Karlsruhe Institute of Technology

Label Propagation for Hypergraph Partitioning

Master's Thesis
by

Vitali Henne

Institute of Theoretical Informatics, Algorithmics
Department of Informatics

Advisors: Prof. Dr. Peter Sanders
Juniorprof. Dr. Henning Meyerhenke
Dr. Christian Schulz
M.Sc. Sebastian Schlag

Date of submission: 05.05.2015

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht, und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 05.05.2015

Zusammenfassung

Viele Probleme in der Informatik lassen sich auf eine Partitionierung oder eine Clusterung eines Graphen reduzieren. Gemäß der klassischen Definition, besteht Graph aus Knoten und Kanten, wobei Kanten exakt zwei Knoten verbinden. Hypergraphen heben diese Beschränkung auf und erlauben es, dass Kanten beliebig viele Knoten verbinden. Aktuelle Ergebnisse motivieren den Einsatz von Hypergraphen zur Modellierung von Problemen in der Informatik, da diese sich mithilfe von Hypergraphen besser und intuitiver repräsentieren lassen. Diese Arbeit generalisiert Label-Propagation, einen Graphenclusteringalgorithmus, an Hypergraphpartitionierung. Wir schlagen drei Varianten von Label-Propagation vor, die durch graphenbasierte Hypergraphmodellierung motiviert sind. Diese evaluieren wir als Clusteringalgorithmen in der Coarseningphase der Multilevel-Partitioning-Heuristik. Desweitern benutzen wir Label Propagation in der Uncoarsening- und Refinementphase als schnellen Lokale-Suche-Algorithmus. Wir vergleichen unsere Algorithmen mit den Hypergraphpartitionierern hMetis und PaToH, die dem aktuellen Stand der Technik entsprechen. Unsere Algorithmen erreichen die besten Ergebnisse für größeres k auf einem VLSI Benchmark: für $k = 128$ haben die durch unsere Algorithmen berechneten Partitionen 2% weniger Schnittkanten als die Partitionen von hMetis und 4% weniger Schnittkanten als die Partitionen von PaToH.

Abstract

Many problems in computer science can be represented by a graph and reduced to a graph clustering or k-way partitioning problem. In the classical definition, a graph consists of nodes and edges which usually connect exactly two nodes. Hypergraphs are a generalization of graphs, where every edge can connect an arbitrary number of nodes. Recent results suggest that some problems in computer science are better and more intuitively modeled with hypergraphs instead of graphs. This thesis investigates the adaptation of label propagation, a graph clustering algorithm, to hypergraph partitioning. We propose three adaptations of label propagation which are motivated by graph-based hypergraph modeling and evaluate them as coarsening strategies in a direct k-way multilevel hypergraph partitioning framework. Furthermore, we propose a greedy local search algorithm inspired by label propagation for the uncoarsening and refinement phase of the multilevel partitioning heuristic. We compare our algorithms to the state-of-the-art hypergraph partitioners hMetis and PaToH. Our results imply that the utilization of label propagation in the multilevel hypergraph partitioning scheme is promising, as we outperform both hMetis and PaToH on VLSI instances for larger values of k: for $k = 128$ our proposed algorithms produce 2% better cuts than hMetis and 4% better cuts than PaToH.

Contents

1 Introduction

Hypergraphs are a generalized form of graphs where an edge can connect more than two nodes. Since every graph can be modeled as a hypergraph, every application of graphs is also an application of hypergraphs. In addition, Estrada and Rodríguez-Velázquez [20] present examples for social, biological, ecological, and technological systems, where normal graphs are unable to represent the structural information of the system. They show that hypergraphs can be used for a better representation of those systems. Moreover, hypergraphs allow an easy and intuitive modeling of various problems in computer science which include computer vision [2], machine learning [1,61], data analysis [46], circuit layout [7], and scientific computing [18].

All of these problems can be (partially) reduced to a clustering or k-way partitioning problem of a hypergraph. Given a hypergraph, a k-way partition is an assignment of the nodes to k disjoint, nonempty sets, called blocks. Usually, we seek a k-way partition that optimizes an objective. A widely used objective is the hypergraph cut – the sum of hyperedge weights which span multiple blocks. Usually, constraints are imposed on the partition: for example, the predominant constraint is the balance constraint, which demands balanced blocks, i.e. the weight of an arbitrary block must not differ greatly from the weight of other blocks. For most objectives it is shown that if this constraint is enforced, the k-way partitioning problem becomes NP-complete for both hypergraphs and graphs [22, 37]. The clustering problem can be seen as a partitioning problem, where k, the number of blocks, is unknown beforehand. Clustering problems usually impose different constraints and optimize different objectives than graph partitioning.

The arguably most successful heuristic for the k-way partitioning problem is called the multilevel partitioning scheme [13, 27]. It is used by both state-of-the-art graph and hypergraph partitioners. On an abstract level, the scheme can be divided into three phases: coarsening phase, initial partitioning phase, and uncoarsening phase. First, the (hyper)graph is successively contracted in the coarsening phase, until it is small enough to be feasibly partitioned in the initial partitioning phase. The employed coarsening strategy has a large impact on the overall quality of the partition. If the smaller graphs don't exhibit similar structural properties as the original graph, the quality of the overall partition will decline. In the uncoarsening and refinement phase the contraction is reversed and the initial partition is projected to the next finer graph. During this phase a local search algorithm is applied, which improves the quality of the partition induced by the coarser graph.

1.1. Problem Statement

Label propagation [47] is a well known graph clustering algorithm which operates in passes, each of which has a linear running time complexity. A constant number

of iterations suffices for a nearly converged, good solution [47]. Therefore, label propagation has a near-linear time complexity on graphs. Recent studies show that label propagation can be successfully used within a multilevel graph partitioning algorithm: Meyerhenke et al. [41] utilize the algorithm as a coarsening strategy and a fast local search strategy. Their results imply that label propagation has much potential on very large, irregular graphs, like social networks. The goal of this thesis is the generalization of label propagation to hypergraphs while still conserving the near-linear time complexity of the algorithm. We focus thereby on the hypergraph k-way partitioning problem. Like Meyerhenke et al. [41], we utilize label propagation both as a coarsening strategy as well as a local search strategy in the multilevel partitioning heuristic.

1.2. Contributions

We propose and evaluate various adaptions of label propagation as a coarsening strategy which are motivated by graph-based hypergraph modeling. We consider the two most common transformations of hypergraphs to graphs. Utilized in label propagation, one results in non-linear running time and good solution quality whereas the other results in linear running time but worse solution quality. In addition, we propose a probabilistic version of label propagation which has a linear running time and produces good results. Furthermore, we develop a fast and greedy local search algorithm which is inspired by label propagation. We propose three configurations (LPFast, LPEco, LPBest) of a direct k-way hypergraph partitioning framework called KaHyPar [28] which utilize these algorithms. We compare these configurations on various hypergraphs originating from very large scale integration (VLSI) problems and sparse matrices (SPM), and compare ourselves to the prominent state-of-the-art hypergraph partitioners, hMetis [32] and PaToH [18]. Our algorithms perform especially well for larger k on VLSI instances. For $k \geq 64$ both LPEco and LPBest outperform hMetis and PaToH on a VLSI benchmark in terms of solution quality. Furthermore, LPEco dominates hMetis in both partition quality and running time on this benchmark set for $k = 128$: LPEco computes 7% better cuts and is 2.5 times faster than the direct k-way variant of hMetis and computes 2% better cuts and is 1.5 times faster than the recursive bisection variant of hMetis. In case of SPM instances, our algorithms produce better partitions than the direct k-way variant of hMetis and the default preset of PaToH for $k \geq 64$, but are outperformed by the recursive bisection variant of hMetis and the quality preset of PaToH.

1.3. Outline

The thesis is organized as follows: Chapter 2 covers basic hypergraph related notations and formally defines the hypergraph k-way partitioning problem. Chapter 3 provides an overview over related work. Chapter 4 describes our modifications to the label propagation algorithm for it to be applicable to hypergraphs. Chapter 5 gives insight on the implementation details. Chapter 6 evaluates our algorithm and presents a comparison with other state-of-the-art algorithms. Finally, Chapter 7 gives a conclusion and provides an outlook for future work.

2 | Preliminaries

This chapter introduces basic concepts and terminology used throughout this thesis. Furthermore, the most prominent partitioning objectives and transformations from hypergraphs to graphs are introduced in the latter part of this chapter.

2.1. Definitions and Terminology

Definition 2.1.1 (Hypergraph)**.** Formally, an *unweighted hypergraph* [14] is defined as a tuple $\mathcal{H} = (V, E)$ with

- a set of *hypernodes*, i.e. $V = \{v_1, \ldots, v_n\}$, $|V| = n$,
- a set of *hyperedges*, i.e. $E = \{e_1, \ldots, e_m\}$, $|E| = m$, $\forall e \in E : \emptyset \neq e \subseteq V$.

In literature, hyperedges are also referred to as *nets*. In case of *weighted hypergraphs*, we extend the tuple with two weight functions, i.e. $\mathcal{H} = (V, E, c, \omega)$ with:

- *node weights*, i.e. $c : V \to \mathbb{R}_{>0}$,
- *edge weights*, i.e. $\omega : E \to \mathbb{R}_{>0}$.

The *weight* of a hypernode $v \in V$ is $c(v)$ and the *weight* of a hyperedge $e \in E$ is $\omega(e)$. We extend $c(\cdot)$ and $\omega(\cdot)$ to sets, i.e.:

$$c(V') := \sum_{v \in V'} c(v) \quad \text{and} \quad \omega(E') := \sum_{e \in E'} \omega(e).$$

Note that each unweighted hypergraph $\mathcal{H} = (V, E)$ can be transformed to a weighted hypergraph $\mathcal{H}' = (V, E, c, \omega)$ with $c \equiv 1$ and $\omega \equiv 1$. For the sake of generality, we will only consider weighted hypergraphs.

Two hypernodes $v_i, v_j \in V$ are *adjacent* iff there exists a hyperedge which contains them both:

$$\exists e \in E : v_i \in e \wedge v_j \in e.$$

A hyperedge $e \in E$ is *incident* to a hypernode $v \in V$, iff v is present in e:

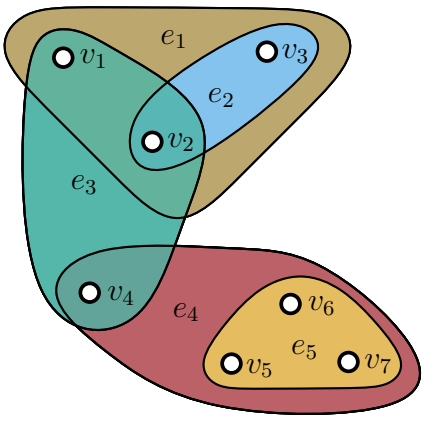
$$v \in e.$$

Given a hyperedge $e \in E$, the elements in e are called *pins*. In this thesis we use $\mathit{pins}[e]$ for the pins of the hyperedge e and $\mathit{hyperedges}[v]$ for the incident hyperedges of hypernode $v \in V$. We extend $\mathit{hyperedges}[\cdot]$ and $\mathit{pins}[\cdot]$ to sets, i.e.:

$$\mathit{hyperedges}[V'] := \{\mathit{hyperedges}[v] \mid v \in V'\} \quad \text{and} \quad \mathit{pins}[E'] := \{\mathit{pins}[e] \mid e \in E'\}$$

For a given hypernode v, we define its *degree* as the number of hyperedges it is incident to:

$$\mathit{deg}(v) := |\{e \in E \mid v \in e\}|.$$



$$
\begin{aligned}
\mathcal{H} &= (V, E, c, \omega) \\
V &= \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\} \\
E &= \{e_1 = \{v_1, v_2, v_3\}, \\
&\quad\ e_2 = \{v_2, v_3\}, \\
&\quad\ e_3 = \{v_1, v_2, v_4\}, \\
&\quad\ e_4 = \{v_4, v_5, v_6, v_7\}, \\
&\quad\ e_5 = \{v_5, v_6, v_7\}\} \\
c &\equiv 1 \\
\omega &\equiv 1
\end{aligned}
$$

Figure 2.1.: *An example hypergraph. Hypernodes are depicted as circles and hyperedges are illustrated as different colored areas. All hypernodes and all hyperedges have weight one. For the sake of simplicity, hypernodes and hyperedges are not annotated with their respective weight.*

Analogously, the *size* or *cardinality* of a hyperedge $e \in E$ is the number of its pins: $|e|$. Figure 2.1 shows an example hypergraph. It has seven hypernodes and five hyperedges, all having weight one. Hypernodes are depicted as circles and hyperedges are illustrated as different colored areas.

Definition 2.1.2 (Contracting hypernodes)**.** Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ and two hypernodes $v, u \in V$ a contraction of v and u merges the two hypernodes into a single hypernode v', with $c(v') := c(v) + c(u)$ and replaces v and u with v' in all hyperedges. Formally, if we contract v and u the resulting hypergraph $\mathcal{H}' = (V', E', c', \omega)$ is defined as

$$
\begin{aligned}
V' :=& (V \cup \{v'\}) \setminus \{v, u\} \\
E' :=& \{e \in E \mid v, u \notin e\} \cup \\
& \{(e \setminus \{v\}) \cup \{v'\} \mid e \in E, v \in e\} \cup \\
& \{(e \setminus \{u\}) \cup \{v'\} \mid e \in E, u \in e\}
\end{aligned}
$$

$$
c(w)' := \begin{cases} c(w) & \text{if } w \notin \{v, u\} \\ c(v) + c(u) & \text{if } w = v'. \end{cases}
$$

Note that through contractions the formation of *parallel* hyperedges, i.e. hyperedges containing the same pins, is possible. Usually, these parallel hyperedges get merged into a single hyperedge which has the accumulative weight of all parallel hyperedges. For the sake of simpler definitions we won't merge parallel hyperedges.

We extend the concept of hypernode contraction to hypernode sets by selecting a representative u from the set, and successively contracting every other hypernode v from the set with u. Note that with this extension we also define hyperedge contraction, since a hyperedge is a set of hypernodes.

Definition 2.1.3 (Hypergraph k-way partitioning problem)**.** Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ and $k > 1$, a k-way *partition* of $\mathcal{H}$ is a set $\Pi = \{V_1, \ldots, V_k\}$ of *blocks* with:

- $\bigcup_{i=1}^{k} V_i = V$,
- $V_i \subseteq V$ and $V_i \neq \emptyset$ for $1 \leq i \leq k$,
- $V_i \bigcap V_j = \emptyset \quad \forall i, j : 1 \leq i < j \leq k$.

We usually seek a partition which optimizes an objective (see Section 2.2).

A partition Π satisfies the *balance criterion* iff

$$c(V_i) \leq (1 + \varepsilon)\frac{c(V)}{k} + \max_{v \in V} c(v) := L_{max}, \quad \text{for } i = 1, 2, \ldots, k$$

for some parameter $\varepsilon \geq 0$. In this case, we call Π an ε-balanced k-partition of $\mathcal{H}$. Like graph partitioning problems [22], the computation of an ε-balanced k-way partition of a hypergraph optimizing an objective is NP-hard [37] for most objectives. Given a k-way partition of $\mathcal{H}$, we define an *indicator function* $\lambda(\cdot)$ for the hypernodes. Given a hypernode $v \in V$, it returns the block of the hypernode:

$$\lambda : V \to \{1, \ldots, k\}$$
$$\forall v \in V_j : \quad \lambda(v) := j$$

Definition 2.1.4 (Connectivity, Connectivity Set)**.** Given a k-way partition $\Pi = \{V_1, \ldots, V_k\}$ of a hypergraph $\mathcal{H} = (V, E, c, \omega)$, we define the connectivity set of a hyperedge $e \in E$ as:

$$\mathit{connectivity\_set}(e) := \{V_i \in \Pi \mid \exists v \in \mathit{pins}[e] : v \in V_i\}$$

The cardinality of a connectivity set is called connectivity:

$$\mathit{connectivity}(e) := |\mathit{connectivity\_set}(e)|$$

In other words: the connectivity of a hyperedge is the number of different blocks which are connected by the hyperedge.

A hyperedge $e_i \in E$ with $\mathit{connectivity}(e_i) > 1$ is called a *border hyperedge*, whereas a hyperedge e_j with $\mathit{connectivity}(e_j) = 1$ is called *internal hyperedge*. We denote the set of all border hyperedges as

$$B(\Pi) := \{e \in E \mid \mathit{connectivity}(e) > 1\}$$

and the set of all internal hyperedges as

$$I(\Pi) := \{e \in E \mid \mathit{connectivity}(e) = 1\}.$$

The set $B(\Pi)$ can also be interpreted as the cut-set induced by partition Π.

Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$, a *clustering* of $\mathcal{H}$ is similar to a k-way partition. We want to divide the hypernodes into disjoint blocks, whilst optimizing an objective. However, in difference to a k-way partition, the number of blocks is not known beforehand. The objectives [46, 61] that are optimized in a clustering problem are different from the objectives used in the k-way partitioning problem. Note that we can not enforce a balance criterion on a clustering, since the number of clusters and therefore the average cluster size is not known beforehand. The resulting sets are called *clusters*.

2.2. Partitioning Objectives

This section focuses on the different objectives used in hypergraph k-way partitioning. We provide an overview of the most prominent objectives. For more information on this topic, we refer the reader to [7,31,45]. In the following, we always assume that $\Pi = \{V_1, \ldots, V_k\}$ is an ε-balanced k-partition of a hypergraph $\mathcal{H} = (V, E, c, \omega)$.

2.2.1. Hypergraph Cut

The most prevalent partitioning objective is called *hypergraph cut* [18,24,32] and is the canonical extension of the standard cut definition in graphs to hypergraphs. Each hyperedge containing at least two pins in different blocks contributes its weight to the hypergraph cut:

$$hCut(\mathcal{H}, \Pi) := \sum_{e \in B(\Pi)} \omega(e). \tag{2.2.1}$$

In other words: the hypergraph cut is the weighted sum of all hyperedges that need to be removed to produce k disjointed parts. In this case the objective is the minimization of the function $hCut(\mathcal{H}, \Pi)$.

2.2.2. Sum of External Degrees

We define the external degree of a hyperedge $e \in E$ as connectivity(e) iff the edge spans multiple blocks, and zero otherwise, i.e.:

$$extDeg(e) := \begin{cases} connectivity(e) & \text{if } connectivity(e) > 1 \\ 0 & \text{else.} \end{cases}$$

The sum of external degrees (soed) [31] is then defined as:

$$soed(\mathcal{H}, \Pi) := \sum_{e \in E} \omega(e) \cdot extDeg(e). \tag{2.2.2}$$

In case of the objective induced by the sum of external degrees, we want to minimize the function $soed(\mathcal{H}, \Pi)$.

2.2.3. $(K-1)$ metric

The $(K-1)$ metric [24] is very closely related to the sum of external degrees:

$$Km1(\mathcal{H}, \Pi) := \sum_{e \in E} \omega(e) \cdot (connectivity(e) - 1). \tag{2.2.3}$$

In case of the objective induced by the $(K-1)$ metric, we want to minimize the function $Km1(\mathcal{H}, \Pi)$.

2.2.4. Absorption

The absorption objective [7, 52] maximizes the sum of weighted fractions of hyperedges "absorbed" by the blocks:

$$absorption(\mathcal{H}, \Pi) := \sum_{V_i \in \Pi} \sum_{\substack{e \in E \\ pins[e] \cap V_i \neq \emptyset}} \frac{|pins[e] \cap V_i| - 1}{|e| - 1} \cdot \omega(e). \tag{2.2.4}$$

Imagine a block V_i and a hyperedge e_j, where one pin of the hyperedge is in block V_i. The absorption for this hyperedge in this block would be 0, increasing by $\frac{\omega(e_j)}{|e_j|-1}$ for each other pin belonging to the same block. The maximal value for the absorption of the hyperedge is $\omega(e_j)$, occurring when all pins of the hyperedge belong to the same block.

2.3. Graph-based Hypergraph Modeling

Since graphs are well studied, a straightforward way to solve the hypergraph k-partitioning problem is to transform the hypergraph to a graph (conserving the properties of the hypergraph regarding the objective) and partition that graph using state-of-the-art graph partitioners like Metis [33] or KaHIP [50].

Generally speaking, hypergraphs can be transformed into graphs by replacing each hyperedge with a weighted graph, where the edge weights are a function of the hypergraphs original weights. This graph is called a *cut-model* [29]. However, Ihler et al. [29] show that a general cut-model which has the same cut as the hyperedge can't exist for an arbitrary partition and hyperedges with cardinality greater than three. In other words: the weight of the cut in the cut-model can not have the same weight as the hyperedge cut (for all possible cuts in the cut-model). Therefore, if we choose to solve the k-way hypergraph partitioning problem via a transformation to a graph, it may result in worse solutions, since the graph does not perfectly mimic the properties of a hypergraph. Furthermore, the graph modeling the hypergraph has usually far more edges, resulting in a more complex problem instance.

Graph-based hypergraph modeling is mostly used in the machine learning community, where hypergraphs model higher-order relationships [1, 2]. Instead of a k-way partition, the problem is to find a good clustering. There are two major classes of cut-models: *Clique Expansion* [1, 62] and *Star Expansion* [1, 62].

2.3.1. Clique Expansion

In the clique expansion cut-model, each hyperedge is replaced with a clique, i.e. a hyperedge $e \in E$ with $|e| = n$ is modeled by an n-clique (a completely connected graph with n nodes and $\frac{n \cdot (n-1)}{2}$ edges). Each hypernode becomes a node, keeping its weight. Usually, the weight of the edges in the clique is uniformly distributed. Formally:

Definition 2.3.1 (Clique Expansion)**.** Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ we define the graph after performing clique expansion on $\mathcal{H}$ as $G = (V, E', c, \omega')$ with:

$$E' := \{(u, v) \mid \exists e \in E : u \in pins[e] \wedge v \in pins[e]\}$$
$$\forall d := (u, v) \in E' : \quad \omega'(d) := \rho \cdot \sum_{\substack{e \in E \\ u,v \in pins[e]}} \omega(e),$$

with ρ being a constant.

Hypergraphs resulting from machine learning problems often exhibit k-uniformity [2], i.e. each hyperedge has size k. Usually, ρ is being selected in respect of k and $n = |V|$.

Note that more complicated weighting functions exist [26, 29, 37]: A popular weighting function is proposed by Lengauer [37]. Edges in the clique resulting from a hyperedge $e \in E$ receive weight $\frac{\omega(e)}{|e|-1}$. In the case of overlapping hyperedges, the score is added up accordingly. However, the selection of the best weighting factor is a research field for itself [26, 29].

2.3.2. Star Expansion

The star expansion cut-model class inserts a new node for each hyperedge, where each pin of the hyperedge becomes an adjacent node to the "hyperedge-node". Thus, we get a *bipartite representation* of the hypergraph, since two "hyperedge-nodes" can not be adjacent. Formally:

Definition 2.3.2 (Star Expansion)**.** Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ we define the graph after performing star expansion on $\mathcal{H}$ as $G = (V', E', c, \omega')$ with:

$$V' := V \cup E$$
$$E' := \{(u, e) \mid e \in E : u \in pins[e]\}$$
$$\forall d := (u, e) \in E' : \quad \omega'(d) := \frac{\omega(e)}{|e|}$$

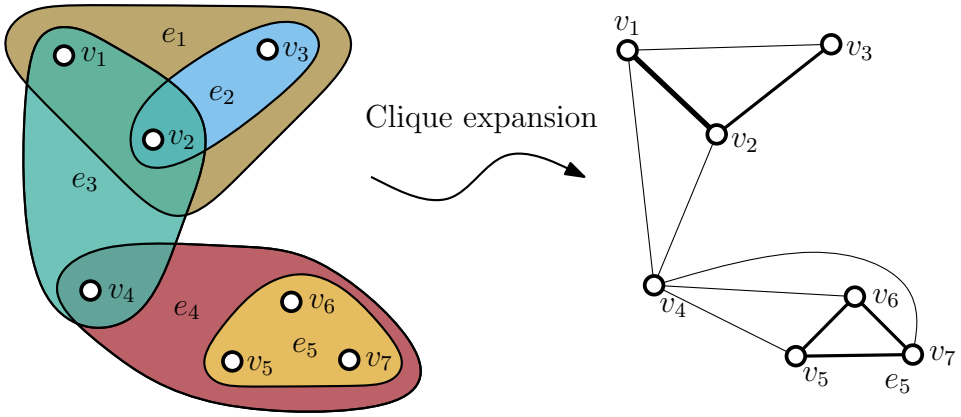


Figure 2.2.: *Clique expansion of the example hypergraph (Figure 2.1). Each hyperedge gets replaced by a clique. The weight of overlapping is added up. The thick lines represent heavier edges.*

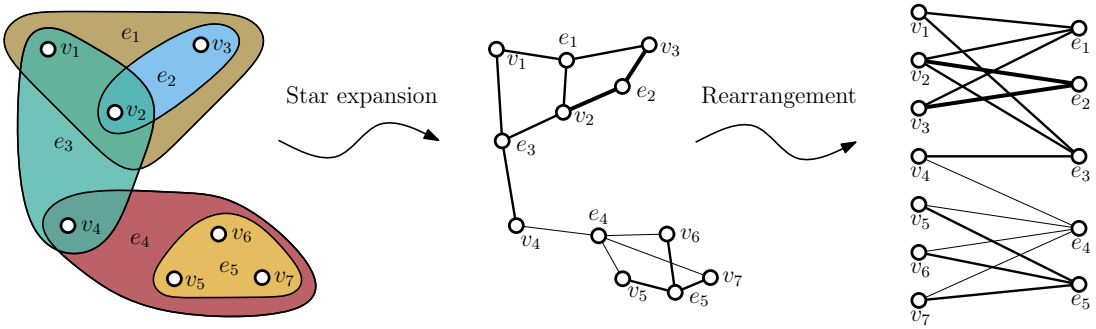


Figure 2.3.: *Star expansion of the example hypergraph (Figure 2.1). Each hyperedge gets replaced by a new node, with edges being added between the "hyperedge-node" and its pins. In the star expansion, these edges have a weight which correlates with the initial hyperedge size. Edges originating from a small hyperedge have therefore a larger weight and are represented by thicker lines. The rearrangement is depicted for emphasis on the bipartite nature of the graph.*

3 | Related Work

This chapter presents the related work of this thesis. In Section 3.1 the multilevel partitioning scheme is discussed in detail with examples for its utilization. Section 3.2 describes the common coarsening schemes employed in the coarsening step of the multilevel partitioning scheme. The usual initial partitioning techniques are mentioned in Section 3.3. In Section 3.4 we introduce local search techniques, which are utilized in the uncoarsening/refinement step of the partitioning scheme. In Section 3.5, a special case of the multilevel partitioning scheme which maximizes the number of levels, is introduced. The most widely used state-of-the-art hypergraph partitioners, PaToH and hMetis, are covered in Section 3.6. Finally, in Section 3.7 we provide a detailed explanation of the original label propagation algorithm and a brief overview of its applications. Note that since the focus of this thesis is hypergraph partitioning, we omit graph partitioning techniques and applications. For a survey on this topic, we refer the reader to [16].

3.1. Multilevel Partitioning Scheme

The multilevel partitioning scheme [13, 27] (Figure 3.1) is a well known and common heuristic for the graph partitioning problem and the hypergraph partitioning problem. It consists of three phases:

(i) coarsening phase

(ii) initial partitioning phase

(iii) uncoarsening/refinement phase

Coarsening Phase. This phase successively coarsens the hypergraph until it is small enough to be efficiently partitioned with an initial partitioning algorithm. The goal of the coarsening, besides the reduction of the size of the hypergraph, is the conservation of structural properties. This is important, since neglecting these properties in the coarsening scheme results in a worse overall quality of the solution. The coarsening is usually realized via *hypernode contractions*.

The objective of this phase is the maximization of the number of hyperedges with cardinality one and the minimization of the size of hyperedges. We want to maximize the number of hyperedges with size one, because these edges can not be cut in the coarser graphs and can be omitted in the next level (simplifying the problem instance). Furthermore, the usual local search algorithms used in the uncoarsening and refinement phase (Fiduccia-Mattheyses derivations) work worse if the hypergraph has many large hyperedges. Therefore, we also want to reduce the size of hyperedges. There exists a large amount of different coarsening schemes. We cover the most prominent in Section 3.2.

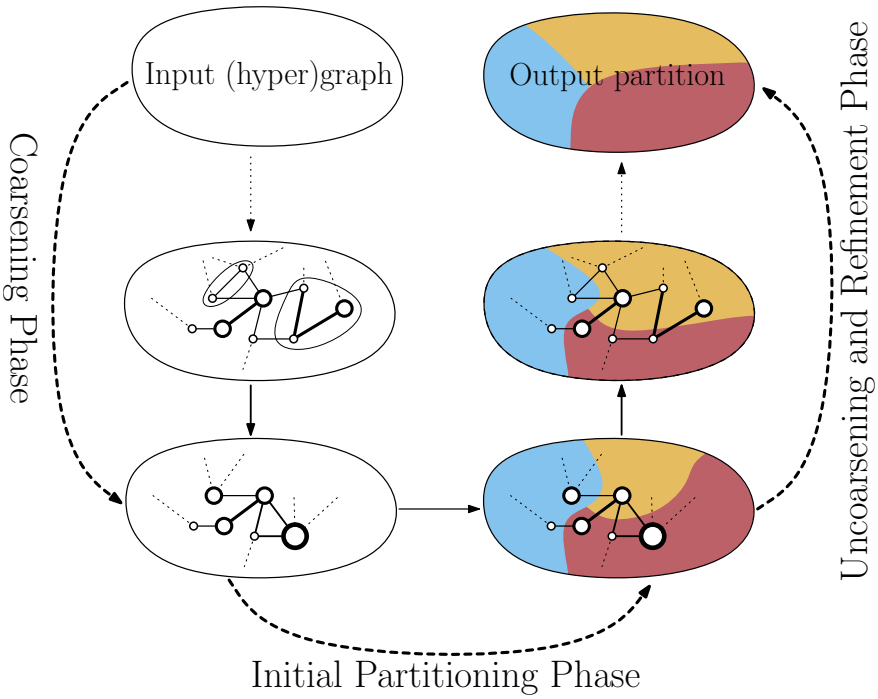


Figure 3.1.: *The multilevel partitioning scheme consists of three phases: coarsening phase, initial partitioning phase and uncoarsening and refinement phase. The input (hyper)graph is successively coarsened in the coarsening phase until it is small enough to be feasibly partitioned in the initial partitioning phase. During the uncoarsening and refinement phase the partitioning is being successively projected to the next finer level, while a local search algorithm improves the solution.*

Initial Partitioning Phase. In this phase, an initial partitioning algorithm computes a partition of the coarsest graph, i.e. the assignment of vertices to blocks. The usual approaches for the computation of the initial partition are a random assignment of nodes to blocks, or grow-based algorithms. In case of the latter, blocks are grown from randomly selected nodes. We discuss the different approaches used in this phase briefly in Section 3.3.

Uncoarsening/Refinement Phase. During this phase, the solution of a coarser graph is projected to the next finer graph. After that, a local search algorithm refines the projected partition. The usual approaches can be divided into Kernighan-Lin (KL) derived heuristics and approaches derived from the Fiduccia-Mattheyses (FM) heuristic. We will explain the original heuristics in Section 3.4.

In Algorithm 1, the pseudocode for the multilevel partitioning scheme is shown. Lines 3-6 belong to the coarsening phase, line 7 to the initial partitioning phase, and lines 8-12 to the uncoarsening and refinement phase.

Algorithm 1: Multilevel Partitioning

```
Input: (hyper)graph G, number of desired blocks k, maximal imbalance ε
Output: partition of G into k parts
repeat                                                  // coarsening phase
    G ← coarsen(G, k)
until G is small enough
P ← initial_partitioning(G, k, ε)                       // initial partitioning phase
repeat                                                  // uncoarsening and refinement phase
    G ← uncoarsen(G, k)
    P ← refine(P, G, k, ε)
until G is completely uncoarsened
return P
```

3.2. Coarsening Schemes

Generally, the coarsening schemes used in the coarsening step of the multilevel partitioning heuristic can be divided into two groups: matching-based coarsening and agglomerative coarsening. Matching-based coarsening contracts matchings, i.e. disjoint pairs of hypernodes. Therefore, the number of hypernodes in the subsequent level gets reduced at most by a factor of two.

In agglomerative coarsening disjoint sets of hypernodes are contracted. In the following we refer to these sets as clusters. There is one problem with this approach. If not controlled, the number of hypernodes in subsequent levels gets possibly reduced by a large factor, since a cluster can get arbitrarily large. This results in very heavy hypernodes in the coarser levels and a small number of levels overall. These nodes negatively impact the quality of the partition: In the initial partitioning phase, they make it difficult for the initial partitioning algorithm to produce a good balanced partitioning. Furthermore, the movement of these very heavy nodes is restricted in the refinement phase, since the partition should remain balanced. A usual solution for this problem is a penalty for large clusters during coarsening: When a hypernode determines its affiliation to a cluster, the score receives a penalty dependent on the cluster size.

There exists a numerous amount of selection criteria when determining which hypernodes should be matched/clustered. In the following we will focus on the strategies utilized in the prominent state-of-the-art hypergraph partitioners hMetis and PaToH. For a more in depth look into selection criteria, we refer the reader to [7].

3.2.1. Edge Coarsening

Edge Coarsening [6, 32, 60] is a matching-based coarsening strategy and is supported in hMetis. It selects random pairs of nodes that are present in the same hyperedge. Essentially, it performs clique expansion on the hyperedges. On this representation it then computes a random matching and contracts this matching. However, this transformation of the hypergraph to a graph is done implicitly during the matching step.

There exists a variation of edge coarsening called *heavy-edge* coarsening. Instead of randomly selecting a hypernode for matching, it selects the unmatched hypernode that is connected via the edge with the largest weight. The weight $\omega'(v, u)$ of an edge connecting two nodes v and u is the sum of all hyperedge weights which contain these two nodes, where each hyperedge e contributes $\frac{\omega(e)}{|e|-1}$ the weight:

$$\omega'(v, u) := \sum_{\substack{e \in E \\ u,v \in pins[e]}} \frac{\omega(e)}{|e| - 1}.$$

3.2.2. Hyperedge Coarsening and Modified Hyperedge Coarsening

Hyperedge coarsening [32] is an agglomerative coarsening scheme which contracts hyperedges that do not share common hypernodes. In this scheme, all hyperedges are initially sorted in decreasing hyperedge weight order. Hyperedges with the same weight are sorted in increasing size order. Then, the hyperedges are visited in this order and for each hyperedge that only contains unclustered hypernodes, the hypernodes are clustered. Therefore, hyperedge coarsening tries to eliminate hyperedges with large weight and those of small size. After all hyperedges have been visited, all hypernodes that were clustered are contracted. Hypernodes that were not clustered are simply copied to the next level.

Hyperedge Coarsening is able to drastically reduce the total hyperedge weight that is left exposed in coarser graphs. Still, it is possible that we ignore many hyperedges, because some of their pins have already been clustered. This leads to two problems: First, the size of hyperedges may not decrease sufficiently, and the weight of hypernodes (i.e. the number of hypernodes that have been contracted) may differ greatly, which impedes the initial partitioning algorithm.

These problems are addressed in the modified hyperedge coarsening scheme [32]. This scheme first performs hyperedge coarsening. After contraction, each non-contracted hyperedge is visited and all hypernodes that do not belong to any other contracted hyperedge are clustered. These hypernodes will then also be contracted before the descent to the next coarser level. Both hyperedge coarsening and modified hyperedge coarsening are implemented in hMetis.

3.2.3. First Choice

Edge coarsening, hyperedge coarsening and modified hyperedge coarsening all share one characteristic, namely that they all find maximal independent groups of hypernodes. Therefore, as soon as some set of hypernodes get matched, no additional hypernodes can be matched to this particular set. Karypis [31] realized that this approach may destroy some cluster structures that naturally exist in the hypergraph. An example for such naturally occurring cluster structures is shown in Figure 3.2. This observation led to an agglomerative coarsening scheme called *first choice* [31]. First choice is derived from the heavy-edge coarsening scheme (Section 3.2.1). Again, this scheme visits all hypernodes in random order. However, for each hypernode it considers all adjacent hypernodes, disregarding whether they have already been matched or not. Therefore, hypernodes are matched which are connected via the heaviest edge (in the clique-expanded

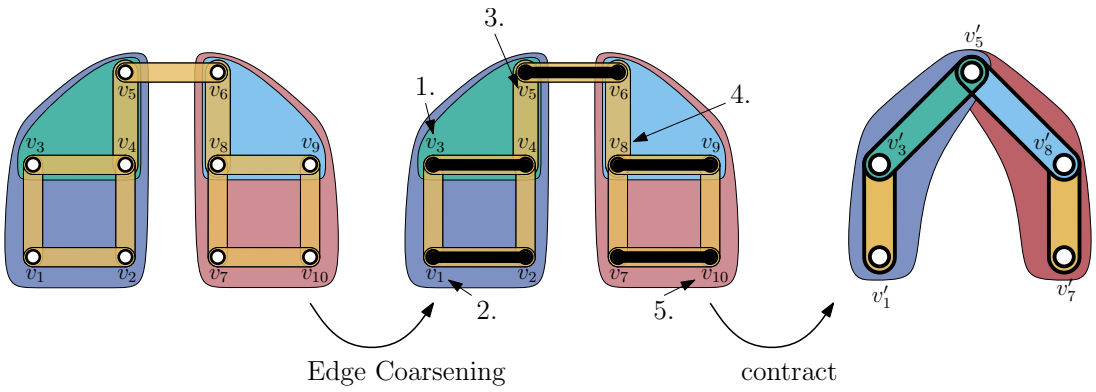


Figure 3.2.: *Edge Coarsening can destroy naturally existing cluster structures in hypergraphs. In the left image, the initial hypergraph is depicted. In the middle image, the matchings returned by edge coarsening are pictured as black lines. The order of the visit was* $v_3, v_4, v_1, v_2, v_5, v_6, v_8, v_9, v_{10}, v_7$. *Since already matched hypernodes are ignored in this scheme, in the middle image we only enumerate hypernodes if they were not matched at the moment of the visit. The rightmost image shows the hypergraph after contraction. We merge parallel hyperedges for the sake of simplicity. The heavier hyperedges are drawn thicker. The hypergraph originates from [31].*

graph), breaking ties in favor of unmatched nodes. This leads to arbitrarily large groups of hypernodes which will be contracted.

