Experimental Evaluation of Algorithms for Streaming Hypergraph Partitioning

Kamal Eyubov

September 5, 2022

3574248

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Marcelo Fonseca Faraj

Acknowledgments

It is an honor for me to write my thesis in a field of graph-related algorithms, in which I have a special interest, and to approach questions and problems in it from angles that have never been tried before, being at the forefront of the research. I would like to thank Prof. Dr. Christian Schulz for this opportunity and for making me acquainted with the problems I would set out to tackle. I thank the research group Algorithm Engineering in general for providing me access to their computational resources. I would like to express special gratitude to Marcelo Fonseca Faraj who helped me with my thesis and always answered my questions regarding the direction of my work and what I should be paying attention to. This work has been challenging, as, at times, there were lots of details in it that required attention and new problems arose. My family and friends provided me with a great deal of moral support during the process. I warmly thank my family: my grandmother, Marsella, my father, Amirkhan, my mother, Nargiz, and my sister, Nigar. I also thank my friends who have been aware of the aforementioned challenges and have been supportive.
Last but not least, I would like to express my gratitude for having known the late Dr. Ramin Mahmudzade who introduced me to computer science, and under whose tutelage I developed a great interest in algorithms in graph theory.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidelberg, September 5, 2022

Kamal Eyubov

Abstract

Partitioning hypergraphs becomes an increasingly difficult task with real-world hypergraphs growing in size. Streaming algorithms are a current trend to tackle increasingly large hypergraphs in a reasonable time. Nevertheless, little effort has yet been dedicated to streaming hypergraph partitioning. In our previous work, we proposed a streaming weighted hypergraph partitioning algorithm. More specifically, we adapted a successful streaming partitioning algorithm for the domain of hypergraphs. We carefully engineered all its details in order to optimize its performance in practice and experimentally tuned our algorithm and show that it produces partitions with approximately 15% lower cut-net on average than the current state-of-the-art streaming hypergraph partitioning algorithm. In this work, we move on to introduce the usage of sampling methods in order to speed up the algorithm even further without causing a significant solution quality loss. We show that a constant running time can be reached regardless of the number of partition blocks. Additionally, we propose a buffered approach for hypergraph partitioning in order to obtain even higher quality solutions while utilizing controllable amounts of operative memory. We demonstrate that it can at least partially achieve intended quality and performance goals. Additionally, we experiment with utilizing our algorithm for graph edge partitioning, as it can be shown to be equivalent to hypergraph vertex partitioning.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Hypergraphs are a generalization of graphs, where a single (hyper)edge called *net* can connect an arbitrary amount of vertices. This kind of data structure makes it possible to represent complex abstract and real-life systems that include but are not restricted to the following ones: Boolean formulas, integrated circuits, objects from numerical linear algebra [28] as well as social networks. In real-life situations, hypergraphs might become too large to be processed on a single machine. An algorithm is therefore needed in order to distribute/partition these large hypergraphs over multiple machines in order to enable parallel processing. This is modeled by the hypergraph partitioning problem, where the vertices of a hypergraph are partitioned in roughly equal parts among machines while minimizing the nets running between those machines. Aforementioned machines are called *partitions*, *clusters* or *blocks* in different works depending on context. In this thesis, we call them *blocks* in order to avoid confusion with the concept of clusters in multilevel graph partitioning algorithms (for example, HeiStream [13]).
The (hyper)graph partitioning task can be shown to be NP-complete when specific objectives such as cut-net minimization are defined. [16] It should also be noted that the (hyper)graph partitioning task is not approximable either, meaning there is no constant factor acting as a bound for solution quality. [6]
In order to address this issue, a set of previously proposed algorithms seek to compute fairly good solutions in a reasonable amount of time. Approaches mostly include multilevel methods such as KaHIP [32] for graphs, KaHyPar [34] and hMETIS [20] for hypergraphs. Those methods usually take the whole (hyper)graph, reduce its size by contracting vertices, apply some initial partitioning on the most reduced level, and gradually refine the blocks while uncontracting the vertices until the input (hyper)graph is restored. The main issue with this family of algorithms is that they require the whole input to have been loaded into the memory. As a result, these algorithms require high amounts of computation resources

like memory in order to perform partitioning. Another issue is the applicability of these algorithms to cases when input is not entirely available at the start of the partitioning, and instead can only be gradually received or *streamed* throughout the process for different reasons.
With the proliferation of huge real-world (hyper)graphs comes the need for partitioning them efficiently using low computational resources. Diverse streaming partitioning algorithms have been proposed to tackle this need within the domain of graphs. In contrast to this, only a limited amount of work has explored algorithms to partition hypergraphs within the streaming model [36, 1, 23]. In my 2021 programming practical, we extended the widely-known streaming partitioning algorithm Fennel for the domain of weighted hypergraphs. Namely, we proposed an adaptation of the algorithm Fennel [38] to partition weighted hypergraphs. We engineered all the details of this adapted algorithm and evaluated it experimentally afterward. To the best of our knowledge, ours was the first streaming algorithm capable of partitioning weighted hypergraphs. [12] Details are provided in Section 3.2.4.

1.2 Our Contribution

In our work, we introduce sampling in order to linearly improve the running time of hFennel [12], our previously proposed streaming hypergraph partitioning algorithm while not causing significant solution quality losses. We experiment with different sampling approaches and parameters in order to determine the one with the least overall quality reduction. Upon testing the algorithm with sampling, we determine that usage of sampling can significantly decrease the running time while preserving the solution quality to some degree. In fact, using a constant amount of sample blocks results in a constant running time over different numbers of blocks.
We also propose a buffered streaming hypergraph partitioning algorithm in which whole chunks of a hypergraph are loaded into a buffer of variable size to be partitioned. A multilevel algorithm is used in order to partition individual chunks. This algorithm aims to provide a trade-off between a high solution quality of multilevel algorithms and low resource usage of the non-buffered streamed algorithm. The algorithm fulfills the aforementioned expectations for large buffer sizes by yielding solutions of higher quality than non-buffered streaming algorithms without having to load whole graphs into the memory.
Since the graph edge partitioning can be viewed as a variation of the hypergraph partitioning problem, we also experiment with graph edge partitioning using our algorithm on graphs that have been slightly modified in order to be accepted as an input for our algorithm. We show that using our algorithms results in a lower amount of replicated vertices but a higher running time and imbalance compared to an algorithm proposed specifically for this task.

1.3 Structure

The remainder of this thesis is organized as follows. It starts with Chapter 2. It contains formal definitions of concepts, some formulas, and notation as well as algorithm settings used throughout the thesis. It is followed by Chapter 3 which describes previously made advancements in graph and hypergraph partitioning, including previously proposed streaming hypergraph partitioning algorithms. It consists of two parts describing algorithms for graph partitioning and algorithms for hypergraph partitioning. Merits of each algorithm are reviewed and the need to propose our algorithm is justified. Among related works, we also include the description and some experiments with hFennel in order to provide some context for this work. The main chapter, Chapter 4, covers our non-buffered (purely streaming) algorithm with sampling and also contains the description of the buffered algorithm. Chapter 5 starts with the setup and data sets used for all experiments. It then details the experiments themselves, including parameter fine-tuning and comparison with other algorithms. The final chapter, Chapter 6, consists of a concluding section summarizing the algorithms and findings from the experiments and a section with future work proposals.

CHAPTER 2

Fundamentals

2.1 General Definitions

Let $H = (V, E)$ be a *hypergraph* with a set of *vertices* V and a set of *nets* $E \in 2^V$, such that $n = |V|$ and $m = |E|$. A vertex $v \in V$ is said to be a *pin* of a net $e \in E$, if $v \in e$. In this work, the term pin will be used interchangeably with terms *hypernode* or *vertex* in the context of nets, while the term net with term *hyperedge*. Let $c : V \to \mathbb{R}_{\geq 0}$ be a vertex weight function, and let $\omega : E \to \mathbb{R}_{>0}$ be an 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 $Inc(v) = \{e \in E \mid v \in e\}$ denote the incident nets of v, meaning all nets in which v is a pin. A net e is said to be *touching* $V' \subseteq V$ if $V' \cap e \neq \emptyset$, meaning that it has pins in V', and it is *contained* in V' if $e \in V'$, meaning that all its pins are in V'. The *graph partitioning* problem consists of assigning each vertex of G to exactly one of k distinct *blocks* respecting a balancing constraint in order to minimize the hyperedge cut or the *cut-net*. More precisely, graph partitioning 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 G. The *cut-net* of a k-partition consists of the total weight of the nets crossing blocks, i.e. $\sum_{i<j} \omega(E_{ij})$, where $E_{ij} := \{e \in E \mid e \cap V_i \neq \emptyset \wedge e \cap V_j \neq \emptyset\}$. An additional *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, $\forall i \in \{1, \ldots, k\}$: $c(V_i) \leq L_{\max} := \left\lceil (1 + \epsilon) \frac{c(V)}{k} \right\rceil$.

2.2 Streaming (Hyper)Graph Partitioning

Streaming graph partitioning as mentioned throughout this thesis works as follows: The partitioning algorithm receives vertices in a stream with the set of their incident edges. As vertices are received, the partitioning algorithm permanently assigns them to one of

the blocks before having received all vertices of a graph. In order to give this assignment process some context, the algorithm has access to the subgraph defined by all the previously received vertices. In order to save memory, it is not able to access all the information about that subgraph. For example, an edge running between two vertices already assigned to blocks is not needed. [36] The algorithm might be designed either to receive vertices one-by-one or to load multiple vertices into a buffer and assign them to blocks. For the rest of this work they will respectively be referred to as *non-buffered* and *buffered* streaming graph partitioning algorithms. Some mentioned works have experimented with different orders in which vertices are received: breadth-first search, depth-first search, random, etc. In order to be concise, we only include results with natural vertex ordering which also corresponds to real-life graph streaming. It is also possible to apply streaming partitioning algorithms in the context of not only graphs but also hypergraphs.

2.3 Multilevel Scheme

In contrast to streaming partitioning, a multilevel scheme requires the whole (hyper)graph to have been loaded into memory. This heuristic works recursively on multiple levels. The (hyper)graph is first *coarsened* over multiple levels, i.e. its number of vertices is reduced until a certain *stop condition* is met. This is done by *contracting* vertices into heavier ones. Contracting a set of vertices $\{v_1, ..., v_x\}$ into a vertex u can be described as follows: The vertices $v_1, ..., v_x$ are replaced by u with weight $c(u) = \sum_{v \in \{v_1,...,v_x\}} c(v)$. For each outgoing edge or net e with $\exists v \in e$ such that $v \in \{v_1, ..., v_x\}$, v is replaced by u. Depending on the implementation of this operation, parallel edges or nets may be replaced by one with its weight being the sum of their weights. The result of coarsening is an approximated "compressed" (hyper)graph. When the (hyper)graph is sufficiently small, at the coarsest level, it is possible to apply an expensive *initial partitioning* algorithm. The block assignment of each vertex in the coarsest level implies the same assignment for all finer-level vertices that were contracted into those coarse-level vertices with equal objective function and balance. The initial number of vertices is then gradually restored. In other words, the coarsening previously done at each level is undone at the same level. During *uncoarsening* or *local improvement*, the latest contracted vertices are uncontracted, meaning vertices and edges are restored, and a *refinement* algorithm is applied, moving vertices between blocks in order to improve the objective or balance. This is repeated over all levels until the initial (hyper)graph is restored. [14]

CHAPTER 3

Related Work

3.1 Graph Partitioning

There has been a large amount of research on graph partitioning. We refer the reader to [4, 7, 35] for extensive material and references. In this section, we summarize the related work on streaming graph partitioning. It contains different heuristics proposed by Stanton and Kliot [36] as well as some additional algorithms proposed by other authors [38, 13]. There is also a wide range of further algorithms that focus on (buffered) streaming edge partitioning [29, 24, 31], but they will not be examined here for the sake of concision.

3.1.1 Non-buffered streaming graph partitioning

Stanton and Kliot [36] examined many natural heuristics to solve graph partitioning problems in a streaming setting. *Balanced* heuristic, upon receiving it, simply assigns each incoming vertex v to the block V_i of the minimal size, reinforcing balance between blocks. Hence, the name.

$$i = argmin_{j \in \{1,\dots,k\}} |V_j| \tag{3.1}$$

Chunking heuristic divides the stream into chunks of block upper bound size L_{max} and fills blocks in a sequence. For this, the heuristic only uses the order number t of the current vertex:

$$i = \lceil \frac{t}{L_{max}} \rceil \tag{3.2}$$

Hashing heuristic uses a hash function in order to assign (or map) vertices to blocks based on their index. The simplest function and the one used by Stanton and Kliot was:

$$i = (v \mod k) + 1 \tag{3.3}$$

It should be noted that the previously mentioned three heuristics all disregard the graph structure, i.e. edges between otherwise relevant nodes. Thus, their expected performance is cutting a relatively high fraction of edges. The main benefit of these simple heuristics is that there is no need to store block assignments for vertices because this information is not accessed while processing future vertices.
Deterministic Greedy heuristic tries to assign a vertex v to the block V_i with which it has the most common edges. The number of edges is counted as the number of neighbor vertices of v, with the set of neighbors denoted by $N(v)$ that are also in a block V_i. In different variations of this heuristic, the number of common edges with each block is weighted by a penalty function based on the capacity of the block, penalizing larger blocks:

$$i = argmax_{j \in \{1,...,k\}} |V_j \cap N(v)| w(j) \tag{3.4}$$

with $w(j)$ being the aforementioned penalty function:

- $w(j) = 1$ for *unweighted greedy*
- $w(j) = 1 - \frac{|V_j|}{L_{max}}$ for *linear weighted* or *LDG*
- $w(j) = 1 - \exp |V_j| - L_{max}$ for *exponentially weighted*

Randomized Greedy heuristic assigns v to a random block V_i. A discrete probability distribution for each block weights the probability of each block by its score obtained via deterministic greedy heuristic:

$$P(i \in \{1, ..., k\}) = \frac{|V_i \cap N(v)| w(i)}{\sum_{j \in \{1,...,k\}} |V_j \cap N(v)| w(j)} \tag{3.5}$$

The randomness is added in hope of improving the solution quality. *Triangles* heuristic prioritizes the block V_i with most triangles formed by vertices in the intersection of V_i and the node neighbors $N(v)$. Blocks are penalized in a similar manner to LDG. Using the $E(V')$ to denote all the edges in V':

$$i = argmax_{j \in \{1,...,k\}} \frac{|E(V_j \cap N(v))|}{\binom{V_j \cap N(v)}{2}} w(j) \tag{3.6}$$

Balance Big heuristic is a combination of the balanced heuristic and deterministic greedy heuristic. Upon receiving a node v, the algorithm decides whether it is high-degree or low-degree based on some function. If it is high-degree, the balanced heuristic is used. Otherwise, the deterministic greedy heuristic is used. This heuristic can be viewed as setting high-degree nodes as anchors only to keep the low-degree nodes around them.

