HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion

Christian Mayer, Ruben Mayer*, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel
Institute of Parallel and Distributed Systems, University of Stuttgart, Germany
email: {firstname.lastname}@ipvs.uni-stuttgart.de
* Technical University of Munich, Germany
email: ruben.mayer@tum.de

***Abstract*—Many important real-world applications—such as social networks or distributed data bases—can be modeled as hypergraphs. In such a model, vertices represent entities—such as users or data records—whereas hyperedges model a group membership of the vertices—such as the authorship in a specific topic or the membership of a data record in a specific replicated shard. To optimize such applications, we need an efficient and effective solution to the NP-hard balanced k-way hypergraph partitioning problem. However, existing hypergraph partitioners that scale to very large graphs do not effectively exploit the hypergraph structure when performing the partitioning decisions. We propose HYPE, a hypergraph partitionier that exploits the neighborhood relations between vertices in the hypergraph using an efficient implementation of neighborhood expansion. HYPE improves partitioning quality by up to 95% and reduces runtime by up to 39% compared to streaming partitioning.**

***Keywords*-hypergraph partitioning; neighborhood expansion**

I. Introduction

Many real-world applications model problems as hypergraphs where connections between vertices are multi-dimensional, i.e., each vertex can directly communicate to n vertices in a group (called "hyperedge") and each vertex can be in multiple groups. Hypergraph partitioning [1], [2], [3], [4], [5], [6], [7], [8] deals with the problem of optimally dividing a hypergraph into a set of equally-sized components. Applications of hypergraph partitioning arise in diverse areas such as VLSI design placement [9], optimizing the task and data placement in workflows [10], minimizing the number of transactions in distributed data placement [11], optimizing storage sharding in distributed data bases [7], and as a necessary preprocessing step in distributed hypergraph processing [12].

Formally, dividing the hypergraph is denoted as the *balanced k-way hypergraph partitioning problem*. The goal is to divide the hypergraph into k equally-sized partitions such that the number of times neighboring vertices are assigned to different partitions is minimal (this is denoted as the (k-1) metric, as introduced later). The balanced k-way hypergraph partitioning problem is NP-hard. Hence, a heuristic approach is needed to solve the problem for massive hypergraphs.

In literature, a couple of heuristic hypergraph partitioning algorithms have been proposed, but they have shortcomings. Streaming hypergraph partitioning [8] considers one vertex at a time from a stream of hypergraph vertices. Based on a scoring function, it greedily assigns each vertex from the stream to the partition that yields the best scoring. While this algorithm has low run-time, it does not consider all relationships between all vertices when deciding on the partitioning, so that partitioning quality suffers. A recent hypergraph partitioning algorithm, Social Hash Partitioner [7], considers the complete hypergraph at once. It iteratively performs random permutations of the current partitioning followed by a greedy optimization to choose a better permutation over a worse one. While this approach generally converges to some form of an improved partitioning and is highly scalable, we argue that random permutations may not be the most effective choice for the partitioning heuristic.

In the related field of *graph* partitioning, a recently proposed algorithm uses *neighborhood expansion* to exploit structural properties of natural graphs [13]. Graphs can be regarded as special cases of hypergraphs, where each hyperedge contains only a single vertex. However, the original neighborhood expansion algorithm for graphs cannot be directly applied to hypergraphs. As hyperedges may contain a very large number of vertices, the neighborhood of a single vertex can be huge, rendering neighborhood expansion infeasible.

In this paper, we are the first researchers who successfully apply neighborhood expansion to hypergraph partitioning. We propose HYPE, a hypergraph partitioning algorithm specifically tailored to real-world hypergraphs with skewed degree distribution. HYPE grows k partitions based on the neighborhood relations in the hypergraph. We evaluate the performance of HYPE on a set of real-world hypergraphs, including a novel hypergraph data set consisting of authors and subreddits from the online board Reddit. Reddit is, to the best of our knowledge, the largest *real-world hypergraph* that has been considered in evaluating hypergraph partitioning algorithms up to now. In our evaluations, we show that HYPE can partition very large hypergraphs efficiently with high quality. HYPE is 39% faster and yields 95% better partitioning quality than *streaming partitioning* [8]. We released the source code of HYPE as an open source project: https://github.com/mayerrn/HYPE.

II. Problem Formulation

In this section, we formulate the hypergraph partitioning problem addressed in this paper.

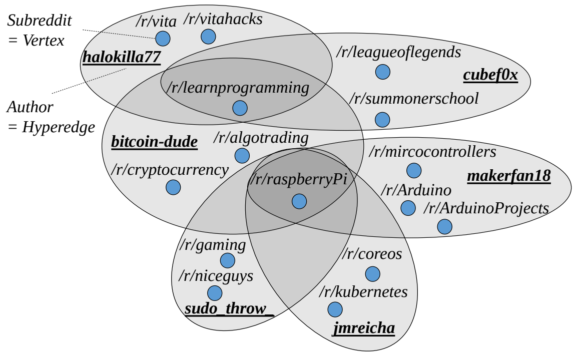


Fig. 1. A small extract of the global Reddit graph.

G	Hypergraph $G = (V, E)$
V	Set of vertices $V \subset \mathcal{N} \times \mathcal{N}$
E	Set of hyperedges $E \subset 2^V$
$N(v)$	Set of adjacent vertices of vertex $v \in V$
$N(X)$	Union of all sets $N(x)$ for $x \in X$
P	Set of partitions $p_i \in P$ with $\lvert P \rvert = k$
A	Function assigning vertices V to partitions P
λ	Balancing factor
C	Current core set of vertices
F	Current fringe
$d_{ext}(u, S)$	External neighbors score of vertex u
s	Maximal size of the fringe
r	Number of fringe candidate vertices

TABLE I
NOTATION OVERVIEW.

Problem Formulation: The hypergraph is given as $G = (V, E)$ with the set of vertices V and the set of hyperedges $E \subset 2^V$. Given vertex $v \in V$, we denote the set of adjacent vertices, i.e., the set of neighbors of v, as $N(v) \subseteq V$. The goal is to partition the hypergraph into k partitions $P = \{p_0, p_1, ..., p_{k-1}\}$ by assigning vertices to partitions. The assignment function $A : V \rightarrow P$ defines for each vertex in V to which partition it is assigned. We write $A(v) = p_i$ if vertex v is assigned to partition p_i. Each hyperedge spans between 1 and k partitions. The optimization objective is the $(k-1)$-cut that sums over each hyperedge the number of times it is assigned to more than one partition, i.e., $\sum_{e \in E} |\{p \in P | \exists v \in E : A(v) = p\}| - 1$. We require that the assignment of vertices to partitions is balanced in the number of vertices assigned to a single partition, i.e., $\forall p_0, p_1 \in P : |\{v \in V | A(v) = p_0\}| < \lambda |\{v \in V | A(v) = p_1\}|$ for a small balancing factor $\lambda \in \mathbb{R}$. This problem is denoted as *balanced k-way hypergraph partitioning problem* and it is NP-hard [14].

