High-Quality Hypergraph Partitioning

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Sebastian Schlag

aus Weinheim

Tag der mündlichen Prüfung: 11. Dezember 2019

Erster Gutachter:	Prof. Dr. Peter Sanders Karlsruher Institut für Technologie
Zweiter Gutachter:	Prof. Dr. Henning Meyerhenke Humboldt-Universität zu Berlin

To my parents, Klaus and Elisabeth Schlag

Abstract

This dissertation focuses on computing high-quality solutions for the NP-hard *balanced hypergraph partitioning problem*: Given a hypergraph and an integer k, partition its vertex set into k disjoint blocks of bounded size, while minimizing an objective function over the hyperedges. Here, we consider the two most commonly used objectives: the *cut-net* metric and the *connectivity* metric.

Since the problem is computationally intractable, heuristics are used in practice – the most prominent being the three-phase multi-level paradigm: During coarsening, the hypergraph is successively contracted to obtain a hierarchy of smaller instances. After applying an initial partitioning algorithm to the smallest hypergraph, contraction is undone and, at each level, refinement algorithms try to improve the current solution.

With this work, we give a brief overview of the field and present several algorithmic improvements to the multi-level paradigm. Instead of using a logarithmic number of levels like traditional algorithms, we present two coarsening algorithms that create a hierarchy of (nearly) n levels, where n is the number of vertices. This makes consecutive levels as similar as possible and provides many opportunities for refinement algorithms to improve the partition. This approach is made feasible in practice by tailoring all algorithms and data structures to the n-level paradigm, and developing lazy-evaluation techniques, caching mechanisms and early stopping criteria to speed up the partitioning process. Furthermore, we propose a sparsification algorithm based on locality-sensitive hashing that improves the running time for hypergraphs with large hyperedges, and show that incorporating global information about the community structure into the coarsening process improves quality. Moreover, we present a portfolio-based initial partitioning approach, and propose three refinement algorithms. Two are based on the Fiduccia-Mattheyses (FM) heuristic, but perform a highly localized search at each level. While one is designed for two-way partitioning, the other is the first FM-style algorithm that can be efficiently employed in the multi-level setting to directly improve k-way partitions. The third algorithm uses max-flow computations on pairs of blocks to refine k-way partitions. Finally, we present the first memetic multi-level hypergraph partitioning algorithm for an extensive exploration of the global solution space.

All contributions are made available through our open-source framework KaHyPar. In a comprehensive experimental study, we compare KaHyPar with hMETIS, PaToH, Mondriaan, Zoltan-AlgD, and HYPE on a wide range of hypergraphs from several application areas. Our results indicate that KaHyPar, already without the memetic component, computes better solutions than all competing algorithms for both the cut-net and the connectivity metric, while being faster than Zoltan-AlgD and equally fast as hMETIS. Moreover, KaHyPar compares favorably with the current best graph partitioning system KaFFPa – both in terms of solution quality and running time.

Acknowledgements

The last couple of years that have lead to this dissertation have been an incredible journey, both from a professional *and* a personal perspective. Hence, I would like to take this opportunity to – at least briefly – express my gratitude.

First and foremost, I am deeply thankful to my advisor Peter Sanders for granting me a position in his group, giving me just the right amount of guidance whenever I needed it, and otherwise allowing me to freely pursue my research interests. Moreover, I would like to thank Henning Meyerhenke for agreeing to co-review this work.

Furthermore, I would like to thank my co-authors Yaroslav Akhremtsev, Michael Axtmann, Timo Bingmann, Tobias Heuer, Henning Meyerhenke, Benoit Morel, Ingo Müller, Peter Sanders, Christian Schulz, Alexandros Stamatakis, and Darren Strash for all the fruitful collaborations.

I really enjoyed the interesting technical as well as non-technical discussions, and the incredibly stimulating atmosphere in our research group, which is why I want to thank all my current and former coworkers – you made this a wonderful place to work. Thank you, Yaroslav Akhremtsev, Julian Arz, Michael Axtmann, Tomáš Balyo, Veit Batz, Timo Bingmann, Johannes Fischer, Daniel Funke, Simon Gog, Demian Hespe, Tobias Heuer, Lorenz Hübschle-Schneider, Moritz Kobitzsch, Sebastian Lamm, Dennis Luxen, Tobias Maier, Vitaly Osipov, Dennis Schieferdecker, Dominik Schreiber, Christian Schulz, Nodari Sitchinava, Jochen Speck, Darren Strash, and Sascha Witt. Especially, I would like to thank Moritz Kobitzsch and Timo Bingmann for being supreme office mates.

Over the years, I had the pleasure to work with many bright students. Especially, I would like to thank Robin Andre, Ivo Baar, Vitali Henne, Tobias Heuer, Lukas Hübner, Peter Oettig, Matthias Schmitt, Daniel Seemaier, and Adrian Zapletal for their excellent cooperations. I am more than happy to see that Tobias Heuer, who accompanied my research projects from early on, has found his way back from industry to pursue a PhD degree and to continue working on hypergraph partitioning.

I thank Timo Bingmann, Daniel Funke, Tobias Heuer, Lorenz Hübschle-Schneider, and Tobias Maier, as well as Stephan Erb, Kirbanu Klein, Amanda Suchán, and Marlene Zahner for proof-reading parts of this work and for many helpful comments and suggestions.

Finally, I would like to thank my family and Marlene Zahner for their love and their everlasting emotional support in my endeavors, as well as my closest friends Mirko Ester, Marcel Fuhr, Jörn Heimann, Tobias Jäger, Phillip Klein, Mario Niebler, Alexander Scheller, Lena Schork, Raphaela Sedlmeier, and Andreas and Talea Wadewitz for keeping me sane and for tolerating all of the countless times I missed social gatherings due to work-related deadlines.

Table of Contents

Chapter 1

1 Introduction

"A good cook need sharpen his blade but once a year. He cuts cleanly. An awkward cook sharpens his knife every month. He chops. I've used this knife for nineteen years, carving thousands of oxen. Still the blade is as sharp as the first time it was lifted from the whetstone. At the joints there are spaces, and the blade has no thickness. Entering with no thickness where there is space, the blade may move freely where it will: there's plenty of room to move. Thus, after nineteen years, my knife remains as sharp as it was that first day.

"Even so, there are always difficult places, and when I see rough going ahead, my heart offers proper respect as I pause to look deeply into it. Then I work slowly, moving my blade with increasing subtlety until — kerplop! — meat falls apart like a crumbling clod of earth. I then raise my knife and assess my work until I'm fully satisfied. Then I give my knife a good cleaning and put it carefully away."

— Sam Hamill and J. P. Seaton, The Essential Chuang Tzu

This chapter sets the stage for the work presented in this dissertation. After motivating the hypergraph partitioning problem in Section 1.1, we briefly summarize the main contributions in Section 1.2. We then discuss the applied research methodology – algorithm engineering – in Section 1.3 and outline the remainder of this work in Section 1.4.

1.1 Motivation

Graph Partitioning. Graph partitioning – that is, the task to divide the vertices of a graph into a fixed number of disjoint blocks of bounded size, such that the number of edges crossing between blocks is minimized – is a classical and well-studied problem in computer science [BS11; Bul+16]. It is commonly applied as a preprocessing technique in settings where the goal is to divide a set of interrelated objects into a certain number of subsets such that objects in the same subset are in some sense strongly related and objects in different subsets are weakly related. In the graph-based representation, the vertices of the graph represent the objects, while edges model relationships, dependencies, or interactions between these objects. Just like Chuang Tzu's Taoist butcher carefully carves along the joints of the ox, partitioning algorithms

try to divide graphs at their "natural joints" such that as few edges as possible remain that connect vertices in different blocks, while ensuring that each block contains roughly the same number of vertices. Well-known applications of graph partitioning include load balancing in parallel computing [Mil+93; SKK00], data distribution for parallel graph analytics [AK06; SMR14; Wan+14; Slo+17], and route planning [Hil+06; Del+11; LS14; Del+17]. In the first two examples, the goal is to distribute units of computation or data evenly among the processors of a parallel computer while minimizing the amount of communication between processors that occurs if dependent computations or data elements are assigned to different processors. In route planning, graph partitioning is used to identify "natural joints" (or bottlenecks) such as highways, rivers or mountain ranges in road networks. This information is then exploited to improve the query performance of shortest-path algorithms.

Limitations of Graph Models. While graphs permit the modeling of pairwise interactions or dependencies, many real-world problems involve more complex relationships between objects [BS11, p.65]. Already in the 1960s and the early 1970s – when graph partitioning just started to become an active area of research – Rutman [Rut64] as well as Schweikert and Kernighan [SK72] observed that the standard graph model is inappropriate for the task of partitioning electrical circuits in the field of very-large-scale integration (VLSI) design. Here, the goal is to divide a circuit into two or more blocks such that the number of external wires interconnecting circuit elements in different blocks is minimized. Minimization of external wires is important, because it reduces signal delays, wiring cost, and the total layout area [YM90; AK95c]. Consider, for example, the circuit consisting of 5 elements connected by 3 wires shown in Figure 1.1 (a). By assigning circuit elements 2 and 3 to one block, and elements 1, 4, and 5 to the other block, we get a partition into two blocks that requires one external wire (b). In the traditional graph model, circuit elements correspond to vertices and a wire connecting multiple circuit elements is represented by adding an edge between every two distinct vertices (c). In this case, however, an optimal partition would put element 5 in one block and all other elements in the other block, since this is the only partition into two blocks that results in only two external edges. The difference between the circuit solution and the graph partition arises, because graphs are unable to concisely represent relationships between *more than two* objects.

Hypergraphs to the Rescue. Hypergraphs are a generalization of graphs, where each (hyper)edge can connect an arbitrary number of vertices. Thus, unlike graphs, which are restricted to dyadic relationships, hypergraphs can be used to model more complex dependencies and interactions. Figure 1.1 (d) depicts the hypergraph representation of the circuit shown in (a) along with the corresponding partition into two blocks. Due to the increased modeling flexibility, the hypergraph correctly represents the wiring of the circuit and thus a "good" partition of the hypergraph corresponds to a "good" partition of the physical circuit. Hypergraphs have since been shown to provide better models in various other application domains. The problem of storage sharding in distributed databases, for example, can be solved more efficiently using hypergraph-based approaches [Cur+10; QKD13; Kum+14; Kab+17]. Another

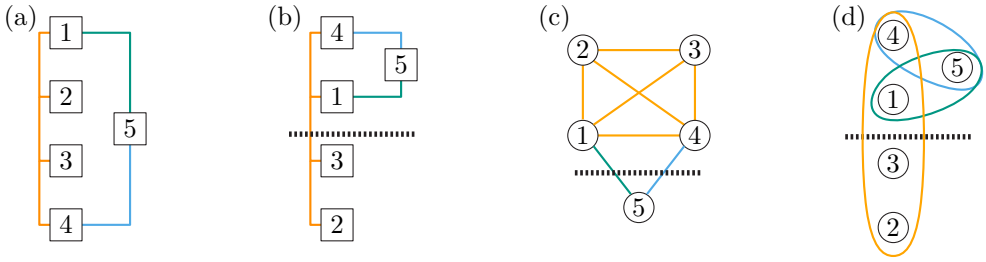


Figure 1.1: Example used by Schweikert and Kernighan [SK72] to demonstrate the shortcomings of the graph model for partitioning electrical circuits: (a) a circuit consisting of 5 elements and 3 electrical wires, (b) a partition of the circuit into two blocks with one wire connecting both blocks, (c) the best graph partition with two edges connecting both blocks, (d) the best hypergraph partition with one hyperedge connecting both blocks.

well-known problem is the parallelization of sparse matrix-vector multiplications in the field of parallel scientific computing [ÇA01a; ÇA01b; UA04; VB05; UA07; YB09; ÇAU10; UÇ10; Bis+12; KUÇ14; PB14; SAA17]. In this area, the benefits of hypergraph-based approaches include the correct modeling of communication volumes and the ability to handle unsymmetric data dependencies [Hen98; Çat99; HK00]. Especially for irregular problems, hypergraph models are regarded to yield better results than graph-based approaches [BS11, p.78]. Furthermore, recent interest in the analysis of group-wise interactions in social networks [Laz+09; Ama+18; Ant+19] has shifted the focus from traditional graph processing frameworks [Mal+10; Chi+15] towards distributed hypergraph processing systems [HC14; HC15; HZY15; Jia+18; Hei+19].

Hypergraph Partitioning. The hypergraph partitioning (HGP) problem is the generalization of the graph partitioning problem, i.e., the goal is to partition the vertex set of the hypergraph into a fixed number of k disjoint blocks of bounded size, while minimizing an objective function defined on the hyperedges. The two most widely-used objective functions are the *cut-net* metric and the *connectivity* metric. Cut-net is a straightforward generalization of the traditional edge-cut objective in graph partitioning (i.e., minimizing the number of hyperedges that cross more than one block). Since a hyperedge can connect more than two vertices, it is possible that it crosses up to k blocks in a partitioned hypergraph. Therefore, the connectivity metric additionally takes into account the actual number of blocks connected by a hyperedge and favors solutions in which hyperedges are split among as few blocks as possible.

The Challenge. Computing optimal solutions for graph and hypergraph partitioning problems is computationally intractable, since both problems are NP-hard [GJS76; Len90], and it is even NP-hard to find good approximate solutions for general graphs [BJ92]. Therefore, heuristic algorithms are used in practice. In both areas, the

most successful heuristic is the three-phased *multi-level* paradigm. In the *coarsening phase*, multi-level algorithms first successively contract the input (hyper)graph to obtain a hierarchy of smaller, structurally similar instances. After applying an *initial partitioning* algorithm to the smallest (hyper)graph, contraction is undone and, at each level, *refinement* algorithms are used to improve the partition induced by the coarser level.

However, the increased modeling flexibility due to arbitrarily-sized hyperedges makes hypergraph partitioning more difficult than graph partitioning in practice [HC14], since algorithms on hypergraphs are considered to be "inherently more complicated than those on graphs" [Kay+12] and thus more complex "in terms of implementation and running time" [Bul+16]. Despite its numerous applications, hypergraph partitioning is still considered to be "relatively less studied than graph partitioning" [Kab+17] and although it "is widely used in both academia and industry, the number of publicly available tools is limited" [ACU08].

1.2 Main Contributions

This dissertation presents several algorithmic advances to the multi-level paradigm, which yield a high-quality hypergraph partitioning system that computes the best solutions for a wide range of benchmark hypergraphs from different application areas for *both* the cut-net and the connectivity metric – while still having a running time comparable to that of hMETIS, the previously best system in terms of solution quality.

This work goes beyond the individual publications [Sch+16a; Akh+17a; HS17a; ASS18a; HSS18a; HSS19a] and technical reports [Hen+15a; Sch+15a; ASS18b; HSS18b] in that we take a holistic approach by putting our improvements in the historical context and presenting all individual contributions in an integrated and consistent manner. Detailed references and attributions will be given over the course of this dissertation.

A Historical Overview of the Field. We provide a comprehensive survey of almost 50 years of hypergraph partitioning history. In contrast to previous overview articles [Kod72; Don88; AK95c; MWW95; Joh96; Lie97; Kah98; CC00; KAV04; PM07; Kuc08; Bul+16] and related work sections in dissertations on hypergraph partitioning [Alp96; Çat99; Lim00; Tri06; Lot16], we present a historical overview – instead of an aggregation of similar concepts and techniques – that traces the lineage of ideas throughout history and thus makes the scientific development of the field more comprehensible.

***n*-Level Hypergraph Partitioning.** We revisit the trade-off between *solution quality* and *running time* inherent in the number of hierarchy levels, which is itself determined by the rate at which successively smaller hypergraphs are produced in the coarsening phase. Instead of using an approximately logarithmic number of levels like traditional multi-level algorithms, we show how to *evade* this trade-off completely by going to the extreme case of (nearly) n *levels*, removing only a single vertex in every

level of the hierarchy. For this purpose, we generalize several basic ideas from the n-level graph partitioning algorithm KaSPar [OS10a; OS10b] to hypergraph partitioning and propose a hypergraph data structure, as well as two coarsening algorithms specifically engineered to fit the n-level paradigm. This very fine-grained approach makes consecutive hierarchy levels as similar as possible and thus provides refinement algorithms with many opportunities to improve the solution. By devising several lazy-evaluation techniques and sophisticated caching mechanisms for the algorithms used in the coarsening and the refinement phase, we reduce the running time by more than two orders of magnitude compared to a naïve adaptation of the n-level approach used in KaSPar for traditional graph partitioning.

Hypergraph Sparsification. Especially for hypergraphs with many large hyperedges, computations involving the set of neighbors of a vertex can have a significant impact on the overall running time of a partitioning algorithm. We alleviate this impact by presenting a preprocessing technique based on locality-sensitive hashing [IM98] that identifies and merges vertices with similar neighborhoods to reduce the sizes of large hyperedges and thus significantly speeds up the overall partitioning process.

Community-Aware Coarsening. Furthermore, we show that traditional coarsening algorithms lack a global view of the problem. Since they are solely guided by local, greedy decisions, they are prone to perform contractions that obscure the naturally existing structure in the hypergraph. We therefore present an approach which incorporates *global* information about the *community structure* into the coarsening process. Community detection is performed via modularity maximization using the Louvain algorithm [Blo+08] on a bipartite graph representation which reflects key structural properties of the hypergraph. Our experimental results indicate that respecting community structure during coarsening not only significantly improves the solutions found by the initial partitioning algorithm, but also consistently improves overall solution quality.

Portfolio of Initial Partitioning Algorithms. We present a portfolio-based approach to initial partitioning to increase diversification. Instead of using a single algorithm to compute the initial partitions of the coarsest hypergraphs, we use several initial partitioning algorithms including random assignment, breadth-first search (BFS), size-constrained label propagation [MSS14], and different variants of greedy hypergraph growing [ÇA99].

Advanced Refinement Algorithms. We present three local improvement algorithms to be applied in the refinement phase to improve the solution quality of an initial partition. Two algorithms are based on the Fiduccia-Mattheyses (FM) heuristic [FM82], but perform a *highly localized* search that starts with only two vertices and then gradually expands by successively considering their neighbors. FM-style algorithms move vertices to other blocks in the order of improvements in the objective, allowing the objective to temporarily worsen to escape from local optima. The first presented algorithm is specifically tailored to improving two-way partitions (i.e., partitions consisting of $k = 2$ blocks) and can therefore be used in a setting where k-way

partitions are computed via *recursive bipartitioning*. The second algorithm has a more global view and is able to directly improve k-way partitions by moving vertices between *all* k blocks. It represents the first *FM-style k-way refinement* algorithm (previously regarded as "rather complex to describe and to implement" [BS11, p. 72]) that can be efficiently employed in the multi-level setting. Current multi-level hypergraph partitioning systems that employ k-way refinement schemes only use weaker greedy local search algorithms which cannot escape from local optima [KK00; ACU08; TK08; Çat+12b]. The third algorithm uses max-flow computations on pairs of blocks to refine k-way partitions. For this, we generalize the *flow-based refinement* framework of the graph partitioner KaFFPa [SS11] from graph to hypergraph partitioning, identify shortcomings of the KaFFPa approach that unnecessarily restrict feasible solutions, and introduce an improved model that overcomes these limitations.

Memetic n-Level Hypergraph Partitioning. As with many metaheuristics, multi-level hypergraph partitioning gives better results if several repeated runs are made with some measures taken to diversify the partitioning process. Still, even a large number of repeated executions can only scratch the surface of the huge search space of possible partitions. In order to explore the global solution space extensively, more sophisticated metaheuristics are needed. This is where memetic algorithms, i.e., genetic algorithms combined with local search, come into play. Memetic algorithms permit effective exploration (global search) and exploitation (local search) of the solution space. We therefore embed our n-level algorithms into an evolutionary framework – generalizing and extending several ideas from the evolutionary graph partitioning algorithm KaFFPaE [SS12] from graphs to hypergraphs – and present the first *memetic* hypergraph partitioning algorithm that uses the multi-level paradigm to effectively exploit the local solution space. Key components of our contribution are new effective recombination operators that incorporate information about the best solutions into the coarsening process and mutation operators that provide a large amount of diversification.

The Karlsruhe Hypergraph Partitioning Framework. The algorithmic components presented in this dissertation form the basis of our open-source hypergraph partitioning framework *KaHyPar* (Karlsruhe Hypergraph Partitioning), which is implemented in C++ and publicly available from `http://www.kahypar.org`. KaHyPar supports both direct k-way and recursive bipartitioning-based partitioning, and allows for optimizing the cut-net metric as well as the connectivity metric. Since its release, it has already been employed in several research efforts [AH19; GLA19; Got19; Jal19b; Net19; PS19; Sch+19a; SSS19b] and has also attracted attention from industry.

An Extensive Experimental Evaluation. The lack of experimental data comparing partitioning algorithms has been criticized several times [Don88; HB97]. Quite often, newly developed algorithms are only evaluated on a few hypergraphs (usually derived from a single application domain) and only compared with a small subset of the available hypergraph partitioning systems. The experimental study presented in this dissertation is, to the best of our knowledge, the *most comprehensive evaluation*

to date in the literature. We compare our algorithms with several state-of-the-art hypergraph partitioning systems: hMETIS [Kar+97a; KK99] and PaToH [ÇA99] (widely used in practice for about 20 years), Mondriaan [VB05] (a still actively developed matrix partitioner), and the recently proposed algorithms Zoltan-AlgD [SS18b] and HYPE [May+18]. Experiments are performed on a wide range of hypergraphs derived from well-established benchmark suites of the VLSI design, the SAT solving, the scientific computing, and the graph partitioning community.

Our results indicate that KaHyPar – already without the memetic component – computes better solutions than *all* competing algorithms for *both* the cut-net and the connectivity metric, while being faster than Zoltan-AlgD and having a running time comparable to that of hMETIS, the previously best system in terms of solution quality. Hence, our algorithms are of particular interest for applications such as VLSI design, where even small improvements in solution quality are considered critical [HB97; WA98]. In settings where running time is of higher importance than solution quality, our experiments suggest that PaToH should be considered the method of choice. Additional experiments show that, when used as a graph partitioner, KaHyPar compares favorably with the current best algorithm KaFFPa [SS11] – computing solutions of slightly higher quality for complex networks and solutions of comparable quality for the instances of the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering [Bad+13] in a comparable amount of time.

1.3 Research Methodology

This dissertation adopts the *algorithm engineering* research methodology [San09; MS10; SW11; SW13; Ang+19], which integrates classical *algorithm theory* with *experimental algorithmics* into a feedback loop of design, analysis, implementation, and experimentation.

Algorithm Theory and Experimental Algorithmics. Algorithm theory, rooted in mathematics, focuses on abstract and well-defined problems and machine models that are particularly amendable to theoretical analysis, with the goal to derive *provable* performance guarantees on running time and/or solution quality for all possible (known and unknown) types of inputs. Focusing on rigorous theoretical analyses and provable worst- or average-case performance guarantees (which are essential in the field of algorithmics), classical algorithm theory considers actual implementations of algorithmic ideas to be out of scope and thus to be left to application developers. Experimental algorithmics (i.e., "the study of algorithms and their performance by experimental means" [McG12]), on the other hand, is mainly concerned with the experimental process itself and thus addresses questions such as how to design and execute experiments, and how to evaluate the experimental results to draw meaningful conclusions. While an implementation is necessary to perform experiments, the actual process of implementing an algorithm is not addressed sufficiently. Yet, transforming a high-level algorithmic pseudocode description into an efficient implementation for mod-

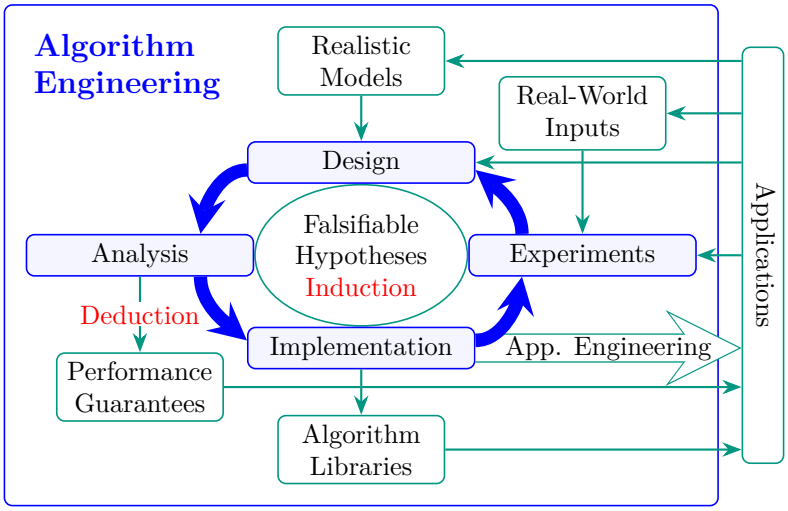


Figure 1.2: The algorithm engineering cycle of design, analysis, implementation, and experiments – interfacing with real-world applications. Adapted from [Bin18].[1]

ern hardware requires overcoming large semantic gaps. Elegant theoretical algorithms might require sophisticated data structures almost impossible to implement efficiently. Furthermore, constant factors – ignored in the asymptotic analysis – suddenly play an important role for the practicability of an implementation.

Algorithm Engineering. Algorithm engineering (AE) addresses the apparent gap between algorithm theory on the one hand and applications, implementations, and experiments on the other hand by providing a holistic approach centered around a cycle of design, analysis, implementation, and experimentation – driven by falsifiable hypotheses. Aiming at practicality, algorithm engineering interacts with the application domain in several important ways. Applications form the basis for *realistic* models, supply the AE process with *real-world* inputs (which are often significantly different from the worst-case instances used in theory), and may influence algorithm design as well as the experimental evaluation. Moreover, algorithm engineering fosters the transfer of knowledge from the research domain to the application domain. Highly efficient, well-engineered algorithm libraries provide clean, generic interfaces to a variety of applications. A dedicated process of *application engineering* aims at transforming research implementations into production-grade software. Furthermore, incorporating realistic models and practical considerations already at the algorithm design phase supports theoretical analyses in deducing performance guarantees that are relevant in practice. A visualization of the algorithm engineering methodology is shown in Figure 1.2.

[1] "Algorithm engineering as a cycle of design, analysis, implementation, and experiments." by Timo Bingmann, used under CC BY 4.0 / Text has been capitalized.

On Performance Guarantees. The partitioning problem we consider in this dissertation is known to be NP-hard [GJS76; Len90]. Furthermore, deriving performance guarantees for the multi-level heuristics used in practice is still an open problem [San09]. Hence, our focus is more on the design, implementation, and experimental evaluation of partitioning algorithms. However, in Section 2.4, we will briefly elaborate on the computational complexity of problems involving partitions of graphs and hypergraphs.

Interactions with Applications. Our work interfaces with the application domain in various ways. The benchmark instances used in the experimental evaluation stem from real-world problems of the VLSI design, the SAT solving, and the scientific computing community, which are transformed into hypergraphs using widely-accepted models. A detailed discussion will be given in Section 2.6.1. Moreover, our key software artifact – the *open-source* partitioning framework KaHyPar – is not only instrumented for experiments (e.g., by providing detailed statistics and timings in a format compatible with SqlPlotTools [Bin14]), but also provides C++ and Julia [Jal19a] library interfaces and can additionally be used as a standalone application. Furthermore, it is designed as an extensible framework to foster the research and development of new heuristics.

Reproducibility. The HGP research community has been criticized for not describing implementations in sufficient detail for others to reproduce the results [CKM99b]. As we will see in Section 3.2, this already applies to the well-known and widely-used Fiduccia-Mattheyses (FM) heuristic [FM82]. While a high-level description of this algorithm is straightforward, several ambiguities and *implicit* design decisions may lead to considerable differences in the solution quality of actual implementations [HHK97; CKM99b; CKM00b]. With multi-level algorithms, the semantic gap between the algorithmic description and the implementation becomes even more pronounced. Caldwell, Kahng, and Markov [CKM00c], for example, note that "[a] lack of documented key implementation details in the literature [...], and the implementation complexity of hMETIS techniques, may be factors contributing to the lack of integration of hMETIS quality partitioning methods in the VLSI community." This is especially problematic, as some of most sophisticated partitioning algorithms to date are only distributed in binary format. However, describing an entire multi-level algorithm in precise pseudocode such that it could be transcribed into an actual implementation is impractical for both the author *and* the reader due to the high complexity of such systems. Therefore, the approach taken in this dissertation is to employ pseudocode as a means to support and enhance the textual description of an algorithm. However, the actual implementations of all algorithms are made *publicly available* through KaHyPar – thus documenting and disclosing all design decisions and implementation details. Moreover, tuning parameters are externalized and accessible via INI configuration files instead of being hidden in the source code. At the time of writing, KaHyPar ranks third (out of 25 scientific software tools) in the softwipe code quality benchmark with a quality score of 8.8/10.[2] To further support reproducibility, all benchmark hypergraphs used in this work are made publicly available [Sch19].

[2] `https://github.com/adrianzap/softwipe/wiki/Code-Quality-Benchmark`

Types of Experimental Publications. In his "Theoretician's Guide to the Experimental Analysis of Algorithms", David Johnson [Joh99] differentiates between four types of publications involving experimentation (with slightly different reasons for doing the actual work of implementing an algorithm): *application papers* (describing the impact of an algorithm on a particular application), *horse race papers* (demonstrating superiority of an algorithm over competitors on standard benchmark sets), *experimental analysis papers* (trying to better understand strengths and weaknesses of algorithmic ideas in practice), and *experimental average-case papers* (generating conjectures about average-case behavior where theoretical analysis is too hard).

The work summarized in this dissertation falls into the *experimental analysis* and *horse race* categories. After describing the algorithmic ideas that form the basis of our framework, we experimentally evaluate their effects on solution quality and running time as well as their interactions. Having determined the best configurations, we then enter into a horse race with the major state-of-the-art competitors to demonstrate the effectiveness of our algorithms, but also to assess today's partitioning algorithms in terms of the time/quality trade-off.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents the foundations of this work and discusses our experimental design as well as our experimental methodology. In Chapter 3, we then present an extensive review of prior work on hypergraph partitioning. Afterwards, Chapter 4 focuses on the algorithmic components of our n-level partitioning system, which is extended into a memetic algorithm in Chapter 5. Both chapters include extensive evaluations of different aspects of the proposed algorithms. Chapter 6 then compares our entire KaHyPar system with the previous state of the art. Finally, Chapter 7 concludes the dissertation with a brief summary and directions for future research.

> **Digressions.** Throughout this dissertation, we use digressions like this to present some additional information, insights or historical context that is related to, but not essential for the understanding of the main text. The idea to differentiate between the main text and additional digressions in such a fashion is adopted from the work of Sebastian Wild [Wil16].

Chapter 2

2 Preliminaries

"When I *use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."*

"The question is," said Alice, "whether you can *make words mean so many different things."*

"The question is," said Humpty Dumpty, "which is to be master—that's all."

— Lewis Carroll, Through the Looking-Glass, and What Alice Found There

Chapter Overview. This chapter covers the fundamentals of this dissertation. In Section 2.1, we first introduce our terminology and the basic notation. Section 2.2 then defines the k-way hypergraph partitioning problem addressed in this work. Section 2.3 gives an overview of related problem formulations that exist in the literature. Afterwards, Section 2.4 briefly discusses the computational complexity of classical cut problems in graphs and hypergraphs. In Section 2.5, we then introduce the most common algorithmic approaches to solve the hypergraph partitioning problem. Finally, Section 2.6 elaborates on our experimental design and experimental methodology.

References. This chapter consolidates the notation and definitions of several publications [Sch+16a; Akh+17a; HS17a; ASS18a; HSS18a; Baa+19a; HSS19a; Sch+19a]. Some text passages are therefore copied verbatim.

2.1 Notation and Definitions

Hypergraphs. A *weighted undirected hypergraph* $H = (V, E, c, \omega)$ is defined as a set of n vertices V and a set of m hyperedges/nets E with vertex weights $c : V \to \mathbb{R}_{>0}$ and net weights $\omega : E \to \mathbb{R}_{>0}$, where each net e is a subset of the vertex set V (i.e., $e \subseteq V$). The vertices of a net are called *pins*.

> The terminology of "nets" and "pins" originates from the VLSI design community, where a circuit is repesented by a netlist, and each signal net represents a set of circuit elements that must be interconnected. Since circuit elements can be part of several nets, each net is assigned a specific point of contact with the circuit element, known as a pin [CKM00b; CKL03]. Both terms have since been widely adopted in the hypergraph partitioning community.

We extend c and ω to sets in the natural way, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e if $v \in e$. $\mathrm{I}(v)$ denotes the set of all

incident nets of v. The set $\Gamma(v) := \{u \mid \exists e \in E : \{v, u\} \subseteq e\}$ denotes the neighbors of v. The *degree* of a vertex v is $d(v) := |\mathrm{I}(v)|$. We use Δ_v to denote the maximum degree, i.e., $\Delta_v := \max_{v \in V} d(v)$. The *size* $|e|$ of a net e (or its *arity* [BS11, p.71]) is the number of its pins, and we use $\Delta_e := \max_{e \in E} |e|$ to denote the maximum net size (also referred to as the *rank* r of the hypergraph). We assume hyperedges to be sets rather than multisets, i.e., a vertex can only be contained in a hyperedge *once*. We use P to denote the multiset of all pins in H. Note that $p := |P| = \sum_{v \in V} d(v) = \sum_{e \in E} |e|$. The number of pins p is used to measure the *size* of a hypergraph, since neither the number of pins per net nor the number of nets per vertex is bounded, and thus $n \in \mathcal{O}(p)$ and $m \in \mathcal{O}(p)$ [FM82]. We use $\overline{d(v)}$ and $\overline{|e|}$ to denote the average vertex degree/average net size, while $\widetilde{d(v)}$ and $\widetilde{|e|}$ are used to refer to the median vertex degree/median net size. Nets of size one are called *single-vertex* nets. We call two nets e_i and e_j *parallel* if $e_i = e_j$. Given a subset $V' \subset V$, the *subhypergraph* $H_{V'}$ is defined as $H_{V'} := (V', \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\})$, and the *section hypergraph* $H \times V'$ is defined as $H \times V' := (V', \{e \in E \mid e \subseteq V'\})$ [Ber75; Ber85]. In this dissertation, hypergraphs are drawn according to the subset standard [Mäk90] (see Figure 2.1 (left)).

Partitions and Clusterings. A *k-way partition* of a hypergraph H is a partition of its vertex set into k *blocks* $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$, and $V_i \cap V_j = \emptyset$ for $i \neq j$. The block that vertex v is assigned to is denoted with $b[v]$. We call a k-way partition Π *ε-balanced* if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1 + \varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some parameter ε. We call a block V_i *overloaded* if $c(V_i) > L_{\max}$ and *underloaded* if $c(V_i) < L_{\max}$.

For each net e, $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e)$ of a net e is the cardinality of its connectivity set, i.e., $\lambda(e) := |\Lambda(e)|$. A net is called a *cut net* if $\lambda(e) > 1$, otherwise (i.e., if $|\lambda(e)| = 1$) it is called an *internal* net. A vertex u incident to at least one cut net is called a *boundary vertex*. The number of pins of a net e in block V_i is defined as $\Phi(e, V_i) := |\{V_i \cap e\}|$. A block V_i is *adjacent* to a vertex $v \notin V_i$ if $\exists e \in \mathrm{I}(v) : V_i \in \Lambda(e)$. We use $\mathrm{B}(v)$ to denote the set of all blocks adjacent to v. Given a k-way partition Π of H, the *quotient graph* $Q := (\Pi, \{(V_i, V_j) \mid \exists e \in E : \{V_i, V_j\} \subseteq \Lambda(e)\})$ contains an edge between each pair of adjacent blocks.

A *clustering* $C = \{C_1, \ldots, C_l\}$ of a hypergraph is a partition of its vertex set. In contrast to a k-way partition, the number of clusters is not given in advance, and there is no balance constraint on the actual sizes of the clusters C_i.

Contractions and Uncontractions. *Contracting* a pair of vertices (u, v) means merging v into u. We refer to u as the *representative* and to v as the *contraction partner*. After contraction, the weight of u becomes $c(u) := c(u) + c(v)$. We connect u to the former neighbors $\Gamma(v)$ of v, by replacing v with u in all nets $e \in \mathrm{I}(v) \setminus \mathrm{I}(u)$. Furthermore, we remove v from all nets $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$. If a contraction leads to parallel nets, we remove all but one from H. The weight of the remaining net e is set to the sum of the weights of the nets parallel to e. Single-vertex nets created by a contraction are removed from the hypergraph, since such nets can never become part of the cut

set. *Uncontracting* a vertex u reverses the contraction. The uncontracted vertex v is put in the same block as u and the weight of u is set back to $c(u) := c(u) - c(v)$.

Graphs. Let $G = (V, E, c, \omega)$ be a weighted (directed) graph. In this dissertation, we use *vertices* and *hyperedges/nets* when referring to hypergraphs and *nodes* and *edges* when referring to graphs. However, we use the same notation to refer to node weights c, edge weights ω, node degrees $d(v)$, and the set of neighbors Γ. In an undirected graph, an edge $(u, v) \in E$ implies an edge $(v, u) \in E$ and $\omega(u, v) = \omega(v, u)$. A *path* $P = \langle v_1, \ldots, v_k \rangle$ is a sequence of nodes such that each pair of consecutive nodes is connected by a directed edge. A *strongly connected component* $U \subseteq V$ is a set of nodes such that for each $u, v \in U$ there exists a path from u to v. Given a set of nodes $V = \{v_1, v_2, \ldots, v_n\}$, an ordering $v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}$ is a bijection $\pi : [1 \ldots n] \to [1 \ldots n]$, where node v_i is the jth node in the ordering if $\pi(j) = i$. A *topological* ordering is a linear ordering $\prec$ of V such that every directed edge $(u, v) \in E$ implies $u \prec v$ in the ordering. A set of nodes $B \subseteq V$ is called a *closed set* if there are no outgoing edges leaving B, i.e., if the conditions $u \in B$ and $(u, v) \in E$ imply $v \in B$. A subset $S \subset V$ is called a *node separator* if its removal divides G into two disconnected components. Given a subset $V' \subset V$, the *induced subgraph* $G[V']$ is defined as $G[V'] := (V', \{(u, v) \in E \mid u, v \in V'\})$. A *matching* $M \subseteq E$ is a set of pairwise non-adjacent edges. The weight of a matching M is defined as the sum of the weights of all edges $e \in M$. A matching M is *maximal* if there is no edge $e \in E \setminus M$ such that $M \cup e$ is a valid matching. A matching M that has maximum weight $\omega(M)$ among all possible matchings is a *maximum weight* matching. A *vertex cover* V' of an undirected graph G is a subset of the node set V such that every edge $e \in E$ has at least one endpoint in V', i.e., $\forall e = (u, v) \in E : u \in V' \vee v \in V'$. A vertex cover is *minimum* if it has the smallest possible size.

Flow Networks. A flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ is a directed graph with two distinguished nodes s and t in which each edge $e \in \mathcal{E}$ has a capacity $c(e) \geq 0$. An (s, t)-flow (or *flow*) is a function $f : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ that satisfies the *capacity constraint* $\forall u, v \in \mathcal{V} : f(u, v) \leq c(u, v)$, the *skew symmetry constraint* $\forall u, v \in \mathcal{V} \times \mathcal{V} : f(u, v) = -f(v, u)$, and the *flow conservation constraint* $\forall u \in \mathcal{V} \setminus \{s, t\} : \sum_{v \in \mathcal{V}} f(u, v) = 0$. The value of a flow $|f| := \sum_{v \in \mathcal{V}} f(s, v)$ is defined as the total amount of flow transferred from s to t. The *residual capacity* is defined as $r_f(u, v) = c(u, v) - f(u, v)$. An edge $e = (u, v)$ is called *saturated* if $r_f(e) = 0$. Given a flow f, $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f, r_f)$ with $\mathcal{E}_f = \{(u, v) \in \mathcal{V} \times \mathcal{V} \mid r_f(u, v) > 0\}$ is the *residual network*. An (s, t)-cut is a bipartition $(\mathcal{S}, \mathcal{V} \setminus \mathcal{S})$ of a flow network $\mathcal{N}$ with $s \in \mathcal{S} \subset \mathcal{V}$ and $t \in \mathcal{V} \setminus \mathcal{S}$. The capacity of an (s, t)-cut is defined as $\sum_{e \in \mathcal{E}'} c(e)$, where $\mathcal{E}' = \{(u, v) \in \mathcal{E} : u \in \mathcal{S}, v \in \mathcal{V} \setminus \mathcal{S}\}$. The max-flow min-cut theorem states that the value $|f|$ of a maximum flow f is equal to the capacity of a minimum cut separating s and t [FF56].

Graph-based Hypergraph Representations. The two most common ways to represent an undirected hypergraph $H = (V, E, c, \omega)$ as an undirected graph are the *clique* and the *bipartite* representation [HM85]. In the *clique* graph $G_x(V, E_x \subseteq V^2)$ of H, each net is replaced with an edge for each pair of vertices in the net, i.e., the edge

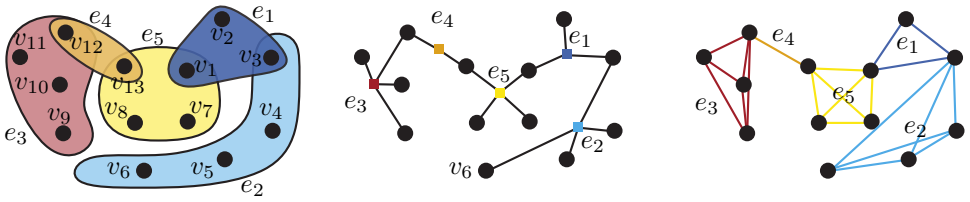


Figure 2.1: A hypergraph with 13 vertices, 5 nets, and 17 pins (left) along with the corresponding bipartite (middle) and clique-net representation (right).

set is defined as $E_x := \{(u, v) : u, v \in e, e \in E\}$ [CP68]. Thus, the pins of a net e with size $|e|$ form a $|e|$-clique in G_x. In the *bipartite* graph $G_*(V \dot{\cup} E, F)$ the vertices and nets of H form the node set and for each net e incident to a vertex v, we add an edge (e, v) to G_* [SK72]. The edge set F is thus defined as $F := \{(e, v) \mid e \in E, v \in e\}$. Each net in E therefore corresponds to a star in G_*, and $|F| = p$. The bipartite graph G_* is also known as the *incidence graph* [Bre13]. In both graph models, node weights c and edge weights ω are chosen according to the problem domain [VH90; HMV92; AK95c; Had95]. The most common weighting scheme for the edges e in the clique graph is $1/(|e| - 1)$, where $|e|$ is the size of the corresponding hyperedge [Len90; HK92c]. An example of a hypergraph, along with its corresponding clique graph and bipartite graph is shown in Figure 2.1.

2.2 The *k*-way Hypergraph Partitioning Problem

Problem Definition. The *k-way hypergraph partitioning problem* is to find an ε-balanced k-way partition Π of a hypergraph $H = (V, E, c, \omega)$ that *minimizes* an objective function over the cut nets for some value of ε. In this dissertation, we consider the two most commonly used cost functions, namely the *cut-net* metric $\mathfrak{f}_c(\Pi) := \sum_{e \in E'} \omega(e)$ and the *connectivity* metric $\mathfrak{f}_\lambda(\Pi) := \sum_{e \in E'} (\lambda(e) - 1)\ \omega(e)$, where E' is the *cut-set* (i.e., the set of all cut nets) [Don88; Çat+15]. While the cut-net metric sums the weights of all nets that connect more than one block of the partition Π, the connectivity metric additionally takes into account the actual number λ of blocks connected by the cut nets. Optimizing each of the objective functions is known to be NP-hard [Len90]. Note that for $k = 2$, $\mathfrak{f}_c(\Pi) = \mathfrak{f}_\lambda(\Pi)$. If $k = 2$ and $\varepsilon > 0$, the problem is referred to as the *bipartitioning* problem. A *bisection* of an unweighted hypergraph is a bipartition in which the number of vertices in the two blocks differs by *at most* one (i.e., $\varepsilon = 0$).

> The cut-net objective is perhaps the most widely studied objective function for hypergraph partitioning [Alp96]. In VLSI placement, a reduced net cut is correlated with shorter wires, while for parallelization of operations like sparse matrix-vector multiplications a smaller cut-size translates to reduced communication between processing elements [PM07]. For $k > 2$, objectives that take the

> connectivity λ of cut nets into account are considered preferable. For VLSI design applications, this is because signal nets that connect more blocks can consume more resources [KLK93; AK95c]. When hyperedges are used to model volumes of data exchanges, their connectivity determines how often an element has to be communicated if data is distributed according to a k-way partition [BS11, p. 67].

Note that both cost functions revert to edge-cut (i.e., the sum of the weights of those edges that have endpoints in different blocks) for plain graphs. We therefore treat the *graph partitioning* problem minimizing the edge-cut as a special case of the hypergraph partitioning problem. For a comprehensive overview over the graph partitioning literature, we refer the reader to the survey of Buluç et al. [Bul+16].

Recursive Bipartitioning and Direct k-way Partitioning. In general, there are two approaches to computing a k-way partition of a hypergraph. If k is a power of two, *recursive bipartitioning* (RB) algorithms obtain the final k-way partition by first computing a bipartition of the initial hypergraph, and then recursing on each of the two blocks. Thus, it takes $\log_2(k)$ such phases until the hypergraph is partitioned into k blocks. If k is not a power of two, the approach has to be adapted to produce appropriately-sized blocks. The 2-way partitions computed in the RB approach form a tree, which is called the *partition tree* [ST97]. In *direct k-way partitioning*, the hypergraph is directly partitioned into k blocks, without the detour over the recursive 2-way approach. We will discuss advantages and disadvantages of both techniques in Section 3.6.3.

Graph-based Hypergraph Partitioning. In general, it is not possible to solve the hypergraph partitioning problem by first modeling the hypergraph as a graph, and then using graph partitioning algorithms on the graph-based hypergraph representation. Ihler, Wagner, and Wagner [IWW93] show that there is no *cut-model* $G = (V, E, \omega)$ for a hypergraph $H = (V, N, \omega)$ such that for any bipartition of the node/vertex set V the weight of the cut edges $E' \subseteq E$ is the same as the weight of the cut-nets $N' \subseteq N$. Furthermore, they show that even the addition of dummy vertices to the graph G does not help unless negative edge weights are allowed – which is not the case for traditional graph partitioning algorithms. Moreover, for the clique-net model, Lengauer [Len90, p. 259] shows that independent of the weighting scheme for the graph edges, there always exists some bipartition with a deviation of $\Omega(\sqrt{|e|})$ from the desired unit cost of cutting an unweighted net.

2.3 Related Problem Formulations

While this dissertation focuses on the traditional hypergraph partitioning problem as described in the previous section, several related problem formulations exist in the literature. In the following, we briefly summarize the most prominent ones with the intent to give the reader a broader overview about existing problem variants.

Other Objective Functions. Several other objective functions exist in the literature [AK95c]. Metrics closely related to the connectivity metric $\mathfrak{f}_\lambda(\Pi)$ are the

$\lambda(\lambda-1)$-metric (used to model the communication volume for an all-neighbor communication pattern) [For+13] and the sum-of-external-degrees (SOED) metric $f_s(\Pi)$, for which each cut net e contributes $\lambda \cdot \omega(e)$ to the cut-size [IKS75; KK00] (sometimes used in the VLSI placement context [AK95c; SS95]). Other objectives, such as the *ratio cut* for bipartitioning, and its generalization to k-way partitioning (named *scaled cost*), not only consider the cut-size, but also take into account the balance of the partition. The ratio cut is defined as the bipartition $\Pi = (A, B)$ that generates the minimum ratio between the size of the cut-set and the product of the block weights, i.e., $f_{rc}(\Pi) := f_c(\Pi)/(c(A) \cdot c(B))$ [WC89; WC91]. Here, the numerator favors small cut-sizes, while the denominator favors a more balanced bipartition. For a k-way partition, the scaled cost metric is defined as $f_{sc}(\Pi) := \frac{1}{n(k-1)} \sum_{i=1}^{k} \frac{E_i}{c(V_i)}$, where $E_i := \omega\{e \in E \mid \lambda(e) > 1 \wedge e \cap V_i \neq \emptyset\}$ [CSZ93; CSZ94].

Naturally Imbalanced Partitions. Motivated by the application of hypergraph partitioning to consensus clustering [SG02a; SG02b] (i.e., combining multiple clustering solutions into a single clustering), in which high-quality partitions often contain highly imbalanced blocks, Yaros and Imielinski [YI13] propose to use an information-theoretic entropy-based balance constraint

$$c_l \leq -\sum_{i}^{k} \frac{c(V_i)}{c(V)} \lg\left(\frac{c(V_i)}{c(V)}\right) \leq c_u, \tag{2.1}$$

where c_l and c_u are user defined bounds, to permit partitioning algorithms to recover naturally imbalanced partitions.

Hyperedge Partitioning. Graph *edge partitioning*, (i.e., computing a partition of the edge set of a graph into blocks of bounded size) is successfully employed in distributed graph processing frameworks for load-balancing computations on large scale-free networks with skewed degree distributions [Gon+12; Li+17]. This sparked interest in the *hyperedge partitioning* problem, in which hyperedges are assigned to blocks, and the vertex cut and the replication metrics are optimized [Yan+16; Yan+18a; Yan+18b]. The former hereby is the analogue to the cut-net metric, while the latter is the analogue to the connectivity metric.

Judicious Hypergraph Partitioning. The judicious version of the hypergraph partitioning problem strives to find a k-way partition that minimizes the *maximum* number of nets a block is connected to. In other words, judicious partitioning attempts to optimize $\min \max(L(V_1), \ldots, L(V_k))$, where $L(V_i) := |\{e \in E \mid \Phi(e, V_i) > 0\}|$ is the load of a block V_i. The problem is known to be NP-hard [ZTY15] and has mainly been studied in the context of extremal combinatorics [BS97; BS02; Sco05]. To the best of our knowledge, algorithmic aspects of the problem were first studied by Alistarh, Iglesias, and Vojnovic [AIV15] in a streaming setting and by Tan, Gui, Wang, Gao, and Yang [Tan+17], whose algorithm relies on heuristically solving minimal set cover problems. Motivated by a data distribution problem in distributed phylogenetic inference, a parallel version of the latter algorithm was recently implemented and

engineered for a specific flavor of judicious hypergraph partitioning where every vertex has the same degree. The corresponding paper [Baa+19a] was jointly published with Ivo Baar, Lukas Hübner, Peter Oettig, Adrian Zapletal, Alexandros Stamatakis, Benoit Morel, and the author of this dissertation.

Multi-Objective and Multi-Constraint Partitioning. Whereas in the traditional hypergraph partitioning problem a single objective is optimized subject to a single balance constraint, there also exist partitioning formulations that address *multiple* objectives and/or *multiple* constraints at once. For multi-objective partitioning, the two most common approaches are to either keep the objectives separate and to assign different priorities to each objective (i.e., the objective with highest priority is optimized directly, while the other objectives are used for breaking ties), or to create a single, "real" multi-objective function that combines the individual objectives numerically [Aba+02; CLW03; SK03; SK06; Çat+12b; Çat+15]. In multi-constraint partitioning, a weight vector is assigned to each vertex and several balance constraints need to be satisfied for a solution to be feasible [Kar99; ÇA01b]. This restricts the solution space, since vertex movements between blocks are more limited [ACU08]. There also exists some work on partitioning problems for which the balance constraints are complex functions of the partition [PH01; KRU11], i.e., vertex weights cannot be simply defined a priori before partitioning, but instead depend on the partition itself.

Partitioning with Vertex Replication. In the replicated partitioning problem formulation, the goal is to further reduce the cut-size of a partition by determining a set of vertices that is then replicated to different blocks. Approaches to solve the problem can be divided into two categories [STA12; YA14]. The first category corresponds to *one-phase* algorithms that perform partitioning and vertex replication simultaneously [KN91; Liu+95a; Liu+95b]. The second category of *two-phase* algorithms first computes a partition optimizing some cut-related objective, and in the second phase performs vertex replication on the solution of the first phase [HE92; HG95; YW95].

2.4 On The Computational Complexity of Cut Problems

In the following, we briefly discuss the computational complexity of some cut problems for graphs and hypergraphs. We start with the problem of computing *minimum cuts*, i.e., a minimum-weight set of (hyper)edges whose removal partitions the vertices into two connected components. We then turn to the more general *minimum k-cut* problem, where the goal is to find a subset of (hyper)edges of minimum weight whose removal partitions the vertex set into (at least) k blocks. Thus, a minimum cut is simply a minimum k-cut for $k = 2$. In both of these problems, there are no restrictions on the sizes of the k blocks. At the end of this section, we therefore look at *balanced k-way partitioning*, i.e., the problem variant addressed in this dissertation, in which the size of each of the k blocks is restricted to at most $(1 + \varepsilon)$ times the average block size.

Minimum Cuts. The minimum cut in a weighted graph can be found in $\mathcal{O}(mn + n^2 \log n)$ time using either Nagamochi and Ibaraki's algorithm [NI92] or the algorithm of

Stoer and Wagner [SW97]. Furthermore, there exists a linear-time $(2+\varepsilon)$ approximation algorithm [Mat93]. For unweighted graphs, the current fastest algorithm due to Henzinger, Rao, and Wang [HRW17] runs in time $\mathcal{O}(m \log^2 n \log\log^2 n)$. The minimum (s,t)-cut problem [HR55] is a closely related problem which asks to find a minimum cut that separates two given nodes s and t. It can be solved by computing a maximum (s,t)-flow [FF56] using e.g. the push-relabel algorithm of Goldberg and Tarjan [GT86] which runs in time $\mathcal{O}(mn \log(n^2/m))$ and is among the fastest algorithms in practice.

For hypergraphs, the fastest known minimum cut algorithms are based on the algorithm of Stoer and Wagner [SW97] and run in $\mathcal{O}(np)$ time for unweighted hypergraphs and in time $\mathcal{O}(np + n^2 \log n)$ for weighted hypergraphs [KW96; MW00]. Chekuri and Xu [CX18] give an $\mathcal{O}(p + \lambda n^2)$ time algorithm for computing a minimum cut for unweighted hypergraphs (where λ is the minimum cut value), and present a $(2+\epsilon)$-approximation algorithm that runs in time $\mathcal{O}(\frac{1}{\epsilon}(p \log n + n \log^2 n))$ for weighted and in time $\mathcal{O}(p/\epsilon)$ for unweighted hypergraphs. Lawler [Law73] shows that the minimum (s,t)-cut problem for hypergraphs can be solved by computing a maximum (s,t)-flow in an auxiliary graph with $n + 2m$ nodes and $2p + m$ edges. Li, Lillis, and Cheng [LLC95] present a push-relabel algorithm that operates directly on the hypergraph.

Minimum k-Cuts. While minimum 2-cuts can be computed in polynomial time, Goldschmidt and Hochbaum [GH94] showed that the minimum k-cut problem is NP-hard when k is part of the input. When k is fixed to a constant, they were able to give a deterministic $\mathcal{O}(n^{k^2} T(n,m))$-time algorithm, where $T(n,m)$ is the time required to compute a minimum (s,t)-cut of graph with n nodes and m edges. The current fastest deterministic algorithm for constant k due to Thorup [Tho08] runs in $\tilde{\mathcal{O}}(mn^{2k-3})$ time [CQX19].[1] Furthermore, there exist several 2-approximation algorithms [SV95; NR01; NK07]. For $k = 3$, the problem can be solved in time $\mathcal{O}(n^2\ T(n,m))$ [NI00].

For hypergraphs and $k = 3$, there exists a polynomial-time algorithm that uses $\mathcal{O}(n^3)$ (s,t)-min cut computations [Xia08; Xia10]. However, no polynomial-time deterministic algorithm is known for any $k > 3$ [FPZ19]. Until recently, the complexity was unknown for any fixed $k \geq 4$ [CL15; CX17]. Chandrasekaran, Xu, and Yu [CXY18] were the first to give a randomized $\mathcal{O}(pn^{2k-1} \log n)$-time algorithm for arbitrary constant k in arbitrary rank hypergraphs. Recently, Fox et al. [FPZ19] presented a Monte Carlo algorithm for hypergraphs with arbitrary net sizes that runs in $\mathcal{O}(mn^{2k-2} \log^2 n)$ time for all $k \geq 3$ and an algorithm that runs in time $\tilde{\mathcal{O}}(n^{\max(r,2k-2)})$ for hypergraphs with constant rank r. Computing minimum k-cuts in hypergraphs is considered to be significantly harder than computing minimum k-cuts in graphs when k is part of the input, as Chekuri and Li [CL15] recently showed that the problem is at least as hard as the densest k-subgraph problem (i.e., given a graph G and an integer k, find a subset of k nodes $V' \subseteq V$ that maximizes the number of edges in the induced subgraph $G[V']$) which is not believed to admit an efficient constant factor approximation unless P=NP [CXY18].

[1] $\tilde{\mathcal{O}}$ hides polylog factors.

Balanced k-way Partitioning. For $k = 2$ and $\varepsilon = 0$, the problem corresponds to the NP-hard graph minimum bisection problem [GJS76], which can be approximated within $\mathcal{O}(\log n)$ [Räc08]. Andreev and Räcke [AR04; AR06] show that for $k \geq 3$ and $\varepsilon = 0$, there is no algorithm with a finite approximation factor for general graphs, unless P=NP. If $\varepsilon = 1$, there exists an $\mathcal{O}(\sqrt{\log k \log n})$ approximation algorithm [KNS09]. For $\varepsilon > 0$, Feldmann and Foschini [FF12; FF15] present an algorithm with approximation factor $\mathcal{O}(\log n)$. However, the running time of that algorithm increases exponentially with decreasing ε [Fel13]. Feldmann [Fel13] furthermore shows that for general graphs, there is no fully polynomial time algorithm (i.e., with running time polynomial in n/ε for any $\varepsilon > 0$) that computes ε-balanced partitions and approximates the cut size within a finite factor of α, unless P=NP (see also [Fel12, Lem. 5.3]).

Wagner and Wagner [WW93] looked at the class of partitioning problems between minimum cut on the one end and minimum bisection on the other end. They show that the problem becomes harder the more balanced the solution has to be. More precisely, if each block has to contain at least C vertices for some constant C, the problem can be solved in polynomial time. However, the problem is NP-hard if the block size $|V_i|$ is restricted to be $|V_i| \geq \alpha n^{\epsilon}$ for some arbitrarily small $\alpha, \epsilon > 0$. The case $|V_i| \geq \log n$ is still open.

Räcke, Schwartz, and Stotz [RSS18] note that "Minimum Hypergraph Bisection is much harder to approximate than Minimum Bisection in graphs [...]" and present an $\mathcal{O}(\sqrt{n} \log^{5/4} n)$ approximation algorithm. Furthermore, they show that the existence of an approximation algorithm with a guarantee of $\mathcal{O}(n^{1/4-\epsilon})$ is unlikely.

2.5 Algorithmic Approaches to Hypergraph Partitioning

Local Search Algorithms. A partitioning approach is based on the *local search* paradigm if it creates a new solution based on (i) a neighborhood structure on the set of feasible solutions, and (ii) the previous history of the search [AK95c; MS08]. The neighborhood structure hereby defines the means of moving from the current solution to a neighboring solution by performing some kind of perturbation. Common perturbation mechanisms used in partitioning algorithms are swaps of two vertices belonging to different blocks, or the movement of a single vertex from one to another block. The search space of feasible solutions is then explored by repeatedly moving from the current solution to a neighboring solution until a certain stopping criterion is fulfilled. Some local search algorithms ignore the previous history of the search (e.g., simple greedy algorithms that only use the current solution to decide to which neighboring solution to move next), while others use it to form more complex neighborhood structures than single vertex moves or pairwise swaps (e.g., a *pass* – in which several vertices are allowed to be moved once and the best solution encountered during these moves is adopted as the new solution).

Hill Climbing / Iterative Improvement (IIP) Algorithms. Hill climbing algorithms (also known as IIP algorithms in the HGP literature) are a special form of

local search algorithms that start with some feasible solution and then iteratively move to the *best* neighboring solution (i.e., the solution that yields the largest improvement in the optimization objective). The search terminates once no neighboring solution is better than the current one. Thus, hill climbing algorithms always converge to a *local optimum* with respect to both the initial solution and the chosen neighborhood structure. The search space is therefore explored by performing multiple restarts of the algorithm (each time using a different starting solution). As we will see in Chapter 3, a large part of the hypergraph partitioning literature is concerned with IIP algorithms. Furthermore, they are employed in all of today's state-of-the-art HGP systems.

Spectral Partitioning. The general idea of spectral graph partitioning [DH72; DH73; Bar82] is to use the spectrum (i.e., the eigenvectors) of the graph's Laplacian matrix to first construct a geometric representation of the graph (e.g., a linear ordering in which highly connected nodes are close to each other), and then use this representation to heuristically infer a partition of the node set. The Laplacian matrix of a graph $G = (V, E)$ is defined as $L = D - A$, where D is the diagonal degree matrix (i.e., $d_{ii} := d(v_i)$) and A is the adjacency matrix of G (i.e., $a_{u,v} := 1$ if $(u, v) \in E$, and zero otherwise). Fiedler [Fie73; Fie75a; Fie75b] showed that the eigenvector $\boldsymbol{x}_2$ corresponding to the second-smallest eigenvalue λ_2 can be used to derive a bipartition of V, because the components of $\boldsymbol{x}_2$ can be interpreted as node weights, and the difference between two weights x_i and x_j provides information about the distance between nodes v_i and v_j [BS94]. Thus, sorting the nodes by their corresponding weights creates an ordering of V in which nodes are "close" to each other. A bipartition can then be inferred by choosing a real number t and assigning all nodes v_i corresponding to entries $x_i \leq t$ to one block and all other nodes to the other block [Spi12, p. 506]. If t is chosen to be the median, the resulting partition is a bisection of the graph. This classical approach has since been extended and improved by several authors [Bar82; PSL90; Sim91; BS93; AK94c; AK95b; AKY99]. In order to partition hypergraphs using the spectral approach, the hypergraph is first transformed into a graph-based representation [AKY99]. Since spectral techniques have been mostly superseded by iterative improvement algorithms and the multi-level paradigm described next [CKM00a; MS12], we refer to the reader to Refs. [PSL90; Moh91; AK95c; CSZ99; Spi07; BS11] for a more detailed introduction and an overview of more advanced spectral partitioning approaches.

The Multi-Level Paradigm. The most prominent heuristic to tackle partitioning problems is the *multi-level* paradigm. Initially used by Barnard and Simon [BS93] to improve the running time of a recursive spectral bisection algorithm for graph partitioning, it was first independently studied by Bui and Jones [BJ93], Hendrickson and Leland [HL93; HL95], Hauck and Borriello [HB95], and Cong and Smith [CS93]. The key idea behind the multi-level approach can be summarized as follows: Instead of attempting to compute a partition directly on the input hypergraph, multi-level algorithms first compute a hierarchy of successively smaller approximations of the original instance that reflect its basic structure. This process continues until the hypergraph is small enough (i.e., contains only a small number of vertices and nets)

to permit even simple partitioning heuristics to find reasonably good solutions. The partition of the smallest hypergraph is then successively propagated back through the hierarchy to derive a partition of the original instance. During this process, local improvement algorithms are used at each level to further refine the solution. This is possible because approximations on higher levels of the hierarchy have more degrees of freedom than those on lower levels since they resemble the input hypergraph more accurately. The three phases of the multi-level paradigm are known as the *coarsening phase* (in which the hypergraph is recursively coarsened to create the hierarchy), the *initial partitioning phase* (in which a solution is computed on the coarsest/smallest hypergraph), and the *uncoarsening / refinement phase* (in which coarsening is undone and the solution is refined further). Figure 2.2 shows an illustration of this process.

In order to create hypergraphs that are *smaller than* but structurally *similar to* the given input hypergraph, algorithms employed in the coarsening phase try to identify naturally existing clusters (i.e., groups of highly-connected vertices) that can be contracted together to create the next level of the coarsening hierarchy. As we will see in the next chapter, state-of-the-art hypergraph partitioning algorithms compute matchings or clusterings using local similarity measures that take into account the neighborhood of each vertex. By adjusting vertex and net weights during contractions as described in Section 2.1, both the balance constraint and the optimization objective are preserved across levels. Thus, an initial partition computed on the coarsest level inherently induces a partition of the hypergraphs on the finer levels that has the same balance and the same solution quality. This, in turn, allows iterative improvement algorithms to efficiently exploit the additional degrees of freedom on each finer level during uncoarsening.

We will take a thorough look at multi-level hypergraph partitioning algorithms and the techniques used in each of the three phases in the literature review presented in the next chapter. However, we would also like to point the reader to the work of Cong and Shinnerl [CS03] for an extensive treatment of the multi-level paradigm in the context of combinatorial optimization and VLSI design.

The hypergraph partitioning algorithms developed in this dissertation instantiate the multi-level approach in its most extreme version – removing only a *single* vertex in every level of the hierarchy. In contrast to the approximately logarithmic number of levels created by traditional multi-level algorithms, this approach creates a hierarchy of (nearly) n levels, which is why it is referred to as the *n-level* paradigm.

Memetic Algorithms. Genetic algorithms (GAs) [Hol75a; Hol75b] are a commonly used metaheuristic for solving optimization problems [BR03; Tal09; HS15]. The general idea behind genetic algorithms is inspired by the Darwinian concept of natural selection in biological evolution [Dar59], namely to use mechanisms such as selection, mutation, recombination and survival of the fittest during the optimization process. A genetic algorithm starts with a population of individuals (in our case partitions of the hypergraph) and evolves the population over several generational cycles (rounds). In each round, the GA uses a selection rule based on the fitness of the individuals of the population to select good individuals, and combines them to obtain improved

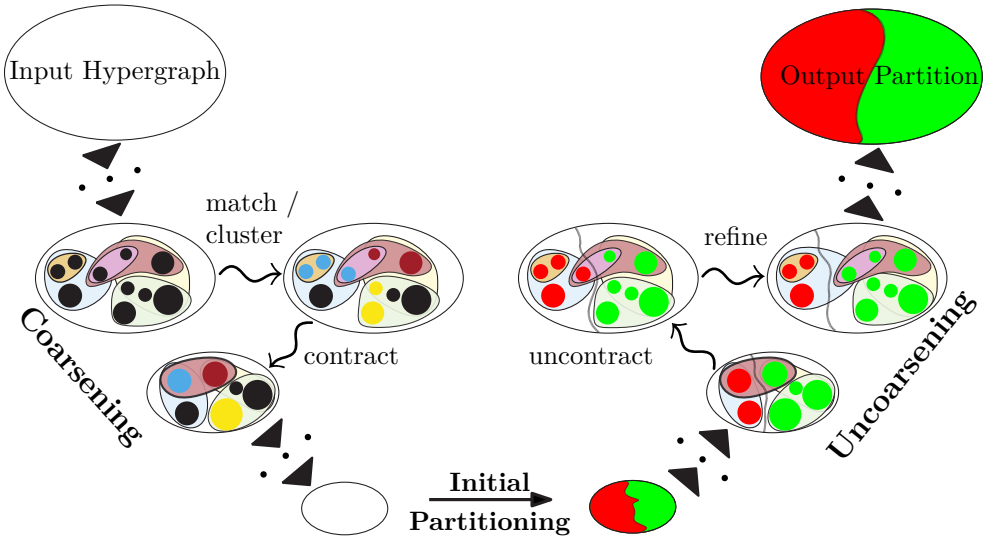


Figure 2.2: Illustration of the multi-level hypergraph partitioning process.

offspring [Gol89]. When an offspring is generated, an eviction rule is used to select one or more members of the population to be replaced by the new offspring. For a genetic algorithm it is of major importance to preserve diversity in the population [Bäc96], i.e., the individuals should not become too similar in order to avoid a premature convergence of the algorithm. This is usually achieved by using mutation operations that perturb individual solutions, and by using eviction rules that take similarity of individuals into account. Memetic algorithms (MAs) were introduced by Moscato [Mos89] and formalized by Radcliffe and Surry [RS94] as an extension to the concept of genetic algorithms. More precisely, a memetic algorithm is a genetic algorithm that is combined with local search [Kim+11], i.e., local search algorithms are used to further improve the individuals and offspring solutions of the genetic algorithm. This hybrid approach allows for effective exploration (global search) and exploitation (local search) of the solution space. We refer to the work of Moscato and Cotta [MC10] for an excellent introduction to field of memetic algorithms.

2.6 Experimental Design and Methodology

Throughout this dissertation, we present the results of a large number of experiments involving different hypergraph partitioning algorithms, as well as different configurations of the same algorithm and various kinds of hypergraphs and graphs. In this section, we therefore detail both our experimental design and our experimental methodology.

2.6.1 Benchmark Sets

The experimental evaluations presented in Chapters 4 to 6 use several benchmark sets, which we briefly summarize here. Tables 2.1 and 2.2 give an overview about the composition of each benchmark set. All data sets are publicly available via the KITopen data repository [Sch19].

The Main Benchmark Set. Benchmark set A is used as the main benchmark set. It was initially assembled by the author of this dissertation for the experimental evaluation presented in Ref. [Sch+16a] and later extended to its current size in Ref. [HS17a]. At the time of writing, it contains 488 hypergraphs derived from four well-known benchmark suites: The ISPD98 VLSI Circuit Benchmark Suite [Alp98], the DAC 2012 Routability-Driven Placement Contest [Vis+12], the SuiteSparse Matrix Collection [DH11], and the international SAT Competition 2014 [Bel+14].

We include all 18 circuits from the ISPD98 benchmark suite and all 10 circuits from the DAC 2012 benchmark suite. These VLSI instances are transformed into hypergraphs by converting the netlist into a set of hyperedges. At the time benchmark set A was assembled, the SuiteSparse Matrix Collection was organized into 172 groups where each group contained matrices of different application areas. From each group, we chose one matrix for each application area that had between 10 000 and 10 000 000 columns. In case multiple matrices fulfilled our criteria, we randomly selected one. In total, we initially included 192 matrices, which are translated into hypergraphs using the row-net model [ÇA99], i.e., each row is treated as a net, each column as a vertex, and empty rows are discarded. From the international SAT Competition 2014 [Bel+14], we randomly selected 100 instances from the application track and converted them into three different hypergraph representations: For *literal* hypergraphs, each Boolean literal is mapped to one vertex and each clause constitutes a net [PM07], while in the *primal* model each variable is represented by a vertex and each clause forms a net. In the *dual* model the opposite is the case [MP14]. Out of the 192 sparse matrix (SPM) hypergraphs and the 100 SAT instances, eight SPM hypergraphs and eight SAT hypergraphs were already excluded in Ref. [Sch+16a] because some of the evaluated algorithms either ran out of memory or did not finish within the given time limit. All subsequent publications [Akh+17a; HS17a; HSS18a; HSS19a] therefore used the 184 matrices and 92 SAT instances which are also used here. All hypergraphs have unit vertex and net weights. Basic properties of all instances are shown in Figure 2.3.

Rationale for Benchmark Set A. At the time we started working on hypergraph partitioning, most previous work only employed a small number of hypergraphs from specific application areas in the experimental evaluations. Until the work of Çatalyürek and Aykanat [ÇA99], for example, hypergraph partitioning algorithms were almost exclusively evaluated on benchmark sets from the VLSI design community. These instances (e.g. the ISPD98 hypergraphs) are known to have several salient properties such as $n \simeq m$, small average vertex degrees and net sizes, as well as the fact that they contain only a small number of very large nets [Cal+99; PM07]. Furthermore, it was already observed by Goldberg and Burstein [GB83], as well as by Liu and Wong

Table 2.1: Overview of different *hypergraph* benchmark sets. Sets B, C, and D are subsets of set A. The last row gives the reference for each benchmark set.

	Benchmark Set				
	A	B	C	D	E
DAC 2012	10	5	4	-	-
ISPD 98	18	10	10	5	-
SAT14 Primal	92	30	18	5	-
SAT14 Dual	92	30	18	5	-
SAT14 Literal	92	30	18	5	-
SPM	184	59	32	5	-
Edge Partitioning	-	-	-	-	46
# Hypergraphs	488	164	100	25	46
Source	[HS17a]	[HS17a]	[ASS18a]	[HSS18a]	[Sch+19a]

Table 2.2: Overview of the two *graph* benchmark sets used in our experiments. The last row gives the reference for each benchmark set.

	Benchmark Set	
	F	G
Complex Networks	21	-
DIMACS Challenge	-	17
Source	[MSS14]	[Bad+13]

[LW98], that a large fraction of all hyperedges in VLSI hypergraphs are actually graph edges (i.e., $|e| = 2$). This can also be seen in Figure 2.3 for the hypergraphs derived from the ISPD98 and the DAC 2012 benchmark sets. Note that the plots showing the average and median vertex degrees and net sizes use a log-scale on the y-axis for all except these two classes.

> The plots also show that reporting *average* net sizes can be misleading. An average net size of 4 might give the impression that we deal with hypergraphs containing mostly small hyperedges, although – in fact – more than half of all nets are actually just plain graph edges.

With the work of Çatalyürek and Aykanat [ÇA99], hypergraphs derived from sparse matrices became of interest to the partitioning community. The fact that the structure of a matrix highly depends on the actual application area lead to our decision to sample a large number of instances from the entire SuiteSparse Matrix Collection. Boolean formulae initially sparked our interest because they were mentioned alongside matrices and logic circuits as a third context for hypergraph partitioning in the work

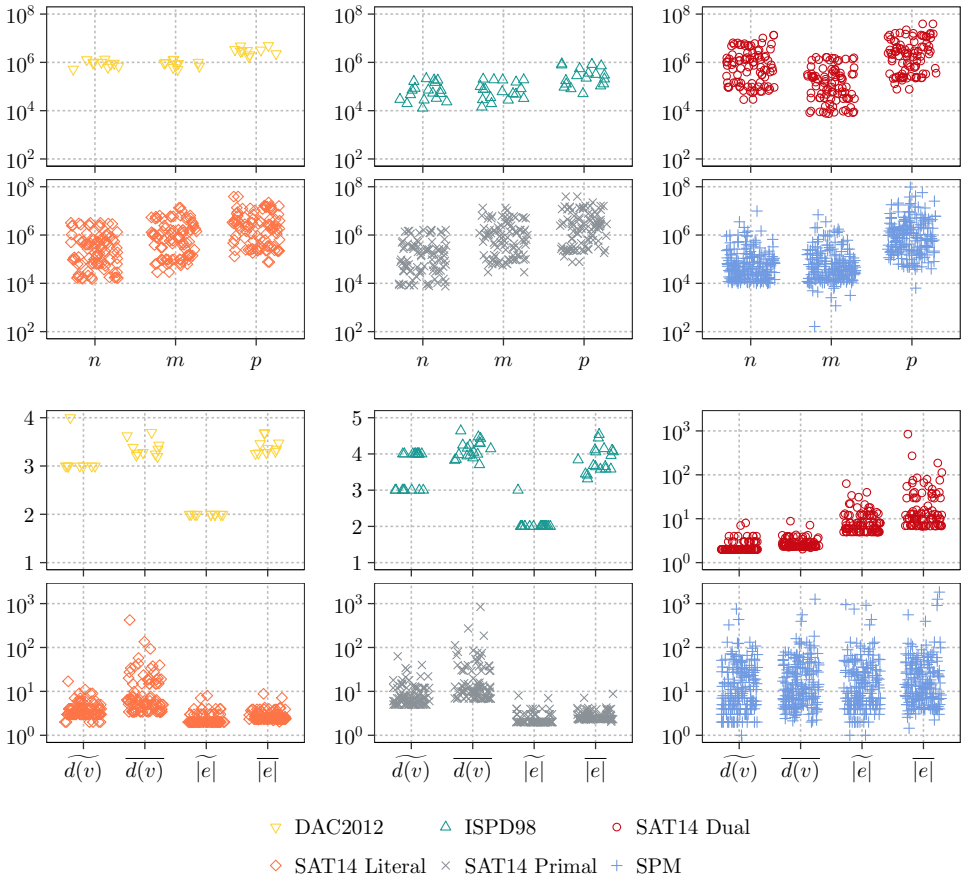


Figure 2.3: Basic properties (number of vertices n, number of nets m, number of pins p, average vertex degree $\overline{d(v)}$, median vertex degree $\widetilde{d(v)}$, average net size $\overline{|e|}$, median net size $\widetilde{|e|}$) of the hypergraphs in the main benchmark set A.

of Papa and Markov [PM07]. While the authors described the literal representation, we were made aware of both the primal and the dual representation though the work of Mann and Papp [MP14; MP17]. With the goal in mind to develop algorithms that are able to compute solutions of very high quality for a wide spectrum of hypergraph partitioning problems, we therefore assembled benchmark set A as described above.

Benchmark Subsets. We additionally use three subsets of benchmark set A. Based on an experiment to estimate the number of hypergraphs necessary to produce the same qualitative results as the entire benchmark set, benchmark set B was initially chosen in Ref. [Sch+16a] to contain the 10 largest ISPD98 VLSI hypergraphs, 30 randomly chosen SAT hypergraphs (literal representation), and 60 randomly chosen

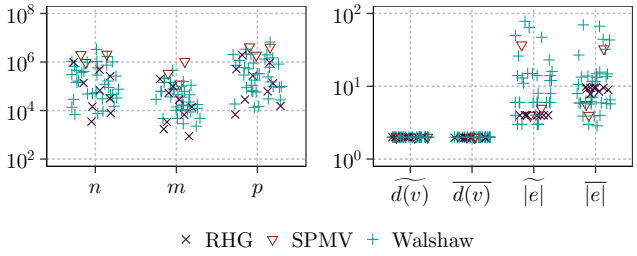


Figure 2.4: Basic properties (number of vertices n, number of nets m, number of pins p, average vertex degree $\overline{d(v)}$, median vertex degree $\widetilde{d(v)}$, average net size $\overline{|e|}$, median net size $\widetilde{|e|}$) of the edge partitioning hypergraphs in benchmark set E.

sparse matrix hypergraphs. In order to incorporate more recent VLSI circuits and more common SAT models, it was then extended in Ref. [HS17a] to also contain the five smallest DAC2012 circuits, as well as the primal and dual representations of each literal SAT hypergraph. For the experiments presented in this work, we remove SPM hypergraph `IMDB` from the benchmark set, because its partitioning time is considerably larger than all other hypergraphs from set B. Benchmark set C was composed in Ref. [ASS18a] such that it included hypergraphs from all instance classes and such that all hypergraphs could be partitioned within the given time limit of eight hours. For some extensive and long running parameter tuning experiments, we additionally use a small subset consisting of 25 hypergraphs (benchmark set D). It was introduced in Ref. [HSS18a] and was also used as parameter tuning benchmark set in Ref. [ASS18a]. Benchmark set D was chosen to contain five small to medium-sized hypergraphs from each class except DAC2012, for which all instances were considered too large.

Case Study on Edge Partitioning. In Section 6.4, we present a case study on graph *edge* partitioning (i.e., the task to partition the edge set of a graph into roughly equally-sized blocks) – a problem that can be solved via hypergraph partitioning. The case study is based on a conference paper jointly written with Christian Schulz, Daniel Seemaier, and Darren Strash [Sch+19a]. In the evaluation, we use a benchmark set of 46 hypergraphs which are derived from a set of benchmark graphs. More precisely, we use all instances of the Walshaw standard graph partitioning benchmark [SWC04], SPMV graphs [Li+17], and random hyperbolic rhgX graphs. SPMV graphs are bipartite locality graphs for sparse matrix vector multiplication (SPMV), which were used in the evaluation of Li et al. [Li+17]. Given a $n \times n$ matrix M (in our case the adjacency matrix of the corresponding graph), an SPMV graph corresponding to an SPMV computation $Mx = y$ consists of $2n$ vertices representing the x_i and y_i vector entries and contains an edge (x_i, y_j) if x_i contributes to the computation of y_j, i.e., if $M_{ij} \neq 0$. The rhgX graphs were chosen since their degree distributions follow a power law (and they are thus targeted by edge partitioning techniques). They are generated using KaGen [Fun+18] with a power law exponent of 2.2 and an average degree of 8.

Each of these graphs is transformed into a hypergraph as follows: The hypergraph

Table 2.3: The 85 instances of the DIMACS Implementation Challenge on Graph Clustering and Graph Partitioning used in our experiments as benchmark set G. The table is adapted from Ref. [Sch13b, p.139]. Note that we excluded graph `uk-2007-05` because it could not be partitioned by any partitioner on our system.

Graph	Values of k				
hugebubbles-00010	4	32	64	256	512
hugetric-00000	2	4	32	64	256
er-fact1.5-scale23	16	32	64	128	256
kron_g500-simple-logn17	2	4	8	16	32
kron_g500-simple-logn21	64	128	256	512	1024
delaunay_n15	8	16	32	64	128
coAuthorsCiteseer	4	8	16	32	64
asia.osm	64	128	256	512	1024
great-britain.osm	32	64	128	256	1024
M6	2	8	32	128	256
NLR	8	32	128	256	512
AS365	64	128	256	512	1024
auto	64	128	256	512	1024
rgg_n_2_18_s0	8	16	32	64	128
G3_circuit	2	4	32	64	256
kkt_power	16	32	64	256	512
nlpkkt160	4	8	16	32	64

contains a vertex for each graph edge and a hyperedge for each graph node. The pins of a hyperedge are those vertices that correspond to the graph edges incident to the graph node that is represented by the hyperedge. More details on the edge partitioning problem will be given in Section 6.4. Basic properties of these hypergraphs are shown in Figure 2.4.

Graph Partitioning Experiments. To evaluate the performance of our algorithm in the context of traditional graph partitioning, we use the 21 large web graphs and social networks used in the work of Meyerhenke et al. [MSS14] (benchmark set F), and the graphs used in the final evaluation of the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering [Bad+13] (benchmark set G). We excluded graph `uk-2007-05` from benchmark set G, because no algorithm involved in our experimental evaluation was able to partition that graph on our system. Basic properties of benchmark sets F and G are shown in Figure 2.5.

2.6.2 System and Setup

System. All experiments presented in this dissertation were performed on the *bwUniCluster* maintained by the Steinbuch Centre for Computing (SCC) at the KIT.

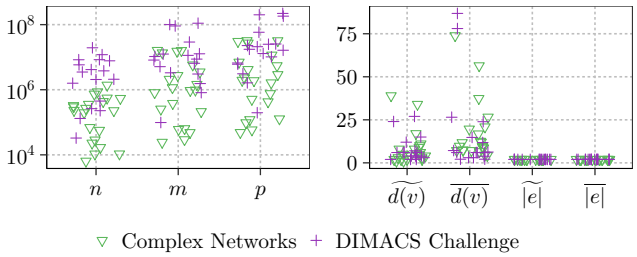


Figure 2.5: Basic properties (number of vertices n, number of nets m, number of pins p, average vertex degree $\overline{d(v)}$, median vertex degree $\widetilde{d(v)}$, average net size $\overline{|e|}$, median net size $\widetilde{|e|}$) of the graphs in benchmark set F (complex networks) and benchmark set G (DIMACS graphs).

Unless mentioned otherwise, we ran all algorithms exclusively on a single core of one of the 512 "thin" compute nodes of this high-performance computing system. Each of these nodes has two Intel Xeon E5-2670 Octa-Core (Sandy Bridge) processors clocked at 2.6 GHz, 64 GB main memory, 20 MB L3- and 8x256 KB L2-Cache and runs Red Hat Enterprise Linux 7.4. When compiling algorithms from source code, we used g++-9.1 with flags `-O3 -march=native`.

Partitioning Setup. Unless mentioned otherwise, all hypergraphs and graphs are partitioned with an allowed imbalance of $\varepsilon = 0.03$ into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks. For each value of k, a k-way partition is considered to be *one* test instance, resulting in a total of 3 416, 1 148, 700 175, 322, and 147 instances for benchmark sets A, B, C, D, E, and F, respectively. For benchmark set G, we use the values of k that were used in the DIMACS challenge (see [Sch13b, p. 139]). Here, each graph is partitioned into 5 different values of k out of $k \in \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ – resulting in a total of 85 instances for set G. The actual numbers of blocks each graph is partitioned into are shown in Table 2.3. We restrict ourselves to an imbalance of $\varepsilon = 0.03$, because the first and thus simplest version of KaHyPar already performed well on partitioning problems with 1% and 10% imbalance [Sch+16a]. Unless mentioned otherwise, all partitioning runs are repeated ten times with different random seeds.