Fennel heuristic. Tsourakakis et al. [38] proposed a one-pass partitioning heuristic named *Fennel*, which is an adaptation of the widely-known clustering objective *modularity* [5]. Roughly speaking, Fennel assigns a node 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. deterministic greedy proposed by Stanton and Kliot, but with an additive penalty instead of a multiplicative one. The assignment decision of Fennel is based on an interpolation of two properties: attraction to blocks with more neighbors and repulsion from blocks with more non-neighbors, or from blocks that are generally already heavy. When $f(|V_i|)$ is a constant, the resulting objective function coincides with the first property. If $f(|V_i|) = |V_i|$, the objective function coincides with the second property. More specifically, the authors defined the Fennel objective function by using $f(|V_i|) = \alpha \times \gamma \times |V_i|^{\gamma-1}$, 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 $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$. Note that in the original paper, the authors assume k to be constant and hence derive a complexity of $O(n + m)$. However, since one has to iterate over all blocks k for each node the complexity of the algorithm depends on k and is given by $O(nk + m)$.

3.1.2 Buffered streaming graph partitioning

Further heuristics introduced by Stanton and Kliot [36] use a buffer. *Prefer Big* heuristic maintains a buffer of size L_{max}. The buffer is filled. Then, all high-degree vertices are assigned via the balanced heuristic. Each time such a vertex is assigned, a new one is loaded. As long as the buffer does not entirely consist of low-degree vertices, the low-degree vertices in the buffer are put on hold. That way, high-degree vertices are given more flexibility to be assigned to blocks that would increase the partition balance. When the buffer consists entirely of low-degree nodes, the deterministic greedy heuristic is used to clear it. This heuristic can be seen as an expansion of the balance big heuristic. *Avoid Big* heuristic can be seen as an inversion of the refer big heuristic as it prioritizes low-degree vertices and de-prioritizes high-degree vertices in a similar manner. *Greedy EvoCut* heuristic uses the EvoCut [3] on a buffer in order to find small Nibbles, or, subsets of vertices with certain requirements to their *conductance*, and assigns entire Nibbles to blocks using the deterministic greedy heuristic. Conductance, as mentioned here, refers to the proportion of outgoing edges between a Nibble and the outside graph to the sum of degrees of all the vertices in it. Thus, they are treated as atomic sets of vertices, partitioning which might result in a high resulting edge-cut of the partitioning. Even though the standalone EvoCut algorithm has solid guarantees, its hypothetical effectiveness does not apply with the use of buffers as applied in this heuristic. However, despite the premise, all methods that use a buffer perform significantly worse than LDG, which had the best overall results in terms of the proportion of cut edges while not relying on a buffer.

HeiStream. Another buffered streaming graph partitioning heuristic called *HeiStream* has been proposed by Faraj and Schulz [13]. This heuristic consists of several stages and levels for each chunk of a graph loaded into a buffer and is able to work with weighted graphs. Unlike Prefer Big or Avoid Big heuristics discussed previously, this heuristic loads mutually exclusive sets of vertices into a buffer akin to Greedy EvoCut. Here, we describe the algorithm.
A *basic model* graph is constructed from the vertices in the buffer of size δ. This is done by initializing the model as a subgraph B of the input graph G consisting of the vertices in the buffer. k artificial vertices are added each corresponding to one of the blocks V_i for $i \in \{1, ..., k\}$ to which vertices have already been assigned. Each one of these artificial nodes is given weight $c(V_i)$. Edges between a vertex u previously assigned to a block V_i and a vertex v in the buffer are added to the model for each such pair of vertices, whereas edges between previously assigned vertices are ignored. Parallel edges between each pair of vertices consisting of a set of edges running between the same pair of vertices are replaced by one edge with the weight equal to the sum of weights of the parallel edges. Therefore, define a notation can be defined:

$$\omega(V_i, v) = \sum_{u \in N(v) \cap V_i} w(u, v) \tag{3.7}$$

V_i as used in this formula can be understood as one of the previously defined artificial vertices. HeiStream also incorporates the building of *extended models* containing artificial ghost vertices and edges that have neither been previously assigned to a block nor been loaded into the buffer. [13] The model is then partitioned using a *Multilevel Fennel* heuristic which follows the scheme described in 2.3 with some modifications. After the model is partitioned with Multilevel Fennel, the vertices of the original graph are assigned to the blocks of their corresponding vertices in the model.
Multilevel Fennel heuristic itself relies on many configurable parameters for each step of the algorithm. The coarsening part uses a clustering technique called *label propagation* [30] algorithm or LPA while using a *size constraint* in order to keep clusters balanced. [26] This works as follows. Note that clusters are different from blocks. At the beginning of LPA, each vertex v is assigned to its own individual cluster $C_v := \{v\}$ containing this vertex only. Throughout a run, each vertex v is assigned or reassigned to a cluster C_i with the largest sum of edges between that cluster and v:

$$i = argmax_{j \in \{1,...,|B|\}} \sum_{u \in C_j \cap N(v)} \omega(u, v) \tag{3.8}$$

It is ensured that v can only be assigned to a cluster C_i with:

$$c(C_i) + c(v) \leq \lceil \frac{L_{max}}{coarsening\_factor} \rceil \tag{3.9}$$

In this case, $coarsening\_factor$ is a tunable parameter, but it is set to a default of 18[1]. LPA runs can be repeated, and at each run, the vertices may be reassigned to different clusters. During fine-tuning, 5 was found to be a good number of LPA runs by Faraj and Schulz. Throughout LPA, the artificial vertices are completely ignored. After the vertices are assigned clusters, all sets of vertices belonging to the same clusters are contracted. Vertex contraction is performed in a similar manner to artificial vertex creation during model construction. This is repeated until either the contractions no longer have any significant effect as measured by the change in the number of resulting coarse vertices, or the amount of coarse vertices has reached a threshold of $max(\frac{|B|}{2xk}, xk)$ in which x is a tunable parameter with 4 being a good value for it. At the coarsest level, an initial partitioning is performed. A modified version of the Fennel heuristic is used, defined as follows. Each coarse node v is assigned to the block V_i with:

$$i = argmax_{j \in \{1,\dots,k\}} \sum_{u \in V_j \cap N(v)} \omega(u,v) - c(v) \times \alpha \times \gamma \times c(V_j)^{\gamma-1} \tag{3.10}$$

The values for α and γ found good by Tsourakakis et al. [38] were used in order to remain consistent with their work. Again, a balancing constraint is enforced in order to keep block weights below L_{max}. In the uncoarsening or refinement stage of a recursion level, the vertices contracted at the current are uncontracted. They are moved between blocks based on the Fennel score. In this stage, in contrast to the Fennel algorithm used for the initial partitioning, when a vertex v is visited, it is removed from its current block, and only those blocks are considered which contain neighbors of v: $V_j \cap N(v) \neq \emptyset$.
It should also be noted that during coarsening, only non-artificial vertices are contracted, and during initial partitioning and refinement, only non-artificial vertices are assigned to or moved between blocks, while the artificial vertices are used only to compute Fennel scores of different blocks as they represent the current state of their respective blocks.

3.2 Hypergraph Partitioning

Although previously mentioned heuristics provide a basis upon which streaming hypergraph partitioning algorithms can be developed, they are not yet applicable to hypergraphs as they don't take into consideration that hyperedges can contain a number of vertices different from 2. Hypergraph partitioning methods include a wide variety of algorithms such as *KaHyPar* [34], HYPE [23], Min-Max [1], hMetis [20], and many others [8, 11, 37, 39]. Many of these algorithms are beyond the scope of this work since they have little in common with our work in streaming hypergraph partitioning algorithms.

[1]Repository for HeiStream: https://github.com/marcelofaraj/Buffered-Partitioning

3.2.1 KaHyPar

KaHyPar [34], proposed by Schlag et al. is a sophisticated hypergraph partitioning algorithm composed of a careful combination of different heuristics. It employs a multilevel approach with multiple phases.
Preprocessing phase is done before the actual n-level part of the algorithm. During this phase, two tasks are carried out: *pin sparsification* and *community detection*. Pin sparsification contracts similar vertices, with similarity being defined by the fraction of shared nets relative to all the incident nets of vertices. This is done in order to reduce the average count of pins in a net, which improves the overall running time of the algorithm. Community detection, on the other hand, tries to divide a hypergraph into disjoint subhypergraphs (communities) such that connections are dense within those communities but sparse between them. This division is later used for the *coarsening* phase. *Coarsening* is similar to the one performed in HeiStream with two key differences. One of them is that instead of contracting vertices only after the clusters for each vertex have been calculated, KaHyPar contracts vertices on the fly before moving on to the cluster calculation for the next vertex. The other difference lies in the fact that the rating function for clusters prefers nets with a low incidence as shown in Equation 3.11.

$$Rating(u, v) := \sum_{e \in Inc(u) \cap Inc(v)} \frac{\omega(e)}{|e| - 1} \tag{3.11}$$

Initial Partitioning is done via the recursive bisection algorithm in the direct k-way setting. While doing so, a portfolio of 9 different initial partitioning algorithms is used, such as Random and BFS-based Partitioning [9, 19], Greedy Hypergraph Growing (GHG) [8] and Size-constrained Label Propagation (similar to the clustering in HeiStream). *Refinement* is possible using two types of algorithms: *Localized* 2*-way or* k*-way FM Local Search* algorithms that follow the FM paradigm [15] by starting the search from the newly uncontracted vertices or *Flow-Based Refinement*, which integrates a flow-based bipartitioning algorithm (FlowCutter) [17] into the n-level framework.
It should also be noted that KaHyPar can be run in two modes: *direct k-way* mode which directly uses the phases mentioned above or *recursive bipartitioning* which repeats the phases mentioned above for each one of its $O(\log k)$ recursion levels. Currently, KaHyPar is the highest quality weighted hypergraph partitioner. It is however computationally intensive.

3.2.2 HYPE

Mayer et al. [23] proposed HYPE, a light-weight hypergraph partitioning algorithm based on *neighborhood expansion*. It works by gradually filling blocks with vertices until they reach a size of $\frac{|V|}{k}$. In that sense, it is somewhat reminiscent of the chunking heuristic proposed by Stanton and Kliot [36]. However, its main difference is in the choice of the next node to be added to the block. For each block V_i, *core set* C_i is defined to represent

vertices already assigned block V_i, *Fringe* set F_i for vertices that are considered for an addition into C_i and the *vertex universe* $V' \subseteq V$ with nodes that can be added into F_i and have not yet been added to any block V_j or core set C_j with $j \leq i$.
For each block, the algorithm adds some seed vertices to the core set and then loops over the following steps until the block is filled. In the first step, r fringe candidates are determined. A set of nets of C_i is constructed: $\{e \in E \mid C_i \cap e \neq \emptyset\}$. A sorted vector is constructed from the set of nets so that nets with lower degrees precede the ones with higher degrees. r pins are picked from the first nets. The pins must be in V'. In the second step, the candidate pins are added to the fringe F_i while keeping only an s amount of pins with the least number of *external neighbors* $d_{ext}(v, F_i)$:

$$d_{ext}(v, F_i) = |\{u \in V \mid \exists e \in E \ : \ v \in e\} \backslash F_i| \tag{3.12}$$

The candidate vertex in F_i with the least number of *external neighbors* is then moved from the fringe to the core set C_i and is, thus, assigned to the block V_i. The vertex is also removed from V'. The values r and s are tunable. Although HYPE requires very low computational resources, it cannot operate as a streaming algorithm and also cannot partition weighted hypergraphs.

3.2.3 Streaming Min-Max

Alistarh et al. [1] have proposed a Min-Max streaming hypergraph partitioning method. Being a streaming heuristic, it receives a vertex v and assigns it to a block on the fly in the following steps.
First, candidate blocks are chosen so that the balancing constraint is not violated. Unlike most streaming algorithms that use a hard constraint $V_i < \lceil \frac{|V|}{k} \rceil$ for each candidate block V_i, the Min-Max algorithm uses a different kind of balancing constraint using the size of the smallest block so far:

$$V_i \leq argmin_{j \in \{1,\dots,k\}} |V_j| \times (1 + \epsilon) \tag{3.13}$$

The balancing constraint used here ensures that the blocks remain balanced throughout the partitioning process. Its main advantage is that its enforcement does not require any knowledge about the total size of the hypergraph. After the candidate blocks are chosen from the set of all blocks, each such block V_j is rated based on the number of nets common for the block and for the vertex v. The block V_i with the highest such number of nets is chosen for the vertex v:

$$i = argmin_{j \in \{1,\dots,k\}} |Inc(v) \cap \{e \in E \mid \exists u \in V_j \ : \ u \in e\}| \tag{3.14}$$

As an optimization measure, sets S_i can be defined for $i \in \{1, ..., k\}$. These sets are empty at the beginning of the algorithm, but each time a vertex v is added to a block V_i, the incident nets of v are added to the set S_i. Thus each set S_i can be used instead of $\{e \in E \mid \exists u \in V_j \ : \ u \in e\}$ and accelerate the counting of nets for each block.

3.2.4 hFennel

The algorithms described previously in this section despite being applicable to the hypergraph partitioning task either do not consider vertex and net weights (HYPE and Min-Max) or do not load vertices in a streaming setting (KaHyPar and HYPE). In order to address those criteria, in a previous project [12], we developed an algorithm inspired by the Fennel one-pass algorithm. Throughout the thesis, this algorithm will be referred to as hFennel.

Overview. The algorithm builds upon the work of Tsourakakis et al. [38] Similarly to Fennel, it makes a pass over the vertices or vertex assigning each vertex v to the block with the highest objective score. The score is calculated in such a way that the blocks with more nets (or a higher net weight sum) to the vertex are given more scores at the same time as the blocks with generally more vertices (heavier blocks) have their scores reduced. Thus, the vertex v is assigned to the block that would minimize the cut-net and maintain some balance:

$$argmax_{j\in\{1,\dots,k\}} Fennel(V_j, v) = argmax_{j\in\{1,\dots,k\}} (g(V_j, v) - f(c(V_j), c(v))) \quad (3.15)$$

The two parts of the Fennel function are expanded (the part pertaining to nets connecting the block and the vertex $g(V_j, v)$ and the additive penalty $f(c(V_j), c(v)))$ in the following paragraphs in order for our algorithm to be able to process weighted hypergraphs.

Minimizing cut-net. In their work, for a block V_j and a vertex v, Tsourakakis et al. used the term $|V_j \cap N(v)|$ for the first part of the Fennel function, which is the number of neighbors of v in V_j. Since this is equal to the number of incident edges of v that connect it to V_j, this term is ideal for unweighted graphs. For hypergraphs, however, there can be different definitions, referred to in [12] as *net criteria*, for a net between a block and a vertex listed as follows. An incident net $e \in E$ of a vertex v (meaning $v \in e$) can be said to be connecting the block V_j and the vertex v if it is:

- *touching* V_j (meaning $e \cap V_j \neq \emptyset$);
- *contained* thoroughly in $V_j \cup v$;
- or something *intermediate* by touching V_j but not touching any other block yet.