Reddit Hypergraph Example: We give an example of the Reddit hypergraph in Figure 1. The Reddit social network consists of a large number of subreddits where each subreddit concerns a certain topic. For example, in the subreddit `/r/learnprogramming`, authors write comments related to the topic of learning to program. Each author writes comments in an arbitrary number of subreddits and each subreddit is authored by an arbitrary number of users. One way to build a hypergraph out of the Reddit data set (see Section IV) is to use subreddits as vertices and authors as hyperedges that connect these vertices. This hypergraph representation provides valuable information about the similarity of subreddits. For instance, if many authors are active in two subreddits (e.g. `/r/Arduino` and `/r/ArduinoProjects`), i.e., many hyperedges overlap significantly in two vertices, it is likely that the two subreddits concern similar content.

The size of the hyperedges and the degree of the vertices resemble a power law degree distribution for real-world graphs such as Reddit, Stackoverflow, and Github. Hence, most vertices have a small degree and most hyperedges have a small size. These parts of the graph with small degrees build relatively independent communities. In the reddit graph, there are local communities such as people who write in the `/r/Arduino` and `/r/ArduinoProjects` subreddits but not in, say the `/r/Baby` subreddit. This property of strong local communities is well-observed for graphs [15]—and it holds for real-world hypergraphs as well. Power law distributions are long-tailed, i.e., there are some hub vertices or edges with substantial sizes or degrees. In graph literature, it has been shown that focusing on optimal partitioning of the local communities at the expense of optimal placement of the hubs is a robust and effective partitioning strategy [16], [17], [18]. In Section III, we show how we exploit this observation in the HYPE partitioner.

III. The Algorithm

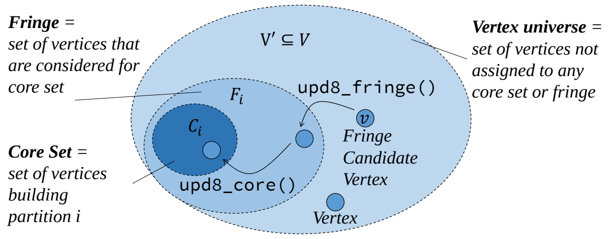
In the following, we first explain the idea of neighborhood expansion in Section III-A. We introduce our novel hypergraph partitioning algorithm HYPE in Section III-B and discuss the balancing of hypergraph partitions in Section III-C. Finally, we present a more formal pseudocode notation of the HYPE algorithm in Section III-D and perform a complexity analysis in Section III-E.

A. Neighborhood Expansion Idea

A practical method for high-quality *graph* partitioning is *neighborhood expansion* [13]. The idea is to grow a core set of vertices via the neighborhood relation given by the graph structure. By exploiting the graph structure, the locality of vertices in the partition is maximized, i.e., neighboring vertices in the graph tend to reside on the same partition. The algorithm grows the core set one vertex at a time until the desired partition size is achieved. In order to partition the graph into k partitions, the procedure of growing a core set C_i is repeated k times for $i \in \{0, 1, ..., k-1\}$.

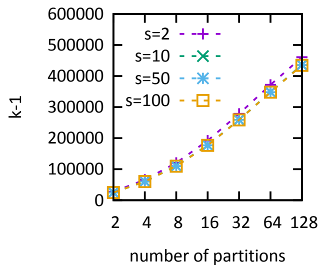
We aim to adopt neighborhood expansion to hypergraph partitioning. To this end, we have to overcome a set of challenges which are related to the different structure of hypergraphs when compared to normal graphs. In particular, the number of neighbors of a vertex quickly explodes as the hyperedges contain multiple neighboring vertices at once. Before we explain in detail those challenges and our approach to tackle them, we sketch the basic idea of neighborhood expansion in the following.

Figure 2 sketches the general framework for growing the core set C_i of partition $p_i \in P$. There are three overlapping

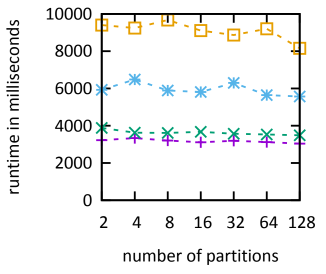


Fringe = set of vertices that are considered for core set
Core Set = set of vertices building partition i
V' ⊆ V
Vertex universe = set of vertices not assigned to any core set or fringe
F_i
C_i
upd8_fringe()
upd8_core()
v
Fringe Candidate Vertex
Vertex

Fig. 2. High-level idea of neighborhood expansion.



(a) Partitioning quality.



(b) Runtime.

Fig. 3. Limiting the fringe size s to a small value (e.g. $s = 10$) keeps partitioning quality intact while reducing runtime significantly (StackOverflow hypergraph).

sets: the vertex universe, the fringe, and the core set. The vertex universe $V' \subseteq V$ is the set of *remaining vertices* that can potentially be added to the fringe F_i, i.e., $V' = V \setminus C_0 ... \setminus C_i \setminus F_i$. The fringe F_i is the set of vertices that are currently considered for the core set. The core C_i is the set of vertices that are assigned to partition $p_i \in P$. The three sets are non-overlapping, i.e., $V' \cap F_i = F_i \cap C_i = C_i \cap V' = \emptyset$.

Initially, the core consists of seed vertices that are taken as a starting point for growing the partition. Based on these seed vertices, the fringe contains a subset of all neighboring vertices. In graph partitioning [13], the fringe contains not a subset but all vertices that are in a neighborhood relation to one of the vertices in the core set. In the Figure 2, a fringe candidate vertex, say vertex v, is moved from the vertex universe to the fringe and then to the core set. In other words, any strategy based on neighborhood expansion must define the two functions `upd8_fringe()` and `upd8_core()`.

As we develop a hypergraph partitioning algorithm based on the neighborhood expansion framework, we define the neighborhood relation and the three vertex sets accordingly. However, migrating the idea of neighborhood expansion from graph to hypergraph partitioning is challenging. The number of neighbors of a specific vertex in a typical hypergraph is much larger than in a typical graph. The reason is that the number of neighbors is not only proportional to the number of incident hyperedges but also to the size of these hyperedges. For example, suppose you are writing a comment in the `/r/Python` subreddit. Suddenly, hundreds of thousands other authors in `/r/Python` are your direct neighbors. In other words, the neighborhood relation is *group based* rather than *bidirectional* which leads to massive neighborhoods. The large number of neighbors changes the runtime behavior and efficiency of neighborhood expansion. For instance, in such a hypergraph, the fringe would suddenly contain a large fraction of the vertices in the hypergraph. But selecting a vertex from the fringe requires $\mathcal{O}(|V|)$ comparisons. This leads to high runtime overhead and does not scale to massive graphs. HYPE alleviates this problem by reducing the search space significantly as described next.