One problem with this approach is that the size of successive coarser graphs may decrease by a large factor, potentially limiting the effect of local search. Karypis solved this problem by stopping the coarsening process as soon as the size of the next coarser graph would decrease by a bigger factor than $1.5 - 1.8$.

3.2.4. Heavy Connectivity Coarsening

Heavy connectivity coarsening [18] is a general term for a matching-based and an agglomerative coarsening scheme utilized by PaToH. They are called *heavy connectivity matching* (HCM) and *heavy connectivity clustering* (HCC). Their objective is to find highly connected hypernode clusters.

The matching-based clustering works as follows: It visits all hypernodes in random order and checks if the current hypernode u has already been matched. If that is not the case, it chooses the unmatched adjacent hypernode v which shares the maximal number of incident hyperedges with u:

$$|hyperedges[u] \cap hyperedges[v]|. \tag{3.2.1}$$

The agglomerative variant of the clustering works very similar to the matching-based, but allows for more than two hypernodes to be clustered together. At the beginning of the coarsening, it assumes that each hypernode u constitutes a singleton cluster $C_u := \{u\}$. It visits all hypernodes in random order. If the current hypernode u has already been clustered ($|C_u| > 1$), that hypernode is

ignored. Otherwise, it considers all adjacent hypernodes of u and selects the hypernode which maximizes the following selection criterion:

$$\arg\max_{v \in V} \frac{|hyperedges[u] \cap hyperedges[C_v]|}{c(C_v \cup \{C_u\})}, \tag{3.2.2}$$

The division by $c(C_v \cup \{C_u\}$ is a penalty for large clusters which restricts the formation of heavy hypernodes on the next coarser level. After the clustering, all clusters are contracted. Note that after one iteration no more singleton clusters remain (assuming there are no isolated hypernodes).

3.2.5. Absorption Clustering

Absorption clustering [18] is a general term for clustering schemes that optimize the absorption metric [7] (See Section 2.2.4). PaToH currently supports three such schemes:

- absorption matching
- absorption clustering using nets
- absorption clustering using pins

All variants visit all hypernodes in random order and select an adjacent hypernode or a cluster (in case of agglomerative schemes) that maximizes a selection criterion. After the computation of the matching or clustering, all matched/clustered hypernodes are contracted. In case of absorption matching the selection criterion is

$$\arg\max_{v \in V} \sum_{\substack{e \in hyperedges[u] \cap \\ hyperedges[v]}} \frac{\omega(e)}{|e| - 1}, \tag{3.2.3}$$

for unmatched hypernodes v, u. This selection criterion favors hypernode pairs which are connected via many, heavy, small hyperedges. Note that this coarsening strategy is identical to heavy edge coarsening strategy in hMetis.

Absorption clustering using nets is the agglomerative version of absorption matching. It uses a very similar selection criterion to Equation 3.2.3:

$$\arg\max_{v \in V} \sum_{\substack{e \in hyperedges[u] \cap \\ hyperedges[C_v]}} \frac{\omega(e)}{|e| - 1}, \tag{3.2.4}$$

with the difference that the similarity is computed to clusters, and single node clusters are allowed to be matched to already matched hypernodes. Ties are resolved in favor of unmatched nodes.

The last absorption clustering scheme is called absorption clustering using pins and is the default coarsening scheme in PaToH. For this variant, the similarity between a hypernode u and a cluster C_v is

$$\arg\max_{v \in V} \sum_{\substack{e \in hyperedges[u] \cap \\ hyperedges[C_v]}} |e \cap C_v| \cdot \frac{\omega(e)}{|e| - 1}. \tag{3.2.5}$$

In other words: Absorption clustering using pins accumulates the score (Equation 3.2.3) for each pin of the hyperedge that is part of the cluster.

Algorithm 2: Best Choice Coarsening

```
Input: hypergraph H = (V, E, c, ω), |V| = n, |E| = m
Output: coarser hypergraph H' = (V', E', c', ω')
pq ← new PriorityQueue
for u ∈ V do
    (u, v, d) ← (u, μ(u), d(u, v))      // the closest hypernode and their score
    pq.insert(u, v, d)
while H' is not small enough do
    (u', v', d') ← pq.pop                // get the currently best score
    H' ← contract(u, v)                  // contract the two nodes
    update (pq, u')                      // update all neighbors of the new node
```

3.2.6. Best Choice

Best choice is a clustering technique which was introduced by Alpert et al. [4] and has since then been heavily used in the Very-Large-Scale Integration (VLSI) domain [42, 54, 56]. They implement their methodology within a leading industrial placement tool called `CPLACE` [8]. The quality of the placement is then measured by the final placements wire length. Note that best choice is not used within the multilevel partitioning scheme [4]. Nevertheless, this algorithm can be utilized as coarsening strategy and the results of Alpert et al. [8] indicate that best choice outperforms edge coarsening and first choice in both quality and performance. Best choice is structurally different to the other agglomerative coarsening schemes presented before. It successively contracts hypernodes during computation, whereas the other coarsening schemes first compute a clustering and contract it afterwards.

This coarsening scheme (re)defines the weight of a hyperedge e as

$$w(e) := \frac{\omega(e)}{|e|}. \tag{3.2.6}$$

The weight of a hyperedge therefore gets inversely scaled by the number of hypernodes it connects. Then, the clustering score for two hypernodes u, v is defined as

$$d(u,v) := \sum_{\substack{e \in hyperedges[u] \cap \\ hyperedges[v]}} \frac{w(e)}{c(u) + c(v)}.$$

In other words: The clustering score is proportional to the total sum of hyperedge weights between u and v and inversely proportional to the summed weight of u and v. Note that if two hypernodes get contracted, the weight of the resulting hypernode is equal to the sum of its uncontracted parents.

The *closest* object to a hypernode u is defined as the adjacent hypernode v, which has the maximal clustering score $d(u, v)$, i.e.

$$\mu(u) := v \Leftrightarrow \operatorname*{arg\,max}_{w \in hyperedges[u]} d(u,w) = v.$$

The idea behind best choice clustering is to always contract hypernodes which have the maximal clustering score. This can be achieved with a *priority queue*,

whose entries are tuples (u, v, d), with v denoting the closest hypernode of u and $d := d(u, v)$ being their clustering score and the key for priority queue. The pseudocode for the algorithm is shown in Algorithm 2.

Best choice works in two phases. In the beginning, each hypernode is inserted with its closest partner and their score into the priority queue. In the second phase, the tuple (u, v, d) with the highest score is removed and the hypernodes u, v are contracted. At this point, it is possible that the closest hypernodes of u and v's neighbors have changed. Therefore, the closest hypernodes in the priority queue for the neighbors need to be updated.

The update step has a large negative impact on the running time, because for hypergraphs with large hyperedges, many entries in the priority queue are updated in each step. Alpert et al. [4] propose a heuristic that deals with this problem. Statistical analysis of the priority queue management [4] shows that

(i) A tuple might be updated a number of times before it reaches the top, thus rendering the first update unnecessary.

(ii) In 96% of all updates, the new score decreased [4].

Motivated by these observations, they propose a *lazy update* technique. In the beginning all objects in the priority queue are marked as valid. The update step in line 7 (Algorithm 2) invalidates all neighbors of the contracted nodes. If the topmost object in the priority queue is marked as invalid, its closest object is recalculated and reinserted into the priority queue. Alpert et al. demonstrate that the lazy update technique leads to a drastic decrease of the running time.

3.3. Initial Partitioning

In the initial partitioning phase the goal is the computation of the initial partition, i.e. to assign hypernodes of the coarsened hypergraph to blocks.

The straightforward approach is to continue the coarsening of the hypergraph, until only k hypernodes remain. This, however, very often results in a highly unbalanced initial partition [58]. This is a problem, because now, the local search algorithm needs to be very effective, since it needs to restore the balance. Moreover, the improvement of the uncoarsening and refinement phase is limited because a lot of moves are infeasible due to the violation of the balance constraint.

There are three general methods which are used for the computation of the initial partition:

(i) random assignment of nodes to blocks

(ii) grow based heuristics

(iii) spectral methods

hMetis [32] supports three different initial partitioning schemes, which are based on the principle of recursive bisection: *balanced random bisection* [34], *Graph Growing Partitioning algorithm* (GGP), and *Greedy Graph Growing Partitioning algorithm* (GGGP) [34].

In case of balanced random bisection the hypernodes are randomly assigned to one of the two blocks, respecting the balance constraint. In contrast, GGP selects

a hypernode and gathers half of the hypernodes in terms of weight as follows: It grows a set of nodes around the randomly chosen hypernode in a breadth-first search fashion until the weight of the set is half of the total hypernodes weight. GGGP is an improvement on the GGP algorithm. As in GGP, it starts with a random hypernode and grows a block around it. The growing process is in contrast to GGP *greedy*, i.e. all border hypernodes are ranked according to the decrease in cut, if the hypernode were to be added to the block.

GGGP is less subject to the initial choice of the hypernode than GGP. However, this comes with the penalty of increased running time. Therefore, fewer initial runs can be performed with GGGP than GGP if we want to keep a similar running time. Nevertheless, the experiments conducted by Karypis et al. [32] suggest that GGGP still achieves a better result.

Karypis et al. [32] argue that the quality of the initial partition may not reflect the quality of the overall partition, i.e. given two initial partitions P_1 and P_2, it is possible that P_2 has a worse quality than P_1, but the final partition based on P_2 has a better quality than the one based on P_1. This is the main reason why exhaustive enumeration is usually not used as the initial partitioning algorithm. Even if the initial partitioning algorithm would compute the optimal partition of the coarsest hypergraph, the used computational time is often better utilized in the uncoarsening and refinement phase, thus leading to a better partition.

Instead, Karypis et al. propose to perform a fixed number of initial partitions and successively propagate partitions whose cut is within the best $x\%$ of the best cut at the current level.

PaToH supports a numerous amount of different initial partitioning schemes, which are all either based on random assignment, breadth-first search, or GGGP. For a more in depth overview of the possible initial partitioning heuristics, we refer the reader to [15].

3.4. Refinement Techniques

This section introduces the Kernighan-Lin (KL) heuristic and the Fiduccia-Mattheyses (FM) heuristic, which can both be used as a local search strategy. The refinement techniques utilized in hMetis and PaToH are either derived from the FM heuristic or from the KL heuristic.

3.4.1. Kernighan-Lin

The initial version of the Kerningham-Lin algorithm [35] refines a *perfectly* balanced ($\varepsilon = 0$) bisection of a graph where all nodes have uniform weight. There exist adaptions of the algorithm for ε-balanced bisections and nodes with different weights. Furthermore, Schweikert and Kernighan [51] propose an adaptation to hypergraphs. The initial version of the algorithm has a non-linear running time ($|E|^2 \log |E|$) [15] and is therefore rarely used in practice. Hence, we will discuss it only briefly. For an in depth look at the different adaptations and optimized implementations, we refer the reader to [15].

The main idea is to define a $gain(\cdot, \cdot)$ function for pairs of hypernodes in different blocks. This function denotes the improvement of the objective function

Algorithm 3: KL algorithm

```
Input: hypergraph H = (V, E, c, ω), perfectly balanced bisection V_1, V_2 of H
Output: Refined bisection V'_1, V'_2 of H
1  V'_1 ← V_1                                                    // initialization
2  V'_2 ← V_2
3  repeat
4      for 1 ≤ ℓ ≤ |V|/2 do
           // find equally heavy, non-locked hypernodes that maximize gain
           v_i^ℓ, v_j^ℓ ← argmax_{v_i^ℓ ∈ V'_1, v_j^ℓ ∈ V'_2, c(v_i^ℓ)=c(v_j^ℓ)} gain(v_i^ℓ, v_j^ℓ)
5          q^ℓ ← gain(v_i^ℓ, v_j^ℓ)
6          V'_1 ← V'_1 \ {v_i^ℓ} ∪ {v_j^ℓ}                       // swap blocks of v_i^ℓ, v_j^ℓ
7          V'_2 ← V'_2 \ {v_j^ℓ} ∪ {v_i^ℓ}                       // swap blocks v_i^ℓ, v_j^ℓ
8          lock v_i^ℓ, v_j^ℓ                                     // don't consider v_i^ℓ, v_j^ℓ in next iterations
9      pos ← argmax_k Σ_{k=1}^{|V|/2} q^k                        // when was the best solution seen
10     max_gain ← Σ_{k=1}^{pos} q^k                              // the improvement of the best solution
11     if max_gain > 0 then
12         for |V|/2 ≥ k ≥ pos do                                // rollback to the best solution
13             V'_1 ← V'_1 \ {v_i^k} ∪ {v_j^k}                   // swap blocks of v_i^k, v_j^k
14             V'_2 ← V'_2 \ {v_j^k} ∪ {v_i^k}                   // swap blocks of v_i^k, v_j^k
15     else
16         for |V|/2 ≥ k ≥ 0 do                                  // undo all swaps
17             V'_1 ← V'_1 \ {v_i^k} ∪ {v_j^k}
18             V'_2 ← V'_2 \ {v_j^k} ∪ {v_i^k}
19 until max_gain ≤ 0                                           // continue as long we improve the bisection
20 return V'_1, V'_2
```

(in our case the cut), if the two hypernodes swap the block they belong to. Note that since the algorithm assumes that the partition is perfectly balanced, and always swaps two nodes with the same weight, the partition remains perfectly balanced.

The algorithm operates in *passes* (one pass is lines 4-18 in Algorithm 3), in which two nodes with the highest gain are greedily swapped. Once a node has been moved, it is ignored in the current pass. Note that the gain can become negative. Next, the overall best partition encountered in this pass is determined and the algorithm checks if the score was positive (the bisection was improved). If so, KL performs a *rollback* to the bisection with this score and starts a new pass. In Algorithm 3 the pseudocode for the KL algorithm is shown.

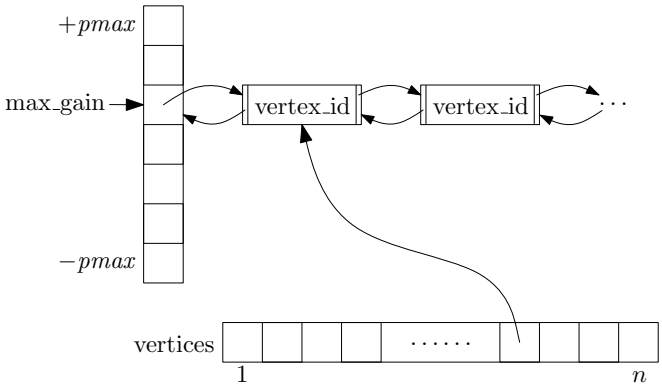


Figure 3.3.: *The data structure proposed by Fiduccia and Mattheyses [21].*

3.4.2. Fiduccia-Mattheyses

There exist many adaptations and improvements to the KL heuristic. The most prominent is the Fiduccia-Mattheyses algorithm [21]. This adaptation reduces the complexity of KL for both graph and hypergraph partitioning. Their main contribution is the introduction of a data structure (called bucket priority queue), which allows for an efficient computation of the node with the maximal gain. Furthermore, they modify the KL heuristic in such a way that they no longer swap pairs of nodes but only move a single node.

The data structure supports four operations:

(i) determine the gain of a vertex

(ii) insert and remove gain elements

(iii) find the elements with the maximal gain

(iv) find the elements with the second highest gain

In Figure 3.3, the proposed gain list structure is shown. It consists of two arrays. The first has size $2pmax+1$, $pmax$ denoting the maximal possible gain. Each entry in this array holds a linked list with all nodes that have the gain corresponding to this entry. The other one holds for each vertex a pointer to its position in one of the linked lists. Furthermore, the index (gain) of the first non-empty linked list is stored explicitly. In case of a bisection, there are two such data structures. One stores for each vertex its gain in case it is moved to the first block, whereas the second one stores its gain in respect to the second block. The first three operations have a constant time complexity, whereas the localization of the elements with the second highest gain is linear in the number of the maximal possible gain.

One problem with the data structure is that the maximal possible gain is dependent on the maximal degree of a hypernode ($= \max d(v) \cdot \max \omega(e)$). Therefore, if the hypergraph contains hypernodes with a wide range of hypernode degrees much space is wasted in the first array. There exist optimized variations of this data structure which cope with this problem, e.g. with the utilization of a *binary search tree* and a *hash map* the time complexity for the second operation becomes logarithmic in the number of non-empty lists. The other operations have a constant time complexity. For more information on this and further optimizations

Algorithm 4: FM algorithm

```
Input: hypergraph H = (V, E, c, ω), bisection V_1, V_2 of H, max. imbalance ε
Output: Refined bisection V_1, V_2 of H
repeat
    initialize the two gain tables
    for 1 ≤ i ≤ |V| do
        select unlocked node v ∈ V, such that its gain is maximal and after
        the transfer the graph has imbalance ≤ ε
        transfer v to the other block
        foreach unlocked vertex v' ∈ V adjacent to v do
            update the gain tables for v'
        lock v
    j ← number of transfers maximizing the gain max_gain
    if max_gain > 0 then                    // rollback to the best solution
        Alter the bisection V_1, V_2 according to j
    else
        Undo all swaps in this iteration
until max_gain ≤ 0                  // continue as long we improve the bisection
return V_1, V_2
```

we refer the reader to [15, 45].

Algorithm 4 outlines the pseudocode for the FM algorithm. Let $\ell := \sum_{e \in E} |e|$ be the number of pins of the hypergraph. The initialization step has a complexity of $\mathcal{O}(\ell)$, the localization of the node with the maximal gain has amortized constant complexity $\mathcal{O}(1)$, the update step has a constant time complexity per node, and is triggered at most $|e|$ times per incident hyperedge $e \in E$. The amortization works as follows: Let v, w be two subsequent hypernodes that both have maximal gain. The cost for the update of the pointer *pmax* is bound by $\mathcal{O}(deg(v) + deg(w))$. This cost, however, is amortized, since the gain values of all neighbors of v and w need to be updated. Therefore the total running time of a single pass of the FM algorithm is amortized $\mathcal{O}(\ell)$, which is a vast improvement when compared to the KL algorithm, which was $\mathcal{O}(|E|^2 \log |E|)$.

Note that the FM algorithm performs random tie-breaking, i.e. if multiple moves have the maximal gain, one of them gets selected at random. An adaptation of FM which employs a different kind of tie-breaking is proposed by Krishnamurthy [36]. He utilizes the concept of "level-gains" for the computation of a bipartition of a hypergraph. His adaptation has a running time of $\mathcal{O}(\xi \cdot \ell)$, where ℓ denotes the number of pins and ξ the number of gain levels. His experimental results imply that the utilization of this tie-breaking improves the quality of the FM algorithm.

In its initial version the FM algorithm refines a bipartition of a hypergraph. Note that this concept can also be extended to k-way local search in a k-way partition: Hendrickson et al. [27] adapt FM to k-FM for graphs. They use $k(k-1)$ gain tables (Figure 3.3), one for each type of move (from block i to block j). Their approach has a complexity of $\mathcal{O}(k|E|)$.

Finally, Sanchis [48] also utilizes $k(k-1)$ priority queues and extends the concept of level gains to a k-way partition of a hypergraph. Her k-way local search algorithm has a running time of $\mathcal{O}(\xi\ell k(\log k + \xi + \max_{v \in V} deg(v) \cdot \max_{e \in E} \omega(e)))$, ξ denoting the number of gain levels and ℓ denoting the number of pins.

3.5. n-Level Partitioning Scheme

An extreme variant of the multilevel partitioning scheme is called the n-level partitioning scheme [44]. The main difference between them is that the n-level scheme contracts only two nodes at each level, whereas the multilevel partitioning scheme contracts an arbitrary amount of nodes at each level. The central idea behind this approach is to make subsequent levels very similar. This leaves the local search algorithm much room to improve the quality of the partition. Furthermore, instead of the computation of a matching or clustering in the coarsening step, they can greedily select two nodes according to a rating function, thus simplifying the coarsening. Another difference to other coarsening strategies is that with the utilization of a priority queue, the *global* best pairs of hypernodes (according to a rating) are contracted, whereas the other coarsening strategies have a very *local* view.

3.6. State-of-the-art Hypergraph Partitioners

The most prominent and widely used state-of-the-art hypergraph partitioners are hMetis [31] and PaToH [18]. They both implement the multilevel partitioning scheme, which is explained in Section 3.1.

3.6.1. hMetis

hMetis was proposed by Karypis et al. in [32] with the focus of partitioning VLSI instances. Hypergraph partitioning is extensively applied in the field of VLSI, where the goal is the placement of thousands of transistors onto a single chip. The usual metric of a placement is called *wirelength* and measures the total wire length of the circuit. A good placement is important because it impacts

(i) the timing performance of the chip,

(ii) the power consumption of the chip,

(iii) the total area of the chip.

A circuit can easily be modeled by a hypergraph: Each gate is represented by a hypernode and all inputs to the gate are grouped into a single hyperedge. A gate thereby realizes a logical operation, e.g. `XOR`. This problem can (partially) be solved with a k-way partition of the hypergraph.

hMetis supports both direct k-way partitioning and *recursive bisection*. Recursive bisection first computes a bisection of the hypergraph, i.e. $k = 2$. Next, it recursively bisects the two disjoint graphs induced by the partition. If $k = 2^n$, it needs to perform $n - 1$ recursions to achieve k different blocks. Note that recursive bisection is also able to solve the k-partitioning problem if k is not a power of two. This is achieved by modifying size ratios of the bisection.

In case of direct k-way partitioning, hypernodes are directly assigned to one of the k different blocks. Furthermore, hMetis currently[1] supports 11 coarsening schemes and 7 refinement schemes. They contain multiple variants of the ideas which we cover in Section 3.2 and Section 3.4.

3.6.2. PaToH

Sparse matrices can also be easily modeled as hypergraphs. Given a matrix A_{ij}, the rows/columns are interpreted as hyperedges. One goal is the parallelization of various operations on the matrix, like the dense-vector sparse matrix product. PaToH was proposed by Catalyrek and Aykanat in [18] as a framework which optimizes the *total communication volume* for a k-way partition. The total communication volume is thereby the amount of data transfers necessary to finish the operation on a matrix while dividing its data onto k processing units. PaToH[2] uses recursive bisection for the computation of the partition and currently[3] supports 17 different coarsening schemes, 13 initial partitioning algorithm variants, and 10 refinement schemes. Many of them are very similar, we cover their basic variants in Section 3.2 and Section 3.4.

3.7. Label Propagation

Label propagation was first introduced by Raghavan et al. [47] as an algorithm for community structure detection (or clustering) in large-scale graphs. Algorithm 5 outlines the pseudocode of original label propagation on graphs proposed in [47]. The time complexity for each iteration is

$$\mathcal{O}(n + m), \tag{3.7.1}$$

since every node is visited once and every edge twice. This results in a near-linear time complexity for the complete algorithm, if the number of maximal iterations is controlled. According to Raghavan et al. [47], 95% of all nodes have the same label that the maximum number of their respective neighbors have by the end of the fifth iteration.

Algorithm 5: Label propagation

```
Input: graph G = (V, E), |V| = n, |E| = m
Output: label[1 ... n]                                  // the labels for each node
1 for v ∈ V do
2     label[v] ← v                                      // initialization
3 while not converged and num_iterations ≤ max_iterations do
4     for v ∈ V in random order do
5         label[v] ← arg max_σ score(v, σ)              // choose "best" adjacent label
```

[1] http://glaros.dtc.umn.edu/gkhome/fetch/sw/hmetis/hmetis-2.0pre1.tar.gz
[2] Version 3.2: http://bmi.osu.edu/umit/software.html
[3] http://bmi.osu.edu/umit/PaToH/manual.pdf

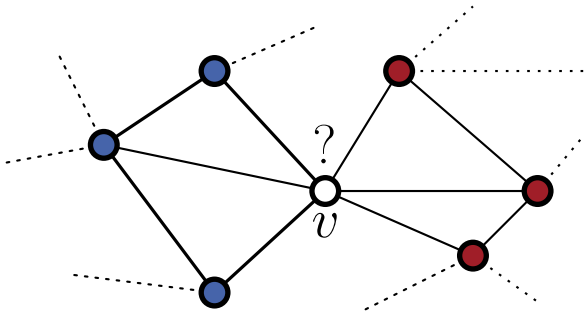


Figure 3.4.: *Non-convergence of the naive label propagation algorithm for an unweighted graph. We want to determine the new label for node v. Both adjacent labels (depicted as different colors) have the same score. Due to the random tie-breaking, the label of v will alternate between these two labels.*

The usual choice for the score function in line 6 is the sum of edge weights, which connect the node to the same label, i.e.

$$\texttt{score}(v, \sigma) := \sum_{\substack{e=(v,u)\in E \\ \texttt{label}[u]=\sigma}} \omega(e). \tag{3.7.2}$$

Ties are broken randomly among all candidates with the maximal score. In case of an unweighted graph, the score simply counts the number of edges connecting the node to that label ($\omega \equiv 1$).

Ideally, the algorithm converges after a constant number of iterations. Figure 3.4 shows an example where the naive version of the algorithm does not converge, because node v alternates between two labels with equal score. Raghavan et al. therefore propose a different stopping criterion: They stop the iterative process as soon as every node has a label with the maximal number of occurrences among its neighbors. After the execution of the algorithm, all nodes sharing the same label are assumed to belong to the same cluster.

Since its introduction, the label propagation algorithm has become very popular in the field of machine learning. Kang et al. [30] use a modified version of the algorithm in the domain of multi-labeled learning. Their proposed framework, Correlated Label Propagation (CLP), is an improvement to the kernel-based k-Nearest Neighbors (kNN) approach for multi-label learning. In contrast to standard label propagation, CLP co-propagates multiple labels in each step. Zheng-Yu et al. [43] use label propagation as a semi-supervised learning algorithm for the word sense disambiguation (WSD) problem. The goal is the assignment of appropriate sense to an occurrence of a word in a given context. Their algorithm fully realizes a global consistency assumption (similar examples should have similar labels) and outperforms support vector machines (SVM) when only a few labels are available. Tang et al. [53] annotate large quantities of images by label propagation over nosily tagged web images. They construct a sparse kNN-graph with both labeled and unlabeled images and propagate the noisy tags to the unlabeled nodes. Their semi-supervised approach significantly outperformed the traditional methods for image annotation. Zhang and Wang [59] also use label propagation as a semi-supervised learning algorithm. They propose a lin-

ear neighborhood model, in which each data point can be reconstructed from its neighborhood. Their approach shows promising results for both synthetic and real world data. An extension to label propagation is proposed by Gregory [25]. His algorithm is able to detect overlapping communities and performs especially well in large networks.

The peer pressure algorithm proposed by Gilbert et al. [23] is very similar to label propagation: A subset of all nodes is selected as leaders. Each node in the graph should have at least one leader in its neighborhood. Then, every node in the graph elects a leader, selecting a cluster to join. In the last step, each node switches its vote to the most popular leader in its neighborhood.

Label propagation is also used in the field of graph partitioning. Ugander and Backstrom [55] propose a balanced label propagation algorithm for the partitioning of large graphs. They modify the initialization step of the algorithm. The number of initial labels is equal to the number of desired blocks. The initial labels of the nodes are being randomly selected among this number, respecting the constraints imposed on the block sizes. Utilizing linear programming, their algorithm is able to enforce various constraints on the block sizes (including the balance constraint).

Furthermore, label propagation can be utilized in the coarsening phase and the uncoarsening phase of the multilevel partitioning scheme [41]. In the coarsening phase, nodes sharing the same label are contracted. If left unmodified, the algorithm can produce very large clusters, resulting in very heavy nodes at the coarsest level. These nodes negatively impact on the quality of the partition, since they restrict effectiveness of the local search algorithm in the uncoarsening step. The heavy nodes can not be freely moved between partitions, because their move would violate the balance constraint. Therefore, Meyerhenke et al. [41] introduced a *size constraint*. With this constraint, labels forming a too large cluster are ignored in line 6 of Algorithm 5, i.e. the score function becomes

$$\texttt{score}(v,\sigma) := \begin{cases} 0 & \text{if } \texttt{label}[v] = \sigma, \\ \sum\limits_{\substack{e=(v,u)\in E \\ \texttt{label}[u]=\sigma}} \omega(e), & \text{else if } \mathit{cluster\_size}(\sigma) + c(v) \leq U, \\ -1 & \text{else.} \end{cases} \tag{3.7.3}$$

for some parameter U.

In addition, label propagation can be parallelized: Meyerhenke et al. [40] propose a parallel version of size-constrained label propagation, which scales well for huge instances. Again, they utilize their parallel implementation in the coarsening phase and the refinement phase of a multilevel graph partitioning framework. Finally, Akhremtsev et al. [3] propose a shared memory parallelized, (semi-)external variant of size-constrained label propagation. Their implementation achieves high-quality partitions while having a running time which is comparable to an efficient internal memory implementation, thus allowing the computation of partitions for huge graphs on commodity hardware.

4 | Label Propagation in Hypergraphs

This chapter discusses our adaptions of label propagation to hypergraphs in the coarsening step of the multilevel partitioning scheme. The straightforward way to perform the adaptation is to transform the hypergraph in a graph and perform label propagation on that graph. In Section 2.3.2 we explain the two usual expansions of the hypergraph called clique expansion and star expansion. First, we investigate how label propagation performs on graphs resulting from these expansions. Next, we propose a probabilistic version of label propagation which maintains linear running time complexity per iteration. This chapter concludes with on overview of extensions to our proposed algorithms and the utilization of label propagation in the refinement step of the multilevel partitioning scheme.

4.1. Label Propagation on the Clique Expanded Hypergraph

Clique expansion replaces each hyperedge of size n with a n-clique. As mentioned in Section 2.3.1, given a hypergraph $\mathcal{H} = (V, E, c, \omega)$, the usual edge weights in the clique originating from a hyperedge $e \in E$ are:

$$\frac{\omega(e)}{|e| - 1}. \tag{4.1.1}$$

Besides these weights, we will further investigate how various other uniform and non-uniform edge weights impact the running time and the quality of the clustering.

Note that the default coarsening algorithms (first choice – Section 3.2.3, and absorption clustering using pins – Section 3.2.5) in hMetis and PaToH can be reinterpreted as algorithms on the clique expanded hypergraph, where each edge in a clique resulting from a hyperedge $e \in E$ has the weight shown in Equation 4.1.1.

Since we want to utilize label propagation in the coarsening step of a multilevel hypergraph partitioning framework, we need to make sure that the weights of the hypernodes in the coarser graph are controlled. Like Meyerhenke et al. [41], we impose a size constraint on the clusters, i.e. a tuning parameter U controls the maximal cluster weight. A node ignores labels of clusters whose weight exceeds U, if the node changes its affiliation to that cluster.

In Algorithm 6 the pseudocode for label propagation on the clique expanded hypergraph is shown. Note that the clique expansion is done implicitly in lines 6-8. For each hypernode we compute a score for each adjacent label. Out of these labels, we select the one with the maximal score. In case of multiple labels having the maximal score, each label gets picked with equal probability. We evaluate a total number of 24 different score functions divided into three classes, which are used in line 9 of Algorithm 6:

Algorithm 6: Label Propagation on the Clique Expanded Hypergraph

```
Input: hypergraph H = (V, E, c, ω), |V| = n, |E| = m
Output: label[1 ... n]                                  // labels for each hypernode
for v ∈ V do
    label[v] ← v                                        // initialization
while not converged and num_iterations ≤ max_iterations do
    for v ∈ V in random order do
        tmp_scores[label[v]] ← 0                        // holds scores for adjacent labels
        for e ∈ hyperedges[v] do
            for p ∈ pins[e], p ≠ v do
                if c(v) + ĉ(label[p]) ≤ U then          // size constraint check
                    tmp_scores[label[p]] += score(v, p, e)
        label[v] ← arg max_σ tmp_scores[σ]              // choose max score label
```

Class 1

- $score_1(v, p, e) := \omega(e)$
- $score_2(v, p, e) := \dfrac{\omega(e)}{|e| - 1}$
- $score_3(v, p, e) := \dfrac{\omega(e)}{|\{\texttt{label}[p] \mid p \in pins[e]\}}$

Class 2

- $score_4(v, p, e) := \begin{cases} score_1(v, p, e) & \text{if } \texttt{label}[p] = \max\limits_{\sigma} |\{v \in pins[e] \mid \texttt{label}[v] = \sigma\}| \\ 0 & \text{else.} \end{cases}$
- $score_5(v, p, e) := \begin{cases} score_2(v, p, e) & \text{if } \texttt{label}[p] = \max\limits_{\sigma} |\{v \in pins[e] \mid \texttt{label}[v] = \sigma\}| \\ 0 & \text{else.} \end{cases}$
- $score_6(v, p, e) := \begin{cases} score_3(v, p, e) & \text{if } \texttt{label}[p] = \max\limits_{\sigma} |\{v \in pins[e] \mid \texttt{label}[v] = \sigma\}| \\ 0 & \text{else.} \end{cases}$

Class 3

$$score_{i,k} := score_i(v, p, e) \cdot \psi_k(v, p, e) \qquad (1 \leq i \leq 3),$$

and $\psi_k(v, p, e)$ being one of the following:

- $\psi_1(v, p, e) := \dfrac{1}{c(v) \cdot \hat{c}(\texttt{label}[p])}$
- $\psi_2(v, p, e) := \dfrac{1}{(c(v) \cdot \hat{c}(\texttt{label}[p]))^2}$
- $\psi_3(v, p, e) := \dfrac{1}{c(v) + \hat{c}(\texttt{label}[p])}$
- $\psi_4(v, p, e) := \dfrac{1}{\log(c(v) \cdot \hat{c}(\texttt{label}[p]))}$
- $\psi_5(v, p, e) := \dfrac{1}{\min(c(v), \hat{c}(\texttt{label}[p]))}$
- $\psi_6(v, p, e) := \dfrac{1}{\max(c(v), \hat{c}(\texttt{label}[p]))}$

with $\hat{c}(\texttt{label}[p])$ counting the weight of all hypernodes which have the same label as p:

$$\hat{c}(\texttt{label}[p]) := c(\{v \in V \mid \texttt{label}[v] = \texttt{label}[p]\}).$$

We investigate three main score functions, $score_{\{1,2,3\}}(\cdot, \cdot, \cdot)$, whose modified versions are found in the second and third class. All three score functions assign a uniform weight for all edges in the clique: $score_1(\cdot, \cdot, \cdot)$ assigns each edge in the clique the weight of the original hyperedge, whereas $score_2(\cdot, \cdot, \cdot)$ uses the widespread (hMetis, PaToH) weight shown in Equation 4.1.1. The last score function in the first class, $score_3(\cdot, \cdot, \cdot)$ penalizes hyperedges that span many labels. The incentive behind $score_3(\cdot, \cdot, \cdot)$ is that hyperedges spanning many labels most likely will still be there in the coarser hypergraph. Seeing as we want to reduce the complexity of the problem, i.e. to minimize the weight of hyperedges exposed in the coarser hypergraph, we prefer hyperedges with fewer labels so it is more likely that all pins of these hyperedges settle on the same label. Note that $score_3(\cdot, \cdot, \cdot)$ assigns a non-constant uniform weight for the clique edges, since the number of labels present in a hyperedge changes over the course of an iteration.

The score functions of the second class use $score_{\{1,2,3\}}(\cdot, \cdot, \cdot)$ as weight, but only assign these weights to clique edges whose nodes have the most prevalent label in the original hyperedge. Note that these score functions assign a non-uniform weight to the clique edges. The motivation behind this approach is to improve the convergence rate of label propagation, since only a subset of all incident labels of a hyperedge contribute to the overall score.

Finally, the score functions of class three are further modifications of the score functions of class one, $score_{\{1,2,3\}}(\cdot, \cdot, \cdot)$. They are inspired by Osipov and Sanders [44] and discourage the formulation of large clusters and therefore result in non-uniform edge weights in the clique expanded hypergraph. The motivation behind only medium- and small-sized clusters is that they ease both the initial partitioning algorithm and the local search algorithm, since moderately heavy hypernodes can be moved more freely. We consider six different modifications, which are classified by the second index of the score function. The evaluation of all these score functions is presented in Chapter 6.

Running Time Complexity. The main downside to label propagation on the clique expanded hypergraph is the non-linear running time per iteration. Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ with $|V| = n$ being the number of hypernodes, the running time for a single iteration of Algorithm 6 is:

$$\mathcal{O}(n + \sum_{e \in E} |e|^2) \tag{4.1.2}$$

This proposition applies, because the graph resulting from clique expansion has n nodes and $\sum_{e \in E} |e|^2$ edges. Our algorithm operates on this graph, visiting each node once and each edge twice (once for each of its incident nodes), resulting in the aforementioned running time.

One way to reduce this running time is to ignore large hyperedges in the clique expansion. hMetis employs this approach and ignores all hyperedges $e \in E$ with $|e| > 50$ in their standard configuration[1]. Note that the focus of hMetis is partitioning VLSI instances where such large hyperedges are seldom (since most gates have few inputs). In the general case, omitting hyperedges which are larger than a constant leads to a systematic error: Consider a hypergraph where all hyperedges have cardinality bigger than this constant. In this graph, we would ignore all hyperedges, which results in no coarsening at all. Instead, we try a different approach, ignoring all hyperedges whose cardinality is larger than the 5%-quantile of all hyperedge cardinalities in the hypergraph. While this does not change asymptotic upper bound for the running time of our algorithm (consider again a graph where all edges have uniform cardinality), it significantly decreases the running time if there are only few very large hyperedges.

4.2. Label Propagation on the Star Expanded Hypergraph

Star expansion replaces each hyperedge with a vertex and connects all pins of the hyperedge to that vertex. As mentioned in Section 2.3.2, the edge weights of the "star graph" resulting from a hyperedge are usually uniformly distributed. Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$, this weight was originally [1]:

$$\frac{\omega(e)}{|e|}. \tag{4.2.1}$$

Note that in contrast to clique expansion, star expansion is not frequently used in the field of hypergraph partitioning. This is because the topology of the star expanded hypergraph does not represent the original topology of the hypergraph correctly, e.g. if two hypernodes were connected by a hyperedge in the original hypergraph, they will be not adjacent in the star expanded graph. This results in a loss of quality. Still, we show in the latter part of this section that label propagation on the star expanded hypergraph has a linear time complexity, which is why we further investigate this expansion type.