The intermediate criterion can be used in the streaming setting in order to minimize the cut-net since it avoids false negatives in the early iterations of the vertex pass (when few vertices have been assigned to blocks) and false positives in the later iterations (when most vertices are already in blocks). Apart from that, instead of counting nets, we calculate the weight sum of those nets in order for hFennel to be able to process weighted hypergraphs. Thus, for a set $Connecting(V_j, v)$ of nets connecting V_j and v, we define $g(V_j, v) := \omega(Connecting(V_j, v))$.

Penalizing imbalance. For hFennel, we defined $f(c(V_j), c(v)) := c(v) \times \alpha \times \gamma \times c(V_j)^{\gamma-1}$ with $\alpha = \omega(E)\frac{k^{\gamma-1}}{c(V)^\gamma}$. This formulation of g had been inspired by the work of Faraj and Schulz [13]. This term subtracts some penalty from the Fennel score of the block V_j based on its weight. It is the direct generalization of the term $\alpha \times \gamma \times |V_j|^{\gamma-1}$ defined by Tsourakakis et al., the difference being that we used the weight of a block instead of its size and a different α. The computation of the new value of α takes the weight sums of all vertices and all nets into consideration. Faraj and Schulz also proposed to multiply the term $\alpha \times \gamma \times c(V_j)^{\gamma-1}$ by $c(v)$ in order to retain a certain property needed for buffered streaming partitioning. We decided to keep that multiplication so that hFennel is better adaptable for modifications in the future. The parameter γ is a tunable variable. The higher it is, the more significance is given to the load balance of blocks.

Values of constants. For our later experiments, α and γ had to be adjusted as well. First, α had to be generalized for weighted graphs. In their work, Tsourakakis et al. choose $\alpha = m\frac{k^{\gamma-1}}{n^\gamma}$. Using a similar reasoning process adapted for weighted hypergraphs, we obtained $\alpha = \omega(E)\frac{k^{\gamma-1}}{c(V)^\gamma}$. The value of γ is later fine-tuned experimentally.

During the experimental fine-tuning, it was determined that the best edge criterion for the task was "intermediate" while the best values of γ were between 1 and 1.5. [12] We then compared the resulting hyperedge cuts on unweighted hypergraphs for the following algorithms: hash partitioning which assigns vertices to blocks based on the application of a hash function causing some randomness, Min-Max, HYPE, as well as hFennel with $\gamma \in \{1, 1.25, 1.5\}$.
The results for unweighted hypergraphs are shown in Table 3.1. It could be observed from the table that the hash algorithm clearly has an inferior hyperedge cut performance since it pretty much assigns vertices to blocks randomly while not violating the balancing constraint. For the values of k from 8 to 64, HYPE also created a partition with a higher cut-net relative to hFennel with $\gamma = 1$, but its cut-net seemed to be lower for $k > 64$. (Additional experiments showed that HYPE is able to generate very highly balanced partitions without needing a hard balancing constraint.) It should be noted, however, that HYPE is not a streaming partitioning algorithm. Min-Max generated a higher cut-net for all values of k present in the experiments, higher even than hFennel with $\gamma = 1$. For hFennel, however, $\gamma = 1.25$ delivered a lower cut-net for $k \leq 128$, but seemed to perform worse than $\gamma = 1.5$ in that regard for higher values of k. In fact, this deviation increased, which made 1.5 the best γ for a large number of blocks and unweighted hypergraphs. The results for weighted hypergraphs are shown in Table 3.2. From both tables, it is evident that hFennel had lower cut-net with the values $k \geq 32$ with $\gamma = 1.25$ or $\gamma = 1.5$ rather than with $\gamma = 1$, although it is difficult to choose between 1.25 and 1.5 for the best γ.

k	Hash	Min-Max	HYPE	hFennel 1	hFennel 1.25	hFennel 1.5
8	1 689 487	1 178 572	631 945	366 296	476 047	499 956
16	1 741 042	1 385 731	782 175	528 963	588 871	612 965
32	1 765 101	1 490 473	900 546	747 721	730 962	743 724
64	1 776 559	1 580 678	1 022 426	991 184	841 689	849 302
128	1 781 726	1 630 663	1 112 838	1 230 781	970 725	973 797
256	1 783 600	1 665 441	1 213 483	1 380 855	1 102 366	1 098 188
512	1 784 311	1 691 693	1 314 013	1 463 686	1 200 505	1 192 208
Mean	1 759 962	1 507 310	969 931	863 242	806 307	818 825

Table 3.1: Rounded geometric means of edge cuts over unweighted hypergraphs for different algorithms for different numbers of blocks k. hFennel with different γ values is shown as "hFennel γ".

k	Hash	hFennel 1	hFennel 1.25	hFennel 1.5
8	16 887 002	3 943 434	5 962 314	5 950 597
16	17 423 919	5 722 992	6 848 847	6 870 662
32	17 684 092	8 108 696	8 081 139	8 063 090
64	17 809 748	10 735 426	9 225 398	9 317 757
128	17 869 296	13 151 307	10 460 890	10 602 726
256	17 892 241	14 614 759	11 848 026	12 021 410
512	17 899 019	15 278 793	12 825 789	12 775 839
Mean	17 634 490	9 244 298	9 015 085	9 057 726

Table 3.2: Rounded geometric means of edge cuts over weighted hypergraphs for different algorithms for different numbers of blocks k. The values are multiples of 10. hFennel with different γ values is shown as "hFennel γ".

CHAPTER 4

Streaming Hypergraph Partitioning

In this chapter, we describe two algorithms that serve as an expansion of the previously proposed heuristics in the field of hypergraph partitioning. Section 4.1 describes an extension of our previous work, namely, hFennel, to which we added sampling in order to decrease its running time. In Section 4.2, we combine parts of HeiStream with KaHyPar in order to develop a prototype buffered hypergraph partitioning algorithm.

4.1 hFennel with Sampling

In [12], we proposed a streaming Fennel-like heuristic for hypergraph partitioning. The heuristic proved to be delivering high-quality hypergraph partitions among other streaming algorithms and is applicable to *weighted* hypergraphs. In this work, we add non-determinism and sampling in order to decrease the overall computational complexity of the algorithm and perhaps improve its running time. Section 4.1 first introduces the hypothesis and the reasoning for the employment of sampling. Then, each subsequent part of it describes a specific part of the algorithm and the reasoning for some design decisions. In conclusion, Section 4.1.5 presents the overall implementation as used later during our experiments.

4.1.1 Hypothesis

Although some work has been done in the field of hypergraph partitioning and streaming graph partitioning algorithms, the concept of using sampling for the task is relatively new and has not been well researched yet. Algorithms like SMGP and SMGP+ [18] or RD-B-GRAP and HD-B-GRAP [27] proposed in the years 2021 and 2022 respectively employ sampling on the very structure of an input graph in order to improve the partitioning time and yield a competitive partitioning quality. Indeed, it is possible to significantly improve the running time of a partitioning algorithm by, for example, reducing the number of edges

or nets considered during the computation, as most such algorithms usually contain loops iterating over incident edges or nets of relevant vertices. Reducing the number of iterations in such loops naturally decreases the running time of those loops by the factor of edges or nets remaining after sampling relative to their original number.
In the case of the streaming (hyper)graph partitioning algorithms that rate blocks based on a certain heuristic and assign an input vertex to the block with the best rating, a different approach to sampling can also be employed. hFennel is one such algorithm as for each vertex v it has to rate k blocks and choose one. [12] Thus, it contains a loop that iterates over k blocks. Similarly to using edge or net sampling in order to reduce iterations over them, one may also reduce the number of iterations over these blocks. We define $c < k$ to be the number of samples. This part of hFennel has an asymptotic complexity of $O(nk)$ without sampling since it has to compute ratings for k blocks for each one of n vertices. With sampling, this complexity is reduced to $O(nc)$. This means sampling results in an algorithm whose complexity is factor $\frac{k}{c}$ lower and which is therefore faster. However, integrating sampling into a streaming (hyper)graph partitioning algorithm comes with some issues that have to be addressed. In the following subsections, we describe the design decisions that we make in order to address those issues.

4.1.2 Sampling Algorithm

One of the important components of every heuristic with sampling is the sampling algorithm. In our specific case, its task is: Given a *population* of k blocks, it must randomly pick a *sample* of size c. The choice of the right sampling algorithm is critical. In our work, we define two main criteria for the sampling algorithm: the algorithm must be as unbiased as possible, and it must have a running time complexity of $O(c)$. In search of such an algorithm, we reviewed several algorithms. The algorithms included reservoir sampling [22, 40, 21] and permutation-based sampling (by using shuffling [21]), which had a complexity of $O(k)$, making these sampling algorithms unviable as it defeats the purpose of having sampling in our algorithm by keeping the overall complexity of our algorithm at $O(nk)$.
The algorithm that we decided to choose is one for *sampling with replacement*. It works by generating and picking a random integer in a range $\{1, ..., k\}$ and repeating this process c times. Random integer generation has a complexity of $O(1)$, wherefore the sampling process is $O(c)$. It should be noted that the algorithm does not check whether an element has been picked. As a result, some elements are picked multiple times, and the resulting sample contains $c' \leq c$ unique elements. An issue arises when the value of $c' << c$. Such a case is undesirable, and it is difficult to guarantee that. The worst hypothetical case scenario might happen if $c = k$. It can be considered the worst case since the closer the value of c is to k, the more the chance of such "collisions". However, we determined in our preliminary experiments that the proportion of unique elements in a sample $\frac{c'}{c}$ remains in a range approximately between 60% and 72%. Thus, it can be concluded that cases with $c' << c$ are unlikely, and this sampling algorithm is viable.

4.1.3 Advanced Infeasibility Handling

hFennel without sampling is able to iterate over blocks V_i for $i \in \{1, ..., k\}$ in order to choose the one with the highest Fennel score for a vertex v while simply disregarding blocks V_i with $c(V_i \cup \{v\}) > L_{max}$. While integrating sampling into hFennel, it is desirable that the sampling of blocks, the filtering of blocks with an admissible load, and choosing the best block are three separate steps. In this case, the sampling operation should come first, as otherwise at least one of the other two steps would iterate over all k blocks, keeping the overall running time complexity of the algorithm at $O(nk)$.
Since sampling is a part of our algorithm, our options while choosing a block V_i for a vertex v are even more limited. We define the set of all blocks $\mathbb{V} := \{V_i \mid i \in \{1, ..., k\}\}$, the set of sampled blocks $\mathbb{V}_{sample} \subseteq \mathbb{V}$ with $|\mathbb{V}_{sample}| = c' \leq c$ and the set of available blocks $\mathbb{V}_{available} := \{V_i \in \mathbb{V} \mid c(V_i \cup \{v\}) \leq L_{max}\}$. As mentioned in [12], for weighted hypergraphs it is possible that for a vertex v, no block V_i is available with $c(V_i \cup \{v\}) \leq L_{max}$, or $\mathbb{V}_{available} = \emptyset$. When using sampling, we are not considering blocks that are only in $\mathbb{V}_{available}$, but the blocks only in $\mathbb{V}_{sample} \cap \mathbb{V}_{available}$, an intersection which is even more likely to be empty. It is worth mentioning that it would be less likely for that intersection to be empty if the order of steps previously defined had the filtering operation as the first one and the sampling operation as the second one because then $\mathbb{V}_{sample} \subseteq \mathbb{V}_{available}$ would hold. However, recall that because of the chosen order of operations the sample is likely to contain and, perhaps, consist only of infeasible blocks. Therefore, an advanced unavailable block resolution is required. To propose such a method, we looked at the existing unavailable block resolution from hFennel without sampling as defined in Section 3.2.4 and try to modify them.
The *Fennel* resolution simply disregards the balancing constraint. [12] Its behavior with sampling can be modified to iterate over all sampled c' blocks and has therefore no further modification potential worth mentioning that would be relevant. The *load* resolution iterates over all blocks while disregarding the Fennel score, and the block with the lowest load is selected. [12] After the blocks have been sampled, the one with the lowest load among c' blocks can be selected. For this resolution, no further modification can be done either. The *random* resolution is, however, different from the previous resolutions. It can be modified in such a way that the block is not chosen among c' sampled blocks like the previous two resolutions but among all k possible blocks similar to how it happens in the algorithm without sampling. Thus, there is a chance that a block V_i can be picked for a vertex v such that $c(V_i \cup \{v\}) \leq L_{max}$. Furthermore, the algorithm can be modified in such a way that not one but a limited amount of attempts can be made picking a random block until a block with a low enough load is found.

Random resolution with attempts. In our work, we modify the random unavailable block resolution in order for it to be able to find a block that can meet the hard balancing constraint L_{max} after several attempts. In order for the overall algorithm running time complexity not to reach $O(nk)$, the number of those attempts must be less than k (for the

case if, for example, all blocks are tried out) for each $v \in V$. In other words, the number of such attempts per node must be bound by some constant d with $d < k$. It is pertinent to mention that the number of attempts is *bounded* by d, and that it is very likely that a right block is found in d' attempts with $d' < d$ for the majority of streamed vertices. This opens up the possibility of using the remaining or unused attempts during future invocations of the random resolution. As a result, the amortized asymptotic number of all attempts remains $O(nd)$ throughout the partitioning algorithm run, while using as many attempts as possible to find a block with low enough load for each vertex.

4.1.4 Sampling Modes

When designing a streaming hypergraph partitioning algorithm with sampling, a decision has to be made as to which subset of all blocks should be used as the population for the sampling algorithm. Assuming that sampling from the set $\mathbb{V}_{neighbor} := \{V \in \mathbb{V} \mid V \cap \bigcup_{e \in Inc(v)} e \neq \emptyset\}$ of neighbor blocks of v may affect the solution quality, we experimented with four sampling modes.

Neighbors mode. A block V_i is a neighbor block of v, if it has one or multiple nets satisfying one of the net criteria as defined in Section 3.2.4. Such blocks have a higher Fennel score due to the weights of the aforementioned nets. The blocks like that are more likely to be the ones with the highest Fennel score. In this sampling mode, we therefore sample blocks from $\mathbb{V}_{neighbor}$ instead of sampling from $\mathbb{V}$. When there are no neighbor vertices in any block yet (as at the beginning of the streaming process), random assignment is used.

All blocks mode. This is the simplest mode the output of which is not affected by the neighbor blocks of a vertex. Since blocks already containing large amounts of vertices are more likely to attract further vertices, some mechanism should be employed that would let incoming vertices be assigned to blocks with no neighbor blocks naturally and without invocation of an unavailable block resolution. This sampling mode addresses that by regarding all blocks for sampling. It should nevertheless be noted that for cases like $|\mathbb{V}_{neighbor}| << k$, this mode might result in samples that contain few or no neighbor blocks, which has an effect on the choice of the highest scoring block.

