FREIGHT: Fast Streaming Hypergraph Partitioning

Kamal Eyubov
Universität Heidelberg, Germany

Marcelo Fonseca Faraj
Universität Heidelberg, Germany

Christian Schulz
Universität Heidelberg, Germany

Abstract

Partitioning the vertices of a (hyper)graph into k roughly balanced blocks such that few (hyper)edges run between blocks is a key problem for large-scale distributed processing. A current trend for partitioning huge (hyper)graphs using low computational resources are streaming algorithms. In this work, we propose FREIGHT: a Fast stREamInG Hypergraph parTitioning algorithm which is an adaptation of the widely-known graph-based algorithm Fennel. By using an efficient data structure, we make the overall running of FREIGHT linearly dependent on the pin-count of the hypergraph and the memory consumption linearly dependent on the numbers of nets and blocks. The results of our extensive experimentation showcase the promising performance of FREIGHT as a highly efficient and effective solution for streaming hypergraph partitioning. Our algorithm demonstrates competitive running time with the Hashing algorithm, with a difference of a maximum factor of four observed on three fourths of the instances. Significantly, our findings highlight the superiority of FREIGHT over all existing (buffered) streaming algorithms and even the in-memory algorithm HYPE, with respect to both cut-net and connectivity measures. This indicates that our proposed algorithm is a promising hypergraph partitioning tool to tackle the challenge posed by large-scale and dynamic data processing.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Hypergraph partitioning, graph partitioning, edge partitioning, streaming

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.15

Related Version *Full Version*: `https://arxiv.org/pdf/2302.06259.pdf`

Funding We acknowledge support by DFG grant SCHU 2567/5-1.

1 Introduction

Graphs are ubiquitous in nature and can be used to represent a wide variety of phenomena such as road networks, dependencies in databases, communications in distributed algorithms, interactions in social networks, and so forth. Nevertheless, phenomena where interactions between entities are not necessarily pairwise are more adequately modeled by hypergraphs, which can capture higher-order interactions [23]. With the massive proliferation of data, processing large-scale (hyper)graphs on distributed systems and databases becomes a necessity for a wide range of applications. When processing a (hyper)graph in parallel, k processors operate on distinct portions of the (hyper)graph while communicating to one another through message-passing. To make the parallel processing efficient, an important preprocessing step consists of partitioning the vertices of the (hyper)graph into k roughly balanced blocks such that few (hyper)edges run between blocks. (Hyper)graph partitioning is NP-hard [16] and there can be no approximation algorithm with a constant ratio for general (hyper)graphs [8]. Thus, heuristics are used in practice. A current trend for partitioning huge (hyper)graphs quickly and using low computational resources are streaming algorithms [36, 5, 20, 13, 14, 25, 19, 3, 35].


© Kamal Eyubov, Marcelo Fonseca Faraj, and Christian Schulz;
licensed under Creative Commons License CC-BY 4.0

The most popular streaming approach in literature is the one-pass model [1], where vertices arrive one at a time including their (hyper)edges and then have to be permanently assigned to blocks. In the domain of graphs, most algorithms are either very fast but do not care for solution quality at all (such as `Hashing` [34]), or are still fast, but much slower and capable of computing significantly better solutions than just random assignments (such as such `Fennel` [36]). Recently, the gap between these groups of algorithms has been closed by a streaming multi-section algorithm [14] which is up to two orders of magnitude faster than `Fennel` while cutting only 5% more edges than it on average. In the domain of hypergraphs, there is a similar gap that has not yet been closed. In particular, there is the same trivial `Hashing` -based algorithm on one side, and more sophisticated and expensive algorithms [3, 35] on the other side.

In this work, we propose `FREIGHT`: a Fast stREamInG Hypergraph parTitioning algorithm that can optimize for the cut-net as well as the connectivity metric. By using an efficient data structure, we make the overall running time of `FREIGHT` linearly dependent on the pin-count of the hypergraph and the memory consumption linearly dependent on the numbers of nets and blocks. Our proposed algorithm demonstrates remarkable efficiency, with a running time comparable to the `Hashing` algorithm and a maximum discrepancy of only four in three quarters of the instances. Importantly, our study establishes the superiority of `FREIGHT` over all current (buffered) streaming algorithms and even the in-memory algorithm `HYPE`, in both cut-net and connectivity measures. This shows the potential of our algorithm as a valuable tool for partitioning hypergraphs in the context of large and constantly changing data processing environments.

2 Preliminaries

2.1 Basic Concepts

Hypergraphs and Graphs. Let $H = (V = \{0, \dots, n-1\}, E)$ be an *undirected hypergraph* with no multiple or self hyperedges, with $n = |V|$ vertices and $m = |E|$ hyperedges (or *nets*). A net is defined as a subset of V. The vertices that compose a net are called *pins*. A vertex $v \in V$ is *incident* to a net $e \in E$ if $v \in e$. Let $c : V \to \mathbb{R}_{\geq 0}$ be a vertex-weight function, and let $\omega : E \to \mathbb{R}_{>0}$ be a net-weight function. We generalize c and ω functions to sets, such that $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. Let $I(v)$ be the set of incident nets of v, let $d(v) := |I(v)|$ be the *degree* of v, let $d_w(v) := w(I(v))$ be the *weighted degree* of v, and let Δ be the maximum degree of H. We generalize the notations $d(.)$ and $d_w(.)$ to sets, such that $d(V') = \sum_{v \in V'} d(v)$ and $d_w(V') = \sum_{v \in V'} d_w(v)$. Two vertices are *adjacent* if both are incident to the same net. Let the number of pins $|e|$ in a net e be the *size* of e, let $\xi = \max_{e \in E}\{|e|\}$ be the maximum size of a net in H.

Let $G = (V = \{0, \dots, n-1\}, E)$ be an *undirected graph* with no multiple or self edges, such that $n = |V|$, $m = |E|$. Let $c : V \to \mathbb{R}_{\geq 0}$ be a vertex-weight function, and let $\omega : E \to \mathbb{R}_{>0}$ be an edge-weight function. We generalize c and ω functions to sets, such that $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. Let $N(v) = \{u : \{v, u\} \in E\}$ denote the neighbors of v. A graph $S = (V', E')$ is said to be a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. When $E' = E \cap (V' \times V')$, S is an *induced* subgraph. Let $d(v)$ be the degree of vertex v and Δ be the maximum degree of G.