In Algorithm 7 the pseudocode for label propagation on the star expanded hypergraph is shown. Note that we perform star expansion implicitly. Furthermore, as in Algorithm 6, we impose a size constraint on the clusters. In contrast

[1] Version 2.0pre1: `http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download`

Algorithm 7: Label Propagation on the Star Expanded Hypergraph

```
Input: hypergraph H = (V, E, c, ω), |V| = n, |E| = m
Output: label_nodes[1 ... n]                          // the labels for each hypernode
1  for x ∈ V ∪ E do
2      if x ∈ V then label_nodes[v] ← v                // initialization hypernodes
3      else label_edges[e] ← e                         // initialization hyperedges
4  while not converged and num_iterations ≤ max_iterations do
5      for x ∈ V ∪ E in random order do
6          if x ∈ V then                               // x is a hypernode
7              tmp_scores[label_nodes[x]] ← 0          // scores for adjacent labels
8              for e ∈ hyperedges[x] do
9                  if c(x) + ĉ(label_edges[e]) ≤ U then   // size constraint
10                     tmp_scores[label_edges[e]] += score(x, e)
11             label_nodes[x] ← arg max_σ tmp_scores[σ]
12         else                                        // x is a hyperedge
13             tmp_scores[label_edges[x]] ← 0          // scores for adjacent labels
14             for p ∈ pins[x] do
15                 tmp_scores[label_nodes[p]] += score(p, x)
16             label_edges[x] ← arg max_σ tmp_scores[σ]
```

to label propagation on the clique expanded hypergraph, we allow hyperedge-nodes to change their label disregarding the size constraint. The reason being that hyperedge-nodes are a representation of hyperedges of the original hypergraph and don't contribute their weight to the final clustering and therefore have no impact on the quality of the partitioning. As with label propagation in the clique expanded hypergraph (Section 4.1), we will consider various score functions, which are used in line 10 and line 15 in Algorithm 7. Again, each node selects the label that has the maximal score with random tie-breaking.

Class 1

- $score_1(v, e) := \omega(e)$
- $score_2(v, e) := \dfrac{\omega(e)}{|e|}$

Class 2

$$score_{i,k}(v, w) := \begin{cases} score_i(v, w) \cdot \xi_k(w) & \text{if } v \text{ is a hyperedge-node} \\ score_i(v, w) \cdot \psi_k(v, w) & \text{else} \end{cases}, (1 \leq i \leq 2).$$

with $\psi_k(\cdot)$ and $\xi_k(\cdot, \cdot)$ being defined as follows:

- $\xi_1(v) := \frac{1}{\hat{c}(\texttt{label\_nodes}[v])}$
- $\xi_2(v) := \frac{1}{\hat{c}(\texttt{label\_nodes}[v])^2}$
- $\xi_3(v) := \frac{1}{\hat{c}(\texttt{label\_nodes}[v])}$
- $\xi_4(v) := \frac{1}{\log(\hat{c}(\texttt{label\_nodes}[v]))}$
- $\xi_5(v) := \frac{1}{\hat{c}(\texttt{label\_nodes}[v])}$
- $\xi_6(v) := \frac{1}{\hat{c}(\texttt{label\_nodes}[v])}$
- $\psi_1(v,w) := \frac{1}{c(v)\hat{c}(\texttt{label\_edges}[\omega])}$
- $\psi_2(v,w) := \frac{1}{(c(v)\hat{c}(\texttt{label\_edges}[\omega]))^2}$
- $\psi_3(v,w) := \frac{1}{c(v)+\hat{c}(\texttt{label\_edges}[\omega])}$
- $\psi_4(v,w) := \frac{1}{\log(c(v)\hat{c}(\texttt{label\_edges}[\omega]))}$
- $\psi_5(v,w) := \frac{1}{\min(c(v),\hat{c}(\texttt{label\_edges}[\omega]))}$
- $\psi_6(v,w) := \frac{1}{\max(c(v),\hat{c}(\texttt{label\_edges}[\omega]))}$

with $\hat{c}(\texttt{label}[p])$ counting the weight of all hypernodes that have the same label as p:

$$\hat{c}(\texttt{label}[p]) := c(\{v \in V \mid \texttt{label}[v] = \texttt{label}[p]\}).$$

Note that since hyperedge-nodes do not have a weight in our model, we need a case differentiation in the second score function class. Like in Section 4.1, the score functions of the second class discourage the formulation of large clusters.

Running Time Complexity. The main advantage of label propagation on the star expanded hypergraph is its linear running time complexity. Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ with $|V| = n$ being the number of hypernodes, $|E| = m$ the number of hyperedges and $\sum_{e \in E} |e| = \ell$ being the number of pins, the resulting graph from star expansion has $n + m$ nodes and ℓ edges. Since every node gets visited exactly once and every edge twice (once per incident node), the total running time for one iteration of label propagation on the star expanded hypergraph is:

$$\mathcal{O}(n + m + \ell). \tag{4.2.2}$$

As mentioned before, the downside of this approach is the expected quality loss when compared with clique expansion, since star expansion adds previously nonexistent nodes to the graph and therefore modifies the topology of the original hypergraph.

4.3. Probabilistic Label Propagation

Label propagation on the clique expanded hypergraph promises good results for the cost of a non-linear running time, whereas label propagation on the star expanded hypergraph has a linear running time with the expectation of worse quality. This section discusses our approach, which tries to combine both worlds. That is, a linear time algorithm that operates on the clique expanded hypergraph. The main idea behind our approach is to only look at a fixed number of pins for each hyperedge, i.e. to reduce the hyperedge size to a fixed constant. These pins are chosen uniformly random in the beginning of each iteration. From now on, we refer to these randomly chosen pins as the *sample* for a hyperedge. We prove that score computation on a hyperedges' sample is an *unbiased estimator* for the score distribution for the labels within a hyperedge. Before getting to the actual proof, we cover notations and definitions.

Definition 4.3.1 (Binomial Coefficient)**.** Given $k, n \in \mathbb{N}_0$ with $0 \leq k \leq n$, the *binomial coefficient* is defined as

$$\binom{n}{k} := \frac{n!}{(n-k)!\,k!}$$

One of the most frequent application of the binomial coefficient is in the field of combinatorics: Given a set containing n distinct objects, $\binom{n}{k}$ is the number of all possible distinct k-element subsets of that set.

Definition 4.3.2 (Unbiased Estimator)**.** An *estimator* is a function which infers the value of some unknown parameter in a statistical model of a universe D using a random sample $X_1, \ldots, X_n$ of elements of D. More formally: An estimator is a function that maps the space of all possible samples to a set of sample estimates. Note that if the sample used is a random variable, the estimator becomes a random variable itself. An estimator $\bar{\gamma}(X_1, \ldots, X_n)$ for a parameter γ is called *unbiased* if its expected value is γ for all possible values of γ:

$$\mathbb{E}(\bar{\gamma}(X_1, \ldots, X_n)) = \gamma$$

Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$, a hyperedge $e \in E$, and labels for the pins of e, we show that inferring the score distribution for the labels within e on the basis of a random sample of the pins is an unbiased estimator of the actual label scores of e. For this purpose we reinterpret e as an urn with a total number of $|e|$ different colored balls, which represent the labels present in this hyperedge.

More formally: We set $D := \mathit{pins}[e]$ and draw a sample $\boldsymbol{X} = (X_1, \ldots, X_S)$ of size S from elements of D without replacement, each with the same probability. Let D_i denote the subset of all elements in D with label i and let $d_i := |D_i|$ denote the total number of elements (pins) in D with label i. Therefore,

$$D = \bigcup_{i=1}^{|V|} D_i \qquad \text{and} \qquad |D| = |e| = d = \sum_{i=1}^{n} d_i.$$

Next, we define $|V|$ random variables $\boldsymbol{Y} = (Y_1, \ldots, Y_{|V|})$, which count the total number of elements in the sample with label j using a sum of indicator variables, showing whether X_i has label j:

$$Y_j := \sum_{i=1}^{S} \mathbf{1}\left(\texttt{label}[X_i] = j\right), \qquad 1 \leq j \leq |V|,$$

with

$$\sum_{j=1}^{n} Y_j = S.$$

Lemma 4.3.3 (Probability Mass Function of Y_j)**.** Given d_j, the number of pins with label j in hyperedge e, the sample size S and $|e|$, the probability of there being exactly y pins in the sample with label j is:

$$\mathbb{P}(Y_j = y) = \frac{\binom{d_j}{y}\binom{|e|-d_j}{S-y}}{\binom{|e|}{S}}. \tag{4.3.1}$$

The probability mass function for Y_j is then defined as:

$$f_{Y_j}(y) := \mathbb{P}(Y_j = y), \qquad y \in \mathbb{N}_0,\ 0 \leq y \leq S. \tag{4.3.2}$$

Proof. Y_j is the number of pins in the sample of e with label j and d_j is the total number of pins in e with label j. The elements in the sample are thereby chosen at random, without replacement and all pins in e have the same probability of being chosen.

Therefore, the nominator of Equation 4.3.1, $\binom{d_j}{y} \cdot \binom{|e|-d_j}{S-y}$, computes the total number of ways to select y pins with label j, and $S-y$ pins that don't have label j. Since the order in which the pins are chosen is ignored and we only count the number of pins with a specific label, we need to divide this number by the total number of ways to draw S many pins from e. This leads to Equation 4.3.1. □

In literature, the distribution of Y_j is known as the hypergeometric distribution [39] parametrized with $|e|, d_j$, and S.

Lemma 4.3.4 (Expected Value of Y_j)**.**

$$\mathbb{E}[Y_j] = S \cdot \frac{d_j}{|e|}.$$

Proof. See Appendix A or [39]. □

In Algorithm 8 the pseudocode for our probabilistic label propagation in hypergraphs is shown. Note that it is very similar to Algorithm 6. The main differences are that we sample each hyperedge in each iteration in line 5 and then use the sampled pins instead of all pins in line 9. Furthermore, we perform

Algorithm 8: Probabilistic Label Propagation

```
   Input: hypergraph H = (V, E, c, ω), |V| = n, |E| = m
   Output: label[1 ... n]                          // the labels for each hypernode
1  for v ∈ V do
2      label[v] ← v                                 // initialization
3  while not converged and num_iterations ≤ max_iterations do
4      for e ∈ E do
5          drawNewSample(e)                         // draw new samples in each iteration
6      for v ∈ V in random order do
7          tmp_scores[label[v]] ← 0                 // holds scores for adjacent labels
8          for e ∈ hyperedges[v] do
9              for p ∈ sample(e), p ≠ v do          // use pins in sample
10                 if c(v) + c(label[p]) ≤ U then   // size_constraint check
11                     tmp_scores[label[p]] += score(v, p, e)
12         max_label ← arg max_σ tmp_scores[σ]      // choose max score label
13         if gain(v, max_label) ≥ 0 then           // check if new label makes sense
14             label[v] ← max_label
```

a check in line 13, which validates if the new label makes sense, since it is possible that the samples for the incident hyperedges were drawn poorly, i.e.:

$$gain(v, \sigma) := \sum_{e \in hyperedges[v]} \begin{cases} 0 & \text{if } \texttt{label}[v] = \sigma \\ -\omega(e) & \text{if } \texttt{label}[v] \text{ is the only label in } e \\ \omega(e) & \text{if } \texttt{label}[v] \text{ occurs once and } \sigma \text{ occurs in } e \end{cases}. \tag{4.3.3}$$

The function $gain(\cdot, \cdot)$ becomes negative if there are many hyperedges that do not contain the new label. This results in new cut hyperedges, which we want to avoid. As in Algorithm 6 and Algorithm 7, each hypernode chooses the label that has maximal score in its neighborhood, with ties being broken randomly.

Next, we modify the score functions used in the label propagation on the clique expanded hypergraph (Section 4.1) in line 11 in such a way that the score distribution in the sample becomes an unbiased estimator for the score distribution in the overall hyperedge:

Lemma 4.3.5 (Unbiased Score Estimation in the Sample of a Hyperedge)**.** Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$, a hypernode $v \in V$, an incident hyperedge $e \in E$ of v, and the sample for e, $sample(e)$ with size S, we define the score for $p \in sample(e)$ as:

$$\overline{score}_x(v, p, e) := score_x(v, p, e) \cdot \frac{|e|}{S}, \quad x \in \{1, \ldots, 12\}. \tag{4.3.4}$$

These modified score functions are unbiased estimators for the score functions presented in Section 4.1.

Proof. Consider label propagation without size constraint on the clique expanded hypergraph: Given a hypernode $v \in V$ and a hyperedge $e \in E$, the final score for a label i in e is:

$$total\_score\_clique(v, i, e) := \sum_{\substack{p \in pins[e], \\ \texttt{label}[p] = i}} score(v, p, e), \tag{4.3.5}$$

where $score(v, p, e)$ is one of the scores defined in Section 4.1. The size constraint is left out for the sake of simplicity, since it unnecessarily enlarges Equation 4.3.5 and does not change our argumentation.

Let Y_i and d_i be defined as above (with Y_i counting the total number of pins the sample of e with label i and d_i counting the total number of pins in the hyperedge e with label i). Note that since all utilized score functions only use the label of a pin, we can rewrite Equation 4.3.5:

$$total\_score\_clique(v, i, e) := d_i \cdot score(v, \eta, e), \tag{4.3.6}$$

with $\eta \in pins[e] : \texttt{label}[\eta] = i$ denoting one arbitrary pin of e with label i. On the other hand, the total score for a label i in hyperedge e in the probabilistic algorithm is:

$$total\_score\_sample(v,i,e) := \sum_{\substack{p \in sample(e),\\ \texttt{label}[p]=i}} \overline{score}(v,p,e) \tag{4.3.7}$$

$$= \frac{|e|}{S} \cdot \sum_{\substack{p \in sample(e),\\ \texttt{label}[p]=i}} score(v,p,e) \tag{4.3.8}$$

$$= \frac{|e|}{S} \cdot Y_i \cdot score(v,\eta,e), \tag{4.3.9}$$

with $\eta \in sample(e) : \texttt{label}[\eta] = i$ denoting one arbitrary pin in the sample of e with label i. The expected value of this score is:

$$\mathbb{E}(total\_score\_sample(v,i,e)) = \frac{|e|}{S} \cdot \mathbb{E}(Y_i) \cdot score(v,\eta,e) \tag{4.3.10}$$

$$= \frac{|e|}{S} \cdot S \cdot \frac{d_i}{|e|} \cdot score(v,\eta,e) \tag{4.3.11}$$

$$= d_i \cdot score(v,\eta,e) \tag{4.3.12}$$

$$= total\_score\_clique(v,i,e). \tag{4.3.13}$$

The first equality holds because of the linearity of the expected value and the second equality holds because of Lemma 4.3.4. □

Running Time Complexity. Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ with $|V| = n$ being the number of hypernodes, $\sum_{e \in E} |e| = \ell$ being the number of pins, and sample size S, the total running time for one iteration of probabilistic label propagation is:

$$\mathcal{O}(|E| \cdot S + n + \ell \cdot S). \tag{4.3.14}$$

The term comes together as follows: In each iteration we first sample all hyperedges. This is bound by the number of samples for each hyperedge $|E| \cdot S$. Next, consider the graph where all hyperedges are removed and each pin in a hyperedge $e \in E$ gets an edge to all pins in the sample of e (See Figure 4.1). Note that since this transformation is done implicitly we can not merge parallel edges. Therefore, this graph has n nodes and $m = \ell \cdot S$ edges. One iteration of our probabilistic label propagation can be reinterpreted as usual label propagation on this graph. The running time for one iteration of label propagation on a graph with n nodes and m edges is $\mathcal{O}(n+m)$ (Equation 3.7.1). In conclusion, Equation 4.3.14 follows.

4.4. Algorithmic Extensions

This section discusses our algorithmic extensions for label propagation in hypergraphs. We first investigate how the order of traversal affects the quality and running time of our proposed algorithms in Section 4.4.1. We discuss and adapt the V-cycle technique, which achieves a better quality of the partitioning for the cost of an increased running time in Section 4.4.2. Next, we propose an adaptive stopping critereon in Section 4.4.3. Finally, this section concludes with the

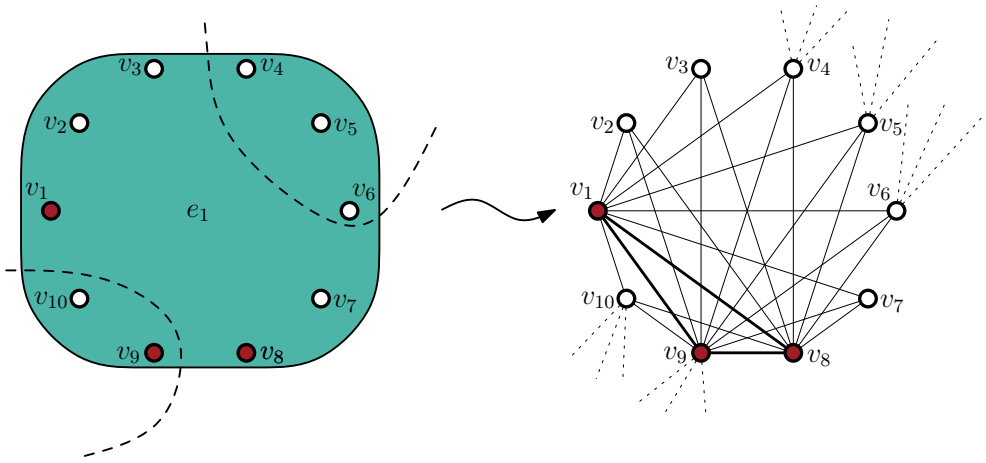


Figure 4.1.: *One iteration of probabilistic label propagation on a hypergraph* $\mathcal{H} = (V, E, c, \omega)$ *can be reinterpreted as label propagation on a subgraph of the clique expanded hypergraph. On the left a hyperedge* $e_1 = \{v_1, \ldots, v_{10}\}$ *is shown. The samples are depicted in red with three being the sample size. On the right, the subgraph is shown. Note that every node besides the nodes in the sample has a degree of three (for this hyperedge). The nodes in the sample ignore all edges that don't connect two nodes in the sample. The thick lines represent the non ignored edges.*

adaptation of label propagation in hypergraphs as local search strategy in the refinement step of the multilevel partitioning scheme (Section 4.4.4).

4.4.1. Node Ordering

All our proposed label propagation adaptations visit the hypernodes in random order and determine the new label for the hypernode according to a score function on adjacent hypernodes and the hyperedge connecting them. Instead of a random traversal, Meyerhenke et al. [41] propose to use an ordering induced by the node degree (increasing). The motivation behind this ordering is that if nodes with a small degree determine their new labels before nodes with a large node degree, the latter already see a cluster structure in their neighborhood when they are visited. This likely results in a better clustering. We adapt their approach to hypergraphs and investigate the impact of other orderings. Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ and a hypernode $v \in V$, we evaluate the following orderings in Chapter 6:

- $order_1(v) := deg(v)$
- $order_2(v) := deg(v) + \sum_{hyperedges[v]} |e|$
- $order_3(v) := deg(v) \cdot (\sum_{hyperedges[v]} |e|)$
- $order_4(v) := deg(v)^2 \cdot (\sum_{hyperedges[v]} |e|)$
- $order_5(v) := deg(v) \cdot \log(\sum_{hyperedges[v]} |e|)$

$order_1(\cdot)$ is the ordering employed by Meyerhenke et al. [41], whereas the other ordering functions penalize hypernodes whose incident hyperedges have a large cardinality. Note that in case of label propagation on the star expanded hypergraph, we need to modify this orderings for hyperedge-nodes:

- $order_1(w) := \begin{cases} |w| & \text{if } w \text{ is a hyperedge-node} \\ deg(w) & \text{else} \end{cases}$
- $order_2(w) := \begin{cases} |w| + \sum_{v \in pins[w]} deg(v) & \text{if } w \text{ is a hyperedge-node} \\ deg(w) + \sum_{e \in hyperedges[w]} |e| & \text{else} \end{cases}$
- $order_3(w) := \begin{cases} |w| \cdot (\sum_{v \in pins[w]} deg(v)) & \text{if } w \text{ is a hyperedge-node} \\ deg(w) \cdot (\sum_{e \in hyperedges[w]} |e|) & \text{else} \end{cases}$
- $order_4(w) := \begin{cases} |w|^2 \cdot (\sum_{v \in pins[w]} deg(v)) & \text{if } w \text{ is a hyperedge-node} \\ deg(w)^2 \cdot (\sum_{e \in hyperedges[w]} |e|) & \text{else} \end{cases}$
- $order_5(w) := \begin{cases} |w| \cdot \log(\sum_{v \in pins[w]} deg(v)) & \text{if } w \text{ is a hyperedge-node} \\ deg(w) \cdot \log(\sum_{e \in hyperedges[w]} |e|) & \text{else} \end{cases}$

4.4.2. V-cycles

V-cycles or iterated multilevel algorithms is a term which generally describes the usage of an already computed partition during the multilevel partitioning scheme, i.e. during the coarsening and refinement phase. It was introduced by Walshaw [57] and further augmented by Sanders and Schulz [49] to more complex cycles called W-cycles and F-cycles.

In V-cycling, the multilevel partitioning scheme is repeated several times and once a partition is computed, edges that span multiple partitions are ignored during coarsening, i.e. these edges won't be contracted or, as in our case, pins belonging to a different block than the node are ignored in the score computation. Furthermore, once a partition is computed in the first iteration, we don't use the initial partitioning algorithm, but simply assign nodes to the current partition in subsequent iterations. Note that the quality of the partition can't decrease in subsequent iterations. This is because contractions are only performed inside the partition, the partition of the previous iteration is inherited, and the local search algorithm in the refinement phase only improves the partition. This leads to high quality partitions. The number of V-cycles is thereby a tuning parameter.

4.4.3. Adaptive Stopping Rule

In the original version of label propagation [47] the stopping criteria are the maximal number of iterations and the check if all nodes have a label that the maximal number of their respective neighbors belong to. In hypergraphs, the validation of the latter constraint is not feasible, since the running time for this step is

$$\mathcal{O}(n + \sum_{e \in E} |e|^2). \tag{4.4.1}$$

Algorithm 9: Label Propagation as Local Search Strategy

```
Input: hypergraph H = (V, E, c, ω), partition Π = {V_1, ..., V_k},
       uncoarsened hypernodes {w_1, ..., w_j}
Output: Π' = {V_1, ..., V_k}                                  // refined partition
1  Q_1 ← {w_1, ..., w_j}                          // set of nodes for the current iteration
2  Q_2 ← {}                                        // set of nodes for the next iteration
3  while Q_1 ≠ {} and num_iterations ≤ max_iterations do
4      for v ∈ Q_1 in random order do
5          tmp_gains[λ(v)] ← 0                    // holds gains for incident blocks
6          for b ∈ incident blocks do
7              for e ∈ hyperedges[v] do
                   // enforce balance criterion
8                  if c(v) + c(b) ≤ L_max then
9                      tmp_gains[b] += gain(v, e, b)      // gain if v's block was b
10         max_block ← choose_block(tmp_gains)   // choose block with max gain
11         if λ(v) ≠ max_block then              // check if v changes its block
12             move v to max_block
               // add adjacent pins to next iteration
13             for e ∈ hyperedges[v] do Q_2 = Q_2 ⋃ pins[e]
14     Q_1 ← {}
15     swap(Q_1, Q_2)
```

The proposition applies, because the verification of these criteria can be reinterpreted as one traversal on the clique expanded hypergraph, which has n nodes and $\sum_{e \in E} |e|^2$ edges.

We propose a different stopping rule: Our adaptations of label propagation to hypergraphs perform at most a constant number of iterations and stop if less than 5% of all hypernodes changed their label in the last iteration.

4.4.4. Label Propagation as Local Search Strategy

This section discusses our adaptation of label propagation as local search strategy in the refinement phase of the multilevel partitioning scheme. Meyerhenke et al. [41] proposed to use label propagation as a fast alternative to Kerninghan-Lin (KL) and KL-variants like Fiduccia-Mattheyses (FM). Instead of evaluating multiple scores, we will use only one that represents the improvement of the quality of the partition, if the current hypernode were to change its affiliation to the block represented by this label. This score will be referred to as the *gain* of a label. In Algorithm 9 the pseudocode for label propagation based local search is shown. Note that there are many differences to the usual label propagation. First, we don't iterate through all hypernodes of the hypergraph, but only consider those which have been uncontracted (in the first iteration) or have a neighbor that has changed its block (consecutive iterations). For the sake of simplicity of the next argument, assume that there is no tie-breaking. Therefore, the block of

a hypernode remains unchanged, if all adjacent hypernodes did not change their block. So there is no need to iterate through all hypernodes, but only a subset of them. This idea is implemented with two hypernode sets which represent the currently active hypernodes and hypernodes that could possibly change their block in the next iteration, because one of its neighbors has changed its block. After the end of one iteration, we clear the first set and swap it with the second one.

Furthermore, we need to modify the size constraint, since we want to enforce an ε-balanced partitioning. This is done in line 8 of the algorithm. As mentioned above, we need to modify our score function to reflect our partitioning objective. Since we use the hypergraph cut objective, our goal is to minimize the sum of all hyperedge weights that span multiple blocks. Therefore, given a block and a hyperedge, our gain function in line 9 returns change of the cut for this hyperedge, if the current node were to change its block, i.e.:

$$\mathit{gain}(v, \mathit{block}, e) := \begin{cases} -\omega(e) & \text{if} \quad \textit{connectivity(e) = 1} \ \textbf{and} \\ & \quad \mathit{block} \notin \mathit{connectivity\_set(e)} \\ \omega(e) & \text{else if } \textit{connectivity(e) = 2} \ \textbf{and} \\ & \quad |\{p \in \mathit{pins}[e] \mid \lambda(p) = \lambda(v)\}| = 1 \\ 0 & \text{else} \end{cases}, \tag{4.4.2}$$

with $\lambda(\cdot)$ denoting the indicator function, which returns the block of a hypernode.

Finally, we employ a different type of tie-breaking (if multiple blocks have the maximal score). As usual, we determine which labels have the maximal score. Out of these, we randomly select one that would reduce the connectivity of incident hyperedges the most:

```
1 Procedure choose_block(label_scores[·])
2 |  max_labels ← {σ | σ = arg max_σ label_scores[σ]}    // max score labels
3 |  max_reduce ← {σ ∈ max_labels | σ = arg max_σ decrease(σ)}
  |  // return random block with max connectivity decrease
4 |  return random_element(max_reduce)
```

with $\mathit{decrease}(\sigma)$ denoting the connectivity decrease in the hypergraph, if the current hypernode would choose block σ. A move of a hypernode $v \in V$ to block σ thereby decreases connectivity of a hyperedge $e \in E$, iff v is the only pin of e that belongs to block $\lambda(v)$ and there exist nodes that belong to block σ in e.

5 | Implementation Details

This chapter presents implementation specific details of our proposed adaptations for label propagation in hypergraphs. First, we discuss various representations of hypergraphs in memory in Section 5.1. Next we discuss the sampling process for our probabilistic label propagation algorithm in detail in Section 5.2. The chapter concludes with the description of our maximal connectivity tie-breaking for label propagation as local search strategy (Section 5.3).

5.1. Representation of Hypergraphs

This section focuses on the representation of hypergraphs in memory, i.e. how hypergraphs should be represented to allow efficient storage and processing. In the following, $\mathcal{H} = (V, E, c, \omega)$ denotes an arbitrary hypergraph and $n = |V|$, $m = |E|$ for the number of hypernodes and hyperedges respectively. Throughout this section, we assume that hypernode weights and hyperedge weights are constant for all hypernodes and hyperedges respectively, i.e. $c \equiv 1$ and $\omega \equiv 1$. Therefore, we do not need to store these weights explicitly. For general hypergraphs not fulfilling this property, the weights can be stored efficiently in two arrays $\mathit{hypernode\_weights}[1 \dots n]$ and $\mathit{hyperedge\_weights}[1 \dots m]$.

5.1.1. Incidence Matrix

A straightforward representation for hypergraphs is the *incidence matrix* [14, 20], $Q(\mathcal{H})$, which has n rows and m columns. An entry q_{ij} in the matrix is one, iff v_i is incident to the hyperedge e_j and zero otherwise. Formally:

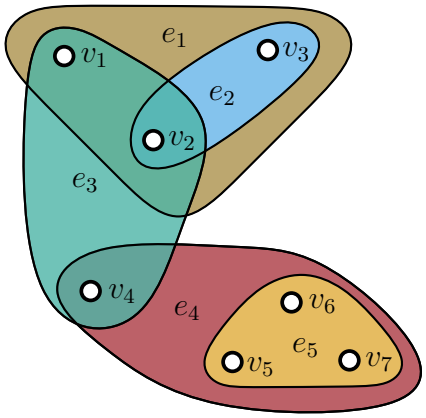
Definition 5.1.1. Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ with n hypernodes and m hyperedges we define the *incidence matrix* of $\mathcal{H}$ as $Q(\mathcal{H}) := (q_{ij}) \in \mathbb{Z}^{n \times m}$ with:

$$q_{ij} := \begin{cases} 1 & \text{if } v_i \in \mathit{pins}[e_j] \\ 0 & \text{else.} \end{cases} \tag{5.1.1}$$

Thus, every boolean matrix can be reinterpreted as a hypergraph. The only difference to the incidence matrices originating from graphs is that each column can have more than two non-zero entries, since a hyperedge can connect more than two nodes. The memory consumption with this approach is

$$\mathcal{O}(|V| \cdot |E|).$$

The representation of hypergraphs as an incidence matrix does not lose any information about the topological properties of the hypergraph. Still, the memory consumption is very large. Figure 5.1 shows the incidence matrix for the example hypergraph (Figure 2.1).



$$Q(\mathcal{H}) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Figure 5.1.: *The incidence matrix, $Q(\mathcal{H})$, for the example hypergraph (Figure 2.1). This matrix has n rows, one for each hypernode and m columns, one for each hyperedge. An entry e_{ij} in the incidence matrix is one iff v_i is incident to e_j and zero otherwise. The sum of non-zero entries in each column corresponds to the size of the hyperedge being represented by the column. Analogously, the sum of non-zero entries in each row corresponds to the hypernode degree of the hypernode represented by this row.*

5.1.2. Incidence Array

Like adjacency arrays for normal graphs, the *incidence array* allows for a compact representation of the hypergraph. We transform the hypergraph to its bipartite representation (Section 2.3.2), replacing each hyperedge with a new node and connecting each pin to it. Next, we represent this graph as an adjacency array. For the sake of clarity, we split the vertex array into two, one for the hypernodes and one for the "hyperedge-nodes". Formally:

Definition 5.1.2. Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ with n hypernodes, m hyperedges, and ℓ pins, we define the *incidence array* as a tuple

$$(\mathbf{VA}[1 \dots n+1], \mathbf{IE}[1 \dots m+1], \mathbf{EA}[1 \dots \ell+1], \mathbf{IE}[1 \dots \ell+1])$$

with four arrays:

- vertex array ($\mathbf{VA}[1 \dots n+1]$)
- incident edges array ($\mathbf{IE}[1 \dots \ell+1]$)
- edge array ($\mathbf{EA}[1 \dots m+1]$)
- incident vertices array ($\mathbf{IV}[1 \dots \ell+1]$)

with:

$$\forall v_i \in V : \mathit{hyperedges}[v_i] = \{\mathbf{IE}[x] \mid \mathbf{VA}[v_i] \leq x < \mathbf{VA}[v_{i+1}]\},$$
$$\forall e_j \in E : \mathit{pins}[e_j] = \{\mathbf{IV}[x] \mid \mathbf{EA}[e_j] \leq x < \mathbf{EA}[e_{j+1}]\}.$$

In other words: for a hypernode v_i all incident hyperedges are stored in the incident edges array, $\mathbf{IE}[\mathbf{VA}[v_i] \dots \mathbf{VA}[v_{i+1}-1]]$. For a hyperedge e_j, all incident

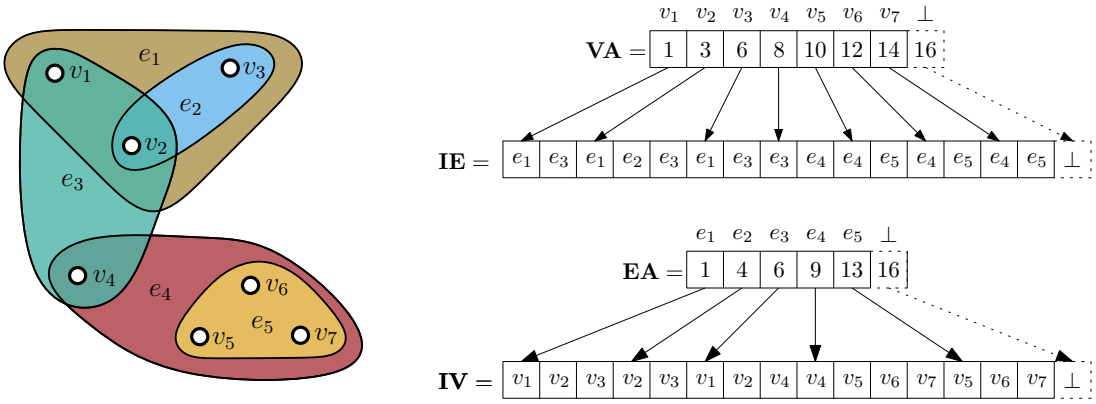


Figure 5.2.: *The incidence array for the example hypergraph (Figure 2.1). Analogous to a adjacency array for graphs, the incidence array stores the incident hyperedges and pins for each hypernode and each hyperedge.*

pins are stored in the incident vertices array, $\mathbf{IV}[\mathbf{EA}[e_j] \dots \mathbf{EA}[e_{j+1} - 1]]$. The memory consumption with this approach is

$$\mathcal{O}(|V| + \sum_{v \in V} deg(v) + |E| + \sum_{e \in E} |e|) = \mathcal{O}(n + m + 2\ell).$$

Figure 5.2 shows the incidence array for the example hypergraph (Figure 2.1). We will use the incidence array as hypergraph representation, since its memory consumption is small (linear in the number of pins) and allows for efficient access to incident hypernodes and hyperedges and adjacent hypernodes.

5.2. Uniform Sampling of Pins in a Hyperedge

In our probabilistic version of label propagation (Algorithm 8) we first draw a sample for each hyperedge at the beginning of each iteration. This section discusses our implementation of this sampling.

Note that all our considered score functions only utilize the label of a pin. Therefore we first need to decide what we want to store in the sample: The pins or their labels. Both approaches have benefits and drawbacks. If we decide to store pins in the samples, it is likely that our algorithm suffers from many *cache misses*. This is because the labels for hypernodes are stored separately in an array. Since hypernodes are usually part of multiple hyperedges, they occur multiple times as a pin. Therefore, we cannot rearrange the order of hypernodes in the labels array to correspond to the order of pins in the sample for all hyperedges. This results in random access to the labels array if we iterate through all pins in the sample.

We can solve this problem if we directly store the labels in the samples for each hyperedge. However, this approach duplicates information: We still have an array which holds the labels for each hypernode. Furthermore, each hyperedge stores the labels of the current sample. This is a problem, because we need to make sure that the labels in all hyperedges are consistent with the labels stored in the array, i.e. if a hypernode changes its label, we need to update its label

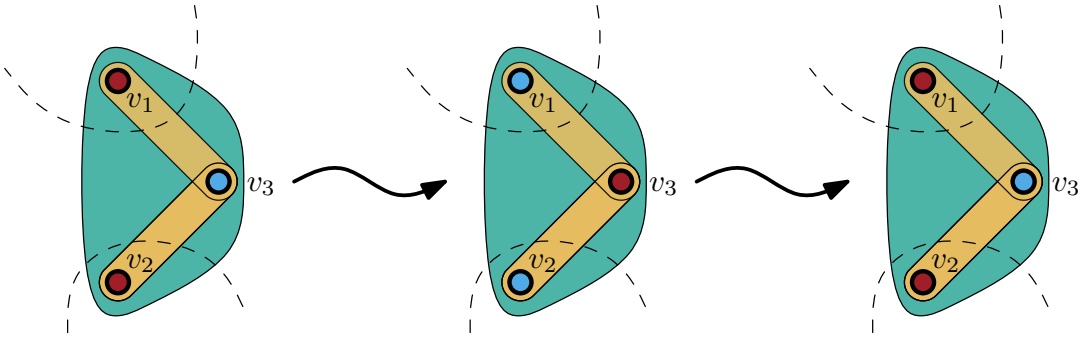


Figure 5.3.: *If we don't utilize the current labels in label propagation, oscillations occur. Each of the three images depicts a segment of a hypergraph through different iterations of label propagation, whereby the labels of each hypernode are color-coded. Both v_1 and v_2 will change their label to the label of v_3 in all iterations, because they are connected to v_3 via two different hyperedges. If we only use the 'old' labels (i.e. the labels in the beginning of each iteration), v_3 will also change its label, because it is the last hypernode with its old label. This leads to oscillations.*

in all incident hyperedges where the hypernode was part of the sample. This is necessary because if we neglect this update step, oscillations occur (Figure 5.3) which impede the quality of the clustering. However, this update step needs further information, which needs to be stored for each hyperedge. Apart from that, this update step comes with an additional computational cost: After computing the new label for a hyperedge, we need to visit all incident hyperedges and update the label in the sample (if the hypernode was sampled in this particular hyperedge).

We implemented the second approach. In the following we give a brief overview on the used data structures. Given a hypergraph $\mathcal{H} = (V, E, c, \omega)$ and the sample size S, we store for each hyperedge $e \in E$:

- an array with the incident labels (`incident_labels`$[1 \dots |e|]$),
- an array with the sampled labels (`sample`$[1 \dots \min(S, |e|)]$),
- an array that holds the location of the label found in `incident_labels`$[i]$ in the sample, (`loc_incident_labels_in_sample`$[1 \dots |e|]$),
- a hash map `label_count_map` which returns, given a label, the number of incident pins with this label.

Furthermore, for each hypernode $v \in V$ we store:

- an integer representing its label (`label`),
- an array, `loc_incident_edges`$[1 \dots deg(v)]$, holding the location of this hypernode's label in the array `incident_labels`$[\cdot]$ in the incident hyperedges.