Non-neighbors mode. This sampling mode addresses the issue with both the first mode and the second one by adding *all* neighbor blocks to the sample of all blocks. For an average $|\mathbb{V}_{neighbor}|$ much smaller than k, adding all neighbor blocks should have no significant effect on the running time complexity of the algorithm. In addition to this neighbor blocks set, additional c' with $c' \leq c$ blocks are added to the sample, which addresses the same issue as the previous sampling mode.

Twofold mode. This method tries to combine the neighbors mode with all blocks mode by picking c'_1 from $\mathbb{V}_{neighbor}$ and c'_2 from all blocks $\mathbb{V}$ with $c'_1 \leq \left\lfloor \frac{c}{2} \right\rfloor$ and $c'_1 \leq \left\lceil \frac{c}{2} \right\rceil$. The issue with this sampling mode is that $\mathbb{V}_{neighbor} \subseteq \mathbb{V}$ which might cause an overlap between the two samples, resulting in an even smaller general sample.

The previously defined sampling modes each have their own pros and cons. And, since it is difficult to theoretically determine the superior one of the aforementioned sampling modes, we do that experimentally in Chapter 5.

4.1.5 Algorithm Overview

An overview of hFennel with sampling with its steps is presented in Algorithm 1. The algorithm first initializes the set $\mathbb{V}$ of all blocks and d_{cumul} which accumulates yet unused attempts as described in Section 4.1.3. It then receives or iterates over vertices v of the hypergraph.
For each v, $d_{cumulative}$ is incremented, and a set of neighbor blocks $\mathbb{V}_{neighbor}$ is initialized. Subsequently, the set of sampled blocks $\mathbb{V}_{sample}$ is initialized depending on the sampling mode. The function call $sample(\mathbb{V}, c)$ returns a set of c' sampled blocks with $c' < c$ as described in Section 4.1.2. Note that if $\mathbb{V}_{sample} = \emptyset$ after the initialization (for example, when v has no neighbors), a random dummy block is added from $\mathbb{V}$. That block may be used in the later steps. After sampling, the set of blocks $\mathbb{V}_{consider}$ to be considered while comparing Fennel scores is initialized by filtering out heavy blocks V_i with $c(V_i \cup \{v\}) > L_{max}$ from the sample set $\mathbb{V}_{sample}$. If $\mathbb{V}_{consider}$ ends up empty, one of the unavailable block resolutions is invoked. The Fennel resolution simply ignores the balancing constraint and adds all sampled resolutions in $\mathbb{V}_{sample}$ into $\mathbb{V}_{consider}$ for them to be considered while comparing Fennel scores. The load resolution assigns v to the lightest block among $\mathbb{V}_{sample}$ before making the overall algorithm move on to the next vertex. The random resolution, unlike the load resolution, makes several attempts in order to find a block V_i that meets the balancing constraint and assigns the v to that block. The function call $pick\_random(\mathbb{V})$ returns a random block from the set $\mathbb{V}$. If the algorithm has not moved on to the next vertex as caused by an invocation of the aforementioned resolutions, Fennel scores of the blocks in $\mathbb{V}_{consider}$ relative to v are calculated, and the vertex v is assigned to the block $V_i \in \mathbb{V}_{consider}$ with the highest score.

4.2 Buffered Hypergraph Partitioning

Work done on HeiStream [13], shows that a buffered streaming algorithm is applicable to the graph partitioning problem where the graphs are weighted and vertices in a buffer are partitioned via a multilevel heuristic. In this work, we attempt to explore a similar idea for hypergraphs in order to achieve a trade-off between resource usage and solution quality.

Algorithm 1 hFennel with sampling.

$\mathbb{V} \leftarrow \{V_i \mid i \in \{1, ..., k\}\}$
$d_{cumulative} \leftarrow 0$
for $v \in V$ **do**
  $d_{cumulative} \leftarrow d_{cumulative} + d$
  $\mathbb{V}_{neighbor} \leftarrow \{V \in \mathbb{V} \mid V \cap \bigcup_{e \in Inc(v)} e \neq \emptyset\}$
  if *neighbors sampling mode* **then**
    $\mathbb{V}_{sample} \leftarrow sample(\mathbb{V}_{neighbor}, c)$
  else if *all blocks sampling mode* **then**
    $\mathbb{V}_{sample} \leftarrow sample(\mathbb{V}, c)$
  else if *non-neighbors sampling mode* **then**
    $\mathbb{V}_{sample} \leftarrow \mathbb{V}_{neighbor} \cup sample(\mathbb{V}, c)$
  else if *twofold sampling mode* **then**
    $\mathbb{V}_{sample} \leftarrow sample(\mathbb{V}_{neighbor}, \lfloor \frac{c}{2} \rfloor) \cup sample(\mathbb{V}, \lceil \frac{c}{2} \rceil)$
  if $\mathbb{V}_{sample} = \emptyset$ **then**
    $\mathbb{V}_{sample} \leftarrow \{pick\_random(\mathbb{V})\}$
  $\mathbb{V}_{consider} \leftarrow \{V_i \in \mathbb{V}_{sample} \mid c(V_i \cup \{v\}) \leq L_{max}\}$
  if $\mathbb{V}_{consider} = \emptyset$ **then**
    if *Fennel resolution* **then**
      $\mathbb{V}_{consider} \leftarrow \mathbb{V}_{sample}$
    else if *load resolution* **then**
      $V_i \leftarrow argmin_{V_j \in \mathbb{V}_{sample}} c(V_j)$
      $V_i \leftarrow V_i \cup \{v\}$
      continue
    else if *random resolution* **then**
      repeat
        $V_i \leftarrow pick\_random(\mathbb{V})$
        $d_{cumulative} \leftarrow d_{cumulative} - 1$
      until $c(V_i \cup \{v\}) \leq L_{max} \vee d_{cumulative} = 0$;
      $V_i \leftarrow V_i \cup \{v\}$
      continue
  $V_i \leftarrow argmax_{V_j \in \mathbb{V}_{consider}} Fennel(V_j, v)$
  $V_i \leftarrow V_i \cup \{v\}$

Section 4.2.1 introduces the general idea behind our prototype algorithm. Sections 4.2.2 and 4.2.3 regard specific aspects of the algorithm unique to the buffered streaming setting.

4.2.1 General Idea

Many high-quality hypergraph partitioning algorithms like the one proposed by Schlag et al. [34], despite yielding superior solution quality with low cut-net values or other metrics, require access to the whole hypergraph before the partitioning may even be initiated. Not only does it make those algorithms inapplicable with streaming hypergraph input, but also uses a large amount of memory by having to store the entire hypergraph structure and takes a long time to partition. This is especially true for multilevel heuristics which conduct different operations at different levels. Streaming partitioning algorithms (including hFennel), on the other hand, mostly have the disadvantage of only having access to one vertex and its immediate vicinity, thus being strongly greedy and likely yielding low-quality solutions. The need arises to be able to flexibly regulate the partitioning algorithm by controlling its solution quality versus resource usage.
Faraj and Schulz [13] addressed this need for the graph partitioning task by proposing HeiStream, a buffered streaming heuristic. Their algorithm takes δ vertices at a time with their immediate vicinity and partitions them using a multilevel heuristic. Our work on buffered streaming hypergraph partitioning, being based partly on HeiStream, reuses different components of HeiStream while expanding those components to the domain of hypergraphs.
Our algorithm iterates over batches $B \subseteq V$ with $|B| = \delta$ vertices. For each batch, an artificial small hypergraph $M = (B', E_B)$ is generated. The small hypergraph, from now on referred to as the *model*, consists of $|B'|$ vertices that contain copies of vertices in the original batch and a set E_B of nets choice of which depends on the design of the algorithm. We define a function $\phi : B \longrightarrow B'$ that maps an original vertex from the batch B to the copy in the model.
When the model is generated, it is partitioned using a hypergraph partitioning heuristic. The choice of a heuristic to partition the model is flexible. The heuristic can be, for example, a multilevel hypergraph partitioning algorithm like KaHyPar, because it delivers high-quality solutions. A high running time complexity of the algorithm is not an issue in this case, since the model size remains $O(\delta)$, resulting in the overall running time complexity of the streaming algorithm being $O(n)$. The spatial complexity of the streaming algorithm is in this case equal to that of the model partitioning algorithm for a hypergraph with δ vertices. These asymptotic complexity estimates assume that parameters other than n and δ do not change. After the model M is partitioned, for each $v' \in B'$ such that an original vertex $v \in B$ exists with $v' = \phi(v)$, we assign the vertex v in the original hypergraph to the same block as v' was assigned in the model. The algorithm then iterates to the next batch.

4.2.2 Model Construction

As mentioned in the previous Section, an integral part of the algorithm constructs a model $M = (B', E_B)$ from batch $B \subseteq V$ of the original hypergraph $H = (V, E)$. This model not only serves as a constant size input for the model partitioning algorithm but also provides it information about the previously partitioned vertices.

Buffer vertices. All vertices in B are copied as they are into the model. No contraction or weight modification is done on them. Since these vertices are not contracted, each net incident to them but not incident to any one of the previously partitioned vertices is also copied with the only modification being that vertices that are neither previously partitioned nor present in B, that are, in other words, future vertices, are omitted. Vertex copies in B' are subject to the model partitioning algorithm.

Artificial vertices. In addition to vertex copies from B, the vertex set B' of the model also contains some *artificial vertices*. They are generated by contracting all vertices previously assigned to a block V_i into a single vertex with the weight $c(V_i)$ for each $i \in \{1, ..., k\}$. While copying a net e incident to a vertex $v \in B$ touching a block V_i, meaning that $V_i \cap e \neq \emptyset$, instead of being kept, pins assigned to V_i are all replaced by the artificial vertex representing V_i. Future vertices are, again, omitted. Since artificial vertices represent previously partitioned vertices in a way that is relevant to the model partitioning algorithm without being subject to it, they are flagged or set as *fixed vertices*. This means the model partitioning algorithm cannot assign them to a different block.

Parallel nets. Since future vertices are omitted, and previously partitioned vertices are contracted into artificial pins of the resulting nets, the occurrence of nets with the same sets of pins otherwise known as *parallel nets* becomes very likely. In order to improve the running time of the model partitioning algorithm by decreasing the overall number of nets in a model, parallel nets are merged. The nets created by merging other nets are given a weight equal to the sum of the weights of the merged nets.
This process can be implemented by first representing nets as vectors of sorted pins. Each pin vector will also have a function or some other way of quickly mapping it to the weight of the corresponding net. The nets represented as pin vectors are all added into one vector of nets, and this vector of nets is sorted lexicographically. After that, it is possible to iterate over the vector of nets and simply extract the first occurrence of a net with weights of all subsequent occurrences added to the weight of the first occurrence.

The model construction process described here can be seen as a generalization of model construction from HeiStream [13] for hypergraphs. The process is visualised in Figure 4.1.

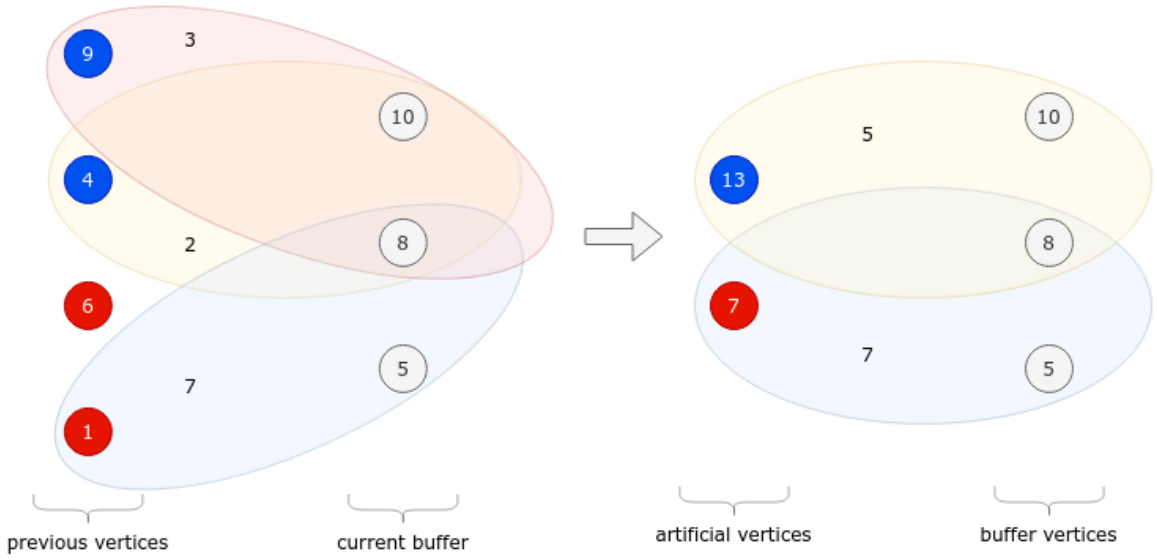

Figure 4.1: Model construction process for a buffer. Numbers indicate weights, and vertex colors (blue and red) denote their already assigned blocks. Note how artificial vertices are created by contracting vertices previously assigned to same blocks while the nets with weights 2 and 3 are merged into one with weight 5.

4.2.3 Imbalance Prescription

Partitioning algorithms usually try to enforce a hard balancing constraint. Using the same imbalance ϵ to partition each buffer ensures that blocks remain balanced over all batches. However, it is too harsh a constraint since we don't need each batch to be partitioned with an ϵ imbalance. On the other hand, using the absolute block constraint to partition each batch is also not ideal, since a multilevel algorithm such as KaHyPar would try to optimize for edge-cut as much as possible in the first batches by roughly putting all nodes in a single block. We propose a balancing constraint to address both these possible issues.
Over the batches, the weight of the constructed model approaches the weight of the overall input hypergraph $c(B') \longrightarrow c(V)$ as it contains artificial vertices with the weight of all previously partitioned vertices. Throughout the run of the algorithm, we should be able to maintain $L'_{max} := \frac{c(B')}{k}(1+\epsilon')$ equal to L_{max} so that there is more flexibility while partitioning early batches. To do so, an individual imbalance ϵ' may be prescribed for each generated model throughout the execution. The following must then hold:

$$L_{max} = \frac{c(V)}{k}(1+\epsilon) = \frac{c(B')}{k}(1+\epsilon') = L'_{max} \tag{4.1}$$

Through some arithmetic manipulations, we can obtain:

$$\epsilon' = \frac{c(V)}{c(B')}(1+\epsilon) - 1 \tag{4.2}$$

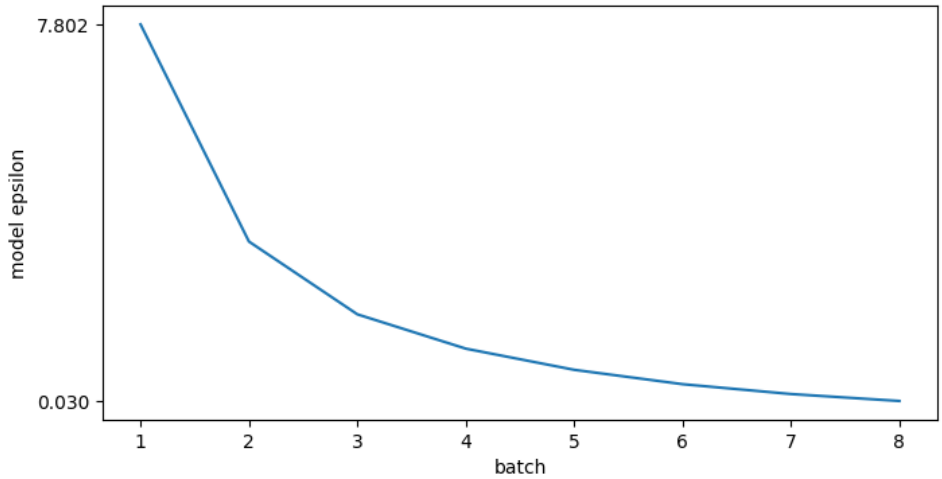

Figure 4.2: The prescribed imbalance for different batches. The weight of the first batch model is 3775 and the total hypergraph weight is 32259. The prescribed imbalance converges to 0.03.