Partitioning. The *(hyper)graph partitioning* problem consists of assigning each vertex of a (hyper)graph to exactly one of k distinct *blocks* respecting a balancing constraint in order to minimize the weight of the (hyper)edges running between the blocks, i.e., the edge-cut

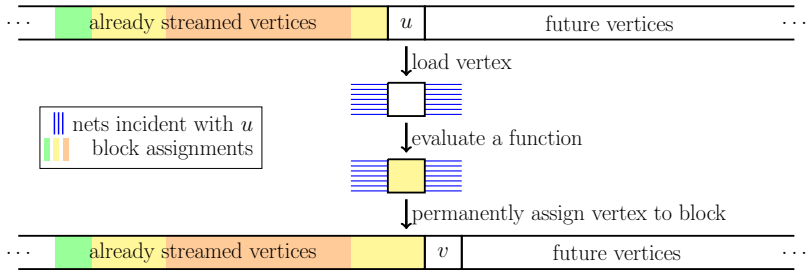


Figure 1 Typical layout of streaming algorithm for hypergraph partitioning.

(resp. cut-net). More precisely, it partitions V into k blocks $V_1,\ldots,V_k$ (i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$), which is called a *k-partition* of the (hyper)graph. The *edge-cut* (resp. *cut-net*) of a k-partition consists of the total weight of the *cut edges* (resp. *cut nets*), i.e., edges (resp. nets) crossing blocks. More formally, let the edge-cut (resp. cut-net) be $\sum_{i<j} \omega(E')$, in which $E' := \{e \in E, \exists \{u,v\} \subseteq e : u \in V_i, v \in V_j, i \neq j\}$ is the *cut-set* (i.e., the set of all cut nets). The *balancing constraint* demands that the sum of vertex weights in each block does not exceed a threshold associated with some allowed *imbalance* ϵ. More specifically, $\forall i \in \{1,\ldots,k\} : c(V_i) \leq L_{\max} := \lceil (1+\epsilon)\frac{c(V)}{k} \rceil$. For each net e of a hypergraph, $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e)$ of a net e is the cardinality of its connectivity set, i.e., $\lambda(e) := |\Lambda(e)|$. The so-called *connectivity* metric (λ-1) is computed as $\sum_{e \in E'} (\lambda(e) - 1)\ \omega(e)$, where E' is the cut-set.

Streaming. Streaming algorithms usually follow an iterative load-compute-store logic. Our focus and the most used streaming model is the *one-pass* model. In this model, vertices of a (hyper)graph are loaded one at a time alongside with their (hyper)edges, then some logic is applied to permanently assign them to blocks, as illustrated in Figure 1. A similar sequence of operations is used to partition a stream of edges of a graph on the fly. In this case, edges of a graph are loaded one at a time alongside with their end-points, then some logic is applied to permanently assign them to blocks. This logic can be as simple as a `Hashing` function or as complex as scoring all blocks based on some objective and then assigning the vertex to the block with highest score. There are other, more sophisticated, streaming models such as the sliding window [27] and the buffered streaming [20, 13], but are beyond the scope of this work.

2.2 Related Work

There is a huge body of research on (hyper)graph partitioning. The most prominent tools to partition (hyper)graphs in memory include `PaToH` [10], `Metis` [21], `hMetis` [22], `Scotch` [28], `HYPE` [24], `KaHIP` [30], `KaMinPar` [18], `KaHyPar` [31], `Mt-KaHyPar` [17], and `mt-KaHIP` [2]. The readers are referred to [11, 9, 33] for extensive material and references. Here, we focus on the results specifically related to the scope of this paper. In particular, we provide a detailed review for the following problems based on the one-pass streaming model: hypergraph partitioning and graph vertex partitioning.

Streaming Hypergraph Partitioning. Alistarh et al. [3] propose `Min-Max`, a one-pass streaming algorithm to assign the vertices of a hypergraph to blocks. For each block, this algorithm keeps track of nets which contain pins in it. This implies a memory consumption

of $O(mk)$. When a vertex is loaded, `Min-Max` allocates it to the block containing the largest intersection with its nets while respecting a hard constraint for load balance. The authors theoretically prove that their algorithm is able to recover a hidden *co-clustering* with high probability, where a co-clustering is defined as a simultaneous clustering of vertices and hyperedges. In the experimental evaluation, `Min-Max` outperforms five intuitive streaming approaches with respect to load imbalance, while producing solutions up to five times more imbalanced than internal-memory algorithms such as `hMetis`.

Taşyaran et al. [35] propose improved versions of the algorithm `Min-Max` [3]. The authors present `Min-Max-N2P`, a modified version of `Min-Max` that stores blocks containing each net's pins instead of storing nets per block, as done in `Min-Max`. In their experiments, `Min-Max-N2P` is three orders of magnitude faster than `Min-Max` while keeping the same cut-net. The authors also introduce three algorithms with reduced memory usage compared to `Min-Max`: `Min-Max-L`ℓ, a modification of `Min-Max-N2P` that employs an upper-bound ℓ to limit memory consumption per net, `Min-Max-BF` which utilizes Bloom filters for membership queries, and `Min-Max-MH` that uses hashing functions to replace the connectivity information between blocks and nets. In their experiments, their three algorithms reduce the running time in comparison to `Min-Max`, especially `Min-Max-L`ℓ and `Min-Max-MH`, which are up to four orders of magnitude faster. On the other hand, the three algorithms generate solutions with worse cut-net than `Min-Max`, especially `Min-Max-MH`, which increases the cut-net by up to an order of magnitude. Moreover, the authors propose a technique to improve the partitioning decision in the streaming setting by including a buffer to store some vertices and their net sets. This approach operates similarly to `Min-Max-N2P`, but with the added ability to revisit buffered vertices and adjust their partition assignment based on the connectivity metric. The authors propose three algorithms using this buffered approach: `REF` that buffers every incoming vertex but only reassigns those that may improve connectivity, `REF_RLX` that buffers all vertices and reassigns all vertices in the buffer, and `REF_RLX_SV` that only buffers vertices with small net sets and reassigns all vertices in the buffer. Their experimental results show that the use of buffered approaches leads to a 5-20% improvement in partitioning quality compared to non-buffered approaches, but with a trade-off of increased runtime.

Streaming Graph Vertex Partitioning. Stanton and Kliot [34] introduced graph partitioning in the streaming model and proposed some heuristics to solve it. Their most prominent heuristic include the one-pass methods `Hashing` and *linear deterministic greedy* (`LDG`). In their experiments, `LDG` had the best overall edge-cut. In this algorithm, vertex assignments prioritize blocks containing more neighbors and use a penalty multiplier to control imbalance. Particularly, a vertex v is assigned to the block V_i that maximizes $|V_i \cap N(v)| * \lambda(i)$ with $\lambda(i)$ being a multiplicative penalty defined as $(1 - \frac{|V_i|}{L_{\max}})$. The intuition is that the penalty avoids to overload blocks that are already very heavy. In case of ties on the objective function, `LDG` assigns the vertex to the block with fewer vertices. Overall, `LDG` partitions a graph in $O(m+nk)$ time. On the other hand, `Hashing` has running time $O(n)$ but produces a poor edge-cut.

