilers for Parallel Computing*. 116–130.

[65] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *ACM Conference on Management of Data (SIGMOD)*. 135–146.

[66] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.* 48, 2, Article 25 (Oct. 2015), 39 pages.

[67] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at What COST?. In *USENIX Conference on Hot Topics in Operating Systems (HotOS)*.

[68] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 201–213.

[69] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*. 456–471.

[70] Lars Relund Nielsen, Kim Allan Andersen, and Daniele Pretolani. 2005. Finding the K Shortest Hyperpaths. *Comput. Oper. Res.* 32, 6 (June 2005), 1477–1497.

[71] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 622–636.

[72] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1–19.

[73] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. 2018. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 9:1–9:14.

[74] Rami Puzis, Manish Purohit, and V. S. Subrahmanian. 2013. Betweenness computation in the single graph representation of hypergraphs.

Social Networks 35, 4 (2013), 561–572.

[75] S. Riazi and B. Norris. 2016. GraphFlow: Workflow-based big graph processing. In *IEEE International Conference on Big Data (Big Data)*. 3336–3343.

[76] A. Ritz, B. Avent, and T. M. Murali. 2017. Pathway Analysis with Signaling Hypergraphs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 14, 5 (Sept 2017), 1042–1055.

[77] Sanjukta Roy and Balaraman Ravindran. 2015. Measuring Network Centrality Using Hypergraphs. In *ACM IKDD Conference on Data Sciences*. 59–68.

[78] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[79] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* 50, 6, Article 81 (Jan. 2018), 81:1–81:35 pages.

[80] Yossi Shiloach and Uzi Vishkin. 1982. An $O(\log n)$ Parallel Connectivity Algorithm. *J. Algorithms* 3, 1 (1982), 57–67.

[81] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 135–146.

[82] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2014. A Simple and Practical Linear-Work Parallel Algorithm for Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 143–153.

[83] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *IEEE Data Compression Conference (DCC)*. 403–412.

[84] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. 2018. Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. *PVLDB* 12, 2 (2018), 154–168.

[85] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *International Conference on Supercomputing (ICS)*. Article 16, 16:1–16:10 pages.

[86] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2019. VEBO: a vertex- and edge-balanced ordering heuristic to load balance parallel graph processing. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 391–392.

[87] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225.

[88] Guangming Tan, Dengbiao Tu, and Ninghui Sun. 2009. A Parallel Algorithm for Computing Betweenness Centrality. In *International Conference on Parallel Processing (ICPP)*. 340–347.

[89] Aleksandar Trifunovic and William J. Knottenbelt. 2008. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel and Distrib. Comput.* 68, 5 (2008), 563–581.

[90] Dengbiao Tu and Guangming Tan. 2009. Characterizing Betweenness Centrality Algorithm on Multi-core Architectures. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. 182–189.

[91] Lei Wang, Liangji Zhuang, Junhang Chen, Huimin Cui, Fang Lv, Ying Liu, and Xiaobing Feng. 2018. Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 276–289.

[92] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *ACM International Conference on Management of Data (SIGMOD)*. 1813–1828.

[93] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Foundations and Trends in Databases* 7, 1-2 (2017), 1–195.

[94] Jie Yan, Guangming Tan, Zeyao Mo, and Ninghui Sun. 2016. Graphine: Programming Graph-Parallel Computation of Large Natural Graphs for Multicore Clusters. *IEEE Trans. Parallel Distrib. Syst.* 27, 6 (2016), 1647–1659.

[95] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65.

[96] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-structured Analytics. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 183–193.

[97] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Matei Zaharia, and Saman P. Amarasinghe. 2017. Making Caches Work for Graph Analytics. In *IEEE International Conference on Big Data (BigData)*. 293–302.

[98] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 121:1–121:30 pages.

[99] Peng Zhao, Chen Ding, Lei Liu, Jiping Yu, Wentao Han, and Xiao-Bing Feng. 2019. Cacheap: Portable and Collaborative I/O Optimization for Graph Processing. *Journal of Computer Science and Technology* 34, 3 (01 May 2019), 690–706.

[100] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. 2006. Learning with Hypergraphs: Clustering, Classification, and Embedding. In *International Conference on Neural Information Processing Systems (NIPS)*. 1601–1608.

[101] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 301–316.

A Artifact Description

A.1 Abstract

The artifact contains the code for the Hygra framework and implementations of the parallel hypergraph algorithms using Hygra. We provide instructions for obtaining or generating the datasets used in this paper as well as scripts for running the experiments in the paper.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Algorithms:** The artifact includes parallel hypergrpah algorithms for betweenness centrality, maximal independent set, k-core decomposition, hypertrees, connected components, PageRank, and single-source shortest paths.
- **Compilation:** A compiler with support for Cilk Plus is used to compile the code. The experiments in the paper used g++ version 5.5.0, which has support for Cilk Plus. (While Hygra can also be compiled with OpenMP, the

numbers reported in the paper are obtained using Cilk Plus.)

- **Datasets:** The datasets consist of real-world hypergraphs from the Stanford Large Network Dataset Collection (SNAP) [59] and the Koblenz Network Collection (KONECT) [55], as well as synthetic random hypergraphs that we generated.
- **Run-time environment:** A Linux operating system should be used and `numactl` should be installed. The experiments in the paper used Ubuntu 16.04. Python 2.7 is used for running the scripts.
- **Hardware:** An x86-based multicore machine should be used. The experiments in the paper used a Dell PowerEdge R930 with four 2.4GHz 18-core E7-8867 v4 Xeon processors and a total of 1TB of RAM.
- **Output:** Running times of the algorithms are output to the console.
- **Experiment Workflow:** Clone the repository and use the provided scripts to run the experiments.
- **Publicly available?** Yes.