Figure 4.2 demonstrates how the value of ϵ' is affected by the batch. It shows how its value gradually converges with the original $\epsilon = 0.03$.
While using multilevel heuristics for the model partitioning algorithm, an issue may arise if the degree of flexibility is too high, and the current state of the blocks is imbalanced as a result. Let's assume, a multilevel partitioning algorithm receives the following task: bipartitioning a model with two fixed vertices u_1 and u_2 with $c(u_1) = 10$ and $c(u_2) = 0$, $L_{max} = 15$ and 20 buffer vertices, each weighting 1. The buffer vertices are contracted, and eventually, two heavy vertices v_1 and v_2 remain at the coarsest level with $c(v_1) = 10$ and $c(v_2) = 10$. In this case, regardless of the initial partitioning, there will be an imbalance of ≈ 0.33. It is unlikely that refinement can restore the balance. Thus, it might be useful to be able to regulate the degree of flexibility with regard to ϵ'. We introduce:

$$\epsilon'' = \epsilon + \alpha\left(\epsilon' - \epsilon\right) \tag{4.3}$$

In this formula, we regulate the scale of the distance of ϵ' from ϵ, and prescribe ϵ'' to the model partitioning algorithm. The parameter $\alpha \in [0, 1]$ is tunable.

CHAPTER 5

Experimental Evaluation

5.1 Experimental Setup

We performed the implementation of our algorithms inside the KaHyPar framework (using C++) and compiled it using gcc 9.3 with full optimization turned on (-O3 flag). In our current implementations, we load the entire hypergraph to memory before streaming its nodes directly from memory. Competing algorithms HYPE and Min-Max were pulled from public GitHub repositories[1][2][3]. We have used a machine with two sixteen-core Intel(R) Xeon(R) Silver 4216 processors running at 2.10 GHz, 93 GB of main memory, and 16 MB of L2-Cache. It runs Ubuntu GNU/Linux 20.04.1 LTS and Linux kernel version 5.4.0-65-generic. Unless explicitly mentioned otherwise, we use $k \in \{8, 16, 32, 64, 128, 256, 512\}$ for our partitioning experiments. We allow an imbalance of 0.03 for all experiments (and all algorithms). All partitions computed by all algorithms have been balanced. Depending on the focus of the experiment, we measure running time, memory usage, and/or edge-cut. In general, we perform ten repetitions per algorithm and instance using different random seeds for initialization, and we compute the arithmetic average of the computed objective functions and running time per instance while fine-tuning the algorithm with sampling. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*. This is done with the running time and the memory usage of the algorithm as well, but not with the imbalance since imbalance is a proportion. Unless explicitly mentioned otherwise, we average all results of each algorithm grouped by k.

[1] Repository for HYPE: https://github.com/mayerrn/HYPE
[2] Repository for Min-Max: https://github.com/DarkWingMcQuack/MMStreamer
[3] Repository for Two-Phase Streaming: https://github.com/mayerrn/two_phase_streaming

5.2 Data Sets

5.2.1 Hypergraphs

For our experiments, we chose *some* hypergraphs from the benchmark data set by Sebastian Schlag [33]. The original set included 18 hypergraphs from the *ISPD98 VLSI Circuit Benchmark Suite* [2] consisting of 18 circuits with sizes ranging from 13,000 to 210,000 modules translated from internal IBM internal designs, 192 hypergraphs from the *University of Florida Sparse Matrix Collection* [10], a large and actively growing set of sparse matrices that arise in real applications, and 100 hypergraphs from the *International SAT Competition 2014* corresponding to different Boolean satisfiability (SAT) problems used to test the performance of SAT solvers submitted by the participants. Two *mutually exclusive* sets were derived: the *Fine-tuning set* and the *Test set* for the experiments described in the following chapters. Each set includes 1 hypergraph from the ISPD98 VLSI Circuit Benchmark Suite, 3 hypergraphs from the International SAT Competition 2014, and 6 hypergraphs from the University of Florida Sparse Matrix Collection. Only 10 hypergraphs in total for each set were chosen in order to avoid overfitting. We picked among the largest hypergraphs by size in hMetis format from each of the original sets in order to measure the solution quality and the performance of our algorithms in somewhat demanding settings. All hypergraphs are unweighted (i.e. have unit edge and vertex weights) and use the hMetis hypergraph input file format [20]. They have additionally been transformed to the edgelist format in order to be used for tests with algorithms that only support this format. In the edgelist format, each line models a node followed by a comma-separated list of edges it is connected to. To investigate the effect of vertex and net weights on the partition quality, some experiments included initializing vertex weights as the number of their incident nets and net weights as the number of their pins, since this kind of setting might be similar to many practical applications of hypergraph partitioning. Individual hypergraphs are described in Table 5.1.

ISPD98 VLSI Circuit Benchmark Suite consists of 18 circuits with sizes ranging from 13,000 to 210,000 modules and were translated from internal IBM internal designs. The designs represent many types of parts, including bus arbitrators, bus bridge chips, memory and PCI bus interfaces, communication adaptors, memory controllers, processors, and graphics adaptors. For each circuit, a cell is considered to be an internal movable object, a pad is an external (perhaps movable) object, and a module is either a cell or a pad. Each circuit is a translation from VIM (IBM's internal data format) into "net/are" format, a simple hypergraph representation originally proposed by Wei and Cheng [41] (see vlsicad.cs.ucla.edu for benchmarks in this format). [2] *University of Florida Sparse Matrix Collection* is a large and actively growing set of sparse matrices that arise in real applications. The Collection is widely used by the numerical linear algebra community for the development and performance evaluation of sparse matrix algorithms. It allows for robust and repeatable experiments: robust because performance results with artificially generated

Hypergraph	Vertices	Nets
Tune set		
ISPD98_ibm16	183 484	190 048
human_gene2	14 340	14 340
nd12k	36 000	36 000
gupta3	16 783	16 783
Rucci1	109 900	1 977 885
ecology1	1 000 000	1 000 000
mono_500Hz	169 410	169 410
sat14_SAT_dat.k90.debugged	2 873 830	5 715 644
sat14_SAT_dat.k85-24_1_rule_2	2 712 808	5 395 102
sat14_SAT_dat.k80-24_1_rule_1	2 551 810	5 074 584
Test set		
ISPD98_ibm15	161 570	186 608
BenElechi1	245 874	245 874
gearbox	153 746	153 746
3Dspectralwave2	292 008	292 008
pre2	659 033	659 033
Hamrle3	1 447 360	1 447 360
para-4	153 226	153 226
sat14_atco_enc3_opt2_05_21	2 968 120	5 933 456
sat14_SAT_dat.k85-24_1_rule_3	2 712 808	5 395 102
sat14_SAT_dat.k75-24_1_rule_3	2 390 788	4 754 042

Table 5.1: Information about individual hypergraphs for both data sets.

matrices can be misleading, and repeatable because matrices are curated and made publicly available in many formats. Its matrices cover a wide spectrum of domains, including those arising from problems with underlying 2D or 3D geometry and those that typically do not have such geometry. [10] *International SAT Competition 2014* hypergraphs correspond to different Boolean satisfiability (SAT) problems used to test the performance of SAT solvers submitted by the participants.

5.2.2 Graphs for Edge Partitioning

Some additional experiments were conducted to test the edge (as in graphs, not hypergraphs) partitioning capabilities of our algorithms. For that, we used a set of 10 graphs with the corresponding file size of at most 250 MB for each. The file format in question stores for each node its weight and its neighbors with the weight of the corresponding edge followed by each neighbor.

To be able to partition graph edges via a hypergraph vertex partitioning algorithm, we first converted graphs into corresponding hypergraphs, and then simply swapped vertices and nets. It is possible because hypergraphs can be represented as bipartite graphs with

hypernodes stored as nodes in one block being in an N-to-N relationship with hyperedges stored as nodes in the other block, thus being equivalent in the hypergraph structure. Then, a node with m' outgoing or incoming edges in a graph becomes a hyperedge with m' pins in the resulting hypergraph, whereas each edge becomes a hypernode with 2 incident nets.

5.3 Fine-tuning hFennel with Sampling

As discussed in Section 4.1, there are some parameters, each with discrete values that need to be tuned. In this section, we do that by starting from a baseline setting with the number of random resolution attempts $d = 1$, sampling mode *all blocks*, and the number of samples $c = 8$. We then go through steps adjusting each parameter while keeping the other parameters constant. The best-found parameter value is then used in the subsequent experiments along with the previously found best values from all previous steps. We, therefore, hope to find the best combination of parameters, which will be used for the comparison with other algorithms.

5.3.1 Fine-tuning Random Resolution Attempts

The first parameter to be tuned is d, because the enforcement of a hard balancing constraint depends on it. If a low value for it is chosen, infeasible blocks might occur while using the random unavailable block resolution, and avoiding infeasible balance is the priority.
In order to test the validity of the hypothesis from Section 4.1.3 about the effect of d on the random unavailable block resolution, we conducted a simple preliminary experiment on a randomly generated graph with no edges with results shown in Figure 5.1. In the experiment, the random resolution tries to assign a weighted node $v \in V$ with $|V| = 1024$ with weights between 0 and 64 to one of 32 blocks. Only, the hard balancing constraint L_{max} was considered. With the optimal block weight being approximately 1008 and the maximum allowed block load being approximately 1038. In the case with $d = 1$, the resolution only receives 1 attempt per each v which is depleted immediately. Therefore, the hard balancing constraint is effectively ignored. In the case with $d = 2$, however, the resolution only receives $x+2$ attempts per each v: x left over from previous invocations and 2 because of the value of d. We can see that there is a sharp improvement in the partition balance. In fact, all blocks have a load less or equal to 1038, which satisfies the balancing constraint.
After seeing such a sharp increase in the hypothetical balance, one could assume that it is generally enough to use $d = 2$. However, that experiment only determined the validity of *a hypothesis about the effect of* d and not its *optimal value*. For the tuning, we look at three values of $d \in \{1, 2, 3\}$. The results of experiments on the tune set with these values are presented in Figure 5.2.
We can observe the following from the experiments: For all values of d the solution seems to have almost the same solution quality. The imbalance is close but below 0.03 for almost

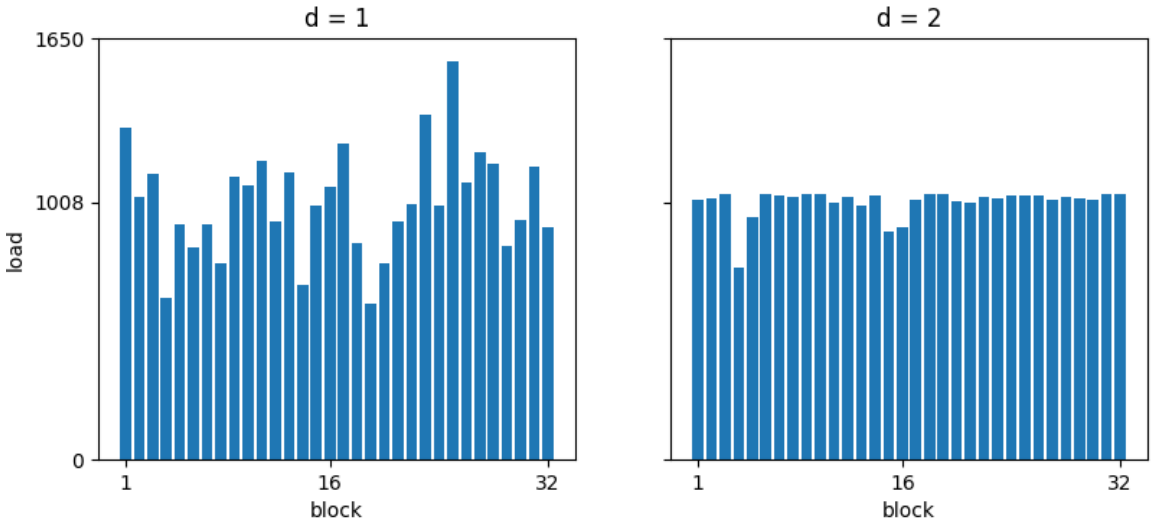

Figure 5.1: Block weight distribution for the random resolution with d attempts for $d \in \{1, 2\}$.

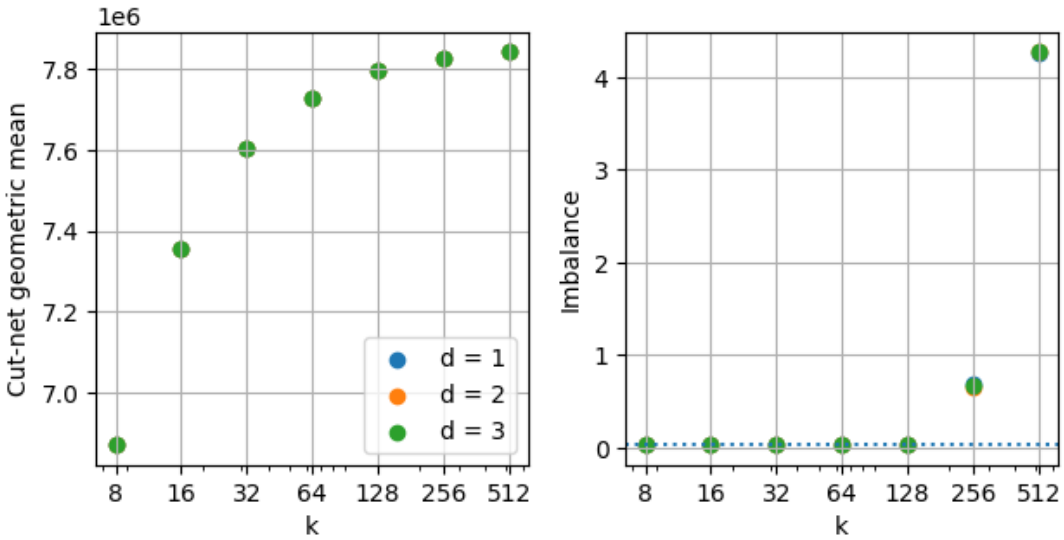

Figure 5.2: Effects of d on solution quality at different values of k. Results seem to be the same across values of d, causing an overlap.

all the experiments except at $k = 256, 512$. (A high imbalance indicator is caused by the hypergraphs ibm16, gupta3, and mono_500Hz, for which there is difficulty assigning multiple heavy nodes towards the end of the execution.) The lack of the effect of d can be explained by the fact that the hypergraphs used in the experiments (similar to hypergraphs used in the real world) have a large number of vertices. This causes a sufficient number of attempts to be accumulated over the earlier iterations of the algorithm. Therefore, it can be assumed that it is enough to use $d = 1$ for the majority of cases.