Tsourakakis et al. [36] proposed `Fennel`, a one-pass partitioning heuristic based on the widely-known clustering objective *modularity* [7]. `Fennel` assigns a vertex v to a block V_i, respecting a balancing threshold, in order to maximize an expression of type $|V_i \cap N(v)| - f(|V_i|)$, i.e., with an additive penalty. This expression is an interpolation of two properties: attraction to blocks with many neighbors and repulsion from blocks with many non-neighbors. When $f(|V_i|)$ is a constant, the expression coincides with the first property. If $f(|V_i|) = |V_i|$, the expression coincides with the second property. In particular, the authors

defined the Fennel objective with $f(|V_i|) = \alpha * \gamma * |V_i|^{\gamma-1}$, in which γ is a free parameter and $\alpha = m\frac{k^{\gamma-1}}{n^{\gamma}}$. After a parameter tuning made by the authors, Fennel uses $\gamma = \frac{3}{2}$, which provides $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$. As LDG, Fennel partitions a graph in $O(m + nk)$ time.

Faraj and Schulz [14] propose a shared-memory streaming algorithm for vertex partitioning which performs recursive multi-sections on the fly. As a preliminary phase, their algorithm decomposes a k-way partitioning problem into a hierarchy containing $\lceil \log_b k \rceil$ layers of b-way partitioning subproblems. This hierarchy can either reflect the topology of a high performance system to solve a process mapping [15, 29] or be computed for an arbitrary k to solve a regular vertex partitioning. Then, an adapted version of Fennel is used to solve each of the subproblems in such a way that the whole k-partition is computed on the fly during a single pass over the graph. While producing an edge-cut around 5% lower than Fennel, their algorithm has theoretical complexity $O((m + nb) \log_b k)$ and experimentally ran up to two orders of magnitude faster than Fennel.

Besides the one-pass model, other streaming models have also been used to solve vertex partitioning. Restreaming graph partitioning has been introduced by Nishimura and Ugander [26]. In this model, multiple passes through the entire input are allowed, which enables iterative improvements. The authors proposed easily implementable restreaming versions of LDG and Fennel: ReLDG and ReFennel, respectively. Awadelkarim and Ugander [5] studied the effect of vertex ordering for streaming graph partitioning. The authors introduced the notion of *prioritized streaming*, in which (re)streamed vertices are statically or dynamically reordered based on some priority. The authors proposed a prioritized version of ReLDG Patwary et al. [27] proposed WStream, a greedy stream algorithm that keeps a sliding stream window. Jafari et al. [20] proposed a shared-memory multilevel algorithm based on a buffered streaming model. Their algorithm uses the one-pass algorithm LDG as the coarsening, initial partitioning, and the local search steps of their multilevel scheme. Faraj and Schulz [13] proposed HeiStream, a multilevel algorithm also based on a buffered streaming model. Their algorithm loads a chunk of vertices, builds a model, and then partitions this model with a traditional multilevel algorithm coupled with an extended version of the Fennel objective.

3 FREIGHT: Fast Streaming Hypergraph Partitioning

In this section, we provide a detailed explanation of our algorithmic contribution. First, we define our algorithm named FREIGHT. Next, we present the advantages and disadvantages of using two different formats for streaming hypergraphs and partitioning them using FREIGHT. Additionally, we explain how we have removed the dependency on k from the complexity of FREIGHT by implementing an efficient data structure for block sorting.

3.1 Mathematical Definition

In this section, we provide a mathematical definition for FREIGHT by expanding the idea of Fennel to the domain of hypergraphs. Recall that, assuming the vertices of a graph being streamed one-by-one, the Fennel algorithm assigns an incoming vertex v to a block V_d where d is computed as follows:

$$d = \underset{i,\ |V_i|<L_{\max}}{\operatorname{argmax}} \left\{ |V_i \cap N(v)| - \alpha * \gamma * |V_i|^{\gamma-1} \right\} \tag{1}$$

The term $-\alpha * \gamma * |V_i|^{\gamma-1}$, which penalizes block imbalance in Fennel, is directly used in FREIGHT without modification and with the same meaning. The term $|V_i \cap N(v)|$, which minimizes edge-cut in Fennel, needs to be adapted in FREIGHT to minimize the intended

metric, i.e., either cut-net or connectivity. Before explaining how this is adapted, recall that, in contrast to graph partitioning, in hypergraph partitioning the incident nets $I(v)$ of an incoming vertex v might contain nets that are already cut, i.e., with pins assigned to multiple blocks. The version of FREIGHT designed to optimize for *connectivity* accounts for already cut nets by keeping track of the block d_e to which the most recently streamed pin of each net e has been assigned. More formally, the connectivity version of FREIGHT assigns an incoming vertex v of a hypergraph to a block V_d with d given by Equation (2), where $I^i_{obj}(v) = I^i_{con}(v) = \{e \in I(v) : d_e = i\}$. On the other hand, the version of FREIGHT designed to optimize for *cut-net* ignores already cut nets, since their contribution to the overall cut-net of the hypergraph k-partition is fixed and cannot be changed anymore. More formally, the cut-net version of FREIGHT assigns an incoming vertex v of a hypergraph to a block V_d with d given by Equation (2), where $I^i_{obj}(v) = I^i_{cut}(v) = I^i_{con}(v) \setminus E'$ and E' is the set of already cut nets.

$$d = \underset{i,\ |V_i|<L_{\max}}{\operatorname{argmax}} \left\{ |I^i_{obj}(v)| - \alpha * \gamma * |V_i|^{\gamma-1} \right\} \tag{2}$$

Both configurations of FREIGHT interpolate two objectives: favoring blocks with many incident (uncut) nets and penalizing blocks with large cardinality. We briefly highlight that FREIGHT can be adapted for weighted hypergraphs. In particular, when dealing with weighted nets, the term $|I^i_{obj}(v)|$ is substituted by $\omega(I^i_{obj}(v))$. Likewise when dealing with weighted vertices, the term $-\alpha * \gamma * |V_i|^{\gamma-1}$ is substituted by $-c(v) * \alpha * \gamma * c(V_i)^{\gamma-1}$, where the weight $c(v)$ of v is used as a multiplicative factor in the penalty term.

3.2 Streaming Hypergraphs

In this section, we present and discuss the streaming model used by FREIGHT. Recall in the streaming model for graphs vertices are loaded one at a time alongside with their adjacency lists. Thus, just streaming the graph (without doing additional compuations, implies a time cost $O(m+n)$. In our model, the vertices of a hypergraph are loaded one at a time alongside with their incident nets, as illustrated in Figure 1. Our streaming model implies a time cost $O(\sum_{e \in E} |e| + n)$ just to stream the hypergraph, where $O(\sum_{e \in E} |e|)$ is the cost to stream each net e exactly $|e|$ times. FREIGHT uses $O(m+k)$ memory, with $O(m)$ being used to keep track, for each net e, of its cut/uncut status as well as the block d_e to which its most recently streamed pin was assigned. This net-tracking information, which substitutes the need to keep track of vertex assignments, is necessary for executing FREIGHT. Although FREIGHT consumes more memory than required by graph-based streaming algorithms which often use $O(n+k)$ memory, it is still far better than the $O(mk)$ worst-case memory required by the state-of-the-art algorithms for streaming hypergraph partitioning [3, 35], all of which are also based on a computational model that implies a time cost $O(\sum_{e \in E} |e| + n)$ just to stream the hypergraph.