The hash map `label_count_map` is mainly used for the validation of a new label in the probabilistic version of label propagation (Equation 4.3.3). Note that if a

hypernode v changes its label, the update step can be efficiently performed with the `loc_incident_labels_in_sample`[$\cdot$] and `loc_incident_edges`[$\cdot$] data structures as follows: We iterate through all incident hyperedges $e \in \mathit{hyperedges}[v]$ of v and check whether v was part of the sample. This can be done by validating the `loc_incident_labels_in_sample`[`loc_incident_edges`[e]] entry, i.e. if it is a valid index in the sample. If this is the case, we simply update the entry in the `sample`[$\cdot$] array pointed to by this index and in `incident_labels`[$\cdot$]. Finally, we need to update the hash map, since the number of pins has changed for this hyperedge. Overall, all update steps for a single iteration take $\mathcal{O}(\sum_{v \in V} \mathit{deg}(v)) = \mathcal{O}(\ell)$ time, where ℓ is the number of pins.

Next, given these data structures per hyperedge and per pin, we explain how our uniform selection of labels for the sample is implemented. Let $e \in E$ be a hyperedge that we want to sample and $x := \min(S, |e|)$ the number of samples. We draw x random numbers $\theta_1, \ldots, \theta_x$, whereby the i-th number is drawn uniformly from $[1, |e| - i + 1]$, e.g. the first number is selected uniformly from $[1, |e|]$. Then, we add `incident_labels`[θ_i] as the i-th label to the sample. Next, we swap `incident_labels`[$|e| - i + 1$] with `incident_labels`[θ_i]. This has to be done because we sample without repetition (Section 4.3) and it eases the computation of the other samples. Since the information stored for hyperedges needs to be consistent with the information stored for hypernodes, we also need to swap `loc_incident_labels_in_sample`[$|e| - i + 1$] with `loc_incident_labels_in_sample`[θ_i] and update the location information for this hyperedge in the sampled pin (in its `loc_incident_edges`[$\cdot$] array). All in all, the total time consumption for the sampling of a hyperedge $e \in E$ is bound by $\mathcal{O}(\min(S, |e|))$.

5.3. Global Maximal Connectivity Decrease Tie-Breaking

In our version of label propagation as local search strategy (Algorithm 9) we employed a more sophisticated tie-breaking rule: In case of multiple labels (blocks) having the maximal score, we select the one which leads to the maximal connectivity decrease in the hypergraph.

The general work flow of this algorithm can be divided into two parts: For each active hypernode we iterate through all incident hyperedges and compute for each incident block the gain we would obtain if we were to move this hypernode to that block. The result of this computation is stored in the `tmp_gains`[$1 \ldots k$] array. In the second step we select the block that has the maximal gain and use the aforementioned tie-breaking rule.

For an efficient computation of the global maximal connectivity decrease we first assume that each move of a hypernode v increases the global connectivity by $\mathit{deg}(v)$ for all blocks. Furthermore, we use an array that contains temporary values per block, which will be used for the computation of the global connectivity decrease if v is moved to a block (`tmp_decrease`[$1 \ldots k$]). This array is initialized with our assumption, $-\mathit{deg}(v)$, for each block. Next, while we iterate through the incident hyperedges $e \in \mathit{hyperedges}[v]$ and incident blocks of v in the first

part of the algorithm, we modify this array:

$$\texttt{tmp\_decrease}[block] \mathrel{+}= con\_decrease(e, v, block) \tag{5.3.1}$$

with

$$con\_decrease(e, v, block) := \begin{cases} 1 & \text{if} \quad |\{p \in pins[e] \mid \lambda(p) = \lambda(v)\}| = 1 \textbf{ and} \\ & \quad block \in connectivity\_set(e) \\ 1 & \text{else if } |\{p \in pins[e] \mid \lambda(p) = \lambda(v)\}| > 1 \textbf{ and} \\ & \quad block \in connectivity\_set(e) \\ 0 & \text{else} \end{cases}, \tag{5.3.2}$$

Finally, let κ denote the number of incident hyperedges of v, where v is the last pin belonging to the block of v:

$$\kappa := |\{e \in hyperedges[v] \mid |\{p \in pins[e] \mid \lambda(p) = \lambda(v)\}| = 1\}| \tag{5.3.3}$$

The global connectivity decrease for a move of v to block b is then given as:

$$\kappa + \texttt{tmp\_decrease}[b] \tag{5.3.4}$$

In the following, we argue why Equation 5.3.4 holds. In the beginning we assume that each hyperedge increases its connectivity by one if v were moved to a different block. Given a hyperedge e and a block b, there are three possibilities: First, our assumption was right and if v was moved to the block, the connectivity of the hyperedge would increase by one. This is exactly then the case if v was not the last pin in the hyperedge belonging to the old block and there are no pins in the hyperedge belonging to the new block. Formally:

$$|\{p \in pins[e] \mid \lambda(p) = \lambda(v)\}| > 1 \quad \wedge \quad b \notin connectivity\_set(e). \tag{5.3.5}$$

In this case, there is no need to modify the entry of our array. The next possibility is that our assumption was wrong and the connectivity of the hyperedge remains unchanged. This is the case if either there are other pins besides v belonging to the old block and there exist pins in the hyperedge belonging to the new block, or if v was the last pin belonging to the old block and no other pin belongs to the new block. Formally:

$$|\{p \in pins[e] \mid \lambda(p) = \lambda(v)\}| > 1 \quad \wedge \quad b \in connectivity\_set(e) \quad \vee \tag{5.3.6}$$
$$|\{p \in pins[e] \mid \lambda(p) = \lambda(v)\}| = 1 \quad \wedge \quad b \notin connectivity\_set(e). \tag{5.3.7}$$

In this case, since the connectivity of the hyperedge doesn't change, we need to increment the entry $\texttt{tmp\_decrease}[b]$. Note that in Equation 5.3.2 we only check for the first condition. We will explain the reason for this shortly, but first let us consider the third possibility: The connectivity of e gets reduced by one. This is then the case if v was the last pin of e belonging to the old block and there exist pins in e belonging to the new block, b. Formally:

$$|\{p \in pins[e] \mid \lambda(p) = \lambda(v)\}| = 1 \quad \wedge \quad b \in connectivity\_set(e). \tag{5.3.8}$$

Algorithm 10: Computation of Connectivity Decrease of a Move
`Input`: hypergraph $\mathcal{H} = (V, E, c, \omega)$, partition $\Pi = \{V_1, \ldots, V_k\}$, node v
`Output`: connectivity decrease per block, `tmp_decrease`$[\cdot]$

```
1 tmp_decrease[1...k] = [−deg(v) ... − deg(v)]
2 κ ← 0
3 for e ∈ hyperedges[v] do
4     if |p ∈ pins[e] | λ(p) = λ(v)| = 1 then
5         κ += 1
6     for b ∈ connectivity_set(e) do
7         tmp_decrease[b] += 1
8 tmp_decrease[1...k] += [κ ... κ]
```

In this case we also need to change the entry `tmp_decrease`$[b]$. Since we assumed that e gets its connectivity increased by one, we need to add two to that entry. Plus one since the connectivity does not increase and plus one because it actually decreases. Note that in Equation 5.3.2 we check for this condition but only increment the entry by one. We will now explain our reasoning for this.

It is possible that v is the only pin in a hyperedge belonging to the old block, $\lambda(v)$. Formally:

$$|\{p \in pins[e] \mid \lambda(p) = \lambda(v)\}| = 1. \tag{5.3.9}$$

In this case the connectivity of this hyperedge can not increase regardless of the new block. This is a problem, since in the beginning we assumed that the connectivity of each hyperedge increases. To cope with this problem we count the number of such hyperedges with κ and add this number to the final connectivity decrease.

However, it is possible that we count the decrease multiple times. Namely, if the move decreases the connectivity (Equation 5.3.8) and if the move doesn't increase connectivity, but v was the last pin with the old block (Equation 5.3.7). In both cases, we need to adjust the value in `tmp_decrease`$[b]$ by -1. Considering these modifications we gain Equation 5.3.2 and Equation 5.3.4. In Algorithm 10 the (optimized) pseudocode for the computation of `tmp_decrease`$[\cdot]$ is shown.

6 Evaluation

This chapter presents the evaluation of our adaptation of label propagation to hypergraphs. First, we describe the hardware details of the machine used for the evaluation in Section 6.1. Next, we cover usual benchmark data sets used in hypergraph partitioning in Section 6.2. In Section 6.3 we briefly cover the architecture of the multilevel partitioning framework in which we integrated label propagation as coarsening strategy as well as local search strategy. The chapter concludes with an overview of parameter optimization in Section 6.4 and the presentation of experimental results in Section 6.5.

6.1. Platform Description

We implemented our proposed algorithms in C++, compiled them with gcc version 4.9.2 and all optimization flags turned on: `-std=c++14 -O3 -mtune=native -march=native`. The utilized system is running Red Hat Enterprise Linux (RHEL) 6.4 with the 2.6.32-431.46.2.el6.x86_64 kernel. It has two Octa-core Intel Xeon processors E5-2670 (Sandy Bridge) which run at a clock speed of 2.6 GHz and have 8x256 KB of level 2 cache and 20 MB level 3 cache. The system machine has 64 GB of main memory.

6.2. Data Sets

The most widely used data sets for evaluation in hypergraph partitioning can be divided into two groups: hypergraphs originating from VLSI instances and hypergraphs originating from sparse matrices.

6.2.1. VLSI Instances

As mentioned in Section 3.6.1, in case of VLSI instances each hypernode represents a gate and hyperedges groups inputs/outputs of gates together. The most popular benchmark data sets of this kind is the ISPD98 Circuit Benchmark Suite [5] and the MCNC benchmark suite [38]. In Table 6.1 and Table 6.2 the general properties of the hypergraphs in the IPSD98 benchmark suite and the MCNC benchmark suite are shown. Note that the MCNC benchmark suite has some very small hypergraphs. We will ignore these instances in our evaluation. For a more detailed analysis (e.g. average hypernode degree, average hyperedge size) of these hypergraphs see Appendix C.

6.2.2. SPM Instances

Sparse matrix instances are also widely used for evaluation of hypergraph partitioning. Given a matrix $A \in \mathbb{Z}^{n \times m}$ we thereby usually interpret columns or rows as hyperedges as follows: The hyperedge representing row/column i contains all

Hypergraph	$\|V\|$	$\|E\|$	$\|pins\|$
ibm01	12 752	14 111	50 566
ibm02	19 601	19 584	81 199
ibm03	23 136	27 401	93 573
ibm04	27 507	31 970	105 859
ibm05	29 347	28 446	126 308
ibm06	32 498	34 826	128 182
ibm07	45 926	48 117	175 639
ibm08	51 309	50 513	204 890
ibm09	53 395	60 902	222 088
ibm10	69 429	75 196	297 567
ibm11	70 558	81 454	280 786
ibm12	71 076	77 240	317 760
ibm13	84 199	99 666	357 075
ibm14	147 605	152 772	546 816
ibm15	161 570	186 608	715 823
ibm16	183 484	190 048	778 823
ibm17	185 495	189 581	860 036
ibm18	210 613	201 920	819 697

Table 6.1.: The ISPD98 Circuit Benchmark Suite contains 18 hypergraphs.

Hypergraph	$\|V\|$	$\|E\|$	$\|pins\|$
fract	149	147	462
primary1	833	904	2 910
struct	1 952	1 920	5 471
primary2	3 014	3 029	11 219
industry1	3 085	2 593	8 837
biomed	6 514	5 742	21 040
industry2	**12 637**	**13 419**	**48 158**
industry3	**15 433**	**21 940**	**65 920**
avqsmall	**21 918**	**22 124**	**76 231**
avqlarge	**25 178**	**25 384**	**82 751**
golem3	**100 312**	**144 949**	**337 892**

Table 6.2.: The MCNC Circuit Benchmark Suite contains 11 hypergraphs. We use the five largest hypergraphs for evaluation, since the computation of a 32-way or 64-way partition does not make sense for too small hypergraphs. The selected hypergraphs are highlighted.

hypernodes j, where $A_{i,j}$ (hyperedges represent rows) or $A_{j,i}$ (hyperedges represent columns) is a non-zero entry. Note that each adjacency matrix of a graph can therefore be reinterpreted as a hypergraph. Hypergraph partitioning of SPM instances can be used for parallel sparse-matrix vector multiplication [18], parallel sparse matrix reordering [17], and parallel computation of the block-diagonal form of the matrix [10], which can be used to solve linear programming problems.

Hypergraph	$\|V\|$	$\|E\|$	$\|pins\|$
add20	2 395	2 395	17 319
add32	4 960	4 960	23 884
bcsstk33	8 738	8 738	591 904
4elt	15 606	15 606	107 362
vibrobox	**12 328**	**12 328**	**342 828**
bcsstk29	**13 992**	**13 992**	**619 488**
memplus	**17 758**	**17 758**	**126 150**
bcsstk30	**28 924**	**28 924**	**2 043 492**
bcsstk31	**35 588**	**35 588**	**1 181 416**
bcsstk32	**44 609**	**44 609**	**2 014 701**
finan512	**74 752**	**74 752**	**596 992**

Table 6.3.: The 11 hypergraphs originating from sparse matrices in Walshaw's Graph Partitioning Archive. We ignore the smaller hypergraphs in our evaluation. The selected instances are highlighted.

A very popular collection of partitioning problems is Walshaw's Graph Partitioning Archive[1]. It consists of 34 graphs that have been very popular as benchmarks for graph partitioning algorithms. Out of these graphs, there are 11 which originate from sparse matrices. These graphs are shown in Table 6.3. As with VLSI instances, we ignore smaller instances. The selected instances are highlighted in Table 6.3.

Furthermore, there exists a huge collection of sparse matrices, called The University of Florida Sparse Matrix Collection [19]. As of February 2015[2] it contains 2547 problems. Hence, it is impractical to consider all contained matrices for evaluation. Instead, we select a subset of those matrices which were used in one of the DIMACS Implementation Challenges[3]. These challenges take place regularly and address various graph problems including the shortest path problem, the traveling salesman problem, graph partitioning, and graph clustering. As of now, there were 11 DIMACS implementation challenges held, out of which the 10th [11, 12] is the most relevant to this thesis, since it addresses graph partitioning and graph clustering. The selected hypergraphs are shown in Table 6.4 and will be used for evaluation of our proposed algorithms.

[1] http://staffweb.cms.gre.ac.uk/ c.walshaw/partition/
[2] http://www.cise.ufl.edu/research/sparse/matrices/
[3] http://dimacs.rutgers.edu/Challenges/

Hypergraph	$\|V\|$	$\|E\|$	$\|pins\|$
af_shell9	504 855	504 855	17 588 875
audikw_1	943 695	943 695	77 651 847
ldoor	952 203	952 203	46 522 475
ecology2	999 999	999 999	4 995 991
ecology1	1 000 000	1 000 000	4 996 000
thermal2	1 228 045	1 228 045	8 580 313
af_shell10	1 508 065	1 508 065	52 672 325
G3_circuit	1 585 478	1 585 478	7 660 826
kkt_power	2 063 494	2 063 494	14 612 663
nlpkkt120	3 542 400	3 542 400	96 845 792
cage15	5 154 859	5 154 859	99 199 551
nlpkkt160	8 345 600	8 345 600	229 518 112
nlpkkt200	16 240 000	16 240 000	448 225 632
nlpkkt240	27 993 600	27 993 600	774 472 352

Table 6.4.: The 14 hypergraphs selected from the University of Florida Sparse Matrix Collection and used in the 10th DIMACS challenge.

6.3. Integration in a k-way Multilevel Partitioning Framework

We integrate label propagation in a direct k-way multilevel partitioning framework called *KaHyPar* (**Ka**rlsruhe **Hy**pergraph **Par**titioning) [28]. This framework implements direct k-way partitioning and utilizes the classical multilevel partitioning scheme, but also supports variations of it, like n-level partitioning [44]. Mainly, there are three modules:

- The coarsening module
- The initial partitioning module
- The refinement module

The framework is very versatile, because the modules can be exchanged independently. For example, the coarsening module can coarsen the hypergraph in such a way that between each level we either contract only a single pair of hypernodes (n-level) or we contract a complete cluster (agglomerative coarsening). We use the first approach, i.e. we contract two nodes in a level of the multilevel partitioning scheme. Furthermore, KaHyPar supports V-cycling and allows for multiple runs of the initial partitioning algorithm.

We evaluate label propagation on the clique expanded hypergraph (Section 4.1), label propagation on the star expanded hypergraph (Section 4.2), and probabilistic label propagation (Section 4.3) as coarsening strategies. Note that since we perform n-level coarsening, we do not contract entire clusters but pairs of hypernodes which belong to the same cluster. The selection of these pairs is thereby random, i.e. we randomly select a cluster which was not completely contracted and two random hypernodes belonging to this cluster. These hypernodes are contracted and the process is repeated as long as there exists a cluster which was not completely contracted. If the hypergraph is still too large after all clusters

have been contracted, we repeat the process, i.e. we compute a new clustering on the coarser hypergraph with label propagation and successively contract that clustering. As initial partitioners we use and evaluate both hMetis [31] and PaToH [18]. Finally, we use our proposed fast local search algorithm based on label propagation as refinement strategy (Section 4.4.4).

6.4. Parameter Optimization

Utilized in the multilevel partitioning scheme, our proposed algorithms have a large amount of parameters which need to be optimized. These parameters can be divided into three groups: parameters for the coarsening phase, parameters for the initial partitioning, and parameters for the refinement phase. We will optimize each parameter set individually, since the optimization of all parameters at once is not feasible due to the large parameter space. Furthermore, we select a subset of our benchmark data set and use only those hypergraphs for tuning. We select all hypergraphs from our data sets which have less than 50 000 hypernodes. Due to space constraints, we will only provide a brief overview over the results of the parameter optimization in this section. For more detailed results on all tuned parameters see Appendix D.

To ease the parameter tuning process, we select a default set of parameters. In detail: $\varepsilon = 0.03$, $k \in \{2, 4, 8, 16, 32, 64\}$, sample size = 20, maximal number of iterations for label propagation in the coarsening step = 3, the number of hypernodes to stop coarsening = $100k$, no node ordering, size constraint = $L_{max} \cdot \frac{1}{20} = ((1+\varepsilon)\frac{c(V)}{k} + \max_{v \in V} c(v)) \cdot \frac{1}{20}$, hMetis as initial partitioning algorithm, and the maximal number of iterations for label propagation as refinement strategy = 3. We partition each hypergraph 10 times with different random seeds and compute the arithmetic mean for the cut and partition time over these 10 runs. Next, we compute the geometric mean over all instances and k, using the arithmetic mean per instance.

6.4.1. Coarsening Phase

First of all we need to determine which variant of label propagation with which score functions and which node ordering we should use. Table 6.5 compares the partition time and cut of the best score function per label propagation variation. The tables showing the detailed results for each variant and each score function are provided in Section D.1. Note that label propagation on the star expanded hypergraph is inferior to label propagation on the clique expanded hypergraph and probabilistic label propagation in terms of solution quality and running

Best Score per Variation	Cut	Partition Time
clique expanded variant, $score_{2,3}$	2537.61	4.24
probabilistic variant, $score_{2,1}$	2539.79	4.31
star expanded variant, $score_{2,5}$	2639.77	11.46

Table 6.5.: Comparison between the best score function for each variant of label propagation as coarsening strategy.

time. The slower running time is mainly because of the local search algorithm in the refinement phase. Star expansion changes the topological properties of the hypergraph. Our employed local search algorithm finds many hypernodes which improve the quality of the partitioning if they change their partition, since the quality of the initial partitioning suffers from the topological change. This results in a larger running time.

Furthermore, there are six score functions (Table 6.6), which are very close in terms of quality and running time for both probabilistic label propagation and label propagation on the clique expanded hypergraph. These score functions are $score_{2,1}$, $score_{2,3}$, $score_{2,6}$, $score_{3,1}$, $score_{3,3}$, $score_{3,6}$. We keep these score functions for both variants, discard all other score functions, and won't further consider label propagation on the star expanded hypergraph as coarsening algorithm.

It is noteworthy that even though label propagation on the clique expanded hypergraph has a slower theoretical running time, the practical running time of the partitioning is very close to the running time of the variation which used probabilistic label propagation as coarsening strategy. On the one hand, this is because the hypergraphs utilized in the parameter tuning all have fairly small hyperedges and on the other hand because the update step in the probabilistic variant also comes with a computational cost which can be omitted in the other variant.

Next, we need to determine how the different node orderings impact our algorithms. The running time for the two remaining variants benefits from node ordering, regardless of which type. We select $score_{2,3}(\cdot)$ and $order_4(\cdot)$ as parameters for both probabilistic label propagation and label propagation on the clique expanded hypergraph, since this configuration combines near best quality with near fastest running time. The full tables depicting the comparison of different node orderings are shown in Section D.2.

Variant and Score Function	Cut	Partition Time
clique expanded variant, $score_{2,3}$	2537.61	4.24
probabilistic variant, $score_{2,1}$	2539.79	4.31
clique expanded variant, $score_{2,1}$	2541.68	4.77
clique expanded variant, $score_{3,1}$	2541.78	4.65
probabilistic variant, $score_{2,3}$	2542.97	4.03
clique expanded variant, $score_{2,6}$	2545.53	4.39
probabilistic variant, $score_{2,6}$	2545.94	4.28
clique expanded variant, $score_{3,3}$	2546.98	4.21
probabilistic variant, $score_{3,1}$	2547.11	4.29
probabilistic variant, $score_{3,6}$	2555.44	4.33
clique expanded variant, $score_{3,6}$	2556.41	4.27
probabilistic variant, $score_{3,3}$	2557.69	3.99

Table 6.6.: The 12 best scores over all variations of our algorithms for the coarsening phase. Note that all scores are very close in terms of the cut and partition time.

Now we tune the sample size and the maximal number of label propagation iterations. For both coarsening variants the solution quality increases with the maximal number of label propagation iterations. Recall that we use an adaptive stopping criterion: we stop label propagation if less than five percent of all hypernodes change their label. This results in only minor changes for both the solution quality and total running time starting at five iterations. Furthermore, in case of probabilistic label propagation, the solution quality increases with the sample size. The improvement diminishes starting at a sample size of 25 (even though the maximal hyperedge size in the tuning set is 585). For figures depicting these results in detail see Section D.3.

Finally, the last parameters left to optimize in the coarsening phase are the size constraint and the number of hypernodes when the coarsening process should stop. It should be noted that the results of this parameter tuning are very similar for probabilistic label propagation and label propagation on the clique expanded hypergraph. The coarsening threshold parameter t has thereby more impact on the quality of the partitioning, whereas the size constraint parameter U has a larger impact on the total running time. Note that the partition time increases with larger values of t. This is counterintuitive because we stop the coarsening process earlier for larger values of t and therefore spend less time for the coarsening. However, this can be explained because for small hypergraphs (as in our case) the running time of the initial partitioning algorithm dominates the running time of the coarsening phase and the refinement phase. Hence, if we stop the coarsening process early, the initial partitioning algorithm needs to partition a larger hypergraph, which results in the aforementioned observation. Further details are shown in Section D.4.

6.4.2. Initial Partitioning

We use hMetis [31] or PaToH [18] as initial partitioning algorithm. Both hypergraph partitioners have many configuration parameters. For the sake of simplicity we will use the default parameters for hMetis and in case of PaToH, we utilize the default parameters of the quality preset. Note that in case of hMetis we need to modify our balance constraint, since hMetis computes the partitioning via recursive bisection in the default case. In this case, the imbalance parameter of hMetis specifies the maximal difference between each successive bisection, e.g. a value of five leads to a 45-55 split at each bisection. In detail: if we provide hMetis with a value of b, the maximal allowed partition size is

$$c(V) \cdot (0.5 + \frac{b}{100})^{\log_2(k)} \tag{6.4.1}$$

for a k-way partitioning of a hypergraph $\mathcal{H} = (V, E, c, \omega)$. Note that in our notation, the maximal allowed partition size is

$$L_{max} = (1 + \varepsilon) \cdot \frac{c(V)}{k} + \max_{v \in V} c(v). \tag{6.4.2}$$

Given the maximal imbalance ε, we determine the imbalance parameter of hMetis as follows:

$$(1+\varepsilon)\cdot\frac{c(V)}{k}+\max_{v\in V}c(v)\stackrel{!}{=}c(V)\cdot(0.5+\frac{b}{100})^{\log_2(k)} \tag{6.4.3}$$

$$(\frac{(1+\varepsilon)}{k}+\frac{\max_{v\in V}c(v)}{c(V)})^{\frac{1}{\log_2(k)}}=0.5+\frac{b}{100} \tag{6.4.4}$$

$$\Rightarrow\quad b=100\cdot((\frac{(1+\varepsilon)}{k}+\frac{\max_{v\in V}c(v)}{c(V)})^{\frac{1}{\log_2(k)}}-0.5) \tag{6.4.5}$$

Besides the choice of the initial partitioning algorithm, we evaluate how the number of runs of the initial partitioning algorithm impacts the quality and running time of our algorithms. There are mainly three things that should be noted: first, the main difference between hMetis and PaToH is the running time. The computation of a partitioning took almost twice as much (for both coarsening variants) if we used hMetis as initial partitioner. This effect increased with multiple runs of hMetis. Besides that, utilizing hMetis increased the quality of the partitioning slightly ($\sim 2\%$ in the geometric mean over all k and all tuning instances). Furthermore, multiple runs of the initial partitioning algorithm improved the solution quality marginally for both hMetis and PaToH. The figures showing these results in detail are found in Section D.5.

6.4.3. Refinement Phase

In the refinement phase, we have only one tuning parameter: the number of maximal iterations for label propagation based local search algorithm. Again, the parameter tuning results for both variants (probabilistic and clique expanded) are very similar in terms of partition quality and running time. For both initial partitioning algorithms (hMetis and PaToH), the solution quality does not improve significantly after five iterations. The maximal number of label propagation iterations as local search strategy has nearly no impact on the the running time. This can be explained by the choice of hypergraphs for parameter tuning. Since all these hypergraphs are fairly small, the most expensive part of the computation is the initial partitioning algorithm. Furthermore, our refinement algorithm has a very local view on the hypergraph, since only two hypernodes are uncontracted at each level. This results in only a few iterations of refinement before a local minimum is found. Since our refinement algorithm is greedy, we stop the refinement process before the maximal number of iterations is reached. Therefore, additional label propagation iterations in the refinement phase have nearly no impact on the total partition time (for smaller hypergraphs). Note that this effect is not present for large hypergraphs. Further results are shown in Section D.6.

6.4.4. V-cycles

The last parameter left is the number of V-cycles. Recall that the V-cycle technique repeats the multilevel partitioning scheme multiple times utilizing the computed partition in the latter iterations. Our experiments imply that the quality of the partitioning increases only marginally after five V-cycles for both coarsening

variants (label propagation on the clique expanded hypergraph and probabilistic label propagation) and both initial partitioning algorithms (hMetis and PaToH). The figures depicting these results in detail are found in Section D.7.

Concluding, we propose three configurations of KaHyPar which utilize label propagation: LPFast, LPEco, and LPBest (Table 6.7). FastLP is the fastest variant and sacrifices solution quality for running time. The other extreme is BestLP, which sacrifices running time for the best possible solution quality. EcoLP tries to combine both worlds, resulting in a good trade off between running time and solution quality.

Parameters	LPFast	LPEco	LPBest
Coarsening Variant	Probabilistic	Probabilistic	Clique Expanded
Score Function	$score_{2,3}$	$score_{2,3}$	$score_{2,3}$
Node Ordering	$order_4$	$order_4$	$order_4$
Sample Size	20	25	-
Number of Iterations in Coarsening	2	3	7
Size Constraint	$L_{max} \cdot \frac{1}{10}$	$L_{max} \cdot \frac{1}{10}$	$L_{max} \cdot \frac{1}{10}$
Coarsening Threshold	$50k$	$170k$	$210k$
Initial Partitioning Algorithm	PaToH	hMetis	hMetis
Number of Initial Partitionings	1	1	3
Number of Iterations in Refinement	3	5	10
Number of V-cycles	1	5	10

Table 6.7.: Proposed parameter sets for KaHyPar.

6.5. Experimental Results

We evaluate our proposed configurations on the data sets described in Section 6.2 and compare ourselves against hMetis[4] and PaToH[5]. In hMetis we use the preset for recursive bisection (from now on referred to as hMetis-RB) and direct k-way partitioning (from now on referred to as hMetis-k). PaToH ignores the random seed parameter if any preset is used. Therefore, we compare our algorithms against both the quality preset of PaToH (from now on referred to as PaToH-Q) and the default configuration of PaToH (from now on referred to as PaToH-D).

We exclude *cage15*, *nlpkkt160*, *nlpkkt200*, and *nlpkkt240* from the following comparisons. This is because PaToH-Q and PaToH were the only algorithms which could partition these hypergraphs in reasonable time. It took both hMetis variants more than 18 hours to compute a single partition of cage15 for $k = 2$. Furthermore, nlpkkt200 and nlpkkt240 could not be partitioned with hMetis because the required amount of memory exceeded the available amount of memory on our machine. LPFast could not partition nlpkkt160, nlpkkt200, nlpkkt240 since PaToH (used as initial partitioning algorithm) crashed during computation. Finally, LPEco and LPBest also could not partition nlpkkt160, nlpkkt200, and nlpkkt240 in reasonable time.

Unless mentioned otherwise, for the following comparison we partition the remaining hypergraphs in $k \in \{2, 4, 8, 16, 32, 64, 128\}$ parts with 10 different

[4]Version 2.0pre1: http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download
[5]Version 3.2: http://bmi.osu.edu/umit/software.html

random seeds each and allow a maximal imbalance of $\varepsilon = 0.03$. We compare geometric means of the best cut, the average cut, and the average execution time for each k. The full table listing detailed per-instance results is found in Appendix E. It should be noted that in our experiments hMetis-k was the only algorithm that often produces imbalanced partitions (especially for $k \geq 64$): Out of 2800 cases 569 partitions ($\sim 20\%$) are imbalanced (up to 11.8% imbalance). Furthermore, in 26 out of 280 possible partitioning problems (40 hypergraphs and 7 different values for k) hMetis-k could not produce an ε-balanced partition during 10 runs. The complete table showing these results in detail is found in Appendix E. Since we don't exclude imbalanced partitions from our comparison, hMetis-k has therefore a slight advantage over the other algorithms.

Evaluation of KaHyPar Configurations. Table 6.8 shows the comparison between our proposed algorithms. LPEco is nearly 2.5 times faster than LPBest while only having $\sim 1\%$ worse solution quality. LPFast on the other hand is nearly 16 times faster than LPBest and produces partitions which are around 10% worse. Note that the relative decrease in solution quality of the average cut of LPFast in respect to LPBest decreases with larger k.

	LPBest			LPEco			LPFast		
k	best cut	avg cut	avg t[s]	best cut[%]	avg cut[%]	avg t[s]	best cut[%]	avg cut[%]	avg t[s]
2	1217.98	1272.09	22.35	+0.48	+1.15	12.47	+6.18	+12.32	2.73
4	2576.76	2706.19	31.27	+1.12	+1.13	14.60	+7.66	+12.32	2.85
8	4428.51	4583.05	42.08	+0.81	+1.24	16.73	+8.63	+11.33	3.02
16	6766.00	6928.05	53.95	+0.85	+1.01	20.26	+8.15	+9.58	3.38
32	9509.01	9659.07	75.57	+1.90	+1.92	27.50	+8.46	+9.78	3.77
64	13184.85	13313.25	98.04	+0.72	+0.92	36.06	+7.03	+7.85	4.30
128	17307.54	17440.95	134.83	+0.97	+1.00	47.25	+7.20	+7.52	5.10
avg	5735.03	5887.33	55.32	+0.98	+1.20	22.47	+7.61	+10.09	3.51

Table 6.8.: Detailed comparison of our proposed algorithms over all benchmark instances. Note that cut values are shown as percentual increases in respect to the values obtained by LPBest.

Comparison to other Hypergraph Partitioners. Recall that during parameter tuning we only considered hypergraphs that have less than 50 000 hypernodes. For the following comparisons we focus on the instances that were not part of our parameter tuning set. This is because we want to avoid the effect of overtuning our algorithms to a specific set of hypergraphs.

In Table 6.9 the comparison of our algorithms with the state-of-the-art hypergraph partitioners (hMetis and PaToH) is shown. Note that LPFast is dominated by PaToH-D and PaToH-Q, as they both have a smaller running time and produce better cuts. LPBest on the other hand is dominated by hMetis-RB and hMetis-k. Finally, LPEco computes better partitions as PaToH-Q and PaToH-D (with a slower running time) and has a faster running time than hMetis-k and hMetis-RB (with worse solution quality). In the following we therefore focus on LPEco.

More in-detail comparison of LPEco against hMetis-k, hMetis-RB, PaToH-Q, and PaToH-D is shown in Table 6.10. Note that relative partition quality of

Algorithm	avg cut	best cut	avg t[s]
LPBest	11368.65	11097.79	184.42
LPEco	11540.71	11248.80	75.80
LPFast	12624.92	12041.04	12.33
hMetis-RB	10665.85	10519.09	165.43
hMetis-k	10892.03	10726.42	98.27
PaToH-Q	11551.88	11551.88	9.90
PaToH-D	12147.36	11627.19	2.39

Table 6.9.: Comparison of our algorithms with PaToH and hMetis on larger hypergraphs.

LPEco to the competition improves for larger k: for $k = 2$, LPEco only computes better partitions than PaToH-D, whereas for $k = 128$ LPEco is outperformed only by hMetis-RB.

	LPEco		hMetis-k		hMetis-RB		PaToH-Q		PaToH-D	
k	avg cut	avg t[s]	avg cut[%]	avg t [s]	avg cut[%]	avg t[s]	avg cut[%]	avg t[s]	avg cut[%]	avg t[s]
2	2235.40	51.36	-4.14	53.86	-6.83	58.57	-2.27	3.10	+2.95	0.83
4	4791.03	56.18	-4.36	64.11	-6.27	108.02	+1.27	5.91	+6.22	1.49
8	8200.48	59.50	-4.60	74.62	-5.48	151.22	-0.16	8.53	+6.06	2.09
16	12727.65	67.02	-2.95	90.74	-4.29	191.51	+0.13	11.16	+5.90	2.66
32	18938.05	80.14	-2.68	116.75	-4.59	229.27	-0.44	13.94	+4.07	3.22
64	27420.42	97.40	-0.62	156.61	-3.21	266.28	+0.50	16.17	+5.64	3.77
128	37659.85	128.61	+1.35	207.10	-1.55	303.20	+1.53	19.05	+6.00	4.31
avg	11540.71	75.80	-5.62	98.27	-7.58	165.43	+0.10	9.90	+5.26	2.39

Table 6.10.: Detailed comparison of LPEco, hMetis, and PaToH on larger instances. The average cuts for hMetis and PaToH are shown as percentual increases in respect to the values obtained by LPEco.

Per Benchmark Set Comparison. Recall that our benchmark set consists of both VLSI and SPM instances. We now examine the hypergraph partitioners in respect to these hypergraph classes. Table 6.11 shows detailed results for the hypergraph partitioners on the complete VLSI benchmark set. LPEco outperforms both hMetis presets and both PaToH presets for $k \geq 64$. Our algorithm achieves the best results for $k = 128$, where it produces 7% better cuts than hMetis-k and 2% better cuts than hMetis-RB, while being 2.5 times and 1.5 times faster (respectively). LPEco also produces 4% better cuts than PaToH-Q and 7% better cuts than PaToH-D.

In Table 6.12 detailed results of the hypergraph partitioners on the complete SPM benchmark set are shown. Note that LPEco performs much worse on this benchmark set. Again, our algorithm improves for larger k. But in difference to the VLSI benchmark set (Table 6.11), LPEco produces worse partitions than hMetis-RB and PaToH-Q regardless of k. We have two explanations for the large differences in solution quality of the two benchmarks. First, in contrast to VLSI instances, SPM instances may not any deeper cluster structure, as they result from various scientific computational problems.

Therefore, if not size-constrained, label propagation (like in the graph case [47]) would find the strongest connected components of the hypergraph. Since we em-

	LPEco		hMetis-k		hMetis-RB		PaToH-Q		PaToH-D	
k	avg cut	avg t[s]	avg cut[%]	avg t[s]	avg cut[%]	avg t[s]	avg cut[%]	avg t[s]	avg cut[%]	avg t[s]
2	875.82	4.39	-6.57	5.34	-7.83	6.25	-2.12	0.49	+5.64	0.10
4	1787.85	5.21	-6.61	7.25	-5.20	11.63	+1.16	0.88	+10.30	0.16
8	2837.64	6.14	-4.52	10.22	-3.62	16.45	+2.87	1.24	+10.73	0.22
16	4081.31	7.75	-2.43	15.36	-2.11	21.35	+2.34	1.55	+9.18	0.28
32	5402.97	11.23	+1.33	24.34	-0.23	26.27	+3.79	1.92	+9.16	0.34
64	6982.90	16.23	+4.16	39.31	+0.98	31.64	+4.21	2.20	+8.49	0.40
128	8693.29	22.91	+7.31	57.01	+2.32	37.83	+4.13	2.58	+7.77	0.46
avg	3460.94	8.93	-1.17	16.49	-2.30	18.71	+2.32	1.37	+8.74	0.25

Table 6.11.: Detailed comparison of LPEco, hMetis, and PaToH on all VLSI benchmark instances. The average cuts for hMetis and PaToH are shown as percentual increases in respect to the values obtained by LPEco.

	LPEco		hMetis-k		hMetis-RB		PaToH-Q		PaToH-D	
k	avg cut	avg t[s]	avg cut[%]	avg t[s]	avg cut[%]	avg t[s]	avg cut[%]	avg t[s]	avg cut[%]	avg t[s]
2	2165.05	51.22	-4.04	58.66	-7.31	66.61	-4.14	4.23	-0.65	1.33
4	4868.89	58.87	-4.77	70.07	-6.76	125.30	-1.98	7.61	+0.71	2.35
8	9025.19	64.98	-3.26	83.43	-6.64	178.63	-3.71	11.17	+1.28	3.30
16	14515.12	74.30	-0.50	102.56	-4.83	226.64	-1.95	14.43	+3.29	4.19
32	22168.83	92.42	-0.61	134.80	-5.60	272.00	-2.69	17.92	+1.24	5.02
64	32571.72	106.22	+1.33	185.46	-4.47	316.34	-1.55	20.34	+3.40	5.82
128	45795.63	125.80	+1.61	240.44	-2.50	357.27	-0.64	24.10	+3.41	6.58
avg	12422.99	78.26	-1.49	111.29	-5.46	194.12	-2.39	12.42	+1.80	3.63

Table 6.12.: Detailed comparison of LPEco, hMetis, and PaToH on all SPM benchmark instances. The average cuts for hMetis and PaToH are shown as percentual increases in respect to the values obtained by LPEco.

ploy a size-constrained version of label propagation, these strongly connected components are represented by many labels. This results in the fact that hypernodes with the same label may not share any structural properties besides the belonging to the strongly connected component. Therefore, in case of SPM instances, our coarsening schemes could produce subpar coarsenings.