Measured Performance Metrics. In all experiments, we measure the quality of the computed partitions (i.e., the cut-net metric $f_c(\Pi)$ or the connectivity metric $f_\lambda(\Pi)$), the imbalance between the block sizes, and the running time of the algorithm. When averaging over multiple runs with different seeds, we use the arithmetic mean.

2.6.3 Aggregate Performance Numbers

In order to concisely present the results of different configurations of our algorithms over a benchmark set $\mathcal{I}$ containing $s := |\mathcal{I}|$ instances, we use the *geometric mean* $\overline{x}_g := \sqrt[s]{\prod_{i=1}^{s} x_i}$ to average solution quality or running times over all benchmark

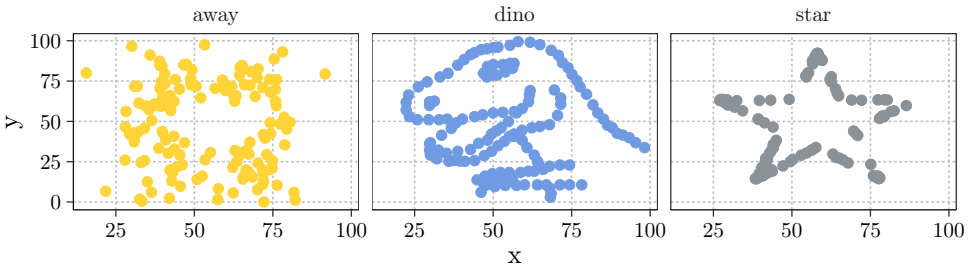


Figure 2.6: Visualization of three examples from the Datasaurus data set [Cai16; LD17; MF17] – a modern variation of Anscombe's quartet [Ans73]. Each of these distinct data sets consists of 142 (x, y) pairs and has identical summary statistics up to two decimal places (e.g., means: $\overline{x} = 54.26$, $\overline{y} = 47.83$ and standard deviations: $\text{sd}_x = 16.76$, $\text{sd}_y = 26.93$), yet looks entirely different when plotted.

instances. Since it holds that $\overline{x}_g = \exp(\frac{1}{s}\sum_{i=1}^{s} \ln x_i)$, the geometric mean can be interpreted as a log-normalized average [HB15]. It is regarded as a good choice for summarizing skewed data [McG12, p. 229], because it is less sensitive to few large values than the arithmetic mean, and thus more robust to outliers. Since both solution quality and running times can vary by up to orders of magnitude depending on the instance to be partitioned and the algorithm that computes the partition, it is therefore more appropriate than the traditional arithmetic mean in our setting.

When comparing a baseline configuration A with a new algorithm configuration B, we use the geometric mean solution quality $\overline{x}_g^A$ of A as a reference and report the *relative percentage change*

$$-\left(\frac{\overline{x}_g^B - \overline{x}_g^A}{\overline{x}_g^A}\right) \cdot 100. \tag{2.2}$$

Thus, if the relative percentage change is positive, we say that the cut/connectivity of configuration B is $x\%$ *less* than the cut/connectivity of configuration A, or that the *improvement* of configuration B over the reference configuration A is $x\%$.

> Note that relative changes are calculated in an asymmetric way, i.e., the relative percentage change differs depending on which of the algorithms is used as a reference point. If algorithm A computes a cut of size 500 and algorithm B computes a cut of size 400, then the cut of algorithm B is $-(400-500)/500 \cdot 100 = 20\%$ *less* then the cut of algorithm A (or the improvement of algorithm B over algorithm A is 20%). Similarly, given the relative difference of $-(500-400)/400 \cdot 100 = -25\%$, the cut of algorithm A is 25% *larger* than the cut of algorithm B. This asymmetry could be addressed by using symmetric indicators of relative difference (see, e.g. Ref. [TVV85]). However, since these indicators are less intuitive and since we only report improvements over a reference configuration, we employ the relative percentage change as defined above.

However, we would like to point out the fact that "any measure of the mean value of data is misleading when there is large variance" [FW86]. Furthermore, examples such as Anscombe's quartet [Ans73] or the Datasaurus collection of data sets [Cai16; LD17; MF17] – three of which are shown in Figure 2.6 – highlight the importance of graphical representations as well as the fact that evaluating data solely based on summary statistics is insufficient and may lead to misleading conclusions. Hence, we mostly employ different means of visualization (i.e., the plots described in the next sections) to explore experimental results.

2.6.4 Evaluating Solution Quality with Performance Profiles

Whenever we compare the solution quality of partitions computed by multiple algorithms in detail, we use the *performance profiles* introduced by Dolan and Moré [DM01; DM02], which are widely adopted for evaluating the performance of optimization algorithms [GS16].

Performance Profiles. For a set of $\mathcal{P}$ algorithms and a benchmark set $\mathcal{I}$ containing $|\mathcal{I}|$ instances, the *performance ratio* $r_{p,i}$ relates the cut computed by partitioner p for instance i to the smallest minimum cut of *all* algorithms, i.e.,

$$r_{p,i} := \frac{\text{cut}_{p,i}}{\min\{\text{cut}_{p,i} : p \in \mathcal{P}\}}. \tag{2.3}$$

The *performance profile* $\rho_p(\tau)$ of algorithm p is then given by the function

$$\rho_p(\tau) := \frac{|\{i \in \mathcal{I} : r_{p,i} \leq \tau\}|}{|\mathcal{I}|}, \tau \geq 1. \tag{2.4}$$

For connectivity optimization, the performance ratios are computed using the connectivity values $\lambda_{p,i}^{-1}$ instead of the cut values. The value of $\rho_p(1)$ corresponds to the fraction of instances for which partitioner p computed the best solution, while $\rho_p(\tau)$ is the probability that a performance ratio $r_{p,i}$ is within a factor of τ of the best possible ratio. Note that since performance profiles only allow to assess the performance of each algorithm relative to the *best* algorithm, the $\rho(1)$ values cannot be used to rank algorithms [GS16] (i.e., to determine which algorithm is the second best etc.). In our experimental analysis, the performance profile plots are based on the *best* solutions (i.e., *minimum* connectivity/cut) each algorithm found for each instance, unless mentioned otherwise. Furthermore, we choose parameters $r_{\text{inf}} \gg r_{p,i}$ for all p, i and $r_{\text{timeout}} \gg r_{\text{inf}}$ such that a performance ratio $r_{p,i} = r_{\text{inf}}$ if and only if algorithm p computed an infeasible solution for instance i, and $r_{p,i} = r_{\text{timeout}}$ if and only if the algorithm could not compute a solution for instance i within the given time limit. In our performance profile plots, performance ratios corresponding to *infeasible* solutions will be shown on the ✗-tick on the x-axis, while instances that could not be partitioned within the time limit are shown implicitly by a line that exits the plot below $y = 1$.

Performance profiles have been shown to be both insensitive to changes in the results on a small number of instances and to be largely unaffected by small changes in the

results over many problems [DM02], which makes them a good tool for comparing the performance of different partitioning algorithms.

Handling Skewed Data. In case the performance ratios are heavily right-skewed, we divide the performance profile plots into three segments with different ranges for parameter τ to reflect various areas of interest. The first segment highlights small values ($\tau \leq 1.1$), while the second segment contains results for all instances that are up to a factor of $\tau = 2$ worse than the result of the best algorithm. Both segments use a linear scale on the x-axis. The last segment contains all remaining ratios, i.e., instances for which some algorithms performed considerably worse than the best algorithm, instances for which algorithms produced infeasible solutions, and instances which could not be partitioned within the given time limit. Since the right-skewed τ values are plotted in the last segment, we use the approach of Tukey's Ladder of Powers [Tuk57; Tuk77b] to find a power transformation that makes the data fit the normal distribution as closely of possible (see also Ref. [McG12, p. 244]). More precisely, we determine a value φ to be used as the exponent in the transformation $\xi(x) = x^{\varphi}$. To do so, we use the `transformTukey` function from the `rcompanion` R package [Man16], which chooses φ such that it minimizes the test statistic of the Anderson-Darling [AD52] test for normality. The resulting exponent used to transform the τ values shown in the last segment is $\varphi := -0.25$. To prevent the transformed values from being reversed due to φ being negative, we employ the transformation $\xi(x) = -(x^{\varphi})$ to preserve the order of the τ values after transformation [McG12, p. 229].

Example. Figure 2.7 (left) shows the performance profiles of three hypothetical partitioning algorithms. We see that out of all three algorithms, algorithm A performs better than both algorithm B and C. It computes the best solutions for around 80% of all benchmark instances. Furthermore, the solution quality of algorithm A is within a factor of 1.3 of the best algorithm for almost all instances. However, we also see that algorithm A could not partition 10% of all instances within the time limit, since its performance profile leaves the plot at $y = 0.90$. Looking at the performance profile of algorithm B, we see that it computed the best solution for around 15% of all benchmark instances, but also computed imbalanced and thus infeasible solutions for more than 30% of all instances. Algorithm C performs worst. For only in around 8% of all benchmark problems, was it able to compute solutions that were within a factor of 2 from the solution of the best algorithm.

2.6.5 Visualizing the Evolution of Solution Quality over Time

In Chapter 5, we present our memetic n-level hypergraph partitioning algorithm. For a predefined amount of time, this algorithm evolves a population of individuals (i.e., partitions) to compute high-quality solutions. In the experimental evaluation, we therefore use *convergence plots* [SS12] to visualize the evolution of solution quality over time.

Convergence Plots. We start by explaining how to compute the data for a single instance i, i.e., a k-way partition of a hypergraph H. Whenever an algorithm computes

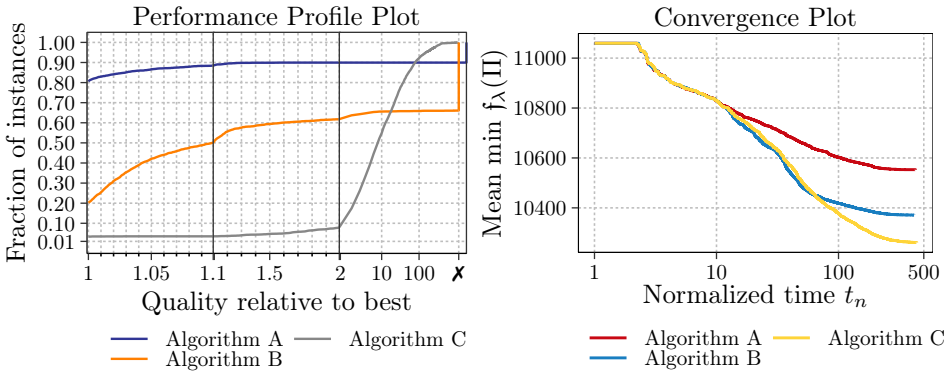


Figure 2.7: Example of a performance profile plot (left) and a convergence plot (right) comparing the solution quality of three different algorithms.

a partition that improves the solution quality, it reports a pair $(t,\ \lambda^{-1})$, where the timestamp t is the current elapsed time and λ^{-1} is the computed connectivity metric. For r repetitions with different seeds s, these r sequences T_s^i of pairs are merged into one sequence T^i of triples (t, s, λ^{-1}), which is sorted by the timestamp t. Since we are interested in the *evolution* of the solution quality, we compute the sequence $T_{\min}^i$ representing *event-based* average values. We start by computing the average connectivity $\overline{c}$ and the average time $\overline{t}$ using the first pair $(t,\ \lambda^{-1})$ of all r sequences T_s^i and insert $(\overline{t}, \overline{c})$ into $T_{\min}^i$. We then sweep through the remaining entries (t, s, λ^{-1}) of T^i. Each entry corresponds to a partition computed at timestamp t using seed s that improved the solution quality to λ^{-1}. For each entry we therefore replace the old connectivity value of seed s that took part in the computation of $\overline{c}$ with the new value λ^{-1}, recompute $\overline{c}$ and insert a new pair $(t, \overline{c})$ into $T_{\min}^i$. $T_{\min}^i$ therefore represents the evolution of the average solution quality $\overline{c}$ for instance i over time. In a final step, we create the *normalized* sequence $N_{\min}^i$, where each entry $(t, \overline{c})$ in $T_{\min}^i$ is replaced by $(t_n, \overline{c})$ where $t_n := t/t_i$ and t_i is the average time a baseline algorithm needs to compute a k-way partition of H.

Average values over *multiple instances* are then obtained as follows: All sequences $N_{\min}^i$ of pairs $(t_n, \overline{c})$ are merged into a sequence $N_{\min}$ of triples $(t_n, \overline{c}, i)$, which is then sorted by t_n. The final sequence $S_{\mathcal{G}}$ presenting event-based *geometric* mean values is then computed as follows: We start by computing the average normalized time $\overline{t_n}$ and the geometric mean connectivity $\mathcal{G}$ over all instances i using the first value of all $N_{\min}^i$ and insert $(\overline{t_n}, \mathcal{G})$ into $S_{\mathcal{G}}$. We then sweep through the remaining entries of $N_{\min}$. For each entry $(t_n, \overline{c}, i)$, we replace the old connectivity value of i that took part in the computation of $\mathcal{G}$ with the new value $\overline{c}$, recompute $\mathcal{G}$ and insert $(t_n, \mathcal{G})$ into $S_{\mathcal{G}}$. The sequence $S_{\mathcal{G}}$ therefore represents the evolution of the solution quality averaged over all instances and repetitions. The plots use a log-scale on the x-axis.

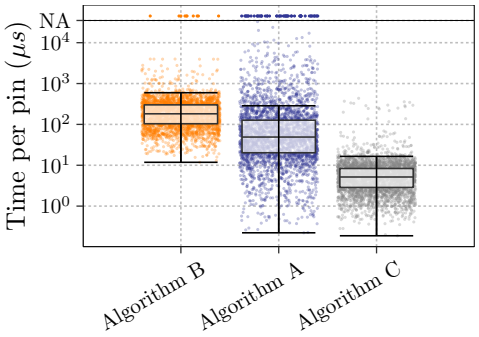


Figure 2.8: Example of the combination of a scatter plot and a box plot used to visualize the running times of different partitioning algorithms.

Example. Figure 2.7 (right) shows an example of a convergence plot comparing the evolution of solution quality over time for three algorithms. We see that while all algorithms start off with the same solution quality, their convergence behavior differs. Algorithm A converges to solutions that are worse on average than the solutions of both algorithm B and algorithm C. Furthermore, we see that while the average solution quality of algorithm C is sometimes slightly worse than the average solution quality of algorithm B, its final solutions are better than those of algorithm B, on average.

2.6.6 Visualizing Running Times

Depending on the structure and the size of the input hypergraphs, as well as on the number k of blocks, the time it takes a multi-level algorithm to partition an instance can sometimes vary by several orders of magnitude. Furthermore, as we will see in the final evaluation of a large set of partitioning algorithms in Chapter 6, the running time of different algorithms also varies by orders of magnitude. We therefore use a combination of a scatter plot (which shows the running time for each particular instance) and a Tuckey box plot [Tuk77a] to visualize running times. The box hereby shows the 1^{st} and 3^{rd} quartiles, the line inside the box indicates the median, and the whiskers indicate the location of the smallest/largest data point still within 1.5 times the inter-quartile range (IQR) of the lower/upper quartile. All data points lying below or above the whiskers are considered to be outliers.

Since the running times are right-skewed, we use a log-scale on the y-axis. Additionally, for each instance, we normalize the running time by the number of pins of the hypergraph. In case the experimental results contain instances that could not be partitioned because an algorithm aborted or did not finish within the given time limit, we visualize these instances above a vertical line labeled “NA”. Note that these instances are excluded from the computation of the box plot in order to avoid distorted results.

Example. An example is shown in Figure 2.8. We see that the median running times of all three algorithms differ. Furthermore, algorithm C is significantly faster than the other two algorithms and is the only algorithm that could partition all instances.

2.6.7 Testing for Statistical Significance

To determine whether or not the differences in solution quality between partitioning algorithms are statistically significant, we follow the guidelines proposed by Demsar [Dem06] and García et al. [GH08; Gar+10]. For comparisons involving two algorithms, we use the Wilcoxon signed rank test [Wil45; Pra59]. When performing all pairwise comparisons between a set of algorithms, we use the Friedman test [Fri37; Fri40] with Iman Davenport modification [ID80] and the Bergmann-Hommel procedure [BH88] to correct the p-values for multiple testing. Wilcoxon signed rank tests are performed using the R package `coin` [Hot+08], while multiple comparisons involving a set of algorithms are done using the `scmamp` package [CS16]. In all cases, we use a 1% significance level. Furthermore, the statistical tests are only based on instances for which *all* algorithms were able to compute a solution, and we disqualify imbalanced and thus infeasible results by setting the corresponding solution quality to a value larger than the quality of the worst feasible solution.

We note that there is an ongoing controversy regarding statistical significance testing and the use of p-values [Nuz14; WL16; MWS17; BD19; KW19] as well as which test procedure to use in which setting [BCM16; BD19]. Therefore, we try not to overstate the results of significance tests and use them only in combination with other means of evaluating experimental data, i.e., the techniques presented in this section.

Chapter 3

3 A Brief History of Hypergraph Partitioning

"Partitioning has been an active area of research for at least a quarter of a century."

— Frank M. Johannes [Joh96], 1996

The Structure of Scientific Revolutions. The history of hypergraph partitioning is a prime example for the structure of scientific revolutions as proposed by Kuhn [Kuh62], who argues that scientific progress can be broken up into four distinct phases. In the first phase of *pre-paradigmatic science*, which only occurs once, a vast number of different approaches is developed concurrently. Once the research community agrees on a common methodology, terminology, and on an appropriate approach to address the problem, it enters the phase of *normal science*. Here, research is guided by a central paradigm that allows an in-depth confrontation with the problem at hand and yields a productive period of incremental improvements. The phase of normal science continues until either more and more problems with the current paradigm start to build up or a revolutionary discovery is made that cannot be reconciled with the current paradigm, which throws science into a *crisis period*. This then marks the beginning of the phase of *revolutionary science*, in which a reexamination of the foundations of the field may culminate in a paradigm shift: A new paradigm supersedes the old one. Afterwards, the scientific community returns to the phase of normal science under the new paradigm.

The Iterative Improvement (IIP) Paradigm. If we look at the partitioning problem from the perspective of the (V)LSI design community, the stage of pre-paradigmatic science starts with the early studies on circuit partitioning in the 1960s (see, e.g., the first surveys of Kodres [Kod72] and Donath [Don88]). With the seminal work of Kernighan and Lin [KL70] on graph partitioning, the extension to hypergraph partitioning by Schweikert and Kernighan [SK72], and the introduction of the linear-time FM algorithm [FM82] then followed a phase of normal science, in which flat, single-level algorithms were continuously developed and successively improved. Although several constructive techniques [HK91a; CHK92; HMV92; HK92b] were proposed until 1995, the literature was mostly dominated by iterative improvement algorithms for several reasons [AK95c, p.16]. Perhaps most importantly, the idea of successively improving a starting solution by moving vertices across the partition boundary is very intuitive and often leads to algorithms that are relatively simple to describe and to implement (although, as we will see in the following sections, the devil is in the details). Furthermore, IIP algorithms are largely independent of the objective

function to be optimized and allow for the integration of additional constraints such as fixed vertices [CL98]. Due to their simplicity, they are also reasonably fast, which permits the use of repeated executions to get the best out of several solutions, and the application as local search algorithms in more advanced metaheuristics such as evolutionary algorithms. Lastly, due to their iterative nature, they allow for a controllable trade-off between solution quality and running time [CL98].

After the introduction of the FM algorithm [FM82], academic research mostly focused on improving its erratic behavior [NOP87; SC88; CW91]. The inherent shortsightedness when choosing the next vertex move and the lack of an advanced tie-breaking scheme for vertices that yield the same improvement, made multiple restarts inevitable in order to achieve good solution quality [HK97; HB97]. Enhancements then focused on strategies that improved the tie-breaking scheme [Kri84; HHK95a; DD96a], relaxed the locking mechanism [DA94; Hof94; CCH98] or the feasibility constraint for intermediate solutions [DT97], improved locality by encouraging tightly connected clusters of vertices to move together [DD96b; Con+97a], or embedded the FM algorithm into metaheuristics [KGV83; SR89a; SR89b; BM94; BM98; AV00]. Furthermore, the initial bipartitioning approach was extended to k-way partitioning in various ways [San89; CSZ97a; CL98]. While for graphs *every* move of a boundary node has an immediate effect on the edges in the cut-set, single vertex moves in hypergraphs may not affect the cut-set at all, because large hyperedges are likely to have more than one pin in multiple blocks of the partition. This, in turn, motivated several algorithms that work on other hypergraph representations such as the dual hypergraph [CLS94] (nets of the hypergraph correspond to vertices in the dual, and for each vertex v of the original hypergraph a net with pins $\mathrm{I}(v)$ is added to the dual) or the net intersection graph [Kah89] (nets of the hypergraph correspond to nodes in the graph and there is an edge between two nodes if the corresponding nets share a vertex).

A Paradigm Shift Towards Two-Level Algorithms. The observed performance deterioration of FM-based algorithms for large hypergraphs [HHK95b], as well as for instances with large net sizes [KK99] or small vertex degrees [GB83; Bui+87; Bui+89][Len90, p. 273][Çat99, p.15] then marked the beginning of a crisis, which lead to a paradigm shift towards two-level approaches. By creating a coarse representation of the input hypergraph through the contraction of highly-connected vertices, these algorithms increase its density and significantly reduce both the *exposed hyperedge weight* [IKS75] as well as the *sizes* of the hyperedges [Saa95]. This significantly reduces the complexity of the search space. Increased density and smaller net sizes furthermore allow move-based IIP algorithms to explore the search space more effectively and increase their ability to escape from local optima, while the reduction in exposed hyperedge weight makes it easier to compute an initial partition with good quality [Kar03, p. 146]. In the two-level setting, an initial solution is first computed and improved on the coarse instance, and, after projecting the solution back to the input hypergraph, then refined once again on the original instance.

The Multi-Level Revolution. "A breakthrough in min-cut partitioning came in 1997 when [19] [i.e., [Kar+97a]] and [2] [i.e., [AHK97]] validated the multilevel partition-

ing paradigm for hypergraphs with their highly successful implementations" [CKM00a]. The multi-level scheme [BS93; BJ93; CS93; HB95; HL95] thus constitutes a revolutionary invention that, within only a couple of years, completely superseded the two-level approach. By allowing the coarsening process to proceed more slowly, the multi-level approach creates a *hierarchy* of successively coarser hypergraphs. During uncoarsening, this then allows refinement algorithms to operate on *multiple scales* of the partitioning problem. On the coarsest levels, single-vertex moves correspond to the movement of entire clusters of vertices of the input hypergraph and therefore allow for rather global optimizations, whereas refinement on finer levels of the hierarchy gives an increasingly more local view for more and more fine-grained improvements. "This permits the refinement algorithm to avoid bad local minima ("basins of attraction") via large steps at high levels, and also find a good final solution via refinement at the low levels" [Alp96, p. 179], making the multi-level paradigm highly effective for computing high-quality partitions of very large hypergraphs.

Chapter Overview. In the following, we will trace the lineage of ideas through almost 50 years of hypergraph partitioning history, starting with the Kernighan-Lin [KL70] and Fidducia-Mattheyses [FM82] algorithms in Sections 3.1 and 3.2 – variations of which are still employed in even the most sophisticated hypergraph partitioning systems today. Section 3.3 then covers the era of flat partitioning algorithms, before we turn to two- and multi-level algorithms in Section 3.4. Due to the large number of competing approaches that have been developed over time, Section 3.5 then presents a taxonomy of the state-of-the-art HGP systems that are still in use today, before we summarize the findings and extract some guiding principles for effective hypergraph partitioning algorithms in Section 3.6.

Disclaimer. The body of literature on hypergraph partitioning is *immense*, which is why we restrict ourselves to techniques that have been explicitly proposed for the standard hypergraph partitioning problem formulation as defined in Section 2.2, and omit contributions that are exclusively devised for partitioning VLSI circuits or FPGAs [NOP87; SC88; CH90; CMR90; Hul90; Hul91; Hag+92; YCL92; KBK93; RS93; WK93; Cho+94; Hag+94; KBZ94; KZB94; SK94; HK95; KB95; KAS97; LW98; CL99; EHS99; CL00b; CLW00; CWC01; CW02a; CW02b; WS02; CW03; Wu+03] and are therefore not applicable to the general setting. Furthermore, the presentation of related work differs from overview articles and surveys [Kod72; Don88; AK95c; MWW95; Joh96; Lie97; Kah98; CC00; KAV04; PM07; Kuc08; Bul+16], as well as related work sections of HGP-related dissertations [Alp96; Çat99; Lim00; Tri06; Lot16] in that we present a *historical overview* instead of an aggregation of similar concepts and techniques, since we believe that the historic perspective makes the scientific evolution of the field more comprehensible. Additionally, we refrain from reporting the conclusions of experimental evaluations regarding the comparisons of different algorithms, since most experiments were only performed on small, restricted benchmark sets and thus should be taken with a grain of salt. Lastly, we note that although this chapter presents a substantial amount of HGP-related research, we cannot claim it to be exhaustive.

Tabular Summaries. The algorithms discussed in this chapter are summarized in several tables. Table 3.1 on page 63 gives an overview over the flat, single-level algorithms presented in Section 3.3. Table 3.2 on page 88 then covers the two-level approaches, while Table 3.3 on page 89 summarizes the multi-level algorithms presented in Section 3.4.

3.1 The Kernighan-Lin (KL) Algorithm

In their seminal paper, Kernighan and Lin [KL70] describe what is now known as the first good heuristic for the graph bisection problem [AK95c]. The key observation underlying the KL algorithm forms the basis for most of today's iterative improvement heuristics.

Central Idea. After initially partitioning an unweighted graph into two subsets of equal size, *some* vertices are assigned to the wrong side of the bisection $\Pi_2 = \{V_1, V_2\}$. If it was possible to identify these subsets $X \subset V_1, Y \subset V_2$ of wrongly assigned vertices, then interchanging X and Y would transform the current solution to the *optimal* bisection $\Pi_2^* = \{V_1^*, V_2^*\}$, with $V_1^* := (V_1 \setminus X) \cup Y$ and $V_2^* := (V_2 \setminus Y) \cup X$. Unfortunately, the problem of identifying X and Y is as hard as the initial graph bisection problem [KL70], which is why KL-type algorithms try to find good approximations of these subsets that improve the current solution.

Gain. Changes in solution quality are captured by the *gain* $g(u)$ of moving a node u from its current block A to the other block B: $g(u) := \omega(\{(u,v) \mid v \in \Gamma(u) \cap B\}) - \omega(\{(u,v) \mid v \in \Gamma(u) \cap A\})$. The first term hereby corresponds to the weight of those incident edges that become *internal* after the move, while the second term accounts for the edges that become part of the cut-set. Thus, a node $v \in A$ that is more strongly connected to the nodes in B than to those in A has a positive gain and can therefore be considered a good candidate for a move. Since the KL algorithm is aimed at improving bisections (i.e., perfectly balanced bipartitions), it is necessary to move a node from B to A each time a node is moved from A to B. The gain of *swapping* two nodes $u \in A$, $v \in B$ thus changes the solution quality by $g(u,v) := g(u) + g(v) - 2\ \omega(\{(u,v) \mid u \in \Gamma(v)\})$.

Algorithm Outline. A pseudocode description is shown in Algorithm 3.1. Given an initial bisection, the algorithm proceeds in a series of *passes*. Each pass tries to determine a sequence of *pairwise* swaps that improves the current solution. To avoid thrashing, every node is allowed to be swapped exactly once per pass. Thus at the beginning of each pass all nodes are marked as unlocked. The algorithm then searches for the pair of unlocked nodes $u \in V_1$ and $v \in V_2$ with the highest gain (i.e., the largest decrease or smallest increase in solution cost). This pair along with the corresponding gain value is added to a sequence of temporary swaps and both nodes become locked so that they are not considered in the remaining steps of the pass. The swap is then simulated by updating the gain values of all unlocked neighbors of u and v accordingly, and the search process continues until all nodes are locked. At this point, the algorithm

Algorithm 3.1 : Kernighan-Lin Iterative Improvement for Graph Bisections

Input : Graph $G = (V, E, c, \omega)$
Input : Bisection $\Pi_2 = \{V_1, V_2\}$
Require : $|V_1| = |V_2| \wedge c \mapsto 1$ *// Algorithm assumes bisection of unweighted nodes*
1 **do** *// Perform series of passes*
2 compute initial gains *// for all vertex pairs ($u \in V_1, v \in V_2$)*
3 **while** not all vertices locked **do** *// Locking mechanism prevents thrashing*
4 find $(u \in V_1, v \in V_2)$ with highest gain $g(u, v)$ *// Gain can be negative*
5 swap(u, v) *// Temporarily switch blocks*
6 lock(u), lock(v) *// Each node is only allowed to move once*
7 update gain values of unlocked neighbors *// to reflect the swap*
8 choose j to maximize $g_{\max} := \sum_{i=1}^{j} g(u_i, v_i)$ *// Find best sequence of swaps*
9 **if** $g_{\max} > 0$ **then** *// Improvement found*
10 $X := \bigcup_{i=1}^{j} \{u_i\}$, $Y := \bigcup_{i=1}^{j} \{v_i\}$ *// Sets of nodes to be swapped*
11 $V_1 := (V_1 \setminus X) \cup Y$, $V_2 := (V_2 \setminus Y) \cup X$ *// Swap nodes to improve quality*
12 unlock all nodes
13 **while** $g_{\max} > 0$ *// Ensure strictly increasing solution quality*
Output : Improved bisection $\Pi_2 = \{V_1, V_2\}$

has completely exchanged both blocks of the bisection and thus the final cut size corresponds to the solution quality of the initial bisection. In order to find the subset of swaps with maximum improvement, the sequence of temporary swaps is traversed and the smallest index j such that the partial sum $g_{\max} := \sum_{i=1}^{j} g(u_i, v_i)$ is maximum, is computed. If $g_{\max} > 0$, the solution quality is improved by exchanging the subsets $X := \bigcup_{i=1}^{j} \{u_i\}$ and $Y := \bigcup_{i=1}^{j} \{v_i\}$. The improved bisection is then used as a starting partition for the next pass. Otherwise, if $g_{\max} = 0$, the algorithm terminates because the current partition constitutes a local minimum.

> During a pass, the algorithm does not stop when a *single* pair of nodes yields a negative improvement. By temporarily worsening the solution quality, the KL algorithm is therefore able to climb out of local minima to some extent. At the same time, it ensures a strict increase in solution quality between passes, because swaps are only executed at the end of a pass if the overall gain is positive. Thus between passes, it can still be seen to act greedily.

Running Time Complexity. Consider a graph $G = (V, E)$ with $|E| = m$ edges and $n = 2t$ nodes that is partitioned into $\Pi_2 = \{V_1, V_2\}$ with $|V_1| = |V_2| = t$. In the following, we reiterate Dutt's analysis [Dut93] of the KL algorithm and consider a single pass of the algorithm. Using an adjacency matrix as graph data structure, calculating the initial gain values for all nodes can be done in $\Theta(n^2)$ time. In order to select the next pair to be exchanged, Kernighan and Lin [KL70] propose to sort the unlocked nodes of each block by their gain values. In iteration k, this takes $\Theta((t-k)\log(t-k))$

time. Afterwards, the sorted lists are scanned to find the next pair to be exchanged. While this takes $\mathcal{O}(t^2)$ time in the worst case, Kernighan and Lin [KL70] assume that in practice only a few node-pairs have to be considered and thus conclude that this step takes $\Theta(t-k)$ time in iteration k. After tentatively swapping the node-pair, the gain values of the remaining unlocked nodes are updated in $\Theta(t-k)$ time. Thus, summing over all $t = n/2$ iterations yields a running time complexity of $\Theta(n^2 \log n)$ per pass, since determining the subsets X and Y to be exchanged at the end of each pass can be done in $\Theta(n)$ time. However, since in the worst case all node-pairs have to be scanned in order to find the next pair to be swapped, the worst case running time is actually $\mathcal{O}(n^3)$. By using a neighborhood search technique to limit the number of scanned node-pairs and by using balanced binary search trees to store and maintain the gain values Dutt [Dut93] improved the running time to $\mathcal{O}(m \max(\log n, \Delta_v))$ per pass. While in theory up to m passes can be necessary to converge to a local optimum for unweighted graphs [AK95c], empirical evidence indicates that, given a good initial bisection, a small constant number of passes suffices [KL70; DK85; Dut93].

Extensions. The KL algorithm can be extended in several ways. Kernighan and Lin [KL70] proposed extensions to deal with unequally sized bipartitions, weighted nodes, and the refinement of k-way partitions.

A graph initially partitioned into two blocks of unequal size n_1 and n_2 can be refined by restricting the number of pairs to be exchanged per pass to $\min(n_1, n_2)$. If the graph should contain *at least* n_1 nodes in one and *at most* n_2 nodes in the other block, $n_2 - n_1$ unconnected dummy vertices are added to V_1. After running the KL algorithm on the modified graph, the dummy nodes can be discarded.

To incorporate node weights, each node v with a weight of $c(v) > 1$ is replaced with a clique of $c(v)$ nodes connected by edges with a sufficiently high edge weight such that they won't be cut during the execution of the algorithm.

While the original algorithm is defined to improve bisections, it can also be used to refine a given k-way partition by repeatedly running it on all pairs of blocks (i, j), where either block i or j has changed since the pair was last considered.

Furthermore, it can be easily adapted to work with fixed vertices (i.e, vertices that are preassigned to a specific block [Alp+99]) by marking them as locked at the beginning of each pass such that they will not be considered for swaps [Len90, p. 264]. Schweikert and Kernighan [SK72] extended the KL algorithm to work with hypergraphs and the improvements proposed by Dutt [Dut93] for graphs are conjectured to also apply to hypergraph partitioning [HHK95a; Alp96].

Limitations. Due to the fact that the algorithm is designed for bisections, the extensions for bipartitions of variable size are somewhat limited. For example, the algorithm is not able to prefer more balanced to less balanced partitions with the same cut [Len90, p. 265]. Moreover, the solution to handle weighted vertices is not deemed viable for most partitioning problems due to the increased graph size [Len90, p. 265] and the fact that some clique vertices might still end up in different blocks of the bisection [BS11, p. 45]. The main disadvantage of the KL heuristic, however, is its computational complexity, which is why most graph and hypergraph partitioning

algorithms use some variation of the linear time algorithm proposed by Fiduccia and Mattheyses [FM82], which we describe next.

3.2 The Fiduccia-Mattheyses (FM) Algorithm

While the KL algorithm was the first successful partitioning heuristic, the FM algorithm [FM82] forms the basis for a wide range of bipartitioning algorithms [HB97], and is employed almost universally in GP and HGP implementations [CS03, p. 80]. Being specifically designed for hypergraph partitioning, it builds on the KL extension of Schweikert and Kernighan [SK72]. Similar to KL, the FM algorithm performs a series of passes that iteratively improve solution quality, while temporarily allowing to worsen the partitioning objective within a pass. However, instead of swapping pairs of vertices, it only moves *one* vertex at a time. This allows the algorithm to handle nonuniform vertex weights and bipartitions with a certain imbalance between the weight of both blocks. Furthermore, and more importantly, Fiduccia and Mattheyses [FM82] show that their algorithm permits an implementation that takes $\mathcal{O}(p)$ time per pass by carefully analyzing the effects of a vertex move on the gains of neighboring vertices and by using bucket priority queues to manage vertex gains.

Before discussing algorithmic details, we first generalize the concept of move gains from graphs to hypergraphs along the lines of Schweikert and Kernighan [SK72] and introduce the notion of balance used in the original FM algorithm.

FM Move Gain. While for graphs *every* incident edge of a node contributes to the gain either positively (in case it is a cut edge) or negatively (in case it is an internal edge), the situation is different for hypergraphs. Since nets can connect an arbitrary number of vertices, the contribution of a net $e \in \mathrm{I}(v)$ to the gain of a vertex v now depends on the actual block assignment of all pins of e. More precisely, the gain $g(v)$ of moving a vertex $v \in V_0$ to block V_1 of the bipartition $\Pi_2 = (V_0, V_1)$ is defined as:

$$g(v) := \omega(\{e \in \mathrm{I}(v) | \Phi(e, V_1) = |e| - 1\}) - \omega(\{e \in \mathrm{I} | \Phi(e, V_0) = |e|\}). \tag{3.1}$$

Only if v is the last pin of e that still resides in block V_0 will its move remove e from the cut-set. Similarly, if e has no pins in block V_1, the movement of v will make e a cut-net. In all other situations, incident nets do not contribute to the gain of vertex v.

Balanced Bipartitions. In order to deal with weighted vertices, a bipartition $\Pi_2 = (V_0, V_1)$ is considered a valid solution if $c(V_0)/c(V) \approx r$ for some specified parameter $0 < r < 1$. Since equality cannot be guaranteed in general, a deviation by the maximum vertex weight is allowed in both directions and thus a partition is considered *balanced* if

$$r \cdot c(V) - \max_{v \in V} c(v) \leq c(V_0) \leq r \cdot c(V) + \max_{v \in V} c(v). \tag{3.2}$$

Algorithm Outline. Given an initial bipartition $\Pi_2 = (V_0, V_1)$, each pass of the FM algorithm proceeds similarly to the KL algorithm. After initially computing the

gains of all vertex moves, the algorithm repeatedly selects the move with the highest gain, temporarily moves the vertex to the opposite block where it is locked to prevent thrashing, and updates the gains of neighboring vertices to reflect the changes. This process continues until every vertex is locked, and the best solution encountered during the pass is used as input for the next pass.

However, since the FM algorithm employs vertex moves instead of pairwise swaps and furthermore accounts for vertex weights, selecting the vertex with the highest gain differs from the KL algorithm. In order to determine whether to move a vertex from block V_0 to block V_1 or vice versa, the moves yielding the highest gains for both directions have to be compared. Unless all vertices in one block are locked, there always exists a vertex that can be moved from block V_0 to block V_1 and one vertex that can be moved in the opposite direction. Furthermore, by the definition of the balance constraint, at least one of these two moves is always feasible. If exactly one move is feasible, this move is chosen. In case both moves are feasible, the FM algorithm prioritizes moves by their gains and performs the one with higher gain, breaking ties in favor of the move that yields the better balance. If only one side contains unlocked vertices, but the highest-gain move would violate the balance constraint, it is discarded and the vertex is locked into its current block.

By using this move selection strategy, the FM algorithm does not require the initial input partition to be balanced – one of the blocks can even be initially empty. In case the balance constraint is violated, the algorithm automatically establishes balance by iteratively moving the highest gain vertex from the overloaded to the underloaded block.

Maintaining Move Gains. Having observed that hypergraphs derived from VLSI circuits are sparse [CKM00d; PM07] and that hyperedge weights are limited to small constant values [Len90, p. 268], Fiduccia and Mattheyses [FM82] exploit the fact that the move gain of any vertex can be bounded from above and from below by the maximum vertex degree Δ_v times the maximum hyperedge weight $\omega_{\max}$ by using two bucket priority queues (one for each block) to maintain all move gains in sorted order. Figure 3.1 depicts their original implementation. Let $p_{\max} := \Delta_v \cdot \omega_{\max}$ be the maximum gain. An array of length $2 \cdot p_{\max} + 1$ is used to represent all possible gain values in the range $[-p_{\max}, \ldots, +p_{\max}]$ and each cell i stores a pointer to a doubly-linked list that contains all unlocked vertices having gain i. Additionally, an array of size n is used to store a pointer to the gain element associated with each vertex. It is easy to see that it takes constant time to insert a vertex into its gain bucket. Since removing a vertex can also be done in constant time, updating the gain of a vertex is a constant-time operation. In order to retrieve a vertex with maximum gain, an additional pointer is used to index the highest-gain bucket that is not empty. During insertions, this pointer can easily be updated if the newly inserted vertex happens to have a larger gain. If, during a max-gain lookup, the corresponding bucket is found to be empty, it is necessary to linearly search downwards for the next nonempty bucket.

If hyperedge weights can be arbitrarily large, more sophisticated bucket priority queue implementations are necessary to keep memory consumption low and to speed

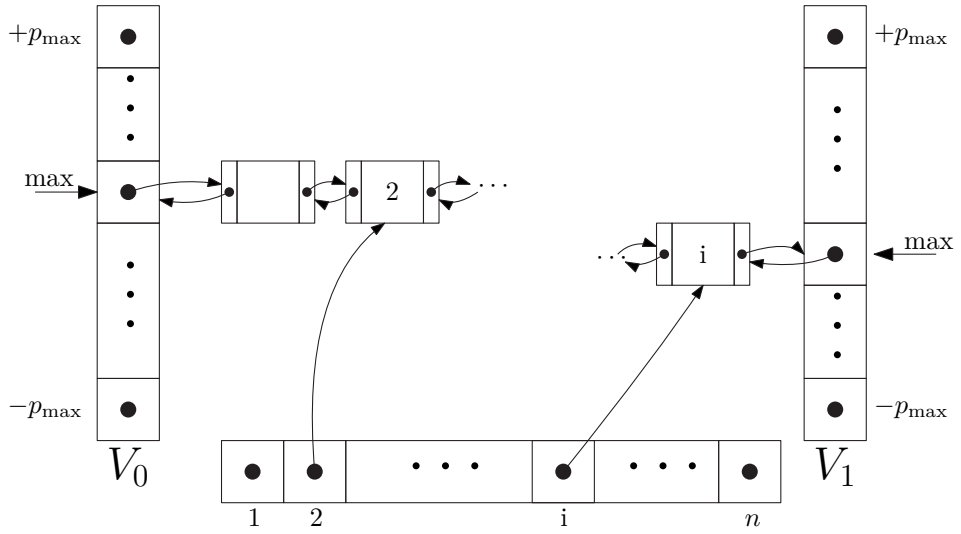


Figure 3.1: The gain bucket list priority queue used in the FM algorithm [FM82].

up the running time for locating the second-highest nonempty gain bucket. Papa and Markov [PM07], for example, propose to replace the bucket array with a combination of a search tree and a hash table.

Running Time Complexity. Initially computing all move gains and inserting them into the corresponding bucket priority queues can be done in $\mathcal{O}(p)$ time by iterating over all nets. After each move, it is necessary to update the gain values of all neighboring vertices. If this is done naively, the FM algorithm would have to perform $\mathcal{O}(p^2)$ gain (re-)computations per pass. However, Fiduccia and Mattheyses [FM82] show that by using delta-gain updates (that only account for the gain changes), a *constant* number of gain updates is required per net in one pass, and that each of these updates takes $\mathcal{O}(|e|)$ time. Since each bucket priority queue gain update can be done in constant time, all gain updates performed in a pass therefore take $\mathcal{O}(p)$ time in total. Finally, we have to account for the movements of the max-gain pointers that are used to find the next move. The total time to search down for a non-empty bucket is $\mathcal{O}(p)$, since each of the $\mathcal{O}(p)$ gain updates changes the max-gain pointer by at most $\omega_{\max}$ and thus the total number of decrements cannot exceed $\mathcal{O}(\Delta_v \omega_{\max} + p\ \omega_{\max})$, which is $\mathcal{O}(p)$ if $\omega_{\max} \in \mathcal{O}(1)$ for sparse hypergraphs.

> Note that it is actually not possible to update the gain of a vertex in constant time by solely using the data structure depicted in Figure 3.1. In order to be able to apply delta-gain updates, it is necessary to retrieve the current gain of a vertex. However, since this is not an $\mathcal{O}(1)$ time operation in this bucket queue implementation, another data structure, e.g. an additional array of size n can be used to explicitly store the absolute gain value for each vertex.

Early Experimental Results. Fiduccia and Mattheyses [FM82] only report results regarding the running time of their algorithm and note the effectiveness of the delta-gain update technique. Dunlop and Kernighan [DK85] compare the KL extension of Schweikert and Kernighan [SK72] to the FM algorithm and show that FM is significantly faster, while producing solutions of comparable quality in most cases. Goldberg and Burstein [GB83] compare the number of passes that each of the algorithms needs until it arrives at a local minimum and show that FM requires slightly more passes than the KL algorithm. This fact is also noted by Lengauer [Len90, p. 268].

Implementation Details. While a high-level description of the FM algorithm is straightforward, there are several ambiguities and implicit design descisions that need to be addressed in an actual implementation, some of which can have a significant impact on partitioning quality [Cal+99; CKM00b]. Being an iterative improvement algorithm, the FM heuristic needs an initial partition as a starting point. In the original implementation, a random partition was used for that purpose [FM82]. Hauck and Borriello [HB97] compared random initial partitions to more sophisticated approaches based of breadth-first (BFS) and depth-first searches (DFS), spectral partitioning [HK92b], and seeded initialization [WC89], in which a slightly modified version of the FM algorithm is used to grow one block of the bipartition around a randomly selected seed vertex. In their experiments, none of these techniques consistently outperformed random initialization. Hauck and Borriello [HB97] conjecture that this is due to the fact that these "smarter" initial partitioning algorithms tend to produce solutions of less variance and thus easily trap the algorithm in a local optimum. Especially in a setting where multiple runs are performed, initial solutions of high variance allow FM to explore the solution space more effectively.

An implicit design decision which is shown to have a significant impact on solution quality relates to the management of the bucket queues [PM07]. In the implementation of Fiduccia and Mattheyses [FM82], doubly-linked lists are used to represent a bucket and the gain array stores pointers to the heads of these lists. Whenever the gain of a vertex is updated, it is removed from its current bucket and inserted at the head of the new bucket. Furthermore, when selecting the next move, the implementation chooses the vertex at the head of the max-gain bucket. Thus, the gain buckets are managed in a last-in first-out (LIFO), stack-like manner. The implications of this design decision are not discussed by Fiduccia and Mattheyses [FM82]. Realizing that a popular FM implementation [San89] actually selected vertices uniformly at random from the max-gain bucket, Hagen et al. [HHK95a; HHK97] examined the impact of this decision by comparing LIFO bucket management with random selection, and an implementation that managed the buckets in a first-in first-out (FIFO) manner (by additionally maintaining a pointer to the tail of each bucket list). Since LIFO significantly outperformed both random and FIFO, implementations at that time almost universally adopted the LIFO scheme [Cal+99]. Intuitively, this effect can be explained by the fact that LIFO bucket management encourages a more localized search, because neighbors of a moved vertex are placed at the heads of the gain buckets and thus are more likely to be moved next. This increases the probability that tightly

connected clusters of vertices are moved together, since the LIFO scheme implicitly prefers moves in a certain area around the cut, which is not the case for FIFO or random [HHK95a; HB97].

Caldwell et al. [Cal+99; CKM99a; CKM99b; CKM00b] further analyze the gain update process. After a vertex is moved, the gains of all pins of incident nets potentially have to be updated. They note that while the method originally proposed by Fiduccia and Mattheyses [FM82] has the side effect of skipping all zero-gain delta gain updates, their algorithm is specific to both the net-cut objective and to bipartitioning. Straightforward implementations therefore may compute gain contributions in different ways that implicitly could lead to delta gains of zero. In this case, a zero-gain update would remove a vertex from its current position and insert it at the head of the same gain bucket. In their experiments using a LIFO FM implementation, performing zero-gain updates lead to significantly worse solutions. Recently, Kim and Yoon [KY15] proposed a further optimization (VLIFO) that handles gain updates differently depending on the sign of the update. If the gain delta is positive, the corresponding vertex is added to the head of the list, whereas updates that decrease the gain cause insertions at the tail. While they report an improvement over plain LIFO for flat FM, the experiments were performed on small and outdated [Alp98; Ady+04] ACM/SIGDA benchmark instances [RB87] (dating back to the late 1980s), so it remains unclear whether these results generalize to modern implementations.

Caldwell et al. [Cal+99; CKM99a; CKM99b] further show that some implementation details even affect each other and thus easily yield misleading conclusions. When selecting the next move to be performed from the max-gain buckets of both priority queues, the original FM algorithm chooses the one that produces a more balanced partition. Caldwell et al. [Cal+99; CKM99b; CKM00b] evaluate three different strategies, namely *towards* (i.e., perform the move in the same direction as the move before), *away* (prefer the move in the opposite direction), and *block 0* (always choose the move to block 0). When the algorithm performed zero delta-gain updates, biasing towards *block 0* gave significantly worse results than the *towards* strategy. However, when skipping zero delta-gain updates, there was no significant difference between *towards* and *block 0*. This effect was much less visible in a multi-level setting than when FM was used directly on the input hypergraph. This shows that stronger partitioning heuristics may actually hide the fact that the underlying flat partitioning engine is implemented badly [CKM00b].

A final tie-breaking decision is to be made when selecting the best partition at the end of a pass, since multiple solutions might yield the same improvement. While this is not specified by Fiduccia and Mattheyses [FM82] it is possible to use the first or last solution encountered, or the one that yields best balance [CKM00b]. While all of the previously mentioned implementation details have some effect on the partitioning results, there exist other optimizations that do not affect the solution quality such as the gain update procedure of Papa and Markov [PM07] that exploits special cases for the cut-net metric and bipartitioning to speed up delta-gain updates.

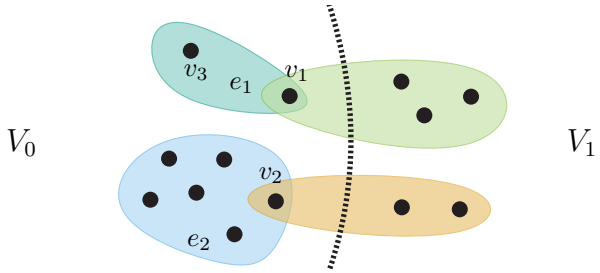


Figure 3.2: Motivation for higher level gains. Moving either v_1 or v_2 to the opposite block V_1 has an immediate gain of *zero*. However after moving v_1, net e_1 can be removed from the cut-set by additionally moving vertex v_3, while several more moves would necessary to remove net e_2.

3.3 Single-Level Partitioning beyond KL/FM

Simulated Annealing. The concept of *simulated annealing* (SA) [KGV83] is inspired by the annealing process in metallurgy (i.e., the controlled heating and cooling of metals to change their structural properties). In its basic form, a SA-based partitioning algorithm performs random vertex moves. If the new solution is better than the previous one, it is accepted immediately. However, worse solutions are also accepted with a certain, continuously decreasing, probability in order to be able to escape from local minima. This slowly decreasing probability hereby corresponds to the cooling process in the physical annealing process. Johnson, Aragon, McGeoch, and Schevon [Joh+89] compared the performance of SA and the KL algorithm, and concluded that neither algorithm consistently outperformed the other. However, the running times of SA were considerably longer, which is why the technique is not considered practical for hypergraph partitioning [AK95c].

LA$_\ell$-FM. Two years after the introduction of the FM algorithm, Krishnamurthy [Kri84] noted that moving vertices solely based on their *immediate* cut-size improvement leads to somewhat erratic behavior, since quite often several moves have the same direct effect, but significantly different ramifications for future moves. An example of such a situation is shown in Figure 3.2. Krishnamurthy [Kri84] therefore equipped the FM algorithm with the capability to look a fixed number ℓ of moves *ahead*. To capture how tightly a net is connected to a block of the bipartition, he introduced the concept of *binding numbers*. The binding number $\beta_A(e)$ of a net e with respect to block A is defined as

$$\beta_A(e) := \begin{cases} |A_F \cap e|, & \text{if } |A_L \cap e| = 0 \\ \infty, & \text{if } |A_L \cap e| > 0, \end{cases} \tag{3.3}$$

where A_F is the set of free (i.e., not yet moved) vertices and A_L is the set of locked (i.e., already moved) vertices in block A. Since each vertex is only allowed to be moved

once per pass, the binding number of a net e becomes infinity as soon as one of its pins is locked in the corresponding block – indicating that e will be connected to the block for the remainder of the pass. Given a bipartition $\Pi = \{A, B\}$ of a hypergraph $H = (V, E)$ with *unit* net weights, the ith level gain $g_i(v)$ of a vertex $v \in A$ is defined as

$$\begin{aligned} g_i(v) := &|\{e \in \mathrm{I}(v) \mid \beta_A(e) = i \wedge \beta_B(e) > 0\}| \\ &- |\{e \in \mathrm{I}(v) \mid \beta_A(e) > 0 \wedge \beta_B(e) = i - 1\}|, \end{aligned} \tag{3.4}$$

for $1 \leq i \leq \Delta_e$. The first level gain exactly corresponds to the original FM gain. For higher levels $i > 1$, the first term counts the nets e whose binding number becomes $\beta_A(e) = i - 1$ after the move, while the second term accounts for those nets whose binding number increases from $\beta_B(e) = i - 1$ to $\beta_B(e) = i$ due to the move. Instead of maintaining only a single gain value per vertex, the LA$_\ell$-FM algorithm with a look-ahead of ℓ maintains a *gain vector* of length ℓ per vertex, where the ith entry corresponds to the ith level gain. Gain vectors are then compared lexicographically in order to determine the next vertex to move. Thus, higher-level gains allow the algorithm to distinguish vertex-moves whose first level gains are equal. Krishnamurthy [Kri84] proposed to choose the look-ahead value ℓ depending on the properties of the hypergraph. More precisely, assuming that the $(2\Delta_v + 1)^\ell$ possible gain vector values are uniformly distributed over the n vertices, the highest-gain bucket is likely to contain only one vertex if $\ell \approx \log n / \log(2\Delta_v + 1)$. Using an ℓ-dimensional bucket data structure, the LA$_\ell$-FM algorithm can be implemented such that each pass takes $\mathcal{O}(\ell p)$ time. Note that while Krishnamurthy [Kri84] specified that random tie-breaking is used if both the move from block A to block B and the move from block B to block A are feasible and have the same gain, the decision which vertex to choose if the max-gain bucket contains more than one vertex was not specified.

> Interestingly, Krishnamurthy [Kri84] also proposes an optimization for the FM algorithm which, to the best of our knowledge, has gone largely unnoticed: If the algorithm keeps track of the number of locked nets (i.e., nets with a locked pin in each of the two blocks), then an FM pass can be terminated *as soon as* the number of locked nets is larger than the size of the currently best cut-set. Once a net becomes locked, it cannot be removed from the cut-set in the remainder of the pass. Thus, no subsequent vertex move will ever yield a smaller cut.

***k*-LA$_\ell$-FM.** Until the work of Sanchis [San86; San89], k-way partitions were computed via recursive bipartitioning and then optionally refined by repeatedly applying a two-way iterative improvement algorithm on pairs of adjacent blocks. As was already noted by Kernighan and Lin [KL70], recursive bipartitioning has the problem that a good solution for the first bipartition divides the instance into two densely connected blocks and thus makes it more difficult to find small cuts later on. Sanchis [San86; San89] furthermore points out that while pairwise refinements can be employed successfully for graph partitioning, the technique is unlikely to be promising when optimizing the *cut-net* metric in hypergraphs, since removing a net from the cut between a pair of blocks does not necessarily improve the cut-net objective in the original k-way

partition. She therefore generalizes Krishnamurthy's level gain concept [Kri84] from two-way to multiple-way partitioning and proposes the first *direct* k-way refinement algorithm. Her k-LA$_\ell$-FM algorithm maintains $k(k-1)$ gain bucket structures (one for each possible move direction) and an additional binary heap to locate the vertex move with highest gain in $\mathcal{O}(\log k)$ time. By analyzing the ℓ-dimensional gain bucket data structure proposed by Krishnamurthy [Kri84] more carefully, Sanchis first shows that the complexity of the LA$_\ell$-FM algorithm is actually $\mathcal{O}(\ell p(\Delta_v + \ell))$ and then devises an improved, more space-efficient bucket structure that allows her k-way extension to take $\mathcal{O}(\ell pk(\log k + \Delta_v \ell))$ time per pass using a look-ahead value of ℓ. The evaluation on *randomly* generated hypergraphs with different net-size distributions shows that the optimal look-ahead level increases with increasing number of blocks as well as with larger net sizes, and decreases with increasing vertex degree. The experimental insights are complemented with a probabilistic analysis to determine the optimal look-ahead based on the vertex-degree and net size distributions of the input hypergraph. In her experiments, k-LA$_\ell$-FM performed better than recursive bipartitioning if a look-ahead $\ell > 1$ was used. Without look-ahead capabilities, the direct k-way partitioning algorithm was not able to consistently outperform the recursive approach.

AlgI. While previous approaches worked directly on the input hypergraph, Kahng [Kah89] proposes an $\mathcal{O}(m^2)$ time algorithm that instead partitions its net intersection graph using random longest BFS paths. The partition of the net intersection graph then induces a partial bipartition of the input hypergraph, which is completed using a winner-loser heuristic [HNS82].

EV/BIPART. Motivated by successes of evolution-based approaches to solve combinatorial optimization problems, Saab and Rao [SR89b] present the first evolutionary algorithm for solving a k-way multi-objective, multi-constraint hypergraph partitioning problem. Since the algorithm only works with *one* individual, it does not use any recombination operators. Instead, the solution initially generated via bin packing is evolved using a non-greedy randomized algorithm that moves vertices to different blocks if their gain is greater than some random value and the move does not violate the balance constraint. In subsequent work [SR89a; SR90], this algorithm is specialized for the case of hypergraph bipartitioning. The initial partition is generated by scanning the vertices in decreasing order of weight and putting the current vertex in the smaller block. Afterwards, a perturbation function first visits all vertices again in order of decreasing weight and compares the gain $g(v)$ of moving vertex v to the opposite block to a random integer r in the interval $[p, 0]$. If $g(v) > r$, the move is added to a set of temporary moves, and the gains of all neighboring vertices are updated. Once all vertices are visited, the algorithm computes a subset of the temporary moves that improves the balance of the bipartition. The tuning parameter p is initially set to -1 and decremented if an iteration does not yield an improvement.

RCut1.0. In algorithms based on the FM paradigm, it is up to the user to specify the allowed imbalance between blocks. Since in practice this imbalance is small, FM-type algorithms tend to compute partitions with blocks of comparable sizes. However, since

hypergraphs derived from VLSI circuits exhibit a certain clustering structure, forcing block sizes to be balanced may lead to solutions with considerably larger cuts than some more imbalanced partitions. Wei and Cheng [WC89; WC91] therefore relax the balance constraint and propose the *ratio cut* objective for bipartitioning that captures *both* the size of the cut-set and the balance between the blocks. The *ratio cut* is defined as the bipartition $\Pi = (A, B)$ that generates the minimum ratio between the size of the cut-set and the product of the block weights $(c(A) \cdot c(B))$. Since the cut-net objective appears in the numerator while the denominator favors balanced partitions, the ratio cut objective favors "natural" (in contrast to artificially sized) bipartitions [HK92b]. Noting that finding the ratio cut in a hypergraph is NP-complete, Wei and Cheng [WC89; WC91] adapt the FM algorithm to optimize the objective. The initial partition is generated by growing blocks from two random seed nodes using the ratio values to prioritize vertices. This solution is then first optimized using a modified FM algorithm that only allows vertex moves in one direction. Afterwards, the full FM algorithm is used to make further improvements.

MCPG. While k-LA$_\ell$-FM [San86; San89] locks a vertex after it is moved for the first time, Vijayan [Vij90] presents a modification that allows every vertex to be moved to each part exactly once before it becomes locked. The algorithm thus has a larger search space at the cost of a higher running time complexity. However, since an experimental evaluation of the newly proposed locking scheme is missing, its benefits remain unclear.

WHB. Kamidoi, Wakabayashi, Miyao, and Yoshida [Kam+91] address a weakness of Kahng's AlgI partitioning algorithm [Kah89], namely the fact that the net intersection graph is missing information about vertex weights. This leads to the problem that the balance constraint can only be accounted for in the completion phase (i.e, after most vertices are already assigned to one of the two blocks), which often yields imbalanced and thus infeasible solutions. They therefore propose the WHB algorithm, which instead of the net intersection graph, works on the *star-expansion* of the hypergraph. Since this graph contains nodes for both vertices and nets of the hypergraph, WHB is able to account for vertex weights already during the computation of the partial bipartition and therefore always produces feasible solutions.

EIG1/EIG-IG/IG-Match. Hagen and Kahng [HK91a] were the first to adopt the spectral graph partitioning technique popularized by Donath and Hoffman [DH72; DH73] and Fiedler [Fie75a] to hypergraph bipartitioning for ratio-cut optimization. The EIG1 algorithm of Hagen and Kahng [HK91a] transforms the hypergraph into a graph G_x using the clique-expansion with uniform edge weight $1/(|e| - 1)$ for a hyperedge e of size $|e|$. After computing the second smallest eigenvector of the Laplacian matrix, the algorithm sorts the vector entries and chooses the splitting rank r that yields the best ratio-cut (using the previously ignored vertex weights). Vertices with rank $> r$ are then assigned to one block; vertices with rank $\leq r$ are assigned to the other block. In subsequent works [HK91b; HK92b], Hagen and Kahng extended the approach to use the net intersection graph G_n instead of the clique-expansion G_x.

A heuristic is then used to complete the partial bipartition of the vertex set induced by the partition of G_n. Later, Cong et al. [CHK92] replaced this heuristic by formulating the *optimal completion* of the bipartition as a minimum vertex cover problem on a bipartite graph derived from G_n.

NETPART. The NETPART algorithm of Hadley, Mark, and Vannelli [HMV92] generalizes the eigenvector technique to k-way hypergraph partitioning optimizing the cut-net metric. Here, the hypergraph is first transformed into a graph using the clique-net model together with an edge weighting scheme that tightly underestimates the number of cut-nets in any k-way partition of the hypergraph. Then, the eigenvector approach of Barnes [Bar82] is used to compute an initial partition of the clique graph. Lastly, the induced hypergraph partition is refined using the k-LA$_\ell$-FM algorithm of Sachnis [San86; San89] with a look-ahead of $\ell = 1$. Since large hyperedges cause large cliques, nets larger than 20 pins are removed from the hypergraph before spectral partitioning, and re-inserted before direct k-way refinement.

SA-TS/EIG-TS. Areibi and Vannelli [AV93a; AV93c; AV00] explore the effectiveness of combining tabu search [Glo89; Glo90] with simulated annealing [AV93c], and with Sanchis' k-LA$_\ell$-FM algorithm [AV93a]. Tabu search in a sense generalizes the locking mechanism used in KL/FM-type algorithms. Instead of allowing each vertex to be moved at most once in each pass, the tabu approach maintains a list of the t most recent moves. Vertices on the tabu list are disallowed to be moved again to avoid cycling, unless the move meets a predefined aspiration criterion (e.g., it improves the currently best solution), in which case the tabu list is disregarded.

[PP93]. Park and Park [PP93] propose an alternative objective function that, similar to the ratio cut, combines cut-size reduction and balancing of block weights into a single objective. However, while the motivation behind ratio cut is to form "natural" bipartitions, the objective function of Park and Park [PP93] is intended to allow FM-based algorithms to *always* move the highest-gain vertex (which could be prohibited by the balance constraint in traditional FM). The new objective function is defined as $\text{cut}(\Pi) + r \cdot \text{bal}(\Pi)$, where $\text{bal}(\Pi) := \sum_{1 \le i < j \le k} |c(V_i) - c(V_j)|$ measures the imbalance between pairs of blocks, and the balancing factor r controls the importance of balance relative to cut-size. The combined objective function is integrated into the k-way framework of Sanchis [San86; San89] (without look-ahead) and takes $\mathcal{O}(k(p + n^2))$ time per pass. A similar concept is proposed by Kim et al. [KLK93] to improve the imbalance of k-way partitions for hypergraphs with a high variance in vertex weights.

KP. Chan et al. [CSZ93; CSZ94] generalize the spectral 2-way partitioning approach of Hagen and Kahng [HK91a] along with the ratio-cut objective to k-way partitioning. The k-way generalization of the ratio cut is termed *scaled cost*. After transforming the hypergraph into the clique expansion G_x (hyperedges larger than 99 pins are ignored) and computing the k smallest eigenvectors, a clustering heuristic is used to infer the final k-way partition from an embedding of the nodes of G_x into a k-dimensional subspace defined by the eigenvectors. The heuristic clustering algorithm takes $\mathcal{O}(nk^2 + nk \log n)$ time.

KC/AGG. Alpert and Kahng [AK93] then extend the approach of Chan et al. [CSZ93; CSZ94] by proposing a new weighting scheme for the graph edges of the clique expansion and employing two simpler algorithms (KC and AGG) to cluster the points in the k-dimensional embedding. Instead of using an edge weight of $1/(|e|-1)$ for a clique edge corresponding to a hyperedge of size $|e|$, they use a uniform weight of $4/(|e|(|e|-1))$. Using this model, any cut net of a bipartition makes an *expected contribution* of 1 to the objective function. After computing the embedding, a minimum diameter clustering is computed using either the cluster-growing KC algorithm [Gon85] in time $\mathcal{O}(n \log k)$ or the cluster merging, agglomerative AGG algorithm in time $\mathcal{O}(n^2)$.

***k*-LA$_\ell$-FM.** Four years after generalizing the LA$_\ell$-FM algorithm [Kri84] to k-way partitioning, Sanchis [San93] revisits k-LA$_\ell$-FM and adapts it to work with both the connectivity metric $(\lambda - 1)$ and the $\lambda(\lambda-1)/2$ metric. While one pass of the algorithm takes $\mathcal{O}(p(\Delta_e + kl)(\log k + \Delta_v \ell))$ time for optimizing the former, one pass optimizing the latter takes $\mathcal{O}(p(\Delta_e + k\ell + k^2)(\log k + \Delta_v \ell))$ time. Experimenting with the look-ahead parameter ℓ, Sanchis notes that improvements using higher-level gains (i.e., $\ell > 1$) were not as large as for cut-net optimization.

DLA. Hoffmann [Hof94] addresses the problem that only allowing a vertex to be moved *once* during an FM pass may artificially constrain the search space. His dynamic locking algorithm (DLA) therefore allows each node to move back and forth between the blocks of a bipartition $\Pi = (A, B)$ a certain, fixed number of times. As in the FM algorithm, a vertex v is locked after it is moved from block A to block B. However, after the move, DLA unlocks all previously locked neighbors $\Gamma(v)$ which are still in block A. Thus, these vertices get another chance of moving back to block B.

PLM/PFM. The locking issue is also addressed independently by Daşdan and Aykanat [DA94; DA96; DA97]. The proposed PLM and PFM algorithms are extensions of Sanchis' k-LA$_\ell$-FM algorithm (for $\ell = 1$) and are based on the results of Daşdan's master thesis [Das93]. In the Partitioning by Locked Moves (PLM) algorithm, each FM-pass is divided into a number N of phases. Within a phase, each vertex is only allowed to be moved once. However, all vertices are unlocked again at the beginning of the next phase. Similar to the FM algorithm, the final block of each vertex is determined at the end of a pass. Partitioning by Free Moves (PFM) goes one step further and completely discards the locking mechanism. Instead, the concept of move mobility is used to decide which vertex is moved next. The mobility of a vertex hereby captures both the gain of the move and the number of times the vertex has already been moved. It increases with increasing gain and decreases as the move count gets larger. Both algorithms start from a random k-way partition and use the LIFO tie-breaking strategy if a gain bucket contains more than one vertex. While the running time per pass is $\mathcal{O}(Npk(k + G_{\max}))$ for PLM, the PFM algorithm takes time $\mathcal{O}(pk + k^2 s + f(k^2 + k\Delta_v \Delta_e s))$, where s is a scale factor used to scale the mobility values in the gain bucket data structure and f is the number of moves per pass.

GRCA. Bui and Moon [BM94; BM98] present a steady-state memetic algorithm (i.e., genetic/evolutionary algorithms combined with local search [Kim+11]) for ratio

cut bipartitioning, which uses a weak variation of the FM algorithm [FM82] as local search engine to improve offspring solutions created via crossover or mutation. To improve the performance of the crossover operation, a preprocessing step optionally re-indexes the vertices by the visiting order of a weighted depth first search on the clique expansion of the hypergraph (hyperedges larger than 20 pins are ignored).

SFC. Alpert and Kahng [AK94c; AK95a] revisit the problem of computing k-way partitions optimizing the scaled cost objective using spectral techniques and propose a restricted partitioning formulation. It is solved by computing a 1-dimensional ordering of the vertices in the k-dimensional embedding using a space-filling curves approach. The restricted formulation (called DP-RP) hereby forces each block of the k-way partition to be a contiguous subset of the ordering and is solved via dynamic programming.

WINDOW. In subsequent works, Alpert and Kahng [AK94a; AK94b; AK96] propose a general framework to construct vertex orderings that preserve the structure of the hypergraph (i.e., strongly connected vertices should be close to each other in the ordering) without the use of spectral techniques. The algorithm uses an *attraction* function which can be instantiated to optimize several objectives (e.g., scaled cost) together with a sliding window technique which ensures that yet unordered vertices are mainly "attracted" by the most recently ordered vertices. Already ordered vertices outside the window only exert an attraction that decreases proportionally with their distance to the end of the window. To create k-way partitions optimizing scaled cost, the DP-RP approach is then used to split the ordering into k contiguous subsets/blocks.

PARABOLI. Riess, Doll, and Johannes [RDJ94] propose the PARABOLI algorithm for optimizing the cut-net and the ratio cut metrics. It transforms the partitioning problem into a linear programming problem on the Laplacian matrix of the clique expansion of the hypergraph, which is then solved using the GordianL [SDJ91] algorithm.

KDualPartFM. Cong et al. [CLS94; CLS96] follow up on the work of Cong et al. [CHK92] and propose a k-way partitioning algorithm that optimizes the cut-net metric by partitioning a new type of dual hypergraph. The new representation is a hybrid between the net intersection graph and the dual netlist hypergraph, where a threshold parameter is used to determine when hyperedge-nodes are connected via hyperedges instead of clique edges. This hybrid hypergraph is initially partitioned using both a greedy and a random initial partitioning algorithm and further refined using k-LA$_\ell$-FM with $\ell = 1$. The solution of the net partition is then transferred back to a partial vertex partition. The set of vertices that could not be assigned to a specific block (because incident nets were in different blocks in the net partitioning solution) are optimally distributed to the k blocks of the vertex partition by solving a flow problem on a specifically constructed assignment network. Finally, the now complete vertex partition is refined once again using the Sanchis' k-way FM algorithm.

FBB. Yang and Wong [YW94; YW96; YW08] were the first to employ repeated, incremental max-flow min-cut computations as a heuristic to find ε-balanced hypergraph

bipartitions. Their Flow-Balanced-Bipartitioning (FBB) algorithm first transforms the hypergraph into a flow network as proposed by Lawler [Law73]. Then, two vertices are chosen uniformly at random to act as source and sink nodes, and an augmenting path algorithm is used to compute a maximum flow. Let A be the set of all vertices that are reachable from the source node via an augmenting path. Then (A, B), with $B = V \setminus A$ comprises a bipartition of the hypergraph. If A violates the balance constraint, all nodes of B along with one additional, randomly chosen node $v \in A$ (the piercing node) are contracted into the sink node and an incremental max-flow computation is used to compute a new bipartition. Including v in the contraction allows FBB to compute a different cut with a larger block B. This process is then repeated until the algorithm finds a feasible bipartition.

PANZA. Solution quality and running time of FBB are improved by the PANZA algorithm of Li et al. [LLC95], which implements max-flow computations on the star-expansion of the hypergraph and introduces advanced, eigenvector-based heuristics to select source and sink, as well as piercing nodes.

MELO. Building on the results of Alpert and Kahng [AK94c; AK95a] and Chan et al. [CSZ93; CSZ94], Alpert et al. [AKY99], and Alpert and Yao [AY94; AY95] propose the multiple eigenvector linear ordering (MELO) heuristic for scaled cost k-way partitioning that uses as many eigenvectors as computationally possible to get a better approximation of the partitioning problem. The d distinct eigenvectors are then heuristically combined into a single vertex ordering, which is split into a k-way partition using the DP-RP approach [AK94c; AK95a]. The running time of MELO is $\mathcal{O}(dn^2)$, where d is the number of eigenvectors used for partitioning.

LIFO-LA$_\ell$-FM. Hagen et al. [HHK95a; HHK97] revisit the FM algorithm [FM82] along with the look-ahead extension of Krishnamurthy [Kri84] and the k-way generalization of Sanchis [San86; San89]. They note that while Fiduccia and Mattheyses [FM82] used a LIFO tie-breaking scheme for vertices within the same gain bucket, the original implementations of Krishnamurthy [Kri84] and Sanchis [San86; San89] selected a vertex uniformly at random in case of ties. Comparing both approaches experimentally with a FIFO strategy reveals that FIFO is considerably worse than random, which in turn is outperformed by LIFO selection. Hagen et al. [HHK95a; HHK97] explain this result by the observation that a LIFO scheme leads to a more *localized* search in a certain area around the cut. Their experiments furthermore show that the choice of the tie-breaking scheme has a greater effect on solution quality than using higher-level gains, since LIFO-FM performed significantly better than FIFO/Random-FM using look-aheads of $\ell = \{2, 3, 4\}$. Lastly, they propose an extension to Krishnamurthy's gain formulation that prefers to move vertices incident to loose nets (i.e., nets that contain at least one locked pin in one of the two blocks). More precisely, the gain formulation shown in Equation 3.4 is extended by adding the term $|\{e \in \mathrm{I}(v) \mid 0 < \beta_A(e) < \infty \wedge \beta_B(e) = \infty\}|$ for all look-ahead levels $\ell > 1$. While the first level therefore still corresponds to the actual FM gain, higher-levels prefer moves towards blocks containing locked vertices. This ensures that in case of

ties those moves are preferred that prevent nets from being locked in the cut-set for the remainder of the pass.

GFM. Noting that FM is highly sensitive with regard to the initial bipartition, Liu, Huang, and Cheng [Liu+95c] propose to compute the initial solution by solving an unconstrained integer mathematical program using a gradient descent approach instead of using randomly generated bipartitions.

LSMC. The work of Fukunaga, J.-H.Huang, and Kahng [FJK96] is also motivated by the high quality variance of solutions computed by the FM algorithm. However, instead of using more advanced initial partitioning techniques, they embed FM into a metaheuristic that works similar to simulated annealing. Their LSMC algorithm starts with a random bipartition which is refined using the FM algorithm until no further improvement is found. Then, a "kick move" is used to perturb the current solution into a new starting partition for FM. If the refined partition is better than the solution before applying the kick move, it is accepted as starting point for the next iteration. Otherwise it is rejected with a certain probability. This process is repeated for a predefined number of iterations and the best solution encountered is returned as the final bipartition. The authors evaluate several possible perturbation strategies out of which the "clustering kick move" performed best. The intuition behind this approach is that tightly connected clusters of vertices are likely to trap the FM algorithm in the current local minimum. This strategy therefore grows two clusters of equal size around two randomly chosen border vertices using breadth-first search and then swaps the clustered vertices to create a perturbed solution.

PROP. Dutt and Deng [DD96a] note that even Krishnamurthy's look-ahead extension [Kri84] of the FM algorithm is not able to accurately predict the future implications of a vertex move on its incident nets. They therefore propose a probabilistic gain formulation (PROP) that additionally associates a probability $p(v)$ that it will *actually* be moved during a pass (i.e., it is part of the sequence of moves that leads to the best solution encountered) with each vertex v. In an iterative fashion, these probabilities are used to compute more accurate vertex gains, which in turn are used to compute more accurate probabilities. After a few cycles, the final gain values will then be used in the FM algorithm. Since computing and maintaining the probabilities is more expensive than using plain FM gains, the running time of PROP becomes $\mathcal{O}(p^2/m \log n)$.

CLIP/CDIP. Furthermore, Dutt and Deng [DD96b; DD96c; DD02] observe that since hypergraphs derived from VLSI circuits contain many highly connected clusters of vertices with various densities, random initial bipartitions are likely to have clusters spanning the cut-set (i.e., vertices of the cluster are contained in both blocks of the bipartition). To improve solution quality, it is therefore necessary to "push" these clusters into one of the two blocks. However, since several vertices of different clusters may have the same gain, both FM and LA$_\ell$-FM cannot distinguish between clusters and thus are likely to work on several clusters simultaneously. Moreover, a cluster is easily locked into the cut-set (i.e., nodes of the cluster are locked in both blocks),

because FM-type algorithms move vertices in both directions. Dutt and Deng [DD96b; DD96c; DD02] therefore propose a framework that implicitly favors moves which help to push one cluster at a time out of the cut-set. To do so, their cluster-oriented iterative improvement partitioner (CLIP) dynamically increases the weights of nets incident to recently moved vertices. The key idea is to divide the original gain formulation into two components: initial gain and delta gain. The initial gain corresponds to the gain computed at the beginning of a pass. The delta gain refers to the gain updates each vertex is subjected to due to the movement of neighboring vertices. CLIP uses the initial gains to insert all vertices in the corresponding bucket data structures. Before starting an improvement pass, all gains are then *reset to zero* by concatenating the linked lists of all gain buckets in decreasing order and inserting this list into the zero-gain bucket. Then, the algorithm proceeds as usual, i.e., it removes the vertex with the highest gain, moves it to the opposite block, and updates the gains of all neighboring vertices. However, since those gains have been reset to zero, CLIP thus implicitly prioritizes neighbors of the just moved vertex – forcing the FM algorithm to work in a *localized area* of the cut-set.

While the CLIP approach thus encourages moves of clustered vertices, it is not able to detect whenever a cluster is completely removed from the cut-set and the removal of a new cluster should start. This shortcoming is addressed with the cluster-detecting iterative-improvement partitioning (CDIP) heuristic, which tracks gains of previously moved vertices and considers a cluster to be removed from the cut-set if the overall improvement stagnates for more then δ moves, where δ is a tuning parameter. After reversing those moves, CDIP then uses the total gain (i.e., initial plus delta gain) to determine a new start vertex for removing the next cluster. Furthermore, unlike at the beginning of a CLIP pass, it only selectively resets the gain values to zero such that information from the previously removed cluster is retained. For each free vertex, this is done by keeping negative gains induced by nets containing locked pins in the same block as the vertex, since these nets indicate that the vertex may belong to an already moved cluster and therefore should remain in its current block. Since both cluster-aware approaches need an underlying IIP engine, the authors propose the usage of FM [FM82], LA$_\ell$-FM [Kri84], and PROP [DD96a] in combination with CLIP and CDIP.

LSR. The work of Cong, Li, Lim, Shibuya, and Xu [Con+97a; Con+97b] is based on Lim's master thesis [Lim97] and similarly to CLIP tries to remove clusters of vertices from the cut-set using a modified FM algorithm. However, instead of increasing locality by splitting the vertex gain into two components, their loose net removal (LR) algorithm intentionally increases the gains of neighbors of a moved vertex which are incident to loose nets. More precisely, for each loose net, the algorithm increases the gains of *free* pins in the unlocked block such that they are more likely to be moved next. This additional gain is added on top of the traditional FM gain, and is devised such that it favors small nets, strong connectivity to the locked block and weak connectivity to the unlocked block. LR therefore tries to (i) prevent loose nets from becoming locked into the cut-set, and (ii) to push them into the block where they

already have locked pins. Additionally, LR is combined with the stable net transition (SNT) algorithm of Shibuya, Nitta, and Kawamura [SNK95] into the LSR algorithm. Instead of restarting LR with a new initial partition after it arrives at a local minimum, LSR compares the cut-set of the final bipartition with the initial cut-set before LR refinement to identify stable nets (i.e., nets that remained cut throughout the run). Then, it randomly chooses a stable net and moves all of its pins into the smaller block – effectively removing it from the cut-set. This process is repeated until a fixed percentage of stable nets is removed, or, since each vertex is only allowed to be moved once, no further moves are possible. SNT thus sufficiently perturbs the current solution to allow the LR algorithm to climb out of its current local minimum, and, at the same time, is faster than repeatedly restarting LR from a new random partition.

ASFM. Buntine, Su, Newton, and Mayer [Bun+97] experimentally analyze a LIFO-FM implementation which starts from a random bipartition. They note that while the first few passes rapidly improve the solution quality, later passes merely correspond to local restarts of the algorithm which yield only minor quality improvements. Motivated by this observation, they propose an adaptive stochastic variant of FM which performs only four full passes and then switches to a combination of shortened passes (that do not move every vertex) and stochastic passes (that reject moves with a certain probability). Furthermore, this variant is embedded into a wrapper that tracks quality improvements over multiple runs, and terminates a pass if it did not sufficiently improve the solution quality compared to the previous runs.

PROP-REX. Most IIP algorithms address the balance constraint by disallowing moves that would yield infeasible solutions. Motivated by the fact that this approach biases the search against moving heavy vertices, Dutt and Theny [DT97] propose a relaxation process that allows a temporary violation of the balance constraint. Their algorithm combines the probabilistic vertex gain approach of PROP [DD96a] with a look-ahead for future violations, and a weighted benefits function that favors high-gain violating moves if it is likely that the violation can be accounted for by moving vertices in the opposite direction.

[BBR97]. Battiti et al. [BBR97] propose a series of greedy construction algorithms that use different gain notions for assigning a vertex to a block of a bipartition. While none of the schemes alone was able to consistently outperform the FM algorithm, using the best scheme to construct an initial partition for FM was seen as a viable alternative to using random initial bipartitions.

FBB. Liu and Wong [LW98] revisit the FBB algorithm and extend it to solve a VLSI circuit-specific multi-way partitioning problem. Although this algorithm is not directly applicable to k-way hypergraph partitioning, we explicitly mention this work because it also contains two improvements for FBB. First, Liu and Wong [LW98] propose the improved flow model described in Section 4.7.2, which distinguishes between two-pin and multi-pin nets. Second, instead of using the partition induced by the max-flow computation, they propose a greedy heuristic that searches for the most desirable min-cut, which is defined as the min-cut that yields blocks with a size close to the

maximum allowed imbalance. This reduces the number of iterations in FBB and, since each iteration increases the cut-size, also improves solution quality.

DEEP/VAR–PROP. With DEEP-PROP, Dutt and Theny [DT98] and Dutt, Arslan, and Theny [DAT99] extend PROP to not only account for the future effects of a vertex move on *incident nets* (first-order information) but also to take into account the implications that uncutting an incident net e has on nets adjacent to e (second-order information). The memory consumption of DEEP-PROP is $\mathcal{O}(m\Delta_v\Delta_e)$ and the running time is $\mathcal{O}(n\overline{\Delta}_v\overline{\Delta}_e^2 \log n) = \mathcal{O}(p^3/m^2 \log n)$, where $\overline{\Delta}_v = p/n$ is the average vertex degree and $\overline{\Delta}_e = p/m$ is the average hyperedge size. The authors therefore also propose a lower-complexity version named VAR-PROP which in general follows similar ideas, but retains the running time complexity of PROP, which is $\mathcal{O}(n\overline{\Delta}_v\overline{\Delta}_e \log n) = \mathcal{O}(p^2/m \log n)$. While these complexities are feasible for partitioning hypergraphs derived from VLSI circuits for which both $\overline{\Delta}_v$ and $\overline{\Delta}_e$ are small constants, they become prohibitive for general hypergraphs.

MMP. Noting that the balance constraint often obstructs the movement of heavy vertices in the FM algorithm, Cherng et al. [CCH98] propose an iterative improvement algorithm that temporarily relaxes the balance constraint and renders FM's locking mechanism unnecessary. The algorithm works in passes, where each pass is divided into a forward and a backwards moving phase. The former only moves vertices from block V_0 to block V_1. The latter starts from the best solution seen so far and moves vertices in the opposite direction until further moves would violate the balance constraint. The best among all feasible partitions then forms the initial solution for the next pass.

K-PM/LR. Although the idea to refine a k-way partition by repeatedly applying a 2-way local search algorithm on pairs of blocks was already proposed by Kernighan and Lin [KL70] in 1970, it was only put into practice 28 years later by Cong and Lim [CL98]. They note that while direct k-way partitioning benefits from a more global view and larger search spaces (by being able to consider all k blocks simultaneously), relatively little progress was made to improve direct k-way algorithms like Sanchis' k-LA$_\ell$-FM [San86; San89; San93]. The lack of research in this direction is justified with the recurring observation that k-LA$_\ell$-FM is susceptible to being trapped in a far-from-optimal local minima, because the large number of potential moves and move directions make the algorithm prone to making wrong decisions.

> Interestingly, Buntine et al. [Bun+97] give another explanation for the observed performance difference between 2-way and k-way FM, which has gone largely unnoticed. They note that "for k-way partitioning with k greater than 2, the final state in the pass is not a mirror of the initial state, so as the pass proceeds, the states just drift away from the initial state and a second local move is not created at the end of the pass." Thus, while near the end of a pass, 2-way FM *again* only makes *local* changes to the solution it started with, its k-way counterpart can be arbitrarily *far away* from the initial solution.