B. HYPE Algorithm

The HYPE algorithm grows the core set for partition $p_i \in P$ one vertex at a time. We load one vertex from the fringe to the core set and update the fringe with fresh vertices from the vertex universe. The decision of which vertex to include into the core and fringe sets is a critical design choice that has impact on the algorithm runtime and partitioning quality.

In this section, we explain the HYPE algorithm in detail, including a discussion of the design choices. In doing so, we first provide the basic algorithm in Section III-B1 and then discuss optimizations of the algorithm in Section III-B2. These optimizations tremendously reduce the algorithm runtime without compromising on partitioning quality.

1) Basic Algorithm:

The approach of growing a core set C_i is repeated in a sequential manner for each partition p_i for $i \in \{1, 2, ..., k\}$. It consists of a four step process. We initialize the computation with step 1., and iterate steps 2. and 3., until the algorithm terminates as defined in step 4.

1) Initialize the core set.
2) Move vertex from vertex universe into fringe.
3) Move vertex from fringe into core set.
4) Terminate the expansion.

We now examine these steps in detail. A formal algorithmic description is given in Section III-D.

1. Initialize the core set: The core set C_i must initially contain at least one vertex in order to grow via the neighborhood expansion. In general, there are many different ways to initialize the core set. This problem is similar to the problem of initializing a cluster center for iterative clustering algorithms such as `K-Means` [19]. Here, the defacto standard is to select random points as cluster centers [20], [21]. In fact, a comparison of several initialization methods for `K-Means` shows that the random method leads to robust clustering quality [22]. As the problem of selecting an initial "seed" vertex from which the core set grows is similar to this problem, we perform random initialization.

2. Move vertex to fringe: The function `upd8_fringe()` determines which vertices move from the *vertex universe* to the fringe F_i. The vertex universe consists of all vertices that are neither in the fringe F_i, nor in the core set C_i of any previous execution of the algorithm for any partition $p_j \in P$ with $j < i$.

The standard strategy of neighborhood expansion is to fill the fringe F_i with all vertices that are neighbors to any core vertex, i.e., $F_i \leftarrow N(C_i) \setminus C_0 \setminus ... \setminus C_i$. But for real-world hypergraphs, this quickly overloads the fringe with a large number of vertices from the vertex universe. To prevent this,

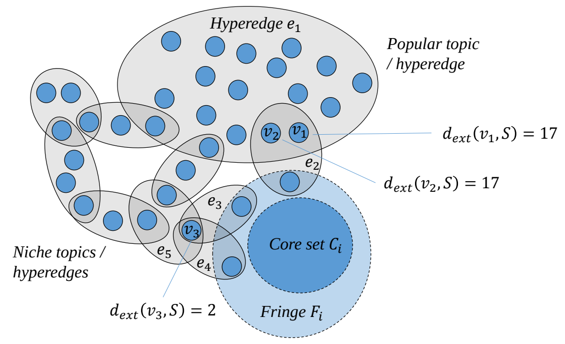


Fig. 4. The external neighbors metric determines which vertex to move into the fringe.

we restrict the fringe to contain only s vertices, i.e., $|F_i| \leq s$. In Figure 3, we validate experimentally that setting s to a small constant value, i.e., $s = 10$, keeps partitioning quality high but reduces runtime by a large factor. For brevity, we omit the discussion of similar results observed for other hypergraphs.

But how do we select the next vertex to be loaded into the fringe? Out of the vertex universe V', we select a vertex to be included to the fringe using a scoring metric as described in the next paragraphs. The intuition behind this scoring metric is to find the vertex that preserves the highest *locality* when assigned to the core set.

To this end, we define *locality* as the frequency that for a given vertex $v \in V'$, a neighboring vertex $v' \in N(v)$ resides on the same partition. High locality leads to low cut sizes and good partitioning quality. To improve locality, our goal is to grow the core set into the smaller local communities and assign all vertices of these smaller communities to the same partition. If a high proportion of neighbors of vertex $v \in V'$ is already assigned to the core set, assigning vertex v to the core set as well will improve locality.

In Figure 4, we see an example hypergraph that has the typical properties of real-world hypergraphs: the size of the hyperedges follows a power law distribution. To grow the fringe, there are three options: include vertex v_1, v_2, or v_3. Intuitively, we want to grow the fringe towards the local communities to preserve locality. We achieve this by selecting vertices based on the *external neighbors* with respect to the fringe F. In other words, we want to add vertices to the fringe that have a *high* number of neighbors in the fringe or the core set, and a *low* number of neighbors in the remaining vertex universe. A vertex with low external neighbors score tends to have high locality in the fringe and the core set. Formally, we write $d_{ext}(v, F_i)$ to denote the number of neighboring vertices of v that are not already contained in the fringe as defined in Equation 1.

$$d_{ext}(v, F_i) = |N(v) \setminus F_i| \tag{1}$$

We denote the vertices for which we calculate the external neighbors score as *fringe candidate vertices*. For each execution of `upd8_fringe()`, we select r fringe candidate vertices. The fringe F_i contains up to s vertices. Hence, after assigning one vertex to the core set, we take the top s vertices out of the $s-1+r$ fringe candidate vertices as the new fringe.

In Section III-B2, we describe three optimizations on the `upd8_fringe()` function that reduce the runtime while keeping the partitioning quality intact.

3. Move vertex to core set: Next, we describe the function `upd8_core()` that moves a vertex from the fringe F_i to the core set C_i. The function simply selects the vertex with smallest (cached) external neighbors score. This vertex v is then moved to the core, i.e., $C_i \leftarrow C_i \cup \{v\}$. This decision is final. Once assigned to the core C_i, vertex v will never be assigned to any other core C_j when considering a partition $j > i$. Hence, we remove the vertex from the remaining set of vertices in the vertex universe, i.e., $V' \leftarrow V' \setminus \{v\}$. In case the fringe can not be filled with enough neighbors, we add a random vertex from the vertex universe to the fringe and proceed with the given algorithm.

4. Terminate the expansion: We terminate the algorithm as soon as the core set contains $\frac{|V|}{k}$ vertices. This leads to perfectly balanced partitions with respect to the number of vertices. Upon termination, we release the vertices in the fringe F_i and store the vertices in the core set C_i in a separate partitioning file for partition i. After this, we restart expansion for the next partition $i \leftarrow i+1$ or terminate if all vertices have been assigned to partitions. In Section III-C, we discuss other possible balancing schemes.

2) Optimization of Fringe Updates: When moving a vertex from the vertex universe to the fringe, we have to be careful in order to *efficiently* select a good vertex. Calculating a score for all vertices in the vertex universe would be much too expensive. For example, suppose we add vertex v_2 to the fringe in the example in Figure 4. Suddenly, all vertices in the huge hyperedge e_1 could become fringe candidate vertices for which we would have to calculate the external neighbors score. Note that to calculate the external neighbors score, we must perform set operations that may touch an arbitrary large portion of the global hypergraph.