Second, our employed refinement algorithm has only a very local view (since we uncontract two hypernodes at each level) and is *greedy*. That is, we only change the block of a hypernode if an improvement of the cut is found (or the total connectivity of the hypergraph is decreased). SPM instances nearly always consist of uniform, large hyperedges. This results in many blocks in the connectivity set of hyperedges. Therefore, our refinement algorithm performs many zero gain moves which do not benefit the solution quality, but possibly prohibit moves on a coarser level of the hypergraph. One solution to this problem is to implement level-gains [36] which would prioritize zero gain moves better than our currently employed connectivity decrease tie-breaking.

7 | Conclusion

In this thesis we investigate the application of label propagation to hypergraph partitioning, especially to the popular multilevel partitioning heuristic. We consider label propagation as coarsening strategy in the coarsening phase and as a local search algorithm in the refinement phase. During the coarsening, we thereby compute a size-constrained clustering of the hypergraph. The hypernodes inside a cluster are then pairwise contracted. We propose three different adaptations of label propagation to hypergraphs, two of which can be seen to operate on graphs modeling the hypergraph. One of them has a non-linear running time and good solution quality, whereas the other one has a linear running time, but suffers from quality loss. The third adaptation is a randomized version of label propagation which has good solution quality and linear running time. Furthermore, we propose a greedy local search algorithm based on label propagation.

We integrate our algorithms in a multilevel direct k-way hypergraph partitioning framework KaHyPar and propose three configurations of that framework: LPFast, LPEco, and LPBest. We compare our algorithms against the state-of-the-art hypergraph partitioners hMetis and PaToH on hypergraphs originating from popular benchmarks for both VLSI and SPM. In hMetis we consider both the recursive bisection variant as well as the direct k-way partitioning variant. In PaToH, we consider both the default preset and the quality preset.

Our algorithms achieve the best results for $k = 128$ on a VLSI benchmark set, where LPEco produces 7% better cuts than the direct k-way partitioning variant of hMetis and 2% better cuts than the recursive bisection variant of hMetis, while being 2.5 times and 1.5 times faster (respectively). Furthermore, LPEco outperforms both variants of PaToH, resulting in 4% better cuts than the quality preset of PaToH and and 7% better cuts than the default preset of PaToH. On the SPM benchmark set, LPEco produces better cuts than the direct k-way variant of hMetis and the default preset of PaToH for $k \geq 64$, but is outperformed by the recursive bisection variant of hMetis and the quality preset of PaToH.

7.1. Future Work

Since the introduction of label propagation [47], the algorithm has risen to prominence, especially in the field of machine learning [30, 43, 53]. It comes as no surprise that many problems [1, 2, 61] are easily and better modeled [2] with a hypergraph than a graph. These problems can (partially) be solved with a clustering of this hypergraph. With minor modifications, our proposed label propagation adaptations can also be applied to hypergraph clustering. However, since the objective of a partitioning generally differs from the objective of a clustering, further score functions for incident labels need to be engineered and evaluated.

Furthermore, our coarsening algorithms can be seen to operate on a graph modeling the hypergraph. We consider two graph classes: in the first, each

hyperedge is replaced by a clique. In the second, each hyperedge is replaced by a node with all incident nodes being connected to the new node. These are but only two models for graph based hypergraph modeling. It is interesting if there exist other types of hypergraph expansion that would benefit the quality and running time of our proposed algorithms.

Moreover, we use label propagation in a direct k-way partitioning framework. The experimental results of hMetis imply that the computation of a k-way partitioning via recursive bisection often results in better quality. Therefore, it is interesting to evaluate whether the same observation applies to our algorithms.

Our algorithms perform considerably worse on SPM instances than on VLSI instances. The utilized SPM benchmark set contains thereby different problem kinds, e.g. optimization problems, 2D/3D problems, and structural problems. It should be noted, that our benchmark set does not cover all problem kinds present in the Florida Sparse Matrix Collection. Future work could further investigate the performance of label propagation in each problem kind and whether problem kinds exist, where label propagation performs well.

Utilizing two kinds of labels like Tang et al. [53], label propagation could also be used for the computation of an initial partition: All but $k \cdot \alpha$ hypernodes, α being a tuning parameter, are initialized with an "empty" label. The remaining hypernodes get one of k labels assigned at random. If we now perform label propagation as long as empty labels remain, the resulting clusters become the blocks of the partition. This process can be repeated multiple times if the initial attempt resulted in a bad partition.

Another interesting adaptation to the initial label propagation algorithm is proposed by Kang et al. [30] and could easily be applied to hypergraphs. Instead of propagating one label, each hypernode could store, update and propagate multiple labels at once.

Our greedy size-constrained label propagation based refinement algorithm only prioritizes zero gain moves based on the global connectivity decrease of the move. Another approach based on level-gains is proposed by Krishnamurthy [36]. Our greedy local search algorithm can be further extended to use level-gains. The prioritization of zero gain moves should lead to an improvement of the overall solution quality, especially for SPM instances.

Finally, our proposed algorithms have the potential to be parallelized, since each hypernode selects its new label based on a score distribution among the present labels in its neighborhood.

Bibliography

[1] S. Agarwal, K. Branson, and S. Belongie. Higher order learning with graphs. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 17–24, New York, NY, USA, 2006. ACM.

[2] S. Agarwal, J. Lim, L. Zelnik-Manor, P. Perona, D. Kriegman, and S. Belongie. Beyond pairwise clustering. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 838–845 vol. 2, June 2005.

[3] Y. Akhremtsev, P. Sanders, and C. Schulz. (semi-)external algorithms for graph partitioning and clustering. *CoRR*, abs/1404.4887, 2014.

[4] C. Alpert, A. Kahng, G.J. Nam, S. Reda, and P. Villarrubia. A semi-persistent clustering technique for vlsi circuit placement. In *Proceedings of the 2005 International Symposium on Physical Design*, ISPD '05, pages 200–207, New York, NY, USA, 2005. ACM.

[5] C.J. Alpert. The ispd98 circuit benchmark suite. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, pages 80–85, New York, NY, USA, 1998. ACM.

[6] C.J. Alpert, J.H. Huang, and A.B. Kahng. Multilevel circuit partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(8):655–667, Aug 1998.

[7] C.J. Alpert and A.B. Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI Journal*, 19(1–2):1 – 81, 1995.

[8] C.J. Alpert, G.J. Nam, and P.G. Villarrubia. Effective free space management for cut-based placement via analytical constraint generation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(10):1343–1353, Oct 2003.

[9] R. Askey. *Orthogonal polynomials and special functions*, volume 21. SIAM, 1975.

[10] C. Aykanat, A. Pinar, and Ü.V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal on Scientific Computing*, 25(6):1860–1879, 2004.

[11] D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.

[12] D.A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. Graph partitioning and graph clustering, 10th dimacs implementation challenge workshop. *Contemporary Mathematics*, 588, 2013.

[13] S.T. Barnard and H.D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, 1994.

[14] C. Berge and E. Minieka. *Graphs and hypergraphs*, volume 7. North-Holland publishing company Amsterdam, 1973.

[15] C.E. Bichot and P. Siarry. *Graph partitioning*. John Wiley & Sons, 2013.

[16] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[17] Ü.V. Çatalyürek. Hypergraph models for sparse matrix partitioning and reordering. 1999.

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

[19] T.A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[20] E. Estrada and J.A. Rodríguez-Velázquez. Subgraph centrality and clustering in complex hyper-networks. *Physica A Statistical Mechanics and its Applications*, 364:581–594, May 2006.

[21] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*, pages 175–181, June 1982.

[22] M.R. Garey and D.S. Johnson. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

[23] J.R. Gilbert, S. Reinhardt, and V.B. Shah. High-performance graph algorithms from parallel sparse matrices. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 260–269. Springer Berlin Heidelberg, 2007.

[24] J. Gong and S.K. Lim. Multiway partitioning with pairwise movement. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*, pages 512–516, Nov 1998.

[25] S. Gregory. Finding overlapping communities in networks by label propagation. *New Journal of Physics*, 12(10):103018, 2010.

[26] S.W. Hadley. Approximation techniques for hypergraph partitioning problems. *Discrete Applied Mathematics*, 59(2):115 – 127, 1995.

[27] B. Hendrickson and R.W. Leland. A multi-level algorithm for partitioning graphs. *SC*, 95:28, 1995.

[28] V. Henne, H. Meyerhenke, P. Sanders, S. Schlag, and C. Schulz. *n*-Level hypergraph partitioning. *to be published*.

[29] E. Ihler, D. Wagner, and F. Wagner. Modeling hypergraphs by graphs with the same mincut properties. *Information Processing Letters*, 45(4):171 – 175, 1993.

[30] F. Kang, R. Jin, and R. Sukthankar. Correlated label propagation with application to multi-label learning. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 1719–1726, 2006.

[31] G. Karypis. Multilevel hypergraph partitioning. In *Multilevel Optimization in VLSICAD*, volume 14 of *Combinatorial Optimization*, pages 125–154. Springer US, 2003.

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

[33] G. Karypis and V. Kumar. Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[34] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[35] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[36] B. Krishnamurthy. An improved min-cut algonthm for partitioning vlsi networks. *IEEE Transactions on Computers*, 33(5):438–446, 1984.

[37] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., 1990.

[38] P.H. Madden. Reporting of standard cell placement results. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(2):240–247, Feb 2002.

[39] A.M. Mathai and R.K. Saxena. On a generalized hypergeometric distribution. *Metrika*, 11(1):127–132, 1967.

[40] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel graph partitioning for complex networks. *CoRR*, abs/1404.4797, 2014.

[41] H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning complex networks via size-constrained clustering. In *Experimental Algorithms*, volume 8504 of *Lecture Notes in Computer Science*, pages 351–363. Springer International Publishing, 2014.

[42] T. Mukherjee. Cluster shaping: A novel optimization technique for large scale vlsi placement. Master's thesis, University of Cincinnati, 2014.

[43] Z.Y. Niu, D.H. Ji, and C.L. Tan. Word sense disambiguation using label propagation based semi-supervised learning. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 395–402, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[44] V. Osipov and P. Sanders. n-level graph partitioning. In *Algorithms – ESA 2010*, volume 6346 of *Lecture Notes in Computer Science*, pages 278–289. Springer Berlin Heidelberg, 2010.

[45] D.A. Papa and I.L. Markov. Hypergraph partitioning and clustering. *Approximation algorithms and metaheuristics*, pages 61–1, 2007.

[46] L. Pu and B. Faltings. Hypergraph Clustering for Better Network Traffic Inspection. In *The 3rd Workshop on Intelligent Security at IJCAI*, 2011.

[47] U.N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007.

[48] L.A. Sanchis. Multiple-way network partitioning. *Computers, IEEE Transactions on*, 38(1):62–81, 1989.

[49] P. Sanders and C. Schulz. Engineering multilevel graph partitioning algorithms. In *Algorithms – ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer Berlin Heidelberg, 2011.

[50] P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[51] D.G. Schweikert and B.W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th Design Automation Workshop*, DAC '72, pages 57–62, New York, NY, USA, 1972. ACM.

[52] W.J. Sun and C. Sechen. Efficient and effective placement for very large circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(3):349–359, Mar 1995.

[53] J. Tang, R. Hong, S. Yan, T.S. Chua, G.J. Qi, and R. Jain. Image annotation by knn-sparse graph-based label propagation over noisily tagged web images. *ACM Trans. Intell. Syst. Technol.*, 2(2):14:1–14:15, February 2011.

[54] N. Tirumalai. Lcplace: A novel vlsi placement methodology based on large cluster formation. Master's thesis, University of Cincinnati, 2014.

[55] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 507–516, New York, NY, USA, 2013. ACM.

[56] N. Viswanathan, M. Pan, and C. Chu. Fastplace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 135–140, Washington, DC, USA, 2007. IEEE Computer Society.

[57] C. Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1-4):325–372, 2004.

[58] C. Walshaw and M. Cross. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

[59] F. Wang and C. Zhang. Label propagation through linear neighborhoods. *Knowledge and Data Engineering, IEEE Transactions on*, 20(1):55–67, Jan 2008.

[60] S. Wichlund. On multilevel circuit partitioning. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '98, pages 505–511, New York, NY, USA, 1998. ACM.

[61] D. Zhou, J. Huang, and B. Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In *NIPS'06*, pages 1601–1608, 2006.

[62] J.Y. Zien, M.D.F. Schlag, and P.K. Chan. Multilevel spectral hypergraph partitioning with arbitrary vertex sizes. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(9):1389–1399, Sep 1999.

A | Proof for the Expected Value of Hypergeometric Distributions

Let Y_j be a random variable which follows a hypergeometric distribution parameterized with e_j, d_j, S. Recall that in our case e_j is the size of a hyperedge which has d_j pins with label j. We draw a sample of size S. Y_j counts then the number of pins in the sample having label j.

The probability mass function for Y_j is (Lemma 4.3.3):

$$f_{Y_j}(y) := \mathbb{P}(Y_j = y) = \frac{\binom{d_j}{y}\binom{e_j - d_j}{S-y}}{\binom{e_j}{S}}, \qquad y \in \mathbb{N}_0,\ 0 \leq y \leq S. \tag{A.1}$$

A direct implication of Lemma 4.3.3 is the following identity:

$$1 \overset{!}{=} \sum_{y=0}^{S} f_{Y_j}(y) = \sum_{y=0}^{S} \frac{\binom{d_j}{y}\binom{e_j - d_j}{S-y}}{\binom{e_j}{S}} \tag{A.2}$$

$$\Rightarrow \quad 1 = \sum_{i=0}^{c} \frac{\binom{a}{i}\binom{b-a}{c-i}}{\binom{b}{c}}, \qquad a, b, c \in \mathbb{N}_0,\ b \geq a \geq c \geq 0 \tag{A.3}$$

The equality in Equation A.2 holds per definition, because $f_{Y_j}(y)$ is a probability mass function and therefore must sum up to 1 over its domain of definition. Since d_j, e_j, and S can be chosen arbitrarily, we can infer Equation A.3. This equality is also known as vandermonde's identity [9] in literature.

Lemma (Expected Value of Y_j)**.**

$$\mathbb{E}[Y_j] = S \cdot \frac{d_j}{e_j}.$$

Proof. Let Y_j, e_j, d_j, and S be defined as above. The mean of a random variable X is per definition: $\sum_{i=1}^{n} x_i \mathbb{P}(x_i)$ with $x_1, \ldots, x_n$ being the possible values of X.

Therefore:

$$\begin{aligned}
\mathbb{E}[Y_j] &= \sum_{y=0}^{S} y \cdot \mathbb{P}(Y_j = y) = \sum_{y=0}^{S} y \cdot \frac{\binom{d_j}{y}\binom{e_j - d_j}{S-y}}{\binom{e_j}{S}} \\
&\overset{(1)}{=} \sum_{y=1}^{S} y \cdot \frac{\binom{d_j}{y}\binom{e_j - d_j}{S-y}}{\binom{e_j}{S}} \\
&\overset{(2)}{=} \sum_{y=1}^{S} y \cdot \frac{\frac{d_j}{y}\binom{d_j - 1}{y-1}\binom{e_j - d_j}{S-y}}{\frac{e_j}{S}\binom{e_j - 1}{S-1}} \\
&= S \cdot \frac{d_j}{e_j} \left(\sum_{y=1}^{S} \frac{\binom{d_j - 1}{y-1}\binom{(e_j - 1) - (d_j - 1)}{(S-1) - (y-1)}}{\binom{e_j - 1}{S-1}} \right) \\
&\overset{(3)}{=} S \cdot \frac{d_j}{e_j} \left(\underbrace{\sum_{i=0}^{c} \frac{\binom{a}{i}\binom{b-a}{c-i}}{\binom{b}{c}}}_{=1 \text{ (Equation A.3)}} \right) \\
&\overset{(4)}{=} S \cdot \frac{d_j}{e_j}.
\end{aligned}$$

The equalities hold, because:

(1) the first term ($y = 0$) is 0

(2) $\binom{n}{k} = \frac{n!}{(n-k)!\,k!} = \frac{n \cdot (n-1)!}{k \cdot ((n-1)-(k-1))!\,(k-1)!} = \frac{n}{k} \cdot \binom{n-1}{k-1}, k > 0$

(3) substituting $i := y - 1, a := (d_j - 1), b := (e_j - 1), c := (S - 1)$

(4) vandermonde's equality.

□

B | Hypergraph Format

Hypergraphs are usually stored in the Metis [33] format. In this format, a hypergraph $\mathcal{H} = (V, E, c, \omega)$ is stored in a plain text file containing $|E| + 1$ lines if the hypernodes are unweighted and $|E| + |V| + 1$ lines in case of weighted hypernodes. Any line stating with '%' is considered to be a comment and therefore ignored. The first non-comment contains either two or three integers. The first integer is the number of hyperedges |E|. The second integer is the number of hypernodes |V|. The optional third integer denotes the type of the hypergraph:

- 0: unweighted hypergraph
- 1: hypergraph with edge weights
- 10: hypergraph with node weights
- 11: hypergraph with node weights and edge weights

In case of an unweighted hypergraph the third integer can be omitted. Following this header the next $|E|$ lines represent the hyperedges. For each hyperedge, the corresponding line contains all pins of the hyperedge. In case of a hypergraph with edge weights, the first integer in each line denotes the weight of the hyperedge. In particular, the i-th line contains the pins of the $i-1$-th hyperedge. Note that the hypernodes start with the integer one.

If the hypergraph has node weights, after these $|E|$ lines follow $|V|$ lines with one integer each. This integer denotes the weight of the hypernode. In particular, the $(|E| + j)$-th line represents the weight of hypernode $j - 1$. Figure B.1 shows a hypergraph and its representation in this format.

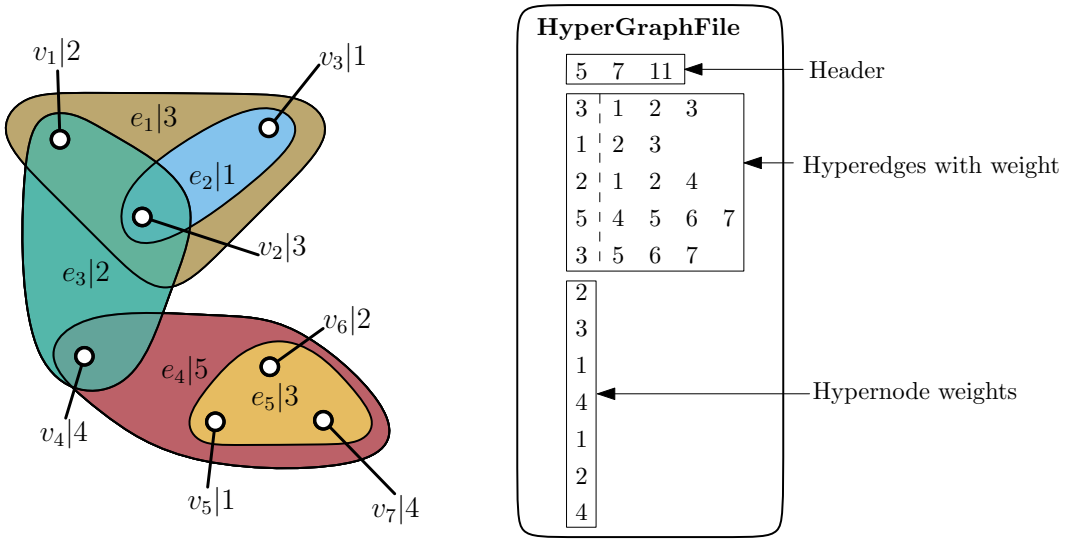


Figure B.1.: *A hypergraph and its representation in the Metis format*

C Detailed Hypergraph Properties of our Benchmark Set

Hypergraph	$\|V\|$	$\|E\|$	$\|pins\|$	HE-Size					HN-Degree					$\frac{\|E\|}{\|V\|}$
				μ	σ	min	90	max	μ	σ	min	90	max	
ibm01	12 752	14 111	50 566	3,58	3,34	2	7	42	3,97	2,33	1	7	39	1,11
ibm02	19 601	19 584	81 199	4,15	5,45	2	9	134	4,14	2,29	1	6	69	1,00
ibm03	23 136	27 401	93 573	3,41	3,11	2	6	55	4,04	3,45	1	6	100	1,18
ibm04	27 507	31 970	105 859	3,31	2,92	2	6	46	3,85	4,65	1	7	526	1,16
ibm05	29 347	28 446	126 308	4,44	4,29	2	13	17	4,30	2,35	1	9	9	0,97
ibm06	32 498	34 826	128 182	3,68	3,28	2	8	35	3,94	1,84	1	6	91	1,07
ibm07	45 926	48 117	175 639	3,65	3,05	2	7	25	3,82	2,41	1	6	98	1,05
ibm08	51 309	50 513	204 890	4,06	5,01	2	7	75	3,99	6,18	1	6	1 165	0,98
ibm09	53 395	60 902	222 088	3,65	3,13	2	7	39	4,16	3,22	1	7	173	1,14
ibm10	69 429	75 196	297 567	3,96	3,56	2	9	41	4,29	3,22	1	7	137	1,08
ibm11	70 558	81 454	280 786	3,45	2,60	2	6	24	3,98	3,17	1	7	174	1,15
ibm12	71 076	77 240	317 760	4,11	3,72	2	10	28	4,47	4,68	1	8	473	1,09
ibm13	84 199	99 666	357 075	3,58	3,01	2	7	24	4,24	3,34	1	7	180	1,18
ibm14	147 605	152 772	546 816	3,58	2,94	2	7	33	3,70	3,18	1	6	270	1,04
ibm15	161 570	186 608	715 823	3,84	3,51	2	8	36	4,43	3,29	1	8	306	1,15
ibm16	183 484	190 048	778 823	4,10	3,61	2	9	40	4,24	2,77	1	6	177	1,04
ibm17	185 495	189 581	860 036	4,54	4,07	2	11	36	4,64	2,49	1	9	81	1,02
ibm18	210 613	201 920	819 697	4,06	3,96	2	8	66	3,89	1,90	1	6	97	0,96
avqlarge	25 178	25 384	82 751	3,26	49,84	2	3	4 042	3,29	1,29	1	6	7	1,01
avqsmall	21 918	22 124	76 231	3,45	53,39	2	4	4 042	3,48	1,40	1	6	7	1,01
biomed	6 514	5 742	21 040	3,66	20,89	2	4	861	3,23	1,06	1	6	6	0,88
fract	149	147	462	3,14	2,26	2	5	17	3,10	1,65	1	5	7	0,99
golem3	100 312	144 949	337 892	2,33	0,99	2	3	38	3,37	1,64	0	5	22	1,44
industry1	3 085	2 593	8 837	3,41	8,37	2	5	320	2,86	1,43	1	5	9	0,84
industry2	12 637	13 419	48 158	3,59	10,97	2	5	585	3,81	1,81	1	7	12	1,06
industry3	15 433	21 940	65 920	3,00	3,23	1	5	325	4,27	1,54	1	6	12	1,42
primary1	833	904	2 910	3,22	2,59	1	5	18	3,49	1,29	1	5	9	1,09
primary2	3 014	3 029	11 219	3,70	3,82	2	7	37	3,72	1,55	1	7	9	1,00
struct	1 952	1 920	5 471	2,85	1,90	2	3	17	2,80	0,67	1	4	4	0,98

Table C.1.: Detailed information for the hypergraphs from the VLSI benchmark set.

Hypergraph	$\|V\|$	$\|E\|$	$\|pins\|$	HE-Size μ	HE-Size σ	HE-Size min	HE-Size 90	HE-Size max	HN-Degree μ	HN-Degree σ	HN-Degree min	HN-Degree 90	HN-Degree max	$\frac{\|E\|}{\|V\|}$
af_shell10	1 508 065	1 508 065	52 672 325	34,93	0,90	15	35	35	34,93	0,90	15	35	35	1,00
af_shell9	504 855	504 855	17 588 875	34,84	1,28	20	35	40	34,84	1,28	20	35	40	1,00
audikw_1	943 695	943 695	77 651 847	82,28	42,45	21	141	345	82,28	42,45	21	141	345	1,00
cage15	5 154 859	5 154 859	99 199 551	19,24	5,74	3	27	47	19,24	5,74	3	27	47	1,00
ecology1	1 000 000	1 000 000	4 996 000	5,00	0,06	3	5	5	5,00	0,06	3	5	5	1,00
ecology2	999 999	999 999	4 995 991	5,00	0,06	3	5	5	5,00	0,06	3	5	5	1,00
G3_circuit	1 585 478	1 585 478	7 660 826	4,83	0,64	2	5	6	4,83	0,64	2	5	6	1,00
kkt_power	2 063 494	2 063 494	14 612 663	7,08	7,40	1	14	96	7,08	7,40	1	14	96	1,00
ldoor	952 203	952 203	46 522 475	48,86	11,95	28	63	77	48,86	11,95	28	63	77	1,00
nlpkkt120	3 542 400	3 542 400	96 845 792	27,34	3,09	5	28	28	27,34	3,09	5	28	28	1,00
nlpkkt160	8 345 600	8 345 600	229 518 112	27,50	2,70	5	28	28	27,50	2,70	5	28	28	1,00
nlpkkt200	16 240 000	16 240 000	448 225 632	27,60	2,42	5	28	28	27,60	2,42	5	28	28	1,00
nlpkkt240	27 993 600	27 993 600	774 472 352	27,67	2,22	5	28	28	27,67	2,22	5	28	28	1,00
thermal2	1 228 045	1 228 045	8 580 313	6,99	0,81	1	8	11	6,99	0,81	1	8	11	1,00
4elt	15 606	15 606	107 362	6,88	0,57	4	7	11	6,88	0,57	4	7	11	1,00
add20	2 395	2 395	17 319	7,23	13,03	2	11	124	7,23	13,03	2	11	124	1,00
add32	4 960	4 960	23 884	4,82	3,68	2	10	32	4,82	3,68	2	10	32	1,00
bcsstk29	13 992	13 992	619 488	44,27	15,64	5	54	71	44,27	15,64	5	54	71	1,00
bcsstk30	28 924	28 924	2 043 492	70,65	31,72	4	107	219	70,65	31,72	4	107	219	1,00
bcsstk31	35 588	35 588	1 181 416	33,20	15,18	2	54	189	33,20	15,18	2	54	189	1,00
bcsstk32	44 609	44 609	2 014 701	45,16	15,48	2	54	216	45,16	15,48	2	54	216	1,00
bcsstk33	8 738	8 738	591 904	67,74	16,11	20	81	141	67,74	16,11	20	81	141	1,00
finan512	74 752	74 752	596 992	7,99	6,28	3	13	55	7,99	6,28	3	13	55	1,00
memplus	17 758	17 758	126 150	7,10	22,04	2	12	574	7,10	22,04	2	12	574	1,00
vibrobox	12 328	12 328	342 828	27,81	16,09	9	47	121	27,81	16,09	9	47	121	1,00

Table C.2.: Detailed information for the hypergraphs from the SPM benchmark set.

D | Parameter Tuning

This section contains further results of our parameter tuning without comment. To ease the parameter tuning process, we select a default set of parameters. In detail: $\varepsilon = 0.03$, $k \in \{2, 4, 8, 16, 32, 64\}$, sample size = 20, maximal number of iterations for label propagation in the coarsening step = 3, the number of hypernodes to stop coarsening = $100k$, no node ordering, size constraint = $L_{max} \cdot \frac{1}{20} = ((1+\varepsilon)\frac{c(V)}{k} + \max_{v \in V} c(v)) \cdot \frac{1}{20}$, hMetis as initial partitioning algorithm, and the maximal number of iterations for label propagation as refinement strategy = 3. We compare thereby the geometric mean of the total partition time and the cut using 10 partitioning trials per hypergraph and k.

D.1. Score Function Comparison

Score Function	Cut	Partition Time
$score_{2,3}$	2537.61	4.24
$score_{2,1}$	2541.68	4.77
$score_{3,1}$	2541.78	4.65
$score_{2,6}$	2545.53	4.39
$score_{3,3}$	2546.98	4.21
$score_{3,6}$	2556.41	4.27
$score_{3,2}$	2580.48	5.48
$score_{2,2}$	2580.49	5.48
$score_{2,5}$	2649.89	3.77
$score_{2}$	2682.72	3.59
$score_{3,5}$	2732.65	3.84
$score_{3}$	2764.08	3.64
$score_{1,2}$	2948.78	5.63
$score_{5}$	2962.48	6.02
$score_{1,1}$	2983.93	4.60
$score_{2,4}$	2984.47	6.48
$score_{1,6}$	3003.50	4.40
$score_{3,4}$	3019.20	6.44
$score_{6}$	3052.83	6.25
$score_{1,3}$	3107.08	3.86
$score_{1,4}$	3244.90	5.79
$score_{4}$	3416.82	6.25
$score_{1,5}$	3424.22	3.81
$score_{1}$	3452.21	3.71

Table D.1.: Comparison of the various score functions for label propagation on the clique expanded hypergraph.

Score Function	Cut	Partition Time
$score_{2,5}$	2639.77	11.46
$score_{2,3}$	2658.96	7.23
$score_{2}$	2672.45	3.70
$score_{2,1}$	2706.08	11.50
$score_{2,6}$	2714.86	6.80
$score_{2,2}$	2808.63	11.64
$score_{1,3}$	2834.39	8.28
$score_{1,1}$	2856.63	16.35
$score_{1,6}$	2887.43	7.10
$score_{2,4}$	2928.62	4.32
$score_{1,5}$	2938.20	15.30
$score_{1,2}$	2944.36	15.29
$score_{1}$	2980.34	4.06
$score_{1,4}$	2986.45	4.32

Table D.2.: Comparison of the various score functions for label propagation on the star expanded hypergraph.

Score Function	Cut	Partition Time
$score_{2,1}$	2539.79	4.31
$score_{2,3}$	2542.97	4.03
$score_{2,6}$	2545.94	4.28
$score_{3,1}$	2547.11	4.29
$score_{3,6}$	2555.44	4.33
$score_{3,3}$	2557.69	3.99
$score_{2,2}$	2579.51	5.28
$score_{3,2}$	2590.12	5.25
$score_{2,5}$	2650.26	3.62
$score_{2}$	2671.71	3.20
$score_{3,5}$	2732.90	3.76
$score_{3}$	2766.98	3.34
$score_{2,4}$	2778.71	5.46
$score_{3,4}$	2825.46	5.53
$score_{1,2}$	3002.69	6.00
$score_{5}$	3042.30	5.33
$score_{1,1}$	3050.77	4.77
$score_{1,6}$	3077.55	4.73
$score_{1,3}$	3119.13	4.10
$score_{6}$	3145.06	5.57
$score_{1,4}$	3183.81	5.65
$score_{1,5}$	3356.52	3.83
$score_{1}$	3373.94	3.41
$score_{4}$	3413.64	5.53

Table D.3.: Comparison of the various score functions for probabilistic label propagation.

Variant and Score Function	Cut	Partition Time
clique expanded variant, $score_{2,3}$	2537.61	4.24
probabilistic variant, $score_{2,1}$	2539.79	4.31
clique expanded variant, $score_{2,1}$	2541.68	4.77
clique expanded variant, $score_{3,1}$	2541.78	4.65
probabilistic variant, $score_{2,3}$	2542.97	4.03
clique expanded variant, $score_{2,6}$	2545.53	4.39
probabilistic variant, $score_{2,6}$	2545.94	4.28
clique expanded variant, $score_{3,3}$	2546.98	4.21
probabilistic variant, $score_{3,1}$	2547.11	4.29
probabilistic variant, $score_{3,6}$	2555.44	4.33
clique expanded variant, $score_{3,6}$	2556.41	4.27
probabilistic variant, $score_{3,3}$	2557.69	3.99

Table D.4.: The 12 best scores over all variations of our algorithms for the coarsening phase. Note that all scores are very close in terms of the cut and partition time.

D.2. Node Ordering Comparison

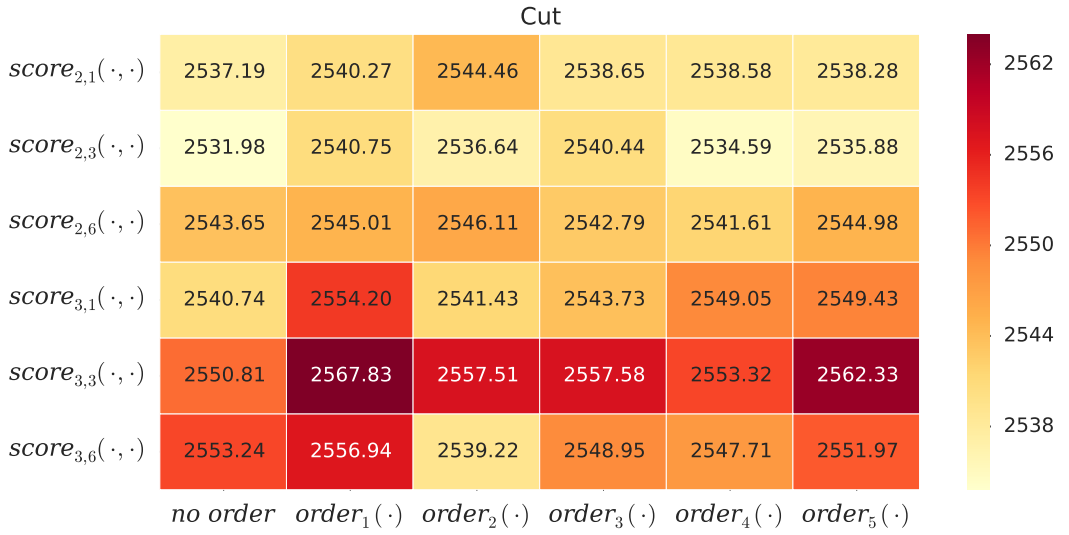


Figure D.1.: *The impact of different node orderings on the cut for label propagation on the clique expanded hypergraph as coarsening strategy. We select* $score_{2,3}(\cdot)$ *and* $order_4(\cdot)$ *as the best parameters for this variant.*

Figure D.2.: *The impact of different node orderings on the total partition time for label propagation on the clique expanded hypergraph as coarsening strategy. We select* $score_{2,3}(\cdot)$ *and* $order_4(\cdot)$ *as the best parameters for this variant.*

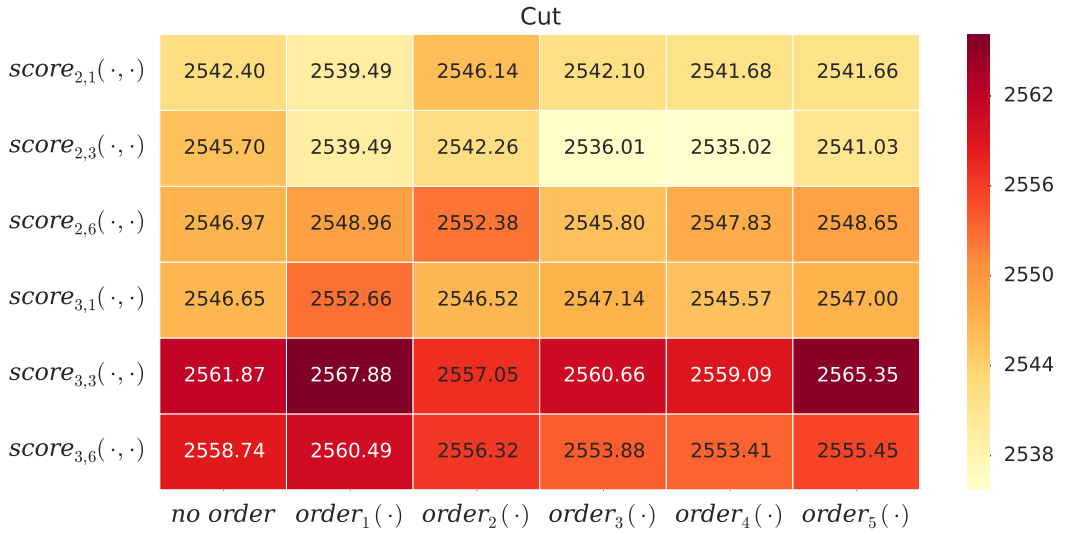


Figure D.3.: *The impact of different node orderings on the cut for probabilistic label propagation as coarsening strategy. We select* $score_{2,3}(\cdot)$ *and* $order_4(\cdot)$ *as the best parameters for this variant.*

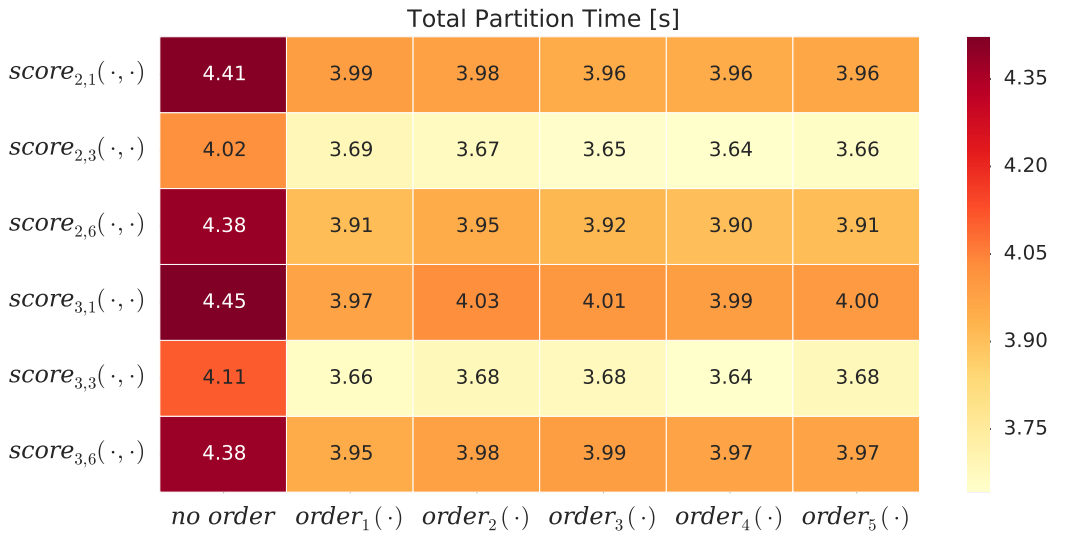


Figure D.4.: *The impact of different node orderings on the total partition time for probabilistic label propagation as coarsening strategy. We select* $score_{2,3}(\cdot)$ *and* $order_4(\cdot)$ *as the best parameters for this variant.*