However, computing a k-way partition via recursive bipartitioning has several draw-

backs as well: Vertices can only move between the blocks of the current subhypergraph (i.e., RB has only a partial view of the whole partitioning problem) and for deeper levels of the hierarchy it becomes increasingly harder to find small cuts, since 2-way FM divides the instance into two densely connected blocks at each level. Cong and Lim [CL98] therefore introduce the K-PM/LR algorithm, which first computes an initial k-way partition via recursive bipartitioning using the LR algorithm [Con+97a; Con+97b; Lim97], and then employs the FM algorithm [FM82] on pairs of adjacent blocks. Blocks are paired based on the improvement achieved in previous passes, and the algorithm stops as soon as a pass refining all possible pairs of blocks did not yield an improvement.

IDP. While previous work was mostly concerned with iterative improvement and spectral or flow-based algorithms, the iterative deletion algorithm of Madden [Mad99] utilizes an entirely different paradigm, which he describes as follows: "Iterative improvement algorithms pursue moves that appear the 'best', while iterative deletion algorithms eliminate moves that appear the 'worst' ". Instead of starting with a valid k-way partition, his algorithm starts with a redundant assignment (i.e., each vertex is assigned to *every* block) and then iteratively removes individual assignments in a greedy manner such that the number of cut hyperedges is reduced until it arrives at a feasible solution.

> To the best of our knowledge, Madden [Mad99] was the first to use both uniform and varying *hyperedge* weights in his experiments. While LIFO-FM performed better than IDP for uniformly weighted nets, its solution quality for hypergraphs with weighted hyperedges degraded substantially. Madden explains this behavior with the fact that the recommended LIFO tie-breaking (which is intended to increase localization) has almost no effect in this case, since the net weights increase the range of possible vertex gains such that there are extremely few ties in the max-gain buckets, rendering LIFO tie-breaking superfluous.

[Are99]. Areibi [Are99] evaluates the use of the greedy random adaptive search (GRASP) procedure [FR89] for k-way hypergraph partitioning. Each GRASP iteration consists of a construction phase, in which an initial k-way partition is constructed based on a simple, randomized greedy algorithm, and a local improvement phase which uses the k-LA$_\ell$-FM algorithm to refine the solution.

Branch-and-Bound. Caldwell, Kahng, and Markov [CKM99d; CKM00e] quantify the suboptimality of solutions computed by FM-based algorithms under tight balance constraints for small instances with a large variance of vertex weights by comparing them with the *optimal bipartitions* computed by a branch-and-bound as well as an exhaustive enumeration algorithm. The latter is based on Gray code enumeration. It starts with all vertices assigned to block zero, and then reassigns one node at a time. The former uses a hybrid hypergraph representation which allows to speed-up the algorithm by exploiting *inevitable cuts* caused by hyperedges of size two. An inevitable cut is hereby induced by an unassigned vertex v and its already assigned neighbors in the two blocks to which v is only connected via regular graph edges. No

matter to which block the vertex will be assigned, the connections to the opposite block will induce additional cut edges. This fact can be used to compute improved lower-bounds on the solution quality and thus to speed up the branch-and-bound algorithm. Furthermore, following Ihler et al. [IWW93], three-pin hyperedges are represented as appropriately weighted cliques in order to increase the number of two-pin nets.

> An interesting observation of Caldwell et al. [CKM99d; CKM00e] is that for hypergraphs with large variance in vertex weights and partitioning problems with tight balance constraints, the instances become more difficult for FM-based algorithms, because "1) the FM algorithm may never reach the feasible part of the solution space (especially if it has trouble finding an initial balance-feasible solution) and 2) even a relative scarcity of feasible moves (from any given feasible solution) can make the algorithm more susceptible to being trapped in a bad local minimum" [CKM00e, p. 1305]. In such situations, the assignment of heavy vertices is even likely to be only determined by the initial partitioning algorithm and may never be changed during refinement, because the corresponding vertex moves are never feasible. This is a problem, because almost all move-based partitioning algorithms were originally proposed for, and almost exclusively evaluated on, hypergraphs with unweighted vertices.

VRW. Caldwell, Kahng, and Markov [CKM99c] and Alpert, Caldwell, Kahng, and Markov [Alp+00] were the first to study the FM algorithm in the presence of fixed vertices (i.e., some vertices are already assigned to a block of the bipartition before partitioning and are not allowed to be moved during the partitioning process). Their experiments indicate that while full FM passes are necessary to achieve high solution quality for classical hypergraph partitioning, it is possible to limit the number of moves to a certain fraction of the vertex set (for all passes except the first) in the presence of fixed vertices and still achieve very good solutions. With VRW, Caldwell et al. [CKM00c] then propose new techniques for FM-based partitioning that are able to actively exploit the presence of fixed vertices and, at the same time, also perform well in their absence. Instead of using random bipartitions as initial solutions, their algorithm uses a "very illegal" initial partitioning algorithm (VILE), which puts all free (i.e., not fixed) vertices into one block. This initial solution is then refined with a modified FM algorithm, which relaxes the feasibility constraint of temporary solutions by accepting all moves that do not *increase* the violation of the balance constraint. Furthermore, at the beginning of a pass, (i) vertex gains are computed in random order, and (ii) neighbors of fixed vertices are moved to the heads of their respective LIFO gain buckets by "wiggling" fixed vertices back and forth between the two blocks. This allows these vertices to move to the block of their fixed neighbor already in the beginning of a pass.

$\mathbf{CLIP_2}$/$\mathbf{LIFO_2}$. Observing that previous partitioning heuristics were mostly proposed for and evaluated on hypergraphs with *unit* vertex weights (with the notable exception of PROP-REX [DT97]), Caldwell, Kahng, and Markov [CKM00d] inves-

tigate the performance deterioration of FM and CLIP implementations for more recent benchmark hypergraphs with *non-uniformly* weighted vertices. They note that mostly instances from the ACM/SIGDA benchmark set [RB87; Brg93] were used whenever previous work experimented with vertex-weighted hypergraphs. However, vertex-weights of those instances are actually *nearly uniform*. In contrast, the more recent ISPD98 benchmark set [Alp98] contains hypergraphs with large variations in vertex weights (with vertices whose weight is larger than 10% of the total weight of the hypergraph).

> They also note that while ACM/SIGDA hypergraphs only contain low-degree vertices (with up to 10 incident hyperedges) but have large nets (with more than 1000 pins), the ISPD98 benchmark hypergraphs (which are still in use today) have "node degrees in the several hundreds; however, [...] no large nets" [CKM00d]. In fact, more than half of the nets of ISPD98 hypergraphs are actually graph edges.

Using these instances, they demonstrate two effects that cause FM/CLIP implementations to perform badly in the presence of vertex weights. First, the strategy of FM-based algorithms to only allow moves that do not violate the balance constraint leads to the problem that nodes heavier than the balance tolerance are never allowed to be moved. Thus, if the initial partition happens to choose a wrong assignment for heavy nodes, the local search algorithm will never be able to recover from this misassignment. Second, they describe the *corking effect*: Since previous work was mainly driven by the development and tuning of heuristics for partitioning unweighted hypergraphs, typical implementations only looked at the first vertex in the highest gain bucket and skipped the entire bucket, if moving this vertex was not feasible. In the presence of vertex weights this is especially bad for CLIP-based heuristics, since CLIP puts all vertices into the zero-gain bucket at the beginning of a pass and the node at the head of this bucket is likely to be a high-degree (and also heavy) vertex. If this vertex cannot be moved, it thus acts as a cork that clogs the bucket queue. While the obvious mitigation for the second issue is to look beyond the first move, Caldwell et al. [CKM00d] also propose to perform a single LIFO-FM pass before employing CLIP. In order to address the first problem, they suggest to temporarily relax the balance constraint by calling the local search procedure multiple times. While for the first call the balance constraint is modified such that every vertex is movable, it is subsequently tightened in later calls, ensuring that the final partition is feasible with regard to the original balance constraint.

Shrink-PROP. Revisiting PROP, Dutt and Deng [DD99; DD00] show that whenever the first pin of a net is moved, PROP either does not adjust its removal probability at all or actually decreases it. This has the side effect that the vertex gains of its remaining free pins are not updated appropriately. They therefore propose the SHRINK-PROP algorithm as an enhancement of PROP which amplifies vertex gains whenever a free cut-net (i.e., a net whose pins have not yet moved) becomes loose. Since this only entails a minor modification of the PROP algorithm, SHRINK-PROP retains the same asymptotic complexity.

[YC00]. Yarack and Carletta [YC00] experimentally analyze the PFM algorithm of Daşdan and Aykanat [DA94; DA96; DA97] and show that its increased running time is due to the fact that in each pass of the algorithm many vertices move multiple times. However, since the major cut-size reduction mainly happens in the first few passes, they adapt PFM to allow vertices to move more freely at the beginning than at the end of a pass, and significantly reduce the vertex mobility for all passes except the first.

SDP. Choi and Ye [CY00] propose a semidefinite programming (SDP) approach to computing bisections that works on the clique-expansion. While generating solutions of reasonable quality, the running time of their algorithm is prohibitively long.

MDC-RS. Areibi [Are00b] proposes an extension of k-LA$_\ell$-FM that first relaxes the balance constraint by a factor of δ. After all nodes are moved, it then creates a feasible solution by greedily moving vertices from overloaded to underloaded blocks. Furthermore, after arriving at a local minimum, the algorithm performs a series of moves that again worsen solution quality in order to allow the next FM pass to explore a different part of the search space. In a subsequent work, this algorithm is then embedded into an evolutionary framework [Are00a].

NGSP. Zhao [Zha00], Tao [Tao02], and Zhao, Tao, and Zhao [ZTZ02] extend the LSR algorithm of Cong et al. [Con+97a; Con+97b] from 2-way to k-way partitioning and propose a minor modification to the gain formulation of LR such that it is able to distinguish between internal nets and nets that are part of the cut-set.

WalkPart. Ramani and Markov [RM03] adapt the biased random walk heuristic of the WalkSAT Boolean Satisfiability solver to hypergraph bipartitioning. Their WalkPart algorithm first randomly selects a net from the cut-set and then either chooses a random pin with probability p, or, with probability $1 - p$, the pin v that minimizes a normalized scoring function. For each net $e \in \mathrm{I}(v)$, the score measures the number of moves required to remove e from the cut-set given the fact that v is moved, and is normalized with the size of e. The score of a vertex v then is the sum of the normalized scores of its incident nets $e \in \mathrm{I}(v)$. Since WalkPart appears to have different local minima than the FM algorithm, the authors propose a hybrid strategy that first runs FM and then applies the WalkPart heuristic to lead FM out of the current local minimum.

LFM. Kim, Kim, and Moon [KKM04] generalize the concept of *lock gains* [KM04] from graphs to hypergraphs and propose an FM variant (LFM), which uses the new gain formulation to select the next vertex move. Traditional FM gains are only employed for tie-breaking. By taking the number of locked and free pins of hyperedges into account, the lock gain concept tries to incorporate information about previous vertex moves into the move selection process. In order to further improve solution quality, LFM is then embedded into a simple steady-state evolutionary algorithm.

[Arm+10]. Armstrong et al. [Arm+10] propose a shared-memory memetic algorithm that performs crossover, mutation, and local search on multiple individuals in parallel

by assigning a fraction of the total population to each core. To exploit the local solution space, the algorithm employs either FM or k-LA$_\ell$-FM.

Hyper-PuLP. In his master thesis, Buurlage [Buu16] generalizes the Partitioning using Label Propagation (PuLP) graph partitioning algorithm [SMR14] to hypergraph partitioning with the intention to create a "reasonable [sic] good partitioning that is not necessarily of highest quality" [Buu16, p. 51], but still better than e.g. a simple zero-cost partitioning scheme such as a cyclic distribution of vertices to blocks [Buu16, p. 64]. The algorithm works directly on the input hypergraph and the label propagation procedure is adapted such that the number of differently labeled pins in all hyperedges is minimized. This is achieved by defining a weighting function for connections to neighboring vertices that (i) prefers not to introduce new labels to a net, (ii) prefers labels that occur often within a net, and (iii) refrains from choosing a label that only few pins of the hyperedge have in common.

SHP. With the introduction of two-level and multi-level algorithms (which will be discussed in the following section), HGP-related research mostly shifted away from flat partitioning. However, with SHP, Kabiljo et al. [Kab+17] recently proposed a distributed-memory algorithm that again directly operates on the input hypergraph. Starting from a random initial partition, the algorithm performs a number of iterations that move vertices between blocks in order to improve solution quality. In each iteration, SHP first computes the move that yields the highest gain for every vertex and records the number of vertices that would like to move from bock V_i to block V_j in a $k \times k$ matrix S. Then, the move probability $p_{i,j}$ for actually moving a vertex $v \in V_i$ to block V_j is calculated as $\min(S_{i,j}, S_{j,i})/S_{i,j}$ for all pairs of blocks. This ensures that, in expectation, the same number of vertices are moved in both directions. Finally, the algorithm draws a random number r from the interval $[0, 1]$ for each vertex $v \in V_i$ and performs the move to block V_j if $r < p_{i,j}$ and $g(v) > 0$. While the general structure of SHP is inspired by the KL algorithm [KL70; SK72], the key difference is the objective function used in the optimization. Noting that directly optimizing objectives such as the connectivity metric can easily trap heuristics in local minima, Kabiljo et al. [Kab+17] suggest to optimize *probabilistic fanout*. The objective is motivated by an application in which hypergraph vertices represent data items, hyperedges represent queries, and each query requires one of its data items only with certain fixed probability p. Thus, for a given k-way partition $\Pi = \{V_1, \ldots, V_k\}$, the probability that a hyperedge e with $\Phi(e, V_i)$ pins in block V_i needs one of those pins is $1 - (1 - p)^{\Phi(e,V_i)}$. The p-fanout of a hyperedge is then defined as the sum of these probabilities over all k blocks, and the objective is to optimize the average p-fanout of all hyperedges. Assuming a constant number of iterations, the algorithm is shown to run in $\mathcal{O}(kp)$ time. In order to cope with partitioning instances for which $k = \Omega(n)$, the authors also propose a recursive bisection variant (SHP-2) by constraining the possible move targets for each vertex at each level. An implementation of SHP-2 in the Apache Giraph [Mal+10; Fou] framework is publicly available.

HYPE. Recently, Mayer et al. [May+18] generalized the neighborhood expansion heuristic originally proposed by Zhang, Wei, Liu, Tang, and Li [Zha+17] for graph

edge partitioning to k-way hypergraph partitioning. Their HYPE algorithm computes a k-way partition by repeatedly growing one block V_i at a time. The growing process starts from a random seed vertex. In order to determine the next vertex v to be added to block V_i, HYPE maintains a set F_i of s neighboring fringe vertices. It then successively adds the fringe vertex with *smallest* external neighbors score d_{ext} to V_i, chooses r new candidate vertices to be added to the fringe, and takes the best $s-1+r$ fringe candidates as the new fringe. Fringe candidates are also chosen based on external neighbors score, which for a given vertex v is defined as $d_{\text{ext}} := |\Gamma(v) \setminus F_i|$. By preferring vertices with small d_{ext} score (i.e., vertices with a large number of neighbors already contained in $F_i \cup V_i$), HYPE tries to preserve structural locality while growing each block. Since updating the fringe can be expensive due to the potentially large neighborhood induced by large hyperedges, Mayer et al. [May+18] propose three optimizations to make fringe updates efficient: (i) hyperedges incident to fringe vertices are traversed in increasing order of their size (since good fringe candidates should have low external degree), (ii) the number r of fringe candidates is limited to $r = 2$, and (iii) d_{ext} is only computed once per vertex and then cached. While the worst case complexity of HYPE is $\mathcal{O}(m \log m + nm)$, the authors note that in practice the observed running time complexity is $\mathcal{O}(m \log m + n)$, due to a constant set of $r = 2$ fringe candidates and a constant fringe set size of $s = 10$.

Table 3.1: Overview of flat, single-level partitioning algorithms. We differentiate between iterative improvement (IIP) algorithms, spectral partitioning (SP), growing heuristics (Grow), flow-based approaches (Flow), and mathematical programming (MP), as well as metaheuristics such as evolutionary algorithms (EA), simulated annealing (SA), and tabu search (TS). ID denotes the iterative deletion algorithm. We furthermore distinguish between bipartitioning (B), recursive bipartitioning (RB), and direct k-way partitioning (K) algorithms.

Algorithm	Mode	Objective	Type	Page
KL	B	$f_c(\Pi)$	IIP	38
FM	B	$f_c(\Pi)$	IIP	41
[KGV83]	B	$f_c(\Pi)$	IIP	46
LA$_\ell$-FM	B	$f_c(\Pi)$	IIP	46
k-LA$_\ell$-FM	K	$f_c(\Pi)$	IIP	47
AlgI	B	$f_c(\Pi)$	Grow	48
EV	K	$f_c(\Pi)$	EA	48
BIPART	B	$f_c(\Pi)$	EA	48
RCut1.0	B	$f_{rc}(\Pi)$	IIP	48
MCPG	K	$f_c(\Pi)$, $f_s(\Pi)$	IIP	49
WHB	B	$f_c(\Pi)$	Grow	49
EIG1	B	$f_{rc}(\Pi)$	SP	49
EIG1-IG	B	$f_{rc}(\Pi)$	SP	49
IG-Match	B	$f_{rc}(\Pi)$	SP	49

Continued on next page

Table 3.1 – *Continued from previous page*

Algorithm	Mode	Objective	Type	Page
NETPART	K	$f_c(\Pi)$	SP	50
SA-TS	K	$f_c(\Pi)$	SA+TS	50
EIG-TS	K	$f_c(\Pi)$	IIP+TS	50
[PP93]	K	$f_c(\Pi)$	IIP	50
KP	K	$f_{sc}(\Pi)$	SP	50
KC/AGG	K	$f_{sc}(\Pi)$	SP	51
k-LA$_\ell$-FM	K	$f_\lambda(\Pi)$	IIP	51
DLA	B	$f_c(\Pi)$	IIP	51
PLM/PFM	K	$f_c(\Pi)$	IIP	51
GRCA	B	$f_{rc}(\Pi)$	EA	51
SFC	K	$f_{sc}(\Pi)$	SP	52
WINDOW	K	$f_{sc}(\Pi)$	SP	52
PARABOLI	B	$f_c(\Pi)$, $f_{rc}(\Pi)$	MP	52
KDualPartFM	K	$f_c(\Pi)$	IIP	52
FBB	B	$f_c(\Pi)$	Flow	52
PANZA	B	$f_c(\Pi)$	Flow/SP	53
MELO	K	$f_{sc}(\Pi)$	SP	53
LIFO-LA$_\ell$-FM	B	$f_c(\Pi)$	IIP	53
GFM	K	$f_c(\Pi)$	IIP	54
LSMC	B	$f_c(\Pi)$	SA+IIP	54
PROP	B	$f_c(\Pi)$	IIP	54
CLIP/CDIP	B	$f_c(\Pi)$	IIP	54
LSR	B	$f_c(\Pi)$	IIP	55
ASFM	B	$f_c(\Pi)$	IIP	56
PROP-REX	B	$f_c(\Pi)$	IIP	56
DEEP/VAR-PROP	B	$f_c(\Pi)$	IIP	57
MMP	B	$f_c(\Pi)$	IIP	57
K-PM/LR	K	$f_c(\Pi)$, $f_\lambda(\Pi)$	IIP	57
IDP	K	$f_c(\Pi)$	ID	58
[Are99]	K	$f_c(\Pi)$	GRASP+IIP	58
VRW	B	$f_c(\Pi)$	IIP	59
CLIP$_2$/LIFO$_2$	B	$f_c(\Pi)$	IIP	59
SHRINK-PROP	B	$f_c(\Pi)$	IIP	60
[CY00]	B	$f_c(\Pi)$	MP	61
MDC-RS	K	$f_c(\Pi)$	IIP	61
NGSP	K	$f_c(\Pi)$	IIP	61
WalkPart	B	$f_c(\Pi)$	IIP	61
LFM	B	$f_c(\Pi)$	IIP	61
Hyper-PuLP	K	$f_\lambda(\Pi)$	IIP	62
SHP	RB/K	fanout	IIP	62

Continued on next page

Table 3.1 – *Continued from previous page*

Algorithm	Mode	Objective	Type	Page
HYPE	K	$f_\lambda(\Pi)$	Grow	62

3.4 From Two-Level To Multi-Level Partitioning

[IKS75]. Only three years after the adaptation of the KL algorithm to hypergraph partitioning through Schweikert and Kernighan [SK72], Ishiga et al. [IKS75] proposed the first two-level algorithm. The algorithm first computes a size-constrained clustering using a cluster-growing technique that favors the formation of highly connected clusters, where connectivity between a set C of vertices is defined as $f(C) := 1 - \text{cut}(C, V \setminus C) / \sum_{v \in C} d(v)$. The algorithm thus builds internally densely and externally sparsely connected clusters. After clustering, the clusters are contracted and the coarse hypergraph is initially partitioned and subsequently refined using the KL algorithm. Their implementation is furthermore able to compute k-way partitions via recursive bipartitioning and ensures feasible solutions by re-balancing both clustered and unclustered vertices. Ishiga et al. [IKS75] observe that clustering significantly reduces the number of nets of the coarser hypergraph, which makes the partitioning problem simpler. Moreover, they note that moving clusters instead of single vertices gives the KL algorithm a more global view on the partitioning problem.

> The work of Ishiga et al. [IKS75] thus already addresses the main problems of flat partitioning algorithms well before the more advanced flat techniques presented in the previous section were developed. While many of those algorithms tried to integrate some notion of foresight and locality into the partitioning process using advanced tie-breaking schemes or gain definitions, the clustering-based preprocessing technique of Ishiga et al. [IKS75] in a sense presaged the successes of the two- and multi-level algorithms developed in the 1990s.

dKLFM. Analyzing the performance of the FM algorithm for hypergraphs H with varying *network ratios* $r(H) := (p - m)/n$ experimentally, Goldberg and Burstein [GB83] show that FM performs poorly for ratios less than 3 and nearly optimally for ratios larger than 5. They note that instances derived from VLSI circuits have ratios in the range of $1.9 < r(H) < 2.5$ and therefore propose a matching-based two-level technique, with the intention to create coarser hypergraphs with increased ratio $r(H)$. The algorithm first computes and contracts a matching, then computes a random bipartition of the coarse hypergraph, and uses the FM algorithm to refine the solution. Following ideas suggested by Goldberg and Gardner [GG83], however, the algorithm does not stop after projecting the coarse solution back to the original instance, since coarsening might have contracted edges that should have been part of the cut-set. Instead, it bipartitions each of the two blocks again, computes and contracts matchings in all four blocks, and then performs another round of FM refinement in order to account for accidentally contracted cut-edges during the first bipartition.

> In a sense, Goldberg and Burstein [GB83] discovered a shortcoming of FM that also has gone largely unnoticed by researchers working on flat partitioning at that time. The network ratio parameter r is closely related to the average vertex degree $\overline{\Delta}_v$. In case of small vertex degrees and in the absence of net-weights, many vertices are likely to have similar gains – rendering KL/FM ineffective. Bui et al. [Bui+87; Bui+89] also point out that iterative improvement algorithms perform better as the average degree of the graph increases. Lengauer [Len90, p. 273] conjectures that for instances with large minimum vertex degree or for dense instances, few local minima exist that are not global minima.

STABLE. Noting that there is no constant factor error bound on the cut-size generated by maximum matching-based approaches, Cheng and Wei [WC90; CW91] use recursive ratio cut bipartitioning to identify "natural" clustering structures in the hypergraph. Cluster sizes are constrained by forcing the algorithm to divide the hypergraph in a fixed number of clusters. After forming a coarse hypergraph by contracting each cluster into a single vertex, an initial bipartition is refined using the FM algorithm. Then, contraction is undone and FM is applied again on the original hypergraph. The authors note that this approach greatly reduces the standard deviation of solution quality compared to using FM directly on the input hypergraph and to using a maximum matching-based coarsening scheme [Bui+89].

PD. Since IIP algorithms easily become trapped in local minima, and due to the fact that this effect becomes even worse in the case of direct k-way partitioning because of the increased search space, the direct k-way primal-dual (PD) algorithm of Yeh, Cheng, and Lin [YCL91a; YCL91b; YCL94; YCL95] also employs the ratio cut-based bipartitioning algorithm of Cheng and Wei [WC90; CW91] for clustering the hypergraph before partitioning. The novelty of PD, however, is a refinement algorithm that alternately uses two kinds of passes. While a *primal* pass uses a k-way extension of the traditional FM algorithm that moves single vertices, a *dual* pass employs a modified FM algorithm that uses a *net*-based move model, which allows moving *multiple* vertices at once to directly remove nets from the cut-set with one move operation. Since gain updates in this model entail gain updates for both neighbors and neighbors of neighbors of moved nets, the complexity of a dual pass becomes $\mathcal{O}(mk\Delta_e^2\Delta_v^2)$. Therefore hyperedges with a size larger than a fixed threshold are ignored in the dual passes in order to control the running time.

[HK92a]. Hagen and Kahng [HK92a] propose a two-level algorithm that tries to use as much global structural information as possible during the coarsening phase. Their $\mathcal{O}(n^3)$ time RW-ST clustering algorithm first finds cycles in a random walk of the hypergraph, uses the cycle information to compute a "sameness" score for *all pairs* of vertices, and then clusters vertices with non-zero sameness values. The FM algorithm is used to refine the initial partition of the coarse hypergraph, which in turn is then used as a starting solution for another FM pass on the original hypergraph.

A similar idea is used in the work of Alpert and Kahng [AK93], who employ the spectral AGG clustering algorithm briefly described in the previous section not only

for flat k-way partitioning, but also use the clustering of the geometrically embedded vertices in a two-level algorithm that uses FM as iterative improvement engine.

HGCEP. The hierarchical gradual constraint enforcing partitioning (HGCEP) algorithm of Shin and Kim [SK93] uses a clustering technique that iteratively merges vertices with high closeness score. Two vertices or clusters are said to be close to each other if they have many nets in common. More precisely, the closeness of two vertices/clusters u and v is defined as

$$\text{closeness(u,v)} := |\{\text{I}(v) \cap \text{I}(u)\}| / \min(\text{I}(v), \text{I}(u)) - \alpha(c(u) + c(v))/\overline{c}, \tag{3.5}$$

where $\overline{c}$ is the average cluster weight and α is a tuning parameter. The second term is hereby used to prefer the formation of balanced clusters. The clustered vertices are then contracted to form the coarse hypergraph, and a random initial partition is refined using a local search algorithm that only performs feasible vertex moves from the heavier to the lighter block in decreasing order of their gain. Additionally, Shin and Kim [SK93] account for the fact that the coarse hypergraph has *varying* vertex weights, by first using a relaxed balance constraint, which is then iteratively tightened after each pass (hence the name *gradual constraint enforcing partitioning*). This gives the algorithm more flexibility to move heavier vertices in early passes, while still ensuring feasibility at the final pass. The clustering, initial partitioning, refinement process is repeated multiple times and the best partition is again refined using the uncontracted original hypergraph. In a later work, Kim, Kim, and Shin [KKS96] improve the HGCEP algorithm by slightly modifying the closeness function and introducing an additional refinement algorithm that is used to improve the clustering. Cluster refinement is done by moving vertices from larger to smaller clusters such that the cut-size between the pair of clusters is reduced, while at the same time producing more balanced clusters.

FMC. Cong and Smith [CS93] propose one of the *first* multi-level HGP algorithms. The entire algorithm works on the clique expansion G_x of the hypergraph, in which nets with $|e| > 5$ are ignored during construction in order to reduce the size of G_x. In the coarsening phase, a clustering algorithm finds cliques of size r_0 and $r_0 + 1$ in G_x, where r_0 is an approximation of the size of the largest clique in the graph. The identified cliques are contracted if the cluster density (i.e., the ratio between the total weight of all edges within the cluster and the number of all possible edges $\binom{c}{2}$) is above a threshold and the resulting cluster does not violate predefined bounds on cluster size and weight. The density threshold is used to prevent cliques introduced by multi-pin nets from automatically being recognized as a cluster. Afterwards, r_0 is recomputed for the clustered graph and the process repeats until a sufficient number of clusters is found. In a post-processing step, yet unclustered nodes are contracted using a matching-based algorithm. After computing and refining an initial solution using the FM algorithm [FM82], the clustering is *not* reversed completely in one step. Instead, clusters are uncontracted along the clustering hierarchy and an FM-based refinement algorithm is employed (i) to improve the balance between blocks and (ii) to improve solution quality on each unclustering level.

[Yan+94]. The two-level algorithm of Yang et al. [Yan+94] uses the RW-ST clustering algorithm of Hagen and Kahng [HK92a] as a preprocessing step. The initial bipartition is then computed using a greedy growing strategy that starts with only one seed vertex in one block and all other vertices in the other block, and then repeatedly moves neighboring vertices from the larger to the smaller block in the order of decreasing gain. During this process, the algorithm keeps track of a certain number of local minima – all of which will then be used as a starting solution for refinement. The proposed refinement algorithm follows the FM paradigm with an additional restriction that only allows neighboring vertices of moved vertices to change blocks. The best of all refined initial solutions is then projected back to the original hypergraph and refined once again using the same algorithm.

BISECT. The BISECT algorithm of Saab [Saa95] identifies and contracts clusters *within* a modified FM algorithm. The key idea behind BISECT is that vertices "that are densely connected tend to stay together after a pass of an iterative algorithm if they were initially together in the same subset of the bisection" [Saa95]. BISECT starts with a random initial partition and performs several bipartition-and-coarsen phases as long as an improvement is found or contractions are possible. In each phase, a modified FM algorithm first only moves vertices in one direction until all nodes are locked or a stopping criterion applies (forward phase). Then, free vertices are moved in the opposite direction to restore balance (re-balance phase). This process is repeated until all vertices are locked. Afterwards, groups of neighboring vertices which moved *together* in a forward or re-balance phase are contracted. Following the author, we will refer to this type of coarsening algorithm as *bisect-and-compact* (BC). The idea here is similar to the motivation of the LIFO bucket management strategy proposed by Hagen et al. [HHK95a] and the CLIP heuristic of Dutt and Deng [DD96b], in that vertices that are part of a move sequence from one block to the other block may constitute a natural cluster.

> While the main motivation for previous two-phase approaches was to reduce the size of the partitioning problem and to increase the average vertex degree in order to reduce the erratic behavior FM-type algorithms, Saab [Saa95] was probably the first to point out an additional advantage of coarsening the hypergraph, namely the reduction of hyperedge sizes. Large cut-nets with many pins in both blocks of the bipartition are difficult to remove from the cut-set using vertex-move-based IIP algorithms. However, the coarsening process will make such nets successively smaller, and thus successively easier to remove. Furthermore, Saab [Saa95] also observes that contracting large clusters of vertices at once may be too aggressive and prevent refinement algorithms from finding good solutions. This especially affects rather dense instances, in which clustering algorithms are susceptible to creating wrong clusters, i.e., groups of vertices that should not all be contained in one block of the bipartition.

Strawman. Hauck and Borriello [HB95; HB97] present a comprehensive evaluation of many bipartitioning techniques existing at that time and combine the best of

them into the multi-level algorithm Strawman [Hau95]. For the coarsening phase, they evaluate a random clustering approach inspired by Bui et al. [Bui+89], the K-L clustering method of Garbers, Prömel, and Steger [GPS90], the bandwidth clustering algorithm of Roy and Sechen [RS93], along with a new connectivity clustering algorithm based on the work of Schuler and Ulrich [SU72]. We briefly explain the latter only, since it performed best in the experimental evaluation. The connectivity clustering algorithm visits all nodes in random order and clusters each vertex u with the neighbor $v \in \Gamma(u)$ with the highest connectivity score

$$\text{con}(u,v) := \frac{1}{c(u) \cdot c(v)} \frac{\psi(u,v)}{(d(u) - \psi(u,v)) \cdot (d(v) - \psi(u,v))}, \tag{3.6}$$

where $\psi(u,v) := \sum_{e \in \{I(u) \cap I(v)\}} \frac{1}{|e|-1}$ is the rating function used in the bandwidth clustering algorithm of Roy and Sechen [RS93]. Thus, while the bandwidth in the numerator favors vertex pairs that have a large number of small nets in common, the denominator favors the formation of small clusters and merging vertex pairs that are strongly connected to each other.

The coarsest hypergraph is then initially partitioned using a random initialization algorithm that first permutes all vertices and then finds the split into two blocks that gives the best balance. This technique performed better than the seeded initialization technique of Wei and Cheng [WC89], a simple BFS- and a DFS-based growing algorithm, and the spectral partitioning algorithms EIG1 [HK91a] and EIG-IG [HK92a]. In the refinement phase, the initial bipartition is then refined using a modified version of LIFO LA_3-FM [Kri84], which allows higher order gains for hyperedges that are not part of the cut-set. This refinement algorithm performed better than the primal/dual FM algorithm of Yeh et al. [YCL91a; YCL94]. Finally, Hauck and Borriello [HB95; HB97] evaluate four uncoarsening strategies (i) no uncoarsening, (ii) complete uncoarsening as in the previous two-level approaches, (iii) multi-level uncoarsening as used by Cong and Smith [CS93], and (iv) multi-level cut-net uncoarsening, which only uncontracts border vertices on each level. While multi-level approaches performed significantly better than no uncoarsening and complete uncoarsening, the experiments did not yield conclusive results on whether option (iii) or option (iv) should be preferred. The authors decided to use option (iii) in Strawman.

CMM. Cherng and Chen [CC96] use the ratio cut partitioning algorithm of Wei and Cheng [WC89] to compute a clustering and then employ a slightly simpler version of their MMP algorithm [CCH98], which has already been described in the previous section, to compute multiple bipartitions of the coarse hypergraph. The best solution is then transferred back and used as initial partition for a final refinement pass on the original hypergraph.

[AV96]. Areibi and Vannelli [AV96] propose a clustering algorithm that sequentially grows clusters around high-degree vertices by merging strongly connected neighbors to the respective seed vertex. The coarse solution is then initially partitioned using a GRASP [FR89] heuristic and refined using a simple dynamic hill climbing algorithm.

After transferring the solution back to the original hypergraph, another local search algorithm is used to refine the partition. In their experiments, the k-LA$_\ell$-FM algorithm of Sanchis [San89], a simple genetic algorithm, and a tabu search approach [AV93b] were used to perform the final refinement.

GMetis. Alpert, Hagen, and Kahng [AHK96] evaluate the use of the multi-level graph partitioning system METIS [KK95c] as a hypergraph partitioning algorithm. Before partitioning, the hypergraph is converted into a graph using a randomized clique-expansion technique that for each net e adds a unit weight edge between $5|e|$ random pairs of vertices (and ignores nets larger than 50 pins). METIS is configured to use a heavy-edge matching technique during coarsening, greedy graph growing for initial partitioning, and a modified version of the FM algorithm that is started with all boundary vertices [HL95; WCE95]. In order to make the approach more stable, METIS is embedded into a simple genetic algorithm, hence the name GMetis.

MKP. Chan, Schlag, and Zien [CSZ96a; CSZ96b; CSZ99] present a k-way multi-level spectral hypergraph partitioning algorithm for scaled-cost optimization that is able to handle arbitrary vertex weights. In contrast to previous spectral partitioning approaches which addressed vertex weights only when splitting the sorted eigenvector [HK91a], their algorithm directly incorporates weight information into the spectral partitioning process via a modified Laplacian. While coarsening and spectral initial partitioning are performed on the clique expansion G_x, the refinement algorithm directly works on the hypergraph. The coarsening algorithm is similar to the heavy edge matching algorithm of METIS [KK95a; KK98a]: Edges are visited in decreasing order of their weight and contracted if one of the endpoints has not yet been clustered. This process then continues until $n/2$ edges were merged or no more contractions are possible. For refinement, the authors repeatedly apply the RCut1.0 algorithm [WC89] on pairs of blocks.

MLH. In a follow-up work, Chan et al. [CSZ97a; CSZ97b] propose another k-way multi-level algorithm that combines spectral partitioning with iterative improvement algorithms. In MLH, the star-expansion is used as graph model for heavy-edge matching-based coarsening and spectral partitioning. The algorithm is special for three reasons: First, it uses a new k-way FM variant, which conceptually lies somewhere between Sanchis' k-LA$_\ell$-FM algorithm [San89] and the pairwise FM approach of Kernighan and Lin [KL70] that was put into practice by Cong and Lim [CL98]. While the former has to deal with $k(k-1)$ possible move directions simultaneously and the latter applies pairwise refinement to all $k(k-1)/2$ pairs of blocks sequentially, the *rotary KLFM* algorithm of Chan et al. [CSZ97a; CSZ97b] considers $2(k-1)$ possible move directions at each step. This is done by choosing each block in turn as the move target and only allowing vertices to move from all other blocks to the target bock and from the target block to all other blocks. Second, spectral information is not only used to create initial partitions, but also employed in the rotary FM algorithm to influence the direction of vertex moves (similar to CLIP encouraging moves in the same direction), to break ties for vertices with same FM gain, and as a mechanism to

escape from local minima. Third, while previous spectral/IIP hybrid algorithms use a single spectral solution as starting partition for refinement, MLH creates and refines *multiple* spectral k-way partitions. This approach is motivated by the observation that the spectral solution is far away from the original problem because of (i) the hypergraph-to-graph conversion, (ii) solving a relaxed problem for spectral partitioning, and (iii) because of multi-level coarsening.

CAMS. In contrast to previous two-level approaches that use dedicated clustering algorithms to coarsen the input hypergraph, Hagen and Kahng [HK97] propose a technique that derives the clustering directly from a certain number of FM-based partitioning solutions. Their clustered adaptive multistart (CAMS) approach starts with a set of t random solutions and works in a series of two-level FM iterations. Each iteration first coarsens the input hypergraph by contracting all vertices that were in the *same* block in *all* t previous bipartitions. Then, t new bipartitions are created by computing t initial random partitions of the coarse hypergraph which are then refined on the coarse and on the fine level via FM local search. These t new solutions are used in the coarsening process of the next iteration. This process continues until two successive iterations did not yield an improvement.

$\mathbf{ML_c}$/$\mathbf{ML_f}$. Motivated by the success of the multi-level paradigm for graph partitioning in the scientific computing community [HL95; KK95b], Alpert et al. [AHK97; AHK98] propose the ML_c and ML_f hypergraph partitioning algorithms. They note that multi-level schemes offer several advantages over two-level approaches: In a two-level scheme, a single clustering phase can lead to hypergraphs that are too coarse – limiting the potential of refinement algorithms to improve the solution. Furthermore, multi-level approaches can be very efficient, since local search algorithms already start with high quality solutions and therefore only need very few passes to converge to a local optimum at each level. Lastly, performing local search at multiple levels of the hierarchy has the benefit that bad local minima can be avoided at the coarser levels (because multiple vertices of the input hypergraph are moved at once), while optimization on the first levels of the hierarchy still allows for fine-grained optimization.

Their coarsening algorithm follows similar ideas as the heavy edge matching algorithm of Karypis and Kumar [KK95b]. However, the key difference to previous multi-level graph and hypergraph partitioning algorithms is a more fine-grained control over the number of hierarchy levels. Instead of computing *maximal* matchings, ML's coarsening algorithm only matches a fraction of vertices at each level (controlled by a *matching ratio* parameter r). This is motivated by the observation that a larger number of levels gives the refinement algorithm more opportunities to improve the solution, while at the same time, the number of passes at each level is likely to be lower because of the smaller differences between successive levels. The matching algorithm visits all vertices in random order. For each vertex u, it tries to find the unmatched vertex v that maximizes the connectivity score

$$\text{con}(u,v) := \frac{1}{c(u)\cdot c(v)} \sum_{e\in\{\text{I}(u)\cap\text{I}(v)\}} \frac{1}{|e|}, \tag{3.7}$$

where the first factor prevents vertex weights in coarser levels from becoming too imbalanced while the second factor prefers vertices connected via a large number of small hyperedges. At each level, the matching algorithm stops as soon as no more matchings are possible or $r|V^i|$ vertices are matched, where $|V^i|$ is the number of vertices on the current level i. While this coarsening scheme is employed in both ML_c and ML_f, ML_c uses the CLIP [DD96b] algorithm in the refinement phase and thus is restricted to bipartitioning, whereas ML_f is a direct 4-way partitioning algorithm that employs 4-LA_1-FM [San93] as local search algorithm to optimize the SOED metric. Alpert [Alp96, p. 198] notes that the current limitation to 4-way partitioning is because of the performance issues of k-LA_ℓ-FM [San93] due to the large number of $\mathcal{O}(k^2)$ possible move directions.

hMETIS-R. The multi-level hypergraph partitioning system hMETIS, proposed by Karypis et al. [Kar+97a; Kar+97b; Kar+99] in 1997, is one of the best-known HGP systems and is still widely used today. Since the initial version uses recursive bipartitioning to compute k-way partitions, it will be referred to as hMETIS-R throughout this dissertation.

The coarsening schemes employed in hMETIS-R are motivated by the observation that a good coarsening algorithm should (i) create small hypergraphs such that a good bipartition of the coarsest hypergraph is not significantly worse than a bipartition of the input hypergraph, and (ii) successively reduce the sizes of the hyperedges since move-based refinement algorithms perform better if most hyperedges are small. The *edge coarsening* (EC) algorithm visits each vertex in random order, and for each vertex u chooses the *unmatched* neighbor $u \in \Gamma(v)$ as contraction partner that maximizes

$$\text{con}(u, v) := \sum_{e \in \{\text{I}(u) \cap \text{I}(v)\}} \frac{\omega(e)}{|e| - 1}. \tag{3.8}$$

Thus, the algorithm prefers to contract vertices that are connected by a large number of small, heavy hyperedges.

The authors note that limiting contractions to heavy-edge maximal matchings has the drawback that only nets of size two (i.e., graph edges) can be removed from the hypergraph. Thus, the total hyperedge weight only decreases slowly during the coarsening process. They therefore propose the *hyperedge coarsening* (HEC) as well as the *modified hyperedge coarsening* (MHEC) algorithm, both of which first compute an independent set of hyperedges and then contract their pins. At each coarsening level, the nets are first sorted non-increasingly by weight, and, for nets with the same weight, non-decreasingly by size. Then, both algorithms select an independent set of hyperedges by visiting the nets in the given order, thereby implicitly preferring heavy nets of small size. Afterwards, the pins of all hyperedges contained in the independent set are contracted. The difference between HEC and MHEC is that the latter additionally employs a post-processing step which, for each hyperedge, contracts all pins that do not belong to any already contracted hyperedge. This leads to more balanced vertex weights and decreases hyperedge sizes more rapidly.

The initial partitioning phase either computes a random balanced bipartition or uses a BFS-based growing algorithm to compute an initial solution. In contrast to other multi-level algorithms, hMETIS-R not only propagates the best but *all* initial partitions to the uncoarsening phase. At each level, all solutions that are 10% worse than the best bipartition are then successively dropped. This approach is motivated by the fact that the cut-size of the coarse hypergraph is only an inaccurate representation of the cut-size of the input hypergraph. Thus, refining many alternative solutions at the coarser levels possibly allows to improve the solution of the final partition.

During the uncoarsening phase, hMETIS-R employs one of four different refinement schemes. It is possible to use the traditional FM algorithm [FM82], or an early-exit variant (FM-EE) that terminates each pass if moving one percent of the vertices did not improve the cut. Furthermore, it contains a greedy hyperedge refinement (HER) algorithm that removes hyperedges from the cut-set by moving sets of vertices at once. Lastly, it is possible to use HER followed by FM.

The greedy hyperedge refinement algorithm follows similar ideas as the dual FM algorithm of Yeh et al. [YCL91a; YCL94], i.e., hyperedges are removed from the cut-set in one step by moving all corresponding pins at once. However, in contrast to dual FM, HER is a plain greedy algorithm that trades the ability to escape from local optima for an improved running time. Each hyperedge is visited once in random order and all of its pins are moved from one block to the other block if the overall improvement is strictly positive.

Additionally, hMETIS-R employs a *multiphase refinement* technique. After computing a bipartition using the standard multi-level paradigm, it uses V-cycles to further improve the solution. Each V-cycle consists of a restricted coarsening phase (in which only vertices belonging to the same block are allowed to be contracted), which is followed by another uncoarsening/refinement phase. Restricted coarsening hereby ensures that the given partition of the input hypergraph can be used as a feasible initial partition of the coarsest hypergraph. While V-cycles always perform a complete pass through the multi-level hierarchy, v-cycles only uncoarsen half of the hierarchy levels and then immediately switch to another coarsening phase. This ensures that only the best solution is improved during the rather expensive refinement passes of the final uncoarsening levels. Figure 3.3 visualizes the different multiphase refinement schemes.

$\mathbf{ML_{AF}}$. Wichlund and Aas [WA98] propose an extension of the ML algorithm of Alpert et al. [AHK97; AHK98] that incorporates information about solutions generated in previous executions into the partitioning process, and uses an adaptive scheme to control the number of uncoarsening levels. ML's coarsening algorithm is modified to contract vertices u and $v \in \Gamma(u)$ with the highest connectivity score

$$\text{con}(u,v) := \frac{1}{c(u) \cdot c(v)} \sum_{e \in \{\text{I}(u) \cap \text{I}(v)\}} \frac{\exp(-\gamma f(e))}{|e|}, \tag{3.9}$$

where the *edge-frequency* $f(e)$ corresponds to the number of times net e appears in the cut-set of the h best solutions so far, and γ is a damping factor. This connectivity

Figure 3.3: Illustration of V-cycles (left), v-cycles (middle), and vV-cycles (right). Black lines indicate operations on the unpartitioned hypergraph, while orange lines are used to incidate operations on the partitioned hypergraph. When coarsening an already partitioned hypergraph, only vertices within the same block are allowed to be contracted in order to ensure non-decreasing solution quality.

score favors vertex pairs that are connected by low-frequency hyperedges, since nets with high edge-frequency are more likely to be part of the cut-set of high quality solutions.

> Note that this approach differs from hMETIS-R's multiphase refinement in that vertices from opposite blocks are explicitly allowed to be contracted. Therefore, it is necessary to compute new bipartitions in the initial partitioning phase as the solution from the previous iteration cannot be used as an initial partition for the current multi-level iteration.

Instead of uncontracting all vertices of a level in the uncoarsening phase, ML_{AF} only uncontracts a fraction α of all contracted vertices of the current level, preferring those with the lowest connectivity score. The value of α hereby depends on the number of refinement passes during the last t uncoarsening levels and is chosen such that an uncoarsening step provides sufficient opportunities for FM to improve the solution. The adaptive uncoarsening scheme thus allows for a more fine-grained optimization compared to standard multi-level approaches.

TLP. With the two level partitioning (TLP) algorithm, Cherng, Chen, Tsai, and Ho [Che+99] propose a two-level approach that, similar to the CMM algorithm of Cherng and Chen [CC96], uses the MMP partitioning algorithm [CCH98] to compute a bipartition of the coarse hypergraph and to further refine the projected solution on the input hypergraph. However, while the clustering phase of CMM was only based on ratio cut partitioning [WC89], the TLP algorithm employs a hybrid clustering scheme that combines ratio cut bipartitioning with a merging phase that contracts highly connected pairs of vertices.

hMETIS-K. Karypis and Kumar [KK98c; KK99; KK00] note that *direct* k-way partitioning can have several advantages over recursive bipartitioning. First, the latter does not allow to directly optimize "global" objectives such as cut-net or connectivity,

because these objectives affect all k blocks simultaneously. Second, direct k-way algorithms are able to enforce tighter balance constraints while still being able to sufficiently explore the search space of feasible solutions, whereas RB-based algorithms need to adaptively adjust their imbalance ratios at each bisection step in order to guarantee the desired final imbalance. Third, it is known that having to partition hypergraphs into two blocks of roughly equal size at each recursion level restricts the search space [ST97].

Karypis and Kumar [KK98c; KK99; KK00] therefore extend the hMETIS system with a *direct* k-way multi-level partitioning algorithm, which we will refer to as hMETIS-K in this dissertation. The hMETIS-K algorithm uses a variation of the edge-coarsening (EC) scheme called *FirstChoice* (FC), because the authors observed that by employing heavy-edge maximal matchings, the EC algorithm is likely to destroy natural clustering structures present in hypergraphs derived from VLSI circuits. The FC scheme therefore lifts the restriction that a vertex is only allowed to be matched with another unmatched neighbor. Instead, vertices are allowed to form arbitrarily-sized clusters. In order to ensure sufficiently many levels in the hierarchy, the FC algorithm is stopped at each level as soon as the number of vertices of the current hypergraph has been reduced by a predefined factor. The coarsening phase stops once the smallest hypergraph consists of $100k$ vertices. This bound is chosen in order to be able to compute balanced k-way partitions during the initial partitioning phase. While previous multi-level algorithms used rather simple initial partitioning schemes, hMETIS-K employs hMETIS-R (itself a multi-level partitioner) as initial partitioning algorithm.

Instead of using a powerful k-way local search algorithm such as k-LA$_\ell$-FM [San86] as refinement engine in the uncoarsening phase, Karypis and Kumar [KK98c; KK99; KK00] propose a simple greedy algorithm that completely gives up the ability to escape from local optima. This decision is motivated by the high complexity of FM-type direct k-way algorithms and the fact that algorithms such as k-LA$_\ell$-FM [San86] easily get trapped in local optima in a single-level context. Furthermore, the ability of FM to escape from local minima via negative gain moves is mainly seen as a means to move clusters of vertices across the cut and deemed less important in the multi-level context, since the movement of a coarse vertex in the lower levels of the hierarchy already corresponds to the movement of an entire subset of vertices of the input hypergraph. The proposed greedy k-way refinement algorithm (GkR) instead performs a number of iterations over a random permutation of the vertex set and moves boundary vertices to the adjacent block that yields the highest (strictly positive) gain. Thus, in each iteration, the greedy algorithm only considers a single move direction (out of all $k-1$ possible directions) for each vertex.

PaToH. Until 1999, research on hypergraph partitioning algorithms was almost exclusively motivated by VLSI design applications. Çatalyürek and Aykanat [ÇA99] then showed that the total communication volume of parallel matrix-vector product computations can be minimized by distributing the rows/columns of the matrix according to a connectivity-optimized k-way hypergraph partition. With PaToH

(Partitioning Tool for Hypergraphs), they therefore proposed the first HGP system which is motivated by an application from the scientific computing community.

During the coarsening phase, PaToH either employs Heavy Connectivity Matching (HCM) or Heavy Connectivity Clustering (HCC). The HCM algorithm in general works similar to the HEM algorithm employed e.g. in hMETIS-R. However, instead of using the edge weights of an implicit clique expansion to determine which vertices are matched together, the contraction partner of a vertex u is chosen to be the neighboring vertex $v \in \Gamma(u)$ with the largest number of shared nets, i.e., the connectivity between two vertices is defined as

$$\text{con}(u, v) := \sum_{e \in \{\text{I}(v) \cap \text{I}(u)\}} \omega(e). \tag{3.10}$$

Similar to hMETIS-K's first choice algorithm, the HCC algorithm does not restrict contractions to pairs of vertices and instead allows each unclustered vertex to also join an already existing cluster of vertices. The rating function of HCM is adapted to work with groups of clustered vertices. More precisely, a vertex u is contracted with either a singleton cluster (i.e., a yet unclustered vertex) or the multi-vertex cluster $C_v \in \Gamma(u)$ that maximizes

$$\text{con}(u, C_v) := \frac{\sum_{e \in \{\text{I}(u) \cap \text{I}(C_v)\}} \omega(e)}{c(u) + c(C_v)}, \tag{3.11}$$

where the denominator is used to prevent the formation of heavy clusters/vertices on the coarser levels. By favoring contraction between vertices with a large number of shared nets, both HCC and HCM try to combine rows of the matrix with similar sparsity patterns (assuming a column-net model). In order to speed-up HCC, nets larger than $4\overline{\Delta}_e$ are ignored in the current bipartitioning step, where $\overline{\Delta}_e$ is the average net size of the current hypergraph.

Once the hypergraph is small enough (e.g, $n \leq 100$), a greedy hypergraph growing (GHG) algorithm is used in the initial partitioning phase to compute an initial bipartition. The GHG algorithm is a straightforward generalization of the GGGP algorithm used in the graph partitioner METIS [KK95c]. All vertices except a randomly chosen seed vertex are put in one block of the bipartition, while the seed vertex remains in the opposite block. Then, vertices are moved to the block of the seed vertex in the order of their FM gains until the balance constraint is satisfied.

The best out of several solutions generated by the GHG algorithm is then refined during the uncoarsening phase using a boundary FM (B-FM) algorithm that only moves boundary vertices from the overloaded to the underloaded block. Similar to the early-exit FM algorithm employed in hMETIS-R [Kar+97a; Kar+97b; Kar+99] a B-FM pass is terminated when no feasible move remains or the last $\max(50, 0.001n_i)$ moves did not yield an improvement, where n_i is the number of vertices of the hypergraph at level i of the multi-level hierarchy. Furthermore, the number of B-FM passes per level is restricted to two [Çat99].

CRC. With CRC, Saab [Saa99; Saa02] adapts the BISECT algorithm to ratio cut bipartitioning. CRC also employs the bisect-and-compact (BC) paradigm that intertwines clustering with a modified two-phase FM algorithm to identify and contract natural clusters of vertices.

PART. One year after the proposal of CRC, Saab [Saa00a; Saa00b] extends the BC-scheme to a multi-level approach. Instead of only computing one clustering like CRC or BISECT, the PART algorithm creates a multi-level hierarchy of partitioned solutions. Furthermore, the two-phase FM algorithm is enhanced with techniques similar to the loose net removal (LR) and the stable net transition (SNT) algorithms of Cong et al. [Con+97a; Con+97b].

MLPart. Caldwell et al. [CKM00c] note that nontrivial interactions between different algorithmic components, as well as the lack of experimental evidence and documentation of the improvements incorporated into a framework since its initial release, complicate a faithful re-implementation of complex HGP algorithms such as hMETIS [Kar+97a; Kar+97b; KK98c; Kar+99; KK99; KK00]. They therefore propose their own multi-level bipartitioning algorithm MLPart that, at least to some extent, is deemed simpler to describe and to implement. During the coarsening phase, MLPart employs a modified edge-coarsening (EC) algorithm that measures the connectivity between two vertices u and $v \in \Gamma(u)$ as

$$\operatorname{con}(u,v) = \frac{1}{c(u)+c(v)} \sum_{e \in \{\mathrm{I}(u) \cap \mathrm{I}(v)\}} \begin{cases} 2 & \text{if } |e| = 2 \\ 1 & \text{else} \end{cases}. \tag{3.12}$$

While the first factor is used to balance vertex weights, the second fator captures the number of pins that would be removed from the hypergraph if u and v are contracted.

Coarsening stops as soon as the number of vertices is reduced to 200. Then, a *randomized engineering method* (REM) is used to compute an initial bipartition. The REM algorithm assigns vertices to blocks in decreasing order of their weights and uses a biased random selection scheme to determine the target block of a vertex. After all vertices are assigned, the initial bipartition is refined using the CLIP algorithm [DD96b]. Since computing a feasible initial partition can be difficult for hypergraphs with a large variation in vertex weights and small imbalance parameter ε, MLPart performs *two* calls to the initial partitioning algorithm. In the first call, the balance constraint is relaxed to twice the largest vertex weight, which ensures that all nodes are able to move. This partition is then used as initial solution for the second call, in which the original balance constraint is used to ensure a feasible solution.

Since CLIP [DD96b] is slower than LIFO-FM [HHK95a], only the latter is used in the uncoarsening phase to further refine the partition. The final bipartition is then improved using a single V-cycle. The coarsening phase of the V-cycle employs another modified version of the EC scheme that uses hMETIS-R's connectivity score [Kar+97a; Kar+97b; Kar+99] scaled with penalty term of $1/\sqrt{c(u)+c(v)}$ to discourage merging large clusters. Furthermore, it enforces hard constraints on the maximum cluster size at each coarsening level.

> It is probably obvious that even though MLPart is intended to be "easy to describe and implement [...]" [CKM00c] a brief description of the algorithm (and even the original description in the paper) is still not enough to allow researchers to re-implement the algorithm. This again supports our claim that for effective research on hypergraph partitioning algorithms, it is vital to make the original implementations open-source.

CoMHP. Ouyang, Toulouse, Thulasiraman, Glover, and Deogun [Ouy+00] propose a *parallel* HGP algorithm that exploits the multi-level paradigm in a very different fashion and uses cooperative search to exchange information about the current solutions between processors. Apart from working in parallel, a key difference of CoMHP to other multi-level algorithms is the way the hierarchy is built. While traditionally matching- and clustering-based coarsening algorithms are used, level i in the hierarchy of CoMHP is created by partitioning the input hypergraph into $k = n/2^i$ blocks using hMETIS-K and contracting the resulting partition. Thus, while in a traditional multi-level hierarchy all vertices of coarse hypergraph H_i are composites of vertices of hypergraph H_{i-1}, this does not have to be the case in CoMHP. After coarsening, each level is iteratively partitioned by a dedicated processor using random initial partitions which are refined with k-LA$_\ell$-FM [San86] and PLM [DA94]. Since solutions from one level can not trivially be projected to solutions of a different level, complex operators are necessary to transfer partitions between processors/levels.

LR/ESC-PM. Unlike most multi-level algorithms which settled for coarsening algorithms that only rely on local connectivity information, the edge-separability-based clustering (ESC) algorithm of Cong and Lim [CL00a; CL04] incorporates more global connectivity information into the clustering decisions by taking into account all *paths* between vertices in the clique expansion G_x of the input hypergraph. Given an edge $e = (u, v)$ of G_x, they define the *edge-separability* of e as the value of the minimum (u, v)-cut in G_x. Since computing the edge separability for all graph edges is prohibitively expensive, they instead get a tight lower bound by using the CAPFOREST subroutine of the MINCUT algorithm of Nagamochi and Ibaraki [NI92]. The coarsening process is then guided by these edge-separability estimates – preferably contracting hard-to-separate vertices. After coarsening, the LR algorithm [Con+97a] is used to compute an initial k-way partition, which is refined in the uncoarsening phase using the K-PM algorithm [CL98] that works on pairs of blocks.

MLP. Motivated by the successes of multi-level algorithms, Cherng and Chen [CC03] combine the ideas of CMM [CC96] and TLP [CC96] into the multi-level bipartitioning algorithm MLP. While the paper lacks a detailed algorithmic description of the two employed coarsening schemes, the initial partitioning phase uses random partitioning, and the MMP [CCH98] algorithm is used to improve the solution in the uncoarsening phase.

Par*k*way 1.0. Except for the parallel cooperative search algorithm of Ouyang et al. [Ouy+00] which partitions multiple levels of the hierarchy using different processors, no *actual* parallel multi-level hypergraph partitioning algorithm that employs parallelism

within the different stages of the multi-level approach existed until 2004. Although considered preliminary work [Tri06, p. 12] the Par*k*way 1.0 algorithm of Trifunović and Knottenbelt [TK04c] can therefore be seen as the first parallel HGP algorithm. The input hypergraph is distributed among t processing elements (PEs) by assigning each PE a contiguous subset of n/t vertices along with all hyperedges that are incident to a vertex of that subset (i.e., hyperedges are replicated across PEs). The FC algorithm [KK00] is executed in parallel on all PEs using

$$\text{con}(u, v) := \sum_{e \in \{\text{I}(u) \cap \text{I}(v)\}} \omega(e) \tag{3.13}$$

as the connectivity score [Tri06, p. 108]. However, only vertices local to each PE are allowed to be clustered together. This is done since the algorithm was designed to partition matrices with a distinctive lower-triangular structure for parallel sparse matrix-vector multiplication, and it was assumed (and experimentally validated) that enough strongly connected vertices are local to each PE to sufficiently coarsen the hypergraph. Contraction is also done in parallel by exchanging the clustering information in a round-robin fashion and using a hash function to assign the responsibility to contract vertices within a specific hyperedge to exactly one PE. Once the hypergraph is small enough to fit onto one machine, a single PE computes an initial k-way partition using hMETIS-K [KK98c; KK99; KK00]. During the uncoarsening phase, each PE is responsible for the vertices of k/t blocks and the corresponding incident hyperedges. Blocks are refined using the FM algorithm [FM82] if $k = 2t$, or hMETIS-K's greedy refinement [KK98c; KK99; KK00] if $k > 2t$. In case $k = t$, PEs form pairs and only one of them refines the induced bipartition using the FM algorithm [FM82].

Par*k*way 2.0. With version 2.0, Trifunović and Knottenbelt [TK04a; TK04b] improve Par*k*way such that it does not rely on structural properties of the input hypergraph during coarsening and such that the parallel k-way greedy refinement algorithm becomes both more scalable and able to effectively maintain the balance constraint.

In Par*k*way 2.0, each of the t PE stores n/t vertices and m/t nets. At the beginning of each coarsening step, an all-to-all communication is used to distribute vertices such that all nets local to each PE are complete, i.e., contain all their pins. Then, each PE locally employs the FC algorithm to compute a clustering. Since this process may cluster local vertices with vertices residing on different PEs, a two-staged process involving two all-to-all communication phases is used to communicate and coordinate non-local clustering decisions such that an upper bound on the maximum allowed coarse vertex weight is satisfied. This is different from Par*k*way 1.0, in which only local vertices were allowed to be clustered together. Furthermore, connection strength between two vertices u and v is measured as

$$\text{con}(u, v) := \frac{1}{c(u) + c(v)} \sum_{e \in \{\text{I}(u) \cap \text{I}(v)\}} \frac{\omega(e)}{|e| - 1} \tag{3.14}$$

in version 2.0 [Tri06, p. 144]. After performing all local contractions, load balancing for the next coarsening step is achieved by distributing responsibilities for parallel

hyperedge removal evenly among the processors and redistributing coarse vertices such that each PE again starts with n_i/t vertices in the subsequent coarsening step, where n_i is the number of (coarse) vertices at level i.

Once the hypergraph is small enough, each PE computes multiple initial k-way partitions using hMETIS-K as partitioning algorithm. The best partition is then improved in the refinement phase using a modified parallel version of hMETIS-K's GkR algorithm. The reason for parallelizing the greedy algorithm instead of a more sophisticated FM-based algorithm is that the latter needs priority queues and gain updates, which would generate a high communication volume in a distributed setting. In the parallel version of GkR, each PE computes the *best feasible* move for each local vertex and sends the sets U_{ij} of vertex moves with *positive* gain for moving from block i to block j to a root PE, which then determines which moves can be made without violating the balance constraint. In order to prevent move conflicts (i.e., moves in opposite directions that, while individually having a positive gain, yield a non-positive gain if both are performed) each refinement pass is further divided into two stages. The first stage only performs moves from higher to lower block numbers, the second stage considers only moves from lower to higher block numbers. Once a vertex is moved in either of the two stages, it becomes locked and is not allowed to be moved again during the current pass.

TPART. Saab's TPART algorithm [Saa04] is an advanced version of PART [Saa00a; Saa00b]. Similar to PART, TPART starts with a random initial bipartition, which is continuously refined using the bisect-and-compact approach (first introduced in the 2-level BISECT algorithm [Saa95]). The main difference between TPART and its predecessor is an improved refinement algorithm (called ALG2) that uses a Tabu Search approach to overcome the limitations of KL/FM-type algorithms that only allow each vertex to move at most once in a pass. To this end, ALG2 incorporates rules that both dynamically unlock locked vertices and allow the movement of high-gain locked vertices under certain conditions.

GA-GC-DHC. Areibi and Yang [AY04] investigate several ways to transform a simple genetic algorithm into a memetic algorithm using different techniques for local search. The best performing one, called GA-GC-DHC, is a memetic two-level algorithm that uses a not further specified clustering algorithm as a preprocessing step to coarsen the hypergraph. The initial population of GA-GC-DHC contains both random solutions as well as "good" solutions generated using the GRASP heuristic [FR89]. Individuals are refined using both the k-LA$_\ell$-FM and the MDC-RS algorithm [Are00b], which temporarily relaxes the balance constraint.

MSN. In order to optimize the total message latency of parallel matrix vector multiplications, Uçar and Aykanat [UA04] propose a communication hypergraph model, in which vertices represent primitive communication operations and nets represent processors. Hypergraphs resulting from this model thus have only very few albeit large nets ($m \leq 128$ in their experimental evaluation). Noting that the performance of recursive bisection-based algorithms degrades for hypergraphs consisting of mostly large

nets, Uçar and Aykanat [UA04] propose the MSN direct k-way multi-level hypergraph partitioner, which, to the best of our knowledge, is the only multi-level algorithm that uses Sanchis' k-LA$_\ell$-FM [San89] in the refinement phase. In the coarsening phase, MSN uses a scaled heavy connectivity matching (SHCM) algorithm, which visits all vertices in random order and for each unmatched vertex u chooses the unmatched neighbor $v \in \Gamma(u)$ as contraction partner that maximizes

$$\text{con}(u, v) := \frac{|\{\text{I}(u) \cap \text{I}(v)\}|}{d(u) + d(v) - |\{\text{I}(u) \cap \text{I}(v)\}|}. \tag{3.15}$$

The SCH connectivity score thus prefers to merge pairs of vertices that have a large number of neighbors in common [Uça99, p. 39]. The coarsest hypergraph is initially partitioned using a simple, not further specified, application-specific constructive algorithm.

Mondriaan 1.01. The Mondriaan software package proposed by Vastenhouw and Bisseling [VB05] is designed to partition rectangular sparse matrices for parallel sparse matrix-vector multiplications and contains a recursive bisection-based multi-level hypergraph partitioning algorithm. Although the name refers to the entire matrix partitioning framework, throughout this dissertation we will use it to denote the HGP algorithm. In the coarsening phase, vertices are visited in order of decreasing degree and PaToH's heavy connectivity matching scheme [ÇA99] (using the connectivity score shown in Eq. 3.10) is used to find the contraction partners of unmatched vertices. Once the hypergraph is small enough, eight random bipartitions are constructed and refined using the FM algorithm [FM82]. The best initial partition is then again refined during the uncoarsening phase using the FM algorithm [FM82].

Zoltan. The distributed hypergraph partitioning algorithm proposed by Devine, Boman, Heaphy, Bisseling, and Çatalyürek [Dev+06] is implemented in the Zoltan toolkit for load balancing and data distribution [Dev+02; Dev+09; Bom+12a]. Although Zoltan's native hypergraph partitioner is actually called PHG (Parallel HyperGraph partitioner), the name of the toolkit is commonly used as a synonym, which is why we also use the name Zoltan to refer to the partitioner. Zoltan differs from Par$k$way [TK04a; TK04b] in a number of ways. While Par$k$way uses direct k-way partitioning, Zoltan employs multi-level recursive bipartitioning. Furthermore, in Par$k$way [TK04a; TK04b], the input hypergraph is distributed one-dimensionally by assigning a contiguous subset of the vertex set to each PE and replicating the pins of incident hyperedges among the processors. Thus, each PE has complete information regarding its local vertices and incident nets. In Zoltan, each processor only has *partial* information about *some* vertices and *some* nets of the hypergraph, because it adopts a two-dimensional data distribution scheme by dividing the row-net incidence matrix of the hypergraph along both rows and columns onto the PEs (assuming that all PEs are conceptually arranged in a grid). This has the benefit of reducing the memory footprint on each PE (since no data is replicated) and allows most communication operations to be done either vertically or horizontally.

In the coarsening phase, a parallel version of HCM is employed that works in multiple rounds. In each round the globally best matching partner for a set of randomly selected, unmatched vertices at each PE is chosen collectively – employing several horizontal and vertical communication phases. Once the hypergraph is small enough, it is replicated to all PEs and a randomized greedy algorithm is used to compute an initial bipartition. The globally best solution is then chosen for the refinement phase. Unlike Par$k$way, which parallelizes hMETIS-K's simple greedy k-way refinement algorithm (GkR) [KK98c; KK99; KK00], the algorithm of Devine et al. [Dev+06] employs a parallel, two-phase version of the FM algorithm [FM82] to improve the solution. In each phase, vertices are only moved in one direction by a set of dedicated PEs to prevent moves from adversely affecting each other. Since Zoltan is based on recursive bipartitioning, the hypergraph is split into two subsets (one for each block) after computing the first bipartition, and each block is reassigned onto a separate subset of PEs, which then independently bipartition the subhypergraphs in parallel. This scheme continues until the final k-way partition is computed.

ParPaToH. The master's thesis of Karaca [Kar06] presents an unfinished attempt at creating a distributed hypergraph partitioner based on PaToH [ÇA99], which is restricted to computing k-way partitions using $t = k$ PEs, where $k = 2^i$ and i is even. Similar to Zoltan, the algorithm uses a two-dimensional data distribution scheme such that each PE only has partial information about some vertices and nets of the hypergraph. During the coarsening phase, the HCM algorithm is first executed on the local hypergraph and local rating decisions are communicated and updated among all PEs owning the same vertices. The final rating result is then redistributed to all involved PEs, and vertices are contracted according to the final matching decisions. Afterwards, subsets of the local vertex sets are exchanged with different PEs to get a more global view on the structure of the hypergraph. After coarsening, the coarsest hypergraph is redistributed to all PEs and then recursively bipartitioned in parallel in a similar fashion as in Zoltan. During uncoarsening, the crossover operations are successively reversed, and a communication-intensive pairwise FM-based refinement scheme is executed in parallel among all PEs that share the same vertex set.

NaturalPART. Li and Behjat [LB06a] propose a clustering algorithm that is used as a preprocessing phase before a partition is computed using hMETIS-R. The key idea of the algorithm is to identify and to contract natural clusters. The algorithm first orders the vertices in ascending order of either vertex degree or number of neighbors and then consecutively considers each vertex in the ordering as a potential cluster seed s. Neighboring vertices $v \in \Gamma(s)$ are considered as potential clustering candidates for s. Then, a bipartition $\Pi = (A, B)$ is formed, where $A := \{s \cup \Gamma(s)\}$ and $B := \{\Gamma(A) \setminus A\}$ (i.e., block B contains all vertices that are directly connected to but not contained in block A). In order to remove all vertices from A that are more strongly connected to vertices in B and therefore should not be part of the cluster around s, a modified FM algorithm is used to move all vertices with positive gain from A to B. Vertices that remain in A after the FM algorithm terminates then form a cluster. The process then continues with another, yet unclustered seed vertex. Once all vertices have been

visited, all clusters are contracted and the coarsened hypergraph is used as input for hMETIS-R. In subsequent works [LB06b; LBK07], the clustering algorithm is extended to also take into account the number of nets that are removed during cluster formation.

Mondriaan 2.0. In his master thesis, Leeuwen [Lee06] evaluates difference scaling factors to be used in combination with Mondriaan's default connectivity score (shown in Eq. 3.10) to account for variations in both vertex weights and net sizes. As a result of his work, scaling the connectivity score between two vertices u and v with $1/\min(d(u), d(v))$ and additionally scaling the weight of each net $e \in \{\mathrm{I}(u) \cap \mathrm{I}(v)\}$ by $1/|e|$ became the new default in version 2.0 of Mondriaan.[1] While inversely scaling with the size of nets is a commonly used technique employed by many algorithms, the degree-based scaling factor additionally prefers to merge vertices for which the incident nets of one are a subset of the incident nets of the other [Lee06, p. 24].

Fixed Vertex Support for Zoltan. In order to minimize both communication volume and migration costs of moving data in adaptive scientific computations, Çatalyürek, Boman, Devine, Bozdag, Heaphy, and Riesen [Çat+07; Çat+09] propose a new hypergraph partitioning model in which fixed vertices and additional nets are used to model the data migration. To be able to partition these models in a distributed setting, the Zoltan hypergraph partitioner is extended to handle fixed vertices. During coarsening, vertices fixed to different blocks are not allowed to be matched. If one of the contracted vertices is fixed to a particular block, the resulting coarse vertex is fixed to the block as well. During initial partitioning, the randomized greedy hypergraph growing algorithm ensures that fixed vertices stay assigned to their respective block. During refinement, fixed vertices are not allowed to be moved to different blocks. Since Zoltan employs recursive bipartitioning to compute k-way partitions, fixed vertex information is incorporated into the bipartitioning process by assigning vertices originally fixed to blocks $1 \leq i \leq k/2$ to block 1, and the remaining vertices originally fixed to blocks $k/2 < i \leq k$ to block 2. This scheme then continues recursively.

kPaToH. Aykanat et al. [ACU08] state that partitioning problems involving multiple constraints or fixed vertices should be solved using a *direct* k-way approach. For multi-constraint partitioning, RB-based approaches (such as Zoltan and PaToH) are said to have problems computing feasible partitions for hypergraphs with large variations in vertex weights, while for partitioning hypergraphs with fixed vertices, the procedure described in the previous paragraph is said to restrict the search space of possible fixed vertex assignments. Aykanat et al. [ACU08] therefore propose the multi-level direct k-way algorithm kPaToH. During coarsening, it uses PaToH's HCM algorithm and disallows any contractions of pairs of fixed vertices. Before initially partitioning the coarsest hypergraph using PaToH, all fixed vertices are temporarily removed. Afterwards, the fixed vertices are then optimally assigned to the blocks of the k-way partition by solving a maximum-weighted bipartite graph matching problem whose

[1] `https://git.science.uu.nl/R.H.Bisseling/mondriaan/blob/master/docs/USERS_GUIDE.html`

solution defines a relabeling of the free vertices to the fixed vertex blocks that minimizes the connectivity metric.

During uncoarsening, kPaToH employs a modified version of the greedy k-way refinement algorithm employed in hMETIS-K, which allows each non-fixed boundary vertex to be moved a certain number of times during each refinement pass.

Par*k*way 2.1. In the journal version of earlier work [TK04a; TK04b], Trifunović and Knottenbelt [TK08] give a more detailed description of version 2.0 and extend it to also include parallel multi-phase refinement, which, however, differs in two major ways from the scheme employed in hMETIS-R. First, unlike the V- and v-cycles employed in hMETIS-R, Par$k$way's multi-phase refinement does not ensure non-decreasing solution quality, because the previous solution is only respected during the restricted coarsening phase. Once the hypergraph is small enough, a new initial k-way partition is computed using the coarsened hypergraph as input. Second, this technique is employed *at each level* of the hierarchy during the uncoarsening phase, slowing down the algorithm by an order of magnitude.

onmetisHP. Noting that the modeling flexibility of hypergraphs comes at the cost of inherently more complicated algorithms and increased running times when compared to graph-based approaches, Kayaaslan, Pinar, Çatalyürek, and Aykanat [Kay+10; Kay+11; Kay+12] try to find a trade-off by solving the hypergraph partitioning problem through graph partitioning on the net intersection graph G_n. However, unlike Kahng [Kah89] (AlgI), Cong et al. [CHK92] (IG-Match), and Cong et al. [CLS94; CLS96] (KDualPartFM), who use graph partitioning by edge separator (GPES) followed by a post-processing phase to complete the induced partial hypergraph partitions, Kayaaslan et al. [Kay+10; Kay+11; Kay+12] propose a *post-processing-free* approach based on recursive bipartitioning and graph partitioning by vertex separator (GPVS). By using an approximate weighting scheme for the nodes in G_n, their algorithm is able to capture the balance constraint on the hypergraph partition to some extent. Furthermore, sophisticated node-removal and node-splitting techniques allow for both cut-net and connectivity optimization. The ONMETIS ordering code of METIS [Kar13] is adapted and modified for implementing the GPVS-based hypergraph partitioning approach. Since onmetisHP cannot guarantee feasible solutions in general, it is mainly interesting for applications in which the notion of imbalance is not well-defined [Kay13, p.65].

UMPa. Motivated by the 10th DIMACS Implementation Challenge on Graph Partitioning and Clustering [Bad+13], Çatalyürek et al. [Çat+12b; Çat+15] propose UMPa, a direct k-way, multi-objective, multi-level hypergraph partitioning algorithm that uses a directed hypergraph model to minimize the maximum communication volume. During coarsening, UMPa uses the FC algorithm. The coarsest graph is then partitioned using PaToH as initial partitioning engine, and the initial solution is further refined using a modified version of kPaToH's multi-move greedy k-way refinement algorithm, which also takes into account two other communication cost metrics via a tie-breaking scheme.

[Çat+12a]. Çatalyürek et al. [Çat+12a] present three different, shared-memory parallel clustering algorithms that can be used in the coarsening phase of multi-level HGP algorithms. While the first one is a straight-forward parallelization of the HCM algorithm using atomic locks as synchronization mechanism, the second one performs HCM in parallel and uses a conflict resolution scheme to resolve cases in which vertices ended up having multiple matching partners. The third algorithm is a straight-forward locking-based parallelization of HCC.

INR/IVR. Deveci, Kaya, and Çatalyürek [DKÇ13] investigate different hypergraph sparsification techniques which can be used in a preprocessing phase to speed up the partitioning process. While *identical net removal* (INR) is commonly used in the coarsening phase (e.g., in PaToH [ÇA99] and Zoltan [Dev+06]), *identical-vertex* (IVR) and *similar net removal* (SNR) have not been studied in the partitioning context. Instead of employing the commonly used approach of fingerprinting, sorting and scanning [Ash95], Deveci et al. [DKÇ13] propose a $\mathcal{O}(p)$-time hashing-based algorithm which can be used for both identical-net and identical-vertex removal. Similar nets are detected using an approach based on min-hash fingerprints [Bro97]. While INR and IVR gave a speedup of 1.18-3.3 *and* slightly improved solution quality on average, SNR only resulted in a slight additional improvement in running time and reduced the solution quality.

Mondriaan 4.0. Fagginger Auer and Bisseling [FB14] revisit Mondriaan's matching-based coarsening algorithm and propose an improved version that increases the matching quality by employing the $\frac{1}{2}$-approximation PGA' algorithm of Drake and Hougardy [DH03a; DH03b], and using a modified tie-breaking rule that prefers low-degree vertices. They furthermore present two approaches to speed up the matching computation for hypergraphs with large nets by approximating $\{\mathrm{I}(u) \cap \mathrm{I}(v)\}$ when trying to find a matching partner $v \in \Gamma(u)$ for an unmatched vertex u.

FEHG. The Feature Extraction Hypergraph Partitioning (FEHG) algorithm proposed by Lotfifar and Johnson [LJ15; LJ16] tries to integrate both local and global structural information of the hypergraph into the coarsening process by transferring ideas from rough set theory [Paw92] to hypergraph partitioning. The algorithm first computes a clustering of the hyperedges. Less important nets are then discarded by representing each cluster using only a single net. The clustering is computed by identifying connected components in a hyperedge connectivity graph. It contains a node for each hyperedge, and two nodes are connected if the corresponding hyperedges e_1 and e_1 are similar, where similarity is defined as having a Jaccard index $J(e_1, e_2) := |e_1 \cap e_2|/|e_1 \cup e_2|$ (which is appropriately scaled to account for hyperedge weights) above a certain threshold s. The algorithm then computes equivalence classes of vertices that are indiscernible with regard to the hyperedge clustering (i.e., they belong to the same clusters). Since the equivalence classes are computed with regard to the entire clustering, they therefore provide some information about the global structure of the hypergraph. Vertices within the same equivalence class are then matched with their most similar neighbor (again using a weighted version of the Jaccard index

as similarity measure), and the contracted matching forms the hypergraph for the next coarsening level. Once the hypergraph is small enough, three different algorithms (random partitioning, a growing technique, and an FM-based partitioning approach) are used to compute an initial partition, which is then refined using the boundary FM algorithm in the uncoarsening phase.

Mondriaan 4.2. In his master thesis, Oort [Oor17] proposes five improvements to the Mondriaan system: (i) An improved implementation of a non-recursive, three-way quick-sort, (ii) an improved gain bucket structure that uses an array instead of a linked list to represent the gain buckets, (iii) an improvement in the implementation of the PGA' algorithm for marking matched vertices, (iv) an algorithm that searches for bipartitions with a cut-size of zero by computing connected components and solving a variant of the subset sum problem, and (v) two algorithms that improve the balance of an existing bipartition by moving those pins of cut-nets that do not have an effect on solution quality. Furthermore, a variation of Mondriaan's coarsening scheme is proposed that allows the contraction of clusters instead of pairs of vertices. Most of these improvements seem to have been integrated into version 4.2 of Mondriaan.[2]

Zoltan-AlgD. Similar to FEHG, the work of Shaydulin and Safro [SS18b; SS18a] and Shaydulin, Safro, and Chen [SSC19] is also motivated by the idea of improving the coarsening phase by incorporating global information into the matching process. The authors extend Zoltan's HCM-based coarsening algorithm to penalize cutting hyperedges that contain similar vertices. This is done by introducing a new vertex-similarity measure called *algebraic distance for hypergraphs* that takes into account distant neighborhoods of vertices, and considers two vertices to be similar if they have similar neighborhoods. The similarity information is incorporated into the coarsening process by scaling the weight of each net with the ratio between the computed algebraic weight of the net, and the average algebraic weight of all nets. The algebraic weight of a net is hereby defined as the inverse of the distance between the two pins of the net that are farthest apart from each other with regard to the algebraic distance. Thus, structural information about the hypergraph is passed to Zoltan's HCM coarsening algorithm indirectly via the modified hyperedge weights. The initial partitioning and uncoarsening phase of Zoltan-AlgD are the same as in Zoltan.

[PS19]. Recently, Preen and Smith [PS18; PS19] used the hypergraph partitioning framework developed in this dissertation to evaluate the usage of evolutionary algorithms as initial partitioning engines in the context of less coarsened hypergraphs. The work is motivated by the fact that all multi-level HGP systems stop the coarsening process at some experimentally-derived predefined threshold. The authors instead propose an adaptive coarsening scheme that stops the coarsening process once contractions start to reduce the number of pins significantly. The idea is to find a "tipping point at the best balance between maximal information content and maximal hypergraph compression" [PS19]. Since the coarsest hypergraphs produced by this adaptive scheme are significantly larger then what traditional initial partitioning heuristics are tuned

[2] `http://www.staff.science.uu.nl/~bisse101/Mondriaan/`

for, EA-based partitioning techniques are shown to be able to explore the larger search spaces more effectively at the cost of an approximately ten-fold increase in running time.

ReBaHFC. Gottesbüren, Hamann, and Wagner [GHW19a; GHW19b] recently generalized the FlowCutter graph bipartitioning algorithm [HS18a] to the HyperFlowCutter algorithm for hypergraph bipartitioning. It uses a series of incremental max-flow min-cut computations (emulating Dinic's max-flow algorithm [Din70] directly on the hypergraph using an approach first proposed by Pistorius and Minoux [PM03]) to optimize balance and cut size in the Pareto sense (similar to FBB [YW94; YW96; YW08]). The proposed ReBaHFC algorithm then uses HyperFlowCutter as a refinement/post-processing algorithm to improve hypergraph bipartitions initially computed via PaToH.

Table 3.2: Overview of two-level hypergraph partitioning algorithms. We distinguish between bipartitioning (B), recursive bipartitioning (RB), and direct k-way partitioning (K) algorithms.

Algorithm	Page	Reference	Clustering	Partitioning	Mode	Objective
[IKS75]	65	[IKS75]	Connectivity Clustering	KL	K	$f_s(\Pi)$
dKLFM	65	[GB83]	Matching	FM	B	$f_c(\Pi)$
STABLE	66	[WC90; CW91]	Ratio Cut	Ratio Cut FM	B	$f_c(\Pi)$
PD	66	[YCL91a; YCL94]	Ratio Cut	Primal/Dual FM	K	$f_s(\Pi)$ $f_c(\Pi)$
[HK92a]	66	[HK92a]	RW-ST	FM	B	$f_c(\Pi)$
[AK93]	66	[AK93]	AGG	FM	B	$f_c(\Pi)$
HGCEP	67	[SK93; KKS96]	Closeness Clustering	GCEP	K	$f_c(\Pi)$
[Yan+94]	68	[Yan+94]	RW-ST	Unnamed	B	$f_{rc}(\Pi)$
BISECT	68	[Saa95]	BC	2-phase FM	B	$f_c(\Pi)$
CMM	69	[CC96]	Ratio Cut	MMP	B	$f_c(\Pi)$
[HK97]	71	[HK97]	CAMS	FM	B	$f_c(\Pi)$ $f_{rc}(\Pi)$
TLP	74	[Che+99]	Hybrid Ratio Cut	MMP	B	$f_c(\Pi)$
CRC	77	[Saa99; Saa02]	BC	2-phase FM	B	$f_{rc}(\Pi)$
GA-GC-DHC	80	[AY04]	Unknown	k-LA$_\ell$-FM, MDC-RS	K	$f_c(\Pi)$
NaturalPart	82	[LB06a]	FM-based	hMETIS-R	RB	$f_c(\Pi)$

Table 3.3: Overview of multi-level hypergraph partitioning algorithms. We distinguish between bipartitioning (B), recursive bipartitioning (RB), and direct k-way partitioning (K) algorithms. Algorithms marked with † are distributed/parallel.