We still let the minimal value d be bounded by c outside the fine-tuning experiment described above. In other words, unless a value d higher than c is specified, the algorithm uses $d = c$. This ensures that the highest possible number of attempts is allowed without

the overall running time complexity of the algorithm becoming more than $O(nc)$. If $d > c$, a complexity of $O(nd)$ is possible in the worst-case scenario.

5.3.2 Choosing the Sampling Mode

The next parameter is the sampling mode, which is one of the following: all blocks, neighbors, non-neighbors, and twofold. It was decided to be the second parameter to be determined as the choices are limited to only four. The results are shown in Figure 5.3.

We can make the following observations from the charts. A clear outlier is the sampling mode all blocks. Although it has a clearly lower running time, its average cut-net result is very high in comparison to the other methods. Conversely, the non-neighbors method has the worst running time while yielding the lowest cut-net. Since we are trying to determine an optimal method with trade-offs, we exclude both previous methods as extremities. It leaves us with the methods neighbors and twofold. We choose to proceed with the method twofold even though it yields a higher average cut-net than method neighbors. The reason for it lies in the fact that the method neighbors tends to assign vertices to blocks consequently as the block choice is only limited to the neighboring blocks, using only the hard balancing constraint. Its cut-net advantage over twofold is, therefore, insignificant.

5.3.3 Finding the Best Proportion of Samples

The concluding configurable part of the algorithm with sampling is the actual amount of samples c to be picked. While working with different values of k, it might be difficult to judge the right absolute number of blocks to be sampled. Therefore, instead of determining the actual value of c, we design experiments to find the right fraction of c relative to k that would result in an optimal trade-off in the solution quality and the running time. Input hypergraphs have unweighted vertices. We present the results in Figure 5.4.

In the figures, it can be observed that the running time seems to be linearly dependent on c. It should be noted that the running time, although linearly dependent, still doesn't reduce by half for $c = \frac{k}{2}$. This can be explained by the fact that the algorithm in its current implementation needs time to load the whole hypergraph into the main memory and to do some additional steps for each vertex apart from looping to find the block with the highest Fennel score, like calculating the neighbor blocks, etc. However, the cut-net values decrease sharply between $c = \frac{k}{8}$ and $c = \frac{2k}{8}$ or $\frac{k}{4}$, while the decrease for the rest of the values is somewhat stable. This means the negative effect on cut-net is most insignificant relative to the running time improvement for values of c at around $\frac{c}{4}$. Therefore, we use $c = \frac{k}{4}$ in our further experiments. The average imbalance also seems to gradually decrease, staying below 0.03.

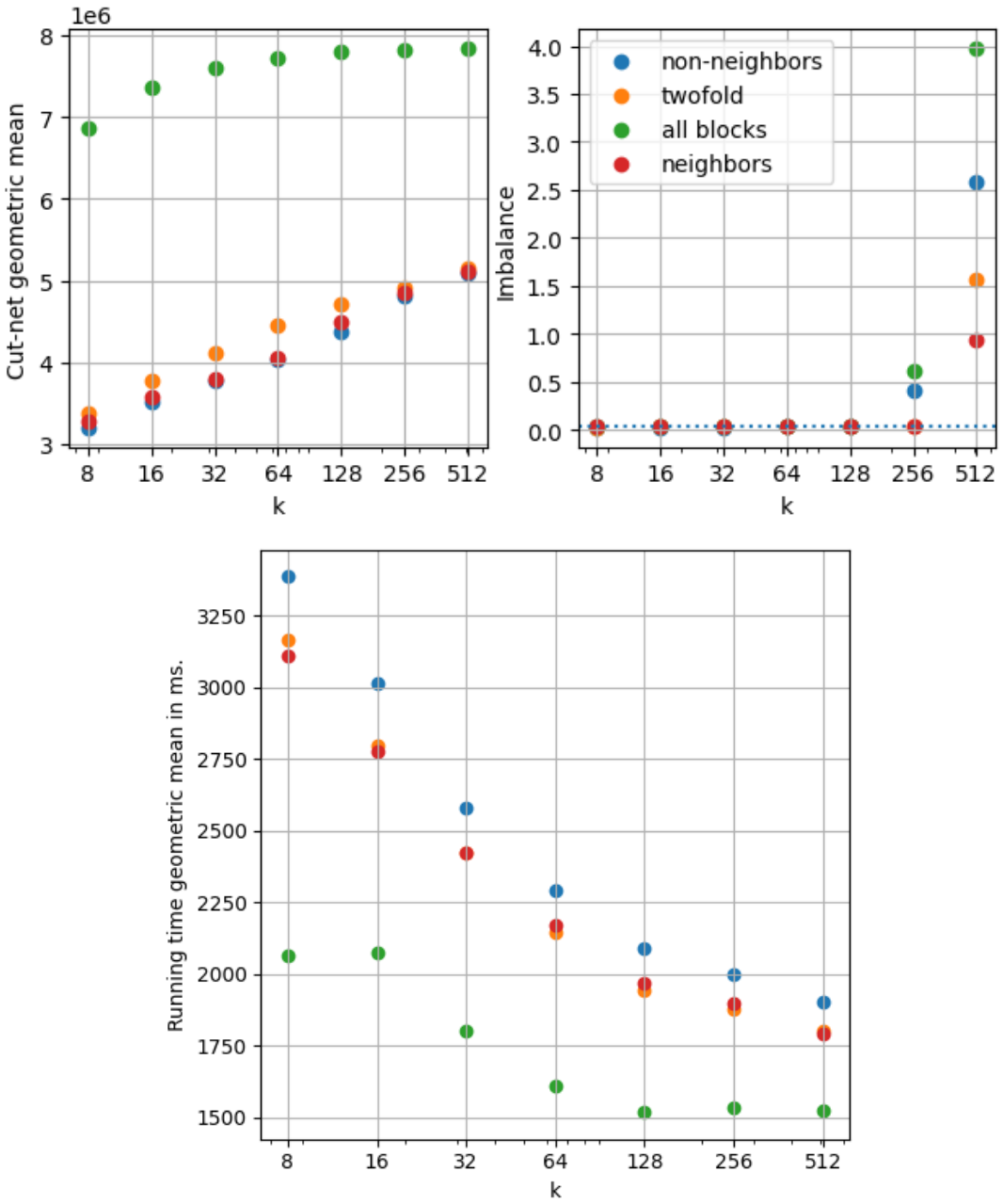

Figure 5.3: Effects of different sampling modes on solution quality and running time.

5.3.4 Choosing the Best Absolute Number of Samples

In some cases, it might be required that the amount of samples c remains constant and independent from k. In order to pick the best absolute value of c that might be used universally, we conduct another set of experiments with the difference being that instead of comparing the quality and performance at values $c \in \{\frac{k}{8}, \frac{2k}{8}, ..., k\}$, we compare them at values $c \in \{8, 16, 32, 64\}$. For the values $k = 32, 64$, we do not remove values of $c > k$ as those values of c still affect the solution quality due to the sampling algorithm we use which

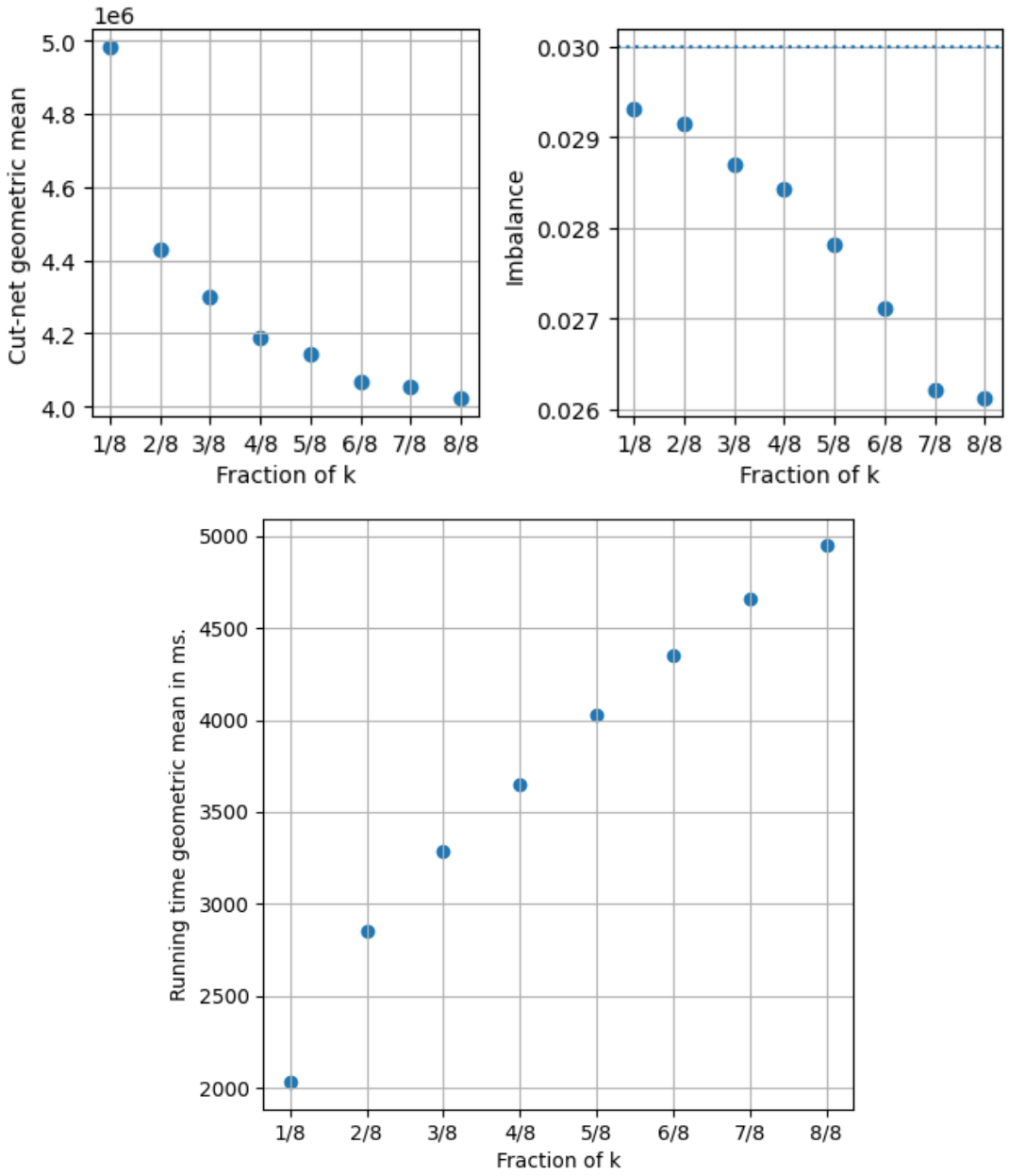

Figure 5.4: Effects of different $\frac{c}{k}$ values on solution quality and running time.

we describe in Section 4.1.2. Input hypergraphs have unweighted vertices. We present the results in Figure 5.5.

In this case, we can again notice two things. Namely, how the running time increases linearly, and that the imbalance is also high. While analyzing the cut-net chart, it can be observed that the least value of c at which a close to minimum cut-net is reached at 16. We use this value in our comparison experiments. The average imbalance, in this case, does not show a downward trend with the increasing c. However, for all experimented values $c > 8$, it is lower and roughly the same.

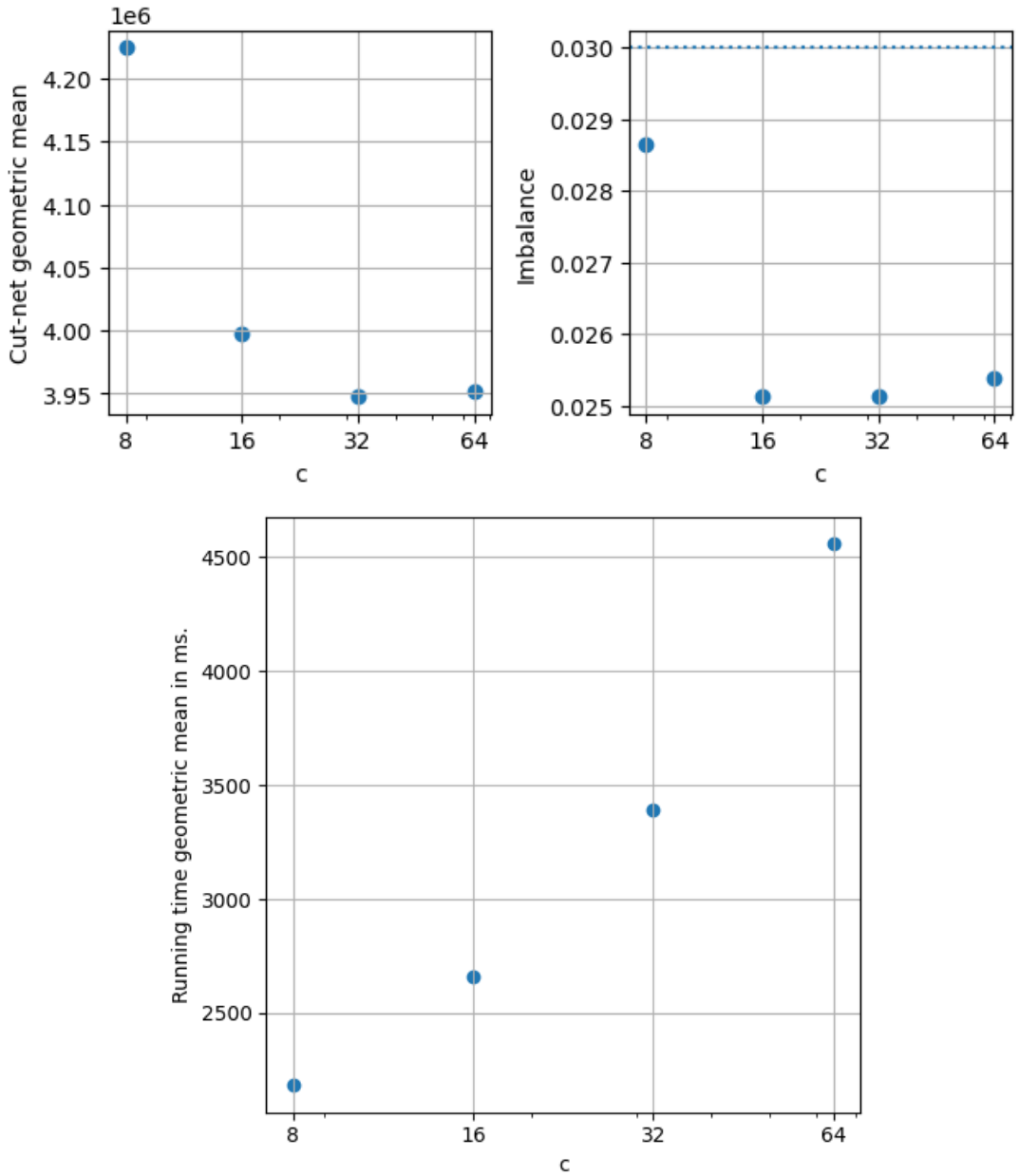

Figure 5.5: Effects of different c values on solution quality and running time.