Our strategy to address this issue involves three steps: (a) select the best fringe candidate vertices from the vertex universe in an *efficient* manner by traversing small hyperedges first, (b) reduce the number of fringe candidate vertices r to $r = 2$, and (c) reduce the computational overhead to calculate the score for a fringe candidate vertex by employing a scoring cache. These three optimizations help us to limit the overhead per decision of which vertex to include into the fringe. Next, we describe the optimizations in detail.

*a) Maximize the chance to select r **good** fringe candidate vertices:* First, we describe how we maximize the chance to select good fringe candidate vertices from the vertex universe. The optimal vertex to add to the fringe has minimal external neighbors score, i.e., $argmin_{v \in V'} d_{ext}(v, F_i)$. Vertices that reside in large hyperedges (e.g. e_1 in Figure 4) have a high number of neighbors. Hence, it is unlikely that these vertices have a low external neighbors score with respect to the fringe F. For example, in Figure 4, vertices v_1 and v_2 have 18

Dataset	Vertices	Hyperedges	#Vertices	#Hyperedges	#Edges
Github [23]	Users	Projects	177,386	56,519	440,237
StackOverflow [23]	Users	Posts	641,876	545,196	1,301,942
Reddit	Subreddits	Authors	430,156	21,169,586	179,686,265
Reddit-L	Comments	Authors & Subreddits	2,862,748,675	21,599,742	5,725,497,350

TABLE II
REAL-WORLD HYPERGRAPHS USED IN EVALUATIONS.

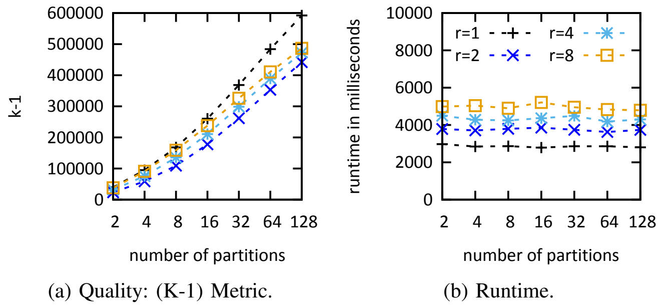


Fig. 5. Limiting the number of fringe candidate vertices r to $r = 2$ leads to the best partitioning quality (StackOverflow hypergraph).

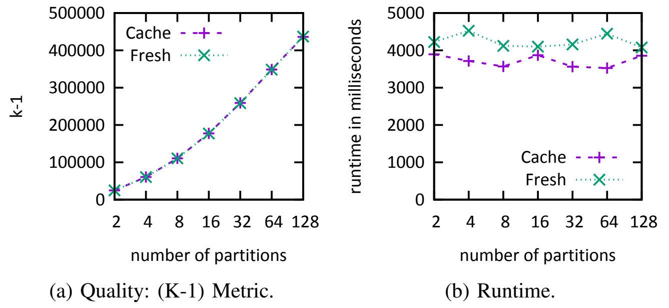


Fig. 6. The caching optimization for external neighbors score computation keeps partitioning quality intact while reducing runtime by up to 20% on the Stackoverflow hypergraph.

neighbors, whereas vertex v_3 has only 4 neighbors. Thus, vertex v_3 has a much higher chance of being the vertex with the minimal external neighbors score. Based on this observation, we optimize the selection of fringe candidate vertices by ordering all hyperedges that are incident to the fringe F_i with respect to their size and consider only vertices in the smallest hyperedge for inclusion into the fringe (e.g., hyperedges e_4, e_3, e_2 with $|e_4| \leq |e_3| \leq |e_2|$ results in the initial selection of hyperedge e_4). When we cannot retrieve r vertices from the smallest hyperedge (because it does not contain enough vertices that are not already in C or F), we proceed with the next larger hyperedge.

b) Reduce the number of fringe candidate vertices to be selected: Next, we limit the number of fringe candidate vertices to $r = 2$ vertices. From these r vertices, we select the vertex with the smaller external neighbors score. The basic principle of selecting the best out of two random choices is known in literature as *"the power of two random choices"* [24] and has inspired our design. We experimentally validated that considering more than two options, i.e., $r > 2$, does not significantly improve the decision quality, cf. Figure 5. Clearly, the lower the number of fringe candidate vertices r is, the lower is the runtime of the algorithm. Interestingly, using two choices, i.e., $r = 2$ leads to *better* partitioning quality than all other settings of r. Apparently, a higher value for r forces the algorithm to consider fringe candidate vertices from larger hyperedges which distracts the algorithm from the smaller hyperedges.

c) Reduce the overhead to compute an external neighbors score for fringe candidate vertices: The external neighbors score requires calculation of the set intersection between two potentially large sets (see Equation 1). This calculation must be done for all fringe candidate vertices. To prevent recomputation, we use a caching mechanism. The score is calculated only when the vertex is accessed for the first time (lazy caching policy). While this means that the cached score may change when including more and more vertices into the fringe, our evaluation results show that partitioning quality stays the same when using caching (see Figure 6). But the benefit of reducing score computations improves runtime by up to 20%.

C. Balancing Considerations

The default balancing objective of the HYPE algorithm leads to a balanced number of vertices on each partition. For n vertices and k partitions, the algorithm repeats the neighborhood expansion, one vertex at a time, until there are exactly $max = n/k$ vertices per partition. Vertex balancing is the standard method for distributed graph processing systems such as Pregel [25]—considering that the workload per partition is roughly proportional to the number of vertices per partition. Therefore, many practical algorithms such as the popular multilevel k-way hypergraph partitioning algorithm [1] focus on vertex balancing.

However, some applications of hypergraph partitioning may benefit from balancing the sum of vertices *and* hyperedges [8]. More precisely, for n vertices and m hyperedges, the algorithm should partition the hypergraph in a way such that each partition is responsible for $\frac{n+m}{k}$ vertices or hyperedges. In the following, we discuss two ideas how HYPE can achieve this. First, we assign a weight $w(v)$ to each vertex v, i.e., the weight $w(v) = 1 + |E_v|$ with E_v being the set of incident hyperedges of vertex v. Then, we repeat the neighborhood expansion algorithm by assigning vertices until each partition has $max = \frac{n+m}{k}$ total weight (or less). The rationale behind this method is the law of large numbers: it is not likely that a single vertex assignment will suddenly introduce a huge imbalance in relation to the already assigned vertices. Second, to achieve perfect edge balancing, we can flip the hypergraph, i.e., viewing each original vertex as a hyperedge and each original hyperedge as a vertex. When balancing the number of vertices in the flipped graph, we actually balance the number of hyperedges in the original graph. After termination of the algorithm, we flip the hypergraph back to the original representation. We leave an investigation of other balancing constraints as future work.