Algorithm	Page	Coarsening	Initial Partitioning	Refinement	Mode	Objective
FMC	67	Cliques (G_x)	Random	FM	B	$f_{rc}(\Pi)$, $f_c(\Pi)$
Strawman	68	Connectivity	Random	LA_3-FM	B	$f_c(\Pi)$
GMetis	70	HEM	GGGP	B-FM	B	$f_c(\Pi)$
MKP	70	HEM (G_x)	KP	RCut1.0	K	$f_{sc}(\Pi)$
MLH	70	HEM (G_*)	Spectral (G_*)	Rotary KLFM	K	$f_c(\Pi)$
ML_c/ML_f	71	HEM	Random	CLIP, k-LA_1-FM	B/K[3]	$f_c(\Pi)$, $f_s(\Pi)$
hMETIS-R	72	EC, HEC, MHEC	Random, BFS	FM, FM-EE, HER	RB	$f_c(\Pi)$
ML_{AF}	73	HEM	Random	FM	B	$f_c(\Pi)$
hMETIS-K	74	FC	hMETIS-R	GkR	K	$f_c(\Pi)$, $f_s(\Pi)$
PaToH	75	HCM, HCC	GHG	B-FM	RB	$f_c(\Pi)$, $f_\lambda(\Pi)$
PART	77	BC	Random	2-phase FM	B	$f_c(\Pi)$
MLPart	77	PinEC, EC	REM+CLIP	LIFO-FM	B	$f_c(\Pi)$
CoMHP	78	hMETIS-K	Random, hMETIS-K	k-LA_ℓ-FM, PLM	K	$f_c(\Pi)$
LR/ESC-PM	78	ESC	LR	K-PM	K	$f_c(\Pi)$, $f_s(\Pi)$

Continued on next page

[3] Only supported for $k = 4$.

Table 3.3 – *Continued from previous page*

Algorithm	Page	Coarsening	Initial Partitioning	Refinement	Mode	Objective
MLP	78	Unknown	Random	MMP	B	$f_c(\Pi)$
Par$k$way 1.0†	78	FC	hMETIS-K	GkR, FM	K	$f_\lambda(\Pi)$
Par$k$way 2.0†	79	FC	hMETIS-K	GkR	K	$f_\lambda(\Pi)$
TPART	80	BC	Random	ALG2	B	$f_c(\Pi)$
MSN	80	SHCM	Constructive	k-LA$_\ell$-FM	K	$f_\lambda(\Pi)$
Mondriaan 1.01	81	HCM	Random	FM	RB	$f_\lambda(\Pi)$
Zoltan†	81	HCM	Random	2-phase FM	RB	$f_\lambda(\Pi)$
ParPaToH†	82	HCM	PaToH	Pairwise FM	K	$f_\lambda(\Pi)$
kPaToH	83	HCM	PaToH	Multi-Move GkR	K	$f_\lambda(\Pi)$
Mondriaan 2.0	83	HCM	Random	FM	RB	$f_\lambda(\Pi)$
Par$k$way 2.1†	84	FC	hMETIS-K	GkR	K	$f_\lambda(\Pi)$
onmetisHP	84	ONMETIS	ONMETIS	ONMETIS	RB	$f_c(\Pi)$, $f_\lambda(\Pi)$
UMPa	84	FC	PaToH	Multi-Move GkR	K	$f_\lambda(\Pi)$
Mondriaan 4.0	85	PGA'	Random	FM	RB	$f_\lambda(\Pi)$
FEHG	85	RSC	Random, FM-based, Linear	B-FM	RB	$f_\lambda(\Pi)$
Mondriaan 4.2	86	PGA'	Random	FM	RB	$f_\lambda(\Pi)$
Zoltan-AlgD	86	AlgD-HCM	Random	2-phase FM	RB	$f_c(\Pi)$
ReBaHFC	87	PaToH	PaToH	PaToH, HyperFlowCutter	B	$f_c(\Pi)$

3.5 A Taxonomy Of State-Of-The-Art HGP Systems

As we have seen over the course of this chapter, a large number of hypergraph partitioning algorithms and systems has been proposed since the seminal work of Kernighan and Lin [KL70]. However, "although hypergraph partitioning is widely used in both academia and industry, the number of publicly available tools is limited" [ACU08]. This section therefore is intended to give a brief overview about today's landscape of HGP tools for solving general hypergraph partitioning problems. Furthermore, we highlight the fact that many implementations have undergone *significant* changes and improvements since their initial publication.

The Current Tool Landscape. Table 3.4 shows a taxonomy of all state-of-the-art HGP algorithms and gives an overview over algorithmic design decisions, as well as supported features and objective functions. We use ◐-marks to indicate that, because of the close relationship between the sum-of-external-degrees objective $f_s(\Pi)$ and the connectivity objective $f_\lambda(\Pi)$, one metric can be implicitly targeted by optimizing the other.[4] Note that only hMETIS supports both RB-based and direct k-way partitioning and thus is the only direct k-way partitioning algorithm that is able to optimize the cut-net metric $f_c(\Pi)$. All other direct approaches target the connectivity metric.

Implementation Complexity. Caldwell et al. [CKM00c] already noted in the year 2000 that a "lack of documented key implementation details in the literature [...], and the implementation complexity of hMETIS techniques, may be factors contributing to the lack of integration of hMETIS-quality partitioning methods in the VLSI community." Since then, the discrepancy between the initial implementations described in the literature – which were already "almost never described in sufficient detail for others to reproduce results" [CKM99b] – and the current versions, has only gotten larger. As of 2019, the latest version of hMETIS (version 2.0pre1, released in May 2007) has been improved in several major ways. The change history[5] states that the current version e.g. now contains new coarsening schemes that improve the solution quality, better refinement algorithms for hypergraphs with non-unit vertex weights, a KPM-based k-way refinement algorithm, and a new way how to use the V-cycle refinement. However, there is no experimental evidence that shows the effectiveness of these improvements.

While the initial release of PaToH contained one matching-based and one clustering-based coarsening algorithm, a single initial partitioning, and a single refinement algorithm, the current version (v3.2) includes 17 different coarsening algorithms, 13 initial partitioning algorithms, 12 refinement algorithms, and a vast number of additional tuning parameters [ÇA11a], most of which are scarcely documented in the user manual [ÇA11b]. In order not to force the algorithm configuration onto the

[4] SHP optimizes fanout, which is closely releated to both connectivity and SOED optimization.
[5] Available online at `http://glaros.dtc.umn.edu/gkhome/metis/hmetis/changes`

Table 3.4: Taxonomy of today's hypergraph partitioning tools.

		hMETIS	PaToH	Par*k*way	Mondriaan	Zoltan	kPaToH	UMPa	SHP	Zoltan-AlgD	HYPE
Design	Multi-Level	●	●	●	●	●	●	●	○	●	○
	Recursive	●	●	○	●	●	○	○	●	●	○
	Direct	●	○	●	○	○	●	●	●	○	●
	Parallel	○	○	●	○	●	○	○	●	○	○
Features	Vertex Weights	●	●	●	●	●	●	●	?	●	○
	Net Weights	●	●	●	○	●	●	●	?	●	○
	Fixed Vertices	●	●	○	○	●	●	○	○	?	○
Metrics	Cut-Net $f_c(\Pi)$	●	●	○	●	●	?	○	○	●	○
	Connectivity $f_\lambda(\Pi)$	◐	●	●	●	●	●	●	◐	●	●
	SOED $f_s(\Pi)$	●	◐	◐	◐	◐	◐	◐	◐	◐	◐
	Language	C	C	C++	C	C	C	C++	Java	C	C++
	Open Source	○	○	●	●	●	○	○	●	●	●

user, PaToH provides three different presets (default, speed, quality). However, the configurations employed in the presets are not documented in the manual. Given the fact that both PaToH and hMETIS – still the most commonly used HGP systems – are closed source, and the fact that evaluations of the improvements are missing, researchers wanting to compare their algorithms are thus best advised to use the configurations provided by the current releases. Although the discrepancy is more severe for closed-source tools, it also affects open-source implementations. For Zoltan, the "basic algorithm remains the same, though several improvements have been made over the years" [RB12], while several of Mondriaan's improvements are only documented in master's theses [Lee06; Oor17]. SHP, on the other hand, is only released as a patch to the Apache Giraph [Fou] framework and has "neither configuration files and parameters, nor scripts, execution instructions, or documentations" [May+18], which makes it almost impossible to reproduce the results.

A Note Of Caution. Differences between versions, however, are not the only problems encountered when working with HGP systems. Even if parameter presets are used, special care has to be taken when configuring the algorithms for different objective functions. This is particularly true for the recursive bipartitioning algorithm hMETIS-R. As we will see in Section 4.2, in RB-based HGP algorithms it is necessary to employ cut-net splitting for connectivity/SOED optimization, and cut-net removal for cut-net optimization. The decision which technique is applied, however, is *not* automatically coupled with the respective objective function in hMETIS-R. Thus,

without explicitly setting the `reconst` parameter [KK98b] to enforce cut-net splitting, solution quality for SOED optimization may degrade significantly. Furthermore, for historic reasons, hMETIS-R uses a different notion of imbalance. An imbalance value of 5, for example, allows each block to weigh between $0.45 \cdot c(V)$ and $0.55 \cdot c(V)$, which corresponds to an allowed imbalance of $\varepsilon = 0.1$ in our problem definition. Moreover, this imbalance is applied *at each bipartitioning step*. Thus, the largest block in a 8-way partition for example in this case is allowed to weigh up to $0.55^3 \cdot c(V)$. In the now standard HGP formulation, this in turn corresponds to an allowed imbalance of $\varepsilon = 0.331$.

When using Mondriaan as a hypergraph partitioner, "[p]roviding the vertex weights is essential, because otherwise Mondriaan will by default weigh all the columns by the number of nonzeroes contained in them, which will lead to unbalanced hypergraph partitions."[6]

3.6 Epilogue – Lessons Learned

Having discussed the main contributions of roughly 50 years of hypergraph partitioning research, we end this journey by summarizing the findings and extracting some guiding principles for effective hypergraph partitioning algorithms. Although flat hypergraph partitioning has recently again gained interest in the area of fast social network partitioning [Kab+17; May+18], multi-level algorithms still dominate the high-quality regime. We therefore first turn to the three phases of the multi-level paradigm before discussing the different approaches to compute k-way partitions.

3.6.1 The Coarsening Phase

While coarsening is mostly said to successively *approximate* the problem, Walshaw [Wal03] highlights the fact that it also *filters* the search space by restricting the solutions that can be visited during refinement: "In other words, by filtering a large amount of irrelevant detail from the solution space (in particular the higher cost solutions which are not close to local optima), the multilevel component allows the refinement algorithm to find regions of the solution space where the objective function has a low average value (e.g. broad valleys)" [Wal03, p. 74].

In the coarsening phase, highly-connected vertices are identified and contracted to create successively coarser hypergraphs. While for graphs local connectivity between two vertices u and v is captured by the weight of the corresponding edge (u, v), the situation is different for hypergraphs. Since hyperedges can have an arbitrary number of pins, two vertices may be connected by several, differently-sized nets. This offers more room for different notions of connectivity (incorporating information about both net weights and net sizes) and explains the variety of rating functions proposed in the literature.

[6] `https://www.staff.science.uu.nl/~bisse101/Mondriaan/Docs/HYPERGRAPH.html`

However, there exist some underlying design goals that are addressed by all coarsening algorithms: (1) Coarse hypergraphs should remain *structurally similar* to the input hypergraph (i.e., retain its important features [Wal03]) to allow initial partitioning algorithms to compute high-quality solutions such that a partition of the coarsest level is not significantly worse than a partition obtained on the input hypergraph [Kar02; Kar03]. (2) In order to improve the ability of move-based local search algorithms to escape from local optima, coarsening should successively reduce the size of the nets, since IIP refinement algorithms are more effective for hypergraphs with small hyperedges [KK99]. (3) Coarsening should reduce the exposed hyperedge weight and remove as many hyperedges as possible that can potentially be cut. Since single-vertex hyperedges cannot be cut and are therefore removed from the hypergraph, this reduces the search space of good solutions and thus leads to simpler instances for initial partitioning [Kar02; Kar03]. (4) The coarsening algorithm should create smaller instances such that solutions of the coarse hypergraphs induce feasible solutions on the input instance [Wal03]. (5) Moreover, the cost of the coarse solution should be the same as the cost of the induced solution on the original hypergraph, in order to ensure that coarsening truly acts as a search space filter by restricting the solutions that can be visited by the refinement algorithm [Wal03]. Furthermore, Hauck and Borriello [HB97] note that (6) a good coarsening algorithm should increase the degrees of coarse vertices to improve the performance of IIP algorithms, which are known to be ineffective for hypergraphs with small vertex degrees [GB83]. Caldwell et al. [CKM00e] additionally highlight the fact that (7) vertex weights of coarser hypergraphs should be balanced, because most move-based algorithms are ill-suited for hypergraphs with large variance in vertex weights (especially for partitioning problems with small imbalance). Lastly, Alpert et al. [AHK97; AHK98] emphasize that (8) the depth of the multi-level hierarchy represents a trade-off between solution quality and running time.

> It is interesting to note that while both matching and clustering algorithms were already employed in some of the first multi-level algorithms for hypergraph partitioning [HB95; AHK97], multi-level graph partitioning algorithms mostly relied on the former [KK95a; KK98a] until the partitioning of scale-free networks with highly irregular structure became a focus of investigation [MSS14; MSS16]. Before that, GP mainly targeted instances with rather regular structure, such as graphs derived from finite element meshes. Thus, while the idea of employing clustering algorithms for coarsening is relatively new in the field of graph partitioning, its effectiveness for irregular problems has already been well understood in the hypergraph partitioning community.

3.6.2 Initial Partitioning and Refinement

Since the coarsest hypergraph is only a rough approximation of the original hypergraph, computing exact solutions in the initial partitioning phase does not result in significant gains [BS11, p. 71] and the best solution on the coarsest hypergraph does not necessarily

translate to the best solution on the input hypergraph. Therefore, existing partitioning systems employ a variety of simple algorithms to compute an initial solution and leave further optimization to the refinement phase.

"Even though sophisticated refinement algorithms can be integrated into the multi-level scheme, we note that current multilevel implementations seem to favor simpler refinement algorithms that possess significantly shorter run-times" [Tri06, p. 75]. While shorter running times are certainly a reason to favor simple algorithms over the more sophisticated flat approaches presented in Section 3.3, we believe the main reason to be a different one: Coarsening algorithms strive to create instances for which KL/FM-type algorithms are known to be effective (i.e., hypergraphs with large minimum vertex degree [Len90, p. 273][Bui+87; Bui+89] and small nets [Kar03, p. 147]). Furthermore, refinement works at multiple levels of granularity during the uncoarsening phase and thus allows to effectively move clusters of vertices at once on coarser levels, while still ensuring fine-grained optimizations on finer levels. Therefore, the multi-level paradigm implicitly addresses the main shortcomings of move-based KL/FM-type algorithms.

3.6.3 Computing k-way Partitions

Multi-way partitions can either be computed via recursive bipartitioning (RB) or direct k-way partitioning. To this day, it remains controversial which approach should be preferred in practice [CL98; ACU08; KB18], because both approaches come with their own problems. The RB-approach acts greedily and lacks global information. As already noted by Kernighan and Lin [KL70], a good solution for the first bipartition divides the hypergraph into two densely connected blocks and thus makes it more difficult to find smaller cuts in later levels of the partition tree. Thus, high-quality solutions on the first recursion levels do not necessarily translate to high-quality k-way partitions. In fact, it has been shown by Simon and Teng [ST97] that RB can produce partitions that are very far away from optimal in the worst case. Additionally, RB-based algorithms are unable to accept less attractive solutions at early bipartitioning steps that would lead to net savings later on [HL93]. Furthermore, the performance of RB degrades for hypergraphs with large nets [UA04], since it becomes difficult for move-based local search algorithms to find meaningful moves that improve solution quality, because large nets are likely to have many pins in both blocks of the bipartition (especially in the first couple of bipartitions). In order to compute feasible solutions under tight balance constraints, RB-based algorithms need to adaptively adjust their imbalance ratios at each bipartitioning step, which can become problematic when partitioning hypergraphs with a large variance in vertex weights [ACU08]. Furthermore, the RB paradigm is not considered appropriate for variations of the standard hypergraph partitioning problem (e.g., multiple constraints, multiple objectives, or fixed vertices) [KK99; ACU08]. Moreover, objectives functions such as the connectivity metric $f_\lambda(\Pi)$ that take into account the connectivity of cut-nets can only be optimized implicitly using the RB-approach by splitting cut-nets after each bipartition [ÇA99]. This in turn restricts the search space because connections to vertices in other blocks are ignored [Das93, p. 39].

Despite these shortcomings, the RB-approach is widely used in practice [Kar+97a;

ÇA99; VB05; Dev+06; Kab+17] because of its computational simplicity and cost-effectiveness [CL98]. Furthermore, direct k-way refinement algorithms [San89] are considered highly susceptible to becoming trapped in local minima that are far away from being optimal [Lim00, p. 39][CL98; KK99; Are01]. Due to the large number of potential move candidates and the $k(k-1)$ possible move directions, they are prone to making wrong decisions [Lim00, p. 40]. Furthermore, direct k-way algorithms are considered "complex to describe and to implement" [BS11, p. 72]. While they seem beneficial for optimizing metrics such as the connectivity objective $\mathfrak{f}_\lambda(\Pi)$, optimizing the cut-net objective $\mathfrak{f}_c(\Pi)$ becomes more difficult in a direct k-way setting, especially for hypergraphs with large nets, in which case the gain of moving a single vertex to another block is likely to be zero [MP14], because large hyperedges tend to connect multiple blocks. Lastly, Buntine et al. [Bun+97] noticed a subtle but significant difference between 2-way and k-way KL/FM-type algorithms: In 2-way partitioning, the final state of a pass is a *mirror* of the initial state, since on termination of a pass the algorithm has swapped both blocks (assuming all vertices are moved). Thus both at the beginning *and* at the end of a pass, a 2-way algorithm makes *local* changes to the starting solution, whereas for k-way partitioning, "the states just drift away from the initial state and a second local move is not created at the end of the pass" [Bun+97]. While 2-way algorithms thus first walk away from the initial solution and then return at the end of a pass, k-way algorithms can easily get lost in the search space.

Chapter 4

n-Level Hypergraph Partitioning

4

"All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer."

— IBM Maintenance Manual, 1925

In this chapter, we present the algorithmic contributions that form the core of the n-level hypergraph partitioning framework KaHyPar (**Ka**rlsruhe **Hy**pergraph **Par**titioning). The chapter is based on four conference publications [Sch+16a; Akh+17a; HS17a; HSS18a] and one journal paper [HSS19a].

Historical Perspective. The original idea of generalizing the n-level approach from graph partitioning [OS10a; OS10b] to hypergraph partitioning came from Peter Sanders. The bachelor thesis of Florian Ziegler [Zie12] can be seen as the starting point of his group's interest in hypergraph partitioning. The thesis was jointly supervised by Peter Sanders, Vitaly Osipov, and Christian Schulz, and presents an approach to hypergraph bipartitioning that contracts *one hyperedge* in each level of the hierarchy. However, this system was two orders of magnitude slower than hMETIS and unable to compute solutions of higher quality, prompting the decision to start from scratch. Joint work of the author of this dissertation with Vitali Henne, Henning Meyerhenke, Peter Sanders, and Christian Schulz lead to the second attempt – a direct k-way n-level hypergraph partitioner that contracted pairs of vertices [Hen+15a]. Despite several interesting ideas (e.g., a refinement algorithm based on size-constrained label propagation that was developed as part of Vitali Henne's master thesis [Hen15a]) and the best quality in the majority of experiments, the algorithm was unable to improve on the state of the art consistently in terms of the time/quality trade-off.

However, we learned from that paper that recursive bipartitioning can be advantageous and thus decided to first focus on a highly optimized n-level recursive bipartitioning algorithm for cut-net optimization, which was presented in a conference publication [Sch+16a] and a technical report [Sch+15a] jointly published with Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. Following the success of this recursive bipartitioning algorithm, we then turned to connectivity optimization and developed a direct k-way partitioning algorithm. It employs locality-sensitive hashing (LSH) techniques in a preprocessing step to cluster (and contract) vertices with similar neighborhoods, uses the recursive bipartitioning algorithm as initial partitioning engine, and is the first multi-level algorithm to efficiently

implement the powerful FM-based local search heuristic for the complex k-way case. The corresponding paper was jointly published with Yaroslav Akhremtsev, Tobias Heuer, and Peter Sanders [Akh+17a]. In subsequent works, this algorithm was then enhanced with an improved coarsening scheme that incorporates global information about the community structure of the hypergraph into the coarsening process [HS17a], and with a refinement framework based on max-flow min-cut computations. The former was jointly published with Tobias Heuer [HS17a]. The latter was jointly published with Peter Sanders and Tobias Heuer in a technical report [HSS18b], a conference paper [HSS18a], and a journal paper [HSS19a].

n-Level Partitioning. Reviewing the historical development of hypergraph partitioning algorithms revealed that several authors had already established the connection between the number of hierarchy levels and the trade-off between solution quality and running time. Saab [Saa95] states that "compaction should proceed slowly in order to achieve high quality solutions". This observation was also made by Alpert et al. [AHK97] who note that "slower coarsening reduces the differences between successively coarser netlists H_i and H_{i+1} which implies that iterative refinement of H_i will take fewer passes to converge" and that "more levels allow more opportunities to refine the current solution at the various levels" [AHK98]. Furthermore, Alpert [Alp96, p. 198] explicitly states that improved solution quality comes "at a significant cost in CPU time" – a fact that was also observed by Karypis [Kar03, p. 139] during the development of hMETIS. The hypergraph partitioning framework presented in this chapter essentially shows how to evade this trade-off completely by going to the extreme case of (nearly) n levels. By engineering the algorithms used in both the coarsening and the refinement phase, and devising lazy-evaluation techniques and sophisticated caching mechanisms, we reduce the running time by more than two orders of magnitude compared to a naïve adaptation of the n-level approach used in KaSPar [OS10a; OS10b] for ordinary graph partitioning.

Chapter Overview. This chapter integrates all aforementioned publications in a consistent manner and extends the presentation in several ways. We start by describing our "semi-dynamic" hypergraph data structure in Section 4.1. It is semi-dynamic in that we are only concerned with efficient vertex and hyperedge *deletions* and the reversal of these operations, and do not consider insertions of additional vertices or nets. The data structure described in this dissertation is a more space efficient version of the data structure presented in the conference paper [Sch+16a], and therefore differs in the way contractions and uncontractions are performed. In Section 4.2, we discuss our approach to computing k-way partitions via recursive bipartitioning and the peculiarities that need to be addressed for cut-net and connectivity optimization. Section 4.3 then presents the employed preprocessing techniques, namely the LSH-based sparsification algorithm and the community-aware coarsening scheme. Afterwards, we address each of the three phases of the multi-level paradigm. Section 4.4 describes two different n-level coarsening schemes and discusses our implementation of parallel net detection. Afterwards, we briefly describe our portfolio-based approach to initial partitioning in Section 4.5. Section 4.6 then presents our localized 2-way and k-way FM-based local

search algorithms for optimizing both the cut-net metric and the connectivity metric. For k-way partitioning, the section extends previous work [Akh+17a] in two ways. First, we additionally discuss gain computation and delta-gain updates for cut-net optimization. Second, we prove the correctness of the delta-gain update procedures for both objective functions. Section 4.7 presents our flow-based refinement framework that uses max-flow computations on pairs of blocks to improve the solution quality of k-way partitions. In Section 4.8, we then define the two main framework configurations used in this work – the recursive bipartitioning algorithm rKaHyPar and the direct k-way partitioner kKaHyPar – before we extensively evaluate different aspects of the algorithms experimentally in Section 4.9. Section 4.10 then concludes this chapter.

References and Attributions. All aforementioned publications (Refs. [Sch+16a; Akh+17a; HS17a; HSS18a; HSS19a]) were almost entirely written by the author of this dissertation. A notable exception is the description of the sparsification algorithm [Akh+17a], which was jointly written with Yaroslav Akhremtsev. Peter Sanders was involved in the editing of all publications and provided many helpful remarks and comments to improve the presentation of the results. Henning Meyerhenke and Christian Schulz were involved in the editing process of Ref. [Sch+16a]. Since the following sections contain text passages from all publications in verbatim, we briefly outline the corresponding references and attribute ideas that were not our own. Section 4.1 and Section 4.2 are based on one conference paper [Sch+16a]. The sparsification algorithm described in Section 4.3.1 was designed by Yaroslav Akhremtsev, Peter Sanders, and the author of this dissertation, and is published in a conference paper [Akh+17a]. The implementation was done by Yaroslav Akhremtsev. Both text and pseudocode were rewritten significantly by the author of this dissertation to improve clarity. Section 4.3.2 is based on the corresponding conference paper [HS17a]. The implementation was largely done by Tobias Heuer who at the time worked as a student assistant for the author. Section 4.4 is based on two conference papers [Sch+16a; Akh+17a]. However, we provide additional insights into the rating function used to evaluate potential contractions and its effects on the coarsening algorithms. Section 4.5 is based on the same two conference papers as Section 4.4. The initial partitioning algorithms were implemented by Tobias Heuer as part of his bachelor thesis [Heu15a]. The thesis was supervised by us and contains detailed descriptions of the algorithms and an extensive experimental evaluation. Section 4.6 is based on two conference publications [Sch+16a; Akh+17a], while Section 4.7 contains large parts of a journal paper [HSS19a] and the corresponding conference publication [HSS18a]. The idea of generalizing the flow-based refinement framework of the graph partitioner KaFFPa [SS11] to hypergraph partitioning initially came from Peter Sanders. The implementation was done by Tobias Heuer as part of his master thesis [Heu18a], which was supervised by us. The experimental evaluations presented in Section 4.9.2 and Section 4.9.7 contain text passages from Ref. [HS17a] and Ref. [HSS19a], respectively.

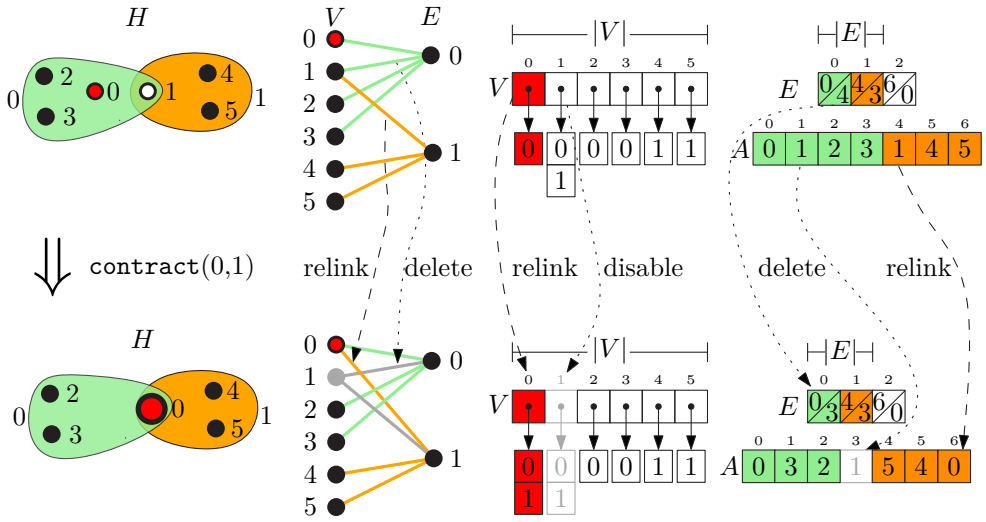


Figure 4.1: Example of a contraction operation. The hypergraph H is depicted on the left, the corresponding bipartite graph representation is shown in the middle, and the adjacency data structure is shown on the right. The contraction leads to an edge deletion operation for net 0 and a relink operation for net 1. Element $E[2]$ is a sentinel element used during uncontractions.

4.1 The Hypergraph Data Structure

Conceptual Overview. Conceptually, we represent the hypergraph H as an undirected *bipartite* graph $G = (V \dot{\cup} E, F)$. The vertices and nets of H form the vertex set. For each net e incident to a vertex v, we add an edge (e, v) to the graph. The edge set F is thus defined as $F := \{(e, v) \mid e \in E \wedge v \in e\}$. When contracting a vertex pair (u, v), we mark the corresponding node v as deleted. The edges (v, e) incident to v are treated as follows: If G already contains an edge (u, e), then net e contained both u and v before the contraction. In this case, we simply delete the edge (v, e) from G. Otherwise, net e only contained v. We therefore have to relink the edge (v, e) to u.

Data Structure. We use a combination of an adjacency list and a modified adjacency array to represent G. An example is shown in Figure 4.1. The adjacency list is used to store the incident nets of each vertex (i.e., the edges leaving nodes $v \in V$ in the bipartite graph representation), while the adjacency array stores the pins of each net (i.e., edges leaving nodes $v \in E$ in G). This representation is motivated by the observations that – after contracting a vertex pair (u, v) – (i) the degree $d(u)$ of representative u is *non-decreasing* (ignoring the removal of single-vertex nets), and (ii) the sizes $|e|$ of incident nets $e \in \mathrm{I}(u)$ are *non-increasing*. To index into the adjacency array A, we use an offset array E that stores the starting positions of the entries in

Algorithm 4.1 : Contract representative u with contraction partner v.

Input : Vertex pair (u, v) to be contracted

```
1  M := {u, v}                                          // Memento to remember contraction
2  c(u) := c(u) + c(v)                                  // Update weight of representative u
3  foreach e ∈ I(v) do                                  // Iterate over all incident nets of v
4      τ := l := E[e].f + E[e].s − 1                    // Last pin slot of net e
5      for i := E[e].f; i ≤ l; ++i do                   // Iterate over all pins of net e
6          if A[i] = v then                             // Find position of v ...
7              swap(A[i], A[l]); −−i                    // ... and move v to the last pin slot
8          else if A[i] = u then τ := i                 // Search for u and remember its position
9      if τ = l then                                    // Net e does not contain u ⇝ relink operation
10         A[l] := u                                    // Re-use slot of v in net e for u
11         V[u].append(e)                               // Add e to the adjacency list of u
12     else                                             // Net e contains both u and v ⇝ delete operation
13         −−E[e].s                                     // Cut off v
14 V[v].disable()                                       // Remove v from the hypergraph
```

Output : Contraction memento $\mathcal{M}$

A ($E[\cdot].f$) and the size of each net $|e|$ ($E[\cdot].s$). Thus, pins of a net e are accessible as $A[E[e].f], ..., A[E[e].f + E[e].s - 1]$, while nets incident to a vertex v are accessed using array V that stores a pointer for each vertex to a vector containing the corresponding incident nets.

Contraction. Contracting a vertex pair $(u, v) \in H$ works as follows: For each net $e \in \mathrm{I}(v)$ we have to determine if the corresponding edge $(v, e) \in G$ can simply be deleted or if a relink operation is necessary. This can be done with one iteration over the pins of e. During this iteration, we swap v with the last pin of e located at position $A[E[e].f + E[e].s - 1]$, and additionally search for vertex u. If we find u, then there is no need to perform a relink operation and we can remove v from e by simply decrementing $E[e].s$. If u was not found, we have to relink e to u. A relink operation adds the undirected edge (u, e) to G. To achieve this in our data structure, we have to add e to the set of incident nets of u, and make u a pin of net e. The latter can be accomplished by reusing the pin slot of v: After the iteration over the pins of e, v is the last entry in the subarray of e. Setting $A[E[e].f + E[e].s - 1] := u$ therefore adds u to the pins of e and simultaneously removes v. The former is done by appending e to the adjacency list of vertex u. Algorithm 4.1 gives the corresponding pseudocode. In order to be able to reverse contractions, we remember each contracted vertex pair in a memento $\mathcal{M}$.

Uncontraction. A pseudocode description of the uncontraction operation can be found in Algorithm 4.2. After re-enabling vertex v and resetting the weight of the representative vertex u, we first mark all incident nets $\mathrm{I}(v)$ as relevant for the current

Algorithm 4.2 : Revert the contraction of vertices u and v.

```
Input : Contraction Memento M = {u, v}
1  V[M.v].enable()                                   // Add vertex v to the hypergraph
2  c(M.u) := c(M.u) − c(M.v)                         // Reset weight of representative vertex u
3  b = [b_0, ..., b_{m−1}] := [false, ..., false]   // Bitset
4  foreach e ∈ V[M.v] do b[e] := true                // Mark all incident nets of v
5  s := |V[M.u]|                                     // The number of nets incident to vertex u
6  for i := 0; i < s; ++i do                         // Iterate over all nets e ∈ I(u)
7      e := V[M.u][i]                                // The ith net e incident to vertex u
8      if b[e] then                                  // Was net e incident to v before contraction?
9          // In this case we have to revert either a delete or a relink operation.
10         x := E[e].f + E[e].s                      // One past the last pin of net e
11         if x ≠ E[e + 1].f ∧ A[x] = M.v then       // Revert delete operation
12             // E[e+1].f always exists because of a sentinel element at position E[m]
13             ++E[e].s                              // Revert the cut-off of v
14         else // Revert relink operation: e ∈ I(v) ∧ e ∉ I(u) before contraction
15             swap(V[M.u][i], V[M.u][s − 1])        // Swap e to the end of u's adjacency list
16             V[M.u].pop()                          // Remove the last entry (i.e., net e) from u's adjacency list
17             −−i; −−s                              // Update loop variables
18             for j := E[e].f; j < E[e].f + E[e].s; ++j do   // Iterate over pins p ∈ e
19                 if A[j] = M.u then A[j] := M.v; break      // Reset the re-used pin slot
```

uncontraction using a bit vector b. We then iterate over all nets $e \in \mathrm{I}(u)$ of the representative u. If net e is also incident to the re-enabled vertex v (i.e., $b[e] = \text{true}$), it is necessary to revert either a delete or a relink operation. It is possible to distinguish between both cases by *peeking* one element past the slot of the last pin of e (see line 11 in Algorithm 4.2). If the pin located at this position is v and we are still in the pin range of net e (i.e., the current size of e is smaller than its original size), we have to revert a delete operation. Otherwise a relink operation needs to be reverted. Deletions can be reversed by simply increasing $E[e].s$ for the corresponding nets. To reverse a relink operation, we remove e from the adjacency list of vertex u, and reset the pin slot of e containing u back to v.

Difference to the Conference Version. The data structure presented in the conference paper [Sch+16a] is a generalization of the graph data structure used in KaSPar [OS10a]. It used a single adjacency array, which is divided into two offset arrays and an incidence array. The offset arrays are used to access the incident nets of each vertex and the pins of each net within the incidence array. Figure 4.2 shows an example of the old data structure for the hypergraph and the contraction operation depicted in Figure 4.1. While delete operations were handled the same way as described above, relink operations were handled differently. Since this data

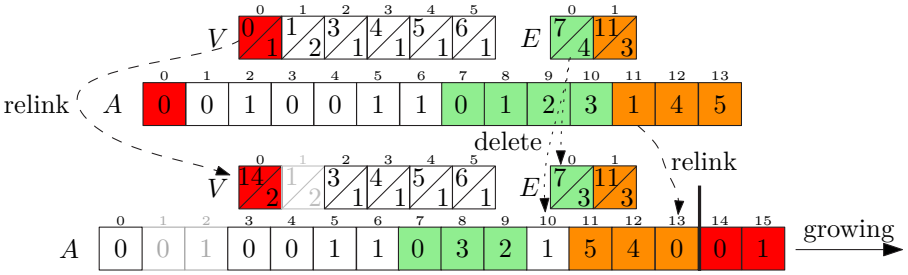


Figure 4.2: The hypergraph data structure used in the conference paper [Sch+16a] is a generalization of the graph data structure used in KaSPar [OS10a] (adapted from [Sch+16a]).

structure used the incidence array A to store both incident nets of vertices and pins of nets, the subarray of representative u had to be copied to the end of A every time $I(v) \setminus I(u) \neq \emptyset$. While the induced space overhead was deemed acceptable in the case of graph partitioning [OS10a], skewed degree distributions and large hyperedges can lead to excessive memory consumption for hypergraphs, which is why we propose the hybrid data structure described in this section.

4.2 Computing k-way Partitions via Recursive Bipartitioning

Motivation. As we have seen in Section 3.6.3, the question whether or not to prefer direct k-way partitioning over recursive bipartitioning is still unresolved. Our first partitioning algorithm [Sch+16a] used recursive bipartitioning to optimize the cut-net metric, before we turned to direct k-way partitioning optimizing both the cut-net [HSS18a; HSS19a] and the connectivity metric [Akh+17a; HS17a; HSS18a; HSS19a]. In the following, we therefore describe our approach for recursive bipartitioning. The general framework is outlined in Algorithm 4.3.

Recursive Bipartitioning. If k is a power of two, the final k-way partition is obtained by first computing a bipartition of the initial hypergraph and then recursing on each block. Hence, it takes $\log_2(k)$ such phases until the hypergraph is partitioned into k blocks. If k is not a power of two, the approach has to be adapted to produce appropriately-sized partitions. Our algorithm uses the following technique to compute a k-way partition via recursive bipartitioning for arbitrary values of k: We compute a 2-way partition of the hypergraph such that one block has a maximum weight of $(1+\varepsilon')\lceil\lfloor k/2\rfloor/k \cdot c(V)\rceil$ and the other block has a maximum weight of $(1+\varepsilon')\lceil\lceil k/2\rceil/k \cdot c(V)\rceil$, where ε' is a suitable adjusted imbalance parameter that ensures that the final k-way partition is ε-balanced. The former block is then partitioned recursively into $k' := \lfloor k/2\rfloor$ blocks, while the latter is partitioned into $k' := \lceil k/2\rceil$ blocks.

Adaptive Imbalance. If the initial imbalance parameter ε was to be used in each bipartitioning step, the weight of the largest block $V_{\max}$ could be larger than the maximum allowed block weight $L_{\max}$, which is why it is necessary to restrict the allowed imbalance at each bipartition. In the graph partitioning framework KaHIP, which uses recursive bipartitioning during initial partitioning, this is achieved by using a restricted imbalance of $\varepsilon' = 0.01$ for each 2-way partition in order to be "not too far away from the default value of 3% imbalance" [Sch13b, p. 49]. Since a fixed parameter may unnecessarily restrict the search space, we choose ε' adaptively for each bipartition, depending on the initial imbalance parameter ε and the target number of blocks k. Our approach is summarized in the following lemma:

Lemma 4.1 (Adaptive Imbalance for Recursive Bipartitioning [Sch+16a])
Let H_0 and H_1 be the hypergraphs induced by a bipartition $\Pi = \{V_0, V_1\}$ of an unweighted hypergraph $H = (V, E, c, \omega)$ for which we want to compute an ε-balanced k-way partition. Using an adaptive imbalance parameter

$$\varepsilon' := \left((1+\varepsilon) \frac{k' \cdot c(V)}{k \cdot c(V_i)} \right)^{\frac{1}{\lceil \log_2(k') \rceil}} - 1$$

to compute a k'-way partition (with $k' \geq 2$) of hypergraph H_0/H_1 via recursive bipartitioning ensures that the final k-way partition of H is ε-balanced.

When computing the very first bipartition for a k-way partition, we set $H_0 := H$, $k' := k$ and therefore $\varepsilon' := (1+\varepsilon)^{(1/\lceil \log_2(k) \rceil)} - 1$.

Proof. To show that using ε' at each bipartition ensures an ε-balanced k-way partition, we use a maximum block weight $L'_{\max} := (1+\varepsilon)\frac{c(V)}{k} \leq L_{\max}$. If the weight of each of the k' blocks of the k'-way partition is below $L'_{\max}$, then the final k-way partition of H is ε-balanced. To ensure this, we have to determine the maximum possible weight one of these blocks can have. Because H_i is split at each bipartitioning step such that one block can be further divided into $\lfloor k'/2 \rfloor$ blocks while the other is further split into $\lceil k'/2 \rceil$ blocks, the vertices of at least two blocks in the final k'-way partition have to be part of $\lceil \log_2(k') \rceil$ bipartitions. Let $V_{\max}$ be such a block and assume without loss of generality that at each bipartitioning step, block $V_{\max}$ has the maximum possible weight. Using the initial imbalance parameter ε at each bipartitioning step would therefore result in a final block weight of

$$c(V_{\max}) := (1+\varepsilon)^{\lceil \log_2(k') \rceil} \frac{c(V_i)}{k'}. \tag{4.1}$$

To ensure that the original k-way partition of H is ε-balanced, we have to choose ε in Eq. 4.1 such that $c(V_{\max}) \leq L'_{\max}$. Thus, when recursively partitioning a hypergraph H_i with weight $c(V_i)$ into k' blocks, we choose a new imbalance parameter ε' as follows:

Algorithm 4.3 : n-Level Recursive Bipartitioning (adapted from [Sch+16a])

Input : Hypergraph $H = (V, E)$ and imbalance parameter ε
Input : Partitioning objective $\mathfrak{O}$
Input : Lowest block number k_l and highest block number k_h

Function partition$(H, \varepsilon, \mathfrak{O}, k_l, k_h)$
  $k = k_h - k_l + 1$ *// Partition H into k blocks with block numbers $k_l, ..., k_h$*
  $\Pi_k := \emptyset$ *// The final k-way partition*
  if $k_l = k_h$ **then** $\Pi_k := V$; **return** Π_k *// Base case*
  $\varepsilon' :=$ *calculate according to Lemma 4.1* *// Adaptive imbalance calculation*
  while H is not small enough **do** *// n-Level Coarsening, described in Section 4.4*
    $(u, v) := \arg\max_{u \in V}$ *score*(u) *// Choose vertex pair with highest rating*
    $H :=$ contract(H, u, v) *// $H := H \setminus \{v\}$*
  $\Pi_2 = (V_0, V_1) :=$ bipartition(H, ε') *// Initial Partitioning, described in Section 4.5*
  while H is not completely uncoarsened **do** *// Uncoarsening/Refinement*
    $(H, \Pi_2, u, v) :=$ uncontract(H, Π_2) *// Uncontract vertex pair (u, v)*
    $(H, \Pi_2) :=$ refine$(H, \Pi_2, u, v, \varepsilon')$ *// Refinement (Section 4.6 and Section 4.7)*
  if $\mathfrak{O} = f_c(\Pi)$ **then** *// Cut-Net Optimization $\rightsquigarrow$ remove cut-nets*
    // Recurse on section hypergraphs
    $\Pi_k := \Pi_k \cup$ partition$(H \times V_0, \varepsilon, \mathfrak{O}, k_l, k_l + \lfloor k/2 \rfloor - 1)$
    $\Pi_k := \Pi_k \cup$ partition$(H \times V_1, \varepsilon, \mathfrak{O}, k_l + \lfloor k/2 \rfloor, k_h)$
  else if $\mathfrak{O} = f_\lambda(\Pi)$ **then** *// Connectivity Optimization $\rightsquigarrow$ split cut-nets*
    // Recurse on subhypergraphs
    $\Pi_k := \Pi_k \cup$ partition$(H_{V_0}, \varepsilon, \mathfrak{O}, k_l, k_l + \lfloor k/2 \rfloor - 1)$
    $\Pi_k := \Pi_k \cup$ partition$(H_{V_1}, \varepsilon, \mathfrak{O}, k_l + \lfloor k/2 \rfloor, k_h)$
  return Π_k
Output : ε-balanced k-way partition $\Pi_k = \{V_1, \ldots, V_k\}$

$$
\begin{aligned}
(1+\varepsilon')^{\lceil \log_2(k') \rceil} \frac{c(V_i)}{k'} \leq L'_{\max} &:= (1+\varepsilon)\frac{c(V)}{k} \\
\Rightarrow \varepsilon' &\leq \left((1+\varepsilon)\frac{k'\ c(V)}{k\ c(V_i)} \right)^{\frac{1}{\lceil \log_2(k') \rceil}} - 1.
\end{aligned} \tag{4.2}
$$

□

Note that for weighted hypergraphs, the desired imbalance ε (and thus the restricted imbalance ε') may not always be achievable (e.g., because of some very heavy vertices). This leads to imbalanced blocks already at early levels of the bipartitioning process. In order to be robust in these cases, we ensure that the adaptive imbalance parameter ε' is between zero and a maximum imbalance value $\varepsilon'_{\max}$, which is set to 0.99 in our implementation.

Cut-Net Splitting and Cut-Net Removal. Depending on the objective function $\mathfrak{O}$ to be optimized, cut-nets need to be treated differently when recursing on the two hypergraphs induced by a bipartition $\Pi = \{V_0, V_1\}$. For cut-net optimization, we recurse on the *section hypergraphs* $H \times V_0$ and $H \times V_1$. These section hypergraphs do not contain the cut-nets, because these nets will always be cut nets in the final k-way partition, and already contribute $\omega(e)$ to the total cut size [ÇA11b]. This simultaneously reduces the number of nets as well as their average size in each section hypergraph, without affecting the partitioning objective. For connectivity optimization, however, all following bipartitions can further increase the connectivity λ of the cut nets. Therefore, it is necessary to recurse on the *subhypergraphs* H_{V_0} and H_{V_1}, in which each cut-net e is *split* into two nets $e_0 = e \cap V_0$ and $e_1 = e \cap V_1$. Single-pin nets can be discarded in this process, as they cannot be cut in further bipartitioning steps.

4.3 The Preprocessing Phase

Before starting the n-level partitioning process, we sparsify hypergraphs with large nets, and infer information about the community structure to guide the coarsening process. Section 4.3.1 describes our pin sparsification algorithm. Community detection is discussed in Section 4.3.2.

4.3.1 Pin Sparsification via Locality-Sensitive Hashing

Motivation. Algorithms employed in each phase of the multi-level framework often perform computations on the vertices and their set of neighbors (e.g., to find the "best" contraction partner $u \in \Gamma(v)$ for vertex v during coarsening). For a given vertex v, this requires iterating over the set of *all* pins $p \in e$ of *all* incident nets $e \in \mathrm{I}(v)$. Especially for hypergraphs with many large nets, these calculations can therefore have a significant impact on the overall running time of the respective algorithm. To alleviate this impact, we employ a *pin* sparsifier as a preprocessing technique that clusters (and contracts) vertices with similar neighborhoods and thus reduces the average hyperedge size.

Central Idea. We consider two vertices u and v to be similar, if they *share* many nets, i.e., if their sets of incident nets $\mathrm{I}(u)$ and $\mathrm{I}(v)$ have a relatively large intersection. Similarity is measured using the Jaccard coefficient $J(A, B) = |A \cap B|/|A \cup B|$, for finite sets A and B. In the following, we will use $J(u, v)$ to denote the Jaccard coefficient of the incident nets of two vertices u and v, i.e., $J(u, v) = |\mathrm{I}(u) \cap \mathrm{I}(v)|/|\mathrm{I}(u) \cup \mathrm{I}(v)|$. The corresponding *distance* metric then is $D(u, v) = 1 - J(u, v)$ [Cha02]. Since calculating these distances/similarities for every pair of vertices would lead to a quadratic-time algorithm, we instead use the *locality-sensitive* hashing (LSH) technique [IM98; GIM99] to identify sets of similar vertices that are "close" to each other with respect to $D(\cdot, \cdot)$. Similar vertices are then contracted to reduce the number of pins in the hypergraph.

Locality-Sensitive Hashing (LSH). The key idea of the LSH approach is to hash elements in such a way that the probability for "close" elements to have equal hash values is high, while the probability for "distant" elements to have equal hash values is low. For the Jaccard distance $D(\cdot,\cdot)$, the following family of hash functions (called *min-hash*) is known to be locality-sensitive: $\mathcal{H} = \{h_\sigma(X) = \min\{\sigma(x)|x \in X|\sigma \in \Sigma\}\}$, where $X \subseteq U$ is a finite set of elements from a finite universe U, and σ is a random permutation from the set Σ of all random permutations of U [Bro97; Bro+97]. It can be proven that $Pr[h_\sigma(A) = h_\sigma(B)] = J(A,B)$ [AI08; LK10], i.e., the *larger* the distance, the *smaller* the collision probability [GIM99]. For an excellent introduction to min-hashing and the locality-sensitive hashing technique, we refer the reader to the work of Leskovec, Rajaraman, and Ullman [LRU14].

Min-Hash Fingerprints. Instead of maintaining all permutations of the set E of hyperedges, we use a set Σ' of hash functions of the form $h(x) = ax+b \bmod \mathcal{P}$ [Bro+00] to simulate the effect of a random permutation σ, i.e., our min-hash family of hash functions becomes $\mathcal{H}' = \{h(v) = \min\{h(e)|e \in \mathrm{I}(v)\}|h \in \Sigma'\}$. A min-hash *fingerprint* for vertex v is then defined as $g_i(v) = (h_1(v), h_2(v), \cdots, h_i(v))$, where each hash function h_j is chosen uniformly at random from $\mathcal{H}'$. In our sparsification algorithm, vertices with the *same* fingerprint will be put in the same cluster, and we consider two fingerprints to be equal if and only if all i hash values are equal.

Since the fingerprints are used to approximate the distances between vertices, the size of the fingerprint (i.e., the number i of min-hashes) affects the probability that vertices are put into the same cluster [LRU14]. By increasing the number of hashes, we decrease the probability that "distant" vertices have the same fingerprint. In fact, this probability decreases exponentially with the size of the fingerprint, since all hash functions are chosen independently and therefore it follows that

$$\begin{aligned} Pr[g_i(x) = g_i(y)] &= Pr[\bigwedge_{j=1,\ldots,i} h_j(x) = h_j(y)] = \prod_{j=1,\ldots,i} Pr[h_j(x) = h_j(y)] \\ &= Pr[h(x) = h(y)]^i = Pr[g_1(x) = g_1(y)]^i, \end{aligned} \tag{4.3}$$

given that all i hash functions $h(\cdot)$ are chosen uniformly at random from $\mathcal{H}'$. However, at the same time, increasing the size of the fingerprint also decreases the probability of "close" vertices ending up in the same cluster. To avoid this problem, we calculate up to l fingerprints for each vertex. Quality and running time of our algorithm therefore depend on the choice of parameters i and l. Since the distance between vertices varies in different parts of a hypergraph, we choose both parameters adaptively.

Related Approaches. The LSH technique is also used by Deveci et al. [DKÇ13] as a sparsification heuristic to identify "similar" nets. While their similar net removal (SNR) algorithm is also based on computing fingerprints consisting of several min-hashes (i.e., by hashing the pins of the hyperedges), it differs from our approach in that only a single, fixed-size fingerprint is used in the similarity estimation, while our adaptive clustering algorithm uses up to l fingerprints, *and* dynamically adapts their sizes. Furthermore, while the contraction of similar vertices in our case is straight-forward,

Algorithm 4.4 : Adaptive Hash Table Construction using LSH-based Fingerprints

Input : The input hypergraph $H = (V, E)$
Input : The set A of active, i.e., unclustered vertices

```
1  Function IdentifySimilarVertices(H, A)
2      i := 1                          // Size of fingerprint g_i(·), i.e., the number of min-hashes used
3      T := {}                         // Final hash table to represent the resulting clusters
4      A' := A                         // Currently active vertices
5      T_prev := {}                    // Temporary hash table for adaptive construction of T
6      while A' ≠ ∅ ∧ i ≤ h_max do      // Successively increase the fingerprint sizes
7          T_cur := {}                 // Temporary hash table for adaptive construction of T
8          foreach v ∈ A' do           // Find similar vertices using fingerprint g_i(·)
9              T_cur.insert(g_i(v), v) // Insert v into T_cur using g_i(v) as key
10         foreach v ∈ A' do           // Evaluate the sets of similar vertices induced by g_i(·)
11             B_cur[v] := {u ∈ T_cur : g_i(u) = g_i(v)}        // Similar vertices w.r.t. g_i(·)
12             B_prev[v] := {u ∈ T_prev : g_{i-1}(u) = g_{i-1}(v)} // Similar vertices w.r.t. g_{i-1}(·)
13             if i ≥ h_min then       // Vertices u ∈ B_cur[v] are similar enough
14                 if |B_prev[v]| = |B_cur[v]| ∨ |B_cur[v]| ≤ c_max then // Explained in the text
15                     foreach u ∈ B_cur[v] do
16                         A' := A' \ {u}      // Omit vertices from the remaining passes
17                         T.insert(g_i(v), u) // Add set of vertices similar to v to result
18         i := i + 1                  // Increase size of fingerprint for next iteration
19         swap (T_cur, T_prev) // Remember current clustering decisions for next iteration
       Output : Hash table T consisting of buckets B of "similar" vertices.
```

it is not obvious how to choose the pin set of a representative net (which combines several similar, but not identical nets into a single one), such that the negative effect on solution quality is minimized.

Identifying Similar Vertices. In Algorithm 4.4 we show how to adaptively compute fingerprints of increasing size to incrementally identify small (i.e., $\leq c_{\max}$) sets of similar vertices. The actual clustering algorithm that uses this information will be described in the following paragraph. The LSH-based algorithm works in passes. In pass i, fingerprints $g_i(\cdot)$ of size i are used to hash similar vertices into the same bucket of a hash table T_{cur}. In order to decrease the probability that "distant" vertices end up in the same bucket, we only consider vertices to be truly similar if the fingerprint size is at least $h_{\min}$. In case the resulting bucket contains at most $c_{\max}$ similar vertices (i.e., it is not larger than our cluster size threshold), or the bucket size did not change in the current pass, we consider the set of similar vertices a candidate set for the clustering algorithm and ignore it in future passes. The algorithm proceeds until either no active vertices remain or the size of the fingerprints becomes larger than $h_{\max}$.

Algorithm 4.5 : LSH-based Clustering for Pin Sparsification

Input : The input hypergraph $H = (V, E)$

```
1  Function LSHClustering(H)
2      C = [v_1, ..., v_n] := [v_1, ..., v_n]      // Initially, each vertex is in its own cluster
3      M = {}                  // Maps cluster number to set of vertices contained in cluster
4      A := V                                                  // Initially all vertices are active
5      j := 1                                                  // Iteration counter for ...
6      while A ≠ ∅ ∧ j ≤ l do          // ... performing similarity detection up to l times
7          T_j := IdentifySimilarVertices(H, A)                          // See Algorithm 4.4
8          foreach v ∈ A do                                    // Visit all active vertices ...
9              B_v := {u ∈ T_j : g^j(u) = g^j(v)}          // Bucket of vertices similar to v
10             foreach u ∈ B_v do                          // ... and cluster those similar to v
11                 if |M[C[v]]| < c_max then                // Cluster of v is not too large
12                     C[u] := C[v]                                  // Add u to the cluster of v
13                     M.insert(C[v], u)                // Insert u into M using C[v] as key
14                     B_v := B_v \ {u}                 // Remove u, since it has been clustered
15             if |M[C[v]]| ≥ c_min then                            // Cluster is large enough
16                 foreach u ∈ M[C[v]] do  A := A \ {u}          // ⇝ deactivate vertices
17         j := j + 1
       Output : C contains the cluster number of each vertex.
```

Adaptive LSH Clustering. The clustering algorithm described in Algorithm 4.5 works in passes and maintains a set of *active* (i.e., unclustered) vertices. In the beginning, all vertices are marked as active. Each pass j then starts by identifying sets of similar vertices using the min-hash fingerprint $g^j(\cdot)$ as described in Algorithm 4.4. These sets are stored in a hash table T_j. Let B_v denote the set/bucket of T_j that contains all vertices u which are similar to vertex v, i.e., $\forall u \in B_v : g^j(v) = g^j(u)$. In order to guarantee $\mathcal{O}(1)$ expected time for insertions and deletions, each bucket B_v is itself represented by a hash table. Each active vertex v is then clustered with similar vertices from B_v as long as the size of the resulting cluster is less than $c_{\max}$. If the size is at least $c_{\min}$, all clustered vertices become inactive and do not participate in the next pass. By bounding cluster sizes from below by $c_{\min}$ and from above by $c_{\max}$, we enforce the formation of reasonably balanced clusters in order to allow the partitioning algorithm to compute feasible solutions of high quality. The clustering algorithms stops as soon as the number of resulting clusters is less than $n/2$ or the maximum number l passes is exceeded. Each cluster is then contracted to a single vertex.

The running time of the algorithm is dominated by the time it takes to identify sets of similar vertices. Each hash table T_j can be computed in time $\mathcal{O}(h_{\max} \cdot \sum_{v \in V} d(v)) = \mathcal{O}(h_{\max} \cdot p)$. Since we perform similarity detection at most l times, the total running time of the adaptive LSH-based clustering algorithm therefore is $\mathcal{O}(l \cdot h_{\max} \cdot p)$.

4.3.2 Detecting Community Structure To Improve Coarsening

Motivation. As we have learned in Section 3.6.1, the goal of the coarsening phase is to create successively smaller but *structurally similar* approximations of the input hypergraph in which both the exposed hyperedge weight as well as the sizes of the hyperedges are successively reduced. This is commonly done by using rating functions to identify and contract highly connected vertices that share a large number of heavy nets with small size, and by allowing the formation of vertex *clusters* instead of enforcing matchings, since their maximality constraint can destroy some naturally existing clustering structures in the hypergraph [Kar03] (see Figure 4.3 (a)–(c) for an example). Furthermore, the algorithms ensure that the distribution of vertex weights does not become too imbalanced at the coarsest level, since this limits the number of feasible initial partitions satisfying the balance constraint. This is done by either enforcing an upper bound on the vertex weight or by integrating a penalty factor into the rating function that discourages the formation of heavy vertices.

However, since coarsening decisions are only based on *local* information, several situations can arise in which the naturally existing structure within the hypergraph is obscured: If multiple neighbors have the same rating score, coarsening algorithms employ different tie-breaking strategies such as randomly choosing one of them or giving preference to vertices that have not yet been clustered [Kar03; Akh+17a] (see Figure 4.3 (d),(e)). Furthermore, a restriction on the maximum allowed vertex weight can lead to situations in which the highest-rated contractions are forbidden by the weight constraint and the coarsening algorithm has to contract vertices with lower rating score (Figure 4.3 (f)). Situations like these arise because all coarsening algorithms are guided by local, greedy decisions based on rating functions that solely consider the weights and sizes of nets connecting candidate vertices and therefore lack a global view of the clustering problem. If information about the community structure were to be known before the coarsening process, these cases could have been prevented explicitly. We therefore propose an approach to combine a *global* view on the problem with *local* coarsening decisions.

Community Detection via Modularity Maximization. Community detection tries to extract an underlying structure from a graph by dividing its nodes into disjoint subgraphs (communities) such that connections are dense *within* subgraphs but sparse between them [Sch07; For10]. Different quality functions are used to judge the goodness of a division into communities. Among those, the most popular quality function is the *modularity* measure of Newman and Girvan [NG04]. It compares the observed fraction of edges within a community with the expected fraction of edges if edges were placed using a random edge distribution that preserves the degree distribution of the graph [FH16]. More formally, given a graph G and disjoint communities $C = \{C_1, \ldots, C_x\}$, modularity is defined as:

$$Q := \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(C_i, C_j), \tag{4.4}$$

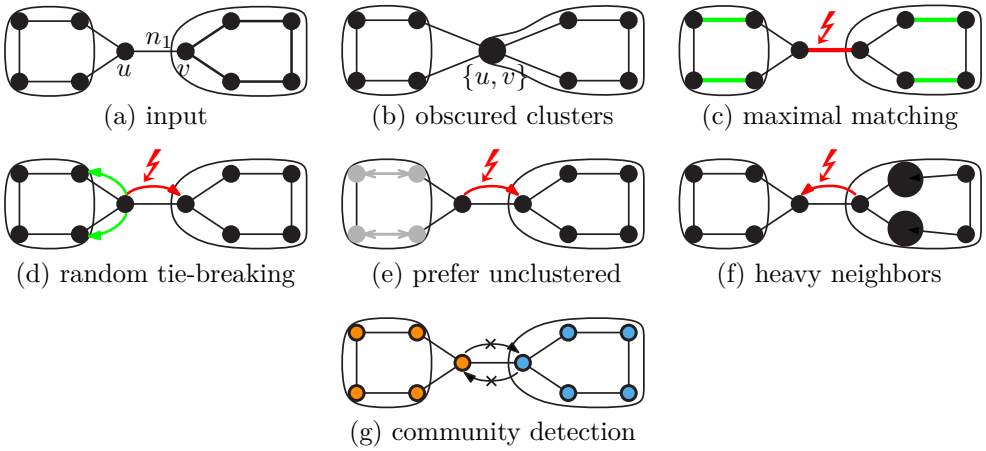


Figure 4.3: (a) Hypergraph with 10 vertices and 13 nets. Nets containing only two vertices are shown as graph edges. By cutting net n_1, the hypergraph can be partitioned into two balanced blocks. (b) Contracting vertex pair (u, v) obscures the naturally existing clustering structure and the cut of size 1. (c)–(f) Properties of coarsening algorithms that lead to the contraction of (u, v): (c) Coarsening based on maximal matchings. (d) Random tie-breaking among all neighbors with same rating score. (e) Preferring unclustered vertices to break ties. (f) Contraction partners with highest rating score are already too heavy. (g) Our approach: Restrict contractions to vertex pairs *within* the same community. This prevents the contraction of (u, v) in all aforementioned cases (source: [HS17a]).

where A_{ij} is the entry of the adjacency matrix A representing edge (i, j), $m = \frac{1}{2}\sum_{ij} A_{ij}$ is the number of edges in the graph, k_i is the degree of node i, C_i is the community of vertex i, and δ is the Kronecker delta. Note that this can be generalized to weighted graphs: A_{ij} represents the weight of edge (i, j), $k_i = \sum_j A_{ij}$ is the weighted degree of node i and $m = \frac{1}{2}\sum_{ij} A_{ij}$ is the sum of all edge weights [New04]. Modularity optimization is known to be NP-hard [Bra+08], but several efficient heuristics exist. Furthermore, there exist several definitions of modularity adapted specifically to bipartite graphs [Bar07; GSA07; SW09; Mur10]. However, we do not consider these definitions in this work, since they do not translate into fast algorithms and therefore only scale to small bipartite graphs [Mur10]. We therefore use the definition shown in Eq. 4.4. We note that there also exist techniques to detect communities in k-partite, k-uniform hypergraphs. In these approaches, hypergraphs are projected to k bipartite graphs and bipartite modularity measures are used to detect community structures [NO09].

The Louvain Algorithm. A fast and widely used algorithm to detect community structure in graphs via modularity maximization is the Louvain method introduced by Blondel et al. [Blo+08]: Initially, each node is assigned to a community of its own.

Then, the algorithm proceeds in two phases that are repeated iteratively. In the first phase, nodes are repeatedly assigned to the neighboring community that maximizes the increase in modularity. This local, greedy optimization stops when no further increase is possible. In the second phase, the graph is coarsened according to the community structure discovered in the first phase by contracting each community into a single node. Then, the process starts again on the coarsened graph and is repeated until the maximum modularity is achieved. The communities of the coarsest graph determine the community structure of the input graph. The algorithm has low computational complexity and is thus suitable for large graphs [LF09; For10].

Community-aware Coarsening Framework. Our framework consists of two phases. First, a (graph-based) community detection algorithm is used to partition the vertices of the hypergraph into a set $C = \{C_1, \ldots, C_x\}$ of internally densely and externally sparsely connected communities. The actual number of communities $|C|$ is determined by the community detection algorithm. Then, a hypergraph coarsening algorithm is applied on each community C_i independently. This can be accomplished by modifying the algorithm to only contract vertices within the *same* community by restricting potential contraction partners of a given a vertex $u \in C_i$ to $\Gamma(u) \cap C_i$. By preventing inter-community contractions, the coarsening algorithm maintains the structural similarity discovered by the community detection algorithm, while still allowing local, intra-community decisions to be based on HGP-specific rating functions. Note that this framework is *independent* of the algorithms used for community detection and coarsening. In the following, we describe our instantiation, which performs community detection via modularity maximization using the Louvain method, and uses one of the coarsening algorithms described in Section 4.4.

Graph-based Hypergraph Representation. In order to employ the Louvain method as community detection algorithm, a suitable graph-based representation of the hypergraph has to be chosen. As described in Section 2.1, the two common models are the clique and the bipartite/star representation. Several issues make the clique representation unsuitable for our purpose: Inserting $\binom{|e|}{2}$ graph edges into the clique graph for every net e destroys the natural sparsity of the hypergraph [AK95c] and therefore may be prohibitively costly in terms of both space and running time. This is important since hypergraphs arising in partitioning problems are typically sparse [Dev+06]. Thus, for sparse instances the number of edges in the clique representation can be as high as $\mathcal{O}(n^2)$. Furthermore and more importantly, this exaggerates the importance of nets with more than two pins [SK72], since large nets automatically imply a high density in the clique representation. We therefore use the *bipartite* representation, which allows us to encode any hypergraph in $\mathcal{O}(p)$ space. In the following, we refer to the graph nodes representing the vertices of the hypergraph as V-nodes and to the nodes representing the nets as E-nodes (see Figure 4.4 for an example).

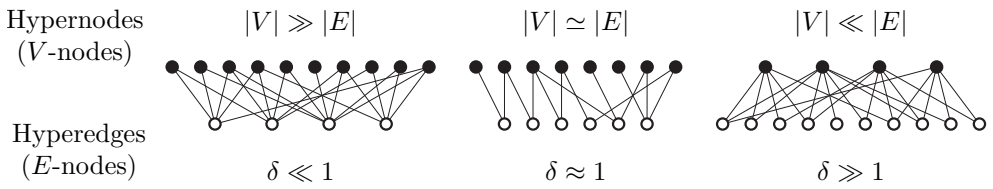


Figure 4.4: Bipartite graph-based representations of hypergraphs of varying edge density δ. For hypergraphs with $\delta \ll 1$, the bipartite graph consists of many V-nodes with low average degree and fewer E-nodes with high average degree (left). If $\delta \approx 1$, the number of V- and E-nodes and their average degrees are roughly equal (middle). Hypergraphs with high ratio $\delta \gg 1$ lead to bipartite representations with fewer V-nodes with high average degree and many E-nodes with low average degree (right) (adapted from [HS17a]).

> Schweikert and Kernighan [SK72], who were the first to use a hypergraph model in the context of circuit partitioning, describe the shortcomings of the clique-net model as follows: "The exaggeration grows rapidly as the number of elements increases; for example, the cost of dividing an 11-element net ranges between 10 and 30 edge cuts, when physically only one wire needs to be cut. Such a grossly disproportionate weighting of an 11-element net, compared with two-element nets, almost insures that all its attached elements will be in one package of the partition - unfortunately 'dragging their tentacles behind them'."

Modeling Peculiarities. By performing community detection on the bipartite graph representation we receive a community partition of *both* the vertices *and* the nets of the hypergraph, since both are represented as (V, E)-nodes in the graph. However, we are only interested in the community structure of the vertices. Therefore we have to take structural properties of the hypergraphs into account. More specifically, we have to consider the *edge density* [DW19]:

$$\delta := \frac{\overline{d(v)}}{\overline{|e|}} = \frac{p/n}{p/m} = \frac{m}{n}, \tag{4.5}$$

where $\overline{d(v)}$ is the average vertex degree and $\overline{|e|}$ is the average net size. If $\delta \approx 1$, the number of V-nodes is roughly equal to the number of E-nodes and $\overline{d(v)} \simeq \overline{|e|}$. If $\delta \gg 1$ then there are more E-nodes than V-nodes and $\overline{d(v)} \gg \overline{|e|}$, whereas if $\delta \ll 1$ the opposite is the case (see Figure 4.4). In case the hypergraph exhibits a low edge density and therefore a large average net size, special care has to be taken in order to ensure that the community structure is not exclusively shaped by the high-degree E-nodes. Similarly, the large number of E-nodes can lead to a community structure that is dominated by the nets of the hypergraph in case $\delta \gg 1$. Hypergraphs with ratio $\delta \approx 1$ do not pose a problem, since the number of V-nodes and E-nodes as well as their degrees are balanced. We account for these structural differences by encoding

additional information about the hypergraph structure into the weights of the bipartite graph edges.

Weighting Graph Edges. We propose three different weights for the edges (v, e) between V-nodes $v \in V$ and E-nodes $e \in E$ as shown in Eq. 4.6. The first scheme uses uniform edge weights as a baseline. Giving each edge an equal weight is expected to provide good clustering results for hypergraphs with $\delta \approx 1$, since for these instances the number of V- and E-nodes as well as their degrees are roughly comparable. The second and third schemes account for the skew in low- and high-density hypergraphs. The weighting function ω_e assigns each edge a weight which is inversely proportional to the size of the net, i.e., smaller nets get a higher influence on the community structure than larger nets. If many small nets are contained within a community, the coarsening algorithm can successively reduce their size and eventually remove them from the hypergraph. Furthermore, this ensures that high-degree E-nodes (i.e., large nets) do not dominate the community structure by attracting too many V-nodes. This edge weight only affects the clustering decisions of V-nodes, since from the perspective of E-nodes each outgoing edge still has uniform weight $1/|e|$. In order to also influence the clustering decision of E-nodes, the third weighting function ω_{de} additionally integrates the hypernode degree into the edge weight. Strengthening the connection between E-nodes and high-degree V-nodes facilitates the formation of communities around high-degree vertices in the hypergraph. Note that it is possible to efficiently choose an appropriate weighting scheme at runtime by calculating the edge density of the hypergraph according to Eq. 4.5 and modifying the edge weights appropriately.

$$\omega(v, e) := 1 \qquad \omega_{\mathrm{e}}(v, e) := \frac{1}{|e|} \qquad \omega_{\mathrm{de}}(v, e) := \frac{d(v)}{|e|} \tag{4.6}$$

4.4 The Coarsening Phase

Motivation. Multi-level coarsening algorithms either compute matchings [AHK98; Kar+99; VB05; Dev+06] or clusterings [HB97; ÇA99; KK00; TK04a] on each level of the coarsening hierarchy using different rating functions to determine the vertices to be matched or clustered together. The contracted vertices then form the vertex set of the coarser hypergraph on the next level. In contrast, n-level partitioning algorithms like the graph partitioner KaSPar [OS10a] create a hierarchy of (nearly) n levels by removing only a *single* vertex between two levels, which completely obviates the need for employing matching or clustering algorithms in the coarsening phase. In this section, we generalize KaSPar's coarsening approach to hypergraphs and overcome its main bottlenecks by introducing a lazy evaluation technique. Furthermore, we propose a simpler n-level coarsening algorithm that retains the solution quality of the KaSPar approach.

Rating Function. Our algorithms adopt the *heavy-edge* rating function also used by hMETIS [Kar+99], Par$k$way [TK08], and PaToH [ÇA11b], which prefers vertex

pairs (u, v) that have a large number of heavy nets with small size in common:

$$\text{con}(u, v) := \gamma(u, v) \cdot \sum_{e \in \{\text{I}(v) \cap \text{I}(u)\}} \frac{\omega(e)}{|e| - 1}, \tag{4.7}$$

where $\gamma(u, v)$ is an optional penalty factor inversely proportional to the product of the vertex weights $c(v)$ and $c(u)$ to keep the distribution of vertex weights in the coarse hypergraphs reasonably uniform. This is also done in the matching-based coarsening algorithm of ML_c for similar reasons [AHK97; AHK98]. The following lemma shows that when the heavy-edge rating function with $\gamma(u, v) = 1/(c(u) \cdot c(v))$ is used to coarsen *graphs*, the rating scores for each node form a non-increasing sequence.

Lemma 4.2 (Non-Increasing Rating Scores for Graph Contractions) *Let $G = (V, E, c, \omega)$ be a graph and let*

$$\mathit{con}(u, v) := \frac{1}{c(u) \cdot c(v)} \cdot \sum_{e \in \{\text{I}(v) \cap \text{I}(u)\}} \frac{\omega(e)}{|e| - 1} = \frac{1}{c(u) \cdot c(v)} \cdot \sum_{e \in \{\text{I}(v) \cap \text{I}(u)\}} \omega(e)$$

be the rating function used to evaluate the importance of contractions. Then contracting a pair of adjacent nodes (u, v) (i.e., an edge $e \in E$) never increases the rating scores of neighboring nodes in $\Gamma(u) \cup \Gamma(v)$.

Proof. It suffices to consider the graph shown in Figure 4.5, since a contraction only affects neighboring nodes. Furthermore, the only effects of a contraction are (i) the increase of the node weight of the representative u by the weight of the contraction partner v, and (ii) the introduction of parallel edges if neighbors are adjacent to both u and v. Assume without loss of generality that the weight of edge $e = (v_1, v_2)$ is larger than $\omega(a)$, $\omega(b)$, and $\omega(c) = \max_{v_i \in \Gamma(v_3) \setminus \{v_1, v_2\}} \omega(v_3, v_i)$ so that vertex pair (v_1, v_2) will be contracted first, and let all nodes have unit weight. The rating score of vertex v_3 is determined by the maximum weight of its incident edges. If $\omega(c) > \omega(a)$ and $\omega(c) > \omega(b)$, then v_3 initially chooses v_4 as contraction partner. Moreover, this choice (and thus the rating score) remains unaffected by the contraction of nodes v_1 and v_2, since $(\omega(a) + \omega(b))/2 < \omega(c)$. Similarly, if either $\omega(a)$ or $\omega(b)$ (or both) are larger than $\omega(c)$, v_3 initially chooses the node connected via the edge $\arg\max(\omega(a), \omega(b))$ as contraction partner. After contraction, the contraction partner is then updated to $\{v_1, v_2\}$ if $(\omega(a) + \omega(b))/2 > \omega(c)$. Otherwise, v_4 becomes the new contraction partner. In this case, however, $\max((\omega(a) + \omega(b))/2, \omega(c)) \leq \max(\omega(a), \omega(b))$ by definition. Similar arguments hold for the cases in which either $\omega(a)$ or $\omega(b)$ (or both) are equal to $\omega(c)$. □

Figure 4.6 however shows that Lemma 4.2 does not generalize to hypergraphs.

Detecting Single-Vertex Nets and Parallel Nets. The contraction of a vertex pair (u, v) can lead to parallel nets (i.e., nets that contain exactly the same vertices) and single-vertex nets in $\text{I}(u)$. In order to reduce the running time of the coarsening

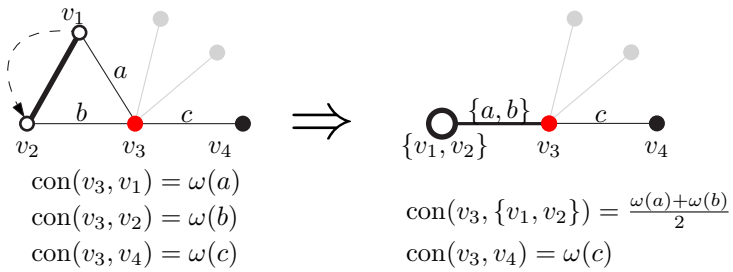


Figure 4.5: Visualization for the proof of Lemma 4.2 assuming unit node weights. The weight of edge (v_1, v_2) is larger than $\omega(a)$, $\omega(b)$, and $\omega(c) = \max_{v_i \in \Gamma(v_3) \setminus \{v_1,v_2\}} \omega(v_3, v_i)$.

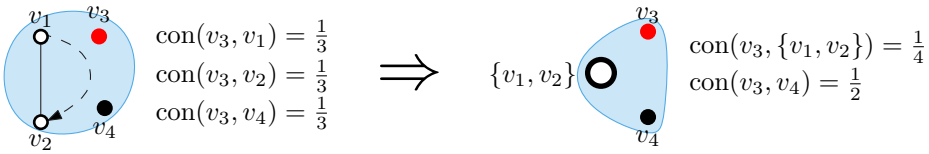


Figure 4.6: Counterexample of Lemma 4.2 for hypergraphs.

algorithms, we continuously detect and remove these nets from the hypergraph. Single-vertex nets are easily identified, because $|e| = 1$. In case of parallel nets, we remove all but one from H. The weight of the remaining net e is set to the sum of the weights of the nets that were parallel to e. Efficient parallel net detection is discussed in detail in Section 4.4.3.

4.4.1 n-Level Hypergraph Coarsening

Algorithm Outline. In the following, we describe the generalization of KaSPar's coarsening algorithm [OS10a] to hypergraphs. At the beginning of the coarsening phase, we compute the ratings $\text{con}(u, v)$ for each vertex u and all *eligible* neighbors $v \in \Gamma(u)$. To avoid imbalanced inputs for the initial partitioning phase, a contraction is only deemed eligible if $c(u) + c(v) \leq \kappa$, where $\kappa := s \cdot \lceil \frac{c(V)}{t \cdot k} \rceil$ is the maximum allowed vertex weight. Parameter s is used to favor the contraction of highly connected vertices, while parameter t is used to control the size of the coarsest hypergraph. Since the initial partitioning algorithm has to compute a k-way partition, the number of vertices in the coarsest hypergraph should be a function of k [KK99].

For each vertex, we then insert the pair (u, v) with the highest rating (ties are broken randomly) into an addressable priority queue (PQ) using the rating score as key. This allows us to efficiently choose the best-rated vertex pair that should be contracted next, and provides a "global" view on high-rated contraction opportunities. The algorithm then iteratively removes and contracts the pair (u, v) with the highest

score until the number of vertices drops below $t \cdot k$ or no eligible vertex is left.

After each contraction, we have to update the PQ such that it reflects the structural changes induced by the contraction. This is done in three steps: First, we compute a new eligible contraction partner v' for the representative vertex u and insert the new pair (u, v') into the PQ. Second, we delete the entry of v from the PQ, since the contraction removed v from the hypergraph. Third, we recompute the ratings for all neighbors $\Gamma(u)$ of representative u and update the PQ accordingly, since the contraction may have influenced/invalidated some of the ratings scores.

Lazy Update Strategy. While this algorithm is adequate for ordinary graph partitioning [OS10a], its running time can easily become prohibitive when coarsening hypergraphs, because even a *single* large hyperedge can significantly increase the size of the neighborhood $\Gamma(v)$ of a vertex v. Continuously re-rating these neighbors therefore becomes the most expensive part of the algorithm, since each rating computation involves an iteration over all pins of all nets incident to the respective vertex.

To improve the running time in these cases, we developed a variation of the re-rating procedure described above. Our *lazy* strategy does not re-rate *any* vertices immediately after contracting a vertex pair (u, v). Instead, all neighbors $\Gamma(u)$ of the representative u are marked as *invalid* after the contraction. If, during coarsening, the PQ returns an invalid vertex, we recompute its rating and update the priority queue accordingly. The advanced update strategy thus *delays* updates of the rating scores until an invalid vertex reaches the top of the priority queue. Given Lemma 4.2, it therefore holds that, for *graphs*, the lazy re-rating approach produces the same "quality" as a coarsening algorithm that always re-rates all neighbors of the contracted vertex pair, because rating scores never increase and thus the partial order of all contractions in the priority queue is never violated by delaying an update. Although this is not the case for hypergraphs in general, we will show in Section 4.9.4 that the lazy strategy is still effective for hypergraph coarsening.

> Our extensive literature research, performed as preparation for the brief history of hypergraph partitioning presented in Chapter 3, revealed that a similar algorithm (named *best choice*) was independently proposed by Alpert, Kahng, Nam, Reda, and Villarrubia [Alp+05; Alp+06] as a clustering technique for hierarchical VLSI *placement*, i.e., the task to assign circuit components to exact locations within the chip area. While this approach was implemented in many placement tools [LBR07], it went unnoticed in both the graph and the hypergraph partitioning community.

4.4.2 Simple and Fast Greedy Coarsening

Motivation. The two general bottlenecks of the coarsening algorithm described in the previous section are (i) the use of a priority queue to determine which vertex pair to contract next, and (ii) the (lazy) recalculation of rating scores for neighboring vertices after each contraction. In the following, we present a much simpler coarsening algorithm that eliminates both bottlenecks without affecting the overall solution

quality [Akh+17a]. It is motivated by the observation that although the lazy-update strategy weakened the order of contractions to some extent, it had no significant effect on partitioning quality. Our simple and fast greedy algorithm therefore gives up the global ordering of contractions entirely to further speed up the coarsening phase.

Algorithm Outline. In general, our algorithm is very similar to the First Choice (FC) algorithm employed in hMETIS-K [KK00]. It works in multiple passes. At the beginning of each pass, we create a random permutation of the current vertex set. For each vertex u, we then determine its contraction partner v and immediately contract (u, v) *on the fly*. Thus, v will be removed from the hypergraph and will not be visited in this or any future passes over the vertex set. Instead of penalizing the formation of heavy vertices (i.e., here, we set the penalty parameter $\gamma(u, v)$ to 1), we employ a tie-breaking mechanism to avoid imbalanced inputs for the initial partitioning phase: For each vertex u, we favor a contraction partner v that has not yet taken part in any contractions during the pass. To speed up the coarsening process in the presence of large nets, we do not evaluate the rating function for nets larger than ι vertices. As in the previous algorithm, we also impose a maximum vertex weight and prevent vertices heavier than κ to be contracted further. A pass ends as soon as every vertex in the random permutation was considered either as representative or as contraction partner. Then, a new pass is started by creating a new random permutation of the remaining vertices. Similar to the algorithm presented in the previous section, the coarsening process is stopped as soon as the number of vertices drops below $t \cdot k$ or no eligible vertex is left.

A Different View on Coarsening. The key difference to FC and related coarsening algorithms lies in the way contractions are handled. While traditional algorithms *first* compute a matching/clustering on each level and *then* use it to create a coarse hypergraph for the next level, we rate *and* contract one vertex at the same time, i.e., after finding the contraction partner $v \in \Gamma(u)$ for a vertex u, we immediately contract (u, v). Thus, while in FC clustering decisions are made for all vertices of the current level *at once*, and more importantly *independently* of one another, our n-level greedy coarsening algorithm adaptively adjusts *every* contraction decision to the current structure of the hypergraph induced by all previous contractions.

> Note that this difference also holds true for the PQ-based coarsening algorithm described in the previous section. However, while the greedy algorithm only has a local view, the PQ-based algorithm is able to always perform the next "globally" best contraction.

4.4.3 Engineering Parallel Net Detection

Algorithm Outline. Parallel nets are detected using an algorithm similar to the one proposed by Hendrickson and Rothberg [HR98], which identifies nodes with identical structure in a graph. For each net $e \in \mathrm{I}(u)$ we create a fingerprint f_e, such that parallel nets have equal fingerprints. These fingerprints are then sorted and a final scan identifies parallel nets: Since nets with unequal fingerprints cannot be parallel by

definition, we have to perform pairwise comparisons of the pin sets only for those nets with same fingerprint and size.[1]

Engineering Aspects. Removing parallel nets does not affect partitioning quality. Instead, it is intended to speed up all phases of the multi-level framework. Therefore, it is necessary to make this operation as efficient as possible, because – having fast coarsening algorithms at hand – it can easily become the new bottleneck during coarsening. We therefore improve the algorithm described above in two ways:

In order to keep the number of pairwise pin set comparisons as low as possible, the fingerprint function should produce few false positives, i.e., equal fingerprints for nets that are not actually parallel. Deveci et al. [DKÇ13] evaluate different fingerprint functions and conclude that $f_e^2 := \sum_{v \in e} v^2$ performed best. While early versions of our algorithm used $f_e^{\oplus} := \left(\bigoplus_{v \in e} v\right) \oplus x$ as fingerprint, for some seed x, our evaluation showed that $f_e^{\oplus}$ creates a significant number of false-positives [Akh+17a].[2] We therefore adopt $f_e^2 := \sum_{v \in e} v^2$ in our algorithm.[3]

While recomputing fingerprints is feasible in traditional multi-level approaches, it becomes expensive in the n-level setting. Instead of calculating the fingerprints for each net $e \in \mathrm{I}(u)$ *from scratch* after each contraction operation, we therefore exploit the fact that the fingerprint function f_e^2 is both *associative* and *commutative*. When constructing the hypergraph data structure we compute the initial fingerprints for each net *once*. After each contraction, we then update the fingerprints of nets $e \in \mathrm{I}(u)$ as follows, distinguishing three cases: In case net e was only incident to representative u before the contraction, the fingerprint remains valid since these nets remain unchanged. If net e contained both u and v before the contraction, we have to remove v from the fingerprint: $f_e^2 := f_e^2 - v^2$. If net e was only incident to v but not to u before the contraction, we have to add u to the fingerprint $f_e^2 := f_e^2 + u^2$ in addition to removing v, since it is incident to u after the contraction.