5.4 Fine-tuning Buffered Hypergraph Partitioning

For the buffered algorithm described in Section 4.2, the only configurable parameter is α from the Equation 4.2. As discussed in that section, using the original value ϵ for a model may lead to an excessive amount of nets cut in order to keep the balance even for the earliest vertex batches, which is generally not necessary. Meanwhile, using the value ϵ' derived from L_{max} without any adjustment might yield an incorrigibly high imbalance in artificial vertices of models constructed for vertices received later. This necessitates finding

an optimal α between 0 and 1 in order to be able to prescribe an optimal allowed imbalance for models. We accomplish it experimentally by running our algorithm for values of buffer size σ in $\{4096, 65536, 1048576\}$, values of k in $\{8, 32, 128, 512\}$ and values of α in $\{0.00, 0.04, 0.08, 0.12, 0.16, 0.20\}$. We choose values $\alpha \leq 0.2$, because the preliminary experiments with values of alpha have yielded unacceptably high imbalance regardless of other parameters. For cut-net comparison, we use the percentual difference from the highest cut-net (or decrease) for each given σ instead of simply comparing cut-net values, because it better represents the effect of α at a given buffer size. The results are shown in Figure 5.6.

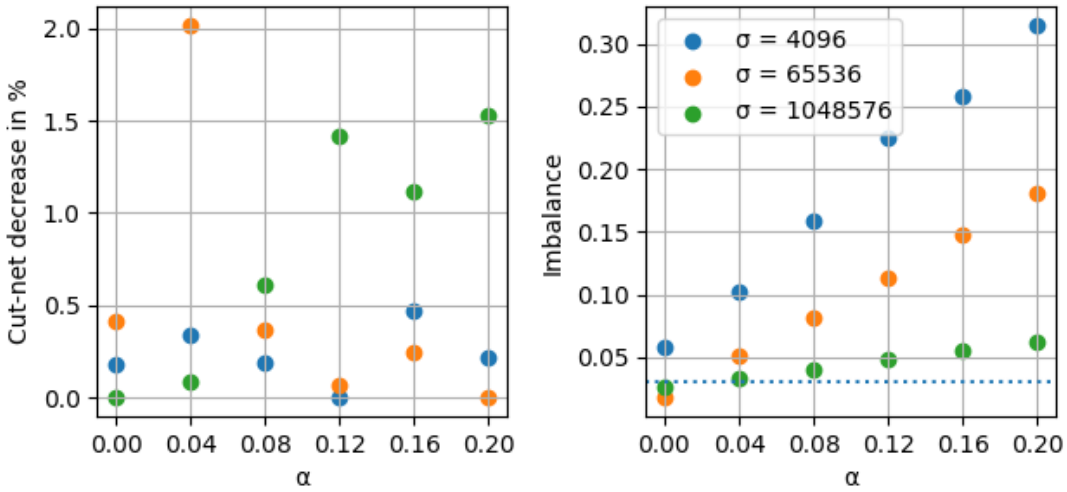

Figure 5.6: Effects of different α on solution quality.

In the figure, we can clearly see that the cut-net decreases for $\sigma = 1048576$. For the other two buffer size values, no significant changes can be observed, although preliminary experiments showed a decrease for values of $\alpha > 0.2$. This can be interpreted as the higher values of α loosening balance requirements and allowing for partitions with less cut-net. However, the imbalance chart shows that already at $\alpha = 0.04$, the prescribed imbalance for models is too high, allowing partitions with an overall imbalance of more than 0.03. It can also be observed that for the lower σ values imbalance grows faster with α than for the higher σ values. In fact, for $\sigma = 4096$, the imbalance at $\alpha = 0$ is already above 0.05. This is caused by KaHyPar struggling to assign as few as 4096 into as many as 512 blocks in hypergraphs such as gupta3.

5.5 Comparison of Algorithms

After having fine-tuned both our algorithms, we compare them with each other and some other streaming hypergraph partitioning algorithms. We present our findings and interpret them. We not only test our algorithms' hypergraph partitioning capabilities but also their applicability in graph edge partitioning.

5.5.1 Hypergraph Partitioning

Streaming algorithms. We experimentally compare different hypergraph partitioning heuristics, including ours, on two versions of the hypergraph test set. One version of the test set contains the original hypergraphs, which are unweighted. As described in Section 5.2.1, for the second version of the test set, we added some vertex and net weights. The results of our experiments are presented in Table 5.2 for unweighted hypergraphs and Table 5.3 for weighted hypergraphs. hFennel with $c = \frac{k}{4}$ proposed in this work, denoted as "Sample $\frac{k}{4}$", and with $c = 16$, denoted as "Sample 16", is compared with the following heuristics: Hash heuristic (used as a baseline), which assigns vertices to blocks based on the hash value of their ID while enforcing the L_{max} balancing constraint, Min-Max algorithm mentioned in Section 3.2.3, HYPE mentioned in Section 3.2.2 and hFennel without sampling. We also include the resulting average imbalance, the partition running time, and the average maximum amount of memory used. Min-Max and HYPE are excluded from experiments on weighted hypergraphs as they only support unweighted input.
The results show that despite yielding lower quality solutions than the hFennel algorithm without sampling, hFennel with sampling still manages to deliver lower cut-net than all other heuristics with the only exception being against HYPE at $k = 8$. Moreover, it is on average at least twice as fast as the original hFennel algorithm for hypergraphs. It delivers on average 20% more cut-net as the original hFennel and uses roughly the same amount of memory. Its resulting imbalance being closer to 0.03 implies that the algorithm might have a slight tendency towards heavy block assignment. The main advantage of the algorithm with sampling can be observed while comparing running time results over different values of k at a constant c (not shown in the tables). At all values of k, the running time stays roughly the same, indicating that the algorithm is indeed $O(nc)$ complex (or $O(n)$ if c is constant) instead of $O(nk)$.

Buffered algorithm. Using the same test set, we test our buffered streaming hypergraph partitioning algorithm. Due to the lack of buffered streaming hypergraph partitioning algorithms in general, we only compare the algorithm against the algorithm with sampling and $c = \frac{k}{4}$ that we use as a baseline, even though it is not a buffered algorithm. The algorithm is tested with the buffer size $\sigma \in \{2^{12}, 2^{16}, 2^{20}\}$. The results are presented in Table 5.4 for unweighted hypergraphs and Table 5.5 for weighted hypergraphs. We use the same statistics (resulting imbalance, running time, and memory usage) as for the non-buffered streaming algorithms.
Some rather unexpected results can be observed, especially with respect to $\sigma = 2^{12}$. The average cut-net for this buffer size is, in fact, larger than the one of the non-buffered algorithm. With growing buffer size, however, the cut-net decreases at a rate such that at $\sigma = 2^{16}$ the cut-net is already lower than that of the non-buffered algorithm. The imbalance for all buffered algorithms is lower, as the algorithm tries to partition each batch evenly, while the memory usage also grows with the buffer size, although the correlation doesn't seem to be linear. The memory usage being lower for the non-buffered algorithm

k	Hash	Min-Max	HYPE	hFennel	Sample $\frac{k}{4}$	Sample 16
8	690 420	500 402	282 279	168 121	305 307	174 950
16	710 819	563 578	328 238	222 641	327 454	237 536
32	714 003	596 210	364 371	286 390	323 056	298 606
64	726 205	616 333	397 641	335 443	336 670	336 671
128	729 166	631 550	432 345	378 403	374 279	381 455
256	730 535	645 662	467 876	424 991	422 931	430 314
512	731 093	657 128	503 161	466 050	464 774	466 478
Mean	718 756	599 308	389 758	309 088	361 063	316 749
	Statistics					
Imbalance	0,030 0	0,029 3	0,000 2	0,025 4	0,029 3	0,024 6
Time (ms)	1 170	24 465	5 916	5 178	2 454	2 155
Memory (MB)	651 375	343 964	974 500	649 347	649 323	649 698

Table 5.2: Performance comparison of different non-buffered streaming hypergraph partitioning algorithms on unweighted hypergraphs.

k	Hash	hFennel	Sample $\frac{k}{4}$	Sample 16
8	6 654 108	1 929 589	3 297 319	2 013 788
16	6 801 219	2 429 856	3 394 803	2 523 997
32	6 816 278	2 894 324	3 428 157	3 054 068
64	6 901 676	3 297 003	3 447 915	3 447 915
128	6 922 366	3 688 419	3 730 972	3 896 448
256	6 932 770	4 252 069	4 248 243	4 381 559
512	6 936 252	4 751 882	4 748 929	4 804 646
Mean	6 851 421	3 186 312	3 725 411	3 314 032
	Statistics			
Imbalance	0,030 0	0,027 0	0,029 9	0,026 7
Time (ms)	1 147	5 313	2 437	2 196
Memory (MB)	652 882	650 976	651 042	651 270

Table 5.3: Performance comparison of different non-buffered streaming hypergraph partitioning algorithms on weighted hypergraphs.

can be explained as being caused by our implementation of the non-buffered algorithm storing additional information about all edges of hypergraphs and can, therefore, be discarded, since, otherwise, it would have been roughly twice as little.

Due to the complexity of the model partitioning algorithm that we use, KaHyPar, it is difficult to pinpoint the exact reasons why the buffered algorithm yields poor results at $\sigma = 2^{12}$. However, some hypotheses can be made. As we have seen in Section 5.4 with hypergraphs such as gupta3, KaHyPar may struggle to partition a model for a buffer size as small as 4096, or 2^{12}. This not only causes a relatively high imbalance in general, but may also cause KaHyPar to try and repartition the hypergraph, or in this case, the model, yielding a higher cut-net during the process. This may be repeated multiple times. Such

k	Sample $\frac{k}{4}$	Buffered 2^{12}	Buffered 2^{16}	Buffered 2^{20}
8	305 307	324 817	140 690	33 761
16	327 454	387 477	184 558	50 875
32	323 056	461 846	223 960	67 070
64	336 670	502 379	274 383	89 490
128	374 279	528 361	326 457	120 913
256	422 931	547 917	379 254	162 892
512	464 774	569 653	437 406	206 475
Mean	361 063	466 609	262 730	88 321
	Statistics			
Imbalance	0,029 323	0,009 269	0,014 019	0,026 586
Time (ms)	2 454	233 090	133 507	161 144
Memory (MB)	649 323	566 874	786 718	1 620 781

Table 5.4: Performance of the buffered streaming hypergraph partitioning algorithm over different buffer sizes on unweighted hypergraphs.

k	Sample $\frac{k}{4}$	Buffered 2^{12}	Buffered 2^{16}	Buffered 2^{20}
8	3 297 319	3 252 631	1 483 478	376 199
16	3 394 803	3 889 431	1 933 520	552 896
32	3 428 157	4 454 358	2 342 519	762 976
64	3 447 915	4 929 579	2 848 683	998 748
128	3 730 972	5 251 441	3 381 624	1 372 988
256	4 248 243	5 500 155	3 915 826	1 838 394
512	4 748 929	5 759 282	4 411 556	2 365 046
Mean	3 725 411	4 638 645	2 725 836	992 127
	Statistics			
Imbalance	0,029 873	0,029 551	0,018 492	0,027 633
Time (ms)	2 437	281 670	157 290	211 942
Memory (MB)	651 042	572 306	802 891	1 692 505

Table 5.5: Performance of the buffered streaming hypergraph partitioning algorithm over different buffer sizes weighted hypergraphs.

repetition also may cause increased running time for each model, hence the issue with the running time for $\sigma = 2^{12}$. Another reason may lie within the treatment of fixed vertices by the recursive bipartitioning algorithm of KaHyPar. It works by removing fixed vertices from a hypergraph, partitioning the remaining vertices, and then adding fixed vertices back while trying different block permutations. The issue is in the fact that the main part of the algorithm excludes fixed vertices. When the buffer size decreases, the relative number of nets or connections to fixed in other words, already partitioned vertices increases for each vertex batch. Thus, more inter-batch nets are being cut, resulting in a higher overall cut-net. When it comes to the running time over buffer sizes, decreased buffer size does not result in a logarithmically, or yet alone, linearly better performance. The reason for this might

be the expensiveness of model construction, which uses sorting algorithms in order to sort nets by their pin set and merge them. It should be noted that although the performance improvement is not large, decreased buffer sizes still improve the overall performance to some extent.

5.5.2 Graph Edge Partitioning

In addition to testing our buffered and non-buffered algorithms for hypergraph partitioning, we test their solution quality and performance for the edge partitioning task. We compare our algorithms to the "Two-Phase" algorithm proposed by Mayer et al. [25], which is a non-buffered linear time out-of-core graph *edge* partitioning algorithm. For the Two-Phase algorithm, instead of using the cut-net metric, we count the number of vertices that have replicas in multiple edge blocks. For our algorithms, the resulting cut-net is equivalent to the number of replicated vertices in the source graph. We present our results in Tables 5.6 and 5.7.
The main thing to notice is how our algorithms deliver a lower number of replicated vertices. However, the Two-Phase algorithm can be considered better suited for the edge partitioning task as it delivers highly balanced solutions for an order of magnitude less average running time while using less memory.

k	hFennel	Sample $\frac{k}{4}$	Sample 16	Two-Phase
32	622 390	622 377	619 177	2 359 112
128	631 051	628 413	628 413	2 408 213
Mean	626 706	625 388	623 778	2 383 536
	Statistics			
Imbalance	0,001 404	0,015 120	0,001 549	0,000 000 1
Time (ms)	54 812	13 908	22 829	2 821
Memory (MB)	2 473 932	2 473 846	2 473 937	422 134

Table 5.6: Performance comparison of different non-buffered streaming algorithms in edge partitioning.

k	Two-Phase	Buffered 2^{12}	Buffered 2^{16}	Buffered 2^{20}
32	2 359 112	770 625	590 891	356 621
128	2 408 213	854 214	695 619	427 561
Mean	2 383 536	811 344	641 120	390 483
	Statistics			
Imbalance	0,000 000 1	0,000 420	0,001 070	0,006 250
Time (ms)	2 821	1 935 008	588 137	882 524
Memory (MB)	422 134	2 350 136	2 351 263	3 957 978

Table 5.7: Performance comparison of different buffered streaming algorithms in edge partitioning.