D. HYPE Pseudocode

Algorithm 1 HYPE algorithm for hypergraph $G = (V, E)$.

```
1: $V' \leftarrow V$
2: for $i \in [0..k-1]$ do
3:     $C_i \leftarrow \{V'.random()\}$
4:     $V' \leftarrow V' \setminus C_i$
5:     $F_i \leftarrow \{\}$
6:     $c =< key, val > \{\}$                    ▷ clear cache
7:     while $|C_i| < \frac{|V|}{k}$ do
8:         UPD8_FRINGE()
9:         UPD8_CORE()
```

Algorithm 1 lists the main loop. We repeat the following method for all partitions $p_i \in P$. After some housekeeping tasks such as filling the core set C_i of partition p_i with a random vertex (line 3), initializing the fringe (line 5), and clearing the $(key, value)$ cache (line 6), we repeat the main loop (lines 7-8) until the core set has exactly $\frac{|V|}{k}$ vertices. The loop body consists of the two functions `upd8_fringe()` and `upd8_core()` that are described next.

Algorithm 2 The function `upd8_fringe()` updates the fringe F_i with vertices from the vertex universe V'.

```
1: function UPD8_FRINGE()
2:
3:     #Determine r fringe candidate vertices
4:     $F_{cand} \leftarrow \{\}$
5:     $X \leftarrow \{e \in E | C_i \cap e \neq \emptyset\}$
6:     $X' \leftarrow [e_0, e_1, ... | e_j \in X \wedge |e_j| > |e_{j-1}|]$
7:     for $e \in X'$ do
8:         for $v \in e \wedge v \notin F_i \wedge \forall j \leq i : v \notin C_j$ do
9:             if $|F_{cand}| < r$ then
10:                $F_{cand} \leftarrow F_{cand} \cup \{v\}$
11:            else
12:                break loop line 7
13:
14:    #Update cache
15:    for $v \in F_{cand} \wedge v \notin c.keys()$ do
16:        $c(v) \leftarrow d_{ext}(v, F_i)$
17:
18:    #Update fringe
19:    $F_i' \leftarrow [v_0, v_1, ... | v_j \in F_i \cup F_{cand} \wedge c(v_{j-1}) < c(v_j)]$
20:    $F_i \leftarrow \{v | v \in F_i'.subsequence(0..s-1)\}$
21:    if $F_i = \emptyset$ then
22:        $F_i \leftarrow \{V'.random()\}$
```

Algorithm 2 lists the `upd8_fringe()` function. The function consists of three steps: determine the r fringe candidate vertices (lines 3-10), update the cache (lines 12-14), and update the fringe (lines 16-20). The first step calculates the fringe candidate vertices F_{cand} by first sorting the hyperedges that are incident to the core set C_i by size (ascending) and then traversing these hyperedges for vertices that can still be added to the fringe (i.e., are not already assigned to the fringe or any core set). The second step updates the cache with the current external neighbors score with respect to the current fringe F_i. The third step sets the fringe to the set of top s vertices with respect to the external neighbors score. If the fringe is still empty after these steps, we initialize it with a random vertex.

Algorithm 3 The function `upd8_core()` updates the core C_i with vertices from the fringe F_i.

```
1: function UPD8_CORE()
2:     $v \leftarrow argmin_{v \in F_i} c(v)$
3:     $C_i \leftarrow C_i \cup \{v\}$
4:     $F_i \leftarrow F_i \setminus \{v\}$
5:     $V' \leftarrow V' \setminus \{v\}$
```

Algorithm 3 lists the `upd8_core()` function. We load the vertex with the minimal cached external neighbors score into the core C_i and remove this vertex from the fringe F_i and the vertex universe V'.

E. Complexity Analysis

For the following analysis, we denote the number of vertices as $n = |V|$ and the number of hyperedges as $m = |E|$. Algorithm 1 repeats for k partitions the procedure of moving $\frac{n}{k}$ vertices from the vertex universe to the fringe and from the fringe to the core. Next, we analyze the runtime for those procedures `upd8_fringe()` and `upd8_core()`.

The function `upd8_fringe()` in Algorithm 2 consists of three steps. First, it determines r fringe candidate vertices from the vertex universe (lines 3-12). As we set r to a small constant ($r = 2$), this step is very fast in practice with the following caveat: The algorithm needs to sort the incident hyperedges with respect to the hyperedge size (line 6). The computational complexity of sorting all hyperedges is $O(m * log(m))$. It is sufficient to sort the hyperedges only once in the beginning of the HYPE algorithm. Recap that the algorithm fills the fringe with $r = 2$ new candidate vertices. In the worst case, the fringe is incident to all hyperedges in the hypergraph. Therefore, selecting the fringe candidate vertices is in $O(m)$ to iterate over all hyperedges. ***This is a pessimistic estimation—in practical cases it is sufficient to check the first few smallest hyperedges to find the*** $r = 2$ ***candidates***. Second, the algorithm updates the cache with fresh external degree scores for new candidates vertices (lines 14-16). It calculates the external degree score at most once for every candidate vertex (it is just read from cache if needed again later). As there are only $r = 2$ fringe candidate vertices in each execution of `upd8_fringe()`, the total number of external degree score calculations is limited to $2 * n$. The overhead of calculating the external degree of a vertex with respect to a set of s fringe vertices is $O(s)$ (cf. Equation 1), but s is a constant ($s = 10$). Hence, the total computational complexity of updating the cache is $O(n)$. Third, the algorithm updates the fringe with vertices from the fringe candidates (lines 18-22). As both the fringe and the fringe candidates have constant sizes $r = 2$ and $s = 10$, the complexity of the third step is $O(1)$.

The function `upd8_core()` in Algorithm 3 selects the vertex with minimal cached external neighbors score from the constant-sized fringe. Thus, the complexity is $O(1)$ including the housekeeping tasks in lines 3-5.

In total, the worst-case computational complexity of the HYPE algorithm is $O(m * log(m) + k * \frac{n}{k} * m + n)) = O(m * log(m) + n * m)$. As highlighted above, in practice we observe that only a small, constant number of hyperedges is checked in order to find the r candidate vertices, so that we observe a complexity of $O(m * log(m) + n)$.

IV. Evaluations

In this section, we evaluate the performance of HYPE on several real-world hypergraphs.

Experimental Setup: All experiments were performed on a shared memory machine with 4 x Intel(R) Xeon(R) CPU E7-4850 v4 @ 2.10GHz (4 x 16 cores) with 1 TB RAM. The source code of our HYPE partitioner is written in C++.

Hypergraph Data Sets: We perform the experiments on different real-world hypergraphs, i.e., Github [23], StackOverflow [23], Reddit and Reddit-L[1], as listed in Table II. All of the hypergraphs show a power law distribution of vertex and hyperedge degrees. In addition to the number of vertices and hyperedges, we report the number of edges. An edge represents an assignment of a vertex to a hyperedge.

We highlight that for this paper, we crawled two large real-world hypergraphs from the Reddit dataset using the relations between authors, subreddits and comments.

Benchmarks: We choose our benchmarks for evaluating HYPE based on 3 categories.