4.5 Portfolio-Based Initial Partitioning

If configured to compute k-way partitions using recursive bipartitioning as described in Section 4.2, KaHyPar uses a portfolio of several algorithms to compute an initial solution. Each algorithm is run several times using different random seeds. The actual number of repetitions as well as the portfolio composition depends on the framework configuration. The partition with the best solution quality and lowest imbalance is used as initial partition and projected back to the original hypergraph. In case all partitions are imbalanced, we choose the partition with smallest imbalance. The portfolio approach increases diversification and produces better results than

[1] We also tried the hashing-based approach proposed by Deveci et al. [DKÇ13]. However, on average, it did not perform better than the sorting approach.

[2] $\oplus$ is the bitwise XOR

[3] In general, permutation checking could be done by using random hash functions or an approach that constructs polynomials from the pin sets of two potentially parallel nets (see, e.g., Ref. [HS18b]). In our case, however, the simple approach described above performed sufficiently well.

single initial partitioning algorithms alone [Heu15a]. In the following, we give a brief overview of the algorithms employed in our initial partitioning portfolio and refer to the corresponding bachelor thesis [Heu15a] for a more detailed description and evaluation.

Random & BFS-based Partitioning. *Random partitioning* randomly assigns vertices to one of the two blocks, provided that the assignment does not violate the balance constraint. In case of a violation, the vertex in question is assigned to the opposite block. If both assignments would lead to overloaded blocks, the vertex is randomly assigned to one of the blocks. *Breadth-First-Search* (BFS) partitioning starts with a randomly chosen vertex and performs a BFS traversal of the hypergraph until the weight of all discovered vertices would exceed the balance constraint. The vertices visited during the traversal constitute the first block V_0, all remaining vertices constitute the second block V_1.

Greedy Hypergraph Growing (GHG). Furthermore, we use different variations of the GHG algorithm proposed by Çatalyürek and Aykanat [ÇA99]. Unlike the original algorithm, which grows a cluster around a randomly selected seed vertex, our versions first compute two pseudo-peripheral vertices as follows: Starting from a random vertex, we perform a BFS. The last vertex visited serves as the starting vertex for the next BFS. This vertex and the last vertex visited by the second BFS are supposed to be "far" away from each other. Therefore, one is used as the seed vertex for block V_0, the other for block V_1. For each block, we maintain a PQ that stores the neighboring vertices of the growing cluster according to a score function. The algorithm then iteratively selects the vertex with the highest gain from one of the PQs, moves the vertex to the corresponding block, and then updates the scores of neighboring vertices. As with most move-based algorithms, each vertex is only allowed to be moved once. We use the FM gain (see Eq. 3.1), as well as the *max-pin* and *max-net* gain definitions (which are also used in PaToH [ÇA11b]) as score functions. The FM gain prefers to move vertices that decrease the cut-size. However, a net e incident to an unassigned vertex v only contributes positively to the gain of vertex v if $\Phi(e, V_{0/1}) = |e| - 1$, i.e., if v is the only pin of e that is not yet assigned to block V_0 or V_1. Otherwise the gain contribution of net e is zero. The max-pin gain uses the number of pins $p \in e$ of incident nets $e \in \mathrm{I}(v)$ that are already assigned to the target block as a measure on how tightly connected v is to the target block (i.e., $g_{\text{max-pin}}(v) := |\{p \in e \mid e \in \mathrm{I}(v) \wedge p \in V_{0/1}\}|$) whereas the max-net gain counts the weights of all nets connected to the target block (i.e., the gain of assigning vertex v to block V_i is defined as $g_{\text{max-net}}(v) := \sum_{e \in E'} \omega(e)$, where $E' := \{e \in \mathrm{I}(v) \mid \Phi(e, V_i) > 0\}$). Using max-pin or max-net gains, incident nets are more likely to contribute positively to the gain of a vertex. Our GHG variants also differ in the way the clusters are grown. The *global* strategy always moves the vertex with the highest score of both PQs to the corresponding block, whereas the *sequential* approach first grows block V_0 and then block V_1. The *round-robin* technique grows both blocks simultaneously. In total, the initial partitioning portfolio therefore contains nine different initial partitioning algorithms based on GHG.

Size-Constrained Label Propagation. The last algorithm is based on the adaptation of *size-constrained label propagation (SCLaP)* [MSS14; MSS16] to HGP local search. The SCLaP-based refinement algorithm was initially proposed in the master thesis of Vitali Henne [Hen15a], which we supervised. Each vertex has a label representing its block. Initially all labels are empty, i.e., all vertices are unassigned. The algorithm starts by searching two pseudo peripheral vertices via BFS as described above. One vertex and τ of its neighbors then get label V_0, while the other vertex and τ of its neighbors get label V_1. The algorithm then works in rounds until it has converged, i.e., no empty labels remain. In each round, the vertices are visited in random order and each vertex u is assigned the label of the neighbor $v \in \Gamma(u)$ that results in the highest FM gain, provided that the resulting cluster does not become overloaded. Ties are broken randomly. Once the algorithm has converged, vertices with the same label then become a block of the bipartition. The tuning parameter τ is used to prevent labels from disappearing over the course of the algorithm, and, based on experimental results [Heu15a], is set to $\tau = 5$ in our implementation.

Direct k-way Partitioning. KaHyPar contains k-way generalizations for all initial partitioning algorithms presented in the previous paragraphs, which, however, are not used for n-level direct k-way partitioning. Instead, we employ our n-level recursive bipartitioning algorithm to compute an initial k-way partition, since it has been shown to perform considerably better than using direct k-way initial partitioning algorithms [Heu15a]. This observation is in line with early experimental results on k-way multi-level graph partitioning [Kar96, p. 42] and the fact that most direct k-way multi-level partitioning algorithms employ (multi-level) initial partitioning algorithms that use recursive bipartitioning [KK00; ACU08; Sch13b; Çat+15].

4.6 Localized 2-way and k-way FM Local Search

Overview. We now turn to our local improvement algorithms. Both 2-way and k-way local search follow the FM paradigm [FM82] and are further inspired by the algorithm used in KaSPar [OS10a; OS10b]. A key difference to the traditional FM algorithm is the way a local search pass is started: Instead of initializing the algorithm with all vertices or all border vertices, we perform a *highly localized* search starting only with the representative and the just uncontracted vertex. The search then gradually expands around this vertex pair by successively considering neighboring vertices. Our 2-way local search algorithm optimizing the cut-net metric $f_c(\Pi)$ is described in Section 4.6.1. It is also used to implicitly optimize the connectivity metric $f_\lambda(\Pi)$ when KaHyPar is configured to use recursive bipartitioning. In this case, we employ cut-net splitting instead of cut-net removal at each bipartitioning step as described in Section 4.2. In Section 4.6.2, we then describe our k-way local search algorithm. Unlike in the case of 2-way partitioning, objective-specific gain computations and delta-gain updates are necessary to permit the algorithm to optimize both objectives. Traditional multi-level FM implementations as well as KaSPar *always* compute the gain of *each* vertex from scratch at each level of the hierarchy. During an FM pass,

these values are then either kept up-to-date by delta-gain updates [ÇA99; PM07] or recomputed whenever necessary [Sch13b]. Since our algorithms start around only two vertices, many gain values would never be used during a local search pass. In Section 4.6.3, we therefore propose a *gain caching technique* that ensures that the gain of a vertex move is calculated at most *once* during *all* local searches along the n-level hierarchy. Since local search is done after each uncontraction, it is necessary to limit the number of vertex moves in each pass, because otherwise the n-level approach could lead to a quadratic number of local search steps in total. In Section 4.6.4, we therefore present two stopping rules that terminate the iterative improvement process before all vertices have been moved. Finally, we briefly discuss implementation details in Section 4.6.5.

Motivation. The benefits of localization are visualized in Figure 4.7, which shows a 4-way partition in which all border vertices have negative gains ($-$), and only few positive gain ($+$) vertices exist further away from the border. Traditional local search algorithms are initialized with *all* border vertices (left). Thus, it is unlikely that once the positive gain moves are encountered, they still yield an overall improvement in solution quality, because the search already progressed too far away from the solution quality of the starting partition. In this case, an FM-style algorithm would rollback all moves – including those with positive gain. If local search is started with only a small set of border vertices (in our case *exactly two*), it explicitly works in a small area of the cut-set and is thus more likely to retain the positive-gain moves.

Simple techniques (such as LIFO tie-breaking) and more advanced approaches such as CLIP/CDIP [DD96b; DD96c; DD02] implicitly enforce localization effects by trying to move clusters of (close) vertices together. However, since they are initialized with all border vertices, the distribution of move gains may still prevent the desired effect. The KaSPar approach [OS10a; OS10b] on the other hand (which has also motivated the localized local search algorithm of KaFFPa [SS11]) explicitly restricts the search space to small areas of the cut-set.

4.6.1 2-way Localized FM Refinement

Algorithm Outline. We use two PQs to maintain the possible moves for all vertices – one for each block. At the beginning of a local search pass, both queues are empty and disabled. A disabled PQ will not be considered when searching for the next move with the highest gain. All vertices are labeled inactive and unmarked. Only unmarked vertices are allowed to become active. To start the local search phase after each uncontraction, we activate the representative and the just uncontracted vertex if they are border vertices. Otherwise, no local search phase is started. *Activating* a vertex v currently assigned to block V_i means that we calculate the FM gain $g_j(v)$ for moving v to the other block $V_j \in \mathrm{B}(v) \setminus \{V_i\}$ and insert v into the corresponding queue P_j using $g_j(v)$ as key. Recall that for a vertex $v \in V_i$ the FM gain is defined as

$$g_j(v) := \sum_{e \in \mathrm{I}(v)} \{\omega(e) \mid \Phi(e, V_j) = |e| - 1\} - \sum_{e \in \mathrm{I}(v)} \{\omega(e) \mid \Phi(e, V_i) = |e|\}, \tag{4.8}$$

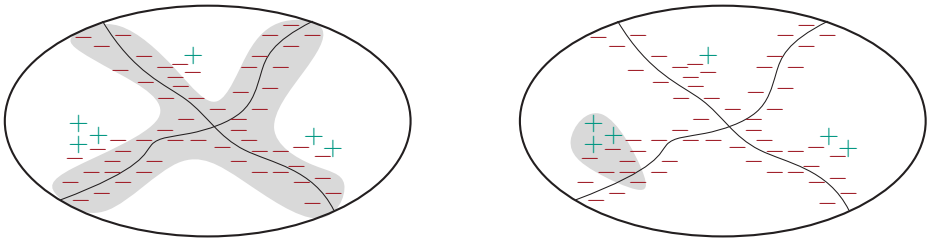


Figure 4.7: Benefits of localization. Minus marks represent negative gain moves, plus marks represent positive gain moves (based on [Sch13b, Figure 4.8]). If the local search algorithm is initialized with all border vertices (left), it is unlikely to find and retain the improvements possible by moving the positive-gain vertices. When starting the search with a small set of vertices (right), the algorithm is more likely to be able to exploit the positive-gain moves to improve the solution.

since a net $e \in \mathrm{I}(v)$ only contributes $+\omega(e)$ to the gain of a vertex $v \in V_i$, if v is the *last* pin in block V_i that is moved to block V_j. Similarly, net e has a gain contribution of $-\omega(e)$ if all of its pins are assigned to block V_i.

After insertion, PQs corresponding to *underloaded* blocks become enabled. Since a move to an overloaded block will never be feasible, any queue corresponding to an overloaded block is left disabled. Similarly to the original FM algorithm, we relax the definition of overloaded blocks to account for variations in vertex weights. Here, we consider a block to be overloaded if it deviates from the maximum allowed block weight by more than than the weight of the currently heaviest vertex (i.e., if $c(V_i) > L_{\max} + \max_{v \in V} c(v)$). The algorithm then repeatedly queries only the *non-empty, enabled* queues to find the move with the highest gain $g_j(v)$, breaking ties arbitrarily. Vertex v is then moved to block V_j and labeled inactive and marked. We then update all neighbors $\Gamma(v)$ of v as follows: All previously inactive neighbors are activated as described above. Neighbors that have become internal are labeled inactive and the corresponding moves are deleted from the PQs. Finally, we perform *delta-gain updates* for all moves of the remaining active border vertices in $\Gamma(v)$: If the move changed the gain contribution of a net $e \in \mathrm{I}(v)$, we account for that change by incrementing/decrementing the gains of the corresponding moves by $\omega(e)$ using the delta-gain-update algorithm of Papa and Markov [PM07]. Once all neighbors are updated, local search continues until either no non-empty, enabled PQ remains or the stopping rule mandates the termination of the current pass. After local search is stopped, we reverse all moves until we arrive at the lowest cut state reached during the search that fulfills the balance constraint. All vertices become unmarked and inactive and the algorithm is then repeated until no further improvement is achieved.

Locked Nets. To further decrease the running time, we exclude nets from gain updates that cannot be removed from the cut in the current local search pass. A net is

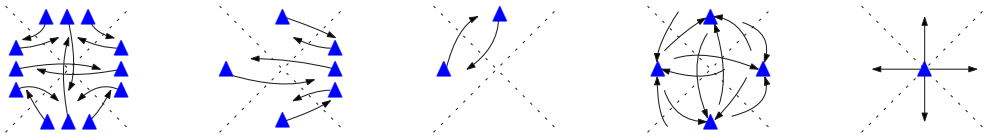


Figure 4.8: Different approaches to FM-based direct k-way refinement. Each triangle represents a priority queue associated with the corresponding block. Note that all variants except the KaSPar and KaFFPa schemes use "from" priority queues, i.e., vertex moves are stored in the PQs associated with the current block of the vertex. In KaSPar, a move from block V_i to block V_j is stored in the PQ associated with V_j, while KaFFPa uses a single PQ to store the highest-gain move for each vertex.

locked in the bipartition once it has at least one marked pin in each of the two blocks. In this case, it is not possible to remove such a net from the cut by moving any of the remaining movable pins to another block. Thus, it is not necessary to perform any further delta-gain updates for locked nets, since their contribution to the gain values of their pins does not change any more. This observation was first described by Krishnamurthy [Kri84]. We integrate locking of nets into our algorithm by labeling each net during a local search pass. Initially, all nets are labeled *free*. Once the first pin of a net is moved, the net becomes *loose*. It now has a pin in one block that cannot be moved again. Further moves to this block do not change the label of the net. As soon as another pin is moved to the other block, the net is labeled *locked* and is excluded from future delta-gain updates.

4.6.2 k-way Localized FM Refinement

Motivation. As we have seen in Chapter 3, there is a large design space for k-way refinement algorithms. On one extreme, there is the single-level k-LA$_\ell$-FM algorithm of Sanchis [San89; San93] which maintains $k(k-1)$ priority queues (one for each possible move direction). To the best of our knowledge, a multi-level version of k-LA$_\ell$-FM has only been implemented in the MSN partitioner [UA04] to optimize the total message latency of parallel matrix vector multiplications by partitioning hypergraphs with very few (i.e. ≤ 128) nets. In general, it is considered to be too slow to be practical and known to produce worse solutions than competing approaches in the single-level context, because it is more easily trapped in local minima [Wan+00; TK04a]. On the other extreme, these problems motivate current multi-level HGP systems [KK00; ACU08; TK08; Çat+12b] to entirely forgo FM-based direct k-way refinement and to instead rely on weaker greedy local search algorithms that cannot escape from local optima. In between these two extremes, there is the rotary KLFM algorithm of Chan et al. [CSZ97a; CSZ97b] and the K-PM approach of Cong and Lim [CL98]. Rotary KLFM uses $2(k-1)$ PQs and in each round only allows moves between a

target block V_i and all other blocks $\Pi \setminus \{V_i\}$, while K-PM only uses two PQs and iteratively improves the k-way partition by moving vertices between all $k(k-1)/2$ pairs of blocks. Furthermore, for graph partitioning there exists the KaSPar [OS10a; OS10b] method that uses k PQs (one for each block), and the k-way local search technique of KaFFPa [SS11] that uses a single priority queue which only stores the highest-gain move for every vertex. Figure 4.8 outlines the different PQ-based approaches. Note that in the figure, each triangle corresponds to a priority queue. Our algorithm is based on the refinement scheme used in KaSPar (i.e., we use k priority queues and each PQ stores the vertex moves to that particular block), since k-LA$_\ell$-FM is not deemed practical, both rotary KLFM and K-PM only have a restricted view on the k-way partition, and the KaFFPa approach makes it necessary to recompute gains after each move in order to identify those with highest gain.

Differences to the 2-way Algorithm. In general, the k-way refinement algorithm follows the same outline as the 2-way algorithm described in the previous section. We therefore focus on the differences to the 2-way algorithm before describing the gain computation and delta-gain update techniques for k-way connectivity and cut-net optimization. We only consider moving a vertex $v \in V_i$ to *adjacent* blocks $\mathrm{B}(v) \setminus \{V_i\}$ rather than calculating and maintaining gains for moves to *all* k blocks. This simultaneously reduces the memory requirements and restricts the search space of the algorithm to moves that are more likely to improve the solution. Thus, the k PQs require $\mathcal{O}(k|V_B|)$ space in total, where V_B is the set of border vertices. In the bipartitioning setting, our refinement algorithm is able to implicitly rebalance an infeasible solution, since a priority queue is disabled once the corresponding block becomes overloaded. Thus, in this case, the algorithm only moves vertices from the overloaded to the underloaded block. This, however, does not apply to the k-way setting. The fact that a PQ corresponding to an overloaded block is disabled does not automatically force the algorithm to move vertices out of the overloaded block, since other moves from/to other blocks may be preferred due to having a larger gain. Therefore, we do not relax the definition of overloaded blocks in our k-way algorithm. Instead, a PQ is disabled as soon as the weight of the corresponding block becomes larger than or equal to $L_{\max}$, and we only perform vertex moves if they are feasible with regard to the original balance constraint. Otherwise they are skipped and the corresponding vertices are locked into their current blocks, since moving them would lead to imbalanced and thus infeasible solutions. After moving a vertex v, we remove all other moves of v from the PQs, because we only allow each vertex to be moved at most once during each pass. Furthermore, in order to speed up local search in the presence of large nets, we do not activate vertices that are only incident to nets with $|e| \geq 1\,000$, as it is unlikely that such a vertex entails a positive-gain move.

Connectivity Metric: Gain Computation & Delta-Gain Updates. When activating a vertex $v \in V_i$, we calculate the *gain* $g_j(v)$ for moving v to all adjacent blocks $V_j \in \mathrm{B}(v) \setminus \{V_i\}$, and insert v into the corresponding queues P_j using $g_j(v)$ as

Algorithm 4.6 : Delta-Gains for Connectivity Metric (adapted from [Akh+17a])

Input : Vertex v that was moved from block V_{from} to V_{to}

```
1  Function DeltaGainUpdate(v, V_from, V_to)
2      foreach e ∈ I(v) do                                   // Iterate over all incident nets ...
3          if Φ(e, V_from) > 1 ∨ Φ(e, V_to) > 2 then continue   // Gain remains unaffected
4          foreach u ∈ e \ {v} do                            // ... and consider each pin
5              if u is marked then continue                  // Skip marked vertices
6              if Φ(e, V_from) = 0 then                      // Connectivity of net e decreased
7                  if ∄ n ∈ I(u) : Φ(n, V_from) > 0 then      // u is not adj. to V_from anymore
8                      P_from.remove(u)
9                      B(u) := B(u) \ {V_from}
10                 else P_from.update(u, −ω(e))              // u remains adjacent to V_from
11             if Φ(e, V_to) = 1 then                        // Connectivity of net e increased
12                 if V_to ∉ B(u) then                       // u was not adjacent to V_to
13                     P_to.insert(u, g_to(u))
14                 else P_to.update(u, ω(e))                 // u was already adjacent to V_to
15             V_u := current block of vertex u
16             if V_u = V_from ∧ Φ(e, V_from) = 1 then        // Moving u will decrease λ(e)
17                 foreach V_i ∈ B(u) \ {V_u} do P_i.update(u, ω(e))
18             else if V_u = V_to ∧ Φ(e, V_to) = 2 then      // Move can't decrease λ(e) anymore
19                 foreach V_i ∈ B(u) \ {V_u} do P_i.update(u, −ω(e))
20             if V_to ∉ B(u) then B(u) := B(u) ∪ {V_to}      // Update adjacent blocks
```

Output : The gains for all moves of all neighbors $\Gamma(v)$ are updated.

key. For connectivity optimization, the gain $g_j(v)$ is defined as

$$g_j(v) := \sum_{e \in \mathrm{I}(v)} \{\omega(e) \mid \Phi(e, V_i) = 1\} - \sum_{e \in \mathrm{I}(v)} \{\omega(e) \mid \Phi(e, V_j) = 0\}. \tag{4.9}$$

The first term sums the weights of all nets for which v is the only pin left in block V_i. For these nets, it is possible to reduce the connectivity by moving v to another block $V_j \in \mathrm{B}(v)$. The second term accounts for the fact that although block V_j may be adjacent to v, it may not be in the connectivity set $\Lambda(e)$ of net e. In this case the gain contribution of net e is either zero if $\Phi(e, V_i) = 1$, or $-\omega(e)$ if $\Phi(e, V_i) > 1$.

After moving a vertex v, we perform *delta-gain updates* for all moves of the remaining active border vertices in $\Gamma(v)$. If the move changed the gain contribution of a net $e \in \mathrm{I}(v)$, we account for that change by increasing/decreasing the gains of the corresponding moves by $\omega(e)$. Moving a vertex v can furthermore change the connectivity λ of a net $e \in \mathrm{I}(v)$, which in turn can affect the set of adjacent blocks $B(\cdot)$ for each neighbor in $\Gamma(v)$. The delta-gain-update algorithm takes these changes into account by inserting moves to new adjacent blocks into the PQs and removing

Figure 4.9: Visualization of the situations that yield a change in the gain contribution of a net $e \in \mathrm{I}(v)$ after moving vertex v for connectivity optimization. After moving the white vertex, the gains of the red vertices change. Cases (i) and (ii) of the proof of Theorem 4.3 are shown on the left. Cases (iii) and (iv) are shown on the right.

moves to blocks that are not adjacent anymore. A pseudocode description of the delta-gain-update process for connectivity optimization can be found in Algorithm 4.6.

Theorem 4.3 (Correctness of Connectivity Delta-Gain Updates)
Let v be the vertex that was moved from block V_{from} to block V_{to}. After performing delta-gain updates as described in Algorithm 4.6, all vertex moves have gain $g_j(\cdot)$.

Proof. To prove the correctness of Algorithm 4.6, we show *how* the move of v from block V_{from} to block V_{to} affects the move-gains of other vertices. By the gain definition shown in Eq. 4.9, only incident nets contribute to the move-gains of a vertex. Therefore, it follows that (i) only the gains of neighbors $u \in \Gamma(v)$ are potentially affected by the move of vertex v from block V_{from} to block V_{to}, and (ii) only nets $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$ can change the gain of a vertex $u \in \Gamma(v)$ – all other nets have no effect. In the following, we call such nets *critical*.

For a given vertex move, the gain contribution of a net e depends on the number of pins $\Phi(e, V_{\text{from}})$ of e in the current block V_{from} of the vertex, and the number of pins $\Phi(e, V_{\text{to}})$ in the target block V_{to} of the move. For each net $e \in \mathrm{I}(v)$ (and thus also for critical nets), the move of v *decreases* $\Phi(e, V_{\text{from}})$ by one, and *increases* $\Phi(e, V_{\text{to}})$ by one. Thus, the gain contribution of a critical net can only change if one of its pins is moved out of blocks V_{from} or V_{to}, or into blocks V_{from} and V_{to}.

Let $u \in \Gamma(v)$ be a neighbor of v, let e be a critical net, and let $g_j(u)$ be the old gain (before the move of v) of moving u from its current block V_i to block $V_j \in \mathrm{B}(u) \setminus \{V_i\}$. Given Eq. 4.9, we distinguish the following four cases (visualized in Figure 4.9):

(i) If v was moved out of u's block (i.e., $u \in V_{\text{from}}$), the gain contribution of net e only changes if $\Phi(e, V_{\text{from}})$ decreases from 2 to 1, since it is not possible that $\Phi(e, V_{\text{from}})$ decreases from 1 to 0 (because u is still in block V_{from}). Before the move, net e thus contributed 0 to the gains of moving u to adjacent blocks $V_j \in \mathrm{B}(u) \setminus \{V_{\text{from}}\}$, because u was *not* the only pin of e in block V_{from}. Since after the move $\Phi(e, V_{\text{from}}) = 1$, net e now contributes $+\omega(e)$ to the gains of moving u to adjacent blocks $\mathrm{B}(u) \setminus \{V_{\text{from}}\}$, because u now *is* the only pin of e in block V_{from}. Thus, $g_j(u) = g_j(u) + \omega(e)$ (Alg. 4.6, line 16).

(ii) If v was moved to the block of u (i.e., $u \in V_{\text{to}}$), the gain contribution of net e only changes if $\Phi(e, V_{\text{to}})$ increases from 1 to 2, since it is not possible that $\Phi(e, V_{\text{to}})$ increases from 0 to 1 (because u was already in block V_{to} before the move). Before the move, net e thus contributed $+\omega(e)$ to the gains of moving u to adjacent blocks $V_j \in \mathrm{B}(u) \setminus \{V_{\text{to}}\}$, because u was the *only pin* of e in block V_{to}. Since after the move $\Phi(e, V_{\text{to}}) = 2$, net e now contributes 0 to the gains of moving u to adjacent blocks $V_j \in \mathrm{B}(u) \setminus \{V_{\text{to}}\}$, because moving u now does not remove block V_{to} from the connectivity set $\Lambda(e)$ of net e. Thus, $g_j(u) = g_j(u) - \omega(e)$ (line 18).

Otherwise, u is neither in block V_{from} nor in block V_{to}. In this case, the gain contribution of net e only changes if its connectivity $\lambda(e)$ increases and/or decreases as the result of the move.

(iii) The former happens if the number of pins $\Phi(e, V_{\text{to}})$ increases from 0 to 1, since it is not possible that $\Phi(e, V_{\text{from}})$ increases from 0 to 1. Before the move, net e thus contributed $-\omega(e)$ to the gain of moving u to block V_{to}, because V_{to} was *not* in the connectivity set $\Lambda(e)$ of e. Since after the move $\Phi(e, V_{\text{to}}) = 1$, net e now contributes 0 to the gain of moving u to block V_{to}, because v now connects e to block V_{to}. Thus, $g_{\text{to}}(u) = g_{\text{to}}(u) + \omega(e)$ (line 11).

(iv) The latter happens if the number of pins $\Phi(e, V_{\text{from}})$ decreases from 1 to 0, since it is not possible that $\Phi(e, V_{\text{to}})$ decreases from 1 to 0. Before the move, net e thus contributed 0 to the gain of moving u to block V_{from}, because V_{from} *was* in the connectivity set of e. Since after the move $\Phi(e, V_{\text{from}}) = 0$, net e now contributes $-\omega(e)$ to the gain of moving u to block V_{from}, because v was the last pin of e in block V_{from}. Thus, $g_{\text{from}}(u) = g_{\text{from}}(u) - \omega(e)$ (line 6).

Note that gain updates are restricted to *unmarked* vertices, as each vertex is only allowed to be moved once in each pass and a vertex becomes marked after it is moved. The case distinctions in line 7 and line 12 follow from the fact that we only allow vertices to move to *adjacent* blocks. Thus, instead of decreasing the gain by $\omega(e)$, we remove the move to block V_{from} if vertex u is not adjacent to block V_{from} anymore after the move of v. Similarly, if $V_{\text{to}} \notin \mathrm{B}(u)$ before the move, we have to calculate the gain of this *new* move from scratch, since it was not allowed before the move of v. □

Cut-Net Metric: Gain Computation & Delta-Gain Updates. The FM gain definition for 2-way partitioning shown in Eq. 4.8 generalizes naturally to k-way partitioning optimizing the cut-net metric $\mathfrak{f}_c(\Pi)$. The only difference to bipartitioning is that vertices can now be moved to *more* than one adjacent block. Thus, only if all but one pin of a net e reside in the *same* block V_j (i.e., $\Phi(e, V_j) = |e| - 1 \rightsquigarrow \lambda(e) = 2$), it contributes $+\omega(e)$ to the gain of moving the pin $p \notin V_j$ to block V_j. Similarly, only a net e with $\Phi(e, V_i) = |e|$ contributes $-\omega(e)$ to the move-gains of its pins. It follows that if a net e connects more than two blocks (i.e., $\lambda(e) > 2$), its gain contribution is always zero, because no single vertex move can remove e from the cut-set.

After moving a vertex v, we perform *delta-gain updates* for all moves of the remaining active border vertices in $\Gamma(v)$. Similar to the delta-gain update procedure for

Algorithm 4.7 : Delta-Gains for Cut-Net Optimization

Input : Vertex v that was moved from block V_{from} to V_{to}

Function DeltaGainUpdate($v, V_{\text{from}}, V_{\text{to}}$)
  foreach $e \in \mathrm{I}(v)$ **do** *// Iterate over all incident nets ...*
    foreach $u \in e \setminus \{v\}$ **do** *// ... and consider each pin*
      if *u is marked* **then continue** *// Skip marked vertices*
      if $\Phi(e, V_{\text{from}}) = 0 \wedge \nexists\, n \in \mathrm{I}(u) : \Phi(n, V_{\text{from}}) > 0$ **then**
        // Connectivity of net e decreased ...
        P_{from}.remove(u) *// ... and u is not adjacent to V_{from} anymore*
        $\mathrm{B}(u) := \mathrm{B}(u) \setminus \{V_{\text{from}}\}$
      if $\Phi(e, V_{\text{to}}) = 1 \wedge V_{\text{to}} \notin \mathrm{B}(u)$ **then** *// Connectivity $\lambda(e)$ increased ...*
        // ... and u was not adjacent to block V_{to} before the move
        P_{to}.insert($u, g_{\text{to}}(u)$)
      if $\Phi(e, V_{\text{from}}) = |e| - 1$ **then** *// Moving v added net e to the cut-set*
        foreach $V_i \in \mathrm{B}(u) \setminus \{V_{\text{from}}\}$ **do** P_i.update($u, \omega(e)$)
      if $\Phi(e, V_{\text{to}}) = |e|$ **then** *// Moving v removed net e from the cut-set*
        foreach $V_i \in \mathrm{B}(u) \setminus \{V_{\text{to}}\}$ **do** P_i.update($u, -\omega(e)$)
      $V_u :=$ current block of vertex u
      if $V_u \neq V_{\text{to}} \wedge \Phi(e, V_{\text{to}}) = |e| - 1$ **then**
        // Moving u to block V_{to} removes net e from the cut-set
        P_{to}.update($u, \omega(e)$)
      if $V_u \neq V_{\text{from}} \wedge \Phi(e, V_{\text{from}}) = |e| - 2$ **then**
        // Moving u to V_{from} does not remove net e from the cut-set anymore
        P_{from}.update($u, -\omega(e)$)
      if $V_{\text{to}} \notin \mathrm{B}(u)$ **then** $\mathrm{B}(u) := \mathrm{B}(u) \cup \{V_{\text{to}}\}$ *// Update adjacent blocks*
  Output : The gains for all moves of all neighbors $\Gamma(v)$ are updated.

connectivity optimization, we have to account for all changes of the gain contributions of all nets $e \in \mathrm{I}(v)$, and also address possible changes in the corresponding connectivity sets $\Lambda(e)$. A pseudocode description of the delta-gain-update procedure for cut-net optimization is given in Algorithm 4.7. Figure 4.10 depicts the different situations that yield connectivity and gain changes.

Theorem 4.4 (Correctness of Cut-Net Delta-Gain Updates)

Let v be the vertex that was moved from block V_{from} to block V_{to}. After performing delta-gain updates as described in Algorithm 4.7, all vertex moves have gain $g_j(\cdot)$.

Proof. The proof follows similar arguments as the proof of the delta-gain update procedure for connectivity optimization in Theorem 4.3. We show *how* the move of v from block V_{from} to block V_{to} affects the move gains of other vertices.

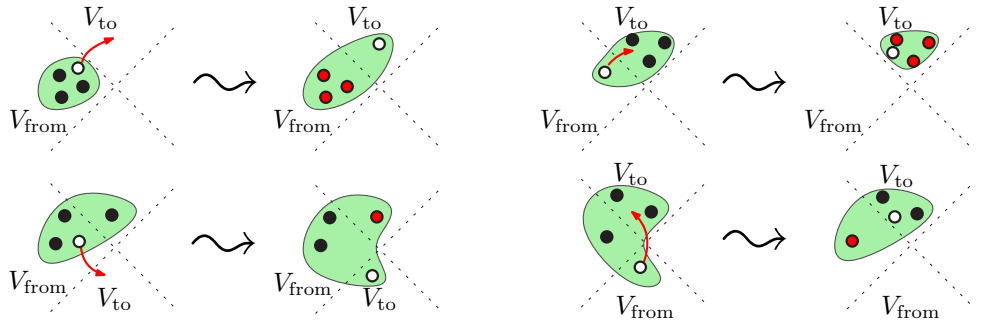


Figure 4.10: Visualization of the situations that yield changes in the gain contribution of a net $e \in \mathrm{I}(v)$ after moving vertex v for cut-net optimization. After moving the white vertex, the gains of the red vertices change.

Recall, that the gain of moving a vertex $u \in V_i$ to an adjacent block $V_j \in \mathrm{B}(u) \setminus \{V_i\}$ for cut-net optimization is defined as

$$g_j(u) := \sum_{e \in \mathrm{I}(v)} \{\omega(e) \mid \Phi(e, V_j) = |e| - 1\} - \sum_{e \in \mathrm{I}(v)} \{\omega(e) \mid \Phi(e, V_i) = |e|\}. \tag{4.10}$$

Thus, moving v only affects the gains of neighboring vertices $u \in \Gamma(v)$ and only nets $e \in \mathrm{I}(v) \cap \mathrm{I}(u)$ can change the gain of a vertex $u \in \Gamma(v)$ – all other nets have no effect. In the following we again call such nets *critical*.

Let $u \in \Gamma(v)$ be a neighbor of v, e be a critical net, and let $g_j(u)$ be the old gain (before the move of v) of moving u from its current block V_i to block $V_j \in \mathrm{B}(u) \setminus \{V_i\}$. We distinguish the following four cases (visualized in Figure 4.10):

(i) If v was moved out of u's block (i.e., $u \in V_{\text{from}}$), the gain contribution of net e only changes if the number of pins $\Phi(e, V_{\text{from}})$ decreases from $|e|$ to $|e| - 1$. Before the move, net e thus contributed $-\omega(e)$ to the gains of moving u to all adjacent blocks $V_j \in \mathrm{B}(u) \setminus \{V_{\text{from}}\}$, because it was an internal net of block V_{from}. Since the move of v made e a cut-net, net e now contributes 0 to the gains of all moves to adjacent blocks $V_j \in \mathrm{B}(u) \setminus \{V_{\text{from}}\}$. Thus, $g_j(u) = g_j(u) + \omega(e)$ (line 12).

(ii) If v was moved to u's block (i.e., $u \in V_{\text{to}}$), the gain contribution of net e only changes if the number of pins $\Phi(e, V_{\text{to}})$ increases from $|e| - 1$ to $|e|$. Since e was a cut-net before the move, it contributed 0 to the gains of moving u to all adjacent blocks $V_j \in \mathrm{B}(u) \setminus \{V_{\text{to}}\}$. Since the move of v removed net e from the cut-set, it now contributes $-\omega(e)$ to the gains of all moves to adjacent blocks $V_j \in \mathrm{B}(u) \setminus \{V_{\text{to}}\}$. Thus, $g_j(u) = g_j(u) - \omega(e)$ (line 14).

Otherwise, u is neither in block V_{from} nor in block V_{to}. In this case, the gain contribution of net e only changes if e could have been removed from the cut-set before the move or can be removed from the cut-set after the move.

(iii) The former happens if the number of pins $\Phi(e, V_{\text{from}})$ in block V_{from} decreases from $|e|-1$ to $|e|-2$. Before the move of v, net e could have been removed from the cut-set by moving u to block V_{from}. The gain contribution of net e therefore changes from $+\omega(e)$ to 0, because after the move there exist two pins (u and v) outside of V_{from}. Thus, $g_{\text{from}}(u) = g_{\text{from}}(u) - \omega(e)$ (line 20).

(iv) The latter happens if the number of pins $\Phi(e, V_{\text{to}})$ in block V_{to} increases from $|e|-2$ to $|e|-1$. After the move, $\lambda(e) = 2$ and vertex u is the only pin that is not in block V_{to}. The gain contribution of net e therefore changes from 0 to $+\omega(e)$, because e can now be removed from the cut-set by moving u to block V_{to}. Thus, $g_{\text{to}}(u) = g_{\text{to}}(u) + \omega(e)$ (line 17).

Note that gain updates are restricted to *unmarked* vertices, since each vertex is only allowed to be moved once in each pass, and a vertex becomes marked after it is moved. The case distinctions in line 5 and line 9 follow from the fact that we only allow vertices to move to *adjacent* blocks. Thus, we remove the move to block V_{from} if vertex u is not adjacent to block V_{from} anymore after the move of v. Similarly, if $V_{\text{to}} \notin \text{B}(u)$ before the move, we calculate the gain of this *new* move from scratch, since it was not allowed before the move of v. □

Excluding Nets from Delta-Gain Updates. To further reduce the running time of the delta-gain algorithms, we exclude nets from the update procedure if their contribution to the gain values of their pins cannot change. For cut-net optimization, this is done by generalizing the locked nets technique described in the previous section to k-way partitioning. For connectivity optimization, the key observation is that after moving a vertex v to a block V_{to}, this block remains connected to all nets $e \in \text{I}(v)$ during this local search pass, because v is not allowed to be moved again. In this case we say that block $V_{\text{to}} \in \Lambda(e)$ has become *unremovable* for net e. Using the following lemma, we exclude nets $e \in \text{I}(v)$ from delta-gain updates after moving a vertex v from V_{from} to V_{to} if both blocks $\{V_{\text{from}}, V_{\text{to}}\} \in \Lambda(e)$ are marked as unremovable:

Lemma 4.5 (Excluding Nets from Delta-Gain Updates)
Let v be the vertex that was moved from V_{from} to V_{to}. If both blocks are marked as unremovable in the connectivity set $\Lambda(e)$ of a net $e \in \text{I}(v)$, net e does not change its gain contribution for any of its pins if the connectivity metric is optimized.

Proof. In the proof of Theorem 4.3, we have seen that the gain contribution of a net $e \in \text{I}(v)$ only changes in four different cases. In the following, we show that none of these apply if both V_{from} and V_{to} are marked as unremovable in $\Lambda(e)$.

Let $u \in \Gamma(v)$ be a neighbor of v, and let e be a net in $\text{I}(v) \cap \text{I}(u)$.

(i) If v was moved out of u's block (i.e., $u \in V_{\text{from}}$), the gain contribution of net e only changes if $\Phi(e, V_{\text{from}})$ decreases from 2 to 1. However, since V_{from} is marked as unremovable, vertex u was moved to V_{from} during the current local search and is thus not allowed to be moved again. Since u is the only pin left in V_{from}, the gain contribution of net e therefore does not change.

(ii) If v was moved to u's block (i.e., $u \in V_{\text{to}}$), the gain contribution of net e only changes if $\Phi(e, V_{\text{to}})$ increases from 1 to 2. However, since V_{to} is already marked as unremovable, u was moved to V_{to} during the current local search and is thus not allowed to be moved again. Since u was the only pin in V_{to} affected by the move, the gain contribution of net e therefore does not change.

Otherwise, u is neither in block V_{from} nor in block V_{to}. In this case, the gain contribution of net e only changes if its connectivity $\lambda(e)$ (iii) increases and/or (iv) decreases as the result of the move. However, since both V_{from} and V_{to} are marked as unremovable, the move cannot change $\lambda(e)$. □

Exclusion from delta gain updates is integrated into our algorithm by labeling the blocks of the connectivity sets $\Lambda(\cdot)$. Initially each block is labeled *removable*. After a vertex v is moved, the label of the target block V_{to} is set to *unremovable* for all nets $e \in \mathrm{I}(v)$. If a vertex cannot be moved because the move would violate the balance constraint, its block becomes unremovable for all nets $e \in \mathrm{I}(v)$. Nets in which both the source and the target block of the current move are unremovable are then excluded from the gain update process.

4.6.3 Caching Gain Values

We now outline the details of the gain cache for both 2-way and k-way refinement.

Gain Cache for 2-way Refinement. We use an array $\rho = [v_1, \ldots, v_n]$ to store the cache entries. Let $\rho[v]$ denote the cache entry for vertex v. After initial partitioning, the gain cache is empty. If a vertex becomes activated during a local search pass, we check whether or not its gain is already cached. If it is cached, the cached value is used for activation. Otherwise, we calculate the gain according to Eq. 4.8, insert it into the cache and activate the vertex. After moving a vertex v with gain $g_j(v)$ to block V_j, its cache value is set to $\rho[v] := -g_j(v)$. The delta-gain updates of its neighbors $\Gamma(v)$ are then also applied to the corresponding cache entries. Thus, the gain cache always reflects the current state of the hypergraph. Since our algorithm performs a rollback operation at the end of a local search pass that undoes vertex moves, we also have to undo delta-gain updates applied on the cache. This can be done by additionally maintaining a *rollback delta-gain cache* that stores the negated delta-gain updates for each vertex. During rollback, this delta cache is then used to restore the gain cache to a valid state.

Each time a local search is started with an uncontracted vertex pair (u, v), we have to account for the fact that the uncontraction potentially affected $\rho[u]$. A simple variant of the caching algorithm just invalidates the corresponding cache entry and re-calculates the gain. Since v did not exist on coarser levels of the hierarchy, its gain must also be computed from scratch. For 2-way refinement, we instead use a more sophisticated variant that is able to update $\rho[u]$ based on information gathered during the uncontraction and that further infers $\rho[v]$ from $\rho[u]$. After uncontraction, we initially set $\rho[v] := \rho[u]$. Both cache entries are then updated by examining each net $e \in \mathrm{I}(u)$. We have to distinguish three cases (see Figure 4.11 for an example):

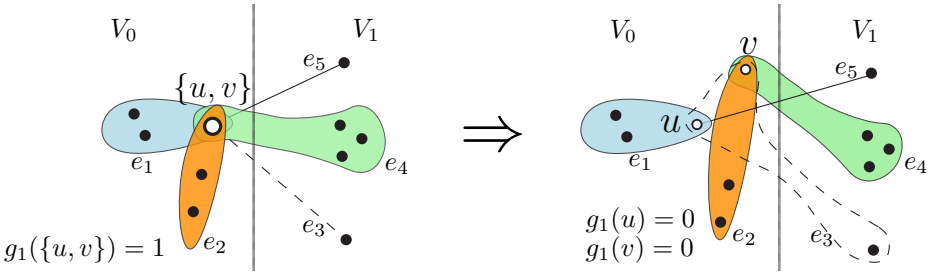


Figure 4.11: Example of an uncontraction operation that affects the cached gain value $\rho[u]$ of the representative u. Since nets $\{e_2, e_4\} \notin \mathrm{I}(u)$ after the uncontraction, they no longer contribute $-\omega(e_2)$ resp. $+\omega(e_4)$ to the gain of u. Similarly, moving u to V_1 now does not remove e_3 from the cut anymore because of v. Its contribution to $\rho[u]$ therefore becomes zero (source: [Sch+16a]).

(i) After uncontraction, u is not incident to net e anymore. If e was a cut net that could have been removed from the cut by moving u to the other block, $\rho[u]$ has to be decremented by $\omega(e)$ (see net e_4 in Fig. 4.11). Similarly, if e was an internal net, moving u would have made it a cut net. In this case, $\rho[u]$ is incremented by $\omega(e)$ (see net e_2 in Fig. 4.11).

(ii) After uncontraction, e contains both u and v. Let V_u be the block of vertex u (and thus also the block of vertex v). If $\Phi(e, V_u) = 2$, the net cannot be removed from the cut anymore by moving either u or v. We therefore have to decrement both $\rho[u]$ and $\rho[v]$ by $\omega(e)$ (see net e_3 in Fig. 4.11).

(iii) Finally, we have to account for nets to which v is not incident (nets e_1 and e_5 in Fig. 4.11). If such a net e can be removed from the cut by moving u, it contributes $\omega(e)$ to $\rho[u]$. We therefore have to decrement $\rho[v]$ by $\omega(e)$ to account for the fact that $e \notin \mathrm{I}(v)$. Similarly, if moving u makes e a cut net, we have to increment $\rho[v]$ accordingly.

Gain Data Structure for k-way Refinement. In order to generalize the 2-way gain cache to k-way partitioning, a redesign of the data structure is necessary. Since in the 2-way setting there is only one possible move for each vertex (i.e., moving it to the other block of the bipartition), a simple array is enough to store the gain values. When performing k-way local search, each vertex can potentially be moved to $k-1$ different blocks. We therefore use a modified version of a folklore data structure to store sparse sets (see, e.g., Ref. [BT93]), i.e., sets, where the number of contained elements is small compared to the size of the universe. An example is shown in Figure 4.12. For each border vertex $v \in V_i$, this data structure uses two arrays D and S to store the set of adjacent blocks $\mathrm{B}(v) \setminus \{V_i\}$ along with corresponding gain values for moving v to these blocks in $\mathcal{O}(k)$ space. A new block V_j can be added to $\mathrm{B}(v)$ in $\mathcal{O}(1)$ time by setting $D[\texttt{size}] := V_j$, $S[j] := \langle \texttt{size}, g_j(v) \rangle$, and incrementing `size`. A block V_j can

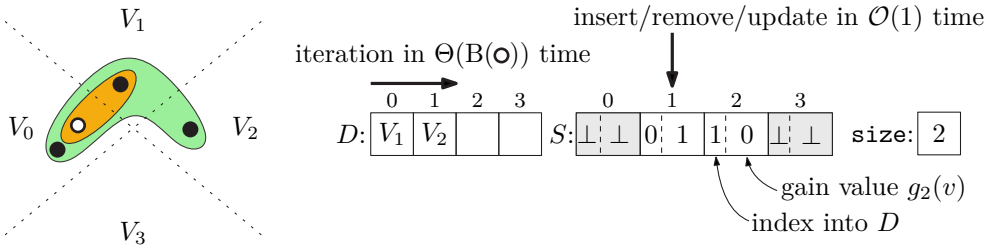


Figure 4.12: Gain cache data structure for k-way refinement that permits insertions, deletions, and gain updates in constant time, as well as iteration over the set of adjacent blocks in time $\Theta(\mathrm{B}(\cdot))$.

be removed from $\mathrm{B}(v)$ by overwriting its entry in D with the last element contained in D and updating the S entry of that element to point to the new position. Afterwards, the S entry of V_j is cleared and `size` is decremented. Using D, it is possible to iterate over $\mathrm{B}(v)$ in time $\Theta(|\mathrm{B}(v)|)$. Furthermore, a specific cache entry can be updated in $\mathcal{O}(1)$ time by updating S.

Gain Cache for k-way Refinement. Let $\rho_v[j]$ denote the cache entry for vertex v and adjacent block $V_j \in \mathrm{B}(v)$. After initial partitioning, we initialize the gain cache with all possible moves of all vertices of the coarsest hypergraph. Each time a local search is started with an uncontracted vertex pair (u, v), we invalidate and recompute their corresponding cache entries. This is necessary since v did not exist on previous levels of the hierarchy and since the uncontraction potentially affected both $\mathrm{B}(u)$ and the corresponding gain values.[4] If a vertex becomes activated during a local search pass, the cached gain values are used for activation. After moving a vertex v with gain $g_j(v)$ from block V_{from} to block V_{to}, its cache value is updated as follows: First, we remove the entry of $\rho_v[\text{to}]$, since $v \in V_{\text{to}}$ after the move and we only cache gain values for moves to adjacent blocks $\mathrm{B}(v) \setminus \{V_{\text{to}}\}$. If v remains connected to V_{from}, we set $\rho_v[\text{from}] := -\rho_v[\text{to}]$. The update of the remaining blocks in $\mathrm{B}'(v) := \mathrm{B}(v) \setminus \{V_{\text{from}}, V_{\text{to}}\}$ then depends on the objective function.

For connectivity optimization, we have to distinguish two cases for each net $e \in \mathrm{I}(v)$: If v was the only pin in V_{from} and $\Phi(e, V_{\text{to}}) \neq 1$ after the move, then moving v to a block $V_j \in \mathrm{B}'(v)$ would have decreased $\lambda(e)$. We thus have to decrement the corresponding cache entries by $\omega(e)$. Similarly, if v was not the only pin in V_{from} and $\Phi(e, V_{\text{to}}) = 1$ after the move, then moving v to a block $V_j \in \mathrm{B}'(v)$ in future local search passes will decrease $\lambda(e)$. The respective cache entries are therefore incremented by $\omega(e)$. Note that we do not have to increment the entry of $\rho_v[\text{from}]$, since it is already up to date.

For cut-net optimization, we have to distinguish two different cases for each net

[4] We also tried a version that – similar to 2-way gain caching – updates the cache entries of u based on the information gathered during uncontraction and then infers the cache entries of v from those of u. However, recomputation turned out to be faster, since updating and inferring the cache values is significantly more complicated.

$e \in \mathrm{I}(v)$: If e was an internal net before the move (i.e., $\Phi(e, V_{\text{from}}) = |e|$), it is now a cut net, because v was moved to V_{to}. Therefore, we have to increment the cache entries of adjacent blocks in $\mathrm{B}'(v)$ by $\omega(e)$, because performing these moves will not change the cut state of net e. Similarly, if the move removed e from the cut-set, the cache entries of blocks $\mathrm{B}'(v)$ have to be decremented by $\omega(e)$, since moving v out of V_{to} would make e a cut net again.

After updating the cache entries of the moved vertex, delta-gain updates of the neighbors $\Gamma(v)$ are then also applied to the corresponding cache entries. Thus the gain cache always reflects the current state of the hypergraph. Similarly to the 2-way case, we additionally maintain a *rollback delta cache* that stores the negated delta-gain updates for each vertex as well as the corresponding add/remove operation for $\mathrm{B}(\cdot)$ in order to be able to restore the gain cache to a valid state during rollback.

4.6.4 Restricting the Search Space via Stopping Rules

Motivation. Unlike multi-level partitioning algorithms, which can afford to spend linear time in refinement heuristics at each hierarchy level, the number of local search steps needs to be limited in the n-level setting. Otherwise the n-level approach could lead to $\mathcal{O}(n^2)$ local search steps in total, if refinement is executed after every uncontraction. We therefore use two stopping rules that terminate a local search pass well before every vertex is moved once.

Simple, Static Stopping Rule. During a local search pass, our static stopping rule monitors the improvement in both solution quality and partition balance and stops the pass once a constant number of i moves neither improved the objective nor the current imbalance.

Adaptive Stopping Rule. The adaptive stopping criterion is a slightly modified version of the stopping rule proposed by Osipov and Sanders [OS10a; OS10b] for n-level graph partitioning. The key idea is to make the decision when to stop the current local search pass dependent on the past history of the search. For this purpose, Osipov and Sanders [OS10a; OS10b] model the gain values in each step as identically distributed, independent random variables whose expectation μ and variance σ^2 is obtained from the previously observed p steps. They show that it is unlikely that local search can still give an improvement if $p > \sigma^2/4\mu^2$, where μ is the average gain since the last improvement, and σ^2 is the variance observed throughout the current local search. We integrate a slightly refined version of this adaptive stopping criterion into our algorithm: On each level, local search performs at least $\log n$ steps after an improvement is found and continues as long as $\mu > 0$. If μ is still 0 after $\log n$ steps, local search is stopped. This prevents the algorithm from getting stuck with zero-gain moves, which is likely for hypergraphs that contain many large nets. Otherwise (i.e., if $\mu \neq 0$) we evaluate the equation and act accordingly.

4.6.5 Implementation Details

Priority Queue Data Structures. We provide both a binary heap-based priority queue and a bucket priority queue. In accordance with the observations described in Section 3.2, the bucket PQ uses a LIFO tie-breaking scheme. Since the coarsening process increases both vertex degrees and – due to parallel net removal – hyperedge weights, the gain span of the bucket PQ becomes $[-\Delta_v \cdot \omega_{\max}, \ldots, +\Delta_v \cdot \omega_{\max}]$. Following Papa and Markov [PM07], we enhance the bucket queue data structure with a binary search tree that stores pointers to non-empty gain buckets in order to be able to find both the highest and the second-highest non-empty gain bucket in constant time. Therefore, the time to insert or delete a gain element becomes logarithmic in the number of non-empty gain buckets. We will evaluate the effects of both data structures in the n-level context in Section 4.9.

Identifying Border Vertices. In both the 2-way and the k-way algorithm, it is necessary to be able to distinguish border vertices from internal vertices efficiently. We therefore maintain the number of incident cut-nets $|\{e \in \mathrm{I}(v) | \lambda(e) > 1\}|$ for each vertex v, which allows us to identify border vertices in constant time. These values are initialized *once* at the end of the initial partitioning phase, and then maintained throughout the uncoarsening/refinement process, i.e., they are updated during uncontractions, and whenever an incident net is added to or removed from the cut-set during refinement.

4.7 Network Flow-Based Refinement

Motivation. Move-based local search algorithms are known to be prone to get stuck in local optima when used directly on the input hypergraph [KK99; KK00]. The multi-level paradigm helps to some extent, since it allows a more global view of the problem at the coarse levels and a very fine-grained view at the fine levels of the hierarchy. However, the performance of move-based approaches degrades for hypergraphs with large hyperedges. In these cases, it is difficult to find meaningful vertex moves that improve the solution quality because large hyperedges are likely to have many vertices in multiple blocks [UA04]. Thus, the gain of moving a single vertex to another block is likely to be *zero* [MP14].

While finding *balanced* minimum cuts in hypergraphs is NP-hard, a minimum cut separating two vertices can be found in polynomial time using network flow algorithms and the max-flow min-cut theorem [GT86]. Flow algorithms find an optimal min-cut and do not suffer the drawbacks of move-based approaches. However, with a few notable exceptions [LR04; AL08; SS11], they were long overlooked as heuristics for balanced partitioning due to their high complexity [YW96; LW98]. Sanders and Schulz [SS11] present a max-flow-based improvement algorithm for graph partitioning which is integrated into the multi-level partitioner KaFFPa and computes high-quality solutions. Motivated by their results, we generalize the max-flow min-cut refinement framework of KaFFPa from graphs to hypergraphs.

Overview. In Section 4.7.1, we first review the KaFFPa approach, identify shortcomings of its flow network model that restrict the search space of feasible solutions, and propose a modification to overcome these limitations. Section 4.7.2 then explains how hypergraphs are transformed into flow networks and presents a technique to reduce the size of the resulting hypergraph flow network. Section 4.7.3 shows how this network can be used to construct a flow problem such that the min-cut induced by a max-flow computation between a pair of blocks improves either the cut-net or connectivity objective of a k-way partition. In addition, we show how the modification of KaFFPa's flow network can be generalized to hypergraphs by exploiting the structure of hypergraph flow networks. Finally, we briefly discuss implementation details and techniques to improve the running time in Section 4.7.4.

4.7.1 Improved Flow-Based Refinement for Graph Partitioning

The KaFFPa Framework. Given an ε-balanced k-way partition $\Pi_k = \{V_1, \ldots, V_k\}$ of a graph $G = (V, E, c, \omega)$, KaFFPa's flow-based refinement algorithm works on *pairs* (V_i, V_j) of adjacent blocks. To coordinate these refinements, the authors propose an *active block scheduling* algorithm, which schedules blocks adjacent in the quotient graph Q as long as their participation in a pairwise refinement step improves solution quality or imbalance.

The basic idea is to build a flow network $\mathcal{N}$ based on the induced subgraph $G[B]$, where $B \subseteq \{V_i, V_j\}$ is a set of nodes around the cut of the bipartition $\Pi_2 := \{V_i, V_j\}$. The size of B is controlled by an imbalance factor $\varepsilon' := \alpha\varepsilon$, where α is a scaling parameter that is chosen adaptively depending on the result of the min-cut computation. If the heuristic found an ε-balanced partition using ε', the cut is accepted and α is increased to $\min(2\alpha, \alpha')$ where α' is a predefined upper bound. Otherwise it is decreased to $\max(\frac{\alpha}{2}, 1)$. This scheme continues until a maximum number of rounds is reached or a feasible partition that did not improve the cut is found.

In each round, the corridor $B := B_i \cup B_j$ is constructed by performing two restricted breadth-first searches (BFS). The first BFS is done in the induced subgraph $G[V_i]$. It is initialized with the boundary nodes of V_i and stops if $c(B_i)$ would exceed $(1 + \varepsilon')\lceil \frac{c(V)}{k} \rceil - c(V_j)$. The second BFS constructs B_j in an analogous fashion using $G[V_j]$. Let $B^{\rightarrow} := \{u \in B \mid \exists (u, v) \in E : v \notin B\}$ be the border of B. Then $\mathcal{N}$ is constructed by connecting all border nodes $B^{\rightarrow} \cap V_i$ of $G[B]$ to an additional source node s and all border nodes $B^{\rightarrow} \cap V_j$ to an additional sink node t using directed edges with an edge weight of ∞. By connecting s and t to the respective border nodes, it is ensured that edges incident to border nodes, but not contained in $G[B]$, cannot become cut edges. All edges of $G[B]$ use the corresponding edge weights of G as capacities. An example is shown in Figure 4.13. For $\alpha = 1$, the size of B ensures that $\mathcal{N}$ has the *cut property*, i.e., *each* (s, t)-min-cut in $\mathcal{N}$ yields a possibly smaller cut in G that is feasible with respect to the original balance constraint of the k-way partition. For larger values of α, this does not have to be the case.

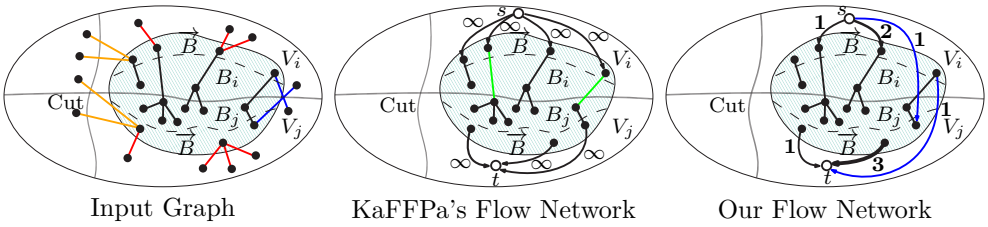


Figure 4.13: The input graph is shown on the left. Colors are used to distinguish non-cut border edges (red), cut border edges (blue), and peripheral edges (orange). In the KaFFPa flow network shown in the middle, green edges cannot be removed from the cut by moving border nodes from V_i to V_j because they are locked in block V_i. Furthermore, nodes incident to peripheral edges are locked in their block, because they are connected to s or t via infinite-capacity edges. Our improved flow network is shown on the right. All nodes of $G[B]$ are movable. Note that the blue edges account for cut border edges (source: [HSS19a]).

Most Balanced Minimum Cuts. After computing a max-flow in $\mathcal{N}$, the algorithm tries to find a min-cut with better balance. This is done by exploiting the fact that *one* (s,t)-max-flow contains information about *all* (s,t)-min-cuts [PQ80]. More precisely, the algorithm uses the 1–1 correspondence between (s,t)-min-cuts and closed sets containing s in the Picard-Queyranne-DAG $D_{s,t}$ of the residual graph $\mathcal{N}_f$ [PQ80]. First, $D_{s,t}$ is constructed by contracting each strongly connected component of the residual graph. Then the following heuristic (called *most balanced minimum cuts*) is repeated several times using different random seeds: Closed sets containing s are computed by sweeping through the nodes of $D_{s,t}$ in reverse topological order (e.g. computed using a randomized DFS). Each closed set induces a differently balanced min-cut and the one with the best balance (with respect to the original balance constraint) is used to improve the cut and/or balance between blocks V_i and V_j.

Improving the Flow Network. In the following we distinguish between *peripheral* edges $E^{\sim} := \{(u,v) \in E : u \in B^{\rightarrow} \wedge v \notin \Pi_2\}$ and *border* edges $E^{\leftrightarrow} := \{(u,v) \in E : u \in B^{\rightarrow} \wedge v \in \Pi_2 \setminus B\}$. By connecting the source s to all nodes $\mathcal{S} = B^{\rightarrow} \cap V_i$ and all nodes $\mathcal{T} = B^{\rightarrow} \cap V_j$ to the sink t using directed edges with infinite capacity, the KaFFPa network ensures that the max-flow computation does not affect border nodes and thus neither peripheral nor border edges. However, this unnecessarily *restricts* the search space of feasible solutions. Since border nodes $B^{\rightarrow}$ are directly connected to either s or t via infinite capacity edges, *every* min-cut $(S, B \setminus S)$ will have $\mathcal{S} \subseteq S$ and $\mathcal{T} \subseteq B \setminus S$. The KaFFPa model therefore prevents (i) *all* min-cuts in which any non-cut border edge becomes part of the cut-set, (ii) *all* cut border edges from being removed from the cut-set, and (iii) *all* border nodes incident to peripheral but not to border edges from changing their block (see Figure 4.13 for an example). This

restricts the space of possible solutions, since the corridor B was computed such that *even* a min-cut along either side of the border would result in a feasible cut in G. Thus, ideally, *all* vertices $v \in B$ should be able to change their block as a result of the (s, t)-max-flow computation in $\mathcal{N}$ – not only vertices $v \in B \setminus B^{\rightarrow}$. This limitation becomes increasingly relevant for graphs with high average node degrees as well as for partitioning problems with small imbalance ε, since high-degree nodes u are likely to have some neighbors $\Gamma(u) \notin G[B]$ (i.e., some of their incident edges are either border or peripheral edges) and tight balance constraints enforce small B-corridors.

We overcome these restrictions by treating border nodes differently. Since peripheral edges are cut edges in the k-way partition of G that are *not* affected if incident border nodes $u \in G[B]$ change their block (i.e., they will remain cut edges in G), border nodes only incident to peripheral edges are *neither* connected to s *nor* t. Furthermore, instead of using infinite capacity edges to connect the source or the sink to the remaining border nodes, we use the sum of the *actual* edge weights of incident border edges as capacities. More precisely, s is connected to border nodes $u \in B^{\rightarrow}$ using an edge (s, u) with capacity $c(s, u) := \sum_{(\Gamma(u) \setminus B) \cap V_i} \omega(u, v)$ and u is connected to t using an edge (u, t) with capacity $c(u, t) := \sum_{(\Gamma(u) \setminus B) \cap V_j} \omega(u, v)$ (see Figure 4.13 for an example). Since it is now possible for a max-flow computation to saturate border edges, we get a flow network that (i) does not lock *any* node $u \in G[B]$ in its block and (ii) correctly models the impact of the max-flow min-cut computation on the solution quality of the k-way partition of G.

4.7.2 Hypergraph Max-Flow Min-Cut Refinement

In the following, we generalize the KaFFPa algorithm to hypergraph partitioning. We first show how hypergraph flow networks $\mathcal{N}$ are constructed in general and introduce a technique to reduce their size by removing *low-degree* vertices. Given a k-way partition $\Pi_k = \{V_1, \ldots, V_k\}$ of a hypergraph H, a pair of blocks (V_i, V_j) adjacent in the quotient graph Q, and a corridor B, Section 4.7.3 then explains how $\mathcal{N}$ is used to build a flow problem $\mathcal{F}$ based on the subhypergraph $H_B = (V_B, E_B)$. By connecting the source node s and the sink node t to specific nodes of the flow network, $\mathcal{F}$ is constructed such that an (s, t)-max-flow computation optimizing the *cut-net* metric in the bipartition $\Pi_2 = (V_i, V_j)$ of H_B improves either the cut-net metric $f_c(\Pi)$ or the connectivity metric $f_\lambda(\Pi)$ in the k-way partition of H. Section 4.7.4 then discusses the integration into KaHyPar and introduces several techniques to speed up flow-based refinement. A pseudocode description of the entire flow-based refinement framework is given in Algorithm 4.8.

From Hypergraphs to Flow Networks. Given a hypergraph $H = (V, E, c, \omega)$ and two distinct vertices s and t, we first reduce the problem of finding an (s, t)-min-cut in H to the problem of finding a minimum-weight (s, t)-node-separator in the star-expansion G_*, where each star-node e has weight $c(e) = \omega(e)$ and all other nodes v have a weight of infinity [HM85]. This network is then transformed into the *edge-capacitated* flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ of Lawler [Law73] as follows: $\mathcal{V}$ contains

Algorithm 4.8 : k-way Flow-Based Refinement Framework (adapted from [HSS19a])

Input : Hypergraph H
Input : k-way partition $\Pi_k = \{V_1, \ldots, V_k\}$
Input : Imbalance parameter ε.

```
1  Algorithm MaxFlowMinCutRefinement(H, ε, Π_k)
2      Q := QuotientGraph(H, Π_k)
3      while ∃ active blocks ∈ Q do                    // In the beginning all blocks are active
4          foreach {(V_i, V_j) ∈ Q | V_i ∨ V_j is active} do    // Choose a pair of blocks
5              Π_old := Π_best := {V_i, V_j} ⊆ Π_k      // Extract bipartition to be improved
6              ε_old := ε_best := imbalance(Π_k)       // Imbalance of current k-way partition
7              α := α'                                 // Use large B-corridor for first iteration
8              do                                      // Adaptive flow iterations
9                  B := computeB-Corridor(H, Π_best, αε)  // As described in Section 4.7.1
10                 H_B := SubHypergraph(H, B)          // Create subhypergraph
11                 N_B := FlowNetwork(H_B)             // As described in Section 4.7.2
12                 F := FlowProblem(N_B)               // As described in Section 4.7.3
13                 f := maxFlow(F)                     // Compute maximum flow on F
14                 Π_f := mostBalancedMinCut(f, F)     // As in Section 4.7.1 & 4.7.2
15                 ε_f := imbalance(Π_f ∪ Π_k \ Π_old)  // Imbalance of new k-way partition
16                 if (cut(Π_f) < cut(Π_best) ∧ ε_f ≤ ε) ∨ ε_f < ε_best then // Better solution
17                     α := min(2α, α'), Π_best := Π_f, ε_best := ε_f // Update α, Π_best, ε_best
18                 else α := α/2                       // Decrease size of B-corridor in next iteration
19             while α ≥ 1
20             if Π_best ≠ Π_old then                  // Improvement found
21                 Π_k := Π_best ∪ Π_k \ Π_old          // Replace Π_old with Π_best
22                 activateForNextRound(V_i, V_j)      // Reactivate blocks for next round
23     return Π_k
```

Output : Improved ε-balanced k-way partition $\Pi_k = \{V_1, \ldots, V_k\}$

all non-star nodes v. For each star-node e, add two *bridging* nodes e' and e'' to $\mathcal{V}$ and a *bridging* edge (e', e'') with capacity $c(e', e'') = c(e)$ to $\mathcal{E}$. For each neighbor $u \in \Gamma(e)$, add two edges (u, e') and (e'', u) with infinite capacity to $\mathcal{E}$. The size of this network can be reduced by distinguishing between star-nodes corresponding to multi-pin nets and those corresponding to two-pin nets in H. In the flow network of Liu and Wong [LW98] the former are transformed as described above, while the latter (i.e., star-nodes e with $|\Gamma(e)| = |\{u, v\}| = 2$) are replaced with two edges (u, v) and (v, u) with capacity $c(e)$. For each such star-node, this decreases the network size by two nodes and three edges. Figure 4.14 shows both networks as well as ours, which we describe in the following.

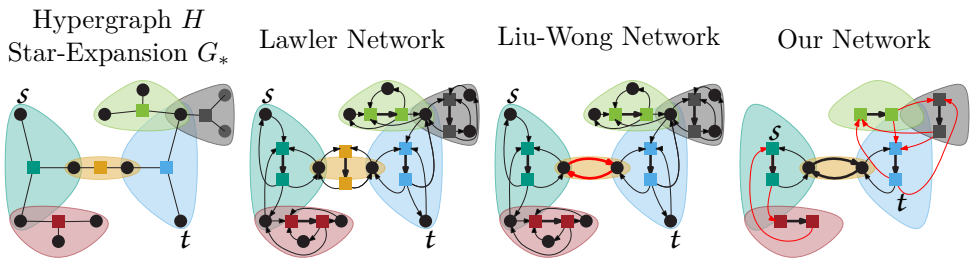


Figure 4.14: Unweighted hypergraph H with overlayed star-expansion G_* and illustration of the hypergraph flow networks. Thin, directed edges have infinite capacity, thick edges have unit capacity. Differences between the networks are highlighted in red: Special treatment of two-pin nets in the network of Liu and Wong [LW98], removal of low-degree vertices in our network (source: [HSS19a]).

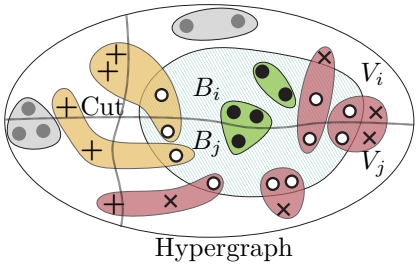
Removing Low-Degree Vertices. We further decrease the network size by using the observation that the (s,t)-node-separator in G_* has to be a subset of the star-nodes, since all other nodes have infinite capacity. Thus, it is possible to replace *any* infinite-capacity node by adding a clique between all adjacent star-nodes without affecting the separator. The key observation now is that an infinite-capacity node v with degree $d(v)$ induces $2d(v)$ edges in the Lawler network [Law73], while a clique between star-nodes induces $d(v)(d(v)-1)$ edges. For non-star nodes v with $d(v) \leq 3$, it therefore holds that $d(v)(d(v)-1) \leq 2d(v)$. We therefore remove all infinite-capacity nodes v corresponding to hypernodes with $d(v) \leq 3$ that are *not* incident to any two-pin nets by adding a clique between all star-nodes $\Gamma(v)$. In case v was either source or sink, we create a multi-source multi-sink problem by adding the star-nodes $\Gamma(v)$ to the set of sources resp. sinks [FF62]. We then apply the transformation of Liu and Wong [LW98].

Reconstructing Min-Cuts. After computing an (s,t)-max-flow f in the Lawler or Liu-Wong network, an (s,t)-min-cut of H can then be computed using a BFS in the residual graph $\mathcal{N}_f$ starting from s [Law73]. Let S be the set of nodes corresponding to vertices of H reached by the BFS. Then $(S, V \setminus S)$ is an (s,t)-min-cut. Since our network does not contain low degree vertices, we use the following lemma to compute an (s,t)-min-cut of H:

Lemma 4.6 (Min-Cut Reconstruction Via Bridging Nodes)

Let f be a maximum (s,t)-flow in the Lawler network $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ of a hypergraph $H = (V, E)$ and $(\mathcal{S}, \mathcal{V} \setminus \mathcal{S})$ be the corresponding (s,t)-min-cut in $\mathcal{N}$. Then for each node $v \in \mathcal{S} \cap V$, the residual graph $\mathcal{N}_f = (\mathcal{V}_f, \mathcal{E}_f)$ contains at least one path $\langle s, \ldots, e'' \rangle$ to a bridging node e'' of a net $e \in \mathrm{I}(v)$. Thus $(A, V \setminus A)$ is an (s,t)-min-cut of H, where $A := \{v \mid \exists e \in E : v \in e \wedge \langle s, \ldots, e'' \rangle \text{ in } \mathcal{N}_f\}$.

Proof. Since $v \in \mathcal{S}$, there has to be some path $s \rightsquigarrow v$ in $\mathcal{N}_f$. By definition of the flow network, this path can either be of the form $P_1 = \langle s, \ldots, e'', v \rangle$ or $P_2 = \langle s, \ldots, e', v \rangle$



$B^{\rightarrow} := \{\circ\}$ inner border nodes
$B^{\leftarrow} := \{\times\}$ outer border nodes
$B^{\sim} := \{+\}$ peripheral nodes
$E^{\rightarrow} := \{$ $\}$ inner nets
$E^{\leftarrow} := \{$ $\}$ outer nets
$E^{\leftrightarrow} := \{$ $\}$ border nets
$E^{\sim} := \{$ $\}$ peripheral nets

Figure 4.15: Vertex and net classification of Section 4.7.3 (source: [HSS19a]).

for some bridging nodes e', e'' corresponding to nets $e \in \mathrm{I}(v)$. In the former case we are done, since $e'' \in P_1$. In the latter case the existence of edge $(e', v) \in \mathcal{E}_f$ implies that there is a positive flow $f(v, e') > 0$ over edge $(v, e') \in \mathcal{E}$. Due to flow conservation, there exists at least one edge $(e'', v) \in \mathcal{E}$ with $f(e'', v) > 0$, which implies that $(v, e'') \in \mathcal{E}_f$. Hence, we can extend the path P_2 to $\langle s, \ldots, e', v, e'' \rangle$. □

Furthermore, this allows us to employ the most balanced minimum cut heuristic as described in Section 4.7.1. By the definition of closed sets it follows that if a bridging node e'' is contained in a closed set C, then all nodes $v \in \Gamma(e'')$ (corresponding to vertices of H) are also contained in C. Thus, we can use the respective bridging nodes e'' as representatives of removed low-degree vertices.

4.7.3 Constructing the Hypergraph Flow Problem

Let $H_B = (V_B, E_B)$ be the subhypergraph of $H = (V, E)$ that is induced by a corridor B computed in the bipartition $\Pi_2 = (V_i, V_j)$. In the following, we distinguish between the set of *inner* border nodes $B^{\rightarrow} := \{v \in V_B \mid \exists e \in E : \{u, v\} \subseteq e \wedge u \notin V_B\}$, the set of *outer* border nodes $B^{\leftarrow} := \{u \in \Pi_2 \setminus V_B \mid \exists e \in E : \{u, v\} \subseteq e \wedge v \in V_B\}$ and the set of *peripheral* nodes $B^{\sim} := \{u \notin \Pi_2 \mid \exists e \in E : \{u, v\} \subseteq e \wedge v \in V_B\}$. Similarly, we now distinguish between *outer* nets $E^{\leftarrow} := \{e \in E : e \cap V_B = \emptyset\}$ with no pins inside H_B, *inner* nets $E^{\rightarrow} := \{e \in E : e \cap V_B = e\}$ with all pins inside H_B, the set of *border* nets $E_B^{\leftrightarrow} := \{e \in E \mid e \in \mathrm{I}(B^{\rightarrow}) \cap \mathrm{I}(B^{\leftarrow})\}$, and the set of *peripheral* nets $E^{\sim} := \{e \in E : e \in \mathrm{I}(B^{\rightarrow}) \cap \mathrm{I}(V \setminus \Pi_2)\}$. A visualization of these definitions is shown in Figure 4.15.

A hypergraph flow problem $\mathcal{F}$ consists of a flow network $\mathcal{N}_B = (\mathcal{V}_B, \mathcal{E}_B)$ derived from H_B and two *additional* nodes s and t that are connected to some nodes $v \in \mathcal{V}_B$. It has the cut property if the max-flow induced min-cut bipartition Π_f of H_B does not worsen the partitioning objective in H. For both the cut-net and the connectivity objective, it thus has to hold that $f_c(\Pi_f) \leq f_c(\Pi_2)$, since $f_\lambda(\Pi) = f_c(\Pi)$ for bipartitions. While outer nets are not affected by a max-flow computation, the max-flow min-cut theorem [FF56] ensures the cut property for all inner nets. Peripheral and border nets however require special attention. Since these nets are only *partially* contained in H_B

and $\mathcal{N}_B$, they will remain connected to all blocks $\Lambda(e) \setminus \{V_i, V_j\}$ regardless of the result of the max-flow computation. It is therefore necessary to "encode" information about peripheral and border nets into the flow problem. For simplicity, we first discuss how this is done using the traditional KaFFPa approach described in Section 4.7.1 and then show how the improved flow model can be generalized from graphs to hypergraphs.

Cut-Net Optimization. In graph partitioning, peripheral edges are not part of the induced subgraph $G[B]$ and thus not contained in the resulting flow network since they have no influence on the edge-cut of the bipartition Π_2. In hypergraph partitioning however, peripheral nets $e \in E^{\sim}$ are *partially* contained in the subhypergraph H_B. Since $\Lambda(e) \setminus \Pi_2 \neq \emptyset$ for these nets, a max-flow min-cut computation in the flow network of H_B cannot remove them from the cut-set of the k-way partition Π_k. To account for that fact, we remove all peripheral nets from H_B before constructing the hypergraph flow network $\mathcal{N}_B$. Border nets $e \in E^{\leftrightarrow}$ on the other hand remain connected to the blocks of their outer border nodes in Π_2. A special case unique to hypergraphs are border nets e that are connected to both V_i and V_j by some outer border nodes $B^{\leftarrow} \cap e$. As with peripheral nets, a max-flow computation in the flow network of H_B will not be able to remove these nets (i.e., border nets $e \in E^{\leftrightarrow} : \Phi(e, V_i \setminus B_i) \geq 1 \ \wedge \ \Phi(e, V_j \setminus B_j) \geq 1$) from the cut, since they are locked in the cut-set of Π_2. We therefore remove them from H_B along with the peripheral nets before constructing the flow network $\mathcal{N}_B$. To account for the remaining border nets that are only connected to either V_i or V_j, we generalize the KaFFPa approach by connecting s to all nodes $\mathcal{S} = B^{\rightarrow} \cap V_i$ if $\Phi(e, V_i \setminus B_i) \geq 1$ and all nodes $\mathcal{T} = B^{\rightarrow} \cap V_j$ to t if $\Phi(e, V_j \setminus B_j) \geq 1$ using directed edges with infinite capacity.

Connectivity Optimization. While border nets are treated the same way as for cut-net optimization, we do not remove peripheral nets $e \in E^{\sim}$ from H_B before constructing the flow network $\mathcal{N}_B$ when optimizing the connectivity metric. Since these nets are partially contained in H_B, a max-flow min-cut computation in $\mathcal{N}_B$ can remove them from the cut-set of Π_2. This decreases the connectivity $\lambda(e)$ and thus improves the overall solution quality $\mathfrak{f}_\lambda(\Pi_k)$ of the k-way partition.

Improving the Model. We exploit the structure of hypergraph flow networks such that (s, t)-max-flow computations can also cut through non-cut border nets. The limitation of the original KaFFPa model becomes increasingly relevant for hypergraph partitioning as large nets are likely to be only partially contained in H_B and thus likely to be border nets. Instead of directly connecting s and t to inner border nodes $B^{\rightarrow}$ as described above and thus preventing all min-cuts in which these nodes switch blocks, we conceptually extend H_B to contain all outer border nodes $B^{\leftarrow}$ and the remaining border nets $E_B^{\leftrightarrow}$. The resulting hypergraph is $H_B^{\leftarrow} = (V_B \cup B^{\leftarrow}, \{e \in E \mid e \cap V_B \neq \emptyset\})$.

The key insight now is that by using the flow network of $H_B^{\leftarrow}$ and connecting s resp. t to the *outer* border nodes $B^{\leftarrow} \cap V_i$ resp. $B^{\leftarrow} \cap V_j$, we get a flow problem that does not lock *any* node $v \in V_B$ in its block, since none of them is directly connected to either s or t. Due to the max-flow min-cut theorem [FF56], this flow problem has the cut property, since all border nets of H_B are now internal nets and all external border

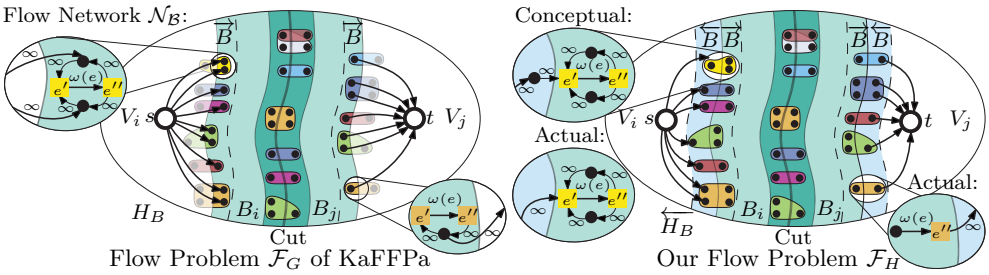


Figure 4.16: Comparison of the KaFFPa [SS11] flow problem $\mathcal{F}_G$ and our flow problem $\mathcal{F}_H$ for bipartitioning. For clarity, the zoomed in view is based on the Lawler network (source: [HSS19a]).

nodes $B^{\leftarrow}$ are locked inside their block. However, it is not necessary to use $H_B^{\leftarrow}$ instead of H_B to achieve this result. For all $v \in B^{\leftarrow}$ the flow network of $H_B^{\leftarrow}$ contains paths $\langle s, v, e' \rangle$ and $\langle e'', v, t \rangle$ that only involve infinite-capacity edges. Therefore, we can remove all nodes $v \in B^{\leftarrow}$ by directly connecting s and t to the corresponding *bridging nodes* e', e'' via infinite-capacity edges without affecting the maximal flow [FF62]. More precisely, in the flow problem $\mathcal{F}_H$, we connect s to all bridging nodes e' corresponding to border nets $e \in E_B^{\leftrightarrow} : e \subset B^{\leftarrow} \cap V_i$ and all bridging nodes e'' corresponding to border nets $e \in E_B^{\leftrightarrow} : e \subset B^{\leftarrow} \cap V_j$ to t using directed, infinite-capacity edges.

Single-Pin Border Nets. Border nets with $|e \cap B^{\rightarrow}| = 1$ can furthermore be modeled more efficiently. For such a net e, the flow problem contains paths of the form $\langle s, e', e'', v \rangle$ or $\langle v, e', e'', t \rangle$ which can be replaced by paths of the form $\langle s, e', v \rangle$ or $\langle v, e'', t \rangle$ with $c(e', v) = \omega(e)$ resp. $c(v, e'') = \omega(e)$. In both cases we can thus remove one bridging node and two infinite-capacity edges. A comparison of the original KaFFPa flow problem $\mathcal{F}_G$ and our improved version $\mathcal{F}_H$ is shown in Figure 4.16.

4.7.4 Implementation Details

Since we perform n-level partitioning, our FM-based local search algorithms are executed each time a vertex is uncontracted. To prevent expensive recalculations, we therefore use the caching techniques described in Section 4.6.3. In order to combine our flow-based refinement framework with FM local search, we not only perform the moves induced by the max-flow min-cut computation but also update the FM gain cache accordingly. Since it is not feasible to execute our flow algorithm on every level of the n-level hierarchy, we use an exponentially spaced approach that performs flow-based refinements after uncontracting $i = 2^j$ vertices for $j \in \mathbb{N}_+$. This way, the algorithm is executed more often on smaller flow problems than on larger ones. To further improve the running time, we introduce the following speedup techniques: (i) We modify active block scheduling such that after the first round the algorithm is only executed on a pair of blocks if at least one execution using these blocks led to an

improvement on previous levels. (ii) We skip flow-based refinement if the cut between two adjacent blocks is less than ten on all levels except the finest. (iii) We stop resizing the B-corridor if the current cut did not improve the previous best solution.

4.8 Framework Configuration – rKaHyPar and kKaHyPar

We provide two different framework configurations – one for recursive bipartitioning (rKaHyPar) and one for direct k-way partitioning (kKaHyPar).[5] In the following, we briefly outline the corresponding parameter settings.

Common Parameters. Both algorithms have several configuration parameters in common. Pin sparsification is enabled for hypergraphs with median net size $|\tilde{e}| \geq 28$. The minimum cluster size for sparsification $c_{\min}$ is set to two and the maximum cluster size $c_{\max}$ is set to ten. The minimum fingerprint size $h_{\min}$ for which two vertices are considered to be similar is set to ten. The maximum fingerprint size $h_{\max}$ is set to 100. The number of passes l is set to five.[6]

For community detection, the edge weighting scheme is chosen dynamically at runtime depending on the edge density δ of the hypergraph. If $\delta \geq 0.75$, uniform edge weights are used, otherwise we use $\omega_{\text{de}}(v, e)$. Furthermore, we restrict the Louvain algorithm to perform at most 100 iterations on each level and stop the first phase of the algorithm if the improvement in modularity is below 0.0001.

For flow-based refinement, we use our flow network and the proposed flow model $\mathcal{F}_H$. The flow problems are solved using the highly-tuned incremental breadth-first search (IBFS) algorithm [Gol+11], which performed best in preliminary experiments [Heu18a]. The maximum scaling parameter α' that controls the size of the B corridor is set to 16. All three speedup techniques described in Section 4.7.4 are enabled by default. Unless stated otherwise, pin sparsification, community detection, and flow-based refinements are enabled in the following experiments.