3.3 Efficient Implementation

In this section, we describe an efficient implementation for FREIGHT. Recall that, for every vertex v that is loaded, FREIGHT uses Equation (2) to find the block with the highest score among up to k options. A simple method to accomplish this task consists of explicitly evaluating the score for each block and identifying the one with the highest score. This results in a total of $O(nk)$ evaluations, leading to an overall complexity of $O(\sum_{e \in E} |e| + nk)$. We propose an implementation that is significantly more efficient than this approach.

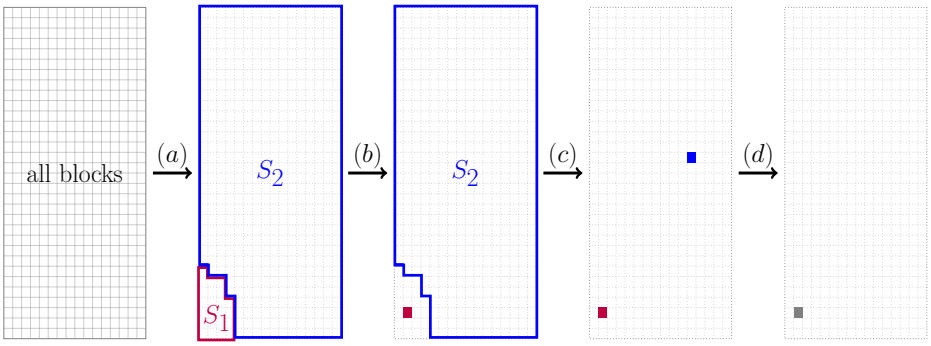


Figure 2 Illustration of the process to solve Equation (2) for an incoming vertex u with $k = 512$ blocks. (a) The k blocks are decomposed into S_1 and S_2, with $|S_1| = O(|I(u)|)$. (b) Equation (3) is explicitly solved at cost $O(|I(u)|)$. (c) Equation (4) is implicitly solved at cost $O(1)$. (d) Both solutions are then evaluated using their `FREIGHT` scores to determine the final solution for Equation (2).

For each loaded vertex v, our implementation separates the blocks V_i for which $|V_i| < L_{\max}$ into two disjoint sets, S_1 and S_2. In particular, the set S_1 comprises blocks V_i where $|I^i_{obj}(v)| > 0$, while the set S_2 comprises the remaining blocks, i.e., blocks V_i for which $|I^i_{obj}(v)| = 0$. Using the sets provided, we break down Equation (2) into Equation (3) and Equation (4), which are solved separately. The resulting solutions are compared based on their `FREIGHT` scores to ultimately find the solution for Equation (2). The overall process is illustrated in Figure 2.

$$d = \operatorname*{argmax}_{i \in S_1} \left\{ |I^i_{obj}(v)| - \alpha * \gamma * |V_i|^{\gamma - 1} \right\} \tag{3}$$

$$d = \operatorname*{argmax}_{i \in S_2} \left\{ |I^i_{obj}(v)| - \alpha * \gamma * |V_i|^{\gamma - 1} \right\} = \operatorname*{argmin}_{i \in S_2} |V_i| \tag{4}$$

Now we explain how we solve Equation (3) and Equation (4). To solve Equation (3), we use the theoretical complexity outlined in Theorem 1 and solve it explicitly. In contrast, Equation (4) is implicitly solved by identifying the block with minimal cardinality. We use an efficient data structure to keep all blocks sorted by cardinality throughout the entire execution, which enables us to solve Equation (4) in constant time.

▶ **Theorem 1.** *Equation (3) can be solved in time $O(|I(v)|)$.*

Proof. The terms $|I^i_{obj}(v)|$ in Equation (3) can be computed by iterating through the nets of v at a cost of $O(|I(v)|)$ and determining their status as cut, unassigned, or assigned to a block. The calculation of the factors $-\alpha * \gamma * |V_i|^{\gamma-1}$ in Equation (3) can be done in time $O(|S_1|) = O(|I(v)|)$, thus completing the proof. ◀

Now we explain our data structure to keep the blocks sorted by cardinality during the whole algorithm execution. The data structure is implemented with two arrays A and B, both with k elements, and a list L. The array A stores all k blocks always in ascending order. The array B maps the index i of a block V_i to its position in A. Each element in the list L represents a bucket. Each bucket is associated with a unique block cardinality and contains the leftmost and the rightmost positions ℓ and r of the range of blocks in A which currently

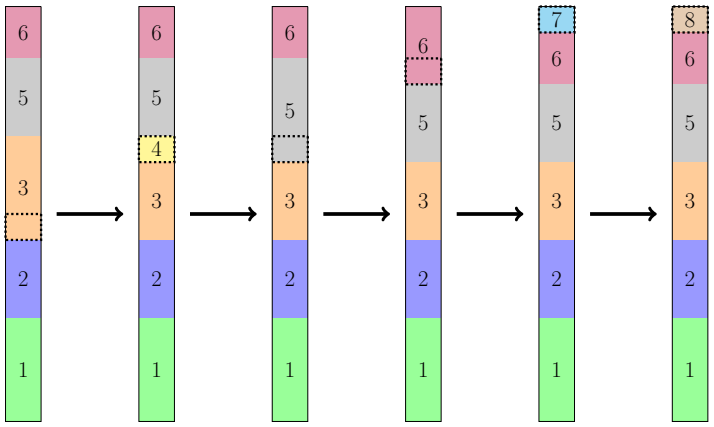


Figure 3 Illustration of our data structure used to keep the blocks sorted by cardinality throughout the execution of `FREIGHT`. The array A is represented as a vertical rectangle. Each region of A is covered by a unique bucket, which is represented by a unique color filling the corresponding region in A. The cardinality associated with each bucket is written in the middle of the region of A covered by it. Here we represent the behavior of the data structure when assigning vertices to the block surrounded by a dotted rectangle five times consecutively.

have this cardinality. Reciprocally, each block in A has a pointer to the unique bucket in L corresponding to its cardinality. To begin the algorithm, L is set up with a single bucket for cardinality 0 which covers the k positions of A, i.e., its paramenters ℓ and r are 1 and k, respectively. The blocks in A are sorted in any order initially, however, as each block starts with a cardinality of 0, they will be ordered by their cardinalities.