Group (I) consists of a wide range of hierarchical partitioners such as hMetis [1], Mondriaan [2], Parkway [3], PaToH [4], Zoltan [5], and KaHyPar [6]. As no partitioner in group (I) consistently outperforms the other partitioners in terms of partitioning quality, scalability and partitioning time, we decided for the well-established and widely used hypergraph partitioner hMETIS. Several recent papers show that hMETIS leads to competitive partitioning performance with respect to the $(k - 1)$ metric [7], [26], [8]. Hence, we chose hMETIS as the representative partitioner from group (I) in this paper. We run hMETIS in two different settings: with and without enforced vertex balancing. Due to the high partitioning quality, hMETIS serves as the benchmark for partitioning quality on small to medium hypergraphs. We used hMETIS in version 2.0pre1 with the parameters `-ptype=rb -otype=soed -reconst`. To enforce vertex balancing, we set[2] the parameter `UB=0.1`.

Group (II) consists of the recently proposed Social Hash Partitioner (SHP) [7]. The authors released the raw source code of SHP. Yet, we could not reproduce their results as neither configuration files and parameters, nor scripts, execution instructions, or documentations were provided. However, in our evaluations we partitioned hypergraphs of similar size as SHP in a similar runtime, even though HYPE uses a purely sequential partitioning algorithm.

Group (III) comprises multiple streaming partitioning strategies proposed by Alistarh et al. [8]. The greedy MinMax strategy constantly outperformed all other strategies according to their paper[3]. Moreover, we designed a novel vertex-balanced variant of MinMax that outperforms the original approach with respect to the (k-1) metric[4]. We denote this variant as *MinMax NB* (**n**ode **b**alanced) in contrast to the standard *MinMax EB* (**e**dge **b**alanced).

Experiments: In all experiments, we capture the following metrics. (1) The (k-1) metric to evaluate the *quality* of the hypergraph partitioning. This is the default metric for partitioning quality [1], [5]. Other partitioning quality metrics such as the *hyperedge-cut* and the *sum of external degree* performed similar in our experiments[5]. (2) The runtime of the algorithm to partition the whole input hypergraph in order to evaluate the *speed* of the partitioning algorithms. (3) The vertex imbalance as a metric to capture the *fairness* of the hypergraph partitioning. We compute vertex imbalance as the normalized deviation between the maximal and the minimal number of vertices assigned to any partition, i.e., $\frac{maxsize - minsize}{maxsize}$.

For each experiment, we increase the number of partitions from 2 up to 128 in exponential steps.

A. Performance Evaluations

The performance evaluations show the benefits of HYPE when partitioning large hypergraphs. Its runtime is independent of the number of partitions, so that it is faster than streaming partitioning when the number of partitions is large. Further, the hierarchical partitioning algorithm hMETIS does not scale to very large hypergraphs, i.e., it cannot partition the Reddit hypergraph, and takes orders of magnitude longer for the smaller hypergraphs. We discuss the detailed results next.

Github: Figure 7a shows the partitioning quality on the Github hypergraph. In the (k-1) metric, hMETIS performs best (e.g., 47 % better than HYPE at k = 128). HYPE performs up to 45% better than MinMax hyperedge-balanced, and up to 34 % better than MinMax vertex-balanced.

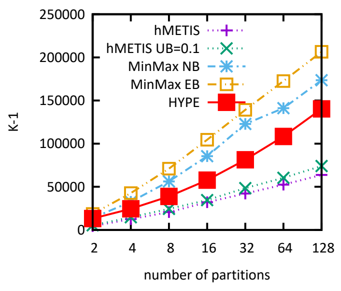
[1]https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/

[2]We determined the UB parameter experimentally such that the measured imbalance is comparable to HYPE and MinMax. Schlag et al. [27] provide an equation to set UB in order to enforce a specific imbalance constraint at a given k.

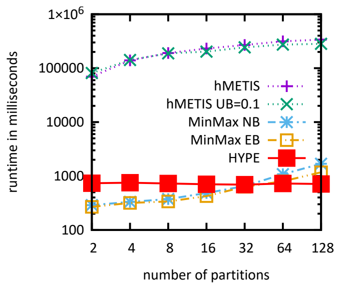
[3]The optimization goal of MinMax is to minimize the maximum number of hyperedges associated with a partition via the partition's vertices. However, both metrics MinMax and (k-1) are closely related: They measure the spread of hyperedges across partitions.

[4]We allowed a slack parameter of up to 100 vertices, cf. [8].

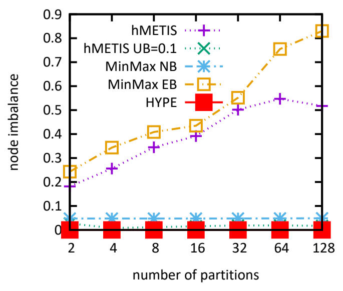
[5]The close relationship between these metrics is described in literature [7].



(a) Quality: (k-1) Metric.

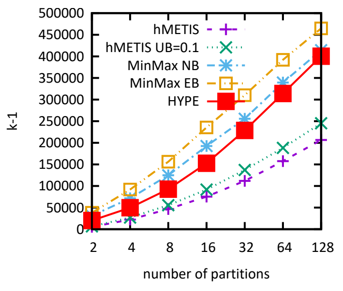


(b) Runtime.

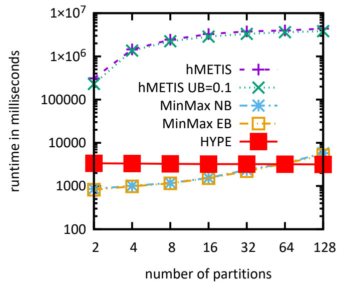


(c) Balancing.

Fig. 7. Evaluations on the Github hypergraph (lower is better).

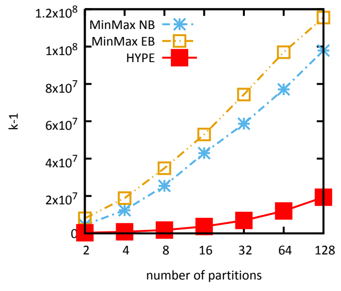


(a) Quality: (k-1) Metric.

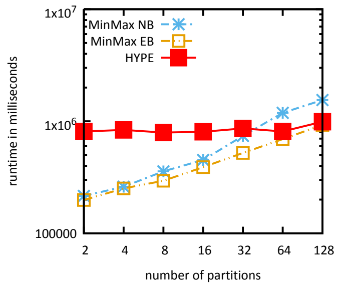


(b) Runtime.

Fig. 8. Evaluations on the StackOverflow hypergraph (lower is better).



(a) Quality: (K-1) Metric.



(b) Runtime.

Fig. 9. Evaluations on the Reddit hypergraph (lower is better).