***r*KaHyPar.** Our recursive bipartitioning configuration uses cut-net splitting when configured to optimize the connectivity metric $f_\lambda(\Pi)$ and cut-net removal for $f_c(\Pi)$-optimization (see Section 4.2). It uses the n-level coarsening algorithm described in Section 4.4.1 with the lazy re-rating strategy and a penalty factor $\gamma(u, v) := 1/(c(v) \cdot c(u))$. The coarsening process is stopped as soon as the number of vertices drops below 320 (i.e., $t = 160$) or no eligible vertex is left. The scaling factor s for the maximum allowed vertex weight during coarsening is set to 3.25. Once the hypergraph is small enough, the portfolio-based approach described in Section 4.5 is used to compute an initial bipartition. Localized local search is performed using the 2-way FM algorithm described in Section 4.6. The algorithm uses the simple stopping rule to restrict the search space and stops after $i = 350$ moves neither improved the objective nor the current imbalance.

[5] The corresponding configurations are publicly available at: `http://kahypar.org/`

[6] These parameters were chosen based on preliminary experiments done by Yaroslav Akhremtsev.

***k*KaHyPar.** Our direct k-way configuration uses the simple and fast greedy coarsening algorithm described in Section 4.4.2. The threshold parameter ι for evaluating the rating function is set to $\iota = 1000$. The scaling factor s for the maximum allowed vertex weight during coarsening is set to 1, and the coarsening process is stopped once the number of vertices drops below $160 \cdot k$ (i.e., $t = 160$) or no eligible vertex is left. For initial partitioning, the k-way configuration uses rKaHyPar with the following parameters: We use the simple and fast greedy coarsening algorithm, s and γ are set to 1, and the initial partitioning coarsening process continues until the number of vertices drops below 300 (i.e., $t = 150$). Initial bipartitions are computed using the portfolio approach. For refinement during initial partitioning, the 2-way localized local search algorithm uses the simple stopping rule and stops after $i = 50$ moves neither improved the cut nor the current imbalance. After computing the initial k-way partition using this rKaHyPar configuration, the partition is further refined using the localized k-way local search algorithm described in Section 4.6.2. The search space is restricted using the adaptive stopping rule.

4.9 Experimental Evaluation

Outline. We now evaluate the different components that make up the direct k-way and the recursive bipartitioning algorithms, starting with the pin sparsifier in Section 4.9.1 and the community aware coarsening framework in Section 4.9.2. Since both techniques were developed for direct k-way partitioning, their usefulness in a recursive bipartitioning setting was not studied in the corresponding conference publications [Akh+17a; HS17a]. In Section 4.9.3, we therefore evaluate both techniques in the context of rKaHyPar. Furthermore, since rKaHyPar uses the PQ-based n-level coarsening algorithm, Section 4.9.4 investigates the effects of using the lazy re-rating approach instead of the traditional re-rating strategy.

We then turn to the refinement phase. In Section 4.9.5, we explore the question whether or not LIFO bucket priority queues still play an important role for FM-based local search algorithms in the n-level setting. Section 4.9.6 then illustrates that both the caching of gain values and the restriction of the search space are essential in order to achieve reasonable running times for n-level FM-based refinement heuristics. In Section 4.9.7, we examine different aspects of the flow-based refinement framework. Section 4.9.8 then concludes the experimental evaluation by analyzing the effectiveness of community-aware coarsening and flow-based refinement when compared to weaker but faster framework configurations.

With one exception, all experiments presented in this section were redone from scratch for this dissertation in order to get a consistent overview. The experimental results used to demonstrate the effects of the different flow networks and max-flow algorithms in Section 4.9.7 are re-used from the corresponding publications [HSS18a; HSS19a], because they were done in isolation and are thus independent of the overall framework configuration.

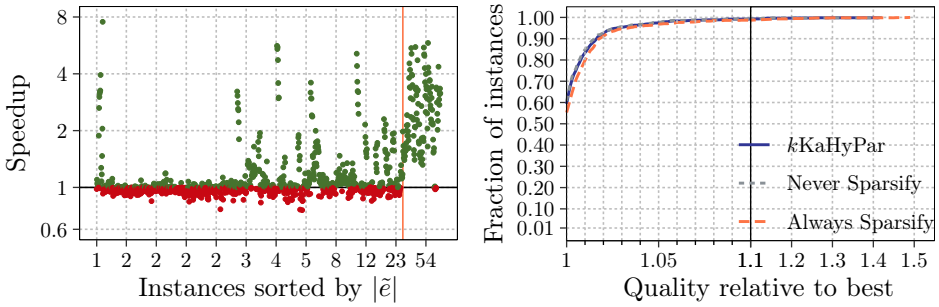


Figure 4.17: Speedup of kKaHyPar using the pin sparsifier over a configuration without sparsification (left). Effects of sparsification on solution quality (right). Experiments are performed on benchmark set B.

4.9.1 Effects of Pin Sparsification

Motivation. The pin sparsifier presented in Section 4.3.1 is intended to improve the running time for hypergraphs with large nets. At the time of publication of the corresponding conference paper [Akh+17a], KaHyPar did not yet include the community-aware coarsening scheme as well as the flow-based refinement framework. Furthermore, the benchmark set at that time did not include instances from the DAC 2012 Routability-Driven Placement Contest [Vis+12] and was missing primal and dual SAT instances [MP14]. Moreover, the decision when to activate the sparsifier was based on experiments using the entire benchmark set available at that time. In order to show that the parameter choice is still reasonable for the KaHyPar configurations used in this dissertation, we therefore re-evaluate the decision to activate the pin sparsifier for hypergraphs with median net size $|\tilde{e}| \geq 28$ on benchmark set B. The experiments use kKaHyPar to optimize the connectivity metric $f_\lambda(\Pi)$.

Results. The results are shown in Figure 4.17. For small median net sizes, the preprocessing still results in a slight slowdown of the partitioning process. The effect is less pronounced than in Ref. [Akh+17a], because both community detection and flow-based refinement increase the running time of the algorithm. However, for hypergraphs with larger median net sizes, the sparsifier still speeds up the partitioning process. We additionally see that the impact on solution quality is negligible, and thus conclude that given the current framework configuration, enabling the sparsifier for all hypergraphs with median net size $|\tilde{e}| \geq 28$ is still a sensible choice. For more details on sparsification, we refer to the dissertation of Yaroslav Akhremtsev [Akh19].

4.9.2 Insights into Community-Aware Coarsening

Motivation. Since the flow-based refinement algorithm did not yet exist at the time when the community-aware coarsening framework was published, we re-evaluate the decisions made in Ref. [HS17a] for kKaHyPar on benchmark set B, again optimizing the

Table 4.1: Edge density classification of the hypergraphs in benchmark set B.

Benchmark Suite	ISPD98	DAC2012	SuiteSparse	SAT14		
				Literal	Primal	Dual
Edge Density δ	≈ 1	≈ 1	$\{\ll, \approx, \gg\}$ 1	$\gg 1$	$\gg 1$	$\ll 1$

connectivity objective $f_\lambda(\Pi)$. For the following experiments, we divide the hypergraphs of set B into three edge density classes (see Table 4.1). The class $\delta \ll 1$ is comprised of all hypergraphs with $\delta < 0.75$. Hypergraphs with $0.75 \leq \delta \leq 1.25$ form class $\delta \approx 1$, while hypergraph with $\delta > 1.25$ are assigned to class $\delta \gg 1$. While VLSI hypergraphs have $|V| \simeq |E|$ and therefore $d \simeq 1$ [CL04; PM07], SAT hypergraphs exhibit different densities. A primal (or literal) hypergraph of a SAT formula with n variables and $m \in \mathcal{O}(n)$ clauses has edge density $\delta \gg 1$, while its dual representation has $\delta \ll 1$. Instances derived from sparse matrices cover all three cases. While it is known that VLSI circuits and complex networks like web graphs and social networks have a naturally existing clustering structure [DD96b; FH16], recent work [AGL12; GL16] suggests the same for industrial SAT instances.

Evaluation of Edge Weighting Schemes. Figure 4.18 summarizes the results of our experiments using different edge weights for the bipartite graph edges as described in Section 4.3.2. For each edge density class, a box plot shows the improvement of kKaHyPar using the community aware-coarsening framework with the corresponding edge weighting scheme over kKaHyPar without community-aware coarsening. The plots show the improvements in average solution quality for the initial partitions (computed by the initial partitioning algorithm) as well as the improvement in average and best solution quality of the final k-way partitions (after uncoarsening and refinement).

Using uniform edge weights for low density hypergraphs worsens the solution quality. However, although the initial solutions are significantly worse in this case, the best solutions are only 1.87% worse on average than those of kKaHyPar without community detection. This shows the strength of the n-level approach combined with strong refinement heuristics. Weighting schemes that encode structural information about the hypergraph into the edge weights perform significantly better. Both ω_e and ω_{de} ensure that the community structure of the bipartite graph is not dominated by high-degree E-nodes (large nets) by incorporating the net sizes into the edge weights. However, we can see that ω_{de} is more stable than ω_e. Its mean improvement is close to the median, always above zero, and always above the mean improvement of ω_e, which shows that additionally strengthening the connection between E-nodes and high-degree V-nodes indeed has a positive impact on solution quality. For hypergraphs with edge density $\delta \approx 1$ uniform edge weights perform best. If the density of the hypergraph is large, all three schemes give comparable results. This can be explained by the fact that if $\delta \gg 1$, most nets are small. This translates to "small stars" in the bipartite graph (or even paths for nets with $|e| = 2$), which do not distort the community structure of V-nodes. Based on these results, we retain the configuration described in Section 4.8.

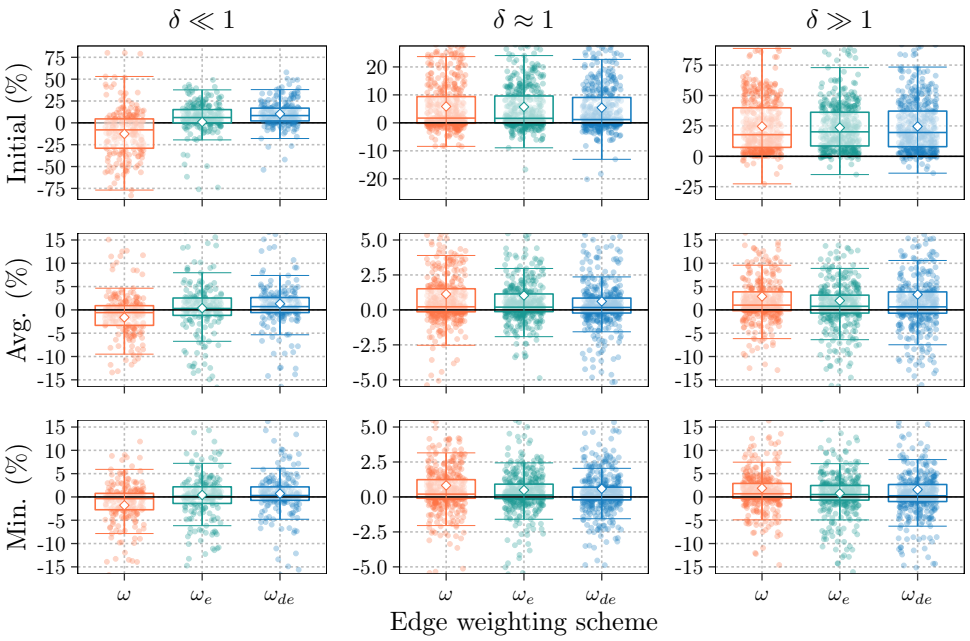


Figure 4.18: Comparing the improvement of kKaHyPar using community-aware coarsening with the different edge weighting schemes presented in Section 4.3.2 over kKaHyPar without community detection. Diamonds show the mean improvement. Experiments are performed on benchmark set B.

Table 4.2: Percentage improvement of kKaHyPar with community-aware coarsening over kKaHyPar without community detection. kKaHyPar uses $\omega(v, e)$ for hypergraphs with medium and high density, and $\omega_{de}(v, e)$ for hypergraphs with low edge density. Experiments are performed on benchmark set B.

Improvement	VLSI		Sparse Matrices		SAT14		
(%)	DAC	ISPD	All	Web/Social[7]	Primal	Literal	Dual
Initial	20.34	13.97	4.05	26.18	23.73	34.20	12.16
Best	3.07	0.79	0.82	4.56	1.62	3.11	0.90
Average	3.70	1.23	1.16	6.55	3.16	4.84	1.50
Worst	3.92	1.77	1.60	8.57	5.50	7.16	1.92

Impact on Solution Quality and Running Time. Table 4.2 as well as Figure 4.19 and Figure 4.20 show the overall effects of community-aware coarsening on

[7]The class 'Web/Social' contains the hypergraphs derived from the following matrices of web graphs and social networks: `cnr-2000`, `NotreDame_actors`, `Stanford`, and `webbase-1M`.

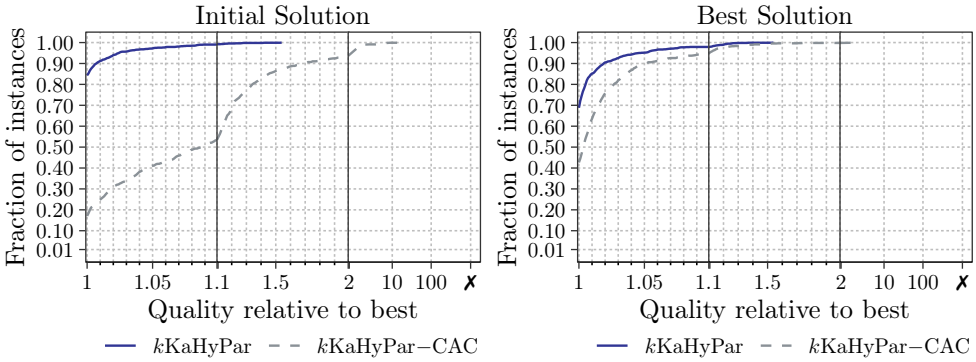


Figure 4.19: Performance profiles for kKaHyPar with and without (−CAC) community-aware coarsening optimizing $f_\lambda(\Pi)$ on benchmark set B.

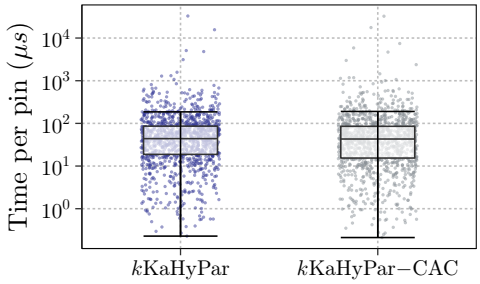


Figure 4.20: Running times of kKaHyPar with and without (−CAC) community-aware coarsening for connectivity optimization on benchmark set B.

kKaHyPar. As can be seen in Table 4.2, using information about the community structure of the hypergraph during coarsening significantly improves the solution quality of the initial k-way partitions on all benchmark sets. The improvements in average and best solution quality indicate that the community-aware configuration is indeed able to compute better solutions than configuration without community detection. Furthermore, the fact that the quality of the worst solutions is also improved shows that community-aware coarsening improves the partitioner's robustness.

Looking at Figure 4.19, we see that the initial solutions of kKaHyPar are better than those of kKaHyPar-CAC (without community-detection) for more than 80% of all instances, and that the solution quality of the final k-way partitions is better for almost 70% of all instances.

Figure 4.20 shows that the running time of kKaHyPar is only slightly larger than that of kKaHyPar−CAC. Finally, we would like to point out the fact that given the similar results presented in the corresponding conference paper [HS17a] for a KaHyPar configuration *without* flow-based refinement, we can conclude that both the flow-based

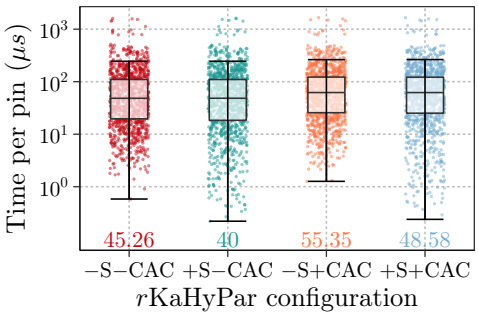


Figure 4.21: Running times of different rKaHyPar configurations with / without community-aware coarsening (+/−CAC) and pin sparsification (+/−S). The geometric mean running time per pin is shown below each box plot. Experiments are performed on benchmark set B.

refinement framework and the community-aware coarsening scheme are *orthogonal* improvements to the multi-level paradigm, as none can compensate for the other.

4.9.3 rKaHyPar with Pin Sparsification and Community Detection

Motivation. After publishing the recursive bipartitioning algorithm for cut-net optimization [Sch+16a], we turned to direct k-way partitioning and connectivity optimization. Thus, improvements such as the pin sparsifier presented in Section 4.3.1 and the community-aware coarsening framework presented in Section 4.3.2 have not yet been considered in the recursive bipartitioning setting. In this section, we therefore evaluate the effects of both techniques on rKaHyPar. The following experiments were performed on benchmark set B, optimizing the cut-net metric $f_c(\Pi)$. Both pin sparsification and community detection algorithms are configured as described in Section 4.8. Sparsification is performed *once* before partitioning, while community detection is done at every bipartitioning step.

Experimental Results. Figure 4.21 and Figure 4.22 summarize the experimental results. Configurations of rKaHyPar with/without pin sparsification are referred to as +/−S, configurations with/without community-aware coarsening are referred to as +/−CAC. Enabling pin sparsification speeds up the algorithm by a factor of 1.13, while community-aware coarsening increases the running time by a factor of 1.22. When enabling both preprocessing techniques, we thus increase the running time only by a factor of 1.07. Similar to the experimental results of kKaHyPar, the performance profiles in Figure 4.22 show that sparsification does not negatively affect solution quality for recursive bipartitioning, while community-aware coarsening slightly improves the quality of the computed partitions. The overall performance difference between rKaHyPar (i.e., configuration +S+CAC) and the old configuration used in Ref. [Sch+16a] (i.e., −S−CAC) is statistically significant ($p = 0.0013$).

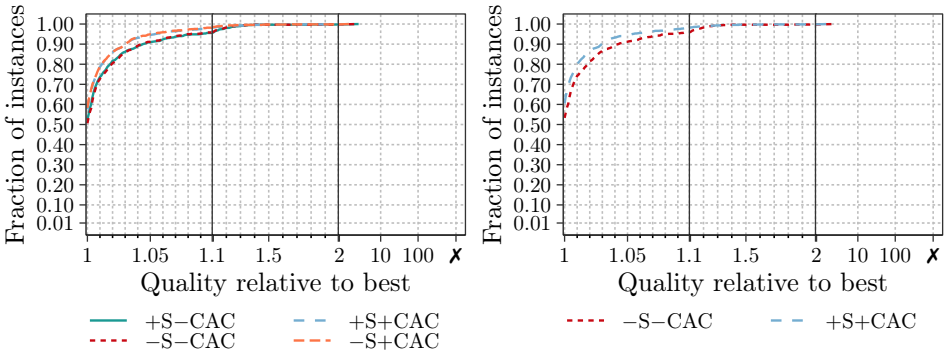


Figure 4.22: Performance profiles of different rKaHyPar configuration with / without community-aware coarsening (+/−CAC) and pin sparsification (+/−S). The plot on the right shows the old configuration (−S−CAC) and the configuration of rKaHyPar (+S+CAC). Experiments are performed on benchmark set B.

It is interesting to note that the improvement is less pronounced for recursive bipartitioning than for direct k-way partitioning. On the one hand, this could be attributed to the fact that preventing "bad" contractions is more important for the latter, because in a direct k-way setting, the initial partitioning algorithm has to compute a k-way partition, whereas in the former it is only necessary to compute 2-way partitions. Thus, several contractions that might be "bad" for k-way partitioning, may not have an equally severe impact on the quality of a 2-way partition. On the other hand, the problems of recursive bipartitioning discussed in Section 3.6.3 might also come into play.

4.9.4 Effects of the Lazy Update Strategy for Coarsening

Motivation. In Section 4.4.1, we proposed a lazy updating strategy for PQ-based n-level coarsening. In this section, we analyze its effect on both running time and solution quality. We use rKaHyPar to optimize the connectivity metric $f_\lambda(\Pi)$ on benchmark set D. Since our focus is on coarsening, the preprocessing techniques (community detection and pin sparsification) are disabled in the experiments.

Experimental Results. Figure 4.23 (left) shows the running times of the coarsening, initial partitioning, and refinement phase of rKaHyPar using both the full and the lazy re-rating strategy for coarsening. Using the lazy re-rating strategy reduces the median running time of the coarsening phase by an *order of magnitude*, while improvements of up to three orders of magnitude are possible for some extreme cases. The running times of the initial partitioning and refinement phase remain unaffected. Looking at solution quality in Figure 4.23 (right), we see that the difference in quality is small. Only for two out of the 175 benchmark instances, the quality difference exceeds a factor of 1.05. A Wilcoxon signed-rank test reveals that the performance difference between

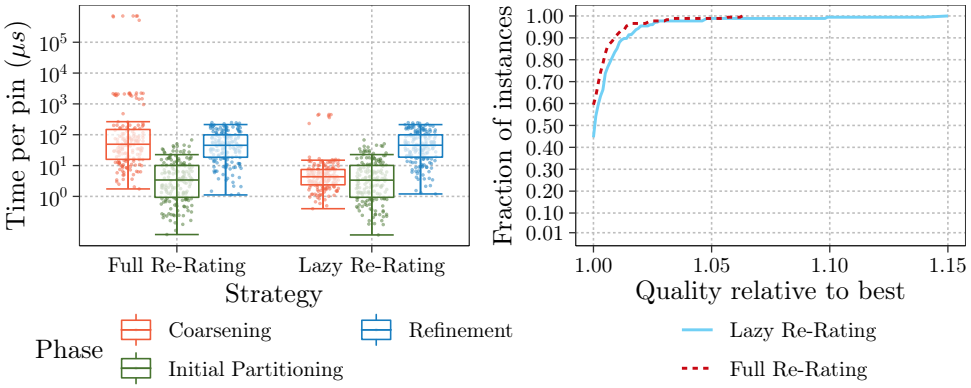


Figure 4.23: Breakdown of the running times of rKaHyPar using either the full or the lazy re-rating strategy (left) and the resulting solution quality (right). Experiments are performed on benchmark set D.

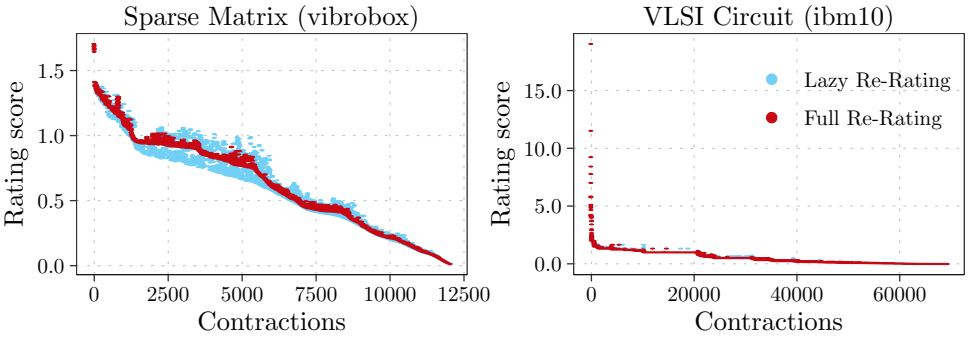


Figure 4.24: Progression of the highest rating scores during one 2-way partitioning run of two hypergraphs using either the full or the lazy re-rating strategy with rKaHyPar.

the full and the lazy re-rating strategy is not statistically significant ($p = 0.0935$). Figure 4.24 visualizes the progression of the highest-rated contractions for both the traditional full re-rating strategy and the lazy approach on two hypergraphs. The visualization is inspired by the work of Alpert et al. [Alp+05]. We see that despite not being identical, the rating scores of the lazy scheme progress in a similar fashion as those of the full re-rating approach. Furthermore, the examples indicate that the hyperedge sizes indeed affect the scores. For the VLSI hypergraph ibm10 (right), which has a median net size of 2, the sequence of rating scores of the lazy strategy deviates only slightly from the sequence of the traditional approach, whereas the deviation is more pronounced for SPM hypergraph vibrobox (left) with a median net size of 24.

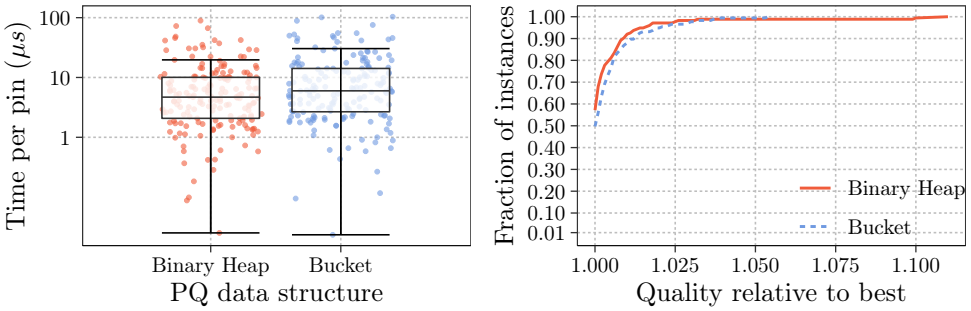


Figure 4.25: Comparing the running times of the *refinement phase* (left) and the solution quality (right) of kKaHyPar using different priority queue data structures. Experiments are performed on benchmark set D.

4.9.5 Effects of Priority Queue Data Structures

Motivation. Historically, many partitioning algorithms used some variation of the bucket priority queue data structure proposed by Fiduccia and Mattheyses [FM82]. However, recent multi-level partitioning algorithms like KaSPar [OS10a] and KaHIP [SS13] instead employ binary heap-based implementations. In order to study the differences in running time and solution quality in the n-level HGP setting, we partitioned the hypergraphs of benchmark set D with kKaHyPar (optimizing the connectivity metric) – using either the LIFO bucket queue data structure described in Section 4.6.5 or our own binary heap implementation. In the following experiments, flow-based refinement is *disabled* because we are only interested in the effects on localized FM local search.

Experimental Results. Figure 4.25 summarizes the results. Looking at the running times of the refinement phase, we see that the binary heap implementation is slightly faster than the bucket-based priority queue. Furthermore, there is no statistically significant difference in solution quality between both approaches ($p = 0.3874$). Only for one out of 175 instances, the performance of the kKaHyPar using binary heaps is worse by a factor of $\tau = 1.104$.

By using the multi-level paradigm and by removing parallel nets, both vertex degrees and hyperedge weights increase during the coarsening phase. This, in turn, increases the range of possible gain values during refinement and results in very few ties in the max-gain bucket, which – similar to what Madden [Mad99] observed for flat partitioning of hypergraphs with varying hyperedge weights — renders LIFO tie-breaking superfluous. Additionally, both the multi-level approach and our technique of only starting local search passes around the just uncontracted vertex pair, already enforce localization. We therefore conclude that the benefits of LIFO bucket queues are outweighed by the n-level (and most likely also the standard multi-level) approach.

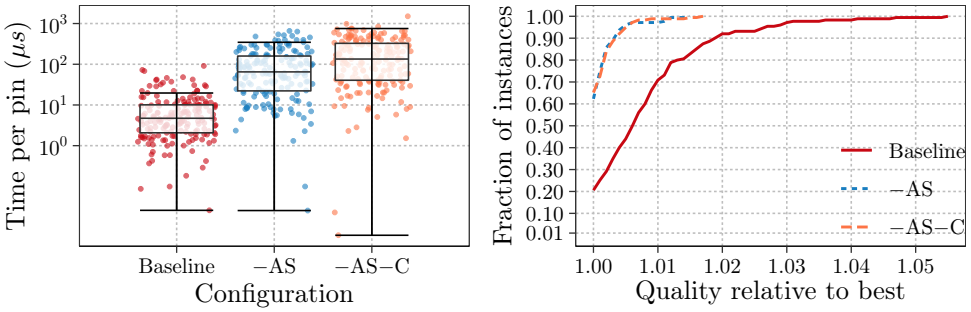


Figure 4.26: Comparing the running times of the *refinement phase* of *k*KaHyPar (baseline) with a configuration that uses the simple stopping rule (−AS), and a configuration that additionally disables gain caching (−AS−C) (left). The corresponding performance profiles are shown in the plot on the right. Experiments are performed on benchmark set D.

4.9.6 Effects of Search Space Restrictions and Caching

Motivation. In the following, we briefly show the effects of gain caching (described in Section 4.6.3) and search space restriction using the stopping rules presented in Section 4.6.4. The experiments are performed on benchmark set D using *k*KaHyPar for connectivity optimization. Flow-based refinement is *disabled* to highlight the running time differences for our localized local search algorithms. When using the simple stopping rule (−AS), the maximum number of fruitless moves is set to $i = 350$, which was the best value in the parameter tuning experiments presented in Ref. [Sch+16a].

Experimental Results. Figure 4.26 (left) shows the running times of the refinement phase for the different configurations. We see that switching from the adaptive to the simple stopping rule (−AS) increases the running time of the refinement phase by more than an order of magnitude. Additionally disabling gain caching (−AS−C) further increases the average running time by roughly a factor of two. As can be seen in the performance profile plot in Figure 4.26 (right), the additional running time spent in the local searches yields better solutions than using the adaptive stopping rule. While this performance difference is statistically significant ($p < 2.23 \times 10^{-16}$), our default *k*KaHyPar configuration uses the adaptive stopping rule in favor of the smaller running times. Figure 4.27 visualizes the local search steps of both the simple and the adaptive stopping rule when partitioning the VLSI hypergraph ibm09 and the primal SAT hypergraph atco_enc2_opt_05_21 into $k = 8$ blocks (i.e., each figure shows all local search steps performed along the entire n-level hierarchy). While the adaptive stopping rule terminates passes early, the simple stopping rule permits much longer search passes in which solution quality degrades far from the quality of the starting partitions. Thus, in total, it performs significantly more moves.

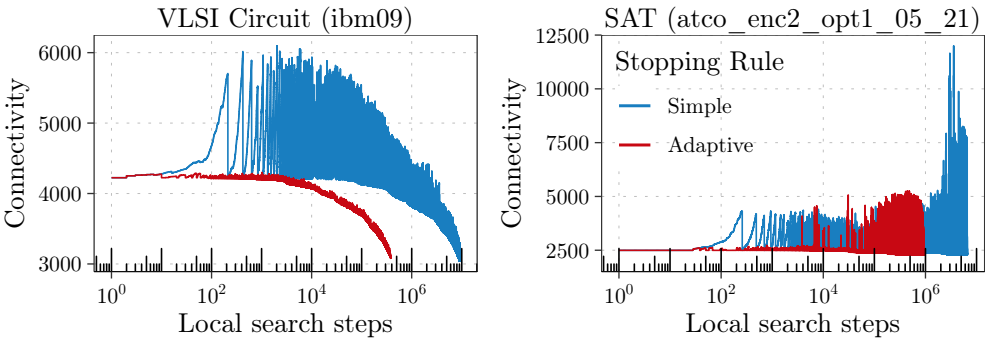


Figure 4.27: Visual comparison of the local search steps performed by both stopping rules for k-way connectivity refinement. Both hypergraphs are partitioned into $k = 8$ blocks. Note the log scale on the x-axis.

Table 4.3: Characteristics of benchmark set B. We use $\overline{x}$ to denote the mean vertex degrees and net sizes, and $\widetilde{x}$ to denote the median vertex degrees and net sizes.

	ISPD98	DAC2012	SuiteSparse	SAT14 Literal	SAT14 Primal	SAT14 Dual
avg. $\overline{d(v)}$	4.2	3.3	25.1	8.2	16.3	2.6
avg. $\widetilde{d(v)}$	3.7	3.2	23.2	4.0	8.3	2.3
avg. $\overline{\lvert e \rvert}$	3.9	3.4	26.8	2.6	2.6	16.3
avg. $\widetilde{\lvert e \rvert}$	2.1	2.0	24.4	2.3	2.3	8.3

4.9.7 Insights into Network Flow-Based Refinement

Motivation. In the following, we examine the different hypergraph flow networks presented in Section 4.7.2 and evaluate their effects on the running times of max-flow algorithms. We show that using our improved flow model $\mathcal{F}_H$ instead of KaFFPa's flow model $\mathcal{F}_G$ yields solutions of higher quality for both graph and hypergraph partitioning. Furthermore, we provide insights into the configuration of the flow-based refinement framework, and demonstrate the effects of the proposed speedup heuristics. In a final evaluation, we then show to what extent using flow-based refinements on top of localized FM local search improves the performance of rKaHyPar and kKaHyPar for both cut-net and connectivity optimization. We would like to note that the evaluation of kKaHyPar for cut-net optimization and the evaluation of rKaHyPar (using both community detection and pin sparsification) for both metrics is new and thus not contained in any of the related publications [HSS18a; HSS18b; HSS19a].

Flow Network Evaluation. To analyze the effects of the different hypergraph flow networks, we compute five bipartitions for each hypergraph of benchmark set B with kKaHyPar (without flow-based refinement) using different seeds. Characteristics of the hypergraphs are shown in Table 4.3. The bipartitions are then used to generate hypergraph flow networks for a corridor of size $|B| = 25\,000$ hypernodes around the cut. Figure 4.28 (top) summarizes the sizes of the respective flow networks in terms of number of nodes $|\mathcal{V}|$ and number of edges $|\mathcal{E}|$ for each instance class. We consider the Lawler network $\mathcal{N}_L$, the Liu-Wong network $\mathcal{N}_W$, and our network $\mathcal{N}_{\text{Our}}$. Furthermore, we show results for $\mathcal{N}^1_{\text{Our}}$ which exploits the fact that the flow problems are based on subhypergraphs H_B by additionally modeling single-pin border nets more efficiently as described in Section 4.7.3. We see that the flow networks of primal and literal SAT instances are the largest in terms of both numbers of nodes and edges. High average vertex degrees combined with low average net sizes lead to subhypergraphs H_B containing many small nets, which then induce many nodes and (infinite-capacity) edges. Dual instances with low average degree and large average net size on the other hand lead to smaller flow networks. For VLSI instances (DAC, ISPD) both average degrees and average net sizes are low, while for SPM hypergraphs the opposite is the case. This explains why SPM flow networks have significantly more edges, despite the number of nodes being comparable in both classes.

As expected, the Lawler network $\mathcal{N}_L$ induces the biggest flow problems. Looking at the Liu-Wong network $\mathcal{N}_W$, we can see that distinguishing between graph edges and nets with $|e| \geq 3$ pins has an effect for all hypergraphs with many small nets (i.e., DAC, ISPD, Primal, Literal). While this technique alone does not improve dual SAT instances, we see that the combination of the Liu-Wong approach and the removal of low degree hypernodes in $\mathcal{N}_{\text{Our}}$, reduces the size of the networks for all instance classes except SPM. Both techniques only have a limited effect on these instances, since both hypernode degrees *and* net sizes are large on average.

Using $\mathcal{N}^1_{\text{Our}}$ further reduces the network sizes significantly. As expected, the reduction in numbers of nodes and edges is most pronounced for instances with low average net sizes because these instances are likely to contain many single-pin border nets.

Flow Algorithm Evaluation. To further see how these reductions in network size translate to improved running times of max-flow algorithms, we use these networks to create flow problems using our flow model $\mathcal{F}_H$ and compute min-cuts using two highly tuned max-flow algorithms, namely the BK-algorithm [BK04] and the incremental breadth-first search (IBFS) algorithm [Gol+11]. These algorithms were chosen because they performed best in the preliminary experiments performed by Tobias Heuer as part of his master thesis [Heu18a]. We compare the speedups of these algorithms when executed on $\mathcal{N}_W$, $\mathcal{N}_{\text{Our}}$, and $\mathcal{N}^1_{\text{Our}}$ to the execution on the Lawler network $\mathcal{N}_L$ in Figure 4.28 (bottom). Both algorithms benefit from improved flow network models on all instance classes except SPM, and the speedups directly correlate with the reductions in network size. SPM hypergraphs have high average vertex degrees and large average net sizes. In this case, the optimizations only have a very limited effect since they target small nets and low degree vertices. While $\mathcal{N}_W$ significantly

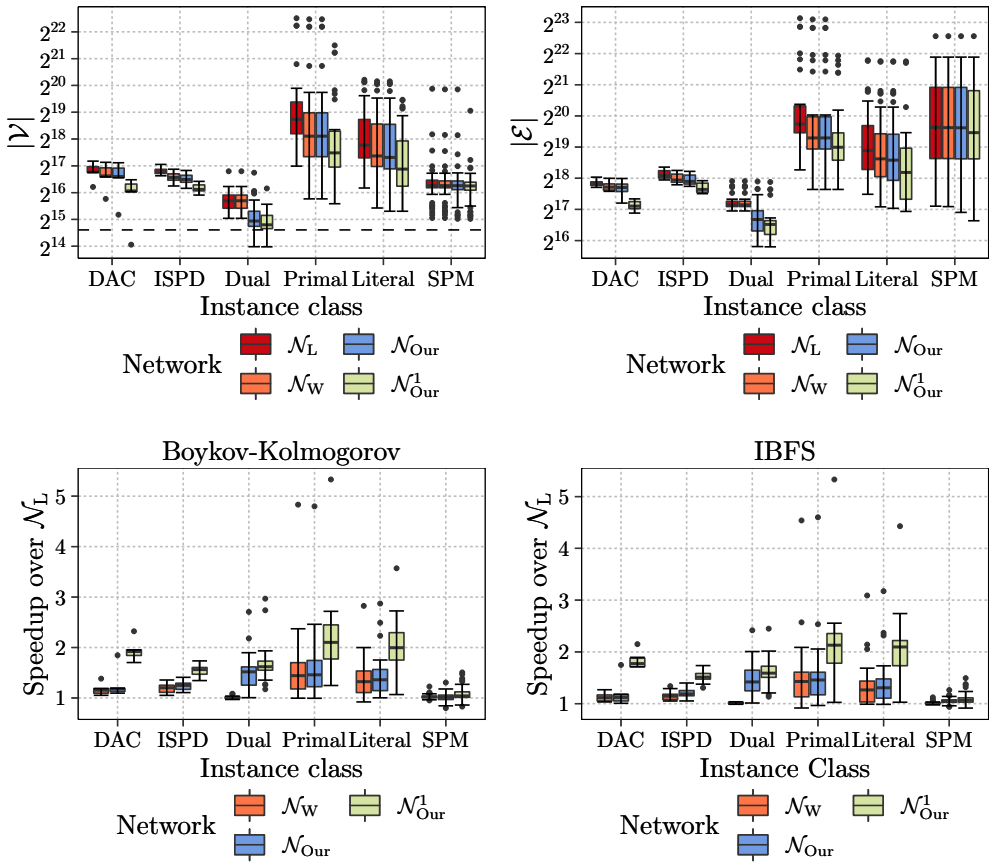


Figure 4.28: Top: Size of the flow networks when using the Lawler network $\mathcal{N}_{\mathrm{L}}$, the Liu-Wong network $\mathcal{N}_{\mathrm{W}}$ and our network $\mathcal{N}_{\mathrm{Our}}$. Network $\mathcal{N}^1_{\mathrm{Our}}$ additionally models single-pin border nets more efficiently. The dashed line indicates 25 000 nodes. Bottom: Speedup of BK [BK04] and IBFS [Gol+11] max-flow algorithms over the execution on the Lawler network $\mathcal{N}_L$. The flow networks are derived from the hypergraphs of benchmark set B (adapted from [HSS19a]).

reduces the running times for Primal and Literal instances, $\mathcal{N}_{\mathrm{Our}}$ additionally leads to a speedup for Dual instances. By additionally considering single-pin border nets, $\mathcal{N}^1_{\mathrm{Our}}$ results in an average speedup between 1.52 and 2.21 (except for SPM instances). Since IBFS performed better than the BK algorithm in the master thesis of Tobias Heuer [Heu18a], the final framework configuration uses $\mathcal{N}^1_{\mathrm{Our}}$ as flow network and the IBFS algorithm for max-flow computations.

Table 4.4: Comparing KaFFPa's flow model $\mathcal{F}_G$ with our model $\mathcal{F}_H$ as described in Section 4.7.3. The table shows the average improvement of $\mathcal{F}_H$ over $\mathcal{F}_G$ (in percent) on benchmark sets D and F.

	Hypergraphs (Set D)			Graphs (Set F)		
α'	$\varepsilon = 1\%$	$\varepsilon = 3\%$	$\varepsilon = 5\%$	$\varepsilon = 1\%$	$\varepsilon = 3\%$	$\varepsilon = 5\%$
1	7.6	8.1	7.7	4.7	4.9	4.8
2	7.5	6.7	4.9	4.6	4.1	3.6
4	6.7	4.0	2.8	4.2	3.2	2.5
8	4.9	2.3	1.5	3.7	2.3	1.9
16	3.2	1.4	1.3	3.0	1.8	1.6

Flow Model Evaluation. We now compare KaFFPa's flow model $\mathcal{F}_G$ to our model $\mathcal{F}_H$ described in Section 4.7.3. The experiments summarized in Table 4.4 were performed using benchmark sets D and F. To focus on the impact of the models on solution quality, we deactivated KaHyPar's FM local search algorithms and only use flow-based refinement without the most balanced minimum cut heuristic. The results confirm our hypothesis that $\mathcal{F}_G$ restricts the space of possible solutions. For *all* flow problem sizes and all imbalances tested, $\mathcal{F}_H$ yields better solution quality. As expected, the effects are most pronounced for small flow problems and small imbalances where many vertices are likely to be border nodes. Since these nodes are locked inside their respective block in $\mathcal{F}_G$, they prevent all non-cut border nets from becoming part of the cut-set. Our model, on the other hand, allows *all* min-cuts that yield a feasible solution for the original partitioning problem. The fact that this effect also occurs for the graphs of set F indicates that our model can also be effective for traditional graph partitioning.[8] Furthermore, we see that the effect is more pronounced for hypergraphs than for graphs, which can be explained by the fact that larger net sizes are likely to induce more border nodes.[9]

Configuring the Algorithm. We now evaluate different configurations of the refinement framework for connectivity optimization on benchmark set D. In the following, kKaHyPar without flow-based refinement is used as a reference and referred to as (−F,−M,+LS), since it neither uses (F)lows nor the (M)ost balanced minimum cut heuristic and only refines partitions using the FM (L)ocal (S)earch algorithm. This basic configuration is then successively extended with specific components. The results of our experiments are summarized in Table 4.5 for increasing scaling parameter α'.[10] In configuration Constant128 all components are enabled (+F,+M,+LS) and

[8] In Ref. [HSS19a], we integrate our improved flow model into KaFFPa and show that it indeed improves the overall performance of the graph partitioning system.

[9] The results of set F differ from the results presented in Ref. [HSS18a], because community detection [HS17a] was disabled in the corresponding experiments. Here, community detection is enabled for both the experiments on set D and on set F.

[10] Preliminary experiments showed that using values larger than 16 for scaling parameter α' did not yield any significant quality improvement that would offset the increased running time.

Table 4.5: Quality and running times for different configurations and increasing α'. Column Avg(%) reports the quality improvement relative to the reference configuration (−F,−M,+LS). Experiments are performed on benchmark set D.

	(+F,−M,−LS)		(+F,+M,−LS)		(+F,−M,+LS)		(+F,+M,+LS)		Constant128	
α'	Avg(%)	t (s)	Avg(%)	t (s)	Avg(%)	t (s)	Avg(%)	t (s)	Avg(%)	t (s)
1	−6.44	9.34	−6.01	9.90	0.29	12.06	0.29	12.32	0.62	49.83
2	−3.64	10.29	−2.53	11.05	0.59	12.66	0.74	13.08	1.18	76.53
4	−2.07	12.15	−0.49	13.09	0.93	13.82	1.22	14.50	1.69	125.34
8	−1.08	15.64	0.78	16.97	1.23	16.06	1.67	17.27	2.20	222.94
16	−0.32	22.52	1.61	24.74	1.58	20.60	2.16	22.65	2.74	428.39
Ref.	(−F,−M,+LS)		6373.88	13.73						

flow-based refinements are performed every 128 uncontractions. It is used as a reference point for the quality achievable using flow-based refinement. In all configurations, the speedup heuristics described in Section 4.7.4 are enabled.

The results indicate that only using flow-based refinement (+F,−M,−LS) is inferior to FM local search in regard to both running time and solution quality. Although the quality improves with increasing flow problem size (i.e., increasing α'), the average connectivity is still worse than the reference configuration. Enabling the most balanced minimum cut heuristic improves partitioning quality, especially as α increases. Configuration (+F,+M,−LS) performs better than the basic configuration for $\alpha' \geq 8$. By combining flows with the FM algorithm (+F,−M,+LS) we get a configuration that improves upon the baseline even for small flow problems. However, comparing this variant with (+F,+M,−LS) for $\alpha' = 16$, we see that using large flow problems together with the most balanced minimum cut heuristic yields solutions of comparable quality. Enabling all components (+F,+M,+LS) and using large flow problems performs best. Furthermore, we see that enabling FM local search slightly improves the running time for $\alpha' = 16$. This can be explained by the fact that the FM algorithm already produces good cuts between the blocks such that fewer rounds of pairwise flow refinements are necessary to improve the solution. Comparing configuration (+F,+M,+LS) with Constant128 shows that performing flows more often further improves solution quality at the cost of slowing down the algorithm by more than an order of magnitude.

Effects of Speedup Heuristics. The effects of the three speedup heuristics presented in Section 4.7.4 are shown in Table 4.6. In the experiments performed on set B (again optimizing connectivity), we start with a kKaHyPar configuration with all heuristics disabled (−S1−S2−S3), and then successively enable the heuristics one by one. Only executing pairwise flow refinements on blocks that lead to an improvement on previous levels (S1) reduces the running time of flow-based refinement by a factor of 1.18, while skipping flows in case of small cuts (S2) results in a further speedup of 1.14. By additionally stopping the resizing of the flow problem as early as possible (S3), we decrease the running time of flow-based improvement by a factor of 1.71 in total, while still computing solutions of comparable quality, which is why the framework is

Table 4.6: Comparison of solution quality (average and minimum $f_\lambda(\Pi)$) and running time of kKaHyPar using the speedup heuristics. Column $t_{\text{refinement}}$ (s) refers to the running time of the refinement phase, column t_{total} (s) to the total partitioning time. Experiments are performed on benchmark set B.

Configuration	Avg.	Min.	$t_{\text{refinement}}$ (s)	t_{total} (s)
−S1−S2−S3	6858.9	6640.1	41.6	69.8
+S1−S2−S3	6863.3	6643.9	35.1	60.8
+S1+S2−S3	6864.2	6646.3	30.7	55.2
+S1+S2+S3	6864.5	6645.0	24.3	47.2

configured to employ all heuristics by default.

Effects on Solution Quality and Running Time. Having configured the flow-based refinement framework, we now evaluate its effects on rKaHyPar and kKaHyPar for both cut-net and connectivity optimization. In both cases, the −F variants *only* differ from the reference configurations described in Section 4.8 in that the flow-based refinement framework is disabled.

Figure 4.29 summarizes the experiments performed on benchmark set B. For both metrics, both kKaHyPar and rKaHyPar significantly outperform their non flow-based counterparts. kKaHyPar computed the best solutions for 97.5% of all instances for cut-net optimization and for 97.6% of all instances for connectivity optimization. Furthermore, its solutions are at most a factor of 1.034 worse than the solutions of kKaHyPar−F for cut-net optimization, and at most a factor of 1.026 for connectivity optimization. For rKaHyPar, the effect of flow-based refinements is slightly less pronounced. When optimizing the cut-net metric, it computes the best solutions for 72.1% of all instances, when optimizing connectivity its solutions are better on 72.5% of all instances.

Looking at Table 4.7, we see that whereas kKaHyPar is more than 2% better on average than kKaHyPar−F for both objectives, the solutions of rKaHyPar are only around 0.6% better on average than those of rKaHyPar−F. A Wilcoxon signed rank test however reveals that the difference in solution quality is statistically significant in *all* cases ($p < 2.22 \times 10^{-16}$). The smaller overall improvements for rKaHyPar can be partially explained by some of the problems of the RB-approach discussed in Section 3.6.3: In recursive bipartitioning-based algorithms, a good solution for the first bipartition divides the instance into two densely connected blocks and thus makes it more difficult for further 2-way partitions to find small cuts. Thus, improved solutions on the first recursion levels do not necessarily translate to overall improved k-way partitions [ST97]. Furthermore, once a net is cut for the first time in one of the bipartitions, it will *remain* a cut net for the rest of the partitioning process, since it is not possible to remove it from the cut-set in the following bipartitions.

Comparing the running times of configurations using the flow-based refinement framework with those that only use localized FM local search, we see that flows increase

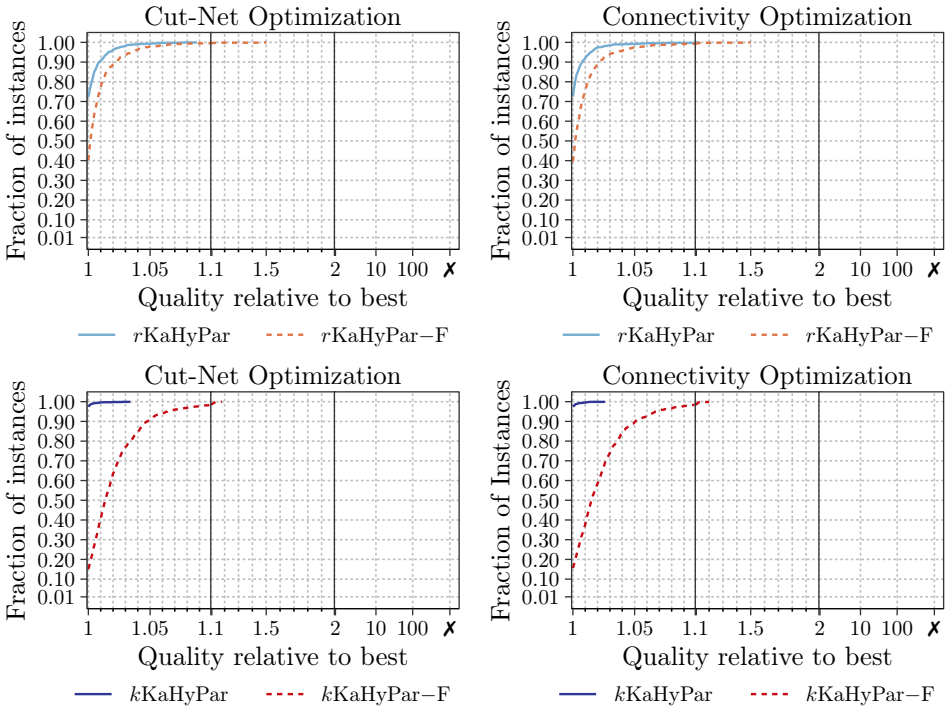


Figure 4.29: Performance profiles of rKaHyPar and kKaHyPar with and without (−F) flow-based refinement optimizing $f_c(\Pi)$ and $f_\lambda(\Pi)$ on benchmark set B.

Table 4.7: Quality improvements (average and minimum $f_\lambda(\Pi)$ and $f_c(\Pi)$) and running times of kKaHyPar and rKaHyPar compared to corresponding configurations without flow-based refinements (−F) on benchmark set B.

Objective	Algorithm	Avg.	Min.	t (s)
$f_\lambda(\Pi)$	kKaHyPar−F	7030.9	6786.7	24.9
	kKaHyPar	2.4 %	2.1 %	47.2
	rKaHyPar−F	7018.8	6810.3	61.2
	rKaHyPar	0.7 %	0.6 %	68.5
$f_c(\Pi)$	kKaHyPar−F	6130.0	5908.5	28.6
	kKaHyPar	2.3 %	2.0 %	57.2
	rKaHyPar−F	6085.4	5898.8	54.3
	rKaHyPar	0.6 %	0.5 %	60.9

the running time of kKaHyPar roughly by a factor of two, whereas the running time of rKaHyPar only increases by a factor of around 1.2. This can be explained by the fact that while flow-based refinement works on all pairs of blocks adjacent in the quotient graph in the direct k-way setting, only two blocks are refined during bipartitioning.

4.9.8 Effectiveness Tests

Motivation. The community-aware coarsening scheme presented in Section 4.3.2 and the flow-based refinement framework presented in Section 4.7 improve solution quality at the cost of an increased running time. In order to evaluate the relative importance of these two algorithmic components, we perform additional *effectiveness tests* on benchmark set B for both cut-net and connectivity optimization. In these tests, we start with the rKaHyPar and kKaHyPar configurations described in Section 4.8 and successively remove flow-based refinement and the community-aware coarsening scheme – yielding successively weaker variants of the respective configurations. In the figures referenced in this section, we use rKaHyPar resp. kKaHyPar to denote the strongest configurations. The removal of the flow-based refinement framework is indicated with suffix −F, the *additional* removal of community-aware coarsening is indicated with suffix −F−CAC.

Virtual Instances. Since weaker configurations run faster, we create a setting in which each configuration has approximately the same amount of time to compute a k-way partition. More precisely, we use the concept of *virtual instances* introduced by Akhremtsev, Sanders, and Schulz [ASS17] to allow the faster configuration to perform additional repetitions. Given the results of r repetitions of two algorithm configurations A and B for one instance I, i.e., a k-way partition of a hypergraph H, a virtual instance is computed as follows: First, we choose one repetition of both algorithms uniformly at random. Let t_1^A and t_1^B be the running times of configuration A and configuration B for that particular repetition, and assume without loss of generality that $t_1^A \geq t_1^B$. We now sample additional repetitions for algorithm B (without replacement) until the total running time of all sampled repetitions exceeds t_1^A, i.e., if the last sample t_ℓ^B of algorithm B would exceed t_1^A, it is accepted with probability $p = (t_1^A - \sum_{i \leq i < \ell} t_i^B)/t_\ell^B$. It has been shown that using this approach, the expected running time of the sampled repetitions of configuration B is the same as the running time of a single repetition of configuration A [Akh19, Thm. 4.1]. The solution quality of configuration A then corresponds to the quality of the single repetition, while the quality of configuration B is the best result of all sampled repetitions. For each of the 1148 actual instances, we perform 10 repetitions per configuration. Similar to Ref. [Akh19], these data are then used to create 100 virtual instances for each of the 1148 actual instances – resulting in a total of 114 800 virtual instances per configuration.[11]

[11] Only for very few instances, slightly more than 10 repetitions would have been needed for the faster configuration. In these cases, we restrict the results of the faster configuration to the *best* of the 10 repetitions, since the impact of those instances on the overall result is considered negligible.

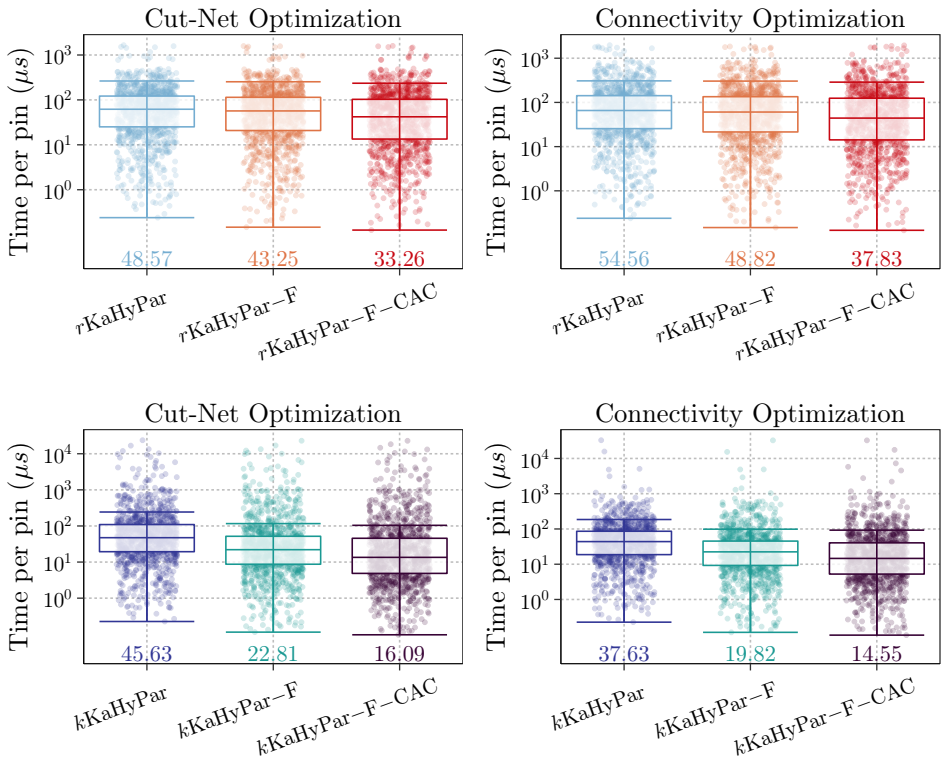


Figure 4.30: Running times of rKaHyPar and kKaHyPar configurations for effectiveness tests. The geometric mean running times per pin are shown below each box plot. Experiments are performed on benchmark set B.

***r*KaHyPar.** Looking at the results of the effectiveness tests on virtual instances in Figure 4.31, we see that using flow-based refinement pays off for both cut-net and connectivity optimization, as weaker configurations, i.e., rKaHyPar−F (without flows) and rKaHyPar−F−CAC (without flows, without community-aware coarsening) do not perform better when given the same amount of time. However, in a configuration without flows (i.e., rKaHyPar−F) the benefits of community-detection can be offset for cut-net optimization by performing more runs using the weakest configuration (i.e., rKaHyPar−F−CAC). For connectivity optimization, both configurations produce comparable results given the same amount of time. However, looking at the running times of all configurations in Figure 4.30 (top) and the performance profiles of the *best* solutions on the *actual* instances in Figure 4.33, we see that the larger running times of more advanced configurations translate into solutions of higher quality, if every algorithm is executed the same number of times.

***k*KaHyPar.** The results of the effectiveness tests on virtual instances for *k*KaHyPar are shown in Figure 4.32. In contrast to recursive bipartitioning, advanced configurations are always more effective than weaker configurations. Neither *k*KaHyPar−F nor *k*KaHyPar−F−CAC is able to outperform the respective stronger configuration if given the same amount of time – both for cut-net as well as connectivity optimization – even though the running time difference between the configurations is more pronounced than for *r*KaHyPar (see Figure 4.30 bottom). These results substantiate the significant performance differences shown in the performance profiles of the *best* solutions on the *actual* instances in Figure 4.34.

> The results of the effectiveness tests for recursive bipartitioning reflect the results presented in Section 4.9.3 (where we first looked at the effects of pin sparsification and community detection on a *r*KaHyPar configuration that used flow-based refinements) and Section 4.9.7 (where we evaluated the flow-based refinement framework for *r*KaHyPar). The observed effects could thus be yet another indicator that the problems of recursive bipartitioning discussed in Section 3.6.3 indeed are relevant in practice. However, this requires a more in-depth experimental analysis.

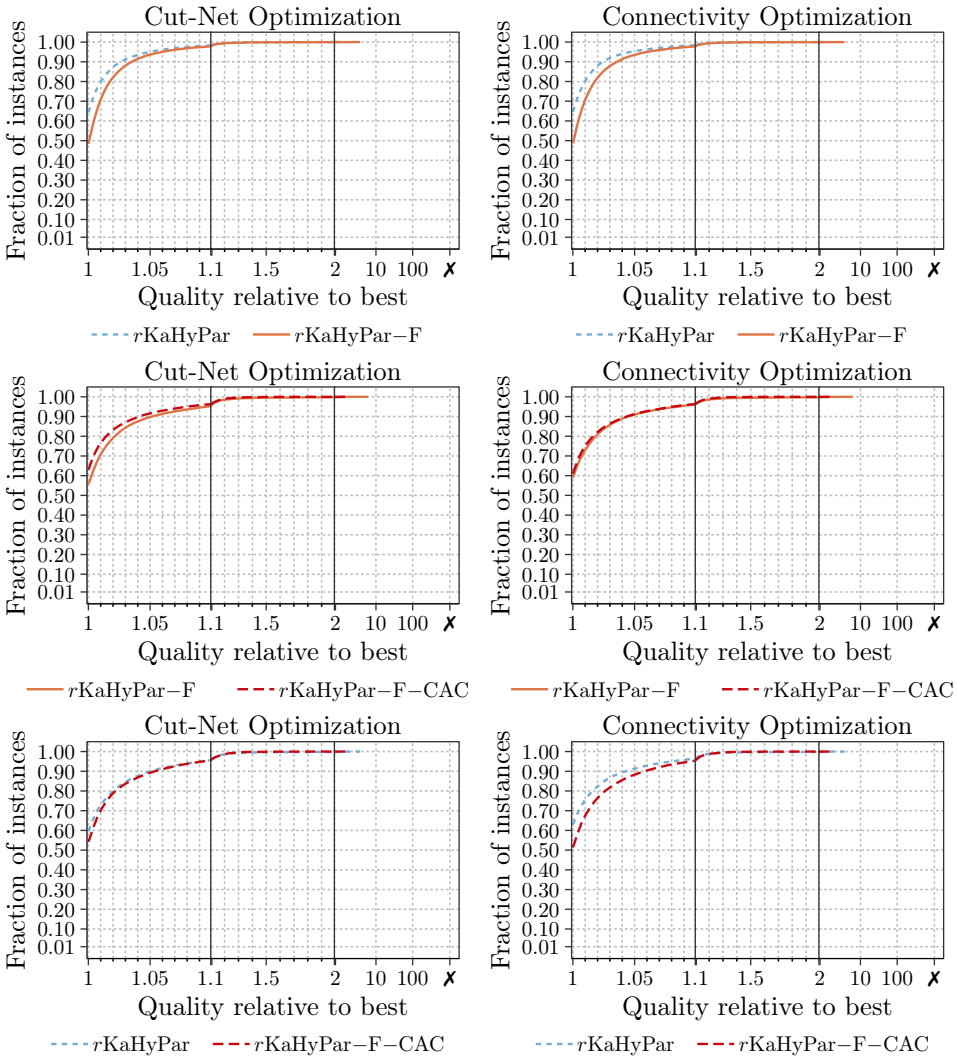


Figure 4.31: Effectiveness tests on benchmark set B for different rKaHyPar configurations using *virtual instances*. Legend: rKaHyPar (baseline), rKaHyPar−F (without flow-based refinement), rKaHyPar−F−CAC (without flow-based refinement & without community-aware coarsening).

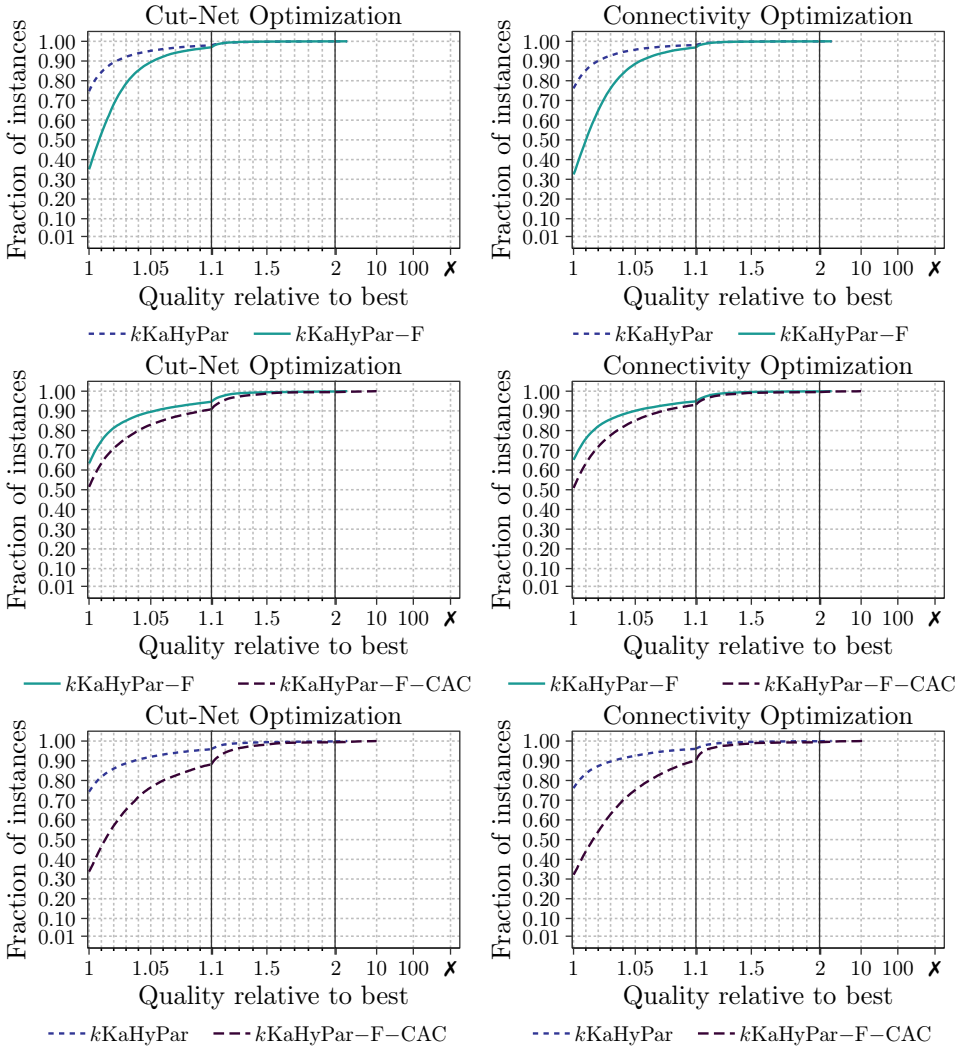


Figure 4.32: Effectiveness tests on benchmark set B for different kKaHyPar configurations using *virtual instances*. Legend: kKaHyPar (baseline), kKaHyPar−F (without flow-based refinement), kKaHyPar−F−CAC (without flow-based refinement & without community-aware coarsening).

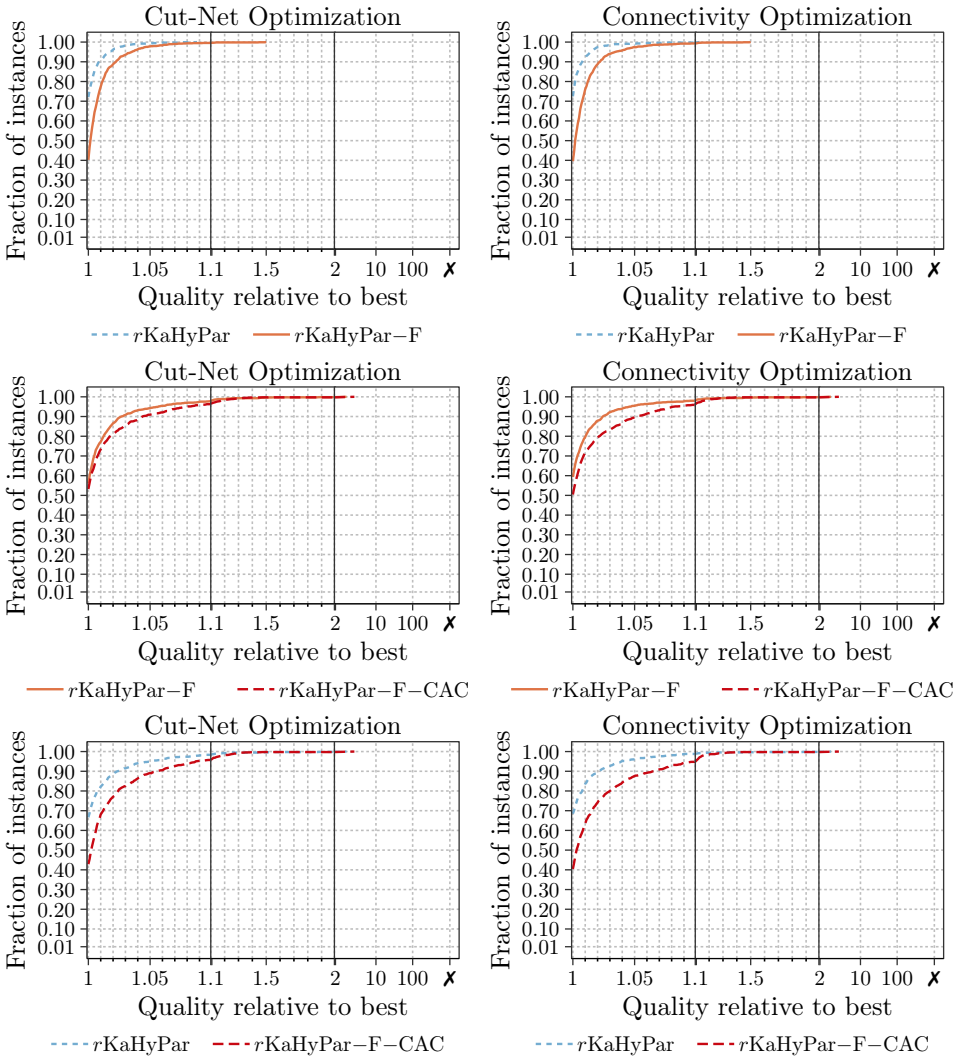


Figure 4.33: Performance profiles of different rKaHyPar configurations using *actual* instances. Legend: rKaHyPar (baseline), rKaHyPar−F (without flow-based refinement), rKaHyPar−F−CAC (without flow-based refinement & without community-aware coarsening). Experiments are performed on benchmark set B.

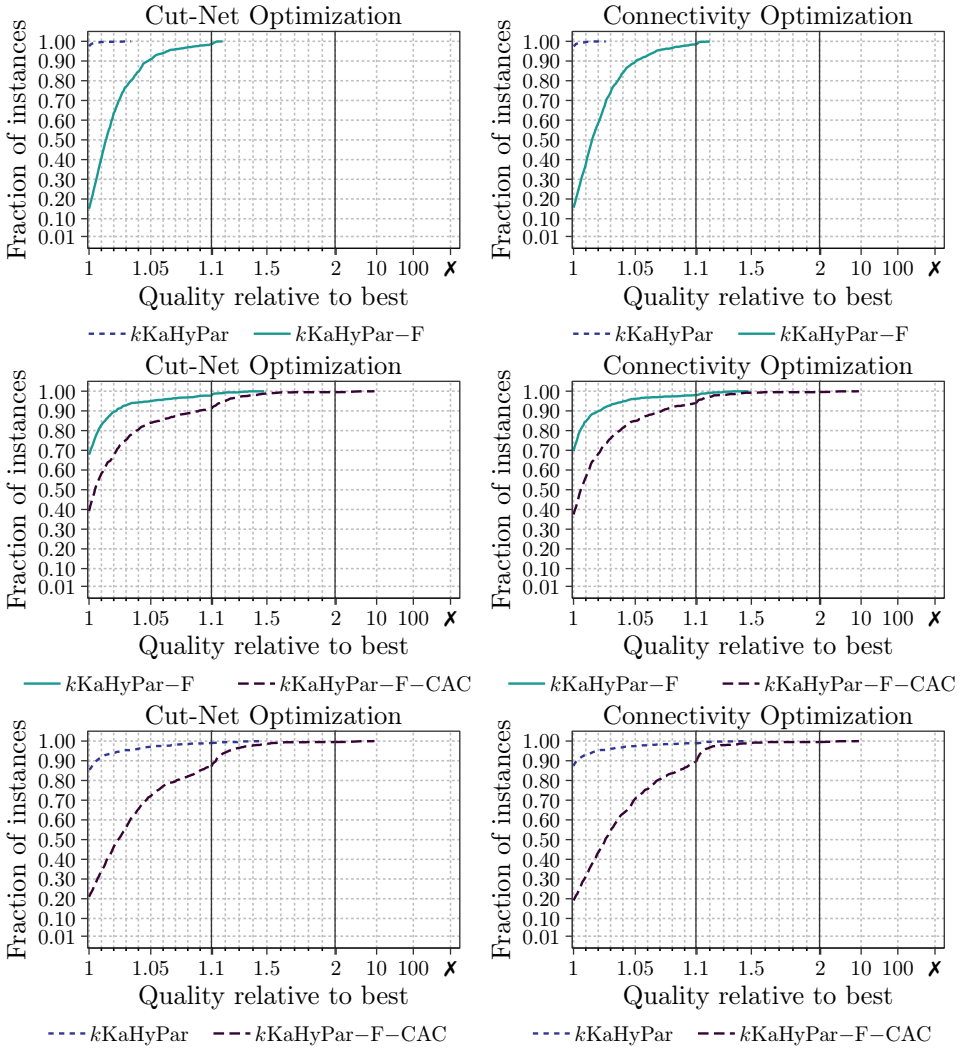


Figure 4.34: Performance profiles of different *k*KaHyPar configurations using *actual* instances. Legend: *k*KaHyPar (baseline), *k*KaHyPar−F (without flow-based refinement), *k*KaHyPar−F−CAC (without flow-based refinement & without community-aware coarsening). Experiments are performed on benchmark set B.

4.10 Concluding Remarks

In this chapter, we presented the different algorithmic components of our hypergraph partitioning framework KaHyPar in a consistent manner by combining and extending the work presented in four conference publications [Sch+16a; Akh+17a; HS17a; HSS18a] and one journal paper [HSS19a]. The central idea underlying KaHyPar is to explicitly evade the trade-off between solution quality and running time inherent in the number of hierarchy levels created by multi-level heuristics. Instead of using an approximately logarithmic number of levels like traditional algorithms, we go to the extreme case of (nearly) n levels. We presented a hypergraph data structure and two coarsening algorithms that are specifically engineered to fit this approach. To speed up partitioning in the presence of many large hyperedges, we proposed an efficient pin sparsifier based on locality sensitive hashing. Furthermore, we showed that traditional coarsening algorithms lack a global view of the hypergraph. Since they are solely guided by local, greedy decisions, they are prone to perform contractions that obscure naturally existing structure. We therefore presented an approach which incorporates global information about the community structure into the coarsening process. Community detection is performed via modularity maximization using the Louvain algorithm on the bipartite graph representation of the hypergraph.

After describing our portfolio-based approach to initial partitioning, we then turned to the refinement phase and presented three local improvement schemes. The first two algorithms are based on the Fiduccia-Mattheyses (FM) heuristic [FM82], but perform a *highly localized* search that starts with only two vertices and then gradually expands by successively considering their neighbors. One algorithm is specifically tailored to improving two-way partitions (i.e., partitions consisting of $k = 2$ blocks) and can therefore be used in a setting where k-way partitions are computed via recursive bipartitioning. The other algorithm has a more global view and is able to directly improve k-way partitions by moving vertices between *all* k blocks. The third algorithm generalizes and improves the *flow-based refinement* framework of the KaFFPa [SS11] and uses max-flow computations on pairs of blocks to refine k-way partitions.

The experimental evaluation presented in this chapter deliberately omitted comparisons to other partitioning systems in order to focus on the framework itself. We demonstrated the effects of different key techniques such as pin sparsification, community-aware coarsening, and flow-based refinement. Furthermore, we addressed open questions that were not (or only very briefly) considered in any of the publications such as the effects of sparsification and community detection in a recursive bipartitioning setting, the effectiveness of the lazy re-rating strategy for PQ-based n-level coarsening, as well as the effects of priority queue data structures, caching and search space restrictions on localized FM-based local search algorithms.

The resulting configurations for recursive bipartitioning (rKaHyPar) and direct k-way partitioning (kKaHyPar) are specifically tailored to high-quality partitioning and engineered to overcome the bottlenecks of straightforward n-level partitioning algorithms that would be adequate for graph partitioning (i.e., KaSPar [OS10a; Osi14]).

Chapter 5

5 Memetic n-Level Hypergraph Partitioning

"If superior creatures from space ever visit earth, the first question they will ask, in order to assess the level of our civilization, is: 'Have they discovered evolution yet?'"

— Richard Dawkins, The Selfish Gene

Motivation. The intuition behind the multi-level approach is that a good partition at one level of the hierarchy will also be a good partition on the next finer level. Hence, depending on the definition of the neighborhood, local search algorithms are able to explore local solution spaces very effectively in this setting. However, they are also prone to get stuck in local optima [KK99]. The multi-level paradigm helps to some extent, since local search has a more global view of the problem at the coarse levels and a very fine-grained view at the fine levels of the multi-level hierarchy. In addition, as with many other metaheuristics, multi-level HGP gives better results if several repeated runs are made with some measures taken to diversify the search.

Still, even a large number of repeated executions can only scratch the surface of the huge space of possible partitionings. In order to explore the global solution space extensively, more sophisticated metaheuristics are needed. Several genetic and memetic hypergraph partitioning algorithms have already been proposed in the literature [BM94; Are00a; AY04; KKM04; Arm+10]. However, *none of them* is considered to be truly competitive with state-of-the-art tools [CKL03]. We believe that this is due to the fact that all of them employ *flat* (i.e., non multi-level) partitioning algorithms to drive the exploitation of the local solution space. In this chapter, we therefore integrate the n-level hypergraph partitioning framework presented in the previous chapter with a genetic algorithm and thus develop the first multi-level memetic algorithm for the hypergraph partitioning problem.

References. This chapter is based on a technical report [ASS18b] and a conference paper jointly published with Robin Andre and Christian Schulz [ASS18a]. The paper was mainly written by the author of this dissertation, with editing by Christian Schulz. Large parts of this chapter were copied verbatim from the paper. The initial implementation of the evolutionary framework was done by Robin Andre as part of his bachelor thesis [And17a], which was supervised by us. This implementation was then improved and integrated into the KaHyPar framework by the author of this dissertation. The experimental evaluation presented in Section 5.4 contains some experimental results of Ref. [HSS19a].

5.1 Overview

We start by explaining the components of our memetic n-level hypergraph partitioning algorithm. Given a hypergraph H and a time limit t, the algorithm starts by creating an initial population of $\mathcal{P}$ *individuals*, which correspond to ε-balanced k-way partitions of H. The population size $|\mathcal{P}|$ is determined dynamically by first measuring the time t_{I} spent to create one individual. Then, $\mathcal{P}$ is chosen such that the time to create $|\mathcal{P}|$ individuals is a certain percentage η of the total running time t: $|\mathcal{P}| := \max(3, \min(50, \eta \cdot (t/t_I)))$, where η is a tuning parameter. The lower bound on the population size is chosen to ensure a certain minimum of diversity, while the upper bound is used to ensure convergence. In contrast to previous approaches [Hul90; BM94; Are00a; KKM04; Arm+10], the population is not filled with randomly generated individuals, but *high-quality* solutions computed by KaHyPar.

To judge the *fitness* of an individual we use the connectivity $\mathfrak{f}_\lambda(\Pi)$ or the cut-net metric $\mathfrak{f}_c(\Pi)$ of its partition Π. The initial population is evolved over several generational cycles using the *steady-state* paradigm [De 06], i.e., only *one* offspring is created per generation. The two-point and multi-point recombination operators described in Section 5.2 improve the average quality of the population by effectively combining different solutions to the HGP problem.

In order to sufficiently explore the global search space and to prevent premature convergence, it is important to keep the population diverse [Bäc96]. This becomes even more relevant in our case, since with KaHyPar we use powerful heuristics to exploit the local solution space. Previous work on evolutionary algorithms for HGP [Hul90; BM94; AY04; KKM04; Arm+10] used simple mutations that change the block of each vertex uniformly at random with a small probability. In contrast to these simple, problem agnostic operators, Section 5.3 presents mutation operators based on V-cycles [Wal04] that exploit knowledge of the problem domain and create offspring solutions in the *vicinity* of the current population. Furthermore, in Section 5.3 we propose a replacement strategy which considers fitness *and* similarity to determine the individual to be evicted from the population.

5.2 Recombination Operators

The evolutionary algorithms for HGP presented in Chapter 3 use simple multi-point crossover operators which split the parent partitions into several parts and then combine these parts to form new offspring (see Figure 5.1 (a)). Since these operators do not take the structure of the hypergraph into account, offspring solutions may have considerably worse fitness than their parents. By generalizing the recombine operator framework of KaFFPaE [SS12] from graphs to hypergraphs, the two-point recombine operator described in this section assures that the fitness of the offspring is *at least as good as the best of both parents*. The edge frequency-based multi-point recombination operator described afterwards gives up this property, but still generates good offspring.

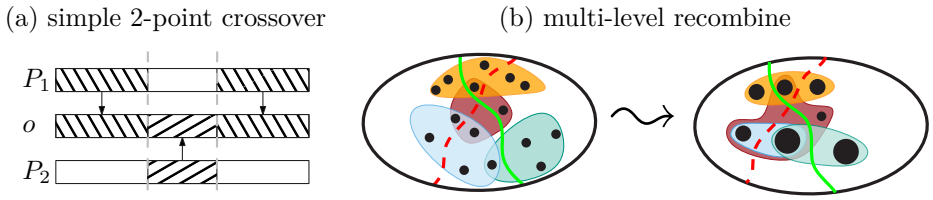


Figure 5.1: (a) Traditional, problem agnostic crossover operation to combine parent partitions P_1 and P_2 to offspring o. (b) Recombination using modified multi-level coarsening to combine two partitions (dashed red line and solid green line). Each cut net e remains in the coarse hypergraph and maintains its connectivity $\lambda(e)$ regarding both partitions (source: [ASS18a]).

Two-Point Recombination. The operator starts with selecting parents for recombination using binary tournament selection (without replacement) [BT96]. Two individuals I_1 and I_2 are chosen uniformly at random from $\mathcal{P}$ and the individual with better fitness (i.e., lower $f_c(\Pi)/f_\lambda(\Pi)$ objective) becomes the first parent P_1. This process is then repeated to determine the second parent P_2. A tournament size of two is chosen to keep the selection pressure low and to avoid premature convergence, since all individuals already constitute high-quality solutions. Both individuals/partitions are then used as input of a modified n-level partitioning scheme as follows:

During coarsening, two vertices u and v are only allowed to be contracted if *both parents agree on the block assignment of both vertices*, i.e., if $b_1[u] = b_1[v] \wedge b_2[u] = b_2[v]$. This is a generalization from multi-level evolutionary GP (i.e., the work of Sanders and Schulz [SS12]), where *edges running between two blocks are not eligible* for contraction and therefore remain in the graph. In other words, our generalization allows two vertices of the same cut net to be contracted as long as the input individuals agree that they belong to the same block. For HGP, this restriction ensures that cut nets e remain in the coarsened hypergraph *and* maintain their connectivity $\lambda(e)$ regarding *both* partitions. This modification is important for the optimization objective, because it allows us to use the partition of the *better* parent as initial partition of the offspring. Since we can skip the initial partitioning phase and therefore do not need a sufficiently large number of vertices in the coarsest hypergraph to compute a good initial partition [KK99], we alter the stopping criterion of the coarsening phase such that it stops when no more contractions are possible. Apart from altering the contraction mechanism and the stopping criterion no modifications of the coarsening algorithms are performed.

The high-quality solution of the coarsest hypergraph contains two different classes of vertices: Those for which both parent partitions agree on a block assignment and those for which they do not (see Figure 5.1 (b) for an example). During the uncoarsening phase, refinement algorithms can then use this initial partitioning to (i) exchange good parts of the solution on the coarse levels by moving few vertices and (ii) to find the

best block assignment for those vertices, for which the parent partitions disagreed. Since KaHyPar's refinement algorithms guarantee non-decreasing solution quality, the fitness of offspring solutions generated using this kind of recombination is always *at least as good as the better of both parents.*

Edge-Frequency Multi-Recombination. The operator described previously is restricted to recombine $p = 2$ partitions to improved offspring of non-decreasing quality. Sanders and Schulz [SS12] specifically restrict their operators to this case, arguing that in the course of the algorithm a series of two-point recombine operations to some extent emulates a multi-point recombination. Here, we present a multi-point recombine operation to partially evaluate this hypothesis in the experimental evaluation. The recombine operator uses the concept of (hyper)edge frequency [WA98] to pass information about the cut nets of the t *best individuals* in the population on to new offspring. The frequency $f(e)$ of a net e hereby refers to the number of times it appears in the cut in the t best solutions: $f(e) := |\{I \in t \mid \lambda(e) > 1\}|$. We use $t = \lceil \sqrt{|\mathcal{P}|} \rceil$, which is a common value in evolutionary algorithms [Del+11]. The multi-recombine operator then uses this information to create a *new* individual in the following way. The coarsening algorithm is modified to prefer to contract vertex pairs (u, v) which share a large number of small, low-frequency nets. This is achieved by replacing the standard heavy-edge rating function of KaHyPar with the rating function [WA98] shown in Eq. 5.1:

$$r(u,v) := \frac{1}{c(v) \cdot c(u)} \sum_{e \in \{\mathrm{I}(v) \cap \mathrm{I}(u)\}} \frac{\exp(-\zeta f(e))}{|e|}. \tag{5.1}$$

This rating function disfavors the contraction of vertex pairs incident to cut nets with high frequency, because these nets are likely to appear in the cut of high-quality solutions. The tuning parameter ζ is used as a damping factor. After coarsening stops, KaHyPar's initial partitioning algorithms are used to compute an initial partition of the coarsest hypergraph, which is then refined during the uncoarsening and local search phase.