When a vertex is assigned to a block V_d, we update our data structure as detailed in Algorithm 1 and exemplified in Figure 3. We describe Algorithm 1 in detail now. In line 1, we find the position p of V_d in A and find the bucket C associated with it. In line 2, we exchange the content of two positions in A: the position where V_d is located and the position identified by the variable r in C, which marks the rightmost block in A covered by C. This variable r is afterwards decremented in line 3 since V_d is now not covered anymore by the bucket C. In lines 4 and 5, we check if the new (increased) cardinality of V_d matches the cardinality of the block located right after it in A. If so, we associate V_d to the same bucket as it and decrement this bucket's leftmost position ℓ in line 6; Otherwise, we push a new bucket to L and match it to V_d adequately in lines 8 and 9. Finally, in line 10, we delete C in case its range $[\ell, r]$ is empty. Figure 3 shows our data structure through five consecutive executions of Algorithm 1. Theorem 2 proves the correctness of our data structure. Theorem 3 shows that, using our proposed data structure, we need time $O(1)$ to either solve Equation (4) or prove that the solution for Equation (3) solves Equation (2). Note that our data structure can only handle unweighted vertices. In case of weighted vertices, a bucket queue can be used instead of our data structure, resulting in the same overall complexity and requiring $O(k + L_{\max})$ memory, while our data structure only requires $O(k)$ memory. The overall complexity of `FREIGHT`, which directly follows from Theorem 1 and Theorem 3, is expressed in Corollary 4.

▶ **Theorem 2.** *Our proposed data structure keeps the blocks within array A consistently sorted in ascending order of cardinality.*

Proof. We inductively prove two claims at the same time: (a) the variables ℓ and r contained in each bucket from L respectively store the leftmost and the rightmost positions of the unique range of blocks in A which currently have this cardinality; (b) the array A contains

Algorithm 1 Increment cardinality of block V_d in the proposed data structure.

```
1: p ← B_d; C ← A_p.bucket;
2: q ← C.r; c ← A_q.id; Swap(A_p, A_q); Swap(B_c, B_d);
3: C.r ← C.r − 1;
4: C′ ← A_{q+1}.bucket;
5: if C.cardinality + 1 = C′.cardinality then
6:     A_q.bucket ← C′; C′.ℓ ← C′.ℓ − 1;
7: else
8:     C″ ← NewBucket(); A_q.bucket ← C″; L ← L ∪ {C″};
9:     C″.cardinality ← C.cardinality + 1; C″.ℓ ← q; C″.r ← q;
10: if C.r = C.ℓ then L ← L \ {C};
```

the blocks sorted in ascending order of cardinality. Both claims are trivially true at the beginning, since all blocks have cardinality 0 and L is initialized with a single bucket with $\ell = 1$ and $r = k$. Now assuming that (a) and (b) are true at some point, we show that they keep being true after Algorithm 1 is executed. Note that line 2 performs the only position exchange in A throughout the whole algorithm. As (a) is assumed, it is the case that V_d swaps positions with the rightmost block in A containing the same cardinality of V_d. Since the cardinality of V_d will be incremented by one and all blocks have integer cardinalities, this concludes the proof of (b). To prove that (a) remains true, note that the only buckets in L that are modified are C (line 3), C' (line 6), and C'' (line 9). Claim (a) remains true for C because V_d, whose cardinality will be incremented, is the only block removed from its range. Claim (a) remains true for C' because line 6 is only executed if the new cardinality of V_d equals the cardinality of C', whose current range starts right after the new position of V_d in A. Bucket C'' is only created if the new cardinality of V_d is respectively larger and smaller than the cardinalities of C and C'. Since (b) is true, then this condition only happens if there is no block in A with the same cardinality as the new cardinality of V_d. Hence, claim (a) remains true for C'', which is created covering only the position of V_d in A. ◀

▶ **Theorem 3.** *By utilizing our proposed data structure, solving Equation (4) or demonstrating that any solution for Equation (3) is also a solution for Equation (2) can be accomplished in $O(1)$ time.*

Proof. Algorithm 1 contains no loops and each command in it has a complexity of $O(1)$, thus the total cost of the algorithm is $O(1)$. Our data structure executes Algorithm 1 once for each assigned vertex, hence it costs $O(1)$ per vertex. Say we are evaluating an incoming vertex v. According to Theorem 2, the block V_d with minimum cardinality is stored in the first position of the array A, hence it can be accessed in time $O(1)$. In case $V_d \in S_2$, then d is a solution for Equation (4). On the other hand, if V_d is in S_1, the `FREIGHT` score of V_d will be larger than the `FREIGHT` score of the solution for Equation (4) by at least $|I_d(v)| > 0$. In this case, it follows that any solution for Equation (3) solves Equation (2). ◀

▶ **Corollary 4.** *The overall complexity of `FREIGHT` is $O\big(\sum_{e \in E} |e| + n\big)$.*

4 Experimental Evaluation

Setup. We performed our implementations in C++ and compiled them using gcc 11.2 with full optimization turned on (-O3 flag). Unless mentioned otherwise, all experiments are performed on a single core of a machine consisting of a sixteen-core Intel Xeon Silver 4216

processor running at 2.1 GHz, 100 GB of main memory, 16 MB of L2-Cache, and 22 MB of L3-Cache running Ubuntu 20.04.1. The machine can handle 32 threads with hyperthreading. Unless otherwise mentioned we stream (hyper)graphs directly from the internal memory to obtain clear running time comparisons. However, note that `FREIGHT` as well as most of the other used algorithms can also be run streaming the hypergraphs from hard disk.

Baselines. We compare `FREIGHT` against various state-of-the-art algorithms. In this section we will list these algorithms and explain our criteria for algorithm selection. We have implemented `Hashing` in C++, since it is a simple algorithm. It basically consists of hashing the IDs of incoming vertices into $\{1, \ldots, k\}$. The remaining algorithms were obtained either from official repositories or privately from the authors, with the exception of `Min-Max`, for which there is no official implementation available. Here, we use the `Min-Max` implementations by Taşyaran et al. [35]. All algorithms were compiled with gcc 11.2.

We run `Hashing`, `Min-Max` [3] and all its improved versions proposed by Taşyaran et al. [35]: `Min-Max-BF`, `Min-Max-N2P`, `Min-Max-L`ℓ, `Min-Max-MH`, `REF`, `REF_RLX`, and `REF_RLX_SV`. (see Section 2.2 for details on the different `Min-Max` versions), `HYPE` [24], and `PaToH` v3.3 [10]. `Hashing` is relevant because it is the simplest and fastest streaming algorithm, which gives us a lower bound for partitioning time. `Min-Max` is a current state-of-the-art for streaming hypergraph partitioning in terms of cut-net and connectivity. The improved and buffered versions of `Min-Max` proposed in [35] are relevant because some of them are orders of magnitude faster than `Min-Max` while others produce improved partitions in comparison to it. `HYPE` and `PaToH` are in-memory algorithms for hypergraph partitioning, hence they are not suitable for the streaming setting. However, we compare against them because `HYPE` is among the fastest in-memory algorithms while `PaToH` is very fast and also computes partitions with very good cut-net and connectivity. Note that KaHyPar [31] is the leading tool with respect to solution quality, however it is also much slower than `PaToH`.