Partitioning runtime is depicted in Figure 7b. hMETIS took 70 to 338 seconds to partition the Github hypergraph, which is orders of magnitude slower than MinMax and HYPE (up to 476 × slower than HYPE). The partitioning runtime of HYPE is independent of the number of partitions, as each partition is filled with vertices sequentially until it is full. Different from that, in MinMax, the partitioning runtime depends on the number of partitions, as MinMax works with a scoring function that computes for each vertex a score for each partition and then assigns the vertex to the partition where its score is best. Hence, for up to 32 partitions, HYPE has a higher runtime than MinMax, up to 2.7 × higher, whereas for 64 and 128 partitions, the runtime of HYPE is lower (up to 2.4 × lower).

In terms of balancing, cf. Figure 7c, HYPE shows perfect vertex balancing, while the MinMax vertex-balanced has a slight imbalance of up to 5%. Unsurprisingly, MinMax hyperedge-balanced has a poor vertex balancing. In hMETIS, when vertex balancing is turned on, the maximum imbalance was 3%, whereas without that flag, vertex imbalance was tremedously higher (up to 55% imbalance). The balancing results are similar for all other hypergraphs, so we will not discuss balancing in the following results any more.

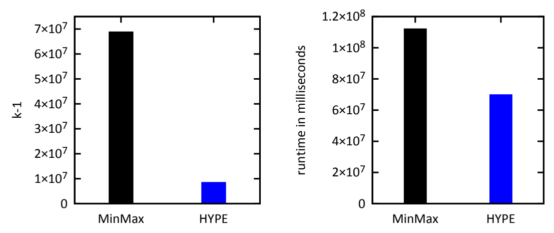
StackOverflow: Figure 8a shows the partitioning quality on the StackOverflow hypergraph. In the (k-1) metric, hMETIS performs best (e.g., 39 % better than HYPE at k = 128). HYPE performs up to 47% better than MinMax hyperedge-balanced, and up to 35% better than MinMax vertex-balanced.

Partitioning runtime is depicted in Figure 8b. hMETIS took between 206 and 4374 seconds (i.e., 73 minutes) to partition the StackOverflow hypergraph, which is orders of magnitude slower than MinMax and HYPE (up to 1,384 × slower than HYPE). In numbers, HYPE takes 3 seconds to build 128 partitions, while hMETIS takes more than 1 hour! The comparison between HYPE and MinMax on StackOverflow leads to similar results as on the Github hypergraph: With up to 32 partitions, MinMax is faster (up to 4.1 × faster), but with 64 and 128 partitions, HYPE is faster (up to 1.8 × faster). Again, the reason for this is that the HYPE runtime is independent of the number of partitions.

Reddit: Figure 9a shows the partitioning quality on the Reddit hypergraph. For this hypergraph, we could not produce results with the hMETIS partitioner, as it crashed or was running for days without returning a result. Consistent to the experiments reported in [7], many partitioners from group (I) are not able to partition such large hypergraphs. Hence, in the following, we only report results for MinMax and HYPE.

On the Reddit hypergraph, the advantage of exploiting local communities in HYPE pays off to the full extent: HYPE outperforms the streaming partitioner MinMax, that ignores the overall hypergraph structure, by orders of magnitude. For 2, 4 and 8 partitions, HYPE achieved an improvement of 95% compared to MinMax hyperedge-balanced, and 93% compared to MinMax vertex-balanced in the (k-1) metric. Thus, HYPE leads to a partitioning quality that is up to **20 × better** than when using MinMax. For 16 partitions, HYPE performs 93% and 91% better, for 32 partitions 91% and 88%, for 64 partitions 88% and 84%, and for 128 partitions 83% and 80% better than MinMax hyperedge-balanced and MinMax vertex-balanced partitioners, respectively.

Comparing the partitioning runtime of HYPE and MinMax in Figure 9b, we see again that the runtime of HYPE is



(a) Quality: (k-1) Metric. (b) Runtime.

Fig. 10. Evaluations on the Reddit-L hypergraph (lower is better).

independent of the number of partitions, whereas MinMax has a higher runtime with growing number of partitions because of its scoring scheme. While at 2 partitions, MinMax is up to 4 × faster than HYPE, at 64 partitions HYPE becomes faster than MinMax, by up to 36% at 128 partitions. As with the other hypergraphs, MinMax vertex-balanced is slightly slower than MinMax hyperedge-balanced as the hyperedge balance can change significantly after assigning a single vertex. This often forces assignment to a single partition (the least loaded) such that the partitions remain balanced. In such cases, the forced partitioning decisions for hyperedge-balanced partitioning can be performed very quickly.

Reddit-L: In Figure 10, we compare partitioning quality and runtime of HYPE against MinMax on the large Reddit-L hypergraph with $k = 128$ partitions. MinMax requires more than 31 hours to partition Reddit-L compared to the 19 hours of HYPE. Although being 39% faster than MinMax, HYPE still outperforms MinMax in partitioning quality by 88%: MinMax has a (k-1) score of 68,709,969 compared to HYPE's 8,357,200. Note that MinMax already belongs to the fastest high-quality partitioners. However, HYPE is able to outperform MinMax because its runtime does not depend on the number of partitions.

B. Discussion of the Results

We conclude that HYPE shows very promising performance in hypergraph partitioning. First, it is able to partition very large hypergraphs, which cannot be partitioned by algorithms from group (I). Second, it consistently provides better partitioning quality than streaming MinMax. On top of that, the HYPE algorithm is comparably easy to implement and to manage because all system parameters are fixed.

V. Related Work

In the last decades, research on hypergraph partitioning was driven by the need to place transistors on chips in Very-large-scale integration (VLSI) design [9], as logic circuits can be modeled as large hypergraphs that are divided among chips. The most popular hypergraph partitioning algorithm from that area is hMETIS [1] which is based on a multilevel contraction algorithm and produces good partitioning quality for medium-sized hypergraphs in the magnitude of up to 100,000 edges.

However, multilevel partitioning algorithms do not scale to large hypergraphs, as shown in our evaluations. Parallel implementations of multilevel partitioning have been proposed [5], but the problem of high computational complexity and memory consumption remains. For instance, *Zoltan* [5] is a parallel multilevel hypergraph partitioning algorithm. The evaluated graphs on Zoltan are relatively small—within a magnitude of up to 30 million edges or less—while using up to 64 parallel machines to process them. Other algorithms of that group are Mondriaan [2], Parkway [3], PaToH [4], and KaHyPar [6]. For hypergraphs with hundreds of millions of edges, these algorithms are not practical as they take hours or even days to complete, if they terminate at all.

The bad scalability of multilevel partitioning algorithms led to the development of more scalable partitioners. *Social Hash Partitioner (SHP)* achieves scalability to very large hypergraphs (up to 10 billion edges) by means of massive parallelization [7]. SHP performs random swaps of vertices between partitions and greedily chooses the best swaps. Random swaps fit well with the objective of parallelization and distribution in SHP, but may not be the most efficient heuristic. Investigating on the phenomenon of scalability versus efficiency [28], we conceived the idea to devise an efficient hypergraph partitioning algorithm.