D.3. Sample Size and Maximal Number of Iterations for Label Propagation in Coarsening Phase

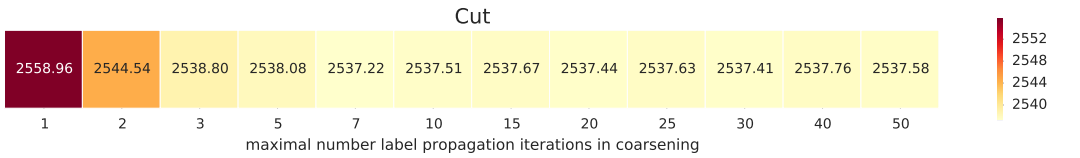


Figure D.5.: *The cut for different number of maximal iterations of label propagation on the clique expanded hypergraph as coarsening strategy.*

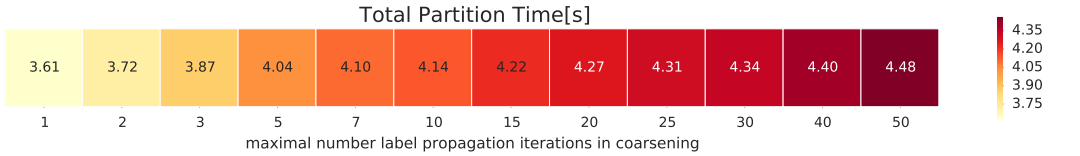


Figure D.6.: *The total partition time for different number of maximal iterations of label propagation on the clique expanded hypergraph as coarsening strategy.*

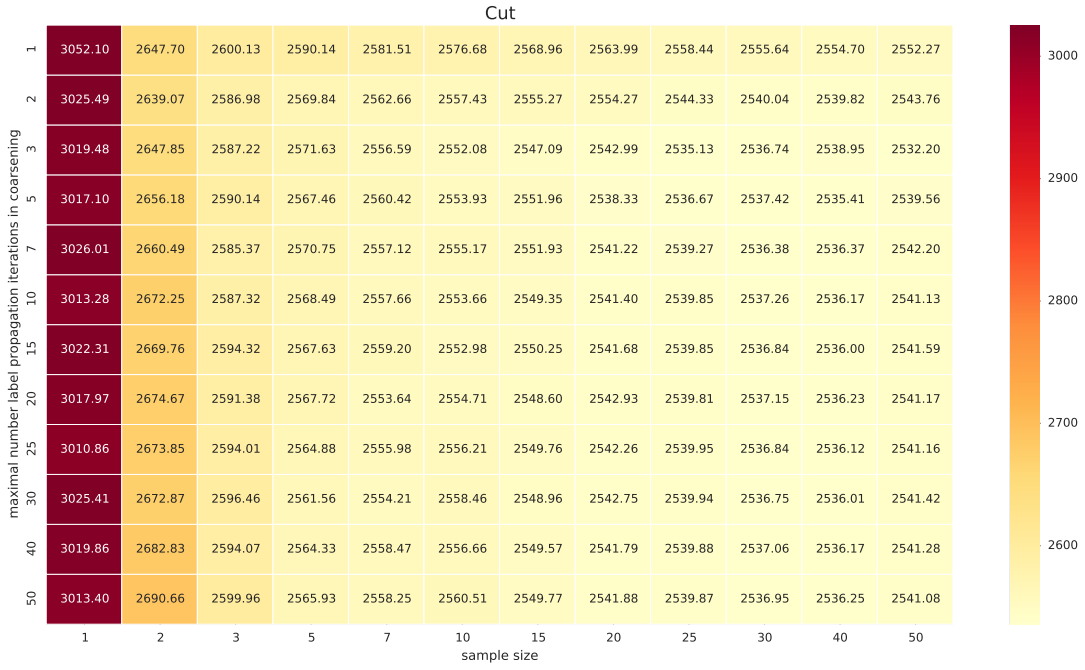


Figure D.7.: *The cut for different number of maximal iterations and sample sizes of probabilistic label propagation as coarsening strategy.*

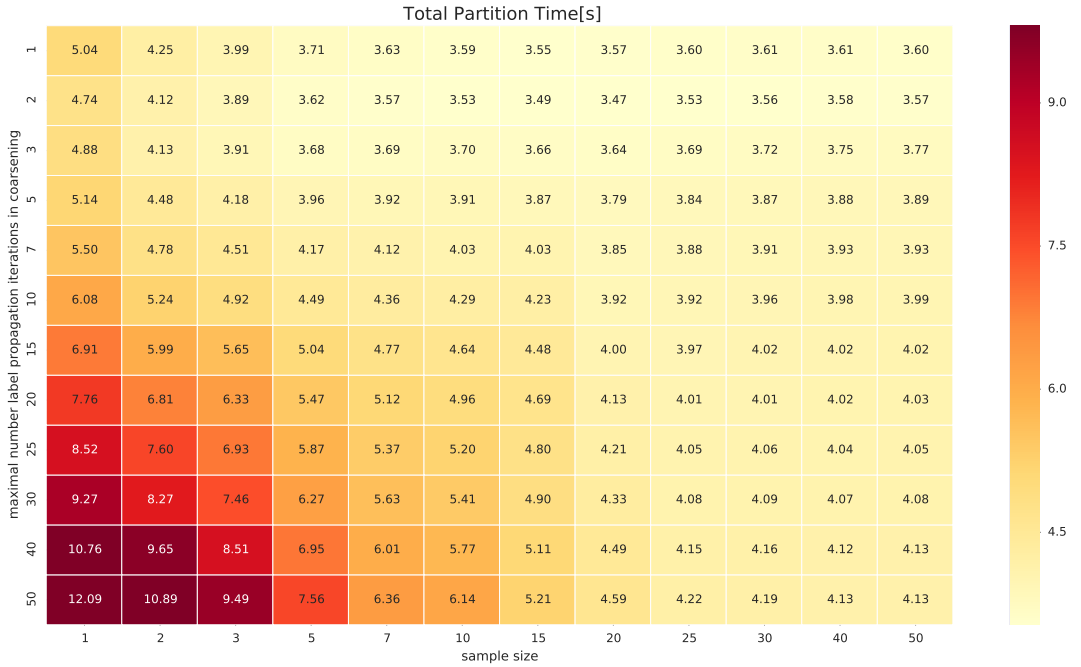


Figure D.8.: *The total partition time for different number of maximal iterations and sample sizes of probabilistic label propagation as coarsening strategy. The large partition time in the left corner is explained by the fact that we select only one sample per hyperedge: we perform the maximal number of label propagation iterations, since many hypernodes change their label (due to the bad sample) and our stopping rule therefore does not stop the coarsening process early.*

D.4. Size Constraint and Coarsening Threshold

In our algorithms, the coarsening threshold for a k-way partitioning is $t \cdot k$ for a tuning parameter t. This means that we leave the coarsening phase, if the total number of hypernodes falls below this threshold. The size constraint for the clusters is controlled by a parameter U and is $\frac{1}{U}(\frac{(1+\varepsilon)}{k} \cdot c(V) + \max_{v \in V} c(v))$ and denotes the maximal weight of all hypernodes sharing the same label. We consider only values of U which are below t, because otherwise the size constraint doesn't allow to reach $t \cdot k$ coarse hypernodes.

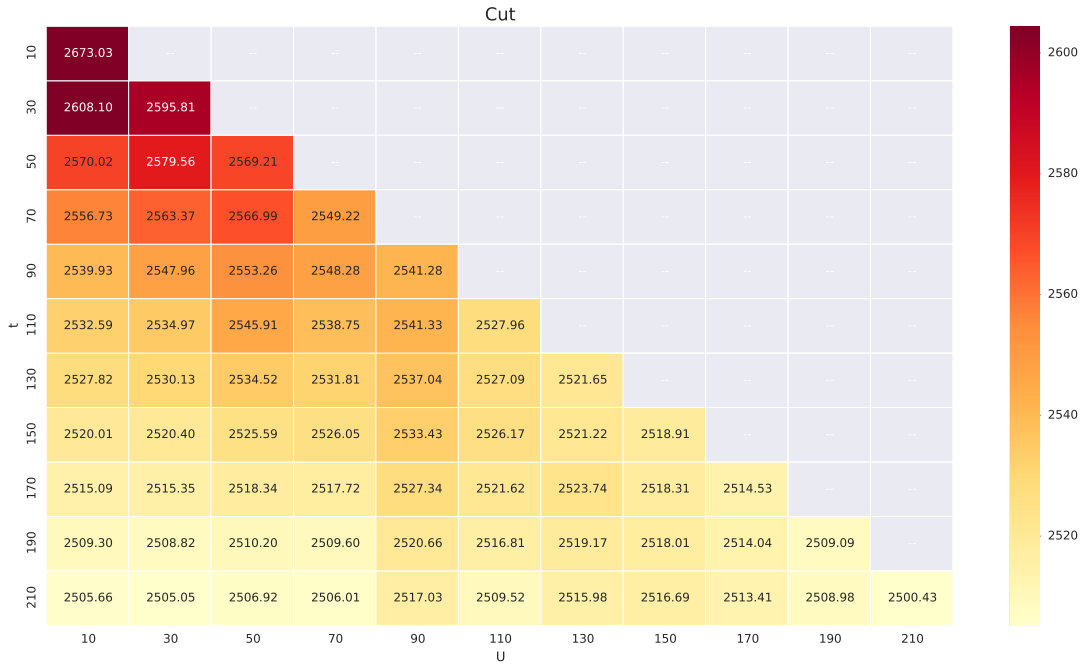


Figure D.9.: *The impact on the cut of different coarsening thresholds and different size constraints for label propagation on the clique expanded hypergraph as coarsening strategy.*

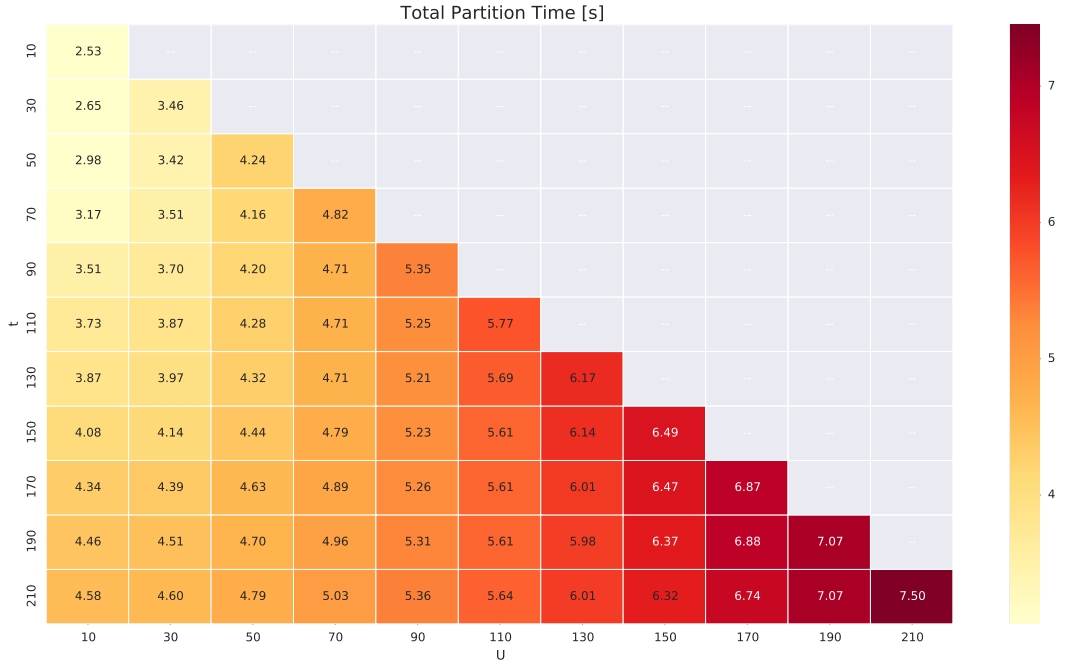


Figure D.10.: *The impact on the total partition time of different coarsening thresholds and different size constraints for label propagation on the clique expanded hypergraph as coarsening strategy.*

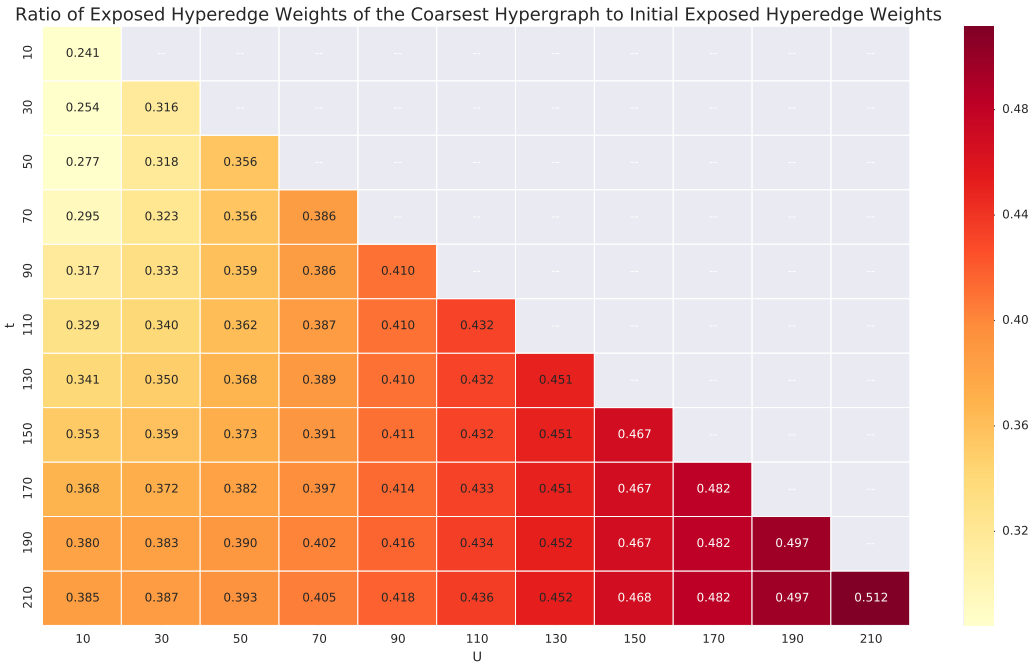


Figure D.11.: *The ratio between the sum of exposed hyperedge weights in the coarsest hypergraph to the sum of exposed hypergraph edges in the initial hypergraph for different coarsening thresholds and different size constraints for label propagation on the clique expanded hypergraph as coarsening strategy. First, note that the ratio correlates with the total partition time: if less hyperedge weights are exposed, our algorithm has a faster running time. This is because for small hypergraphs the running time for the initial partitioning algorithm dominates the other phases. Second, our algorithm performs better if the ratio is not reduced too much. This is explained by the fact that if we coarsen too long, many structural properties of the hypergraph get destroyed in the coarser hypergraphs.*

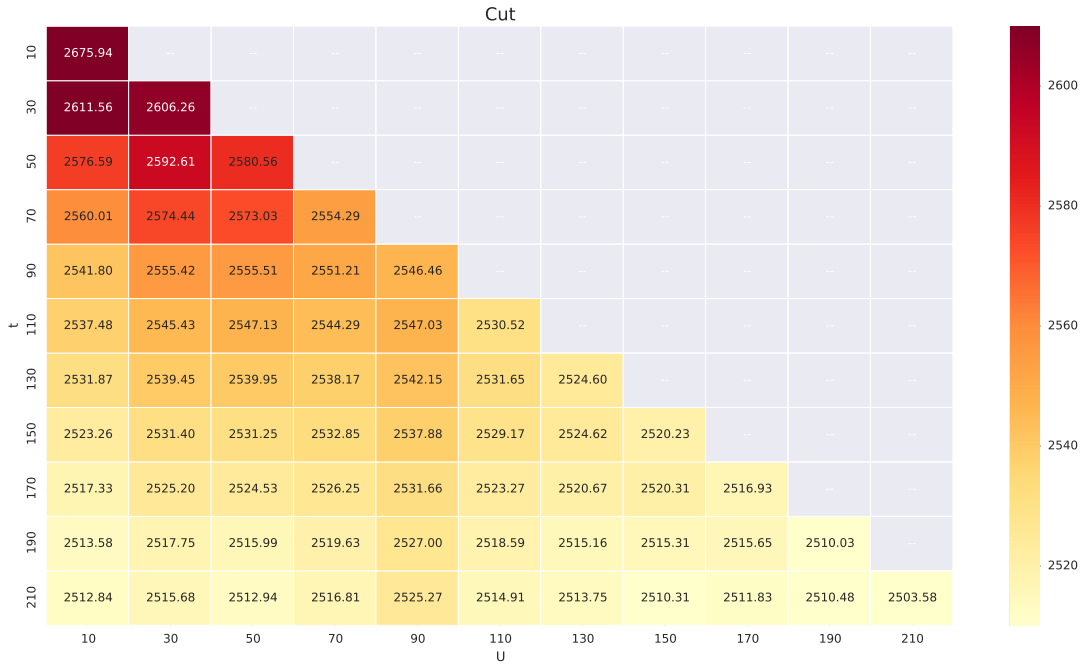


Figure D.12.: *The cut for different coarsening thresholds and different size constraints for probabilistic label propagation as coarsening strategy.*

Figure D.13.: *The total partition time for different coarsening thresholds and different size constraints for probabilistic label propagation as coarsening strategy.*

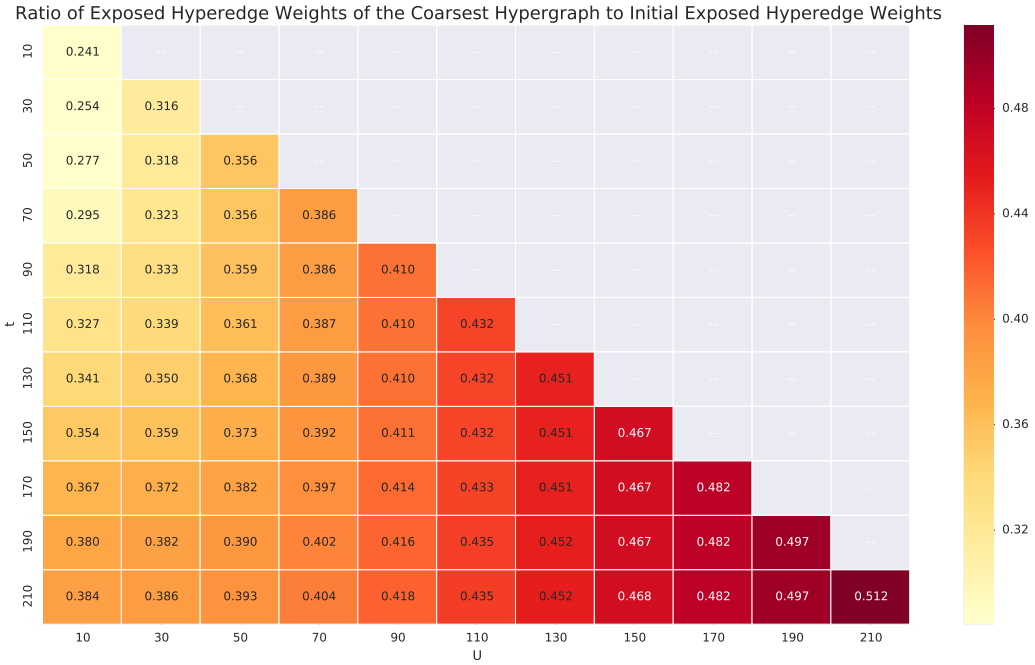


Figure D.14.: *The ratio between the sum of exposed hyperedge weights in the coarsest hypergraph to the sum of exposed hypergraph edges in the initial hypergraph for different coarsening thresholds and different size constraints for probabilistic label propagation as coarsening strategy. The same observations as in the clique expanded variant apply: First, note that the ratio correlates with the total partition time: if less hyperedge weights are exposed, our algorithm has a faster running time. This is because for small hypergraphs the running time for the initial partitioning algorithm dominates the other phases. Second, our algorithm performs better if the ratio is not reduced too much. This is explained by the fact that if we coarsen too long, many structural properties of the hypergraph get destroyed in the coarser hypergraphs.*

D.5. Initial Partitioner Comparison

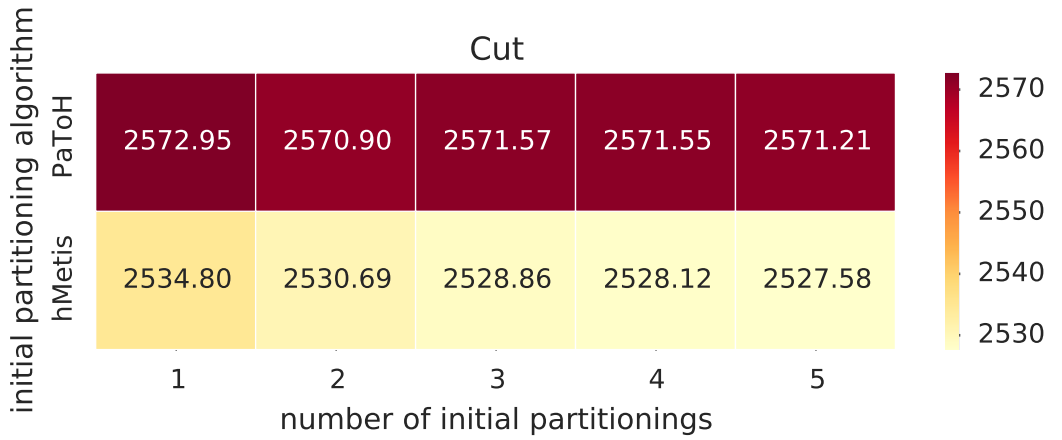


Figure D.15.: *The cut for different initial partitioners and multiple runs of the initial partitioner with label propagation on the clique expanded hypergraph as coarsening strategy.*

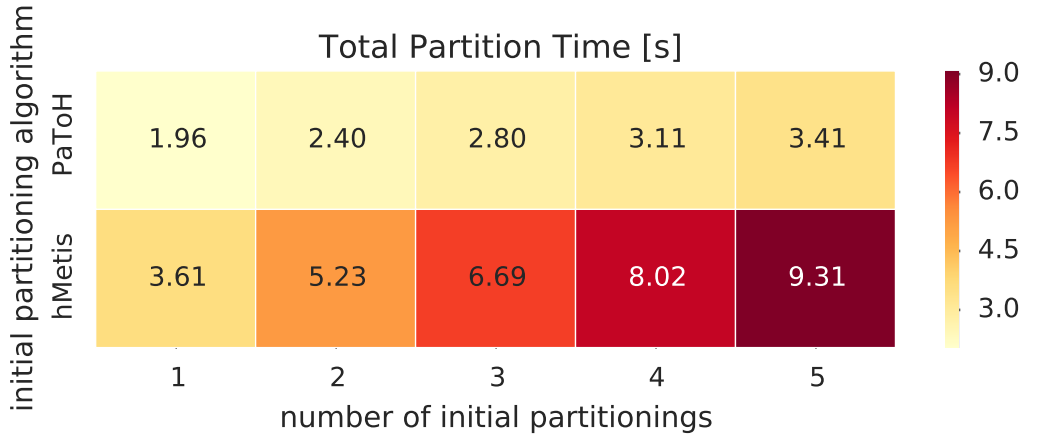


Figure D.16.: *The total partition time for different initial partitioners and multiple runs of the initial partitioner with label propagation on the clique expanded hypergraph as coarsening strategy.*

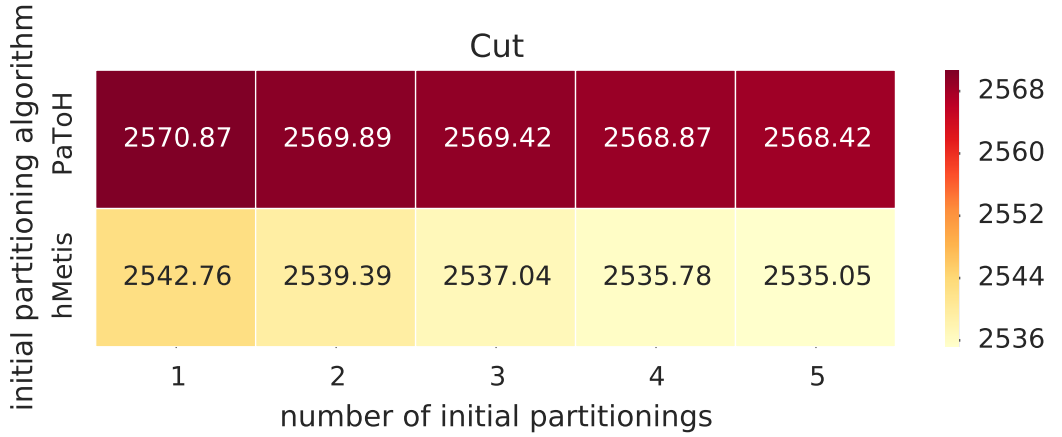


Figure D.17.: *The cut for different initial partitioners and multiple runs of the initial partitioner with probabilistic label propagation as coarsening strategy.*

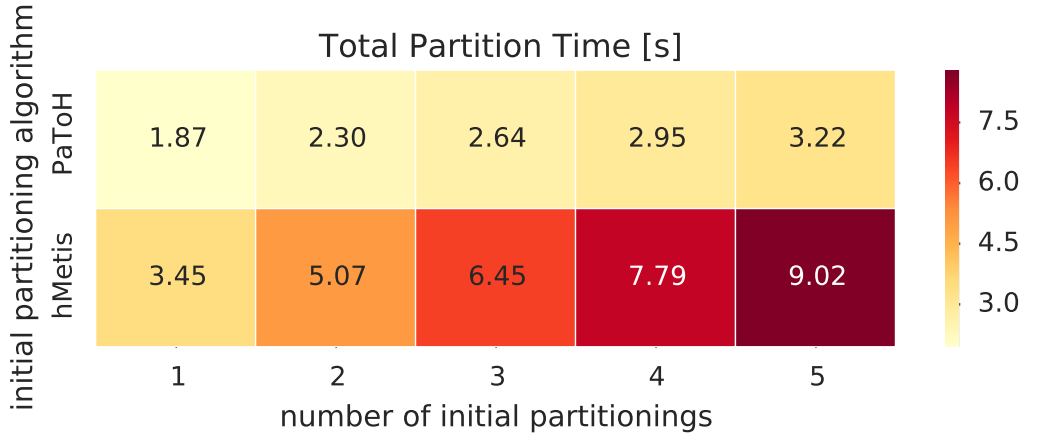


Figure D.18.: *The total partition time for different initial partitioners and multiple runs of the initial partitioner with probabilistic label propagation as coarsening strategy.*

D.6. Maximal Number of Iterations for Label Propagation in the Refinement Phase

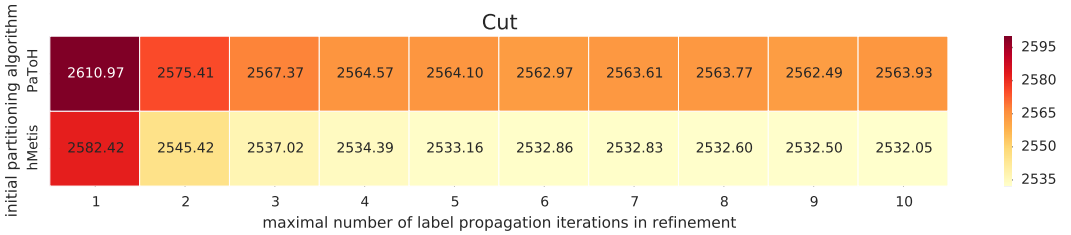


Figure D.19.: *The cut for different initial partitioners and different maximal number of iterations for label propagation as local search algorithm and label propagation on the clique expanded hypergraph as coarsening strategy.*

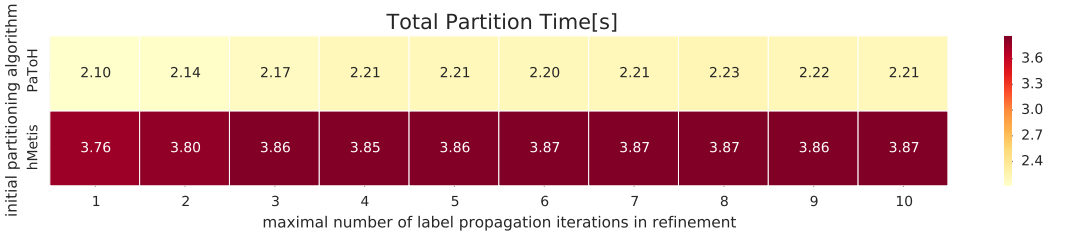


Figure D.20.: *The total partition time for different initial partitioners and different maximal number of iterations for label propagation as local search algorithm and label propagation on the clique expanded hypergraph as coarsening strategy. The minuscule change in the total partition time is because first, the running time of the initial partitioning algorithm dominates. Second, our greedy local search algorithm has only a very local view on the hypergraph and therefore does not perform the maximal number of iterations, since a local minimum is easily found.*

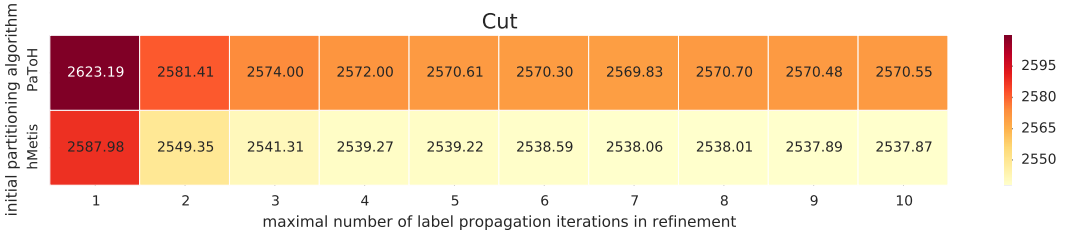


Figure D.21.: *The cut for different initial partitioners and different maximal number of iterations for label propagation as local search algorithm and probabilistic label propagation as coarsening strategy.*

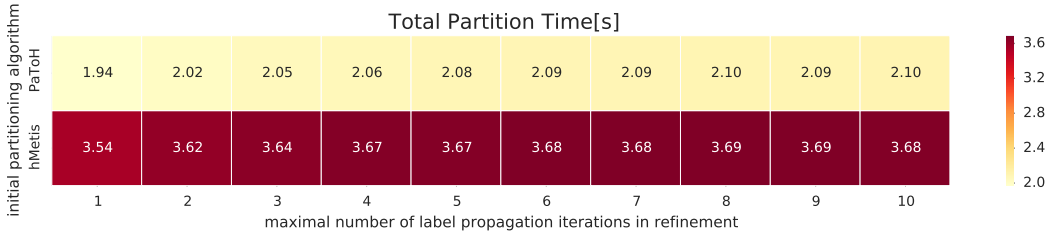


Figure D.22.: *The total partition time for different initial partitioners and different maximal number of iterations for label propagation as local search algorithm and probabilistic label propagation as coarsening strategy. As with the clique expanded variant, the minuscule change in the total partition time is because first, the running time of the initial partitioning algorithm dominates. Second, our greedy local search algorithm has only a very local view on the hypergraph and therefore does not perform the maximal number of iterations, since a local minimum is easily found.*

D.7. V-cycles

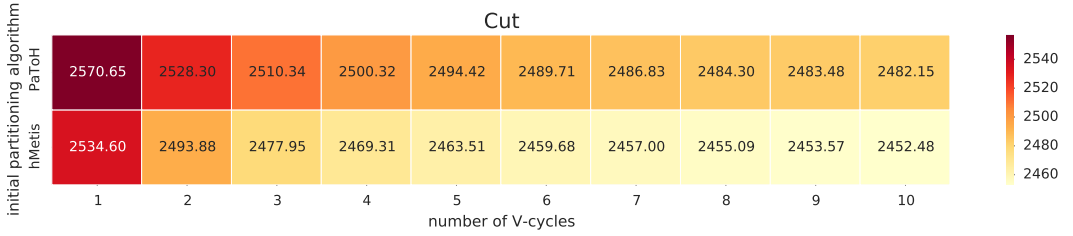


Figure D.23.: *The cut for different initial partitioners and different numbers of V-cycles utilizing label propagation on the clique expanded hypergraph as coarsening strategy.*

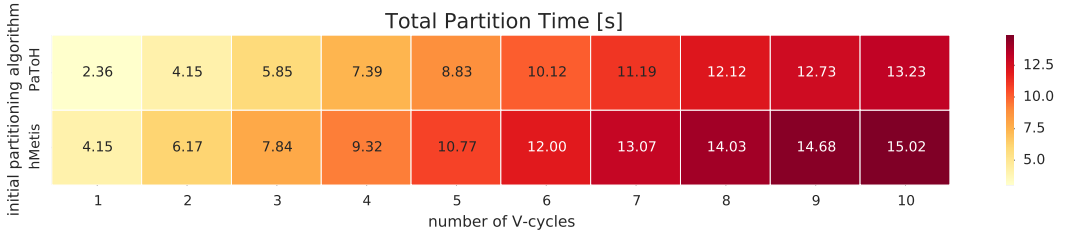


Figure D.24.: *The total partitioning time for different initial partitioners and different numbers of V-cycles utilizing label propagation on the clique expanded hypergraph as coarsening strategy.*

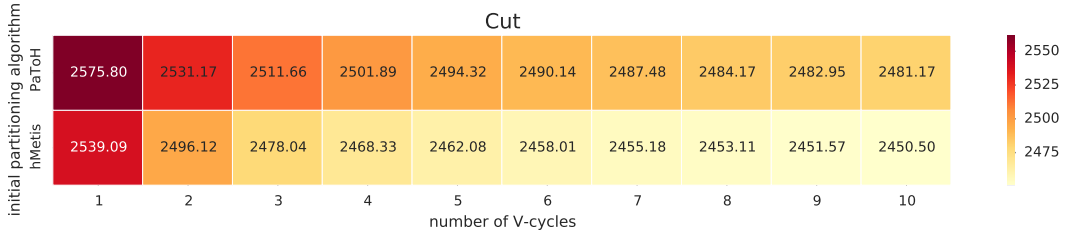


Figure D.25.: *The cut for different initial partitioners and different numbers of V-cycles utilizing probabilistic label propagation as coarsening strategy.*

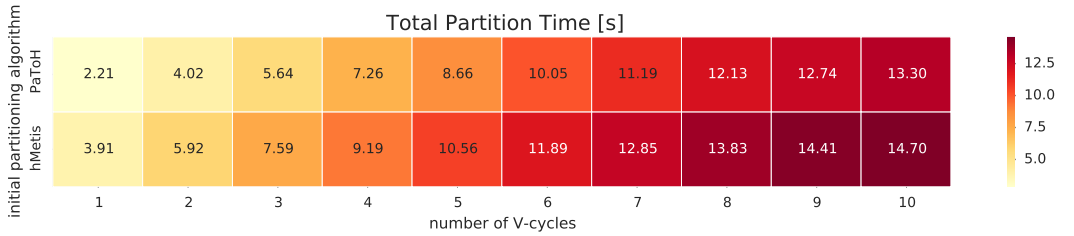


Figure D.26.: *The total partitioning time for different initial partitioners and different numbers of V-cycles utilizing probabilistic label propagation as coarsening strategy.*

E | Detailed Results

In this section we provide further results without comment.

Hypergraph	k	imbalance max	imbalance avg	count
ibm01	32	0.0301	0.0301	1
	64	0.0700	0.0500	10
	128	0.1000	0.0711	9
ibm02	64	0.1010	0.0730	10
	128	0.0974	0.0805	10
ibm03	64	0.0414	0.0414	2
	128	0.1050	0.0635	8
ibm04	64	0.0349	0.0326	2
	128	0.0512	0.0434	3
ibm05	64	0.0545	0.0436	2
	128	0.0870	0.0630	6
ibm06	16	0.0595	0.0595	1
	64	0.0965	0.0598	8
	128	0.0787	0.0543	10
ibm07	32	0.0348	0.0348	1
	64	0.0404	0.0390	2
	128	0.0474	0.0446	2
ibm08	16	0.0561	0.0492	4
	32	0.0661	0.0661	1
	64	0.0798	0.0611	4
	128	0.0948	0.0628	5
ibm09	16	0.0321	0.0321	1
	32	0.0467	0.0416	4
	64	0.0611	0.0493	7
	128	0.0981	0.0601	9
ibm10	16	0.0320	0.0315	2
	32	0.0516	0.0498	2
	64	0.0645	0.0485	6
	128	0.0866	0.0516	7
ibm11	64	0.0644	0.0477	8
	128	0.1178	0.0875	10
ibm12	64	0.0432	0.0369	3
	128	0.0594	0.0450	3
ibm13	32	0.0600	0.0469	2
	64	0.0509	0.0489	3
	128	0.0653	0.0486	8
ibm14	32	0.0377	0.0352	2
	64	0.0581	0.0466	5
	128	0.0789	0.0609	8
ibm15	32	0.0438	0.0438	1
	64	0.1006	0.0660	3
	128	0.0879	0.0639	10
ibm16	128	0.0809	0.0577	7
ibm17	128	0.1131	0.0715	3
ibm18	16	0.0456	0.0456	1
	32	0.0688	0.0443	4
	64	0.0887	0.0481	8
	128	0.0802	0.0589	9
avqlarge	8	0.0407	0.0407	1
	16	0.0540	0.0437	8
	32	0.0889	0.0576	10
	64	0.0838	0.0602	10
	128	0.0914	0.0685	10
avqsmall	8	0.0354	0.0352	2
	16	0.0628	0.0465	9
	32	0.0628	0.0530	10
	64	0.0758	0.0630	10
	128	0.0930	0.0738	10
golem3	32	0.0501	0.0501	1
	64	0.0370	0.0360	2
	128	0.0804	0.0455	9
industry2	16	0.0481	0.0418	2
	32	0.0684	0.0494	6
	64	0.0758	0.0561	10
	128	0.0909	0.0616	10
industry3	32	0.0393	0.0373	2
	64	0.0744	0.0613	6
	128	0.0826	0.0562	10
G3_circuit	64	0.0404	0.0404	1
	128	0.0348	0.0348	1
audikw_1	128	0.0442	0.0389	2
kkt_power	16	0.0519	0.0519	1
	32	0.0530	0.0449	6
	64	0.0720	0.0519	8
	128	0.1177	0.0965	10
ldoor	32	0.0665	0.0562	4
	64	0.0763	0.0457	7
	128	0.0774	0.0620	10
thermal2	32	0.0352	0.0352	1
	64	0.0471	0.0429	2
	128	0.0496	0.0395	6
bcsstk29	32	0.0411	0.0388	6
	64	0.0639	0.0425	10
	128	0.0727	0.0527	10
bcsstk30	32	0.0631	0.0537	2
	64	0.0553	0.0465	2
	128	0.0708	0.0476	8
bcsstk31	32	0.0701	0.0571	4
	64	0.0844	0.0655	10
	128	0.1111	0.0846	10
bcsstk32	16	0.0531	0.0531	1
	32	0.0746	0.0466	6
	64	0.0946	0.0673	10
	128	0.0917	0.0719	10
memplus	8	0.0396	0.0396	1
	16	0.0514	0.0389	6
	32	0.0703	0.0560	10
	64	0.0827	0.0680	10
	128	0.0935	0.0763	10
vibrobox	64	0.0881	0.0544	4
	128	0.0619	0.0515	5

Table E.1.: The imbalanced partitions computed by hMetis-k.