A.2.2 How Delivered

The artifact is available on Github at https://github.com/jshun/ppopp20-ae.

A.2.3 Hardware Dependencies

An x86-based multicore machine should be used for the experiments. To run all experiments, 1TB of RAM is needed. However, 200GB of RAM is sufficient to run all of the experiments except for the ones on Rand2 and the clique-expanded graph for Friendster. The total storage required for all of the datasets is 1TB. Excluding the large datasets (Rand2 and the clique-expanded graph for Friendster), the total storage required is 313GB.

A.2.4 Software Dependencies

A Linux operating system with `numactl` should be used to run the experiments. The artifact uses Cilk Plus for parallelism, and so a compiler with support for Cilk Plus should be installed. Python 2.7 is used for running the scripts.

A.2.5 Datasets

The real-world hypergraphs were downloaded from the Stanford Large Network Dataset Collection (SNAP) [59] and the Koblenz Network Collection (KONECT) [55], and converted to Hygra format using the `communityToHyperAdj` and `KONECTtoHyperAdj` programs, respectively, provided in the `utils/` directory. The synthetic hypergraphs were generated using the `randHypergraph` program in the `utils/` directory.

For the weighted versions of the hypergraphs, weights were added using the `adjHypergraphAddWeights` program in the `utils/` directory.

A.3 Installation

After cloning the repository and installing the software dependencies, the provided scripts can be used to compile and execute the programs. The code for the hypergraph algorithms is in the `apps/hyper/` directory and can be compiled manually by navigating to that directory and typing "`export CILK=1; make -j`". The programs in the `utils/` directory can be compiled in the same way.

For inputs where the total number of neighbors of vertices and hyperedges exceeds $2^{32} - 1$, the `LONG` environment variable should be defined prior to compilation. For inputs where the total number of vertices and hyperedges exceeds $2^{32} - 1$, the `EDGELONG` environment variable should be defined prior to compilation.

A.4 Experiment Workflow

The `runall` script at the top-level directory will run all experiments without the large Rand2 input and the clique-expanded Friendster graph. The `runall-quick` script at the top-level directory will skip the scalability tests for all of the inputs except for a small dataset, and will also skip the experiment on varying thread counts on Rand1. These two scripts will download the necessary datasets for the experiments. Individual experiments may be run as described below.

To download all of the datasets, navigate to the `inputs/` directory and type "`./download_datasets`". This will take a few hours. The following command line arguments may be passed to the `download_datasets` script to download only a subset of the datasets: `LARGE` will download only the large datasets (Rand2 and the clique-expanded Friendster graph); `RAND1` will only download the Rand1 dataset for testing performance on varying thread counts; `SIZES` will only download the random hypergraphs of varying sizes for testing performance as a function of input size; and `DIRECTION` will only download the com-Orkut and LiveJournal datasets for testing the performance of sparse, dense, and hybrid traversals as well as the performance of using different thresholds in the direction optimization.

The `run_scalability` script provided in the `apps/hyper/` directory will run all of the hypergraph algorithms both on a single thread and on all available cores of the machine. By default, all datasets except Rand2 will be used. This script will take several days to complete. To include Rand2 in the experiments, type "`./run_scalability LARGE`". To run the experiments on only a small dataset, which will terminate quickly, type "`./run_scalability QUICK`".

The `run_varying_threads` script in the `apps/hyper/` directory will run all of the algorithms on a varying number of threads on the Rand1 dataset.

The `run_varying_hyperedges` script in the `apps/hyper/` directory will run all of the algorithms using all available cores on random hypergraphs with a varying number of hyperedges.

The `run_directions` script in the `apps/hyper/` directory will test the parallel performance of sparse, dense, and hybrid traversals for all of the algorithms on the com-Orkut and LiveJournal datasets.

The `run_thresholds` script in the `apps/hyper/` directory will test the parallel performance of all of the algorithms on the com-Orkut and LiveJournal datasets using different thresholds for the direction optimization.

The `run_clique` script in the `apps/` directory will test the parallel performance of breadth-first search, connected components, and SSSP in Ligra on the clique-expanded graph for Friendster.

A.5 Evaluation and Expected Result

The results of the scalability experiments correspond to the numbers reported in Table 3 and Figure 2. The results of the experiments on random hypergraphs of different sizes correspond to the numbers reported in Figure 3. The results of the experiments on different traversal modes correspond to the numbers reported in Figures 4 and 5. The results of the experiments on different thresholds for the direction optimization correspond to the numbers reported in Figures 6 and 7.

The running times obtained in the experiments may differ from the numbers reported in the paper if a different machine and/or compiler is used.

A.6 Experiment Customization

If `numactl` is not installed, the scripts can be modified to run without `numactl` by deleting the "`numactl -i all`" statements (potentially with some performance degradation on multi-socket machines).

Individual hypergraph algorithms can be tested by running the executables in `apps/hyper/` with the desired dataset as input. The "`-s`" flag should be passed if the hypergraph is symmetric. For traversal algorithms, one can pass the "`-r`" flag followed by an integer to indicate the ID of the source vertex (by default, vertex 0 is used as the source). The programs are run for three trials by default, but one can change the number of trials by passing the "`-rounds`" flag followed by an integer indicating the desired number of trials.

To test on other hypergraphs, datasets with communities can be downloaded from the Stanford Large Network Dataset Collection (SNAP) [59] and bipartite graphs can be downloaded from the Koblenz Network Collection (KONECT) [55]. SNAP datasets can be converted to Hygra format using the `communityToHyperAdj` program in the `utils/` directory. KONECT datasets can be converted to Hygra format using the `KONECTtoHyperAdj` program in the `utils/` directory.