Instances. We selected hypergraphs from various sources to test our algorithm. The considered hypergraphs were used for benchmark in previous works on hypergraph partitioning. Prior to each experiment, we converted all hypergraphs to the appropriate streaming formats required by each algorithm. We removed parallel and empty hyperedges and self loops, and assigned unitary weight to all vertices and hyperedges. In all experiments with streaming algorithms, we stream the hypergraphs with the natural given order of the vertices. We use a number of blocks $k \in \{512, 1024, 1536, 2048, 2560\}$ unless mentioned otherwise. We allow a fixed imbalance of 3% for all experiments (and all algorithms) since this is a frequently used value in the partitioning literature. All algorithms always generated balanced partitions, except for `HYPE` which generated highly unbalanced partitions in around 5% of its experiments.

We use the same benchmark as in [31]. This consists of 310 hypergraphs from three benchmark sets: 18 hypergraphs from the ISPD98 Circuit Benchmark Suite [4], 192 hypergraphs based on the University of Florida Sparse Matrix Collection [12], and 100 instances from the international SAT Competition 2014 [6]. The SAT instances were converted into hypergraphs by mapping each boolean variable and its complement to a vertex and each clause to a net. From the Sparse Matrix Collection, one matrix was selected for each application area that had between 10 000 and 10 000 000 columns. The matrices were converted into hypergraphs using the row-net model, in which each row is treated as a net and each column as a vertex.

Methodology. Depending on the focus of the experiment, we measure running time, cut-net, and-or connectivity. We perform 5 repetitions per algorithm and instance using random seeds for non-deterministic algorithms, and calculate the arithmetic average

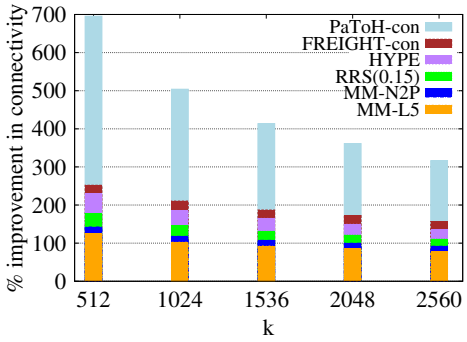
of the computed objective function and running time per instance. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*.

Given a result of an algorithm A, we express its value σ_A (which can be objective or running time) as *improvement* over an algorithm B, computed as $\left(\frac{\sigma_B}{\sigma_A} - 1\right) * 100\%$; We also use *performance profiles* to represent results. They relate the running time (quality) of a group of algorithms to the fastest (best) one on a per-instance basis (rather than grouped by k). The x-axis shows a factor τ while the y-axis shows the percentage of instances for which A has up to τ times the running time (quality) of the fastest (best) algorithm. Bar charts and boxplots are also employed to represent our findings. We use bar charts to visualize the average value of an objective function in relation to k, where each algorithm is represented by vertical bars of a given color with origin on the x-axis. The bars for every value of k have a common origin and are arranged in terms of their height, allowing all heights to be visible. We use boxplots to give a clear picture of the dataset distribution by displaying the minimum, maximum, median, first and third quartiles, while disregarding outliers.

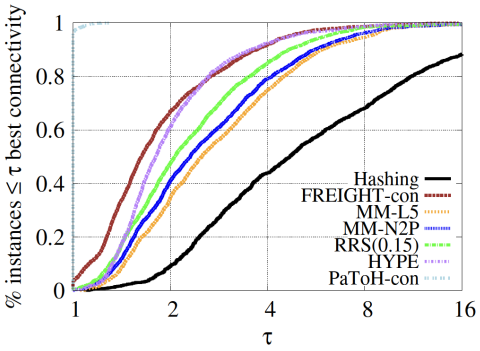
4.1 Results

In this section, we show experiments in which we compare `FREIGHT` against the current state-of-the-art of streaming hypergraph partitioning. As already mentioned, we also use two internal-memory algorithms [24, 10] as more general baselines for comparison. We focus our experimental evaluation on the comparison of solution quality and running time. Observe that `PaToH` and `FREIGHT` have distinct versions designed to optimize for each quality metric (i.e., connectivity and cut-net). For a meaningful comparison, we only take into account the relevant version when dealing with each quality metric, however, both versions are still considered for running time comparisons. To differentiate between the versions, suffixes `-con` and `-cut` are added to represent the connectivity-optimized and cut-net versions respectively. For clarity, we refrain from discussing state-of-the-art streaming algorithms that are *dominated* by another algorithm. We define a dominated algorithm as one that has worse running time compared to another without offering a superior solution quality in return, or vice-versa. In particular, we leave out `Min-Max` and `Min-Max-BF` since they are dominated by `Min-Max-N2P`, which is referred to as `MM-N2P` hereafter. Similarly, we omit `Min-Max-MH` because it is dominated by `Hashing`. We use a buffer size of 15% for testing the buffered algorithms `REF`, `REF_RLX`, and `REF_RLX_SV`, following the best results outlined in [35]. We omit the first two of them since they are dominated by the latter one, which is referred to as `RRS(0.15)` from now on. Since `Min-Max-L`ℓ is not dominated by any other algorithm, we exhibit its results with $\ell = 5$, as seen in the best results in [35], and we refer to it as `MM-L5` from this point.

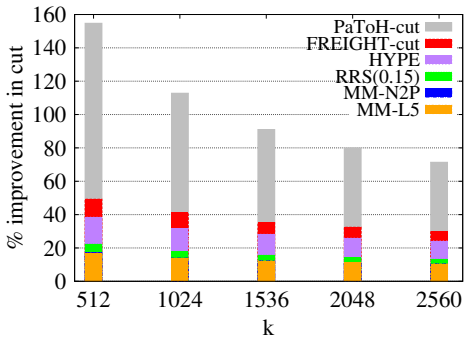
Connectivity. We start by looking at the connectivity metric. In Figure 4a, we plot the average connectivity improvement over `Hashing` for each value of k. `PaToH-con` produces the best connectivity on average, yielding an average improvement of 443% when compared to `Hashing`. This is in line with previous works in the area of (hyper)graph partitioning, i.e. streaming algorithms typically compute worse solutions than internal memory algorithms, which have access to the whole graph. `FREIGHT-con` is found to be the second best algorithm in terms of connectivity, outperforming both the internal memory algorithm `HYPE` and the buffered streaming algorithm `RRS(0.15)`. On average, these three algorithms improve 194%, 171%, and 136% over `Hashing`, respectively. Finally, `MM-N2P` and `MM-L5` compute solutions which improve 111% and 96% over `Hashing` on average, respectively. In direct



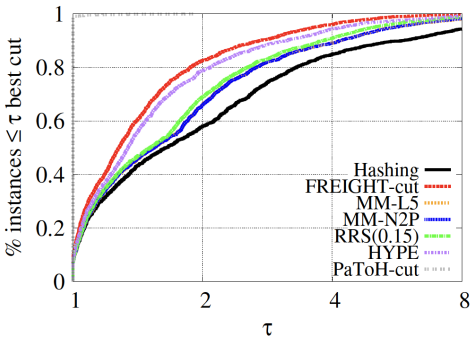
(a) Connectivity improvement plot over `Hashing`.