Table E.2.: Detailed results of the various hypergraph partitioners.

		LPFast			LPEco			LPBest			hMetis-k			hMetis-RB			PaToH-Q			PaToH-D		
k	Hypergraph	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
2	ibm01	283	300,6	0,16	263	274,4	0,75	223	264,3	1,09	203	206,3	0,86	203	203,1	1,17	252	252	0,15	265	290,3	0,03
2	ibm02	375	435,7	0,31	362	388,9	1,57	362	389,9	2,90	351	359,7	2,82	344	349,4	3,97	375	375	0,21	369	401,5	0,04
2	ibm03	1 050	1 184,2	0,35	994	1 030,9	1,82	984	1 012,6	3,47	959	962,4	2,60	957	960,2	2,93	989	989	0,25	990	1 016,5	0,04
2	ibm04	640	692,6	0,41	597	613,5	1,92	601	621,3	3,85	581	585,5	2,82	580	583	3,22	628	628	0,30	599	634,4	0,05
2	ibm05	1 766	1 869,1	0,52	1 728	1 765	2,73	1 726	1 748,4	5,25	1 726	1 729,7	6,11	1 721	1 726,3	6,81	1 730	1 730	0,28	1 740	1 757,8	0,07
2	ibm06	1 029	1 114,4	0,54	1 000	1 037,4	2,66	1 017	1 039	4,66	984	998,7	3,92	981	988,8	4,67	1 051	1 051	0,38	990	1 058,6	0,07
2	ibm07	1 016	1 220,9	0,76	971	1 051,6	3,98	994	1 019	8,55	953	970,5	5,63	925	950,8	6,66	1 002	1 002	0,37	997	1 028,4	0,08
2	ibm08	1 200	1 260,1	0,91	1 177	1 188,1	4,74	1 176	1 180,4	11,11	1 144	1 151,4	6,76	1 141	1 145	9,05	1 165	1 165	0,46	1 157	1 220,1	0,10
2	ibm09	644	777,3	0,89	627	635,5	4,19	625	639,2	6,81	627	632,1	5,08	627	631,6	5,39	638	638	0,49	639	683,9	0,10
2	ibm10	1 461	1 772,3	1,39	1 328	1 410,1	7,51	1 326	1 375,8	15,55	1 317	1 340	9,54	1 328	1 340,7	10,44	1 658	1 658	0,73	1 517	1 763,6	0,14
2	ibm11	1 097	1 214,7	1,27	1 078	1 101,5	6,61	1 079	1 171,9	13,08	1 063	1 068,8	7,11	1 065	1 067,7	8,02	1 105	1 105	0,57	1 078	1 184,9	0,12
2	ibm12	2 235	2 333,4	1,53	2 070	2 106	8,36	2 057	2 111,2	19,21	1 995	2 052,3	13,14	1 951	1 969	15,14	1 988	1 988	0,74	2 071	2 180,8	0,16
2	ibm13	877	986,9	1,76	835	885,4	9,86	842	882,8	11,48	833	843,4	11,26	837	850,4	10,52	891	891	0,70	877	1 134,5	0,16
2	ibm14	2 078	2 227,8	3,37	2 010	2 099	19,16	1 975	2 087,6	36,99	1 874	1 911,1	21,05	1 869	1 881,2	23,80	2 107	2 107	1,26	2 086	2 292,7	0,28
2	ibm15	2 808	3 184,4	4,25	2 782	2 838,1	22,92	2 754	2 859,3	42,64	2 781	2 837,3	28,91	2 744	2 808,6	29,75	2 768	2 768	1,56	2 790	3 149,2	0,35
2	ibm16	2 364	2 522,3	4,84	2 065	2 298,5	27,17	2 003	2 164,9	62,07	2 006	2 046,8	30,84	1 913	1 978,5	34,61	2 080	2 080	1,30	2 128	2 314,3	0,42
2	ibm17	2 732	3 422,3	5,37	2 463	2 769,5	30,06	2 462	2 519,5	63,73	2 318	2 354,4	42,23	2 317	2 343,7	46,42	2 535	2 535	2,03	2 462	2 721,8	0,47
2	ibm18	2 020	2 211,5	5,31	1 858	2 036,1	29,79	1 686	1 825	69,60	1 873	1 983,2	37,12	1 683	1 878,7	40,41	2 001	2 001	1,78	1 739	2 054,4	0,46
2	avqlarge	160	204,9	0,41	144	157,6	1,37	166	170	8,67	142	143	0,69	141	141,9	0,93	152	152	0,51	149	167,1	0,04
2	avqsmall	162	230,8	0,38	143	153,8	1,32	154	159,1	10,00	142	144,1	0,67	142	142,3	0,83	165	165	0,49	149	168,9	0,03
2	golem3	1 353	1 376	1,65	1 333	1 343,5	7,76	1 333	1 345,8	10,27	1 333	1 338,7	5,63	1 329	1 336	6,26	1 347	1 347	0,60	1 340	1 350,5	0,13
2	industry2	211	245,8	0,17	191	216,1	0,63	183	190,4	1,71	187	194,2	0,82	178	185,5	1,14	190	190	0,12	197	257,6	0,03
2	industry3	295	321,6	0,21	293	306,8	0,65	286	306,1	0,88	282	282,6	1,04	282	282	1,35	284	284	0,12	282	293,9	0,03
2	G3_circuit	2 481	2 637,3	106,27	2 356	2 518,5	555,35	2 262	2 505	845,47	2 326	2 412,8	282,62	2 142	2 142	258,41	2 430	2 430	9,74	2 208	2 355,2	2,15
2	af_shell10	5 440	5 831,5	88,05	5 290	5 388	430,70	5 270	5 325	439,52	5 250	5 294	382,75	5 250	5 250	359,60	5 290	5 290	21,47	5 250	5 392	10,57
2	af_shell9	2 010	2 101,5	12,93	1 910	1 987,5	58,15	1 875	1 964	79,14	1 850	1 896	111,56	1 770	1 770	102,96	1 855	1 855	6,13	1 830	1 911	2,46
2	audikw_1	11 502	11 841,3	96,08	10 761	11 112	536,43	10 680	11 083,8	1 082,36	10 914	11 083,2	676,12	10 986	11 140,2	865,87	10 860	10 860	97,02	10 962	11 199,3	37,39
2	ecology1	2 036	2 074,4	44,92	2 000	2 010,8	239,95	2 000	2 015,4	319,52	2 000	2 000,2	104,63	2 000	2 000	127,21	2 000	2 000	3,24	2 000	2 007,2	1,03
2	ecology2	2 022	2 072	44,97	2 000	2 018,2	219,80	2 000	2 019	280,82	2 000	2 000	106,11	1 998	1 998,6	129,93	2 000	2 000	3,34	2 000	2 024,4	1,04
2	kkt_power	9 898	11 709	189,99	9 731	9 779,1	642,16	9 731	9 798,9	709,36	8 104	8 699	382,56	8 104	8 219,4	437,80	10 050	10 050	54,02	9 853	10 509,5	9,74
2	ldoor	3 500	3 693,2	40,99	3 304	3 421,6	204,16	3 311	3 355,1	242,99	3 276	3 327,8	327,59	3 206	3 273,9	353,13	3 584	3 584	23,44	3 381	3 520,3	12,75
2	nlpkkt120	74 372	86 123,5	854,23	70 890	75 026	4 079,82	70 015	72 921,8	9 079,01	70 606	71 812,6	5 618,75	58 560	58 560	4 735,45	58 560	58 560	94,77	60 208	64 489,8	32,81
2	thermal2	1 064	1 160,7	71,22	1 010	1 064,4	387,75	1 002	1 057	799,66	973	998,9	236,38	964	978,1	258,10	990	990	9,29	980	997,8	2,27
2	bcsstk29	378	405,3	0,50	360	382,2	1,46	377	387,4	1,66	360	363,6	2,70	360	360	3,17	360	360	0,42	372	387	0,12
2	bcsstk30	553	639,6	1,06	528	579,9	2,95	527	561,4	5,12	535	544,5	5,73	527	552,8	7,59	578	578	1,28	528	577,9	0,56
2	bcsstk31	746	801,1	1,00	678	771,1	3,80	669	735,5	5,56	674	695,6	8,44	667	672,7	9,85	678	678	0,69	665	744,7	0,22
2	bcsstk32	1 016	1 074	1,02	846	919,4	3,63	868	940,3	4,98	986	1 029,1	9,77	958	1 036,9	14,18	918	918	1,40	966	1 032,9	0,39
2	finan512	146	147	1,73	146	147,1	4,01	146	147,2	5,31	146	146,8	7,65	146	146,3	9,37	148	148	0,50	147	147,9	0,15
2	memplus	2 904	2 958,5	0,35	2 888	2 914,4	1,64	2 863	2 883,1	2,98	2 696	2 710,8	1,63	2 465	2 513,7	2,23	2 983	2 983	0,22	2 922	3 051,2	0,06
2	vibrobox	2 333	2 476,4	0,76	2 119	2 305,1	3,62	2 092	2 312,4	4,03	1 990	1 990	4,81	1 990	1 990	5,51	1 990	1 990	0,51	1 990	2 213,3	0,13
4	ibm01	572	679,8	0,18	544	615,2	0,97	585	618,2	1,83	495	520,4	1,48	535	537,2	2,50	640	640	0,23	546	656,5	0,04
4	ibm02	730	824,6	0,33	716	766,6	2,23	665	726,5	4,49	648	681,8	4,01	703	714,7	6,93	705	705	0,40	689	839,2	0,08
4	ibm03	2 006	2 113,3	0,37	1 777	1 870,7	2,29	1 730	1 818,3	5,17	1 670	1 697,6	3,51	1 703	1 733,5	5,27	1 887	1 887	0,47	1 832	1 991,6	0,07
4	ibm04	1 823	1 947,6	0,41	1 778	1 826,2	2,29	1 665	1 803,4	6,13	1 651	1 673,2	4,08	1 692	1 711,4	6,53	1 815	1 815	0,55	1 830	1 906	0,09
4	ibm05	3 378	3 613,2	0,55	3 034	3 141,4	2,96	3 038	3 121,2	8,78	3 045	3 065,6	8,82	3 091	3 113,7	11,29	3 145	3 145	0,64	3 162	3 211,5	0,12
4	ibm06	1 542	1 717,2	0,58	1 529	1 612,6	3,17	1 516	1 541,3	6,45	1 495	1 515,5	5,58	1 696	1 727,8	8,46	1 546	1 546	0,59	1 573	1 826,4	0,11
4	ibm07	2 432	2 691,8	0,83	2 300	2 366,5	4,66	2 244	2 319,2	10,70	2 235	2 256,7	7,63	2 216	2 258,7	12,55	2 364	2 364	0,71	2 380	2 524,4	0,14
4	ibm08	2 503	2 619,7	1,01	2 422	2 441,9	5,54	2 410	2 427,9	13,31	2 382	2 407,4	9,92	2 468	2 475,6	15,85	2 470	2 470	0,94	2 510	2 627,6	0,18
4	ibm09	1 779	1 922,8	0,93	1 734	1 779,6	5,11	1 731	1 776	9,36	1 700	1 726,6	6,65	1 734	1 746	11,29	1 868	1 868	0,89	1 821	2 178,8	0,16
4	ibm10	2 678	3 347,8	1,42	2 520	2 680,8	8,21	2 507	2 652,9	20,04	2 423	2 488,8	11,84	2 481	2 551,3	20,45	2 861	2 861	1,25	2 775	3 139,7	0,25
4	ibm11	2 820	3 077,4	1,35	2 529	2 626,1	7,51	2 510	2 596,8	15,68	2 484	2 513,8	9,33	2 492	2 530,5	15,50	2 755	2 755	1,13	2 810	3 038,2	0,21
4	ibm12	4 333	4 775	1,63	4 067	4 316	9,03	4 133	4 316	22,78	3 994	4 046,7	14,84	3 918	3 984,7	24,89	4 848	4 848	1,35	4 323	4 603,4	0,27
4	ibm13	2 363	2 769,8	1,77	2 031	2 116,7	10,42	1 965	2 086,6	22,05	1 845	1 901,3	13,15	1 953	1 971,3	20,01	2 182	2 182	1,27	2 016	2 402,2	0,28

Table E.2.: Detailed results of the various hypergraph partitioners.

		LPFast			LPEco			LPBest			hMetis-k			hMetis-RB			PaToH-Q			PaToH-D		
k	Hypergraph	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
4	ibm14	3 763	4 039	3,48	3 464	3 574	19,64	3 470	3 568,5	47,85	3 370	3 436,8	26,24	3 379	3 433,2	40,62	3 568	3 568	2,23	3 541	3 981,2	0,50
4	ibm15	5 276	5 584,9	4,35	4 865	5 198,9	24,27	4 866	5 112,8	56,85	4 836	4 911,2	33,44	4 825	4 984,8	52,91	5 245	5 245	2,61	5 599	5 970,5	0,63
4	ibm16	4 805	5 121,1	4,99	4 442	4 726,4	27,77	4 365	4 600,4	67,84	4 175	4 276,6	38,70	4 180	4 261,8	63,24	4 622	4 622	3,37	4 586	4 986,1	0,74
4	ibm17	6 527	7 370,7	5,55	5 791	6 336,4	30,66	5 832	6 133,3	77,94	5 685	5 900,3	52,25	5 665	5 727,9	82,20	5 783	5 783	3,57	6 388	7 024,5	0,85
4	ibm18	3 592	3 800,4	5,46	3 310	3 417,8	30,08	3 153	3 376,6	74,64	3 111	3 208,8	46,38	3 191	3 228,5	77,44	3 950	3 950	3,29	3 866	4 150	0,81
4	avqlarge	331	355,6	0,47	292	304,9	1,93	278	308,4	8,51	254	273,6	1,08	271	276,7	1,70	330	330	0,57	302	330,4	0,04
4	avqsmall	302	332,8	0,46	285	305,9	1,67	288	299,4	11,18	254	282,2	0,91	250	274,5	1,52	301	301	0,54	299	329,6	0,04
4	golem3	2 272	2 323,5	1,66	2 218	2 240,3	9,50	2 218	2 242,3	15,98	2 232	2 248,8	6,89	2 225	2 230	10,87	2 244	2 244	1,06	2 271	2 295,5	0,22
4	industry2	364	437,5	0,18	343	382,5	0,82	322	368,4	2,21	324	340,8	1,31	320	360,8	2,23	366	366	0,21	395	442,6	0,06
4	industry3	799	923,3	0,23	733	775,3	1,21	726	773,5	2,72	740	771,6	1,86	742	744,3	2,93	753	753	0,24	754	847,6	0,05
4	G3_circuit	6 035	6 483,1	106,58	5 927	6 167,8	567,52	5 755	5 978,3	1 024,33	5 778	5 927,6	312,77	5 278	5 278	488,09	6 640	6 640	15,21	5 680	6 268,8	3,87
4	af_shell10	12 535	13 198	88,13	11 790	11 987,5	449,58	11 510	11 754,5	768,89	11 335	11 614	466,23	10 905	11 043	688,74	11 840	11 840	40,62	11 495	11 879,5	20,57
4	af_shell9	5 025	5 186,5	12,80	4 545	4 705	66,38	4 530	4 647	81,07	4 495	4 535	116,57	4 370	4 380	194,86	4 550	4 550	11,87	4 500	4 719	4,72
4	audikw_1	34 587	35 327,1	101,78	32 961	33 747	681,29	32 850	33 267,6	2 976,99	33 396	33 673,2	729,31	33 864	34 680,9	1 699,25	34 800	34 800	191,89	34 200	34 970,1	73,00
4	ecology1	3 981	4 109,5	44,99	3 860	3 960,7	246,00	3 830	3 908,8	463,95	3 842	3 877,4	144,87	3 667	3 777,7	234,12	3 938	3 938	7,60	3 745	3 977,5	1,81
4	ecology2	3 973	4 099,7	45,08	3 884	3 961,2	244,98	3 872	3 922,7	404,61	3 843	3 872,6	136,97	3 757	3 848	233,99	3 945	3 945	7,35	3 919	3 982,8	1,82
4	kkt_power	20 222	27 774,9	193,02	18 263	20 193,7	955,16	18 872	20 365,4	1 787,55	18 616	19 437	413,41	17 538	17 981,7	766,20	23 246	23 246	62,85	18 923	21 236,7	16,21
4	ldoor	7 119	7 476,7	41,17	6 748	6 892,9	209,17	6 489	6 782,3	429,12	6 510	6 645,8	370,63	6 538	6 640,2	629,39	7 266	7 266	59,89	6 790	7 282,1	24,89
4	nlpkkt120	155 966	165 572,3	861,28	144 505	150 469,1	4 160,38	136 494	141 991,2	8 955,70	141 014	143 655,9	5 680,63	116 152	116 211,2	8 574,06	119 387	119 387	236,32	125 448	134 492,7	59,39
4	thermal2	3 268	3 499,9	71,25	3 090	3 301,8	388,06	3 037	3 244	873,17	2 984	3 022,9	257,54	2 891	2 915,1	488,52	3 035	3 035	17,43	2 989	3 039,9	4,14
4	bcsstk29	1 182	1 260,8	0,51	1 116	1 149,6	2,19	1 170	1 191	2,25	1 092	1 104	3,17	1 086	1 088,4	6,84	1 158	1 158	0,58	1 146	1 198,2	0,22
4	bcsstk30	1 630	1 779,4	1,11	1 518	1 614	4,49	1 502	1 594,6	7,31	1 488	1 544,7	7,03	1 524	1 598,1	18,56	1 572	1 572	2,46	1 576	1 657,5	1,08
4	bcsstk31	1 814	2 010,9	1,02	1 681	1 800	5,04	1 714	1 806,8	7,33	1 660	1 711,8	11,37	1 687	1 720	19,21	1 975	1 975	1,69	1 725	1 953,3	0,42
4	bcsstk32	1 720	2 199	1,04	1 711	1 971,6	4,70	1 720	2 040,4	5,63	1 693	1 764,8	11,84	2 118	2 207,2	27,52	1 696	1 696	1,82	1 707	2 134,8	0,74
4	finan512	292	309	1,74	293	294,4	4,08	292	301,7	5,47	293	294,3	8,14	292	293,2	18,01	296	296	0,93	295	303,2	0,27
4	memplus	4 990	5 087,3	0,35	4 741	4 964,9	1,47	4 621	4 896,2	4,34	4 133	4 147,1	2,25	3 974	4 069,5	3,20	4 422	4 422	0,26	4 354	4 488,2	0,07
4	vibrobox	4 036	4 353,2	0,86	3 831	3 961,4	3,93	3 848	3 985,7	6,09	3 796	3 798,5	7,79	3 598	3 604	9,73	3 660	3 660	0,73	3 642	3 945,5	0,18
8	ibm01	934	1 019,8	0,18	831	904	1,58	831	883,3	4,01	809	820,2	2,52	808	823,4	3,80	875	875	0,34	920	978,2	0,05
8	ibm02	2 248	2 345,4	0,44	1 932	2 104,3	2,83	2 054	2 145	7,37	2 013	2 069,9	7,78	2 005	2 054,3	10,37	1 963	1 963	0,69	1 986	2 162,5	0,11
8	ibm03	2 823	2 908	0,43	2 631	2 732,7	2,55	2 618	2 699,4	6,23	2 427	2 447,9	5,68	2 504	2 521,6	7,63	2 704	2 704	0,61	2 869	3 012,9	0,10
8	ibm04	3 127	3 212,3	0,51	2 967	3 027,5	3,20	2 905	3 001,9	8,23	2 808	2 859,2	6,48	2 850	2 909	9,37	3 009	3 009	0,75	3 165	3 269,8	0,12
8	ibm05	4 844	5 090	0,64	4 573	4 709,4	3,92	4 489	4 698,6	9,76	4 408	4 486,5	12,84	4 489	4 553,5	14,03	4 698	4 698	0,86	4 764	4 884,8	0,15
8	ibm06	2 442	2 516,5	0,68	2 425	2 453,4	4,28	2 411	2 441,1	9,70	2 408	2 419,4	8,60	2 435	2 457,2	11,24	2 530	2 530	0,84	2 482	2 582,6	0,14
8	ibm07	3 663	3 888,2	0,87	3 535	3 621,7	4,94	3 496	3 565,1	16,38	3 366	3 434,7	10,71	3 459	3 513,9	16,22	3 584	3 584	1,17	3 780	3 908,8	0,20
8	ibm08	3 798	3 864,3	1,05	3 557	3 609,2	5,90	3 522	3 592,2	19,17	3 539	3 588,1	13,87	3 679	3 703,6	21,80	3 735	3 735	1,36	3 785	3 982,8	0,25
8	ibm09	2 990	3 213,4	0,96	2 740	2 856	6,24	2 720	2 816,1	14,18	2 658	2 723,8	8,96	2 719	2 737,7	15,79	2 953	2 953	1,22	3 024	3 199,8	0,23
8	ibm10	4 415	5 033,4	1,53	4 263	4 504,3	8,66	4 331	4 470	27,12	4 056	4 136,8	16,65	4 216	4 298	28,35	4 658	4 658	1,84	4 463	4 939,3	0,34
8	ibm11	3 887	4 172,8	1,38	3 732	3 925,5	7,92	3 642	3 839	21,92	3 572	3 644,7	11,96	3 750	3 793,4	22,51	3 826	3 826	1,48	4 024	4 253,3	0,29
8	ibm12	6 452	7 093,6	1,69	6 175	6 442,1	9,48	6 142	6 386,6	29,64	5 999	6 135,3	17,70	6 139	6 235,6	33,33	6 287	6 287	1,97	6 641	7 048,6	0,38
8	ibm13	3 717	4 043,2	1,90	3 099	3 287,6	10,88	3 067	3 252,3	27,67	2 891	3 012,2	17,10	3 053	3 094,5	28,35	3 718	3 718	2,05	3 245	3 867,6	0,39
8	ibm14	5 699	6 092,9	3,55	5 342	5 448,2	21,76	5 286	5 400,5	55,45	5 078	5 185,3	31,15	5 070	5 295,5	59,06	5 879	5 879	3,28	5 688	6 151,5	0,70
8	ibm15	7 712	8 003,6	4,44	6 719	7 020,8	25,95	6 648	6 899,3	65,87	6 523	6 741,4	38,98	6 639	6 740,3	71,35	7 735	7 735	3,97	7 772	8 430,8	0,87
8	ibm16	7 740	8 274	5,09	6 989	7 240,2	30,24	7 034	7 119,8	77,57	6 737	6 903,8	43,75	6 927	6 982,3	87,30	7 911	7 911	4,84	7 860	8 237,4	1,03
8	ibm17	11 171	12 089,4	5,70	9 977	10 312,3	33,42	9 966	10 149,9	85,44	9 551	9 897,6	59,47	9 654	9 775,6	116,90	10 142	10 142	5,35	10 691	11 805,1	1,18
8	ibm18	6 599	7 131,7	5,58	6 050	6 414,1	31,59	5 890	6 201,5	87,62	5 773	6 098,6	52,15	5 934	6 111,5	102,37	6 582	6 582	4,68	6 817	7 168,1	1,11
8	avqlarge	449	492,8	0,54	382	408,8	2,26	383	406,3	9,54	397	417,7	1,68	407	411,8	2,60	485	485	0,64	457	511,1	0,05
8	avqsmall	463	474	0,51	387	416,2	2,09	396	424,2	9,70	381	422,4	1,54	391	407,7	2,33	466	466	0,59	475	508,7	0,05
8	golem3	2 955	2 993,3	1,69	2 885	2 915,5	9,70	2 885	2 896,4	17,59	2 884	2 900,1	8,12	2 874	2 879,7	15,37	2 939	2 939	1,40	2 953	2 977,2	0,31
8	industry2	786	823,2	0,19	695	713,2	1,40	693	713,7	3,61	601	635,1	2,31	633	651,5	3,30	712	712	0,30	713	766	0,07
8	industry3	1 618	1 749,2	0,26	1 530	1 589,3	1,45	1 530	1 544,4	5,77	1 561	1 582,7	3,25	1 524	1 535,9	4,63	1 617	1 617	0,35	1 579	1 697,7	0,07
8	G3_circuit	11 265	11 711,4	106,76	10 607	10 912,2	568,80	10 556	10 730,4	1 260,45	10 394	10 635,7	328,02	9 609	9 685,4	657,37	10 471	10 471	20,22	10 546	11 002	5,35
8	af_shell10	23 370	23 996	88,38	22 100	22 192,5	450,28	21 500	21 870,5	893,20	21 440	21 694	527,40	20 655	21 340,5	973,74	21 595	21 595	64,59	21 545	22 617,5	30,28
8	af_shell9	9 690	9 847,5	12,87	8 995	9 196,5	67,85	8 845	8 979,5	127,34	8 830	8 903	145,23	8 515	8 540,5	286,05	8 825	8 825	17,52	8 915	9 321	6,94

Table E.2.: Detailed results of the various hypergraph partitioners.

		LPFast			LPEco			LPBest			hMetis-k			hMetis-RB			PaToH-Q			PaToH-D		
k	Hypergraph	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
8	audikw_1	82 548	84 649,2	112,91	77 712	78 523,5	948,86	76 767	77 647,8	6 317,91	77 688	78 567,6	741,06	80 445	81 373,5	2 433,03	79 962	79 962	280,54	80 547	82 917	106,80
8	ecology1	7 611	7 847,3	45,02	7 228	7 562,9	246,34	7 057	7 333,7	520,66	6 994	7 069,2	165,81	6 668	7 042,8	324,69	7 372	7 372	10,67	7 346	7 654,5	2,49
8	ecology2	7 639	7 912,6	45,27	7 172	7 547,2	245,22	7 214	7 549,4	532,64	7 040	7 102,3	153,97	6 717	7 249,3	324,76	7 671	7 671	8,78	7 090	7 610	2,50
8	kkt_power	36 541	50 151,2	196,59	35 544	41 052,5	1 031,97	35 733	41 691,8	2 221,08	33 947	35 911,9	436,46	33 212	33 944,1	1 064,56	38 654	38 654	99,68	35 970	39 119,2	21,40
8	ldoor	13 321	13 689,9	41,16	12 439	12 716,2	208,96	11 816	12 189,1	451,14	11 039	11 777,5	384,03	11 921	12 149,2	903,98	12 222	12 222	79,74	12 607	13 221,6	36,66
8	nlpkkt120	232 762	245 806,8	868,95	216 473	220 911,2	4 137,79	203 959	210 088,4	8 992,77	210 257	214 035,5	5 720,41	172 792	172 947,4	11 642,52	174 576	174 576	348,70	184 414	195 314,1	83,48
8	thermal2	7 789	8 270,2	71,50	7 269	7 385,7	388,47	7 068	7 216,9	900,86	7 046	7 091,9	261,67	6 799	6 822,4	671,76	7 037	7 037	25,09	7 080	7 302,3	5,80
8	bcsstk29	2 410	2 486,4	0,54	2 310	2 349	2,58	2 334	2 359	3,70	2 298	2 351,4	4,68	2 220	2 237,4	11,11	2 428	2 428	1,11	2 469	2 536,3	0,33
8	bcsstk30	3 515	3 708,8	1,15	3 202	3 397,7	5,53	3 247	3 385,9	9,84	3 206	3 323,6	9,92	3 264	3 336,1	27,85	3 394	3 394	4,72	3 402	3 672,7	1,59
8	bcsstk31	3 669	3 964,9	1,03	3 390	3 455,6	6,41	3 377	3 467,2	11,38	3 361	3 425	13,05	3 244	3 391,7	29,20	3 494	3 494	2,37	3 602	3 867,4	0,62
8	bcsstk32	4 025	4 335,7	1,07	3 836	3 937,7	5,91	3 689	3 920,5	8,16	3 752	3 979,8	14,75	3 792	4 089,4	40,24	4 125	4 125	2,77	3 804	4 276,6	1,08
8	finan512	588	618,1	1,75	585	595,1	4,15	585	588	8,59	589	590	9,95	584	585,1	27,23	592	592	1,53	591	621,1	0,41
8	memplus	6 044	6 134,6	0,38	6 037	6 236,5	1,78	5 821	6 055	5,45	5 023	5 057,5	3,22	5 061	5 118,6	4,12	5 101	5 101	0,35	5 132	5 230,8	0,08
8	vibrobox	5 497	5 757,6	0,95	4 876	5 103,5	4,77	4 903	4 978,2	12,75	5 932	6 018,6	13,48	4 867	4 913	13,32	4 897	4 897	0,84	5 027	5 248,1	0,22
16	ibm01	1 379	1 409,3	0,25	1 270	1 321,9	1,94	1 279	1 311,8	5,05	1 267	1 275,7	4,97	1 273	1 291,8	5,23	1 348	1 348	0,42	1 392	1 443,5	0,07
16	ibm02	3 602	3 686,3	0,58	3 465	3 499,2	6,24	3 409	3 462,5	17,47	3 410	3 448,6	14,26	3 452	3 470,4	13,31	3 398	3 398	0,84	3 448	3 549,2	0,14
16	ibm03	3 548	3 652,4	0,53	3 353	3 415,4	4,39	3 352	3 401,2	12,07	3 291	3 317,9	9,69	3 246	3 298,4	9,85	3 584	3 584	0,70	3 662	3 756,2	0,13
16	ibm04	4 157	4 252,6	0,54	3 967	4 033,7	3,30	3 880	3 935,7	15,59	3 882	3 946,3	10,33	4 031	4 058,1	12,31	4 035	4 035	0,92	4 217	4 346,5	0,15
16	ibm05	5 903	6 096,6	0,70	5 612	5 744,6	4,18	5 510	5 578	17,44	5 769	5 870,3	18,56	5 472	5 543,5	16,42	5 519	5 519	0,98	5 631	5 798,5	0,17
16	ibm06	3 428	3 507,2	0,70	3 246	3 355,8	4,76	3 248	3 289,1	18,90	3 262	3 336,9	13,54	3 370	3 379,8	14,11	3 421	3 421	1,00	3 472	3 624,9	0,18
16	ibm07	5 138	5 239,2	1,03	4 835	4 922,6	7,03	4 828	4 897,2	18,06	4 744	4 793,6	16,07	4 761	4 791,9	21,52	5 147	5 147	1,42	5 210	5 353	0,25
16	ibm08	4 950	5 102,5	1,24	4 772	4 812,6	8,16	4 690	4 746	21,53	4 687	4 832,9	19,69	4 888	4 954,3	26,81	5 090	5 090	1,72	5 111	5 260,1	0,31
16	ibm09	4 411	4 516,4	1,08	4 093	4 177,3	6,89	4 007	4 118,1	17,16	3 886	3 967,7	13,33	3 982	4 016,1	20,96	4 201	4 201	1,57	4 228	4 610,7	0,29
16	ibm10	6 889	7 134,2	1,78	6 264	6 604,4	11,78	6 185	6 456,3	30,46	5 884	6 024,1	22,86	6 151	6 275,1	35,83	6 667	6 667	2,31	6 597	6 867,6	0,43
16	ibm11	5 715	5 997,3	1,58	5 348	5 445,6	10,36	5 360	5 463	27,01	5 276	5 349,9	17,40	5 301	5 540	29,17	5 608	5 608	2,02	5 813	6 071,7	0,37
16	ibm12	9 058	9 547,6	1,93	8 587	8 823	12,56	8 389	8 828,2	32,60	8 256	8 336	23,97	8 374	8 489,7	43,83	9 098	9 098	2,37	9 426	9 687,8	0,48
16	ibm13	6 401	6 632,2	1,98	5 734	5 867,2	14,20	5 567	5 838,3	36,99	5 541	5 599,4	24,46	5 610	5 658,2	38,34	5 968	5 968	2,85	6 156	6 560,2	0,50
16	ibm14	8 993	9 586,1	3,86	8 596	8 812,7	23,26	8 426	8 704,4	59,39	8 360	8 460,7	40,37	8 451	8 531,8	74,65	9 050	9 050	4,53	9 186	9 529,7	0,88
16	ibm15	9 397	10 180,1	4,70	8 884	9 299	27,06	8 764	9 311,4	71,49	8 743	8 923,7	49,68	8 740	9 043,2	90,86	9 930	9 930	5,08	10 117	10 765,2	1,07
16	ibm16	11 984	12 538,4	5,48	11 425	11 716,8	31,94	11 368	11 479,8	81,93	10 783	10 975,9	54,81	10 859	11 007,3	108,34	11 379	11 379	5,94	11 724	12 324,1	1,28
16	ibm17	17 344	18 043,3	6,14	15 182	15 703,4	35,15	15 218	15 540,5	89,09	14 734	15 086,7	71,92	15 080	15 313	146,84	16 543	16 543	7,01	16 125	17 402,6	1,47
16	ibm18	9 785	10 134,2	5,90	8 880	9 057	35,61	8 744	8 940,8	91,10	8 907	9 094,8	65,77	9 027	9 175,2	126,69	9 576	9 576	5,94	9 796	10 285,8	1,38
16	avqlarge	678	710,2	0,69	612	623,1	3,12	593	608,7	11,07	623	639,4	2,90	602	624,4	3,63	688	688	0,69	722	757,8	0,07
16	avqsmall	678	719,4	0,62	606	626,2	2,73	595	615,4	11,16	624	643,1	2,83	599	620,3	3,28	637	637	0,68	693	752,6	0,06
16	golem3	3 768	3 805,1	1,74	3 668	3 699,3	10,36	3 651	3 682	25,71	3 677	3 697,8	10,43	3 657	3 678,1	19,73	3 732	3 732	1,84	3 805	3 822,2	0,39
16	industry2	1 122	1 180	0,24	1 012	1 047,5	1,63	1 027	1 063,2	4,44	985	999,8	4,31	952	969,5	4,46	1 068	1 068	0,35	1 061	1 119,8	0,09
16	industry3	2 505	2 553,6	0,34	2 350	2 391,9	2,62	2 314	2 354,7	7,04	2 292	2 356,5	5,76	2 279	2 320	6,37	2 405	2 405	0,48	2 396	2 522,7	0,10
16	G3_circuit	18 830	19 361	107,30	17 188	17 807,7	570,19	17 209	17 589,6	1 257,87	17 314	17 453,6	336,14	15 951	16 027,4	819,39	17 847	17 847	26,85	17 358	18 374,9	6,68
16	af_shell10	37 470	38 299	88,64	34 580	35 008	451,76	33 780	34 266	944,65	33 990	34 218,5	551,03	33 050	33 530	1 258,71	33 690	33 690	78,21	35 105	35 834	39,81
16	af_shell9	18 020	18 469,5	13,04	16 775	16 988,5	69,35	16 660	16 797	147,18	16 410	16 616	154,85	16 190	16 260,5	371,48	16 700	16 700	27,20	16 915	17 353,5	9,12
16	audikw_1	134 394	136 075,5	127,16	128 532	130 377	1 402,83	128 250	129 184,2	13 937,32	129 114	130 464,6	768,66	133 719	136 167,6	3 020,54	133 212	133 212	364,56	135 351	137 769,9	138,22
16	ecology1	11 753	11 969,9	45,28	11 460	11 558,9	246,70	11 252	11 415,1	545,22	11 283	11 365,9	173,90	10 559	10 955,2	397,33	11 182	11 182	13,91	11 748	11 914,1	3,14
16	ecology2	11 776	11 985,2	45,46	11 408	11 553	246,85	11 187	11 400,8	545,20	11 293	11 372,3	174,72	10 334	10 964,9	402,93	11 741	11 741	12,26	11 748	11 942,2	3,15
16	kkt_power	76 825	93 215,4	203,36	61 505	69 895	1 051,66	60 690	70 784	2 475,85	59 227	63 293,4	550,10	54 068	54 696,8	1 311,94	64 547	64 547	145,94	61 163	68 433,8	25,66
16	ldoor	22 834	23 963,8	41,35	20 979	21 696,5	210,07	20 811	21 084	475,48	20 944	21 203	386,94	20 587	21 132,3	1 095,73	22 001	22 001	88,92	22 344	23 104,9	48,16
16	nlpkkt120	359 489	375 222,8	881,47	329 150	340 738,1	4 178,32	324 474	328 624,9	9 130,27	332 833	336 909,4	5 918,25	286 134	286 609,6	14 444,24	288 506	288 506	420,81	303 593	311 787,2	106,30
16	thermal2	12 874	13 168,1	71,67	11 875	12 082,5	389,45	11 790	11 925,4	903,75	11 800	11 891	266,80	11 485	11 571,2	808,48	11 826	11 826	32,29	12 076	12 354,9	7,29
16	bcsstk29	3 900	3 985,2	0,56	3 654	3 754,1	2,52	3 706	3 756,7	5,07	3 828	3 898,4	7,87	3 726	3 774,6	15,42	3 870	3 870	1,41	3 920	4 003,1	0,42
16	bcsstk30	7 013	7 355	1,43	6 558	6 724,8	8,84	6 627	6 703,6	21,42	6 884	7 207,3	14,39	6 844	7 069,3	35,28	6 817	6 817	5,66	7 120	7 601,5	2,04
16	bcsstk31	6 480	6 592,1	1,14	5 812	5 950,4	6,89	5 880	5 979,5	17,10	5 875	5 961,4	18,30	5 832	5 922,8	38,89	6 171	6 171	2,64	6 402	6 702,5	0,82
16	bcsstk32	7 003	7 293	1,12	6 403	6 589,6	6,36	6 541	6 652,8	13,19	6 536	6 863,3	18,31	6 518	6 711,1	53,60	6 801	6 801	4,02	6 875	7 181,9	1,41
16	finan512	1 174	1 208,6	1,87	1 172	1 178,7	5,77	1 175	1 179,6	10,93	1 180	1 180,9	14,34	1 168	1 172,1	37,34	1 183	1 183	2,22	1 184	1 264,8	0,55
16	memplus	6 538	6 689,7	0,56	6 544	6 659,5	2,68	6 417	6 578	6,05	5 703	5 736,6	5,52	5 719	5 774,6	4,94	5 756	5 756	0,41	5 733	5 900	0,09

Table E.2.: Detailed results of the various hypergraph partitioners.