5.3 Mutation Operations and Diversification

Mutation Operators. We define two mutation operators based on V-cycles [Wal04]. Both operators are applied to a random individual I of the population. The V-cycle technique reuses an already computed partition as input for the n-level approach and iterates coarsening and local search phases several times using different seeds for randomization. This approach has been applied successfully as mutation operator in evolutionary GP [SS12], therefore we also adopt it for HGP. During coarsening, the quality of the solution is maintained by only contracting vertex pairs (u, v) belonging to the same block (i.e., $b[u] = b[v]$). By distinguishing two possibilities for initial partitioning, we define two different mutation operators: The first one uses the current partition of the individual as initial partition of the coarsest hypergraph and guarantees

non-decreasing solution quality. The second one employs KaHyPar's portfolio of initial partitioning algorithms to compute a *new* solution for the coarsest hypergraph. During uncoarsening, local search algorithms improve the solution quality and thereby further mutate the individual. Since the second operator computes a new initial partition which might be different from the original partition of I, the fitness of offspring generated by this operator can be worse than the fitness of I.

Replacement Strategy. All recombination and mutation operators create one new offspring o. In order to keep the population diverse, we evict the individual *most similar* to the offspring among all individuals whose fitness is equal to or worse than o. Previous work on bipartitioning [BM94; KKM04] used the Hamming distance as a metric to measure the similarity between partitions. We propose a more sophisticated similarity measure that takes into account the connectivity $\lambda(e)$ of each cut net e. For each individual, we compute the multi-set $D := \{(e, m(e)) : e \in E\}$, where $m(e) := \lambda(e) - 1$ is the multiplicity (i.e., number of occurrences) of e. Thus, each cut net e is represented $\lambda(e) - 1$ times in D. The difference of two individuals I_1 and I_2 is then computed as $d(I_1, I_2) := |D_1 \ominus D_2|$, where $\ominus$ is the symmetric difference.

5.4 Experimental Evaluation

Motivation. The purpose of the experimental evaluation presented in this section is twofold. We first evaluate the impact of the different algorithmic components of our memetic algorithm and compare the best configurations with the repeated execution of two KaHyPar configurations. For historic reasons (i.e., the memetic algorithm [ASS18a] was published and developed concurrently to the flow-based refinement framework [HSS18a]), these experiments were done without the use of flow-based refinements. In a second evaluation, we therefore analyze the effects of additionally using the flow-based refinement framework in the memetic algorithm in order to exploit the local solution space. The experiments presented in this section are a combination of results from Ref. [ASS18a] and Ref. [HSS19a].

Setup and Methodology. The algorithmic components of the memetic algorithm are evaluated on the 25 hypergraphs of benchmark set D, while the comparison with repeated KaHyPar executions and the additional experiments with flow-based refinement are done using the 100 hypergraphs of benchmark set C. In all experiments, we optimize the connectivity metric $\mathfrak{f}_\lambda(\Pi)$. For experiments on benchmark set D, each configuration is given *two* hours time *per* test instance to compute one solution, and we perform *five* repetitions with different seeds for each test instance and algorithm configuration. Furthermore, we restrict these experiments to partitioning into $k = 32$ blocks with an imbalance of $\varepsilon = 0.03$. For experiments on benchmark set C, we partition the hypergraphs into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks, and each algorithm is given *eight* hours time *per* test instance. We again perform *five* repetitions with different seeds for each test instance and algorithm. Due to the large amount of computing time necessary to perform these experiments, we always partition 16

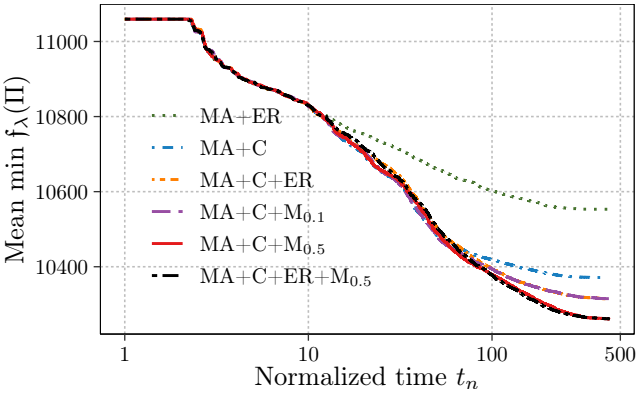


Figure 5.2: Influence of the algorithmic components of our memetic algorithm. Experiments are performed on benchmark set D with $k = 32$.

instances in parallel on a single compute node.[1]

Influence of Algorithmic Components. All memetic configurations are based on kKaHyPar without flow-based refinement and determine their population size $\mathcal{P}$ dynamically such that $\eta = 15\%$ of the total time is spent to create the initial population. According to the results of Wichlund and Aas [WA98], the damping factor ζ used for edge frequency calculations is set to $\zeta = 0.5$ (see Eq. 5.1). We use a naming scheme to refer to different configurations of the memetic algorithm. All configuration names start with MA followed by abbreviations for the added recombine and mutation operations (multiple abbreviations are used to add multiple operations). Abbreviation +C refers to using two-point recombine operations, +ER refers to using multi-recombine operations, and finally +M$_{0.1/0.5}$ adds mutation operations with a mutation chance of 10 (resp. 50) percent. Whenever a mutation operation is performed, both operators have a 50 percent change of being chosen.

Figure 5.2 compares the different MA configurations. Of all configurations, MA+ER, which relies only on multi-point recombine operations, performs worst. Comparing its performance with MA+C (which uses only two-point recombine operations), we can see that it is indeed beneficial to guarantee non-decreasing solution quality for combine operations. However, combining both recombination operators results in a performance similar to MA+C+M$_{0.1}$. This can be explained by the fact that multi-recombines also act as mutation operator in that they do not guarantee non-decreasing quality. Due to the fact that the strong n-level local search engine kKaHyPar computes high-quality solutions, we see that a significant amount of mutations is necessary to ensure diversity in the population. While MA+C+M$_{0.1}$ (10% mutation chance performed best for evolutionary graph partitioning in [SS12]) performs equally well

[1] Compared to partitioning a single instance on a single node, we did not observe considerably different results.

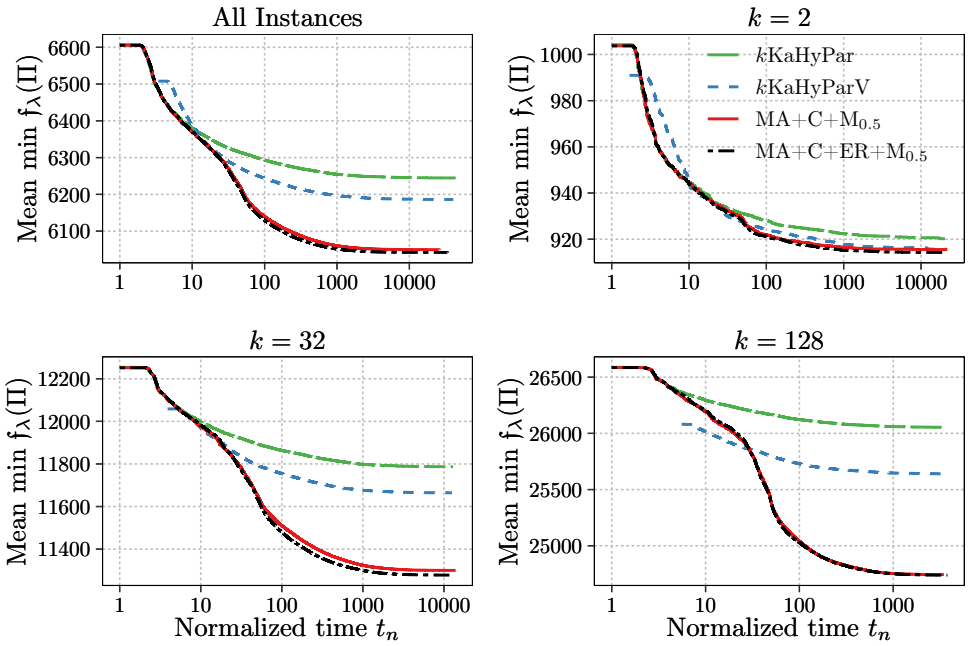


Figure 5.3: Convergence plots for all instances and for different values of k. Experiments are performed on benchmark set C.

as MA+C+ER, increasing the mutation rate to 50% (MA+C+M$_{0.5}$) improves the overall performance of the algorithm. Moreover, we see that using both recombination operators and mutations (MA+C+ER+M$_{0.5}$) also performs well. Since MA+C+M$_{0.5}$ and MA+C+ER+M$_{0.5}$ show the best convergence behavior, we restrict ourselves to these configurations for the remaining experiments.

Comparison with *k*KaHyPar. We now compare the two best-performing MA configurations with repeated executions of two kKaHyPar configurations. The first configuration corresponds to the kKaHyPar variant described in Section 4.8 (without flow-based refinement). Since it is known that global search strategies are more effective than plain restarts [SS11], we augment the first configuration with V-cycles (in a similar fashion as the first mutation operator) using a maximum number of 100 V-cycle iterations per partitioner call. This enhanced version of kKaHyPar constitutes the second configuration and is referred to as kKaHyParV. While both non-evolutionary algorithms *repeatedly partition* each instance until the time limit is reached, the memetic algorithms evolve a population of solutions. Figure 5.3 and Table 5.1 compare the performance of the memetic algorithms with repeated executions of the plain KaHyPar configurations. When looking at convergence plots, note that kKaHyParV starts later than all other algorithms and has an initial better solution quality. This is due to the fact it uses up to 100 V-cycles before reporting the first solution.

Table 5.1: Improvement in solution quality (in %) of the two best performing memetic configurations over both kKaHyPar and the V-cycling version kKaHyParV. Experiments are performed on benchmark set C.

	kKaHyPar vs. MA		kKaHyParV vs. MA	
k	+C+$M_{0.5}$	+C+ER+$M_{0.5}$	+C+$M_{0.5}$	+C+ER+$M_{0.5}$
all	3.2	3.3	2.2	2.3
2	0.9	0.9	0.3	0.4
4	1.3	1.4	0.8	1.0
8	2.6	2.8	1.8	2.0
16	3.4	3.5	2.4	2.5
32	4.2	4.4	3.1	3.4
64	4.7	4.8	3.4	3.5
128	5.2	5.2	3.6	3.6

The improvements of the memetic algorithms increase with increasing k. This is expected as the search space of possible partitionings increases with the number of blocks. Looking at Table 5.1, we see that both memetic algorithms on average outperform kKaHyPar, culminating in an improvement of 5.2% for $k = 128$. Furthermore, both MA+C+$M_{0.5}$ and MA+C+ER+$M_{0.5}$ are able to improve upon the new V-cycling version kKaHyParV for all values of k and perform 3% better on average than kKaHyParV for $k \geq 32$. While the difference in solution quality between both memetic algorithms is small on average, a Wilcoxon matched pairs signed rank test reveals that the improved solution quality of MA+C+ER+$M_{0.5}$ is statistically significant ($p = 0.0027$).

Effects of Flow-Based Refinement. Figure 5.4 summarizes the result of additionally enabling the flow-based refinement framework (+F) for configuration MA+C+ER+$M_{0.5}$. Since using flow-based refinement increases the running time of kKaHyPar approximately by a factor of two, MA+C+ER+$M_{0.5}$+F spends more time on improving individual solutions of the population than MA+C+ER+$M_{0.5}$. Nevertheless, we can see in both the convergence plot and the performance profile that using stronger refinement algorithms to exploit the local solution space improves the overall solution quality. While the average improvement is small, the performance difference is statistically significant ($p < 2.22 \times 10^{16}$). The default configuration of the memetic algorithm therefore always uses the flow-based refinement framework.

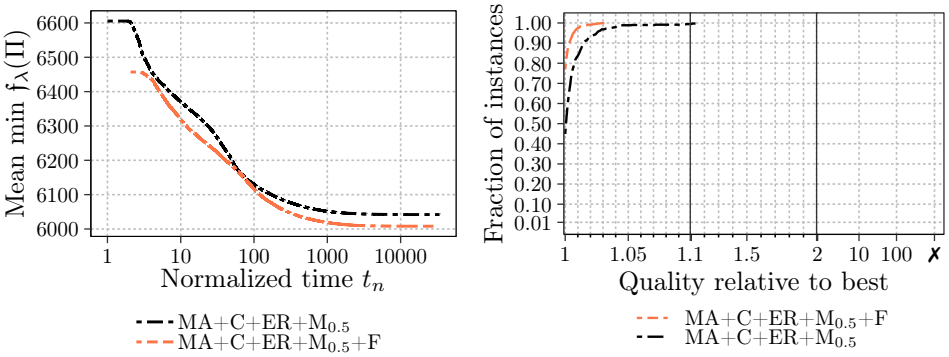


Figure 5.4: Convergence plot (left) and performance profile (right) comparing the performance of the memetic algorithm with and without flow-based refinement. Experiments are performed on benchmark set C.

Chapter 6

6 Experimental Evaluation – Comparison to Other Systems

"There is a disappointing lack of data comparing partitioning algorithms."

— William E. Donath [Don88]

Motivation. The quote of Donath shown above dates back to the year 1988 when multi-level partitioning algorithms were yet to be invented. It was picked up again by Hauck and Borriello [HB97] in 1997, who note that it "still holds true, with many approaches but few overall comparisons". Although the development of new HGP tools has somewhat slowed down after the year 2000, there are still at least ten hypergraph partitioning systems available today (see Section 3.5) – and we still lack an extensive experimental evaluation.

In this dissertation, we so far have only compared different KaHyPar configurations to each other and thus still owe the reader a comparison to the state of the art. The goal of this chapter therefore is to take the next step in addressing the problem stated by Donath by performing an *extensive* experimental evaluation using the benchmark sets described in Section 2.6.1. After discussing our reasoning for choosing a set of *seven* state-of-the-art HGP algorithms as competitors in Section 6.1, we compare the performance of rKaHyPar and kKaHyPar for cut-net and connectivity optimization with those systems in Section 6.2 – both in terms of solution quality and running time. As we will see, the differences in running time between the algorithms can be up to several orders of magnitude. In Section 6.3, we therefore evaluate a subset of the best performing algorithms in a setting where each algorithm is given the same – large – amount of time to compute a solution for each instance. Furthermore, we use this section to demonstrate the effectiveness of our memetic algorithm presented in Chapter 5. Since the experiments in Section 6.2 and Section 6.3 use benchmark hypergraphs that were also used during the development of KaHyPar, we use the graph *edge* partitioning problem (which can be solved via hypergraph partitioning) as a case study in Section 6.4 to demonstrate the performance of KaHyPar on hypergraphs that were *never* used during its development – again considering all seven other HGP algorithms. Finally, in Section 6.5, we evaluate our algorithms in the context of traditional graph partitioning – comparing them with the current best sequential graph partitioning algorithm KaFFPa [SS11].

References. This chapter contains experimental results and text passages from several publications. More precisely, Section 6.1 is based on Refs. [Sch+16a; HSS19a]. Section 6.2 is based on Ref. [HSS19a] and Section 6.3 expands on the experimental evaluation presented in Ref. [ASS18a]. The case study on edge partitioning is based

on a conference paper [Sch+19a] and a technical report [Sch+18a] jointly written and published with Christian Schulz, Daniel Seemaier, and Darren Strash. Parts of these two publications were copied verbatim. Parts of the graph partitioning experiments presented in Section 6.5 were taken from Ref. [HSS19a]. In all cases, the experimental evaluation was done by the author of this dissertation. Note that at the beginning of each of the following sections, we precisely state which data was taken from previous publications and which experiments were done exclusively for this dissertation.

6.1 Partitioning Systems

Hypergraph Partitioning Algorithms. We compare different KaHyPar configurations to the k-way (hMETIS-K) and recursive bisection variants (hMETIS-R) of hMETIS 2.0 (p1) [Kar19], PaToH 3.2 [Çat19] using both the default (PaToH-D) and the quality preset (PaToH-Q), Zoltan-AlgD [Sha19], Mondriaan version 4.2.1 [Bis+19], and HYPE [ME19]. We choose these tools for the following reasons: PaToH produces better quality than Zoltan's native parallel hypergraph partitioner (PHG) in sequential mode [Dev+06; Bom+12b]. Parkway does not run in sequential mode and was found to be comparable to Zoltan's PHG in sequential mode [Dev+06]. The algebraic distance-based coarsening algorithm of Zoltan-AlgD has been shown to improve the performance of Zoltan's PHG in sequential mode for the cut-net metric [SS18b; SS18a; SSC19]. Furthermore, Zoltan-AlgD also performed better than Zoltan for connectivity optimization in Ref. [Sch+19a]. MLPart is restricted to bipartitioning [CKM00c; CRX03] and was outperformed by both hMETIS [PM06] and PaToH [Çat]. ReBaHFC is also restricted to bipartitioning and does not perform better than KaHyPar for $\varepsilon > 0$ [GHW19a; GHW19b]. The performance of SHP is deemed comparable to the performance of Zoltan and Mondriaan [Kab+17]. UMPa does not improve on PaToH when optimizing single objective functions that do not benefit from the directed hypergraph model [Çat+15]. Furthermore, kPaToH [ACU08] did not perform better than PaToH in preliminary experiments [Akh+17a]. Lastly, HYPE so far has only been evaluated on four hypergraphs and only been compared to hMETIS-R [May+18].

Graph Partitioning Algorithms. For graph partitioning, we restrict the experimental comparisons to the Karlsruhe Fast Flow Partitioner (KaFFPa) [SS11] from the multi-level graph partitioning framework KaHIP [SS13]. KaFFPa has been shown to perform better than competing graph partitioning algorithms such as METIS [KK98a], SCOTCH [Pel19], or DibaP [Mey08]. In the experiments, we use both the `strong` (KaFFPa-Strong) and the `strongsocial` (KaFFPa-StrongS) configurations. The former is designed for high-quality partitions of mesh graphs, while the latter is targeted at partitioning complex networks such as web graphs and social networks.

A Note on Algorithm Configuration. Almost all partitioning systems have a considerable number of tuning parameters and configurable subroutines, many of which interact in nontrivial ways. We therefore refrain from tuning these systems ourselves and instead use the configurations provided by the authors, which have been

shown to work well in the corresponding publications.

However, special care has to be taken when using hMETIS. The system does not permit direct optimization of the connectivity metric $f_\lambda(\Pi)$. Instead, it optimizes the sum-of-external-degrees (SOED) metric $f_s(\Pi)$, which is closely related to the connectivity metric, since $f_\lambda(\Pi) = f_s(\Pi) - f_c(\Pi)$ for hypergraphs with unit net weights (i.e., each cut-net contributes λ times its weight to the objective). In all experiments that use the connectivity metric as optimization objective, we therefore set both hMETIS variants to optimize $f_s(\Pi)$ and then calculate $f_\lambda(\Pi)$ accordingly. This approach is also used by the authors of hMETIS-K [KK99]. Furthermore, when using hMETIS-R for SOED optimization, we explicitly set the `reconst` parameter to force cut-net splitting as proposed by Karypis and Kumar [KK98b]. When optimizing the cut-net metric $f_c(\Pi)$, the parameter is omitted to allow the partitioner to perform cut-net removal.

As we have discussed in Section 3.5, hMETIS-R defines the maximum allowed imbalance differently [Kar+99]. We therefore translate our imbalance parameter ε to ε' as described in Eq. (6.1) such that it matches our balance constraint after $\log_2(k)$ bipartitions:

$$\varepsilon' := 100 \cdot \left(\left((1+\varepsilon) \, \frac{\lceil \frac{c(V)}{k} \rceil}{c(V)} \right)^{\frac{1}{\lceil \log_2(k) \rceil}} - 0.5 \right). \tag{6.1}$$

> We would like to mention that – with the notable exception of Trifunović [Tri06, p. 149] – this approach is largely ignored in the hypergraph partitioning community (even in the original hMETIS-K paper [KK99]). We consider this to be problematic, since – as we will see in this chapter – in most of our experiments this hMETIS-R configuration always produces balanced partitions and outperforms its direct k-way counterpart.

For Zoltan-AlgD, we set the parameter `phg_edge_size_threshold` to 1.0 in order to prevent the algorithm from removing hyperedges whose number of pins divided by the number of vertices in the hypergraph exceeds this threshold.[1]

Dealing with Imbalanced Partitions. In previous publications, a partition of an instance was considered to be infeasible if the average imbalance (averaged over all repetitions) was above the predefined imbalance parameter ε. Here, we follow a different approach: We discard all runs that produced an imbalanced solution and only report an instance as infeasible, if *all* runs produced imbalanced partitions. Hence, the results presented in this section differ from the results of the respective publications.

6.2 Experimental Results

In this section, we compare rKaHyPar and kKaHyPar to the state-of-the-art HGP systems on benchmark set A for connectivity as well as cut-net optimization.

[1] This was also suggested in a personal correspondence with Ruslan Shaydulin.

Methodology. The evaluation uses experimental results on benchmark set A from previous publications [HS17a; HSS19a]. More precisely, we use the partitioning results of hMETIS-R, hMETIS-K, PaToH-D, PaToH-Q, and HYPE for connectivity optimization. For cut-net optimization, we use the results of hMETIS-R, hMETIS-K, PaToH-D, and PaToH-Q (HYPE does not allow optimizing the cut-net metric). All comparisons involving benchmark set A are therefore based on 3 222 instances, since 194 out of all 3 416 instances were already excluded in Ref. [HS17a] because either PaToH-Q could not allocate enough memory or other partitioners did not finish in time. For all algorithms except HYPE, the experimental data contain the computed cut/connectivity values, imbalances, and running times per instance for ten repetitions with different seeds. For HYPE, the data contain the results of *one* iteration using the default preset (which is not randomized and was also used in the experiments of Mayer et al. [May+18]), since employing randomization did not improve solution quality in our preliminary experiments presented in Ref. [HSS19a]. Each partitioner had a time limit of eight hours *per* instance and seed.

Experiments involving *r*KaHyPar, *k*KaHyPar, Zoltan-AlgD, and Mondriaan are new and were done on the same system used in the experiments presented in Refs. [HS17a; HSS19a] (see Section 2.6.2).[2] For all experiments, we use the same number of repetitions and the same time limit of eight hours.

6.2.1 Solution Quality

Connectivity Optimization. Figure 6.1 (left) summarizes the results for connectivity optimization. The performance profile plot shows that both KaHyPar configurations outperform the competing algorithms. The direct k-way algorithm *k*KaHyPar computes the best partitions for 62.4% of all benchmark instances. The recursive bipartitioning-based algorithm *r*KaHyPar computes the best solutions for 28.3% of all instances. Furthermore, the solution quality of both algorithms is within a factor of 1.1 of the best algorithm in more than 90% of all cases. As can be seen in Table 6.1, the performance difference between all algorithms is statistically significant for all pairwise comparisons except PaToH-Q and hMETIS-K, and PaToH-D and Mondriaan.

Comparing the performance profiles of PaToH-Q with the performance profile of Mondriaan, we can see that PaToH-Q performs better than the most recent version of Mondriaan on our benchmark set. Given the fact that the performance difference of PaToH-D and Mondriaan is not statistically significant, this confirms the results of previous studies that suggested that Mondriaan's hypergraph partitioner can be seen as being inferior to PaToH [Riy03; Bis+12] (at least when using the quality preset). Moreover, we note that HYPE (the only non-multi-level algorithm) performs considerably worse than the multi-level systems. This echoes the intuition that by providing a more global view of the partitioning problem on coarser levels, multi-level approaches enable local search algorithms to explore local solution spaces very

[2] We perform new experiments for Zoltan-AlgD, because previous experiments were done with an earlier version of the algorithm that was missing some significant performance improvements.

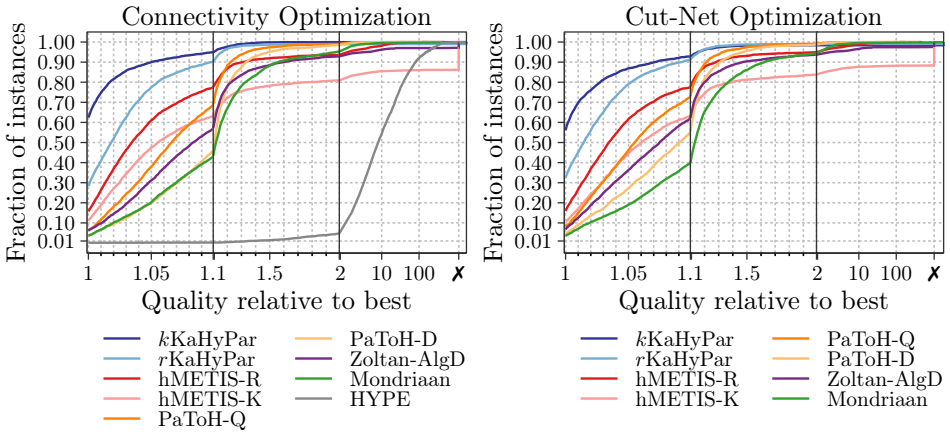


Figure 6.1: Performance profiles comparing *k*KaHyPar and *r*KaHyPar with other partitioners for connectivity (left) and cut-net optimization (right).

effectively. While the performance profile of hMETIS-R is within a factor of 1.1 of the best algorithm for more than 77% of all instances, there are some instances for which it performs significantly worse than the best. Although less solutions of PaToH-Q and PaToH-D are within a factor 1.1 from the best, the performance profiles indicate that the worst quality ratios of PaToH-Q and PaToH-D are smaller than those of hMETIS-R. Note that the same effect is also visible when comparing Zoltan-AlgD with PaToH-D or Mondriaan.

Looking at infeasible solutions, we see that Zoltan-AlgD computes imbalanced solutions for around 3% of all instances and that around 14% of all partitions computed by hMETIS-K are imbalanced. The fact that hMETIS-K often produces imbalanced partitions was also observed in the graph partitioning experiments of Schulz [Sch13b, p. 128]. A possible explanation for this behavior could be the fact that, according to the related publications [KK98c; KK99; KK00], hMETIS-K does not limit the maximal vertex weight during the coarsening phase (neither indirectly via a penalty factor in the rating function nor directly via hard weight constraints). This, in turn, could lead to many heavy vertices at the coarser levels of the hierarchy, which makes it harder for the initial partitioning algorithms to compute balanced partitions.

In Figure 6.2 (left) and Figure 6.3 (left), we compare each KaHyPar configuration individually to all other partitioning systems, since in the plot shown in Figure 6.1 (left) the performance ratios of *k*KaHyPar and *r*KaHyPar affect each other. We see that in this setting, the performance difference to the other algorithms is even more pronounced. *k*KaHyPar and *r*KaHyPar now compute the best solutions for 72.7% (resp. 54.1%) of all instances, and the solution quality is within a factor of 1.1 of the best algorithm for 96.1% (resp. 94.5%) of all instances.

Figure 6.4 shows the performance profiles for individual instance classes. We see that *k*KaHyPar performs best across all classes. Furthermore, the plots show that hMETIS

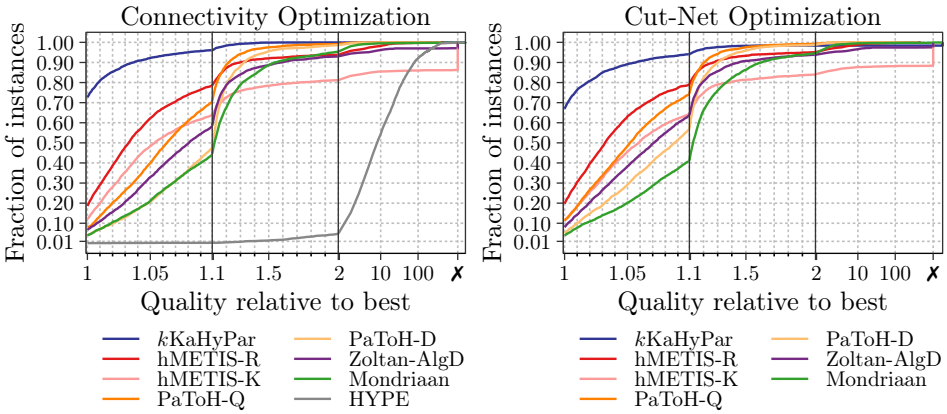


Figure 6.2: Performance profiles comparing kKaHyPar with other partitioners for connectivity optimization (left) and cut-net optimization (right).

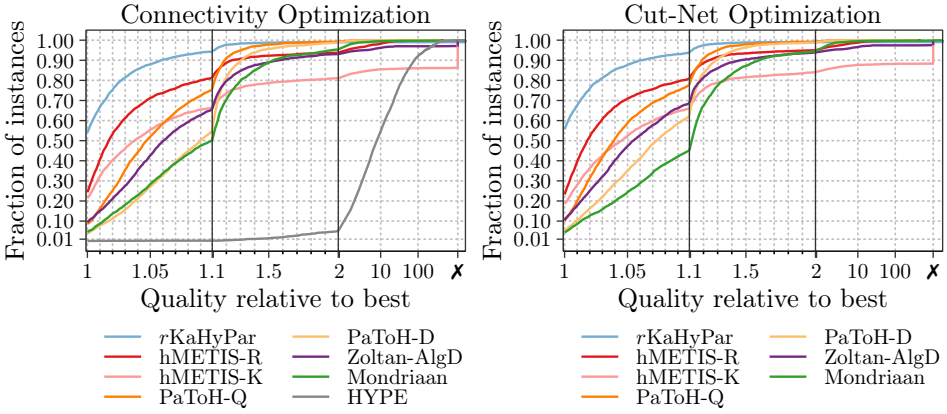


Figure 6.3: Performance profiles comparing rKaHyPar with other partitioners for connectivity optimization (left) and cut-net optimization (right).

– which stems from the area of VLSI design and was extensively tuned and evaluated on the ISPD98 benchmark set – indeed performs better on hypergraphs derived from VLSI circuits (DAC2012 and ISPD98) than on other instance classes. The same observation can be made for PaToH, which was designed for partitioning hypergraphs derived from sparse matrices (SPM), and to some extent also for Mondriaan (also designed for matrix partitioning). Additionally, we see that for more than 30% of all SAT dual instances, all partitioners except kKaHyPar and rKaHyPar compute partitions that are more than a factor of 1.1 worse than the best solution.

Looking at the results for specific values of k in Figure 6.5, we see that for $k = 2$, rKaHyPar computes the best solutions for slightly more instances than kKaHyPar.

Table 6.1: Results of the significance tests for all pairwise algorithm comparisons (p-values) for connectivity optimization. The value 0 is used to denote results that were below machine precision.

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)
(a) kKaHyPar	-	0	0	0	0	0	0	0	0
(b) rKaHyPar	0	-	0	0	0	0	0	0	0
(c) hMETIS-R	0	0	-	0	0	0	0	0	0
(d) hMETIS-K	0	0	0	-	0.0451	0	0	0	0
(e) PaToH-Q	0	0	0	0.0451	-	0	0	0	0
(f) PaToH-D	0	0	0	0	0	-	0	0.233	0
(g) Zoltan-AlgD	0	0	0	0	0	0	-	0	0
(h) Mondriaan	0	0	0	0	0	0.233	0	-	0
(i) HYPE	0	0	0	0	0	0	0	0	-

Table 6.2: Results of the significance tests for all pairwise algorithm comparisons (p-values) for cut-net optimization. The value 0 is used to denote results that were below machine precision.

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
(a) kKaHyPar	-	9.26×10^{-12}	0	0	0	0	0	0
(b) rKaHyPar	9.26×10^{-12}	-	0	0	0	0	0	0
(c) hMETIS-R	0	0	-	0	0	0	0	0
(d) hMETIS-K	0	0	0	-	1.51×10^{-10}	0	3.8×10^{-9}	0
(e) PaToH-Q	0	0	0	1.51×10^{-10}	-	0	0	0
(f) PaToH-D	0	0	0	0	0	-	0	1.51×10^{-10}
(g) Zoltan-AlgD	0	0	0	3.8×10^{-9}	0	0	-	0
(h) Mondriaan	0	0	0	0	0	1.51×10^{-10}	0	-

This could be due to the fact that it uses the simple stopping rule for FM-based refinement and thus performs more local search operations. With increasing k, however, kKaHyPar performs better. Also note that, with increasing k, the performance difference between the algorithms becomes more pronounced. Already for $k \geq 4$, out of all competing systems, only hMETIS is able to compute the best solution for more than 10% of all instances. The fact that the performance of rKaHyPar decreases with increasing k could be seen as an indication that the theoretical problems of recursive bipartitioning-based algorithms discussed in Section 3.6.3 indeed occur in practice. Çatalyürek et al. [Çat+15] observed that the partitioning quality of Zoltan deteriorates as k increases and attributed this effect to its simplified refinement algorithm. The figure shows the same effect for Zoltan-AlgD, which only differs from Zoltan in that it uses algebraic distance-based coarsening.

So far, we compared the performance of all algorithms relative to the performance of the respective *best* algorithm. In Figure 6.6 and Figure 6.7 we assess the performance of kKaHyPar and rKaHyPar relative to each of the competing state-of-the-art algorithms individually. We see that both configurations outperform the other partitioning systems in pairwise comparisons. Furthermore, looking at Figure 6.6 (top left), we see that kKaHyPar performs better than rKaHyPar for connectivity optimization.

Cut-Net Optimization. Note that in the following evaluation we exclude HYPE, because it is designed to optimize the connectivity metric and does not support cut-net optimization. The experimental results summarized in Figure 6.1 (right) yield similar conclusions as for connectivity optimization. kKaHyPar computes the best partitions for 56.1% of all instances. Furthermore, its solution quality is within a factor of 1.1 of the best algorithm in 92.9% of all cases. In the case of rKaHyPar, the corresponding percentages are 32.5% and 91.2%, respectively. Table 6.2 shows that for cut-net optimization, the performance difference between all pairs of algorithms is considered statistically significant.

The individual comparisons of each KaHyPar configuration with all other partitioning systems shown in Figure 6.2 (right) and Figure 6.3 (right) also reveal similar results as for connectivity optimization. However, we note that the performance of kKaHyPar slightly decreases (compared to connectivity optimization).

Looking at the performance profiles for different instance classes shown in Figure 6.8, we see that the rKaHyPar performs slightly worse on DAC2012 instances for cut-net optimization than for connectivity optimization (compare Figure 6.4). Furthermore, we would like to point out the fact that the solution quality of rKaHyPar is comparable to the solution quality of kKaHyPar for dual SAT instances (which was not the case for connectivity optimization). This could be explained by the fact that these instances have many low degree vertices and many large hyperedges. For such hypergraphs, optimizing the cut-net metric via direct k-way partitioning is difficult: If large nets have multiple pins in several blocks of the k-way partition, FM-based algorithms are less likely to find meaningful moves, because the gain of moving a single vertex to another block is likely to be zero. Moreover, the effects of the flow-based refinement framework are also limited for cut-net optimization in the direct k-way setting, since improving the cut around two blocks of the k-way partition does not necessarily yield an overall improvement in the k-way partition.

The performance profiles for different values of k shown in Figure 6.9 again yield similar conclusions as the performance profiles for connectivity optimization shown in Figure 6.5. The same is true for the individual pairwise comparisons of kKaHyPar and rKaHyPar with the other partitioning systems shown in Figure 6.10 and Figure 6.11, which confirm that both configurations perform better than each individual competitor.

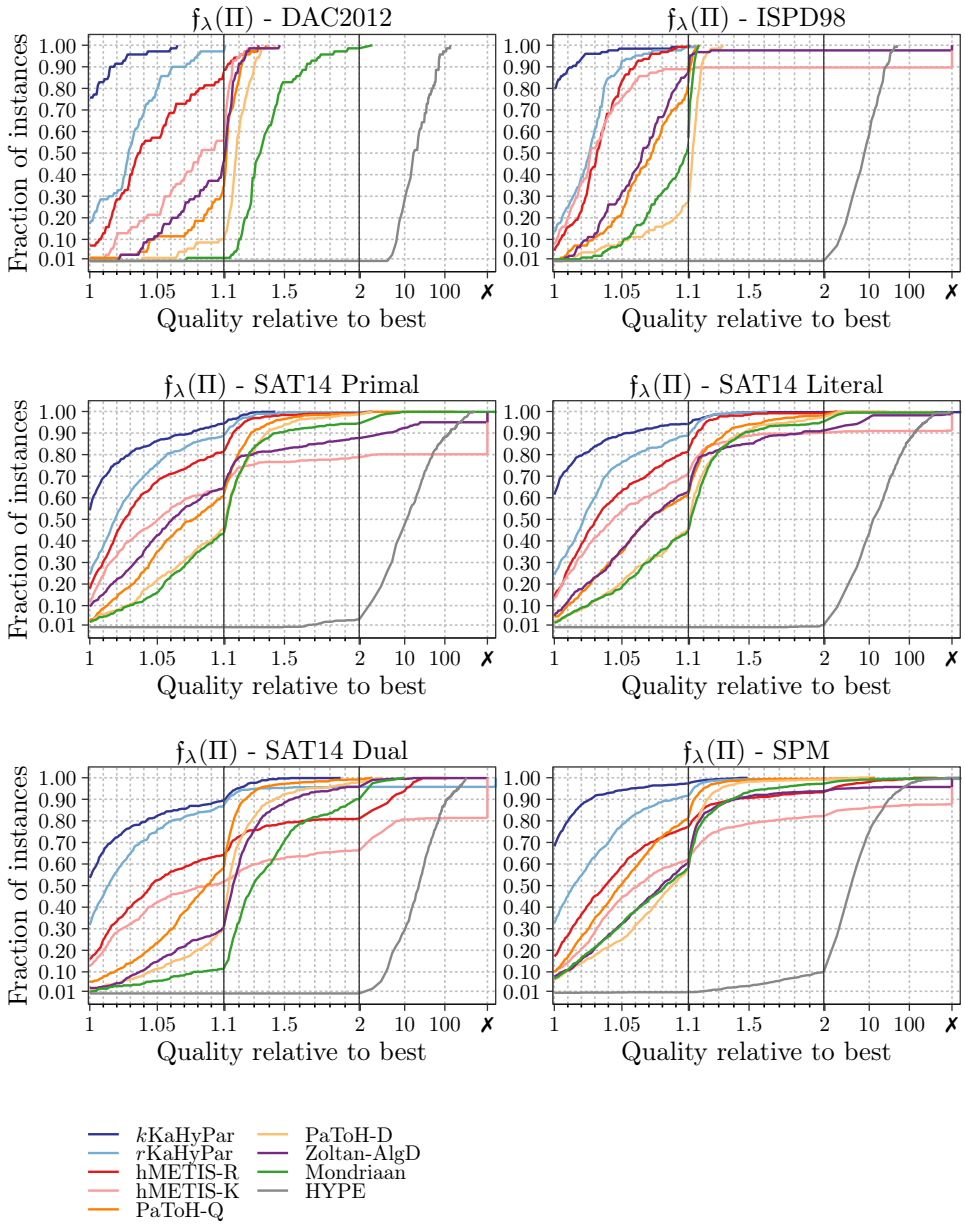


Figure 6.4: Performance profiles for *connectivity* optimization comparing both KaHyPar configurations with the other partitioners for different instances classes.

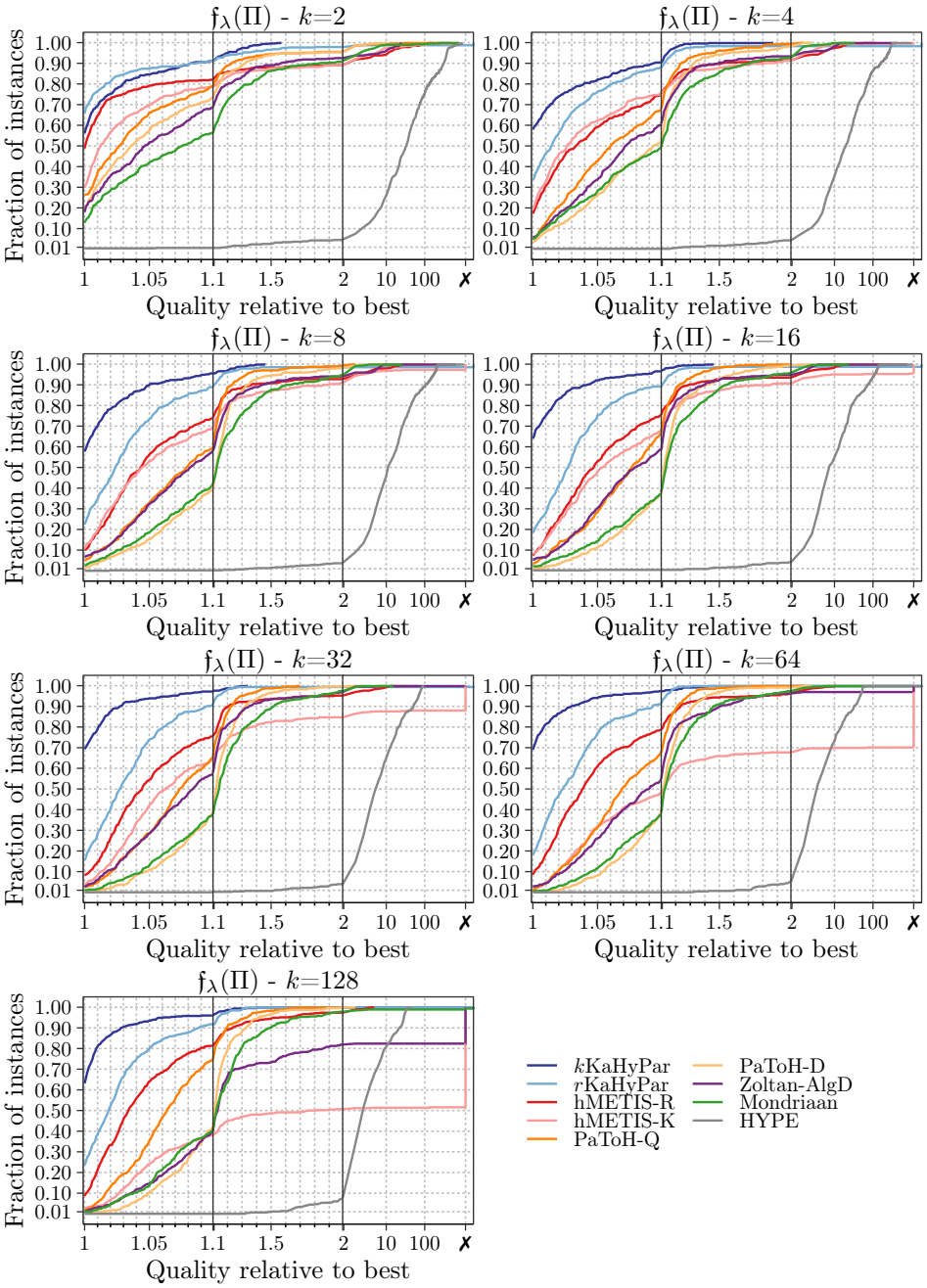


Figure 6.5: Performance profiles for *connectivity* optimization comparing both KaHyPar configurations with the other partitioners for different values of k.

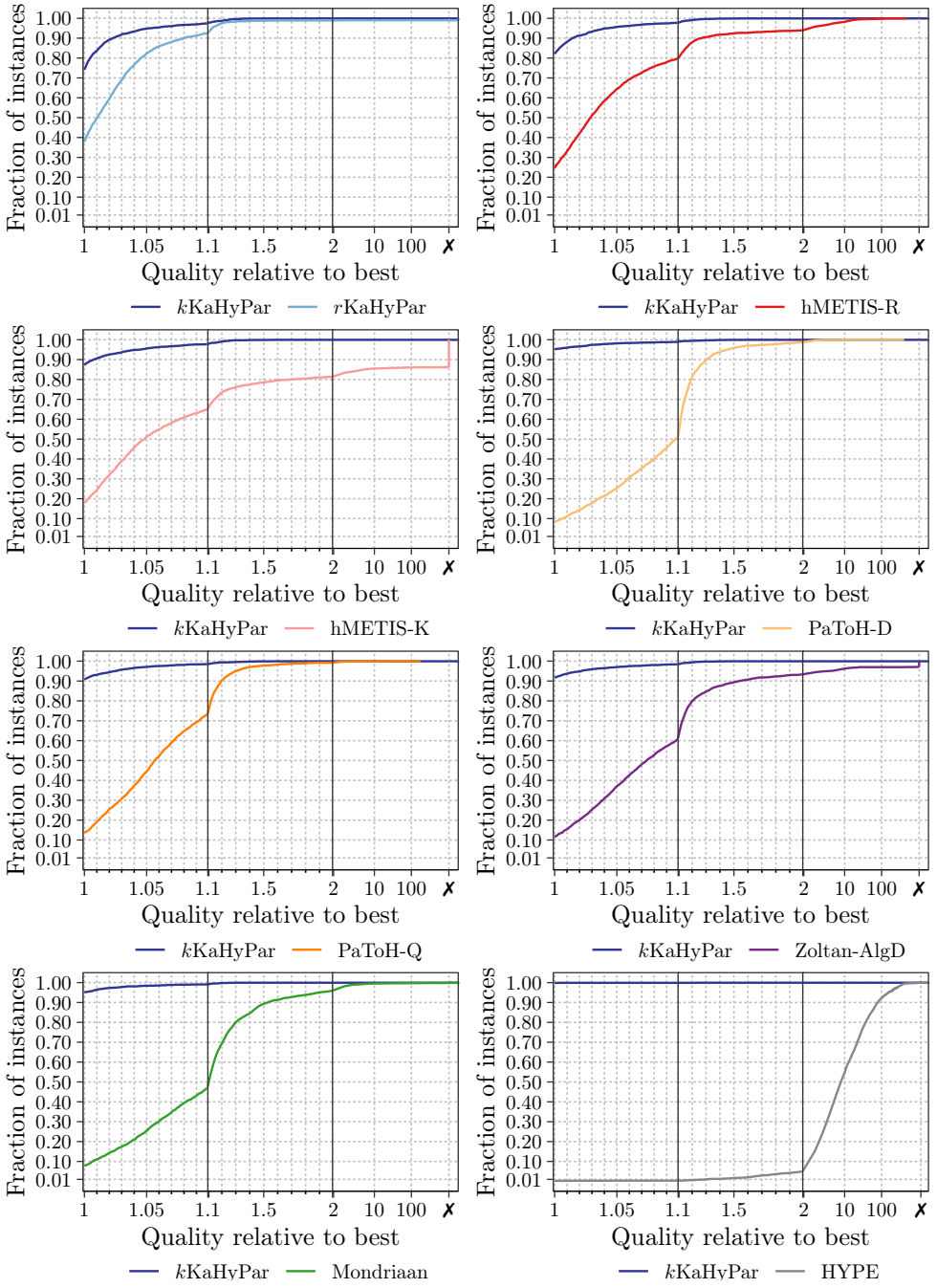


Figure 6.6: Performance profiles for *connectivity* optimization comparing *k*KaHyPar with every algorithm individually.

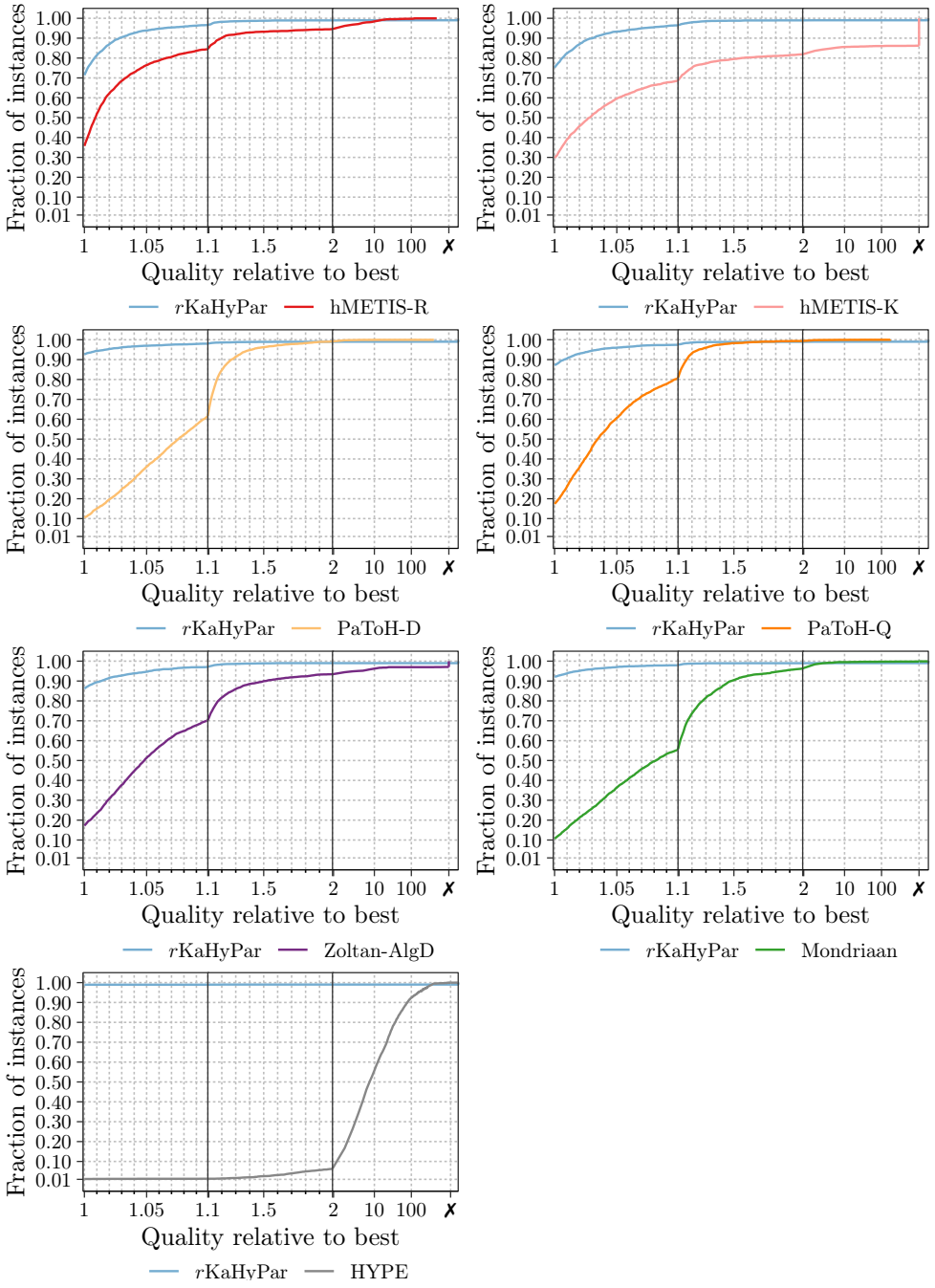


Figure 6.7: Performance profiles for *connectivity* optimization comparing *r*KaHyPar with every algorithm individually.

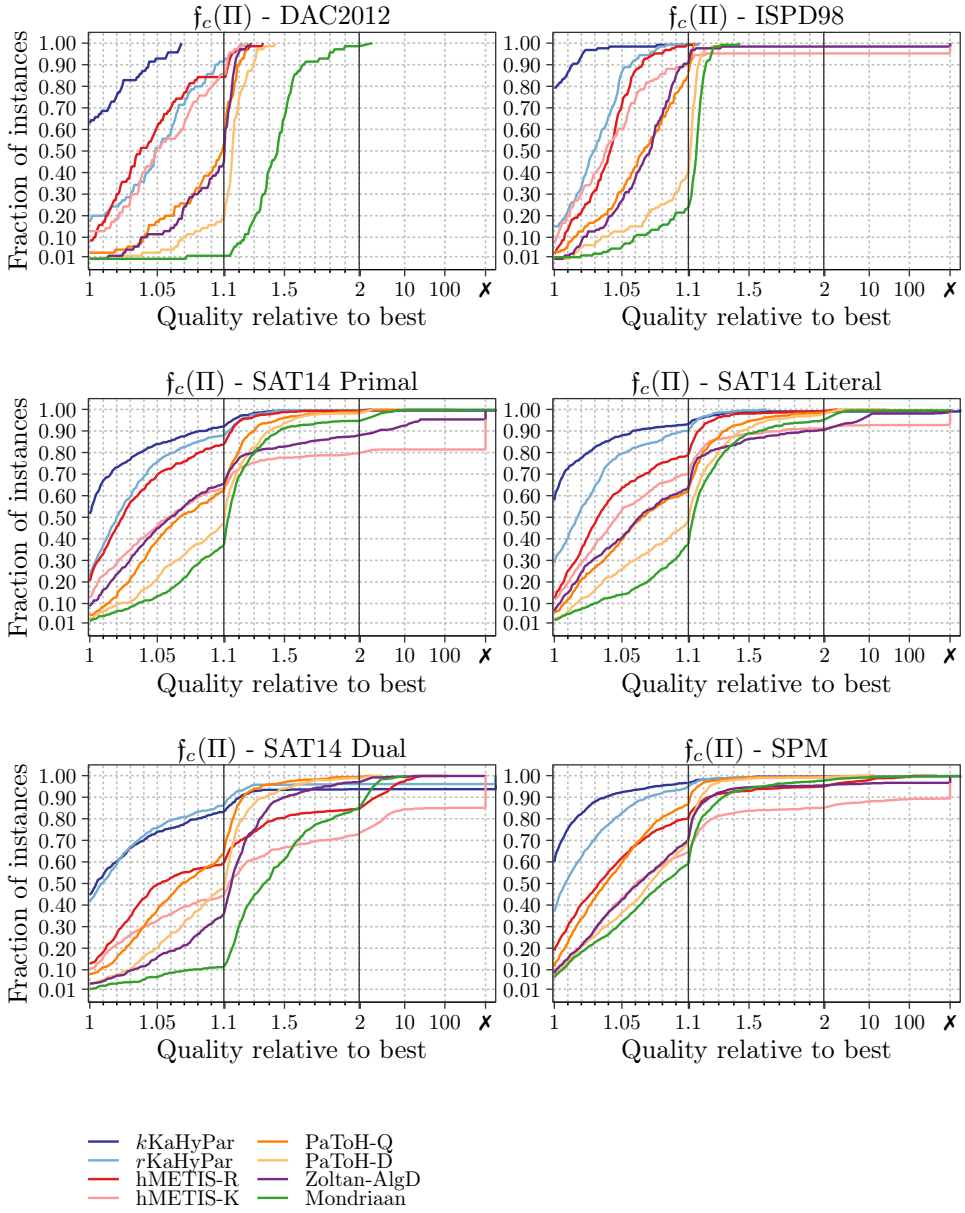


Figure 6.8: Performance profiles for *cut-net* optimization comparing both KaHyPar configurations with the other partitioners for different instance classes.

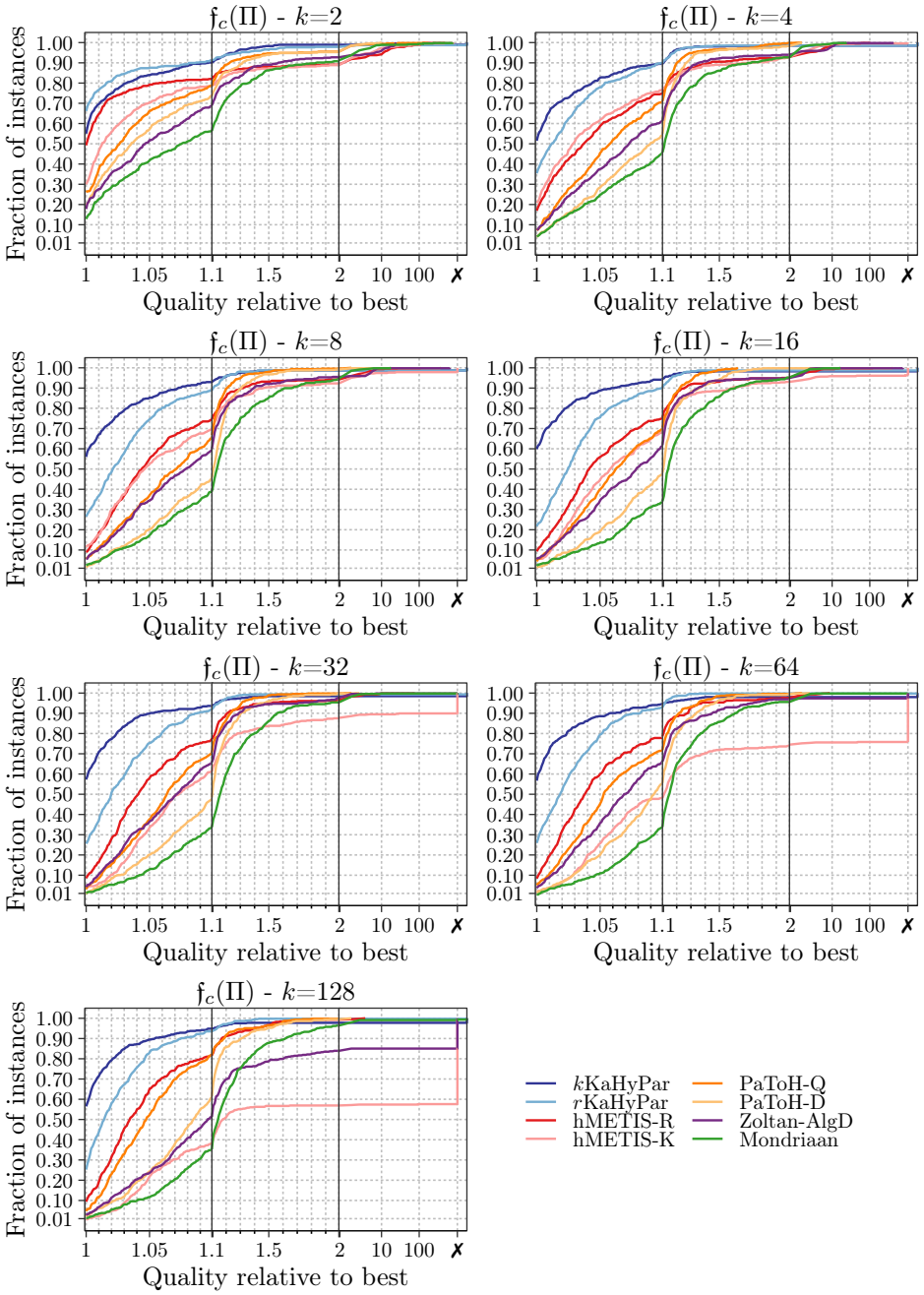


Figure 6.9: Performance profiles for *cut-net* optimization comparing both KaHyPar configurations with the other partitioners for different values of k.

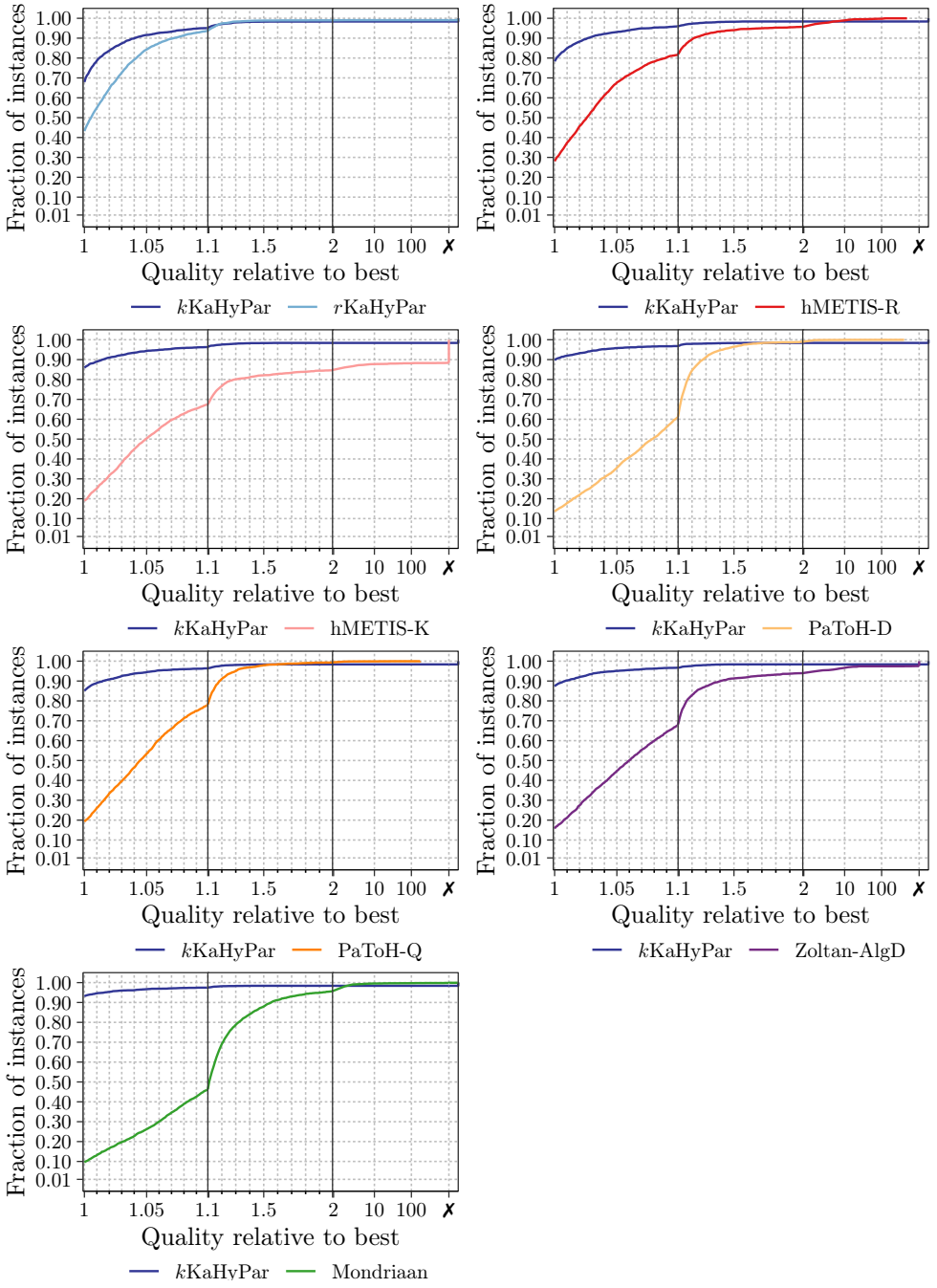


Figure 6.10: Performance profiles for *cut-net* optimization comparing *k*KaHyPar with every algorithm individually.

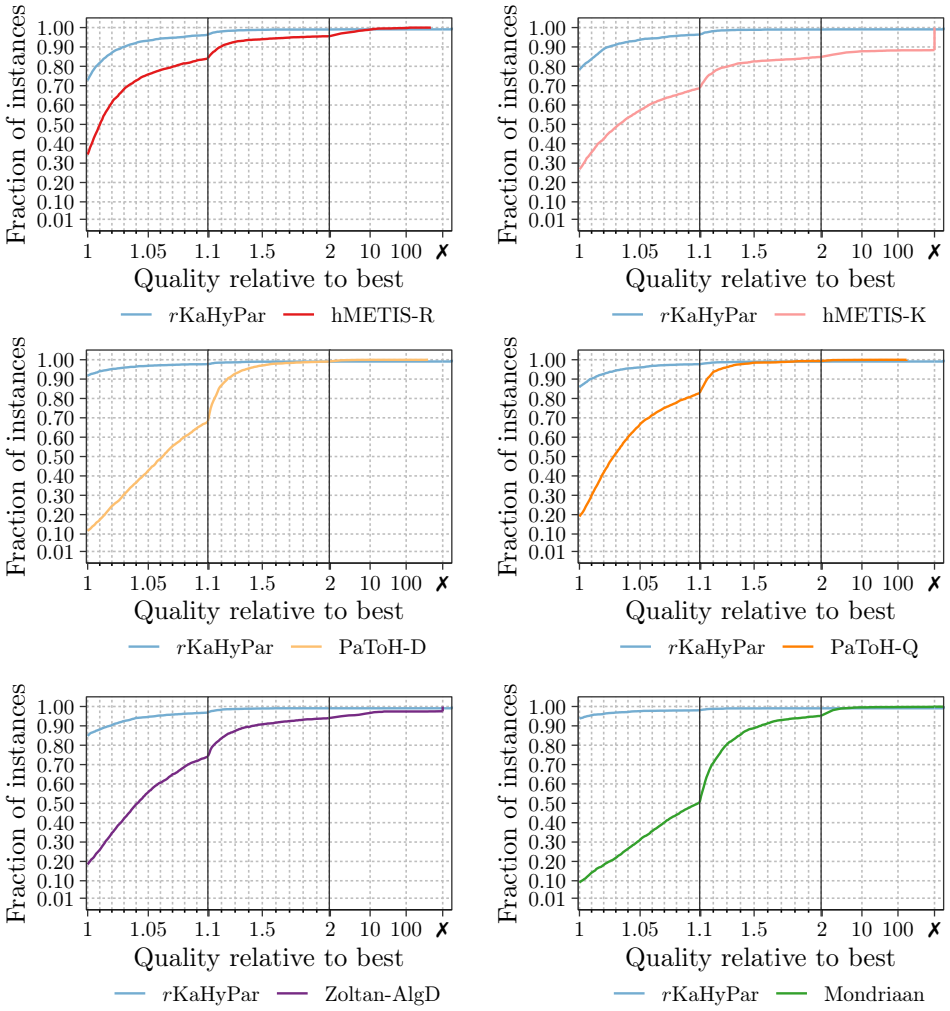


Figure 6.11: Performance profiles for *cut-net* optimization comparing *r*KaHyPar with every algorithm individually.

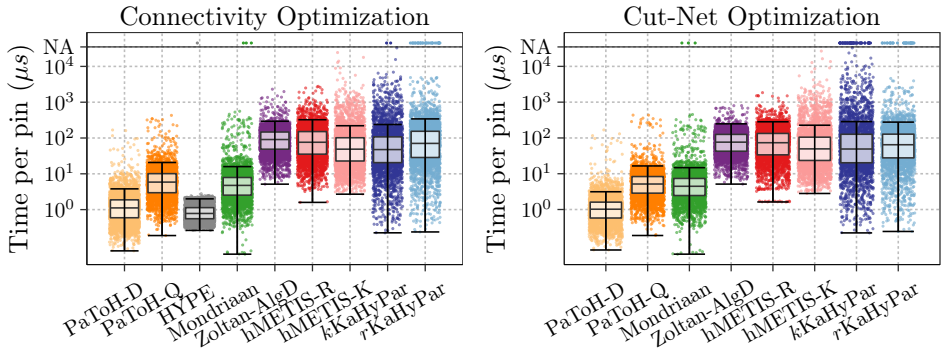


Figure 6.12: Running times of kKaHyPar, rKaHyPar, and other partitioners for connectivity optimization (left) and cut-net optimization (right).

6.2.2 Running Time

We now turn our attention to the running times for connectivity and cut-net optimization. Note that the running time plots presented in this section show instances that could not be partitioned above the vertical line labeled "NA". In the case of kKaHyPar and rKaHyPar, these instances could not be partitioned within the given time limit. HYPE reported an invalid connectivity value for one instance, while Mondriaan aborted partitioning for 6 instances in total.

Connectivity Optimization. The running times of all algorithms for connectivity optimization are shown in Figure 6.12 (left). We see that although being based on the n-level paradigm and employing more complex techniques such as sparsification, community-aware coarsening, and flow-based refinements, the running times of kKaHyPar and rKaHyPar are comparable to the running times of both hMETIS configurations and to the running times of Zoltan-AlgD. Out of all 3 222 instances, rKaHyPar could not finish within the time limit in 32 cases and kKaHyPar in 2 cases. HYPE reported an invalid connectivity value for one instance and Mondriaan aborted partitioning for 3 instances. Note that for both PaToH-Q and PaToH-D, as well as for Mondriaan and HYPE, the median running time is more than an order of magnitude smaller than the median running times of the other multi-level systems.

Looking at the running times for specific instance classes in Figure 6.13, we see that these observations hold for each individual class. Additionally, we see that kKaHyPar is slightly faster on average than rKaHyPar. Figure 6.14 shows the running times for specific values of k. While the running times increase for all multi-level algorithms as k increases, we see that the running time of HYPE is independent of k.

Cut-Net Optimization. The running times for cut-net optimization are summarized in Figure 6.12 (right) for all instances. Figure 6.15 and Figure 6.16 show the running times for individual instance classes and different values of k, respectively. We can see that the running times of rKaHyPar and kKaHyPar are again comparable to

the running times of hMETIS-R, hMETIS-K, and Zoltan-AlgD on average. However, the number of instances that could not be partitioned within the given time limit of eight hours increases to 52 instances for kKaHyPar, while rKaHyPar could not partition 30 instances. We note that PaToH-Q, PaToH-D, and Mondriaan are again an order of magnitude faster than the other partitioners.

Looking at the running times per instance class in Figure 6.15, we observe that most timeouts of the KaHyPar configurations occur for dual SAT instances. Since kKaHyPar was able to partition these instances within the time limit for connectivity optimization (see Figure 6.13), this could be seen as an indication that the direct k-way FM-based refinement algorithm has problems finding meaningful moves when optimizing the cut-net metric on hypergraphs with many large nets.

6.2.3 The Time/Quality Trade-Off

The previous section showed that KaHyPar computes superior solutions for most instances; however at the cost of higher running times than, for example, PaToH. Figure 6.17 concisely summarizes the trade-off between solution quality and running time for connectivity (top) and cut-net optimization (bottom) on a per-instance basis. The plots show the running time of each algorithm relative to the running time of kKaHyPar on the x-axis and the solution quality relative to kKaHyPar on the y-axis. Note that the plot only shows instances that could be partitioned by all algorithms. Since we plot (kKaHyPar/Algorithm)-1 on the y-axis, points above zero correspond to instances where the solution of the respective partitioner was better than the solution of kKaHyPar, while for points below zero kKaHyPar produced solutions of higher quality. A point in the first quadrant therefore represents an instance for which an algorithm computed a solution of higher quality in less time. The x-axis uses a log-scale, while the y-axis uses a cube root scale to reduce right skewness. Imbalanced and thus infeasible solutions are plotted at the y position labeled with "imb".

For both connectivity and cut-net optimization, kKaHyPar seems to be the method of choice for high-quality partitioning, while rKaHyPar can be seen as a viable alternative (e.g. for partitioning hypergraphs with many large nets when optimizing the cut-net metric $\mathfrak{f}_c(\Pi)$). Moreover, kKaHyPar performs better than hMETIS-R, hMETIS-K, and Zoltan-AlgD – computing solutions of higher quality in a comparable amount of time for most instances. If running time is more important than solution quality, both PaToH-D and PaToH-Q currently seem to be the method of choice, as both perform better than Mondriaan. At the cost of considerably worse solutions, the flat partitioning algorithm HYPE can be even faster than PaToH for connectivity optimization.

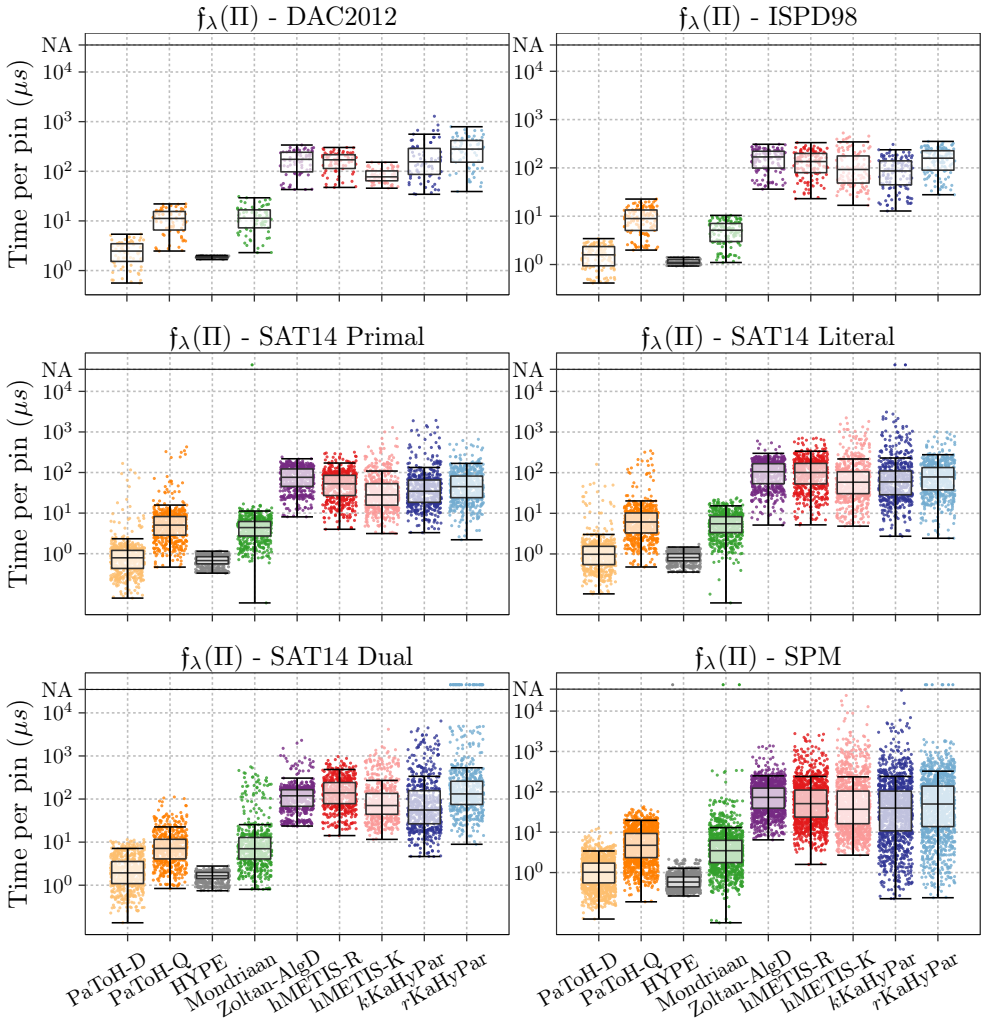


Figure 6.13: Comparing the running times of all algorithms for *connectivity* optimization for different instance classes.

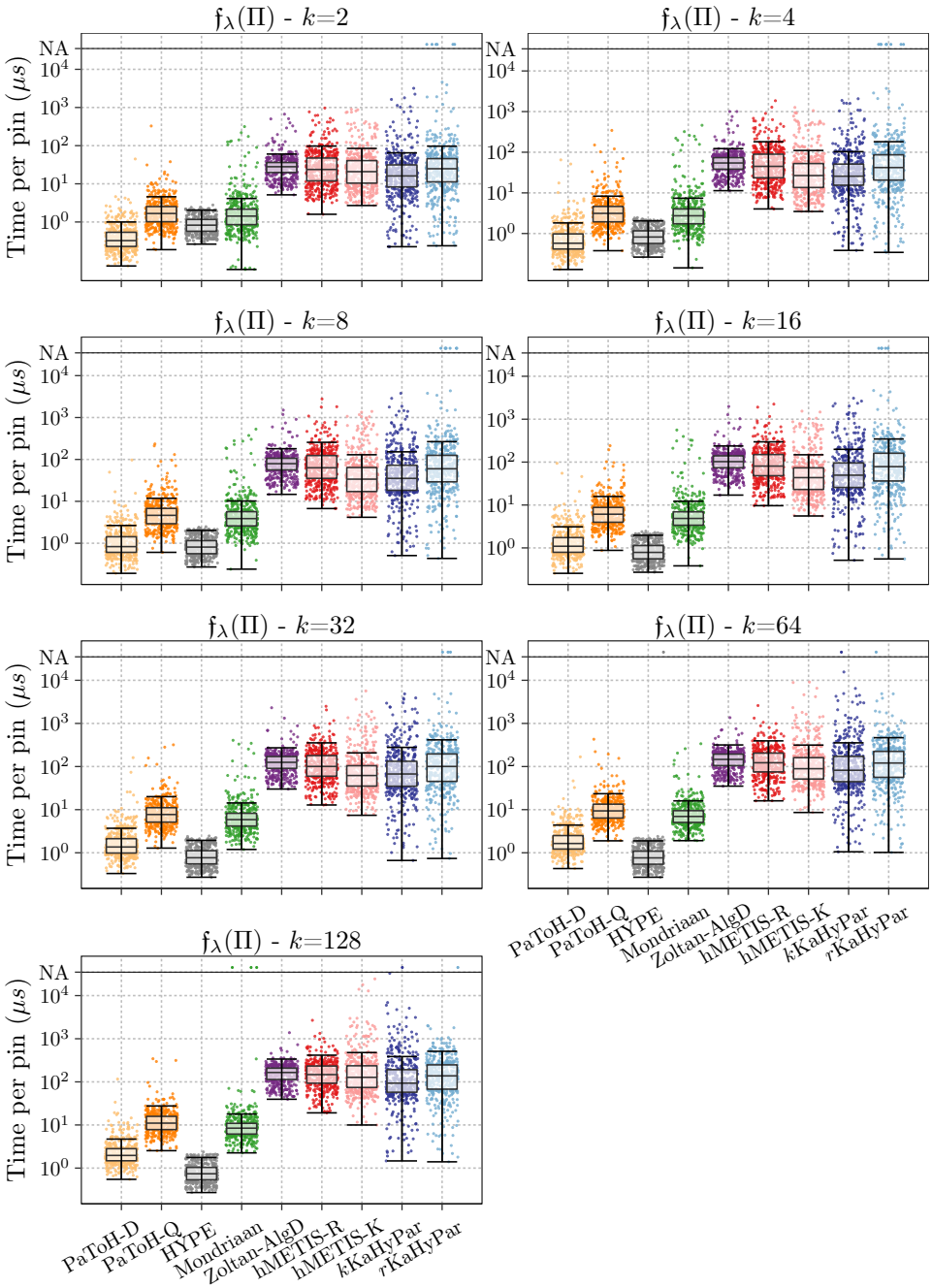


Figure 6.14: Comparing the running times of all algorithms for *connectivity* optimization for different values of k.

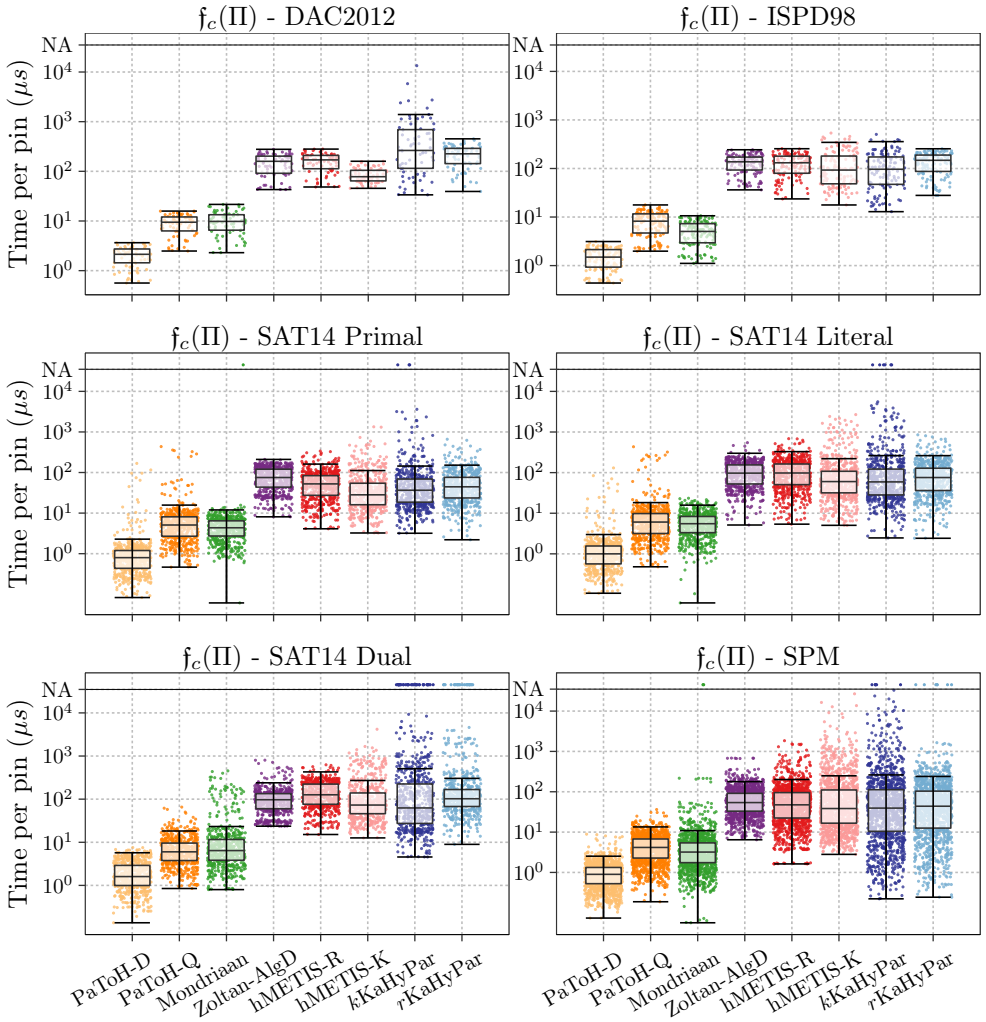


Figure 6.15: Comparing the running times of all algorithms for *cut-net* optimization for different instance classes.

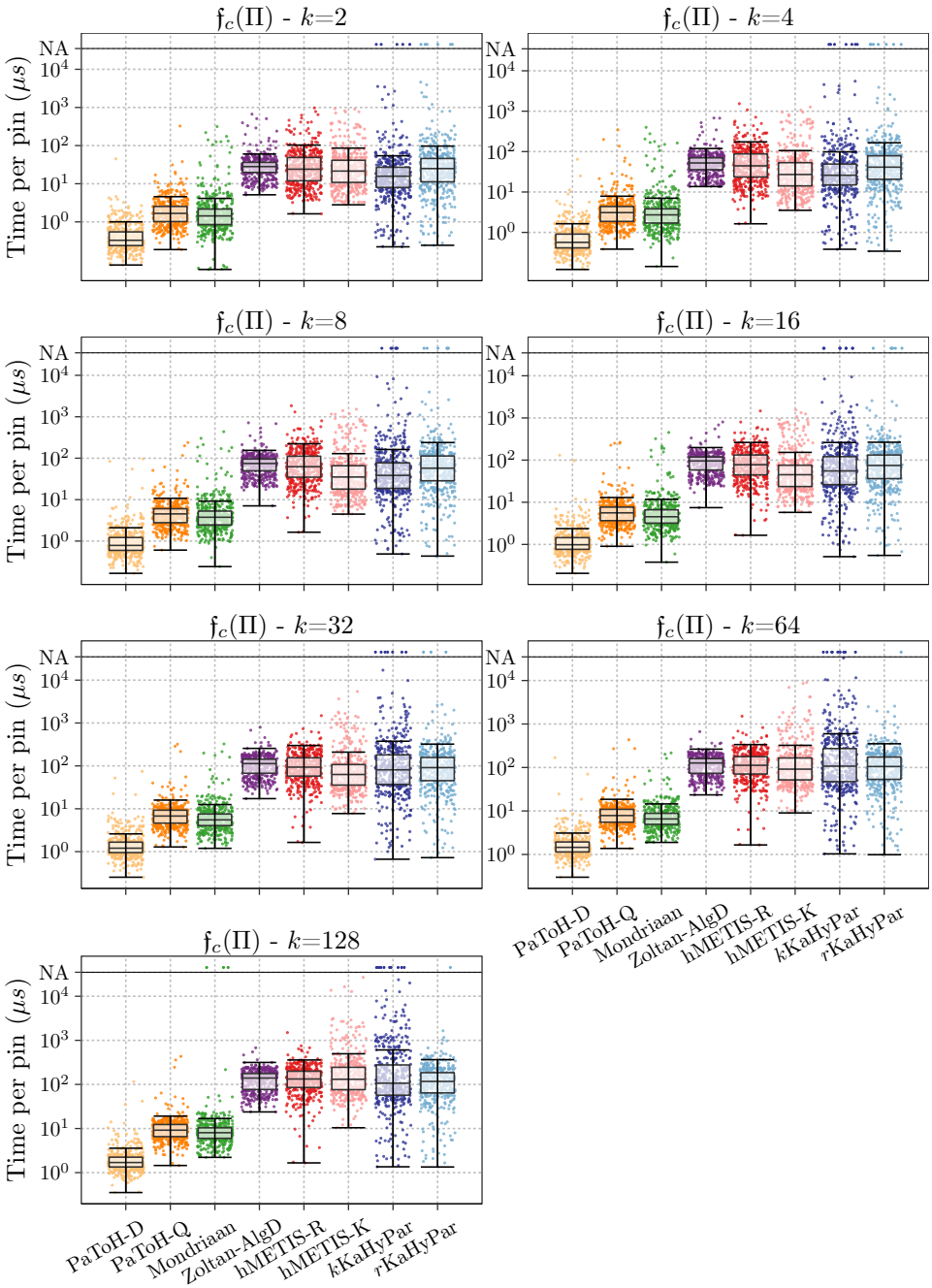


Figure 6.16: Comparing the running times of all algorithms for *cut-net* optimization for different values of k.