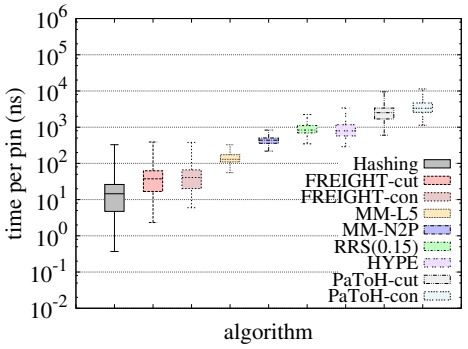
(b) Connectivity performance profile.



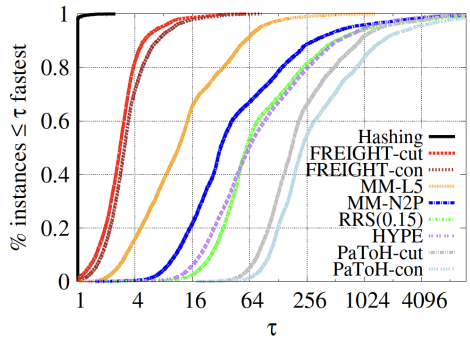
(c) Cut-net improvement plot over `Hashing`.



(d) Cut-net performance profile.



(e) Running time boxplots.



(f) Running time performance profile.

Figure 4 Comparison against the state-of-the-art streaming algorithms for hypergraph partitioning. We show performance profiles, improvement plots over `Hashing`, and boxplots. Note that `PaToH-con`, `PaToH-cut`, and `Hashing` align almost perfectly with the y-axis in Figures 4b, 4d, and 4f, respectively. Also the curves and bars of `MM-N2P` and `MM-L5` roughly overlap with one another in Figure 4d and Figure 4c.

comparison, FREIGHT-con shows average connectivity improvements of 8%, 24%, 39%, and 50% over HYPE, RRS(0.15), MM-N2P, and MM-L5, respectively. Note that each algorithm retains its relative ranking in terms of average connectivity over all values of k.

In Figure 4b, we plot connectivity performance profiles across all experiments. PaToH-con produces the best overall connectivity for 96.4% of the instances, while FREIGHT-con produces the best connectivity for 3.1% of the instances and no other algorithm computes the best connectivity for more than 0.35% of the instances. The connectivity produced by FREIGHT-con, HYPE, RRS(0.15), MM-N2P, MM-L5, and Hashing are within a factor 2 of the best found connectivity for 67%, 61%, 47%, 41%, 34%, and 9% of the instances, respectively. In summary, FREIGHT-con produces the best connectivity among (buffered) streaming competitors, outperforming even in-memory algorithm HYPE.

Cut-Net. Next we examine at the cut-net metric. In Figure 4c, we plot the cut-net improvement over Hashing. PaToH-cut produces the best overall cut-net, with an average improvement of 100% compared to Hashing. FREIGHT-cut is found to be the second best algorithm with respect to cut-net, superior to internal-memory algorithm HYPE and buffered streaming algorithm RRS(0.15). These three algorithms improve connectivity over Hashing by 37%, 30%, and 17% respectively. Finally, both MM-N2P and MM-L5 improve connectivity by 13% on average over Hashing. In direct comparison, FREIGHT-cut shows average connectivity improvements of 6%, 18%, 22%, and 22% over HYPE, RRS(0.15), MM-N2P, and MM-L5, respectively. Each algorithm preserves its relative ranking in average cut-net across all values of k.

In Figure 4d, we plot cut-net performance profiles across all experiments. In the plot, PaToH-cut produces the best overall connectivity for 98.0% of the instances, while FREIGHT-cut and HYPE produce the best cut-net for 6.8% and 5.2% of the instances and all other streaming algorithms (RRS(0.15), MM-N2P, MM-L5, and Hashing) produce the best cut-net for 4.8% of the instances. The cut-net results produced by FREIGHT-cut, HYPE, RRS(0.15), MM-N2P, MM-L5, and Hashing are within a factor 2 of the best found cut-net for 83%, 79%, 69%, 66%, 66%, and 58% of the instances, respectively. This shows that FREIGHT-cut produces the best cut-net among all (buffered) streaming competitors and even beats the in-memory algorithm HYPE.

Running Time. Now we compare the algorithms' runtime. Boxes and whiskers in Figure 4e display the distribution of the running time per pin, measured in nanoseconds, for all instances. Hashing, FREIGHT-cut, and FREIGHT-con are the three fastest algorithms, with median runtimes per pin of 15ns, 38ns, and 41ns, respectively. MM-L5, MM-N2P, HYPE, and RRS(0.15) follow with median runtimes per pin of 130ns, 437ns, 792ns, and 833ns, respectively. Lastly, the algorithms with the highest median runtime per pin are PaToH-cut and PaToH-con, with 2 516ns and 3 333ns respectively. The measured runtime per pin for both HYPE and PaToH align with values reported in prior research [32].

In Figure 4f, we show running time performance profiles. Hashing is the fastest algorithm for 98.3% of the instances, while FREIGHT-cut is the fastest one for 1.2% of the instances and no other algorithm is the fastest one for more than 0.4% of the instances. The running time of FREIGHT-cut and FREIGHT-con is within a factor 4 of that of Hashing for 82% and 72% of instances, respectively. In contrast, for only 16% of instances does this occur for MM-L5, and for less than 0.4% of instances for all other algorithms. The close running times of FREIGHT to Hashing are surprising given FREIGHT's superior solution quality compared to Hashing and all other streaming algorithms and even HYPE.

Further Comparisons. For graph vertex partitioning FREIGHT and Fennel are mathematically equivalent. However, FREIGHT exhibits a lower computational complexity of $O(m+n)$ compared to the standard implementation of Fennel, which has a complexity of $O(m+nk)$ due to evaluating all blocks for each node. To optimize its performance for this use case, we have implemented an optimized version of FREIGHT with a memory consumption of $O(n+k)$, matching that of Fennel. In our experiments, we utilized the same graphs as in [14] and tested with $k \in \{512, 1024, 1536, 2048, 2560\}$. On average, FREIGHT proves to be 109 times faster than the standard implementation of Fennel. Moreover, the performance gap is found to increase as the value of k grow, with FREIGHT reaching up to 261 times faster than Fennel in some instances.