Another approach to partition very large hypergraphs are *streaming* algorithms. Streaming hypergraph partitioning takes one vertex at a time from a stream of vertices, and calculates a score for each possible placement of that vertex on each of the partitions. The vertex is then placed on the partition where its placement score is best, and cannot be removed any more. Alistarh et al. [8] proposed different heuristic scoring functions, where greedily assigning vertices to the partition with the largest overlap of incident hyperedges is considered best. There are two issues with the streaming approach. First, by only taking into account a single vertex at a time and placing it, information about the neighborhood of that vertex is not exploited although available in the hypergraph. Second, the complexity of the algorithm depends on the number of partitions, as the scoring function is computed for each vertex on each partition. For a large number of partitions, streaming partitioning becomes slow.

A closely related problem is *balanced k-way graph partitioning* which faces similar challenges such as billion-scale graph data and the need for fast algorithms. Multilevel graph partitioners such as METIS [29] and ParMETIS [30] do not scale very well. Spinner [31] is a highly scalable graph partitioner that, like SHP, performs iterative random permutations and greedy selection of the best permutation. There is a large number of streaming graph partitioning algorithms, such as HDRF [17], H-load [32], and ADWISE [33]. The "neighborhood heuristic" by Zhang et al. [13] follows a completely different approach by exploiting the graph structure when performing partitioning decisions. The algorithm grows a core set by successively adding neighbors of the core set to a fringe set. However, the given heuristic can not be applied directly to hypergraph partitioning as the calculation of scores is way too expensive in hypergraphs (see Section III). While hypergraphs can be transformed into bipartite graphs, graph partitioning algorithms cannot be used to perform hypergraph partitioning. First, the bipartite graph representations contain one artificial vertex per hyperedge that destroys the vertex balancing requirement of hypergraph partitioning. Second, the (k-1) metric is ignored by graph partitioning algorithms.

In recent years, several distributed hypergraph systems emerged that fueled the need for efficient massive hypergraph partitioning. These systems are inspired from the area of distributed graph processing systems and apply the vertex-centric programming model from graph processing to hypergraph processing. For instance, HyperX [12] allows applications to specify vertex and hyperedge programs which are then executed iteratively by the system. Also, Mesh [34] builds upon the popular GraphX system [35] and shows promising performance. These systems show significant reduction of processing latency with improved partitioning quality.

VI. Conclusions

In this paper, we propose HYPE, an effective and efficient partitioner for real-world hypergraphs. The partitioner grows k core sets in a sequential manner using a neighborhood expansion algorithm with several optimizations to reduce the search space. Due to the simplicity of the design and focus on the hypergraph structure, HYPE is able to partition the large Reddit hypergraph with billions of edges in less than a day. This is the partitioning of one of the largest real-world hypergraph reported in literature. HYPE not only improves partitioning quality by up to 95% compared to streaming hypergraph partitioning, but *reduces* runtime as well by 39%.

A promising line of future research on HYPE is to explore how to grow the k core sets in parallel. In this scenario, several core sets *"compete"* for inclusion of attractive vertices, so the crucial questions are how to minimize the number of "collisions" and how to deal with collisions when they happen.

References

[1] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proc. of the 36th ACM/IEEE Design Automation Conference*, 1999.

[2] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM review*, vol. 47, no. 1, pp. 67–95, 2005.

[3] A. Trifunovi and W. J. Knottenbelt, "Parallel multilevel algorithms for hypergraph partitioning," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 563 – 581, 2008.

[4] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, Jul 1999.

[5] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proc. 20th Int. Parallel Distributed Processing Symposium*, 2006.

[6] T. Heuer and S. Schlag, "Improving coarsening schemes for hypergraph partitioning by exploiting community structure," in *16th Int. Symposium on Experimental Algorithms, (SEA 2017)*, 2017.

[7] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, and A. Shalita, "Social hash partitioner: a scalable distributed hypergraph partitioner," *Proc. of the VLDB Endowment*, vol. 10, no. 11, pp. 1418–1429, 2017.

[8] D. Alistarh, J. Iglesias, and M. Vojnovic, "Streaming min-max hypergraph partitioning," in *Proc. of the 28th Int. Conf. on Neural Information Processing Systems - Volume 2*, 2015.

[9] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[10] U. V. Çatalyürek, K. Kaya, and B. Uçar, "Integrated data placement and task assignment for scientific workflows in clouds," in *Proc. of the 4th Int. Workshop on Data-intensive Distributed Computing*, 2011.

[11] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 48–57, Sep. 2010.

[12] J. Huang, R. Zhang, and J. X. Yu, "Scalable hypergraph learning and processing," in *Proc. of the IEEE Int. Conf. on Data Mining*, 2015.

[13] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, "Graph edge partitioning via neighborhood heuristic," in *Proc. of the 23rd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2017.

[14] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, Nov 2006.

[15] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs." in *OSDI*, 2012.

[17] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proc. of the 24th ACM Int. on Conf. on Information and Knowledge Management*, 2015.

[18] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *Nature*, vol. 406, no. 6794, p. 378, 2000.

[19] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[20] P. S. Bradley and U. M. Fayyad, "Refining initial points for k-means clustering." in *ICML*, vol. 98. Citeseer, 1998, pp. 91–99.

[21] S. S. Khan and A. Ahmad, "Cluster center initialization algorithm for k-means clustering," *Pattern recognition letters*, vol. 25, no. 11, pp. 1293–1302, 2004.

[22] J. M. Pena, J. A. Lozano, and P. Larranaga, "An empirical comparison of four initialization methods for the k-means algorithm," *Pattern recognition letters*, vol. 20, no. 10, pp. 1027–1040, 1999.

[23] J. Kunegis, "Konect: the koblenz network collection," in *Proc. of the 22nd Int. Conf. on World Wide Web*, 2013.

[24] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optimization*, vol. 9, pp. 255–304, 2001.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. of the SIGMOD Int. Conf. on Management of data*, 2010.

[26] W. Yang, G. Wang, K.-K. R. Choo, and S. Chen, "Hepart: A balanced hypergraph partitioning algorithm for big data applications," *Future Generation Computer Systems*, vol. 83, pp. 250 – 268, 2018.

[27] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz, "k-way hypergraph partitioning via n-level recursive bisection," in *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2016, pp. 53–67.

[28] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *Proc. of the 15th USENIX Conf. on Hot Topics in Operating Systems*, 2015.

[29] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[30] ——, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 71–95, 1998.

[31] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *Proc. of the 33rd Int. Conf. on Data Engineering (ICDE)*, 2017.

[32] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel, "Graph: Traffic-aware graph processing," *IEEE Trans. on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1289–1302, 2018.

[33] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, L. Laich, L. Rieger, and K. Rothermel, "Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing," in *Proc. of the 38th Int. Conf. on Distributed Computing Systems (ICDCS)*, 2018.

[34] B. Heintz, S. Singh, C. Tesdahl, and A. Chandra, "Mesh: A flexible distributed hypergraph processing system," 2016.

[35] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." in *OSDI*, 2014.