		LPFast			LPEco			LPBest			hMetis-k			hMetis-RB			PaToH-Q			PaToH-D		
k	Hypergraph	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
16	vibrobox	6 454	6 588,1	1,20	6 271	6 300,3	7,74	6 230	6 281,2	16,42	7 293	7 381,9	22,74	6 018	6 174,8	16,89	6 284	6 284	1,06	6 477	6 557,5	0,25
32	ibm01	1 800	1 838,7	0,35	1 702	1 719	4,22	1 672	1 699,3	11,92	1 733	1 752,4	9,02	1 715	1 732,1	6,95	1 803	1 803	0,54	1 814	1 893,6	0,09
32	ibm02	4 675	4 759,9	0,63	4 471	4 501,6	7,01	4 366	4 419,4	33,23	4 650	4 760,9	22,34	4 462	4 498,7	15,51	4 469	4 469	1,04	4 582	4 664	0,16
32	ibm03	4 256	4 370,4	0,57	4 084	4 122,8	4,98	4 097	4 115,2	13,51	4 194	4 213,3	15,78	4 080	4 134,8	12,15	4 254	4 254	0,97	4 369	4 460,3	0,15
32	ibm04	5 358	5 497,7	0,69	5 097	5 140,3	6,52	5 006	5 072	18,10	5 072	5 095,2	17,78	5 128	5 165,2	14,99	5 342	5 342	1,23	5 456	5 572	0,18
32	ibm05	6 529	6 603,5	0,79	6 130	6 211,4	7,12	5 992	6 181,1	19,11	6 703	6 814,7	27,63	6 255	6 347,3	19,11	6 637	6 637	1,15	6 265	6 443	0,20
32	ibm06	4 479	4 530,9	0,87	4 235	4 286,8	8,14	4 203	4 250	22,55	4 433	4 504,3	21,68	4 341	4 394,3	17,35	4 544	4 544	1,42	4 561	4 692	0,23
32	ibm07	6 549	6 723	1,08	6 227	6 263,5	11,89	6 117	6 210,7	32,94	6 359	6 409,8	25,78	6 378	6 416,2	25,56	6 488	6 488	1,70	6 754	6 871,7	0,31
32	ibm08	6 371	6 569,1	1,35	6 113	6 154,4	13,85	6 040	6 115,4	38,34	6 307	6 355,7	30,41	6 329	6 428	32,33	6 684	6 684	2,16	6 510	6 705,6	0,37
32	ibm09	5 894	6 048,7	1,31	5 504	5 564,4	11,74	5 506	5 543,2	31,97	5 447	5 496,7	22,85	5 503	5 564	26,98	5 803	5 803	2,06	6 124	6 189,1	0,36
32	ibm10	9 560	9 887,5	1,87	8 964	9 163	13,17	8 628	8 795,2	55,53	8 531	8 634,5	35,36	8 638	8 799,5	44,20	8 988	8 988	2,98	9 197	9 479,4	0,53
32	ibm11	8 249	8 493	1,65	7 617	7 753	11,51	7 465	7 587,6	46,33	7 536	7 595,2	28,34	7 600	7 634,9	35,87	7 841	7 841	2,59	8 179	8 363,6	0,46
32	ibm12	11 929	12 460	2,03	11 122	11 376,3	13,95	10 820	11 010,7	58,42	10 820	10 939,3	37,38	11 049	11 155,1	51,46	11 676	11 676	3,14	11 956	12 331,6	0,58
32	ibm13	8 313	8 701,9	2,26	7 874	8 089,2	15,71	7 815	8 060,4	40,46	7 696	7 848	38,06	7 743	7 903,8	48,26	8 578	8 578	3,26	8 406	8 820,5	0,61
32	ibm14	14 583	14 971,6	4,03	13 180	13 318,6	31,91	13 086	13 234,5	84,40	12 735	12 878,2	56,96	12 759	12 890,7	89,16	13 666	13 666	5,55	13 805	14 253,2	1,07
32	ibm15	15 162	15 665,3	5,23	13 969	14 212,3	35,46	13 675	14 042,8	93,50	13 477	13 729,1	69,06	13 441	13 604,1	110,93	14 927	14 927	6,38	15 069	15 671,5	1,28
32	ibm16	16 971	17 728,1	5,75	15 711	16 177,6	41,84	15 533	15 997,6	110,00	15 493	15 638,1	73,34	15 562	15 777,9	128,19	16 464	16 464	7,37	17 030	17 371,5	1,52
32	ibm17	22 593	23 057,4	6,64	20 263	20 658,9	46,20	19 918	20 311,5	122,12	19 776	19 936,2	94,44	20 105	20 303,1	166,76	20 886	20 886	8,62	21 811	23 072,8	1,75
32	ibm18	14 223	14 818,3	6,21	13 385	13 711,5	38,84	13 367	13 449,6	124,72	13 453	13 651,3	88,16	13 479	13 853,2	151,33	14 060	14 060	7,48	14 567	14 946,9	1,64
32	avqlarge	912	938,3	1,00	841	850,5	4,98	831	847	12,25	924	937,6	5,82	846	867,5	4,94	966	966	0,79	1 016	1 033,4	0,09
32	avqsmall	932	960,6	0,95	847	855,1	4,68	840	852,9	13,81	939	951,5	5,48	849	869,3	4,49	913	913	0,74	979	1 041,9	0,08
32	golem3	4 639	4 682,8	1,84	4 482	4 534,2	11,77	4 477	4 506,2	28,74	4 542	4 581,6	14,06	4 455	4 494,5	23,66	4 600	4 600	2,08	4 697	4 742,5	0,48
32	industry2	1 365	1 423,5	0,31	1 298	1 324,4	3,30	1 282	1 306,2	9,33	1 401	1 442,1	7,73	1 276	1 314,2	5,60	1 355	1 355	0,40	1 403	1 436,7	0,10
32	industry3	3 213	3 389,8	0,37	3 028	3 055,8	5,20	3 005	3 049,7	14,67	3 075	3 123,9	10,62	3 052	3 076,7	8,23	3 133	3 133	0,62	3 183	3 323,1	0,13
32	G3_circuit	26 812	27 891,4	107,90	25 695	26 528,1	573,57	24 957	25 407,3	1 266,65	25 584	26 043,4	347,24	23 917	24 199	957,99	25 967	25 967	37,99	26 286	27 726,2	7,96
32	af_shell10	57 665	58 527,5	89,36	54 405	54 844	454,47	53 230	53 698,5	949,77	52 340	53 208	560,08	51 995	52 919	1 512,51	53 340	53 340	98,12	54 020	55 063,5	49,17
32	af_shell9	29 810	30 166	13,15	27 580	27 859	69,47	27 100	27 500	158,59	26 850	27 113,5	164,40	26 800	26 957,5	452,62	27 250	27 250	32,65	27 800	28 225,5	11,28
32	audikw_1	198 723	201 496,2	149,44	190 776	193 411,5	1 932,98	187 203	189 367,2	23 068,90	190 656	192 657,6	819,98	200 793	202 715,1	3 517,65	196 476	196 476	442,18	200 628	203 876,4	167,05
32	ecology1	18 394	18 671,6	45,38	17 650	18 050	249,11	17 607	17 898,9	550,53	17 365	17 500,8	185,62	15 605	16 779,4	460,38	17 960	17 960	17,36	17 958	18 433,1	3,79
32	ecology2	18 446	18 706,1	45,59	18 039	18 259,9	248,89	17 574	17 880	550,89	17 099	17 404,1	183,80	15 831	16 546,1	473,67	18 366	18 366	17,10	18 017	18 382,6	3,81
32	kkt_power	167 253	179 554,3	211,10	123 626	131 495,5	1 087,20	95 772	99 721,1	2 533,78	106 753	111 711,5	619,80	86 610	87 692,9	1 545,61	103 367	103 367	159,05	107 817	111 476,4	29,37
32	ldoor	37 800	39 208,4	41,75	35 819	36 388,1	211,70	35 511	35 973	478,16	34 797	35 707,7	393,46	35 595	35 980,7	1 276,01	36 862	36 862	111,35	37 198	38 202,5	59,42
32	nlpkkt120	505 421	515 469	896,67	466 449	471 456,7	4 244,98	450 516	458 640,2	9 248,88	458 401	465 345,2	5 918,25	396 606	397 339,2	16 197,80	403 292	403 292	524,99	421 811	433 883,7	127,39
32	thermal2	20 961	21 278,6	71,93	19 697	19 820	391,16	19 465	19 586,1	909,24	19 436	19 537,5	273,79	19 116	19 195,3	918,85	19 559	19 559	38,47	20 014	20 253	8,70
32	bcsstk29	5 340	5 498,5	0,65	5 262	5 408,8	3,38	5 313	5 404,9	6,17	5 427	5 520,6	14,21	5 296	5 393,8	19,68	5 525	5 525	1,69	5 668	5 808,1	0,50
32	bcsstk30	11 066	11 433,3	1,66	10 249	10 387	13,68	10 149	10 300	37,13	11 020	11 370,8	27,42	10 957	11 161,9	42,62	10 942	10 942	6,81	11 001	11 415,9	2,43
32	bcsstk31	9 678	10 018,7	1,39	9 202	9 342,9	12,99	9 096	9 216,3	31,23	9 370	9 701,3	29,68	9 081	9 299,1	48,96	9 382	9 382	3,87	9 681	10 070,8	1,00
32	bcsstk32	10 676	10 918,2	1,30	10 032	10 232,3	10,11	10 082	10 219,7	21,10	10 562	10 848,5	27,56	10 207	10 504,7	67,01	10 551	10 551	4,71	10 703	11 113	1,72
32	finan512	2 352	2 402,1	1,99	2 348	2 351,2	13,72	2 347	2 352,1	34,87	2 357	2 360,7	29,79	2 336	2 342,4	50,54	2 366	2 366	2,95	2 366	2 366,8	0,72
32	memplus	6 908	7 098	0,55	7 022	7 158,9	4,32	6 977	7 117,5	10,88	6 389	6 419,5	12,38	6 273	6 290,2	5,87	6 359	6 359	0,45	6 400	6 475,5	0,09
32	vibrobox	7 392	7 516	1,44	7 157	7 223,9	8,76	7 121	7 178,6	19,39	8 440	8 492,9	38,58	6 905	6 989,5	20,24	7 172	7 172	1,26	7 242	7 342,9	0,28
64	ibm01	2 351	2 387,3	0,38	2 240	2 259,3	5,18	2 224	2 235,6	14,84	2 359	2 375,1	14,77	2 289	2 295	8,90	2 388	2 388	0,60	2 412	2 455,2	0,11
64	ibm02	5 401	5 438,1	0,78	5 184	5 248,6	12,73	5 213	5 233	37,30	5 879	5 911,5	35,45	5 310	5 337,6	17,61	5 344	5 344	1,13	5 364	5 449,7	0,19
64	ibm03	4 966	5 038,5	0,74	4 791	4 828,3	10,05	4 765	4 793,7	28,98	5 121	5 151,5	26,28	4 891	4 923	14,83	4 973	4 973	1,08	5 118	5 191,7	0,18
64	ibm04	6 418	6 491,8	0,92	6 133	6 169,2	12,90	6 087	6 126,8	36,99	6 445	6 507,2	28,23	6 329	6 387,2	17,98	6 496	6 496	1,40	6 652	6 744	0,22
64	ibm05	6 940	7 067,6	0,85	6 525	6 617,3	13,84	6 540	6 568,7	39,63	7 646	7 670,7	42,89	6 887	7 048,5	21,89	6 895	6 895	1,18	6 914	7 018,9	0,22
64	ibm06	5 410	5 458,9	0,92	5 186	5 221,6	14,95	5 147	5 186,4	43,50	5 720	5 759,4	37,38	5 317	5 359,6	21,27	5 487	5 487	1,38	5 648	5 706	0,26
64	ibm07	8 062	8 195,7	1,32	7 636	7 739,3	13,63	7 605	7 691,5	37,56	8 056	8 136,3	41,33	7 940	7 997,9	30,78	8 196	8 196	2,03	8 326	8 396,9	0,36
64	ibm08	8 094	8 218,3	1,61	7 837	7 902,5	15,71	7 875	7 906,9	43,22	8 301	8 333,3	48,88	8 259	8 330,4	36,68	8 320	8 320	2,39	8 480	8 566,7	0,43
64	ibm09	7 830	8 042,7	1,42	7 414	7 504,6	13,86	7 380	7 499,4	38,11	7 483	7 520,9	39,77	7 445	7 527,5	32,99	7 864	7 864	2,27	8 158	8 327	0,43
64	ibm10	12 691	12 827,9	2,33	11 795	11 933,9	23,49	11 710	11 784,5	65,36	11 837	11 918,8	56,46	11 781	11 883,5	51,55	12 303	12 303	3,31	12 559	12 709,9	0,62
64	ibm11	10 592	10 723,4	2,03	9 934	10 007,8	19,77	9 813	9 914,6	54,50	9 907	10 004,1	46,51	9 809	10 013,9	43,55	10 284	10 284	3,16	10 778	10 938,5	0,54
64	ibm12	15 561	15 928,3	2,49	14 795	14 995,4	25,09	14 718	14 821,2	68,41	14 525	14 635,8	60,06	14 551	14 755,3	59,03	15 335	15 335	3,74	15 979	16 210	0,67

Table E.2.: Detailed results of the various hypergraph partitioners.

		LPFast			LPEco			LPBest			hMetis-k			hMetis-RB			PaToH-Q			PaToH-D		
k	Hypergraph	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
64	ibm13	12 881	13 071,1	2,72	12 183	12 327,9	27,14	12 098	12 215,9	75,15	12 159	12 243,5	61,83	12 161	12 254,3	57,84	12 892	12 892	3,94	13 201	13 380,5	0,72
64	ibm14	19 165	19 495,2	4,60	17 934	18 164,5	35,45	17 816	18 024	93,49	17 823	17 917,1	87,22	17 781	17 857,7	104,04	18 573	18 573	6,64	19 111	19 332,5	1,23
64	ibm15	20 854	21 057,9	5,49	19 125	19 505,4	38,88	19 002	19 268,2	100,80	18 949	19 101,7	102,89	18 768	18 994,7	128,36	19 793	19 793	7,75	20 327	21 040	1,51
64	ibm16	22 999	23 504,3	6,38	21 377	21 619,2	46,11	21 011	21 356,1	119,53	20 969	21 124,2	106,24	20 939	21 193,5	147,66	21 937	21 937	8,59	22 734	23 133	1,75
64	ibm17	28 768	29 291,9	7,26	26 887	27 165,7	50,66	26 817	26 970,6	131,99	26 051	26 237,3	134,78	26 505	26 792,6	185,28	29 612	29 612	10,14	29 665	30 102,9	1,99
64	ibm18	19 976	20 263,6	6,91	18 706	18 925,6	52,32	18 489	18 854,3	136,94	18 656	18 822,1	126,46	18 790	19 058,1	171,99	19 571	19 571	8,91	20 018	20 491,1	1,87
64	avqlarge	1 214	1 232,9	1,49	1 142	1 156,6	8,96	1 132	1 144,1	24,02	1 330	1 344,2	11,21	1 170	1 176	6,77	1 267	1 267	0,89	1 314	1 342	0,11
64	avqsmall	1 215	1 227,9	1,45	1 156	1 167,5	8,40	1 140	1 154	24,15	1 329	1 340,3	10,90	1 178	1 190,7	6,24	1 243	1 243	0,85	1 304	1 341,4	0,11
64	golem3	5 950	5 989	1,91	5 751	5 768,6	12,66	5 715	5 748,9	41,20	5 777	5 835,6	21,92	5 673	5 714,5	29,22	5 838	5 838	2,47	5 986	6 029,3	0,58
64	industry2	1 733	1 766	0,36	1 672	1 700,4	3,97	1 660	1 680,3	11,42	1 822	1 839,1	12,21	1 701	1 722	7,08	1 778	1 778	0,51	1 788	1 823,6	0,11
64	industry3	4 215	4 302,7	0,52	3 986	4 010,4	6,47	3 971	3 988,3	18,50	4 204	4 223,1	18,18	4 053	4 075,6	10,51	4 178	4 178	0,72	4 237	4 329,9	0,15
64	G3_circuit	41 885	42 587,7	109,11	39 761	40 684,2	579,18	39 105	39 951,9	1 282,35	40 243	40 573	372,30	37 935	38 199,1	1 050,94	41 219	41 219	38,08	40 717	42 387,1	9,24
64	af_shell10	84 240	85 429,5	90,12	79 105	79 689,5	461,83	78 295	78 560	963,35	78 030	78 754,5	571,85	76 905	77 646	1 751,95	78 110	78 110	118,82	81 065	81 562	58,40
64	af_shell9	45 415	45 762	13,65	42 200	42 832,5	73,60	41 765	42 270,5	165,74	41 850	42 005,5	176,20	41 560	41 805	531,32	42 580	42 580	36,31	43 355	43 647,5	13,42
64	audikw_1	279 999	283 434	180,40	269 121	270 977,1	2 938,52	265 482	265 963,5	45 301,95	270 621	273 032,7	940,62	282 318	284 542,8	3 974,58	272 472	272 472	482,41	283 095	287 144,1	192,87
64	ecology1	27 056	27 407,4	45,91	25 967	26 182,1	255,00	25 789	25 917,1	564,65	25 315	25 594,8	197,76	23 216	23 875,5	529,98	25 247	25 247	20,00	26 517	27 073,4	4,47
64	ecology2	27 139	27 480,6	45,91	26 116	26 294,5	251,75	25 746	25 838,3	564,53	25 277	25 609,2	199,53	23 349	23 902,6	537,28	25 951	25 951	20,23	26 460	26 950,5	4,48
64	kkt_power	239 253	257 409,2	217,19	162 022	167 248,8	1 106,68	160 131	165 885,8	2 651,58	163 635	172 896,8	708,97	133 652	136 666,4	1 727,55	153 857	153 857	182,53	164 257	170 326,1	32,32
64	ldoor	58 807	60 022,6	42,44	55 937	56 369,6	215,83	54 684	55 269,2	497,57	55 209	55 837,6	411,61	55 461	56 001,4	1 484,94	57 050	57 050	129,05	57 855	58 896,6	70,43
64	nlpkkt120	649 470	654 203,1	912,59	594 773	600 479,5	4 302,61	577 788	583 528,5	9 365,74	594 092	599 176,4	6 210,84	503 760	504 327,9	18 099,16	519 271	519 271	579,15	543 153	562 896,9	147,18
64	thermal2	31 071	31 388,3	72,38	29 332	29 636,1	394,39	29 048	29 117,2	939,54	29 036	29 289,5	287,47	28 494	28 591,2	1 028,03	29 339	29 339	43,19	30 298	30 678,7	10,10
64	bcsstk29	7 459	7 597,4	0,69	7 169	7 340,7	4,64	7 355	7 390,2	8,97	7 254	7 314,3	29,18	7 115	7 242	23,36	7 731	7 731	1,83	7 758	7 890,2	0,56
64	bcsstk30	15 094	15 335,6	2,14	14 488	14 690,7	19,47	14 305	14 452,5	55,46	15 927	16 072,5	49,40	15 219	15 488	49,36	14 859	14 859	7,29	15 461	15 832,5	2,70
64	bcsstk31	13 882	14 482,5	1,46	13 260	13 473,2	15,91	13 234	13 326,3	41,97	13 889	14 021,9	55,40	13 101	13 325,2	59,77	13 846	13 846	4,57	14 088	14 578,7	1,17
64	bcsstk32	15 800	16 123,1	1,36	14 511	14 780,3	12,30	14 558	14 753,2	29,30	15 521	15 812,9	44,46	14 940	15 135,5	77,41	15 273	15 273	6,02	15 809	16 073,9	2,00
64	finan512	9 709	10 091,8	2,53	9 169	9 207,8	24,66	9 116	9 183,7	69,62	9 063	9 077	61,78	8 915	8 946,5	67,39	9 142	9 142	4,04	9 605	10 188,2	0,92
64	memplus	7 411	7 507,6	0,60	7 493	7 591,2	4,56	7 464	7 617,8	11,62	6 962	7 020,2	38,08	6 701	6 782,5	7,08	6 914	6 914	0,50	6 880	6 940	0,11
64	vibrobox	8 179	8 249,5	1,64	7 959	8 012,4	9,33	7 923	7 990,7	22,38	9 635	9 683,8	59,93	7 831	7 878,7	23,64	7 971	7 971	1,25	8 023	8 135,4	0,31
128	ibm01	2 996	3 021,3	0,59	2 870	2 892,4	6,43	2 853	2 879,1	18,72	3 100	3 113,6	20,21	2 955	2 972,3	11,42	2 973	2 973	0,74	3 087	3 113,9	0,13
128	ibm02	6 030	6 051,9	0,85	5 881	5 910,9	14,28	5 854	5 894,7	41,34	6 771	6 788,2	42,91	6 065	6 111,4	20,17	6 027	6 027	1,24	6 122	6 173,4	0,21
128	ibm03	5 877	5 939,2	0,81	5 668	5 687,5	12,12	5 632	5 672	34,91	6 204	6 291,5	36,66	5 801	5 846,3	17,85	5 951	5 951	1,24	5 990	6 074,4	0,21
128	ibm04	7 576	7 653,6	1,00	7 248	7 283,9	15,30	7 201	7 227,6	44,11	7 916	7 971	40,62	7 564	7 639,6	22,02	7 683	7 683	1,57	7 811	7 946,3	0,25
128	ibm05	7 182	7 256,2	1,00	7 019	7 098,2	15,87	7 017	7 043,7	45,72	8 691	8 747,6	57,79	7 479	7 580,8	24,36	7 300	7 300	1,32	7 418	7 525,3	0,25
128	ibm06	6 445	6 497,6	1,25	6 263	6 287,1	17,96	6 203	6 241,9	53,12	7 206	7 232,4	53,47	6 456	6 502,5	24,91	6 613	6 613	1,65	6 702	6 768,3	0,30
128	ibm07	9 564	9 665,8	1,71	9 236	9 299,6	24,90	9 157	9 256,6	72,43	10 126	10 201,9	60,66	9 570	9 636,4	35,39	9 826	9 826	2,42	9 914	10 024,1	0,41
128	ibm08	9 764	9 829,1	1,99	9 542	9 597,8	27,53	9 431	9 504,5	79,25	10 250	10 363	68,22	9 971	10 039,3	41,49	10 035	10 035	2,64	10 092	10 232,6	0,48
128	ibm09	10 067	10 178,8	1,91	9 562	9 595,1	27,25	9 492	9 553,9	80,06	9 955	10 007,3	59,20	9 843	9 923,8	39,40	10 006	10 006	2,79	10 348	10 513,1	0,49
128	ibm10	15 719	15 937,1	2,48	14 900	14 998,2	26,98	14 430	14 580,3	120,25	15 234	15 294,6	85,60	15 000	15 061,3	59,78	15 254	15 254	3,97	15 744	15 867,1	0,71
128	ibm11	13 818	14 025,5	2,21	12 938	12 998,2	34,68	12 850	12 948,6	100,10	13 372	13 441,3	71,95	13 131	13 230,4	50,79	13 478	13 478	3,65	13 884	14 197,8	0,63
128	ibm12	19 486	19 667,2	2,69	18 685	18 925,4	28,20	18 435	18 524	131,18	19 270	19 358,4	93,87	18 676	18 832,8	70,08	19 162	19 162	4,34	19 639	19 888,5	0,76
128	ibm13	16 403	16 661,7	2,91	15 622	15 753,6	31,37	15 394	15 506,4	131,64	16 248	16 326,3	90,36	15 707	15 822	66,48	16 486	16 486	4,78	16 835	17 051,3	0,82
128	ibm14	23 969	24 307,3	5,43	22 559	22 851,9	56,06	22 431	22 682,2	157,41	23 243	23 412,9	125,75	22 990	23 141	117,75	23 647	23 647	7,61	24 331	24 605,2	1,39
128	ibm15	27 330	27 727,6	6,51	25 512	25 618,4	63,46	25 270	25 480,3	177,05	25 574	25 715	147,45	25 192	25 305,9	149,80	26 720	26 720	8,96	27 554	27 858,8	1,72
128	ibm16	28 867	29 088,1	7,47	26 923	27 025,7	71,57	26 728	26 905,3	198,67	27 602	27 784	152,48	27 223	27 444	167,45	28 115	28 115	9,72	28 983	29 347,1	1,97
128	ibm17	37 590	38 030,3	8,46	35 226	35 482,5	83,99	35 035	35 218,7	235,25	35 531	35 617,8	192,30	35 274	35 571,8	210,12	37 172	37 172	11,39	38 086	38 546,3	2,22
128	ibm18	26 249	26 419,1	7,45	24 146	24 531,6	80,03	23 908	24 362,3	220,78	25 285	25 391,5	176,25	24 773	24 911,4	191,64	25 584	25 584	10,17	25 887	26 294,4	2,10
128	avqlarge	1 573	1 591,6	2,20	1 556	1 585,9	14,72	1 563	1 573,1	34,68	1 769	1 776,9	19,63	1 640	1 657,3	9,57	1 733	1 733	1,01	1 767	1 793,3	0,14
128	avqsmall	1 544	1 581,8	2,16	1 549	1 561,6	13,63	1 534	1 543,8	35,11	1 722	1 745,2	18,04	1 618	1 627,1	8,74	1 680	1 680	1,08	1 687	1 724,9	0,13
128	golem3	7 614	7 669,7	2,22	7 359	7 389	18,57	7 286	7 331,5	51,53	7 501	7 548,1	34,33	7 279	7 320,7	35,40	7 514	7 514	3,11	7 775	7 837,7	0,71
128	industry2	2 035	2 069,4	0,47	1 942	1 978	4,95	1 939	1 959,6	14,20	2 315	2 340,5	16,74	2 032	2 076	9,08	2 057	2 057	0,59	2 117	2 145,3	0,13
128	industry3	5 309	5 350,7	0,80	5 040	5 090,5	8,04	5 012	5 055,7	23,04	5 269	5 294,4	24,44	5 066	5 090,5	13,58	5 253	5 253	0,94	5 339	5 406,1	0,18
128	G3_circuit	68 075	68 693	111,20	65 205	65 770,2	597,77	64 618	64 956,1	1 335,13	65 227	65 704,9	421,42	63 095	63 867,4	1 193,03	66 471	66 471	52,01	67 727	68 680,7	10,64
128	af_shell10	123 985	124 945	91,28	116 215	117 071,5	469,05	114 685	115 262	1 000,78	112 770	114 015	606,54	114 345	114 926,5	1 982,89	115 410	115 410	138,47	118 340	118 801,5	67,56

Table E.2.: Detailed results of the various hypergraph partitioners.

		LPFast			LPEco			LPBest			hMetis-k			hMetis-RB			PaToH-Q			PaToH-D		
k	Hypergraph	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
128	af_shell9	66 365	66 978	14,03	62 720	63 148,5	81,47	61 915	62 190,5	193,42	61 405	61 782,5	206,49	62 220	62 553,5	607,81	62 765	62 765	42,48	63 985	64 528	15,56
128	audikw_1	383 058	385 398	232,69	368 523	370 497,3	4 617,00	365 031	365 223	75 958,25	374 292	377 195,1	1 164,06	385 416	387 915	4 324,49	375 576	375 576	568,18	388 149	389 672,4	215,12
128	ecology1	39 333	39 764,2	46,45	37 991	38 324,1	260,00	37 225	37 481,9	581,86	36 548	36 921,3	224,26	33 163	34 163,7	594,17	37 054	37 054	23,18	38 747	39 317,4	5,19
128	ecology2	39 557	39 851,5	46,60	37 964	38 236,3	259,96	37 283	37 535,4	581,82	36 351	36 938,5	225,02	33 510	34 196,3	593,44	37 974	37 974	21,99	38 037	38 970,9	5,21
128	kkt_power	395 744	406 125,4	222,61	215 979	219 527,6	1 148,15	215 514	219 559,2	2 730,85	235 792	241 952,6	847,12	201 856	204 049,2	1 884,65	223 054	223 054	211,62	240 786	245 974,2	34,89
128	ldoor	89 194	90 356	43,12	83 755	84 236,6	225,67	82 418	82 989,9	529,62	84 168	84 784,7	441,99	83 860	84 725,2	1 629,32	86 359	86 359	158,38	87 087	88 189,5	81,19
128	nlpkkt120	866 898	876 305,9	935,67	808 394	812 933,4	4 414,33	773 812	777 508,6	9 849,49	795 836	801 331,9	6 702,57	713 933	714 740,7	19 430,12	721 912	721 912	649,47	748 049	757 780,6	166,38
128	thermal2	46 460	46 650,8	73,19	44 086	44 282,4	406,71	43 639	43 888	956,65	43 546	43 780,7	314,43	43 097	43 219,2	1 091,11	44 048	44 048	52,66	45 139	45 511,4	11,51
128	bcsstk29	10 094	10 215,4	0,82	9 979	10 048,7	5,72	9 953	10 076,8	12,99	9 979	10 088,8	38,06	10 764	10 847,6	25,66	9 865	9 865	2,06	10 004	10 058,6	0,61
128	bcsstk30	20 277	20 506,7	2,51	19 503	19 708,4	25,27	19 113	19 276,6	68,51	21 288	21 493,1	74,78	20 657	20 921	53,13	19 633	19 633	8,11	20 146	20 339	2,90
128	bcsstk31	19 045	19 205,5	2,12	18 418	18 565,7	19,16	18 167	18 367,3	53,12	19 180	19 311,3	92,71	18 150	18 260,6	69,10	18 828	18 828	5,41	19 438	19 548,3	1,33
128	bcsstk32	21 582	21 701,9	1,91	20 345	20 582	15,27	20 390	20 658,8	37,01	21 788	21 972,9	76,66	20 343	20 853,1	91,15	21 315	21 315	7,41	22 008	22 326,3	2,27
128	finan512	21 415	21 736,5	3,59	18 230	18 374,2	49,80	18 168	18 301	143,97	18 136	18 148,3	110,18	17 792	17 921,2	86,43	19 319	19 319	5,13	20 962	21 736,5	1,10
128	memplus	7 799	7 833,3	0,75	7 773	7 818,3	6,13	7 792	7 827,6	14,72	7 547	7 570,1	68,98	7 317	7 364,1	8,72	7 288	7 288	0,56	7 288	7 333,3	0,12
128	vibrobox	9 051	9 145,9	2,06	8 932	8 971,7	11,55	8 880	8 925,9	28,38	10 344	10 389,5	79,75	8 722	8 793,8	27,26	9 011	9 011	1,59	8 957	9 054,8	0,33

	LPFast			LPEco			LPBest			hMetis-k			Hmetis-RB			PaToH-Q			PaToH		
k	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
2	1293.23	1428.75	2.73	1223.84	1286.66	12.47	1217.98	1272.09	22.35	1195.01	1215.84	14.79	1171.85	1188.72	17.09	1248.28	1248.28	1.23	1228.67	1324.22	0.29
4	2774.10	3039.49	2.85	2605.61	2736.83	14.60	2576.76	2706.19	31.27	2514.55	2577.15	19.02	2526.13	2576.26	31.93	2731.73	2731.73	2.20	2668.52	2904.13	0.50
8	4810.66	5102.52	3.02	4464.57	4640.02	16.73	4428.51	4583.05	42.08	4341.92	4455.04	24.95	4325.87	4411.86	45.33	4641.06	4641.06	3.16	4657.01	4946.72	0.70
16	7317.29	7592.07	3.38	6823.37	6998.15	20.26	6766.00	6928.05	53.95	6766.14	6885.27	34.42	6653.30	6768.89	58.27	7032.57	7032.57	3.99	7160.38	7462.80	0.88
32	10313.29	10603.99	3.77	9690.10	9844.74	27.50	9509.01	9659.07	75.57	9749.54	9894.00	50.37	9447.11	9594.02	70.93	9941.56	9941.56	4.96	10128.06	10408.10	1.07
64	14112.04	14358.19	4.30	13279.17	13436.25	36.06	13184.85	13313.25	98.04	13706.44	13832.35	76.00	13109.44	13251.74	84.18	13667.30	13667.30	5.66	14012.64	14282.06	1.25
128	18554.30	18752.95	5.10	17474.79	17614.94	47.25	17307.54	17440.95	134.83	18334.40	18469.49	105.10	17500.24	17658.21	98.24	17980.32	17980.32	6.67	18415.08	18653.36	1.43
avg	6171.60	6481.10	3.51	5791.09	5957.75	22.47	5735.03	5887.33	55.32	5782.92	5879.75	37.13	5653.77	5740.09	50.57	5975.10	5975.10	3.50	6017.61	6299.44	0.78

Table E.3.: The geometric mean over all instances for the different hypergraph partitioners. For each k and each hypergraph we thereby aggregated the 10 runs into one using the arithmetic mean.

	LPFast			LPEco			LPBest			hMetis-k			Hmetis-RB			PaToH-Q			PaToH		
k	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
2	880.19	996.57	0.91	830.74	875.82	4.39	825.76	863.61	9.27	805.44	818.27	5.34	795.00	807.23	6.25	857.24	857.24	0.49	842.03	925.21	0.10
4	1822.06	2010.90	0.96	1697.61	1787.85	5.21	1673.39	1761.27	12.82	1621.32	1669.64	7.25	1658.05	1694.81	11.63	1808.58	1808.58	0.88	1787.82	1971.91	0.16
8	2967.95	3151.12	1.04	2721.47	2837.64	6.14	2714.17	2809.38	17.08	2633.93	2709.41	10.22	2689.61	2734.88	16.45	2919.18	2919.18	1.24	2941.88	3142.10	0.22
16	4270.90	4430.79	1.20	3981.37	4081.31	7.75	3936.68	4036.35	22.29	3913.44	3982.33	15.36	3930.48	3995.17	21.35	4176.69	4176.69	1.55	4261.77	4456.03	0.28
32	5681.28	5850.93	1.38	5324.89	5402.97	11.23	5253.74	5339.78	33.10	5405.15	5474.79	24.34	5316.36	5390.77	26.27	5607.52	5607.52	1.92	5727.04	5898.00	0.34
64	7295.49	7406.63	1.64	6907.98	6982.90	16.23	6866.08	6929.25	45.23	7220.50	7273.33	39.31	6982.09	7051.27	31.64	7276.81	7276.81	2.20	7451.60	7575.75	0.40
128	9008.54	9108.99	2.01	8622.40	8693.29	22.91	8552.14	8619.49	68.44	9269.70	9328.75	57.01	8825.19	8895.39	37.83	9052.08	9052.08	2.58	9246.67	9368.79	0.46
avg	3583.64	3780.89	1.26	3361.16	3460.94	8.93	3331.49	3422.53	24.04	3363.46	3420.31	16.49	3332.54	3381.40	18.71	3541.13	3541.13	1.37	3573.94	3763.46	0.25

Table E.4.: The geometric mean over the VLSI instances for the different hypergraph partitioners. For each k and each hypergraph we thereby aggregated the 10 runs into one using the arithmetic mean.

	LPFast			LPEco			LPBest			hMetis-k			Hmetis-RB			PaToH-Q			PaToH		
k	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]	best cut	avg cut	avg t[s]
2	2176.48	2326.04	12.17	2067.14	2165.05	51.22	2060.61	2148.23	73.52	2037.90	2077.55	58.66	1980.83	2006.74	66.61	2075.51	2075.51	4.23	2048.63	2151.00	1.33
4	4899.06	5315.33	12.40	4652.17	4868.89	58.87	4620.80	4838.67	104.47	4553.23	4636.54	70.07	4465.32	4539.88	125.30	4772.54	4772.54	7.61	4587.87	4903.26	2.35
8	9246.60	9794.51	12.75	8722.29	9025.19	64.98	8588.53	8886.18	142.49	8538.36	8730.82	83.43	8228.11	8425.68	178.63	8690.43	8690.43	11.17	8669.51	9140.67	3.30
16	15160.37	15732.06	13.73	14142.98	14515.12	74.30	14078.19	14389.33	178.36	14192.07	14442.03	102.56	13561.49	13813.91	226.64	14231.84	14231.84	14.43	14448.24	14993.32	4.19
32	23106.99	23706.00	14.69	21782.95	22168.83	92.42	21219.90	21537.67	230.92	21655.85	22033.55	134.80	20563.98	20927.14	272.00	21572.95	21572.95	17.92	21903.28	22443.83	5.02
64	34455.22	35159.46	15.75	32148.62	32571.72	106.22	31875.23	32208.85	279.18	32623.19	33005.53	185.46	30743.05	31116.21	316.34	32065.58	32065.58	20.34	32930.00	33677.79	5.82
128	49315.59	49814.02	18.00	45443.53	45795.63	125.80	44921.31	45257.49	337.43	46132.73	46535.03	240.44	44187.60	44650.63	357.27	45503.00	45503.00	24.10	46768.95	47357.19	6.58
avg	12876.32	13437.23	14.09	12089.86	12422.99	78.26	11958.86	12264.01	170.83	12038.60	12237.62	111.29	11559.07	11745.03	194.12	12126.57	12126.57	12.42	12177.64	12646.65	3.63

Table E.5.: The geometric mean over the SPM instances for the different hypergraph partitioners. For each k and each hypergraph we thereby aggregated the 10 runs into one using the arithmetic mean.