5 Conclusion

In this work, we introduce FREIGHT, a highly efficient and effective streaming algorithm for hypergraph partitioning. Our algorithm leverages an optimized data structure, resulting in linear running time with respect to pin-count and linear memory consumption in relation to the numbers of nets and blocks. The results of our extensive experimentation demonstrate that the running time of FREIGHT is competitive with the Hashing algorithm, with a maximum difference of a factor of four observed in three fourths of the instances. Importantly, our findings indicate that FREIGHT consistently outperforms all existing (buffered) streaming algorithms and even the in-memory algorithm HYPE, with regards to both cut-net and connectivity measures. This underscores the significance of our proposed algorithm as a highly efficient and effective solution for hypergraph partitioning in the context of large-scale and dynamic data processing.

References

1 Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: An experimental study. *Proc. VLDB Endow.*, 11(11):1590–1603, 2018. doi:10.14778/3236187.3236208.

2 Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-quality shared-memory graph partitioning. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, volume 11014 of *Lecture Notes in Computer Science*, pages 659–671. Springer, 2018. doi:10.1007/978-3-319-96983-1_47.

3 Dan Alistarh, Jennifer Iglesias, and Milan Vojnovic. Streaming min-max hypergraph partitioning. In *Advances in Neural Information Processing Systems*, pages 1900–1908, 2015. doi:10.5555/2969442.2969452.

4 Charles J. Alpert. The ISPD98 circuit benchmark suite. In Majid Sarrafzadeh, editor, *Proceedings of the 1998 International Symposium on Physical Design, ISPD 1998, Monterey, CA, USA, April 6-8, 1998*, pages 80–85. ACM, 1998. doi:10.1145/274535.274546.

5 Amel Awadelkarim and Johan Ugander. Prioritized restreaming algorithms for balanced graph partitioning. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 1877–1887. ACM, 2020. doi:10.1145/3394486.3403239.

6 Anton Belov, Daiel Diepold, Marijn Heule, and Matti Järvisalo. The sat competition 2014. http://www.satcompetition.org/2014/, 2014.

7 Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE transactions on knowledge and data engineering*, 20(2):172–188, 2007. doi:10.1109/TKDE.2007.190689.

8 Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Inf. Process. Lett.*, 42(3):153–159, 1992. doi:10.1016/0020-0190(92)90140-Q.

9 Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-49487-6_4.

10 Ümit V. Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011. doi:10.1007/978-0-387-09766-4_93.

11 Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *ACM Computing Surveys*, 2023. doi:doi.org/10.1145/3571808.

12 Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011. doi:10.1145/2049662.2049663.

13 Marcelo Fonseca Faraj and Christian Schulz. Buffered streaming graph partitioning. *ACM J. Exp. Algorithmics*, 27, October 2022. doi:10.1145/3546911.

14 Marcelo Fonseca Faraj and Christian Schulz. Recursive multi-section on the fly: Shared-memory streaming algorithms for hierarchical graph partitioning and process mapping. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 473–483, 2022. doi:10.1109/CLUSTER51413.2022.00057.

15 Marcelo Fonseca Faraj, Alexander van der Grinten, Henning Meyerhenke, Jesper Larsson Träff, and Christian Schulz. High-quality hierarchical process mapping. In *18th International Symposium on Experimental Algorithms, SEA*, volume 160 of *LIPIcs*, pages 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SEA.2020.4.

16 Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 47–63. ACM, 1974. doi:10.1145/800119.803884.

17 Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable Shared-Memory Hypergraph Partitioning. In *Proceedings of the Symposium on Algorithm Engineering and Experiments ALENEX*, pages 16–30, 2021. doi:10.1137/1.9781611976472.2.

18 Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep multilevel graph partitioning. In *29th Annual European Symposium on Algorithms, ESA*, volume 204 of *LIPIcs*, pages 48:1–48:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.48.

19 Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 439–450. IEEE, 2019. doi:10.1109/IPDPS.2019.00054.

20 Nazanin Jafari, Oguz Selvitopi, and Cevdet Aykanat. Fast shared-memory streaming multilevel graph partitioning. *Journal of Parallel and Distributed Computing*, 147:140–151, 2021. doi:10.1016/j.jpdc.2020.09.004.

21 George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 35. IEEE Computer Society, 1996. doi:10.1109/SC.1996.32.

22 George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In Mary Jane Irwin, editor, *Proceedings of the 36th Conference on Design Automation*, pages 343–348. ACM Press, 1999. doi:10.1145/309847.309954.

23 Renaud Lambiotte, Martin Rosvall, and Ingo Scholtes. From networks to optimal higher-order models of complex systems. *Nature physics*, 15(4):313–320, 2019. doi:10.1038/s41567-019-0459-y.

24 Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. HYPE: massive hypergraph partitioning with neighborhood expansion. In *IEEE International Conference on Big Data (IEEE BigData)*, pages 458–467. IEEE, 2018. doi:10.1109/BigData.2018.8621968.

25 Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 685–695. IEEE, 2018. doi:10.1109/ICDCS.2018.00072.

26 Joel Nishimura and Johan Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1106–1114, 2013. doi:10.1145/2487575.2487696.

27 Md Anwarul Kaium Patwary, Saurabh Kumar Garg, and Byeong Kang. Window-based streaming graph partitioning algorithm. In *Proceedings of the Australasian Computer Science Week Multiconference, ACSW*, pages 51:1–51:10. ACM, 2019. doi:10.1145/3290688.3290711.

28 François Pellegrini and Jean Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical report, TR 1038-96, LaBRI, 1996. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=94b913363b57e019b8a32529b076a8d4181587ac.

29 Maria Predari, Charilaos Tzovas, Christian Schulz, and Henning Meyerhenke. An mpi-based algorithm for mapping complex networks onto hierarchical architectures. In *Euro-Par 2021: Parallel Processing - 27th International Conference on Parallel and Distributed Computing*, volume 12820 of *LNCS*, pages 167–182. Springer, 2021. doi:10.1007/978-3-030-85665-6_11.

30 Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013. doi:10.1007/978-3-642-38527-8_16.

31 Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 53–67. SIAM, 2016. doi:10.1137/1.9781611974317.5.

32 Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithms (JEA)*, 2022. doi:10.1145/3529090.

33 Christian Schulz and Darren Strash. Graph partitioning: Formulations and applications to big data. In *Encyclopedia of Big Data Technologies*. Springer, 2019. doi:10.1007/978-3-319-63962-8_312-2.

34 Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2012. doi:10.1145/2339530.2339722.

35 Fatih Taşyaran, Berkay Demireller, Kamer Kaya, and Bora Uçar. Streaming Hypergraph Partitioning Algorithms on Limited Memory Environments. In *HPCS 2020 - International Conference on High Performance Computing & Simulation*, pages 1–8. IEEE, 2021. URL: https://hal.archives-ouvertes.fr/hal-03182122.

36 Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014. doi:10.1145/2556195.2556213.