CHAPTER 6

Discussion

6.1 Conclusion

In this work, we proposed and experimented with two algorithms. One of them was based on an online heuristic for hypergraphs proposed by us previously, and the other one was a prototype buffered streaming hypergraph partitioning algorithm. We fine-tuned their parameters and compared them to other similar partitioning algorithms.
For hFennel with sampling, we determined optimal parameters such as the number and proportion of sampled blocks, the sampling mode, etc. We first explained our hypothesis regarding whether and how sampling might yield balanced partitions with little gain in cut-net and within a smaller amount of time. The algorithm with sampling proved to be very efficient and, in some cases, capable of delivering roughly the same solution quality as hFennel as measured by the cut-net metric while being remarkably faster, even though there were a few exceptions to that while working with hypergraphs from the tune set (See Section 5.3). It furthermore proved to result in a lower number of replicated vertices while being used for the edge partitioning task than a specialized edge partitioner.
We also tried to find the best setting for the buffered hypergraph partitioner. Although there is some limited success in cut-net reduction or a reduced running time and memory consumption in some cases, a part of the experiments, especially for low buffer size, did not yield expected improvements, and the running time did not decrease significantly with the buffer size. The issues regarding the solution quality seemed to be caused by the internal usage of KaHyPar, which was not intended to be used as a part of a buffered algorithm. Performance-related issues were primarily caused by our use of KaHyPar libraries in order to load hypergraphs and construct models. Overall, the performance and solution quality of the buffered algorithm can not be considered conclusive as algorithms and libraries specifically developed for buffered streaming partitioning are yet to be tested with the algorithm.

6.2 Future Work

Our work conducted on streaming hypergraph partitioning algorithms and the algorithms we propose can be considered as yet a step in the field. While attempting to answer some questions, some further questions arose which can, perhaps, be answered by some future work proposed in this section.

Weighted edgelist. In our experiments, input hypergraphs were loaded via KaHyPar framework. It uses the *hMetis* [20] input format. In this format, the first m lines store the weight and pin IDs for each net followed by n lines with vertex weights. Thus, the whole hypergraph must be loaded before the actual partitioning starts. A format similar to or based on the edgelist format can be developed in order to truly be able to receive a weighted hypergraph in a stream. The edgelist format used by algorithms like MinMax is for the unweighted hypergraphs. The weighted version of it may, for example, have the weight after each vertex and net mention. For example:

```
4 73: 1 37, 2 84
2 98: 1 37
5 49: 2 84, 3 79, 52
```

This would be a hypergraph with vertex 4 of weight 73 connected to nets 1 of weight 37 and 2 of weight 84.

HeiStream for hypergraphs. Our prototype buffered algorithm uses KaHyPar to partition models. A Fennel-based multilevel algorithm can be employed for model partitioning in order to accelerate our buffered partitioning algorithm and perhaps improve its solution quality. In light of the related work, an obvious example of such an algorithm is HeiStream. It employs a heuristic called *Multilevel Fennel* in order to partition models. Since HeiStream as described by Faraj and Schulz in [13] uses L_{max} throughout the algorithm, there is no need to prescribe permitted imbalance for each model. On the software side, it can be accomplished in two possible ways: On one hand, KaHyPar modules can be expanded, adding a coarsener like the one used in HeiStream, a modified version of hFennel that we previously proposed as an initial partitioning algorithm, and another modified version of hFennel for hypergraphs as a refiner. Model construction from our prototype may be reused. On the other hand, HeiStream itself should be expanded to be able to process not just edges but also hyperedges.
Some other changes may be applied to our prototype algorithm as well. One of them is a construction of an *extended model*, which not only includes artificial vertices created from previously partitioned vertices and buffer vertices but also includes some *ghost vertices*. These are vertices that are neither in the buffer nor have been previously partitioned, in other words, "future" vertices. [13] Faraj and Schulz also extended HeiStream to operate in a *restreaming* setting, which makes over the same graph with a modified algorithm. This can hypothetically be applied to hypergraph partitioning as well.

Other metrics. In our work, we used cut-net minimization as the main objective. In some contexts, other metrics are used to measure the solution quality. For example, *connectivity minus* 1 objective not only considers the cut nets but also the number of times a net is cut, in other words, their connectivity with 1 subtracted from it. For the hFennel algorithm for hypergraphs proposed by us previously, minimization of connectivity minus 1 can be theoretically achieved by using the net criterion *touching* instead of *intermediate* described in Section 3.2.4 as it accepts nets that have already been cut during execution.
During analysis, the geometric mean over different hypergraphs is used in order to determine an "average" absolute cut-net for an algorithm. This is done because total hypergraph net weights and hence the resulting cut-net values differed in orders of magnitude, making large cut-net values dominant if the arithmetic mean was to be used. The result is vulnerable to zero values and cannot be used to make a judgment unless compared to that of other algorithms. An alternative would be to use proportions of cut-net values relative to total hypergraph net weights $\frac{\omega(cut(E))}{\omega(E)}$ where $cut(E)$ denotes the set of cut nets. The resulting value is always between 0 and 1 inclusively. Thus, using an arithmetic average in this context is not only feasible in regard to unbiasedness but also informative of overall algorithm solution quality.

Zusammenfassung

Die Hypergraphpartitionierung wird bei an Größe zunehmenden realen Hypergraphen immer schwieriger. Strömende Algorithmen sind ein aktueller Trend, die dafür da sind, immer größere Hypergraphen in angemessener Zeit anzugehen. Allerdings wurde bislang wenig Aufwand für strömende Hypergraphpartitionierungsalgorithmen betrieben. In unserer vorherigen Arbeit haben wir einen gewichteten Hypergraphpartitionierungsalgorithmus vorgeschlagen. Insbesondere haben wir einen erfolgreichen strömenden Algorithmus an den Bereich der Hypergraphen angepasst. Wir haben sorgfältig alle Details dazu entwickelt, um seine Leistung in der Praxis zu optimieren, und es experimentiell abgestimmt, und wir zeigen, dass es Partitionen mit durchschnittlich ca. 15% weniger Hyperkantenschnitt ergibt als das derzeitig modernste strömende Hypergraphpartitionierungsalgorithmus. Bei dieser Arbeit fahren wir fort und führen die Nutzung der Probenahme vor, um den Algorithmus weiter zu beschleunigen ohne einen signifikanten Qualitätsverlust der Lösung zu verursachen. Nebenbei schlagen wir einen gepufferten Ansatz für Hypergraphpartitionierung, um noch höhere Lösungsqualitäten bei der Nutzung regelbarer Arbeitsspeichermengen zu erhalten. Wir experimentieren außerdem mit der Nutzung unseres Algorithmus der Graphkantenpartitionierung, weil sich diese der Hypergraphknotenpartitionierung äquivalent erweist.

Bibliography

[1] Dan Alistarh, Jennifer Iglesias, and Milan Vojnovic. Streaming Min-max Hypergraph Partitioning. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 1900–1908, 2015. https://proceedings.neurips.cc/paper/2015/hash/83f97f4825290be4cb794ec6a234595f-Abstract.html.

[2] 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. https://doi.org/10.1145/274535.274546.

[3] Reid Andersen and Kevin J. Lang. An Algorithm for Improving Graph Partitions. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 651–660. SIAM, 2008. http://dl.acm.org/citation.cfm?id=1347082.1347154.

[4] C.E. Bichot and P. Siarry. *Graph Partitioning*. ISTE. Wiley, 2013. https://books.google.de/books?id=KUHLscW8D2cC.

[5] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008. https://doi.org/10.1109/TKDE.2007.190689.

[6] Thang Nguyen Bui and Curt Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992. https://doi.org/10.1016/0020-0190(92)90140-Q.

[7] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In Lasse Kliemann and Peter

Sanders, editors, *Algorithm Engineering: Selected Results and Surveys*, pages 117–158. Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-49487-6_4.

[8] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel Distributed Systems*, 10(7):673–693, 1999. https://doi.org/10.1109/71.780863.

[9] 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. https://doi.org/10.1007/978-0-387-09766-4_93.

[10] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011. https://doi.org/10.1145/2049662.2049663.

[11] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Çatalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006. https://doi.org/10.1109/IPDPS.2006.1639359.

[12] Kamal Eyubov. Streaming Hypergraph Partitioning. Practical report, 2021.

[13] Marcelo Fonseca Faraj and Christian Schulz. Buffered Streaming Graph Partitioning. *ACM Journal of Experimental Algorithmics*, jun 2022. https://doi.org/10.1145/3546911.

[14] Marcelo Fonseca Faraj, Alexander van der Grinten, Henning Meyerhenke, Jesper Larsson Träff, and Christian Schulz. High-Quality Hierarchical Process Mapping. In Simone Faro and Domenico Cantone, editors, *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*, volume 160 of *LIPIcs*, pages 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. https://doi.org/10.4230/LIPIcs.SEA.2020.4.

[15] Charles M. Fiduccia and Robert M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In James S. Crabbe, Charles E. Radke, and Hillel Ofek, editors, *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*, pages 175–181. ACM/IEEE, 1982. https://doi.org/10.1145/800263.809204.

[16] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some Simplified NP-Complete Graph Problems. *Theoretical Computer Science*, 1(3):237–267, 1976. https://doi.org/10.1016/0304-3975(76)90059-1.

[17] Michael Hamann and Ben Strasser. Graph Bisection with Pareto Optimization. *ACM Journal of Experimental Algorithmics*, 23, 2018. https://doi.org/10.1145/3173045.

[18] Shixun Huang, Yuchen Li, Zhifeng Bao, and Zhao Li. Towards Efficient Motif-based Graph Partitioning: An Adaptive Sampling Approach. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 528–539. IEEE, 2021. https://doi.org/10.1109/ICDE51399.2021.00052.

[19] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration Systems*, 7(1):69–79, 1999. https://doi.org/10.1109/92.748202.

[20] George Karypis and Vipin Kumar. hMETIS*: A Hypergraph Partitioning Package. https://course.ece.cmu.edu/~ee760/760docs/hMetisManual.pdf, 1998.

[21] Donald Ervin Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998. https://www.worldcat.org/oclc/312898417.

[22] Kim-Hung Li. Reservoir-Sampling Algorithms of Time Complexity O(n(1 + Log(N/n))). *ACM Transactions on Mathematical Software*, 20(4):481–493, dec 1994. https://doi.org/10.1145/198429.198435.

[23] Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion. In Naoki Abe, Huan Liu, Calton Pu, Xiaohua Hu, Nesreen K. Ahmed, Mu Qiao, Yang Song, Donald Kossmann, Bing Liu, Kisung Lee, Jiliang Tang, Jingrui He, and Jeffrey S. Saltz, editors, *IEEE International Conference on Big Data (IEEE BigData 2018), Seattle, WA, USA, December 10-13, 2018*, pages 458–467. IEEE, 2018. https://doi.org/10.1109/BigData.2018.8621968.

[24] 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 *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pages 685–695. IEEE Computer Society, 2018. https://doi.org/10.1109/ICDCS.2018.00072.

[25] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. Out-of-Core Edge Partitioning at Linear Run-Time. In *38th IEEE International Conference on Data Engi-*

neering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022, pages 2629–2642. IEEE, 2022. https://doi.org/10.1109/ICDE53745.2022.00242.

[26] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning Complex Networks via Size-Constrained Clustering. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 2014. https://doi.org/10.1007/978-3-319-07959-2_30.

[27] Adnan El Moussawi, Ricardo Rojas Ruiz, and Nacéra Bennacer Seghouani. Sampling-based Label Propagation for Balanced Graph Partitioning. In Dan Feng, Steffen Becker, Nikolas Herbst, and Philipp Leitner, editors, *ICPE '22: ACM/SPEC International Conference on Performance Engineering, Bejing, China, April 9 - 13, 2022*, pages 223–230. ACM, 2022. https://doi.org/10.1145/3489525.3511698.

[28] David A. Papa and Igor L. Markov. Hypergraph Partitioning and Clustering. In Teofilo F. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007. https://doi.org/10.1201/9781420010749.ch61.

[29] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. HDRF: Stream-Based Partitioning for Power-Law Graphs. In James Bailey, Alistair Moffat, Charu C. Aggarwal, Maarten de Rijke, Ravi Kumar, Vanessa Murdock, Timos K. Sellis, and Jeffrey Xu Yu, editors, *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*, pages 243–252. ACM, 2015. https://doi.org/10.1145/2806416.2806424.

[30] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-scale Networks. *Physical Review E*, 76:036106, Sep 2007. https://doi.org/10.1103/PhysRevE.76.036106.

[31] Hooman Peiro Sajjad, Amir H. Payberah, Fatemeh Rahimian, Vladimir Vlassov, and Seif Haridi. Boosting Vertex-Cut Partitioning for Streaming Graphs. In Calton Pu, Geoffrey C. Fox, and Ernesto Damiani, editors, *2016 IEEE International Congress on Big Data, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 1–8. IEEE Computer Society, 2016. https://doi.org/10.1109/BigDataCongress.2016.10.

[32] Peter Sanders and Christian Schulz. KaHIP – Karlsruhe High Qualtity Partitioning Homepage. http://algo2.iti.kit.edu/documents/kahip/index.html.

[33] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. *k*-way Hypergraph Partitioning via *n*-Level Recursive Bisection. In Michael T. Goodrich and Michael Mitzenmacher, editors, *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 53–67. SIAM, 2016. https://doi.org/10.1137/1.9781611974317.5.

[34] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-Quality Hypergraph Partitioning. *ACM Journal of Experimental Algorithmics*, mar 2022. Just Accepted. https://doi.org/10.1145/3529090.

[35] Christian Schulz and Darren Strash. Graph Partitioning: Formulations and Applications to Big Data. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019. https://doi.org/10.1007/978-3-319-63962-8_312-2.

[36] Isabelle Stanton and Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In Qiang Yang, Deepak Agarwal, and Jian Pei, editors, *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, pages 1222–1230. ACM, 2012. https://doi.org/10.1145/2339530.2339722.

[37] Aleksandar Trifunovic and William J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563–581, 2008. https://doi.org/10.1016/j.jpdc.2007.11.002.

[38] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In Ben Carterette, Fernando Diaz, Carlos Castillo, and Donald Metzler, editors, *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 333–342. ACM, 2014. https://doi.org/10.1145/2556195.2556213.

[39] Brendan Vastenhouw and Rob H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005. https://doi.org/10.1137/S0036144502409019.

[40] Jeffrey Scott Vitter. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985. https://doi.org/10.1145/3147.3165.

[41] Yen-Chuen A. Wei and Chung-Kuan Cheng. Towards Efficient Hierarchical Designs by Ratio Cut Partitioning. In *1989 IEEE International Conference on Computer-Aided Design, ICCAD 1989, Santa Clara, CA, USA, November 5-9, 1989. Digest of*

Technical Papers, pages 298–301. IEEE Computer Society, 1989. https://doi.org/10.1109/ICCAD.1989.76957.