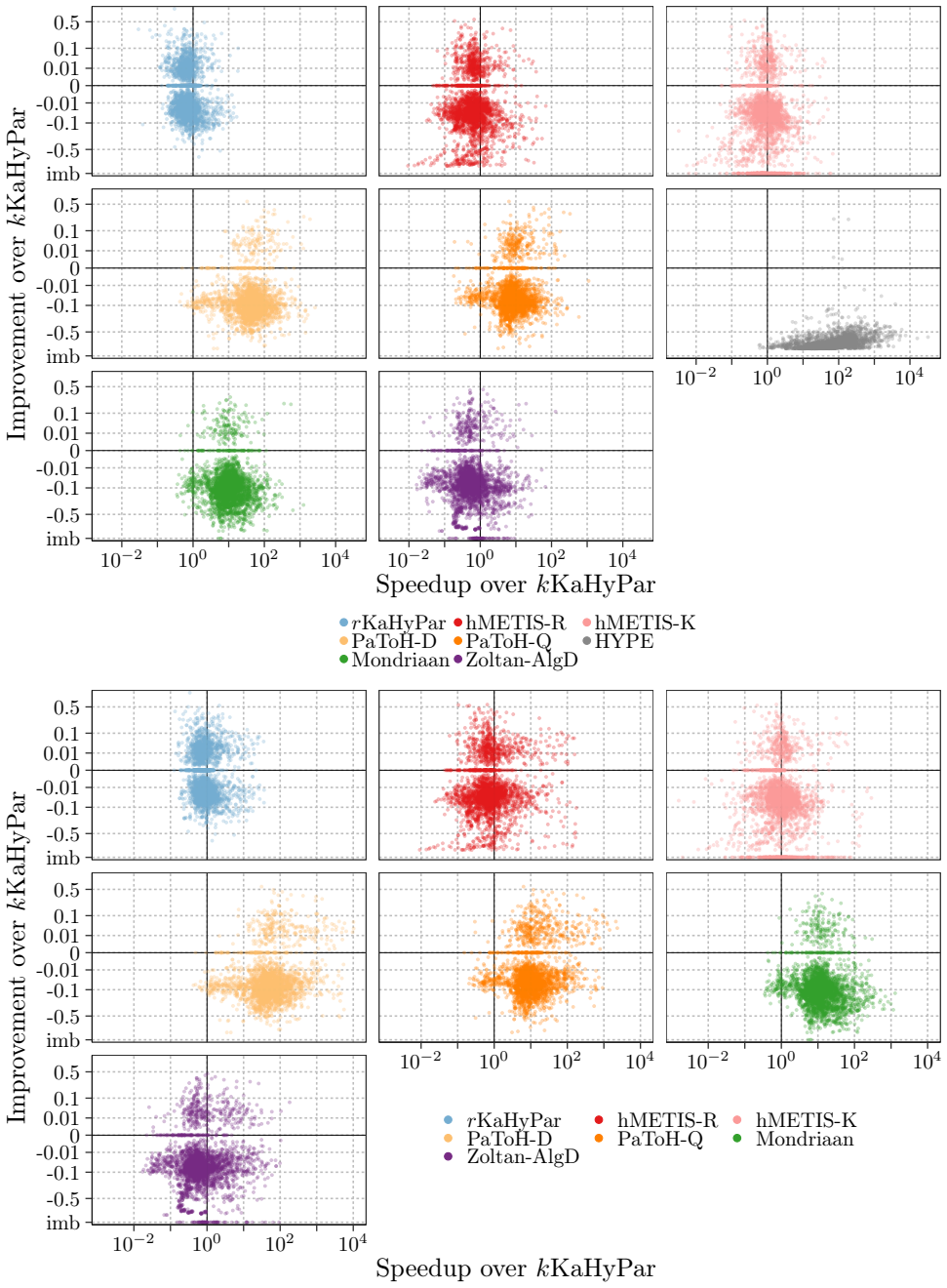


Figure 6.17: Visualization of the trade-off between running time and solution quality for connectivity optimization (top) and cut-net optimization (bottom). The values of all algorithms are relative to *k*KaHyPar.

6.3 Effectiveness Tests using Repeated Executions

Motivation. In the experiments presented in the previous section, each partitioning algorithm was executed the same number of times for each instance (i.e., ten times with different random seeds). However, we have seen that the difference in running time between algorithms can be up to three orders of magnitude. In this section, we therefore investigate the partitioning performance in a setting, where each algorithm is given the *same* fairly large amount of time to partition each instance – thus trying to answer the question whether multiple repetitions of a very fast algorithm can yield similar or even better results than few repetitions of a slower, more advanced algorithm. Furthermore, this evaluation is relevant for tasks such as application-specific integrated circuit (ASIC) design, where one "can afford to allow the partitioner to run for hours or days, since it will take weeks to create the final implementation" [HB95].

Methodology. This section uses experimental results from Ref. [ASS18a] and Ref. [HSS19a]. In both publications, the 100 hypergraphs of benchmark set C were partitioned into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks with an imbalance of $\varepsilon = 0.03$. The connectivity metric $f_\lambda(\Pi)$ was used as optimization objective. For each hypergraph H and each value of k, every partitioning algorithm was given *eight* hours time to compute a solution. We performed five repetitions with different seeds for each test instance and algorithm. Due to the large amount of computing time necessary to perform these experiments, we always partitioned 16 instances in parallel on a single node of the compute cluster.[3] We use the partitioning results of PaToH-D, hMETIS-R, and hMETIS-K from Ref. [ASS18a]. In addition, we use the results of PaToH-Q (which was not evaluated in Ref. [ASS18a]), and the memetic algorithm presented in the Section 5.4 (i.e., configuration MA+C+ER+$M_{0.5}$+F) from Ref. [HSS19a]. For simplicity, we refer to the memetic algorithm as kKaHyPar-E in this section. We perform new experiments for kKaHyPar and rKaHyPar using the same system and the same experimental setup. While all non-evolutionary algorithms *repeatedly* partition each instance until the time limit is reached, kKaHyPar-E evolves a population of solutions.

Experimental Evaluation. In the previous section, we saw that for a small set of instances, hMETIS computes solutions that are significantly worse than the solutions of the respective best algorithm. Since these solutions significantly skew the geometric mean and therefore would yield misleading conclusions, we employ performance profiles instead of showing the evolution of solution quality over time using convergence plots (as in Section 5.4).

The experimental results are summarized in Figure 6.18. The performance profile plot on the left is based on the best solutions computed by all algorithms after eight hours of continuous partitioning. We see that by using the memetic framework described in Chapter 5, kKaHyPar-E is able to effectively explore the global solution

[3] Compared to partitioning a single instance on a single node, we did not observe considerably different results.

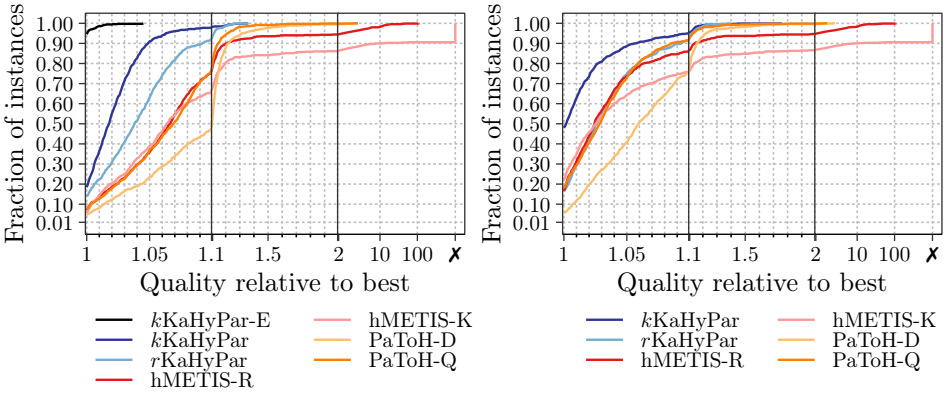


Figure 6.18: Performance profile comparing *k*KaHyPar, *r*KaHyPar, and *k*KaHyPar-E with other partitioners on benchmark set C. The left plot uses the *best* result that each system computed after partitioning each instance for eight hours. The right plot compares the very first results of *k*KaHyPar and *r*KaHypar (reported during the eight hour time period) to the *best* results produced by the other algorithms.

space – computing the best partitions for 94.6% of all instances while never being a factor of 1.045 worse than the best algorithm. However, even in this setting (and even competing with *k*KaHyPar-E), *k*KaHyPar computes the best solutions for 18.4% and *r*KaHyPar for 13.9% of all instances.

Furthermore, we observe that – when given a relatively large time limit for repeated executions with different seeds – the best solutions found by PaToH-Q seem to compare favorably with the best solutions of hMETIS-R and hMETIS-K. Note that this observation is also reflected in the results of the significance tests reported in Table 6.3. Only the difference between the solutions of hMETIS-R and hMETIS-K, hMETIS-R and PaToH-Q, and the solutions of hMETIS-K and PaToH-Q are not considered to be statistically significant. Hence, the results clearly indicate that it does not suffice to use the best solutions of repeated executions of a faster partitioner to achieve the same solution quality as *k*KaHyPar, *r*KaHyPar, or *k*KaHyPar-E.

In Figure 6.18 (right), we strengthen this argument for *k*KaHyPar by comparing the *best* solutions found by the competing algorithms while repeatedly partitioning each instance five times for eight hours to the solutions of only five *single* partitioning calls to *k*KaHyPar and *r*KaHyPar (i.e., the *first* results reported for each instance and seed during the eight hour time period). We see that, in this setting, the solution quality of PaToH-Q and hMETIS-R becomes comparable to the quality of *r*KaHyPar, while *k*KaHyPar still performs better than the other partitioning systems – computing the best solutions for around 50% of all benchmark instances. The results of the significance tests presented in Table 6.4 show that the performance difference between *k*KaHyPar and the other partitioners is still statistically significant. Note that we did

Table 6.3: Results of the significance tests for all pairwise algorithm comparisons (p-values) on benchmark set C. The value 0 is used to denote results that were below machine precision.

	(a)	(b)	(c)	(d)	(e)	(f)	(g)
(a) kKaHyPar	-	0	0	0	0	0	0
(b) rKaHyPar	0	-	0	0	0	0	0
(c) kKaHyPar-E	0	0	-	0	0	0	0
(d) hMETIS-R	0	0	0	-	0.608	0.882	0
(e) hMETIS-K	0	0	0	0.608	-	0.608	0
(f) PaToH-Q	0	0	0	0.882	0.608	-	0
(g) PaToH-D	0	0	0	0	0	0	-

Table 6.4: Results of the significance tests for all pairwise algorithm comparisons (p-values) on benchmark set C using only the very first results of kKaHyPar and rKaHyPar. The value 0 is used to denote results that were below machine precision.

	(a)	(b)	(c)	(d)	(e)	(f)
(a) kKaHyPar	-	0	0	0	0	0
(b) rKaHyPar	0	-	1	1	1	0
(c) hMETIS-R	0	1	-	0.418	1	0
(d) hMETIS-K	0	1	0.418	-	0.603	0
(e) PaToH-Q	0	1	1	0.603	-	0
(f) PaToH-D	0	0	0	0	0	-

not include kKaHyPar-E in this plot, because it has to create an initial population before performing recombination or mutation operations and thus takes considerably more time than kKaHyPar or rKaHyPar to report the first results.

6.4 Case Study: Graph Edge Partitioning

Motivation. Traditional (node-based) graph partitioning has been essential for making efficient distributed graph algorithms in the Think-Like-A-Vertex model of computation [MWM15]. In this model, node-centric operations are performed in parallel by mapping nodes to processing elements (PEs) and executing node computations in parallel. Nearly all algorithms in this model require information to be communicated between neighbors – which results in network communication if stored on different PEs – and therefore high-quality graph partitioning directly translates into less communication and faster overall running time. However, node-centric computations have serious shortcomings on power-law graphs, which have a skewed degree distribution. In such networks, the overall running time is negatively affected by very high-degree nodes, which can result in more communication steps. To combat these effects, Gonzalez et al. [Gon+12] introduced edge-centric computations,

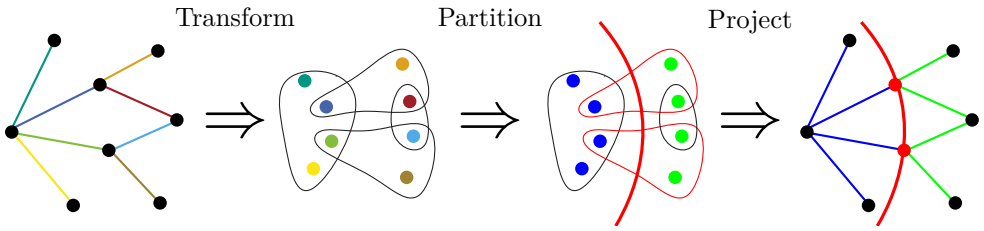


Figure 6.19: Computing an edge partition via hypergraph partitioning as proposed by Li et al. [Li+17]. After transforming the graph into a hypergraph and computing a connectivity-optimized hypergraph partition, the hypergraph solution induces a partition of the edge set of the graph.

which duplicate node-centric computations across edges to reduce communication overhead. In this model, *edge partitioning* – partitioning the edge set of a graph into roughly equally-sized blocks – must be used to reduce the overall running time. However, like node-based partitioning, edge partitioning is NP-hard [BLV14]. Since the problem can be solved directly via hypergraph partitioning, we use it as a case study to demonstrate the performance of KaHyPar on instances that were *never* used during the development or the tuning of the framework's algorithmic components.

The Edge Partitioning Problem. Let $G = (V, E, c, \omega)$ be an undirected, weighted graph. Similar to the node partitioning problem, the *edge partitioning problem* asks for blocks of edges $E_1, \ldots, E_k$ that partition E, i.e., $E_1 \cup \cdots \cup E_k = E$, $E_i \neq \emptyset$ for $1 \leq i \leq k$, and $E_i \cap E_j = \emptyset$ for $i \neq j$. The *balance constraint* demands that $\forall i \in \{1..k\} : \omega(E_i) \leq (1+\varepsilon)\lceil \frac{\omega(E)}{k} \rceil$. The objective is to minimize the *vertex cut* $\sum_{v \in V} |VC(v)| - 1$ where $VC(v) := \{i : \mathrm{I}(v) \cap E_i \neq \emptyset\}$. Intuitively, the objective expresses the number of required *replicas* of nodes: If a node v has to be copied to each block that has edges incident to v, the number of replicas of that node is $|VC(v)| - 1$.

Li, Geda, Hayes, Chen, Chaudhari, Zhang, and Szegedy [Li+17] noted that an edge partition of a graph G can be computed by transforming G into a hypergraph H, partitioning H into k blocks while optimizing the connectivity metric $f_\lambda(\Pi)$, and then using the hypergraph partition to infer an edge partition of G. The hypergraph contains a vertex for each graph edge $e \in E$, and a hyperedge for each graph node $v \in V$, which contains the graph edges to which the corresponding node is incident. An example of this approach is shown in Figure 6.19.

Methodology. In our conference paper [Sch+19a], we present a fast parallel auxiliary graph construction algorithm in the *distributed* setting, that – combined with advanced parallel node partitioning algorithms – yields high-quality edge partitions in a scalable way. For this dissertation, we restrict our focus to the part of the experimental evaluation presented in the paper that investigates the usage of *sequential* hypergraph partitioning algorithms to solve the edge partitioning problem directly. For additional

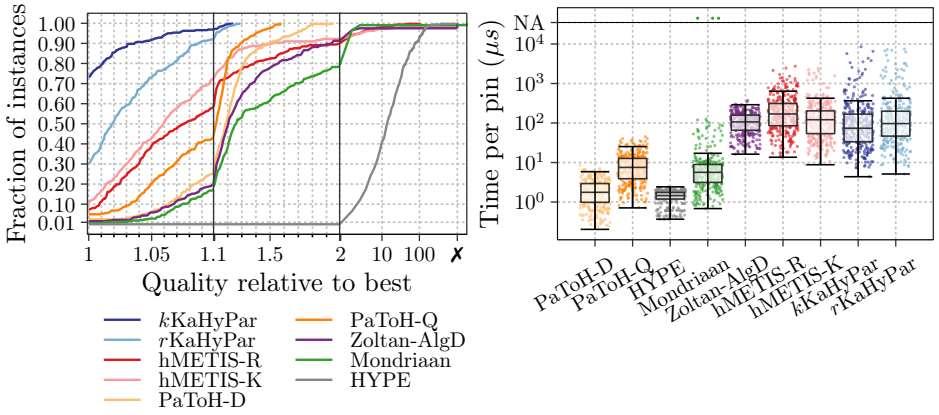


Figure 6.20: Comparing the results of all algorithms for the edge partitioning experiments on benchmark set E: solution quality (left) and running time (right).

comparisons with approaches based on partitioning auxiliary graphs and with dedicated edge partitioning algorithms, we refer the reader to the conference publication.

In the following experimental evaluation, we compare both *k*KaHyPar and *r*KaHyPar to hMETIS-R, hMETIS-K, PaToH-D, PaToH-Q, Zoltan-AlgD, Mondriaan, and HYPE. The experiments are performed on benchmark set E. We use an imbalance of $\varepsilon = 0.03$ and partition each hypergraph into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks. For each instance and each algorithm (except HYPE), we perform five repetitions with different seeds. As before, for HYPE, we report the results of one iteration using the default configuration, since employing randomization did not improve solution quality [HSS19a]. We use the results of hMETIS-R, hMETIS-K, PaToH-D, and Zoltan-Alg-D reported in the conference paper [Sch+19a] and perform new experiments for *k*KaHyPar and *r*KaHyPar, as well as for PaToH-Q, HYPE and Mondriaan (which were not evaluated in the paper).

Experimental Evaluation. The results of our experiments are summarized in Figure 6.20 and Figure 6.21. Considering the performance profile plot in Figure 6.20 (left), we see that out of all partitioning algorithms *k*KaHyPar again performs best – computing the best edge partitions on 73.3% of all benchmark instances. It is followed by *r*KaHyPar which computes the best solutions on 30.1% of all instances. *k*KaHyPar is never more than a factor of 1.24 worse than the best algorithm; *r*KaHyPar never more than a factor of 1.29. As can be seen in Figure 6.21, if each configuration is compared individually to all other partitioning systems, *k*KaHyPar computes the best solutions on 83.8%, and *r*KaHyPar on 68.9% of all instances. Interestingly, and in contrast to the results presented in Section 6.2, hMETIS-K performs better than hMETIS-R in this case study. This could be seen as an indication that, whenever hMETIS-K is able to compute feasible solutions, its performance could be comparable to that of hMETIS-R.

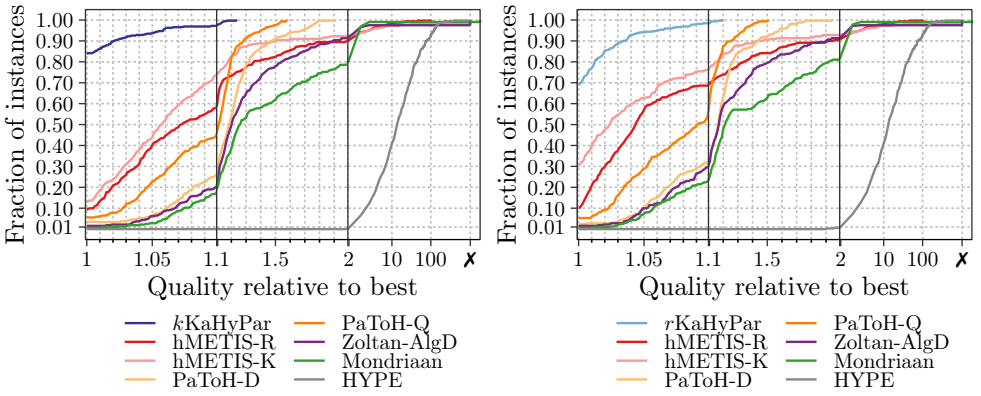


Figure 6.21: Results of the edge partitioning experiments on benchmark set E – comparing both KaHyPar configurations individually to the other partitioners.

Table 6.5: Results of the significance tests for all pairwise algorithm comparisons (p-values) for the edge partitioning experiments on benchmark set E. The value 0 is used to denote results that were below machine precision.

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)
(a) kKaHyPar	-	0.0135	0	0	0	0	0	0	0
(b) rKaHyPar	0.0135	-	0	7.93×10^{-10}	0	0	0	0	0
(c) hMETIS-R	0	0	-	0.01	0.46	0	0	0	0
(d) hMETIS-K	0	7.93×10^{-10}	0.01	-	0.000249	0	0	0	0
(e) PaToH-Q	0	0	0.46	0.000249	-	8.88×10^{-16}	0	0	0
(f) PaToH-D	0	0	0	0	8.88×10^{-16}	-	0.778	0.000351	0
(g) Zoltan-AlgD	0	0	0	0	0	0.778	-	0.000663	0
(h) Mondriaan	0	0	0	0	0	0.000351	0.000663	-	1.33×10^{-15}
(i) HYPE	0	0	0	0	0	0	0	1.33×10^{-15}	-

Table 6.5 summarizes the results of the pairwise significance tests. We note that indeed the performance difference between hMETIS-R and hMETIS-K is not statistically significant, as is the difference between hMETIS-R and PaToH-Q. However, the difference between hMETIS-K and PaToH-Q is statistically significant. Additionally, we see that given our significance level of $\alpha = 0.01$, there is no statistically significant performance difference between kKaHyPar and rKaHyPar for this benchmark set. Figure 6.20 (right) shows that the running times of both KaHyPar configurations are again comparable to that of hMETIS-K, hMETIS-R, and Zoltan-AlgD. As before, both PaToH configurations, as well as HYPE and Mondriaan are considerably faster than the other partitioning tools.

Concluding Remarks. The experimental results yield similar conclusions than the experiments on benchmark set A. Since benchmark set E was not used during the development of KaHyPar, this provides further evidence of our claim that KaHyPar can be seen as the new state of the art for high-quality hypergraph partitioning.

6.5 Case Study: Traditional Graph Partitioning

Motivation. Since hypergraph partitioning is a generalization of graph partitioning, we now compare kKaHyPar and rKaHyPar to the graph partitioner KaFFPa in order to evaluate how good our algorithms perform for traditional graph partitioning tasks.

Methodology. In this section, we use benchmark set F (consisting of web graphs and social networks) and benchmark set G (graphs used in the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering [Bad+13]). We refer to Section 2.6.1 for more details on both benchmark sets. For benchmark set F, we use the experimental results from Ref. [HSS19a] for kKaHyPar, KaFFPa-StrongS, and KaFFPa-StrongS* (a modified version that uses our improvements to KaFFPa's flow network as described in Section 4.7.1). Every graph was partitioned ten times with different seeds into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks at an imbalance of $\varepsilon = 0.03$. We perform additional experiments for rKaHyPar using the same setup and the same machine. In order to be consistent with the experiments presented in the previous sections, we impose a time limit of eight hours per instance for each algorithm.

The experiments on benchmark set G are new. We use the same values of k that have been used in the DIMACS challenge. Each instance is partitioned ten times with different seeds at an imbalance of $\varepsilon = 0.03$. For this benchmark set, we compute partitions using rKaHyPar, kKaHyPar, KaFFPa-Strong, KaFFPa-Strong* (using our improved flow network), and KaFFPa-StrongS*. As in the previous experiments, we set the time limit to eight hours per instance for each algorithm.

Effects of the Improved Flow Network. Before comparing KaFFPa with KaHyPar, we show that our improvements to KaFFPa's flow network as described in Section 4.7.1 indeed improve the performance of the graph partitioner. Figure 6.22 (left) compares KaFFPa-StrongS to our modified variant KaFFPA-StrongS* on the web graphs and social networks of benchmark set F, while Figure 6.22 (right) compares KaFFPa-Strong to our modified variant KaFFPa-Strong* on the DIMACS graphs of benchmark set G. We see that – in both settings – our improved variants often compute solutions of higher quality than the current version of KaFFPa. While this improvement is considered statistically significant ($p = 1.09 \times 10^{-7}$ on benchmark set F, $p = 4.27 \times 10^{-7}$ on benchmark set G), the performance profiles show that the difference in solution quality is within a few percent for almost all instances. This is expected, since all KaFFPa configurations use sophisticated FM-based local search algorithms in addition to the flow-based refinement algorithm. The running times shown in Figure 6.23 reveal that our improved versions are only marginally slower than the current version of KaFFPa. We therefore restrict the following evaluation to KaFFPa-Strong* and KaFFPa-StrongS*.

Comparison with KaHyPar – Complex Networks. The experimental results for the web graphs and social networks of benchmark set F are summarized in Figure 6.24 and Figure 6.25. We restrict the comparison to KaFFPa-StrongS*, since the `strongsocial` configuration is specifically designed for these types of graphs.

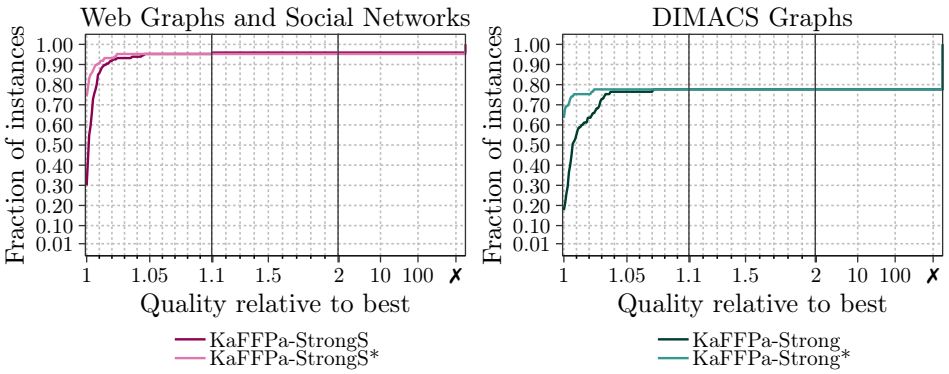


Figure 6.22: Performance profiles comparing KaFFPa-StrongS and KaFFPa-Strong with the variants that use our improved flow network (-Strong* and -StrongS*) on benchmark set F (left) and benchmark set G (right).

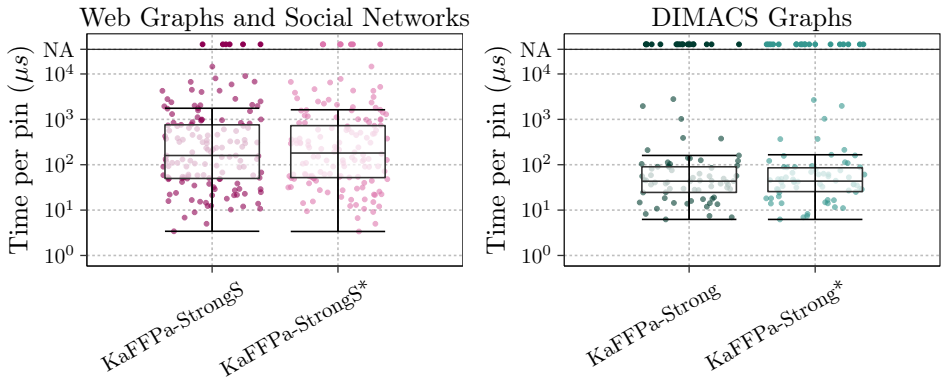


Figure 6.23: Comparing the running times of KaFFPa-StrongS and KaFFPa-Strong with the variants that use our improved flow network (-Strong* and -StrongS*) on benchmark set F (left) and benchmark set G (right).

Considering the performance profiles in Figure 6.24 (left), we see that kKaHyPar computes the best solutions for 57.8% of all instances (i.e., in 85 of 147 cases). It is followed by KaFFPa-StrongS* (31.9%), and rKaHyPar (12.2%). Figure 6.25 compares kKaHyPar and rKaHyPar individually to KaFFPa-StrongS*. We see that kKaHyPar computes the best solutions for more than 60% of all instances and is never more than a factor of 1.18 worse than KaFFPa-StrongS*. However, while kKaHyPar performs slightly better than KaFFPa-StrongS*, rKaHyPar is inferior to the graph partitioner. The results of the significance tests presented in Table 6.6 seem to validate these observations. Figure 6.24 (right) shows that KaHyPar is slightly faster on average than KaFFPa-StrongS*, which could not partition 7 instances within the time limit.

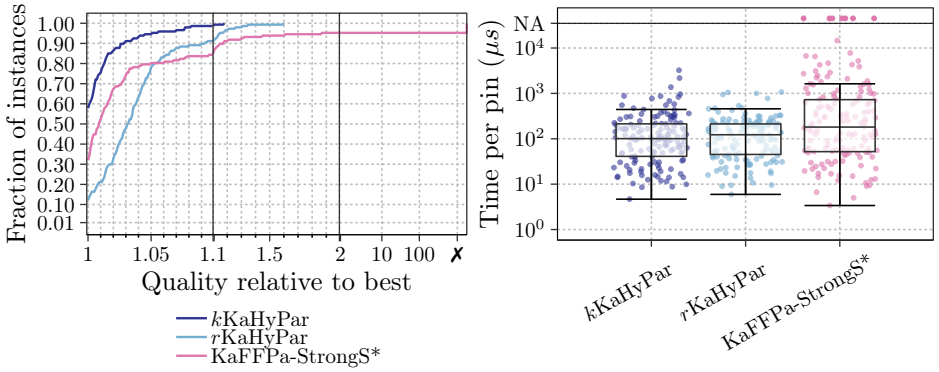


Figure 6.24: Comparing the results of kKaHyPar and rKaHyPar to KaFFPa-StrongS* on benchmark set F: solution quality (left) and running time (right).

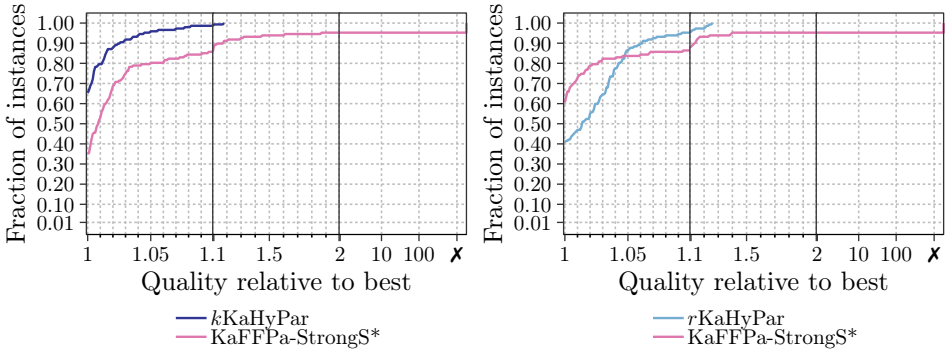


Figure 6.25: Performance profiles comparing kKaHyPar and rKaHyPar individually to KaFFPa-StrongS* on benchmark set F.

Comparison with KaHyPar – DIMACS Graphs. The results for benchmark set G are summarized in Figure 6.26 and Figure 6.27. The performance profiles depicted in Figure 6.26 (left) show that none of the algorithms were able to partition all instances – either because the partitioners could not finish within the time limit of eight hours, or because they could not allocate enough memory to perform the partitioning. Note that only rKaHyPar was able to compute partitions for more than 80% of all instances. KaFFPa-StrongS* computes the best solutions for 36.5% of all instances (i.e., in 31 out of the 85 cases), KaFFPa-Strong* for 32.9% of all instances, kKaHyPar and rKaHyPar compute the best solutions in 21.2% (resp. 9.4%) of all cases. Furthermore, we see that partitions of KaFFPa-Strong* are at most a factor of 1.05 worse than the partitions computed by the best algorithm. Figure 6.27 compares both KaHyPar configurations individually to the KaFFPa configurations. We see that while the performance of kKaHyPar is comparable to the performance of both

Table 6.6: Results of the significance tests for all pairwise algorithm comparisons (p-values) for the graph partitioning experiments on benchmark set F.

	(a)	(b)	(c)
(a) kKaHyPar	-	4.73×10^{-13}	0.000188
(b) rKaHyPar	4.73×10^{-13}	-	0.000267
(c) KaFFPa-StrongS*	0.000188	0.000267	-

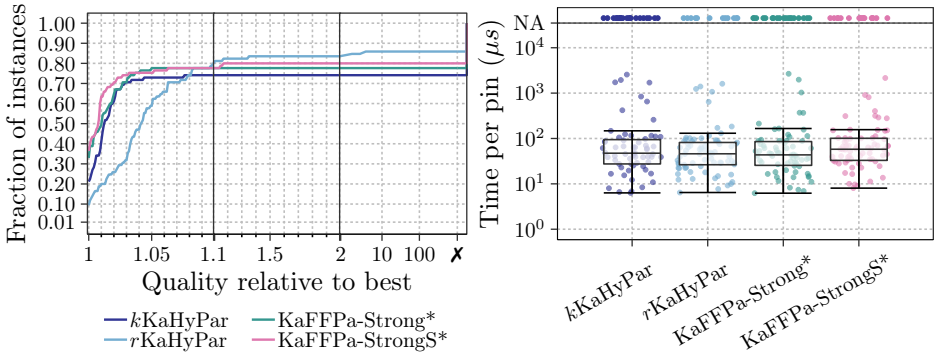


Figure 6.26: Comparing the results of kKaHyPar and rKaHyPar to KaFFPa-Strong* and KaFFPa-StrongS* on benchmark set G: solution quality (left) and running time (right).

KaFFPa configurations, rKaHyPar can be seen as inferior to the graph partitioning algorithms. Indeed, the results of the significance tests on the 56 instances that could be partitioned by all algorithms, shown in Table 6.7, only reveal a statistically significant difference between the solution quality of kKaHyPar and rKaHyPar, as well as between rKaHyPar and both KaFFPa configurations. Looking at Figure 6.26 (right), we see that the running times of all algorithms are comparable.

Concluding Remarks. Given these results, we conclude that kKaHyPar is also effective in the context of graph partitioning. In a comparison with the strongest KaFFPa configurations, it computes solutions of slightly higher quality for complex networks, and solutions of similar quality for the DIMACS graphs in a comparable amount of time.

Table 6.7: Results of the significance tests for all pairwise algorithm comparisons (p-values) for the graph partitioning experiments on benchmark set G.

	(a)	(b)	(c)	(d)
(a) kKaHyPar	-	6.04×10^{-5}	0.0634	0.0634
(b) rKaHyPar	6.04×10^{-5}	-	4.67×10^{-9}	5.59×10^{-10}
(c) KaFFPa-Strong*	0.0634	4.67×10^{-9}	-	0.661
(d) KaFFPa-StrongS*	0.0634	5.59×10^{-10}	0.661	-

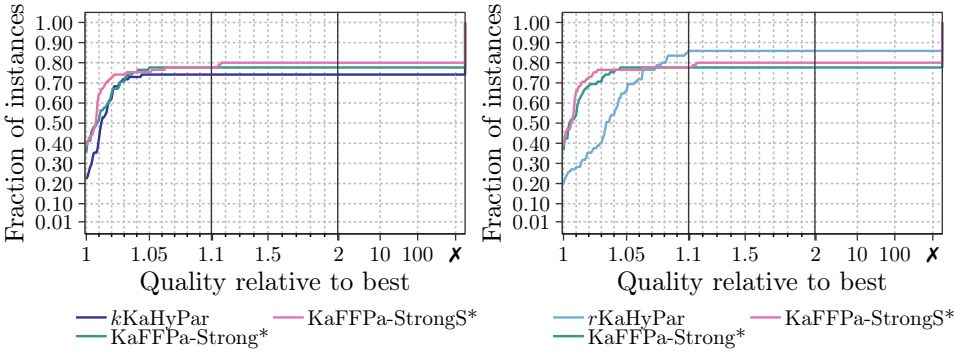


Figure 6.27: Performance profiles comparing kKaHyPar and rKaHyPar individually with KaFFPa-Strong* and KaFFPa-StrongS* on benchmark set G.

Chapter 7

7 Conclusion

"There is no real ending. It's just the place where you stop the story."

— Frank Herbert, California State College, Fullerton Interview (1969)

Having arrived at the final chapter of this dissertation, we briefly summarize the results in Section 7.1 and elaborate on possibilities for future work in Section 7.2.

7.1 Summary

With this dissertation, we presented several improvements to the multi-level paradigm for solving balanced hypergraph partitioning problems. After providing a comprehensive survey of almost 50 years of hypergraph partitioning history, we showed that the trade-off between solution quality and running time, inherent in the number of hierarchy levels, can be evaded by adopting an n-level approach and removing only a single vertex in every level. This was made feasible by specifically tailoring the hypergraph data structure, as well as the coarsening and refinement algorithms, to the n-level paradigm and by developing and adopting lazy-evaluation techniques, caching mechanisms, and early termination criteria to speed up the partitioning process. Compared to a naïve adaptation of the n-level approach used in KaSPar for traditional graph partitioning, our engineered hypergraph partitioning algorithms are more than two orders of magnitude faster.

Especially for hypergraphs with many large hyperedges, computations on the set of neighbors of a given vertex, which require iterating over all pins $p \in e$ of all incident nets $e \in \mathrm{I}(v)$ of a vertex v, can have a significant impact on the overall running time of a partitioning algorithm. To alleviate this impact, we proposed a pin sparsifier based on locality-sensitive hashing that identifies and contracts vertices with similar neighborhoods in a preprocessing phase prior to partitioning – reducing the number of pins and thus the average hyperedge size.

Moreover, we showed that traditional coarsening algorithms can be improved by incorporating global information about the community structure of the hypergraph into the coarsening process. Community detection is performed via modularity maximization using the Louvain algorithm on the bipartite graph representation. This approach is made suitable for a wide spectrum of instances by appropriately weighting the edges of the bipartite graph based on the edge density of the hypergraph.

In order to increase diversification during the initial partitioning phase, we presented a portfolio approach based on a large number of simple initial partitioning algorithms including random and BFS-based assignment, size-constrained label propagation, as well as several variants of greedy hypergraph growing.

To further improve the quality of the initial solution in the refinement phase, we presented two FM-based local search algorithms that perform a highly localized, gradually expanding search around a single uncontracted vertex pair. One algorithm is specifically tailored to the refinement of 2-way partitions and thus intended to be used in a recursive bipartitioning setting. The other algorithm represents the first FM-style direct k-way refinement heuristic that can be efficiently employed in the multi-level context. Additionally, we generalized the flow-based refinement framework of the graph partitioner KaFFPa from graphs to hypergraphs, identified shortcomings of the flow model employed in KaFFPa that unnecessarily restrict feasible solutions, and introduced an improved model that overcomes these limitations.

In order to explore the global solution space of possible partitions more extensively than using repeated runs with different seeds, we additionally embedded our algorithms into an evolutionary framework and presented the first memetic multi-level hypergraph partitioning algorithm.

All algorithmic contributions of this dissertation are made publicly available through the open-source Karlsruhe Hypergraph Partitioning (KaHyPar) system. KaHyPar supports both direct k-way partitioning and recursive bipartitioning, and is able to optimize the cut-net metric as well as the connectivity metric. Furthermore, it is designed as an extensible framework to foster the research and development of new partitioning heuristics. Since its release, it has already been employed in several research efforts [AH19; GLA19; Got19; Jal19b; Net19; PS19; Sch+19a; SSS19b] and has also attracted attention from industry.

Experiments demonstrating the effects of the different algorithmic components that make up KaHyPar were discussed in Chapter 4 and Chapter 5. In Chapter 6, we then presented a comprehensive experimental evaluation comparing the recursive bipartitioning algorithm rKaHyPar and the direct k-way algorithm kKaHyPar to the state-of-the-art hypergraph partitioning algorithms hMETIS-R, hMETIS-K, PaToH-D, PaToH-Q, Mondriaan, Zoltan-AlgD, and HYPE. We demonstrated that both KaHyPar configurations compute better solutions than all competing systems for *both* the cut-net metric and the connectivity metric on a wide range of benchmark hypergraphs. Although our system is based on the n-level paradigm and employs more complex techniques such as sparsification, community detection, and flow-based refinements, it was shown to be faster than Zoltan-AlgD and to have a running time comparable to that of hMETIS-R and hMETIS-K. It can thus be regarded as the new method of choice for computing *high-quality* solutions for hypergraph partitioning problems. In addition, our experimental study revealed that if speed is more important than solution quality, PaToH still provides the best time/quality trade-off of all tested algorithms – performing better than Mondriaan and HYPE.

In a setting where all algorithms were given the same fairly large amount of time to find a better partition through multiple runs, we were able to demonstrate that

our memetic algorithm effectively explores the global solution space – substantially outperforming the other competitors as well as the non-evolutionary KaHyPar configurations. Moreover, we showed that even if we compare single partitioning calls of kKaHyPar to the best solutions reported by competing algorithms after repeatedly partitioning each instance for several hours, kKaHyPar still performs better than its competitors. This highlights the fact that our improvements cannot simply be offset by repeatedly running a faster, simpler algorithm for a large amount of time. In a final evaluation, we demonstrated that KaHyPar is also effective for traditional graph partitioning tasks – computing slightly better solutions than our improved version of the current best system KaFFPa for complex networks and solutions of similar quality for the graphs of the 10th DIMACS implementation challenge in a comparable amount of time.

A result that presented itself in all experiments is the observation that – while outperforming competing algorithms by a significant margin – rKaHyPar seems to be inferior to kKaHyPar in general. Given that both algorithms employ the same or similar advanced techniques, this could be seen as supporting evidence for preferring direct k-way partitioning over recursive bipartitioning when aiming for high-quality solutions. Still, our experiments indicated that rKaHyPar can be of use for partitioning problems involving hypergraphs with many large nets, or on systems with a limited amount of main memory.

7.2 Outlook

Although the story of this dissertation ends with this section, there is indeed no real ending for research on hypergraph partitioning algorithms, which is why we would like to highlight several opportunities for future work.

As we have seen in Chapter 6, computing solutions of very high quality currently comes at the cost of considerably larger running times. Thus, parallelization is an important issue. Shared-memory algorithms for community detection already exist in the literature [SM16]. The community-aware coarsening approach presented in this dissertation restricts contractions to vertices within the same community. This already yields a promising path for coarse-grained, inter-community parallelization of the coarsening algorithms we presented. While the initial partitioning of the coarsest instance can be done independently in parallel, parallelizing the n-level FM-based local search algorithms or the flow-based refinement framework may pose a greater challenge, though the techniques proposed by Akhremtsev et al. [ASS17] for shared-memory graph partitioning constitute a viable starting point. However, parallelization should not be restricted to the shared-memory setting. Our experimental evaluation of distributed hypergraph partitioning algorithms in the context of graph edge partitioning [Sch+19a] revealed that while HGP algorithms outperform auxiliary-graph-based graph partitioning approaches in the sequential setting, the opposite currently seems to be the case in the distributed setting. This encourages further research into distributed hypergraph partitioning algorithms.

By incorporating global information about the community structure into the coarsening process, we prevented coarsening algorithms from performing contractions that potentially obscure naturally existing clustering structure. A recently published technical report by Sybrandt et al. [SSS19b] proposes to capture structural properties of the input hypergraph via graph embeddings computed on the bipartite hypergraph representation. This information is then used to prioritize the contraction of similar vertices (i.e., vertices with similar latent features). While the approach is currently too slow to be efficiently used in practice, the experimental results suggest that for small values of k, the embedding-based approach can lead to a significant improvement in solution quality. Hence, engineering this method to become feasible in terms of running time and making it more robust for larger values of k would be an interesting direction for future work.

Our historical overview of the field revealed that the research community (including us) mainly focused on unweighted benchmark instances. However, as shown by Caldwell et al. [CKM00d], the performance of move-based refinement heuristics can in some cases deteriorate significantly in the presence of non-uniformly weighted vertices. While multi-level algorithms in general try to keep vertex weights reasonably balanced during the coarsening phase, current measures may not be enough to allow standard initial partitioning algorithms to compute good, feasible starting solutions. Thus, integrating bin packing approaches into the portfolio of initial partitioning algorithms and augmenting refinement algorithms with additional rebalancing heuristics seems to be a promising approach to make multi-level systems more robust in practice. Moreover, our own work on community-aware coarsening also focused on hypergraphs with uniform net and vertex weights. In an upcoming bachelor thesis, we will therefore revisit weighted hypergraphs and study the robustness of preprocessing, coarsening, initial partitioning, and refinement algorithms for these instances.

Finally, with the increasing interest in distributed graph and hypergraph processing systems [Mal+10; Gon+12; Gon+14; HC14; HC15; HZY15; Jia+18; Hei+19], the trade-off between solution quality and running time/scalability currently seems to be explored almost exclusively with a focus on the latter by using rather simple hashing-based techniques or (hyper)graph growing heuristics [MS15; Pet+15; Zha+17; May+18; Han+19]. However, the experimental evaluation presented here, as well as our work on graph edge partitioning [Sch+19a], indicated that the increased partitioning speed of algorithms such as HYPE [May+18] or its graph-based predecessor NE [Zha+17] comes at the cost of a considerable decrease in solution quality. Thus, while this dissertation focused on computing high-quality partitions in a reasonable amount of time, a promising direction of future work is to shift the focus towards high-speed HGP systems with reasonable quality.

Appendix

List of Algorithms

List of Figures

List of Tables

List of Theorems

Bibliography

[Aba+02] C. Ababei, N. Selvakkumaran, K. Bazargan, and G. Karypis. „Multi-Objective Circuit Partitioning for Cutsize and Path-Based Delay Minimization“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 2002, pages 181–185. [see page 17]

[ACU08] C. Aykanat, B. B. Cambazoglu, and B. Uçar. „Multi-Level Direct k-Way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices“. In: *Journal of Parallel and Distributed Computing* 68.5 (2008), pages 609–625. [see pages 4, 6, 17, 83, 91, 95, 121, 124, 182]

[AD52] T. W. Anderson and D. A. Darling. „Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes“. In: *The Annals of Mathematical Statistics* 23.2 (June 1952), pages 193–212. [see page 31]

[Ady+04] S. N. Adya, M. C. Yildiz, I. L. Markov, P. G. Villarrubia, P. N. Parakh, and P. H. Madden. „Benchmarking for Large-Scale Placement and Beyond“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 23.4 (Apr. 2004), pages 472–487. [see page 45]

[AGL12] C. Ansótegui, J. Giráldez-Cru, and J. Levy. „The Community Structure of SAT Formulas“. In: *15th International Conference of Theory and Applications of Satisfiability Testing (SAT)*. Edited by A. Cimatti and R. Sebastiani. Springer, 2012, pages 410–423. [see page 148]

[AH19] P. Andres-Martinez and C. Heunen. „Automated distribution of quantum circuits via hypergraph partitioning“. In: *Physical Review A* 100 (3 Sept. 2019), page 032308. [see pages 6, 216]

[AHK96] C. J. Alpert, L. W. Hagen, and A. B. Kahng. „A Hybrid Multilevel/Genetic Approach for Circuit Partitioning“. In: *Asia Pacific Conference on Circuits and Systems (APCCAS)*. Nov. 1996, pages 298–301. [see page 70]

[AHK97] C. J. Alpert, J.-H. Huang, and A. B. Kahng. „Multilevel Circuit Partitioning“. In: *34th Conference on Design Automation (DAC)*. June 1997, pages 530–533. [see pages 36, 71, 73, 94, 98, 115]

[AHK98] C. J. Alpert, J.-H. Huang, and A. B. Kahng. „Multilevel Circuit Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 17.8 (1998), pages 655–667. [see pages 71, 73, 94, 98, 114, 115]

[AI08] A. Andoni and P. Indyk. „Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions“. In: *Communications of the ACM* 51.1 (2008), pages 117–122. [see page 107]

[AIV15] D. Alistarh, J. Iglesias, and M. Vojnovic. „Streaming Min-Max Hypergraph Partitioning“. In: *Neural Information Processing Systems (NIPS)*. Dec. 2015, pages 1900–1908. [see page 16]

[AK06] A. Abou-Rjeili and G. Karypis. „Multilevel Algorithms for Partitioning Power-Law Graphs“. In: *20th International Parallel and Distributed Processing Symposium (IPDPS)*. Apr. 2006. [see page 2]

[AK93] C. J. Alpert and A. B. Kahng. „Geometric Embeddings for Faster and Better Multi-Way Netlist Partitioning“. In: *30th Conference on Design Automation (DAC)*. 1993, pages 743–748. [see pages 51, 66, 88]

[AK94a] C. J. Alpert and A. B. Kahng. *A General Framework for Vertex Orderings, with Applications to Netlist Clustering*. Technical report 940018. UCLA CS Dept. Technical Report, 1994. [see page 52]

[AK94b] C. J. Alpert and A. B. Kahng. „A General Framework for Vertex Orderings, with Applications to Netlist Clustering“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1994, pages 63–67. [see page 52]

[AK94c] C. J. Alpert and A. B. Kahng. „Multi-Way Partitioning Via Spacefilling curves and Dynamic Programming“. In: *31st Conference on Design Automation (DAC)*. June 1994, pages 652–657. [see pages 20, 52, 53]

[AK95a] C. J. Alpert and A. B. Kahng. „Multiway Partitioning Via Geometric Embeddings, Orderings, and Dynamic Programming“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 14.11 (Nov. 1995), pages 1342–1358. [see pages 52, 53]

[AK95b] C. J. Alpert and A. B. Kahng. „Multiway Partitioning via Geometric Embeddings, Orderings, and Dynamic Programming“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 14.11 (1995), pages 1342–1358. [see page 20]

[AK95c] C. J. Alpert and A. B. Kahng. „Recent Directions in Netlist Partitioning: A Survey“. In: *Integration: The VLSI Journal* 19.1-2 (1995), pages 1–81. [see pages 2, 4, 14–16, 19, 20, 35, 37, 38, 40, 46, 112]

[AK96] C. J. Alpert and A. B. Kahng. „A General Framework for Vertex Orderings with Applications to Circuit Clustering“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4.2 (1996), pages 240–246. [see page 52]

[Akh+17a] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. „Engineering a Direct k-way Hypergraph Partitioning Algorithm“. In: *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Jan. 2017, pages 28–42. [see pages 4, 11, 23, 97–99, 103, 110, 118, 119, 126, 146, 147, 170, 182]

[Akh19] Y. Akhremtsev. „Parallel and External High Quality Graph Partitioning“. PhD thesis. Karlsruhe Institute of Technology, 2019. [see pages 147, 163]

[AKY99] C. J. Alpert, A. B. Kahng, and S.-Z. Yao. „Spectral Partitioning with Multiple Eigenvectors“. In: *Discrete Applied Mathematics* 90.1-3 (1999), pages 3–26. [see pages 20, 53]

[AL08] R. Andersen and K. J. Lang. „An Algorithm for Improving Graph Partitions“. In: *19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2008, pages 651–660. [see page 136]

[Alp+00] C. J. Alpert, A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Hypergraph Partitioning with Fixed Vertices [VLSI CAD]“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 19.2 (2000), pages 267–272. [see page 59]

[Alp+05] C. J. Alpert, A. B. Kahng, G.-J. Nam, S. Reda, and P. Villarrubia. „A Semi-Persistent Clustering Technique for VLSI Circuit Placement“. In: *International Symposium on Physical Design (ISPD)*. Apr. 2005, pages 200–207. [see pages 117, 153]

[Alp+06] C. J. Alpert, A. B. Kahng, G.-J. Nam, S. Reda, and P. Villarrubia. „A Fast Hierarchical Quadratic Placement Algorithm“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 25.4 (2006), pages 678–691. [see page 117]

[Alp+99] C. J. Alpert, A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Partitioning with Terminals: A "new" Problem and New Benchmarks“. In: *International Symposium on Physical Design (ISPD)*. Apr. 1999, pages 151–157. [see page 40]

[Alp96] C. J. Alpert. „Multi-Way Graph and Hypergraph Partitioning“. PhD thesis. University of California, Los Angeles, 1996. [see pages 4, 14, 37, 40, 72, 98]

[Alp98] C. J. Alpert. „The ISPD98 Circuit Benchmark Suite“. In: *International Symposium on Physical Design (ISPD)*. Apr. 1998, pages 80–85. [see pages 23, 45, 60]

[Ama+18] F. Amato, V. Moscato, A. Picariello, F. Piccialli, and G. Sperli. „Centrality in heterogeneous social networks for lurkers detection: An approach based on hypergraphs“. In: *Concurrency and Computation: Practice and Experience* 30.3 (June 2018). [see page 3]

[And17a] R. Andre. „Evolutionary Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, Nov. 2017. [see page 171]

[Ang+19] E. Angriman, A. van der Grinten, M. von Looz, H. Meyerhenke, M. Nöllenburg, M. Predari, and C. Tzovas. „Guidelines for Experimental Algorithmics: A Case Study in Network Analysis“. In: *Algorithms* 12.7 (2019), page 127. [see page 7]

[Ans73] F. J. Anscombe. „Graphs in Statistical Analysis“. In: *The American Statistician* 27.1 (1973), pages 17–21. [see pages 29, 30]

[Ant+19] A. Antelmi, G. Cordasco, B. Kaminski, P. Pralat, V. Scarano, C. Spagnuolo, and P. Szufel. „SimpleHypergraphs.jl – Novel Software Framework for Modelling and Analysis of Hypergraphs“. In: *16th International Workshop Algorithms and Models for the Web Graph (WAW)*. July 2019, pages 115–129. [see page 3]

[AR04] K. Andreev and H. Räcke. „Balanced Graph Partitioning“. In: *16th Symposium on Parallelism in Algorithms and Architectures (SPAA)*. June 2004, pages 120–124. [see page 19]

[AR06] K. Andreev and H. Räcke. „Balanced Graph Partitioning“. In: *Theory of Computing Systems* 39.6 (Oct. 2006), pages 929–939. [see page 19]

[Are00a] S. Areibi. „An integrated genetic algorithm with dynamic hill climbing for VLSI circuit partitioning“. In: *GECCO 2000*. July 2000, pages 97–102. [see pages 61, 171, 172]

[Are00b] S. Areibi. „Simple Yet Effective Techniques to Improve Flat Multiway Circuit Partitioning“. In: *Canadian Conference on Electrical and Computer Engineering (CCECE)*. May 2000, pages 394–398. [see pages 61, 80]

[Are01] S. Areibi. „Recursive and Flat Partitioning for VLSI Circuit Design“. In: *13th International Conference on Microelectronics (ICM)*. Oct. 2001, pages 237–240. [see page 96]

[Are99] S. Areibi. „GRASP: an effective constructive technique for VLSI circuit partitioning“. In: *Canadian Conference on Electrical and Computer Engineering (CCECE)*. May 1999, pages 462–467. [see pages 58, 64]

[Arm+10] E. Armstrong, G. W. Grewal, S. Areibi, and G. Darlington. „An investigation of parallel memetic algorithms for VLSI circuit partitioning on multi-core computers“. In: *Canadian Conference on Electrical and Computer Engineering (CCECE)*. 2010, pages 1–6. [see pages 61, 171, 172]

[Ash95] C. Ashcraft. „Compressed Graphs and the Minimum Degree Algorithm“. In: *SIAM Journal on Scientific Computing* 16.6 (1995), pages 1404–1411. [see page 85]

[ASS17] Y. Akhremtsev, P. Sanders, and C. Schulz. „High-Quality Shared-Memory Graph Partitioning“. In: *European Conference on Parallel Processing (Euro-Par)*. Springer, Aug. 2017, pages 659–671. [see pages 163, 217]

[ASS18a] R. Andre, S. Schlag, and C. Schulz. „Memetic Multilevel Hypergraph Partitioning“. In: *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, July 2018, pages 347–354. [see pages 4, 11, 24, 26, 171, 173, 175, 181, 204]

[ASS18b] R. Andre, S. Schlag, and C. Schulz. „Memetic Multilevel Hypergraph Partitioning“. In: *Computing Research Repository (CoRR)* (Feb. 2018), pages 1–16. arXiv: 1710.01968. [see pages 4, 171]

[AV00] S. Areibi and A. Vannelli. „Tabu search: A meta heuristic for netlist partitioning“. In: *VLSI Design* 11.3 (2000), pages 259–283. [see pages 36, 50]

[AV93a] S. Areibi and A. Vannelli. „A Combined Eigenvector Tabu Search Approach for Circuit Partitioning“. In: *Custom Integrated Circuits Conference (CICC)*. May 1993, pages 9.7.1–9.7.4. [see page 50]

[AV93b] S. Areibi and A. Vannelli. „Advanced Search Techniques for Circuit Partitioning“. In: *Quadratic Assignment and Related Problems, Proceedings of a DIMACS Workshop*. May 1993, pages 77–98. [see page 70]

[AV93c] S. Areibi and A. Vannelli. „Circuit Partitioning Using a Tabu Search Approach“. In: *International Symposium on Circuits and Systems (ISCAS)*. May 1993, pages 1643–1646. [see page 50]

[AV96] S. Areibi and A. Vannelli. „An Efficient Clustering Technique for Circuit Partitioning“. In: *International Symposium on Circuits and Systems (ISCAS)*. Volume 4. IEEE. 1996, pages 671–674. [see page 69]

[AY04] S. Areibi and Z. Yang. „Effective Memetic Algorithms for VLSI Design = Genetic Algorithms + Local Search + Multi-Level Clustering“. In: *Evolutionary Computation* 12.3 (2004), pages 327–353. [see pages 80, 88, 171, 172]

[AY94] C. J. Alpert and S.-Z. Yao. *Spectral Partitioning: The More Eigenvectors, The Better*. Technical report 940036. UCLA CS Dept. Technical Report, 1994. [see page 53]

[AY95] C. J. Alpert and S.-Z. Yao. „Spectral Partitioning: The More Eigenvectors, The Better“. In: *32nd Conference on Design Automation (DAC)*. June 1995, pages 195–200. [see page 53]

[Baa+19a] I. Baar, L. Hübner, P. Oettig, A. Zapletal, S. Schlag, A. Stamatakis, and B. Morel. „Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning“. In: *18th International Workshop on High Performance Computational Biology (HiCOMB)*. IEEE, 2019. [see pages 11, 17]

[Bäc96] T. Bäck. *Evolutionary Algorithms in Theory and Practice – Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996. [see pages 22, 172]

[Bad+13] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*. Volume 588. Contemporary Mathematics. American Mathematical Society, Feb. 2013. [see pages 7, 24, 27, 84, 210]

[Bar07] M. J. Barber. „Modularity and Community Detection in Bipartite Networks“. In: *Physical Review* 76 (6 Dec. 2007). [see page 111]

[Bar82] E. Barnes. „An Algorithm for Partitioning the Nodes of a Graph“. In: *SIAM Journal on Algebraic Discrete Methods* 3.4 (1982), pages 541–550. [see pages 20, 50]

[BBR97] R. Battiti, A. A. Bertossi, and R. Rizzi. „Randomized Greedy Algorithms for the Hypergraph Partitioning Problem“. In: *Randomization Methods in Algorithm Design (DIMACS Workshop)*. Dec. 1997, pages 21–36. [see page 56]

[BCM16] A. Benavoli, G. Corani, and F. Mangili. „Should We Really Use Post-Hoc Tests Based on Mean-Ranks?“ In: *Journal of Machine Learning Research* 17 (2016), pages 1–10. [see page 34]

[BD19] D. Berrar and W. Dubitzky. „Should significance testing be abandoned in machine learning?“ In: *International Journal of Data Science and Analytics* 7.4 (June 2019), pages 247–257. [see page 34]

[Bel+14] A. Belov, D. Diepold, M. Heule, and M. Järvisalo. *The SAT Competition 2014*. `http://www.satcompetition.org/2014/`. 2014. [see page 23]

[Ber75] C. Berge. „Isomorphism Problems for Hypergraphs“. In: *Combinatorics*. Edited by M. Hall Jr. and J. H. van Lint. Volume 16. Springer, 1975, pages 205–214. [see page 12]

[Ber85] C. Berge. *Graphs and Hypergraphs*. Elsevier, 1985. [see page 12]

[BH88] B. Bergmann and G. Hommel. „Improvements of General Multiple Test Procedures for Redundant Systems of Hypotheses“. In: *Multiple Hypothesenprüfung / Multiple Hypotheses Testing*. Edited by P. Bauer, G. Hommel, and E. Sonnemann. Springer, 1988, pages 100–115. [see page 34]

[Bin14] T. Bingmann. *SqlPlotTools – Gnuplot and Pgfplots from SQL Statements*. 2014. URL: `http://panthema.net/2014/sqlplot-tools/`. [see page 9]

[Bin18] T. Bingmann. „Scalable String and Suffix Sorting – Algorithms, Techniques, and Tools“. PhD thesis. Karlsruhe Institute of Technology, July 2018, pages 1–396. [see page 8]

[Bis+12] R. H. Bisseling, B. O. Auer Fagginger, A. N. Yzelman, T. van Leeuwen, and Ü. V. Çatalyürek. „Two-Dimensional Approaches to Sparse Matrix Partitioning“. In: *Combinatorial Scientific Computing* (2012), pages 321–349. [see pages 3, 184]

[Bis+19] R. Bisseling, B. Fagginger Auer, T. van Leeuwen, W. Meesen, M. van Oort, D. Pelt, B. Vastenhouw, and A.-J. Yzelman. *Mondriaan for Sparse Matrix Partitioning*. 2019. URL: `https://www.staff.science.uu.nl/~bisse101/Mondriaan/`. [see page 182]

[BJ92] T. N. Bui and C. Jones. „Finding Good Approximate Vertex and Edge Partitions is NP-Hard“. In: *Information Processing Letters* 42.3 (May 1992), pages 153–159. [see page 3]

[BJ93] T. N. Bui and C. Jones. „A Heuristic for Reducing Fill-In in Sparse Matrix Factorization“. In: *6th SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*. 1993, pages 445–452. [see pages 20, 37]

[BK04] Y. Boykov and V. Kolmogorov. „An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision“. In: *Transactions on Pattern Analysis and Machine Intelligence* 26.9 (2004), pages 1124–1137. [see pages 157, 158]

[Blo+08] V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. „Fast Unfolding of Communities in Large Networks“. In: *Journal of Statistical Mechanics: Theory and Experiment* 10 (2008). [see pages 5, 111]

[BLV14] F. Bourse, M. Lelarge, and M. Vojnovic. „Balanced Graph Edge Partition“. In: *20th International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, Aug. 2014, pages 1456–1465. [see page 207]

[BM94] T. N. Bui and B. R. Moon. „A Fast and Stable Hybrid Genetic Algorithm for the Ratio-Cut Partitioning Problem on Hypergraphs“. In: *31st Conference on Design Automation (DAC)*. IEEE, June 1994, pages 664–669. [see pages 36, 51, 171, 172, 175]

[BM98] T. N. Bui and B. R. Moon. „GRCA: A Hybrid Genetic Algorithm for Circuit Ratio-Cut Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 17.3 (1998), pages 193–204. [see pages 36, 51]

[Bom+12a] E. G. Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine. „The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering and Coloring“. In: *Scientific Programming* 20.2 (2012), pages 129–150. [see page 81]

[Bom+12b] E. Boman, K. Devine, V. Leung, S. Rajamanickam, L. A. Riesen, and Ü. V. Çatalyürek. *Zoltan User's Guide*. `http://www.cs.sandia.gov/Zoltan/ug_html/ug_alg_patoh.html`. 2012. [see page 182]

[BR03] C. Blum and A. Roli. „Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison“. In: *ACM Computing Surveys* 35.3 (2003), pages 268–308. [see page 21]

[Bra+08] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. „On Modularity Clustering“. In: *IEEE Transactions on Knowledge and Data Engineering* 20.2 (2008), pages 172–188. [see page 111]

[Bre13] A. Bretto. *Hypergraph Theory: An Introduction*. Springer, 2013. [see page 14]

[Brg93] F. Brglez. „A D&T Special Report on ACM/SIGDA Design Automation Benchmarks: Catalyst or Anathema?“ In: *IEEE Design & Test of Computers* 10.3 (July 1993), pages 87–91. [see page 60]

[Bro+00] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. „Min-Wise Independent Permutations“. In: volume 60. 3. Elsevier, June 2000, pages 630–659. [see page 107]

[Bro+97] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. „Syntactic Clustering of the Web“. In: *Computer Networks* 29.8-13 (1997), pages 1157–1166. [see page 107]

[Bro97] A. Z. Broder. „On the Resemblance and Containment of Documents“. In: *Compression and Complexity of Sequences*. IEEE, 1997. [see pages 85, 107]

[BS02] B. Bollobás and A. D. Scott. „Problems and Results on Judicious Partitions“. In: *Random Structures and Algorithms* 21.3-4 (2002), pages 414–430. [see page 16]

[BS11] C. Bichot and P. Siarry, editors. *Graph Partitioning*. Wiley, 2011. [see pages 1–3, 6, 12, 15, 20, 40, 94, 96]

[BS93] S. T. Barnard and H. D. Simon. „A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems“. In: *6th SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*. Mar. 1993, pages 711–718. [see pages 20, 37]

[BS94] S. T. Barnard and H. D. Simon. „Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems“. In: *Concurrency - Practice and Experience* 6.2 (1994), pages 101–117. [see page 20]

[BS97] B. Bollobás and A. D. Scott. „Judicious Partitions of Hypergraphs“. In: *Journal of Combinational Theory* 78.1 (1997), pages 15–31. [see page 16]

[BT93] P. Briggs and L. Torczon. „An Efficient Representation for Sparse Sets“. In: *ACM Letters on Programming Languages and Systems* 2.1-4 (1993), pages 59–69. [see page 133]

[BT96] T. Blickle and L. Thiele. „A Comparison of Selection Schemes used in Evolutionary Algorithms“. In: *Evolutionary Computation* 4.4 (1996), pages 361–394. [see page 173]

[Bui+87] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser. „Graph Bisection Algorithms with Good Average Case Behavior“. In: *Combinatorica* 7.2 (1987), pages 171–191. [see pages 36, 66, 95]

[Bui+89] T. N. Bui, C. Heigham, C. Jones, and F. T. Leighton. „Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms“. In: *26th Conference on Design Automation (DAC)*. June 1989, pages 775–778. [see pages 36, 66, 69, 95]

[Bul+16] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. „Recent Advances in Graph Partitioning“. In: *Algorithm Engineering - Selected Results and Surveys*. 2016, pages 117–158. [see pages 1, 4, 15, 37]

[Bun+97] W. L. Buntine, L. Su, A. R. Newton, and A. Mayer. „Adaptive Methods for Netlist Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1997, pages 356–363. [see pages 56, 57, 96]

[Buu16] J.-W. Buurlage. „Self-Improving Sparse Matrix Partitioning and Bulk-Synchronous Pseudo-Streaming“. Master Thesis. Utrecht University, Netherlands, 2016. [see page 62]

[ÇA01a] Ü. V. Çatalyürek and C. Aykanat. „A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices“. In: *15th International Parallel and Distributed Processing Symposium (IPDPS)*. Apr. 2001, page 118. [see page 3]

[ÇA01b] Ü. V. Çatalyürek and C. Aykanat. „A Hypergraph-Partitioning Approach for Coarse-Grain Decomposition“. In: *ACM/IEEE Conference on Supercomputing*. Nov. 2001, page 28. [see pages 3, 17]

[ÇA11a] Ü. V. Çatalyürek and C. Aykanat. „PaToH (Partitioning Tool for Hypergraphs)“. In: *Encyclopedia of Parallel Computing*. 2011, pages 1479–1487. [see page 91]

[ÇA11b] Ü. V. Çatalyürek and C. Aykanat. *PaToH: Partitioning Tool for Hypergraphs*. `https://www.cc.gatech.edu/~umit/PaToH/manual.pdf`. 2011. [see pages 91, 106, 114, 120]

[ÇA99] Ü. V. Çatalyürek and Cevdet Aykanat. „Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication“. In: *IEEE Transactions on Parallel and Distributed Systems* 10.7 (1999), pages 673–693. [see pages 5, 7, 23, 24, 75, 81, 82, 85, 95, 96, 114, 120, 122]

[Cai16] A. Cairo. *Download the Datasaurus: Never trust summary statistics alone; always visualize your data*. `http://www.thefunctionalart.com/2016/08/download-datasaurus-never-trust-summary.html`. 2016. [see pages 29, 30]

[Cal+99] A. E. Caldwell, A. B. Kahng, A. A. Kennings, and I. L. Markov. „Hypergraph Partitioning for VLSI CAD: Methodology for Heuristic Development, Experimentation and Reporting“. In: *36th Conference on Design Automation (DAC)*. June 1999, pages 349–354. [see pages 23, 44, 45]

[Çat] Ü. V. Çatalyürek. *ISPD98 Benchmark*. `http://bmi.osu.edu/umit/PaToH/ispd98.html`. [see page 182]

[Çat+07] Ü. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdag, R. T. Heaphy, and L. A. Riesen. „Hypergraph-Based Dynamic Load Balancing for Adaptive Scientific Computations“. In: *21st International Parallel and Distributed Processing Symposium (IPDPS)*. Mar. 2007, pages 1–11. [see page 83]

[Çat+09] Ü. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdag, R. T. Heaphy, and L. A. Riesen. „A Repartitioning Hypergraph Model for Dynamic Load Balancing“. In: *Journal of Parallel and Distributed Computing* 69.8 (2009), pages 711–724. [see page 83]

[Çat+12a] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. „Multithreaded Clustering for Multi-level Hypergraph Partitioning“. In: *26th International Parallel and Distributed Processing Symposium (IPDPS)*. May 2012, pages 848–859. [see page 85]

[Çat+12b] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. „UMPa: A Multi-Objective, Multi-Level Partitioner for Communication Minimization“. In: *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*. Feb. 2012, pages 53–66. [see pages 6, 17, 84, 124]

[Çat+15] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. „Hypergraph Partitioning for Multiple Communication Cost Metrics: Model and Methods“. In: *Journal of Parallel and Distributed Computing* 77 (2015), pages 69–83. [see pages 14, 17, 84, 121, 182, 187]

[Çat19] Ü. V. Çatalyürek. *PaToH (Partitioning Tools for Hypergraph)*. 2019. URL: `https://www.cc.gatech.edu/~umit/software.html`. [see page 182]

[Çat99] Ü. V. Çatalyürek. „Hypergraph Models for Sparse Matrix Partitioning and Reordering“. PhD thesis. Bilkent University, 1999. [see pages 3, 4, 36, 37, 76]

[ÇAU10] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar. „On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe“. In: *SIAM Journal on Scientific Computing* 32.2 (2010), pages 656–683. [see page 3]

[CC00] S.-J. Chen and C.-K. Cheng. „Tutorial on VLSI partitioning“. In: *VLSI design* 11.3 (2000), pages 175–218. [see pages 4, 37]

[CC03] J.-S. Cherng and S.-J. Chen. „An Efficient Multi-Level Partitioning Algorithm for VLSI Circuits“. In: *16th International Conference on VLSI Design (VLSID)*. Jan. 2003, page 70. [see page 78]

[CC96] J.-S. Cherng and S.-J. Chen. „A Stable Partitioning Algorithm for VLSI Circuits“. In: *Custom Integrated Circuits Conference (CICC)*. 1996, pages 163–166. [see pages 69, 74, 78, 88]

[CCH98] J.-S. Cherng, S.-J. Chen, and J.-M. Ho. „Efficient Bipartitioning Algorithm for Size-Constrained Circuits“. In: *International Conference on Computer Design VLSI in Computers and Processors (ICCD)* 145 (1 Jan. 1998), pages 37–45. [see pages 36, 57, 69, 74, 78]

[CH90] A. Chatterjee and R. I. Hartley. „A New Simultaneous Circuit Partitioning and Chip Placement Approach Based on Simulated Annealing“. In: *27th Conference on Design Automation (DAC)*. June 1990, pages 36–39. [see page 37]

[Cha02] M. Charikar. „Similarity Estimation Techniques from Rounding Algorithms“. In: *34th ACM Symposium on Theory of Computing (STOC)*. ACM, May 2002, pages 380–388. [see page 106]

[Che+99] J.-S. Cherng, S.-J. Chen, C.-C. Tsai, and J.-M. Ho. „An Efficient Two-Level Partitioning Algorithm for VLSI Circuits“. In: *Asia South Pacific Design Automation Conference (ASP-DAC)*. Jan. 1999, pages 69–72. [see pages 74, 88]

[Chi+15] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. „One Trillion Edges: Graph Processing at Facebook-Scale“. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pages 1804–1815. [see page 3]

[CHK92] J. Cong, L. W. Hagen, and A. B. Kahng. „Net Partitions Yield Better Module Partitions“. In: *29th Conference on Design Automation (DAC)*. June 1992, pages 47–52. [see pages 35, 50, 52, 84]

[Cho+94] N.-C. Chou, L.-T. Liu, C.-K. Cheng, W.-J. Dai, and R. Lindelof. „Circuit Partitioning for Huge Logic Emulation Systems“. In: *31st Conference on Design Automation (DAC)*. June 1994, pages 244–249. [see page 37]

[CKL03] J. Cohoon, J. Kairo, and J. Lienig. „Evolutionary Algorithms for the Physical Design of VLSI Circuits“. In: *Advances in Evolutionary Computing: Theory and Applications*. Springer, 2003, pages 683–711. [see pages 11, 171]

[CKM00a] A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Can Recursive Bisection Alone Produce Routable Placements?“ In: *37th Conference on Design Automation (DAC)*. June 2000, pages 477–482. [see pages 20, 37]

[CKM00b] A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Design and Implementation of Move-Based Heuristics for VLSI Hypergraph Partitioning“. In: *ACM Journal of Experimental Algorithmics (JEA)* 5 (2000). [see pages 9, 11, 44, 45]

[CKM00c] A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Improved Algorithms for Hypergraph Bipartitioning“. In: *Asia South Pacific Design Automation Conference (ASP-DAC)*. 2000, pages 661–666. [see pages 9, 59, 77, 78, 91, 182]

[CKM00d] A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Iterative Partitioning with Varying Node Weights“. In: *VLSI Design* 3 (2000), pages 249–258. [see pages 42, 59, 60, 218]

[CKM00e] A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Optimal Partitioners and End-Case Placers for Standard-Cell Layout“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 19.11 (2000), pages 1304–1313. [see pages 58, 59, 94]

[CKM99a] A. E. Caldwell, A. B. Kahng, and I. L. Markov. *Design and implementation of move-based partitioners*. Technical report 990015. UCLA Computer Science Department, 1999. [see page 45]

[CKM99b] A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning“. In: *1st Workshop on Algorithm Engineering & Experiments (ALENEX)*. 1999, pages 177–193. [see pages 9, 45, 91]

[CKM99c] A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Hypergraph Partitioning with Fixed Vertices“. In: *36th Conference on Design Automation (DAC)*. June 1999, pages 355–359. [see page 59]

[CKM99d] A. E. Caldwell, A. B. Kahng, and I. L. Markov. „Optimal Partitioners and End-Case Placers for Standard-Cell Layout“. In: *International Symposium on Physical Design (ISPD)*. Apr. 1999, pages 90–96. [see pages 58, 59]

[CL00a] J. Cong and S. K. Lim. „Edge Separability Based Circuit Clustering with Application to Circuit Partitioning“. In: *Asia South Pacific Design Automation Conference (ASP-DAC)*. 2000, pages 429–434. [see page 78]

[CL00b] J. Cong and S. K. Lim. „Performance Driven Multiway Partitioning“. In: *Asia South Pacific Design Automation Conference (ASP-DAC)*. 2000, pages 441–446. [see page 37]

[CL04] J. Cong and S. K. Lim. „Edge Separability-Based Circuit Clustering with Application to Multilevel Circuit Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 23.3 (2004), pages 346–357. [see pages 78, 148]

[CL15] C. Chekuri and S. Li. *A Note on the Hardness of Approximating the k-Way Hypergraph Cut Problem*. `http://chekuri.cs.illinois.edu/papers/hypergraph-kcut.pdf`. 2015. [see page 18]

[CL98] J. Cong and S. K. Lim. „Multiway Partitioning with Pairwise Movement“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1998, pages 512–516. [see pages 36, 57, 58, 70, 78, 95, 96, 124]

[CL99] J. Cong and S. K. Lim. *Performance Driven Multiway Partitioning*. Technical report 990032. CS Department of UCLA, 1999. [see page 37]

[CLS94] J. Cong, W. J. Labio, and N. Shivakumar. „Multi-Way VLSI Circuit Partitioning Based on Dual Net Representation“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1994, pages 56–62. [see pages 36, 52, 84]

[CLS96] J. Cong, W. J. Labio, and N. Shivakumar. „Multiway VLSI Circuit Partitioning Based on Dual Net Representation“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 15.4 (Apr. 1996), pages 396–409. [see pages 52, 84]

[CLW00] J. Cong, S. K. Lim, and C. Wu. „Performance Driven Multi-Level and Multiway Partitioning with Retiming“. In: *37th Conference on Design Automation (DAC)*. June 2000, pages 274–279. [see page 37]

[CLW03] Y. Cheon, S. Lee, and M. D. F. Wong. „Stable Multiway Circuit Partitioning for ECO“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 2003, pages 718–725. [see page 17]

[CMR90] J. P. Cohoon, W. N. Martin, and D. S. Richards. „Genetic Algorithms and Punctuated Equilibria in VLSI“. In: *1st Parallel Problem Solving from Nature Workshop (PPSN)*. 1990, pages 134–144. [see page 37]

[Con+97a] J. Cong, H. P. Li, S. K. Lim, T. Shibuya, and D. Xu. „Large Scale Circuit Partitioning with Loose/Stable Net Removal and Signal Flow Based Clustering“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1997, pages 441–446. [see pages 36, 55, 58, 61, 77, 78]

[Con+97b] J. Cong, H. P. Li, S. K. Lim, T. Shibuya, and D. Xu. *Large Scale Circuit Partitioning with Loose/Stable Net Removal and Signal Flow Based Hierarchical Clustering*. Technical report 970005. CS Department of UCLA, 1997. [see pages 55, 58, 61, 77]

[CP68] H. R. Charney and D. L. Plato. „Efficient Partitioning of Components“. In: *5th Annual Design Automation Workshop*. ACM, 1968, pages 16.1–16.21. [see page 14]

[CQX19] C. Chekuri, K. Quanrud, and C. Xu. „LP Relaxation and Tree Packing for Minimum k-Cuts“. In: *2nd Symposium on Simplicity in Algorithms (SOSA @ SODA)*. Jan. 2019, 7:1–7:18. [see page 18]

[CRX03] J. Cong, M. Romesis, and M. Xie. „Optimality, Scalability and Stability Study of Partitioning and Placement Algorithms“. In: *International Symposium on Physical Design (ISPD)*. Apr. 2003, pages 88–94. [see page 182]

[CS03] J. Cong and J. R. Shinnerl. *Multilevel Optimization in VLSICAD*. Kluwer Academic Publishers, 2003. [see pages 21, 41]

[CS16] B. Calvo and G. Santafé. „scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems“. In: *The R Journal* 8.1 (2016), pages 248–256. [see page 34]

[CS93] J. Cong and M. Smith. „A Parallel Bottom-Up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design“. In: *30th Conference on Design Automation (DAC)*. June 1993, pages 755–760. [see pages 20, 37, 67, 69]

[CSZ93] P. K. Chan, M. D. F. Schlag, and J. Y. Zien. „Spectral K-Way Ratio-Cut Partitioning and Clustering“. In: *30th Conference on Design Automation (DAC)*. June 1993, pages 749–754. [see pages 16, 50, 51, 53]

[CSZ94] P. K. Chan, M. D. F. Schlag, and J. Y. Zien. „Spectral k-Way Ratio-Cut Partitioning and Clustering“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 13.9 (1994), pages 1088–1096. [see pages 16, 50, 51, 53]

[CSZ96a] P. K. Chan, M. D. F. Schlag, and J. Y. Zien. „Multi-Level Spectral Hypergraph Partitioning with Arbitrary Vertex Sizes“. In: *International Conference on Computer-Aided Design (ICCAD)*. IEEE, Nov. 1996, pages 201–204. [see page 70]

[CSZ96b] P. K. Chan, M. D. F. Schlag, and J. Y. Zien. *Multi-Level Spectral Hypergraph Partitioning with Arbitrary Vertex Sizes*. Technical report UCSC-CRL-96-15. University of California at Santa Cruz, July 1996. [see page 70]

[CSZ97a] P. K. Chan, M. D. F. Schlag, and J. Y. Zien. „Hybrid Spectral/Iterative Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1997, pages 436–440. [see pages 36, 70, 124]

[CSZ97b] P. K. Chan, M. D. F. Schlag, and J. Y. Zien. *Hybrid Spectral/Iterative Partitioning*. Technical report UCUC-CRL-97-09. University of California at Santa Cruz, 1997. [see pages 70, 124]

[CSZ99] P. K. Chan, M. D. F. Schlag, and J. Y. Zien. „Multilevel Spectral Hypergraph Partitioning with Arbitrary Vertex Sizes“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 18.9 (1999), pages 1389–1399. [see pages 20, 70]

[Cur+10] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. „Schism: A Workload-Driven Approach to Database Replication and Partitioning“. In: *Proceedings of the VLDB Endowment* 3.1 (Sept. 2010), pages 48–57. [see page 2]

[CW02a] Y. Cheon and M. D. F. Wong. „Design Hierarchy Guided Multilevel Circuit Partitioning“. In: *International Symposium on Physical Design (ISPD)*. 2002, pages 30–35. [see page 37]

[CW02b] J. Cong and C. Wu. „Global Clustering-Based Performance-Driven Circuit Partitioning“. In: *International Symposium on Physical Design (ISPD)*. Apr. 2002, pages 149–154. [see page 37]

[CW03] Y. Cheon and M. D. F. Wong. „Design Hierarchy-Guided Multilevel Circuit Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 22.4 (2003), pages 420–427. [see page 37]

[CW91] C.-K. Cheng and Y.-C. Wei. „An Improved Two-Way Partitioning Algorithm with Stable Performance [VLSI]“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 10.12 (1991), pages 1502–1511. [see pages 36, 66, 88]

[CWC01] C.-C. Cheung, Y.-L. Wu, and D. I. Cheng. „Further Improve Circuit Partitioning Using GBAW Logic Perturbation Techniques“. In: *Design, Automation and Test in Europe (DATE)*. Mar. 2001, pages 233–239. [see page 37]

[CX17] C. Chekuri and C. Xu. „Computing Minimum Cuts in Hypergraphs“. In: *28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Jan. 2017, pages 1085–1100. [see page 18]

[CX18] C. Chekuri and C. Xu. „Minimum Cuts and Sparsification in Hypergraphs“. In: *SIAM Journal on Computing* 47.6 (2018), pages 2118–2156. [see page 18]

[CXY18] K. Chandrasekaran, C. Xu, and X. Yu. „Hypergraph k-Cut in Randomized Polynomial Time“. In: *29th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Jan. 2018, pages 1426–1438. [see page 18]

[CY00] C. C. Choi and Y. Ye. „Application of Semidefinite Programming to Circuit Partitioning“. In: *Approximation and Complexity in Numerical Optimization: Continuous and Discrete Problems*. Edited by P. M. Pardalos. Springer, 2000, pages 130–137. [see pages 61, 64]

[DA94] A. Daşdan and C. Aykanat. „Improved Multiple-Way Circuit Partitioning Algorithms“. In: *ACM/SIGDA International Workshop on Field-Programmable Gate Arrays*. 1994. [see pages 36, 51, 61, 78]

[DA96] A. Daşdan and C. Aykanat. *Two Novel Multiway Circuit Partitioning Algorithms using Relaxed Locking*. Technical report BU-CEIS-9609. Bilkent University, 1996. [see pages 51, 61]

[DA97] A. Daşdan and C. Aykanat. „Two Novel Multiway Circuit Partitioning Algorithms Using Relaxed Locking“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 16.2 (1997), pages 169–178. [see pages 51, 61]

[Dar59] C. Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life*. Murray, 1859. [see page 21]

[Das93] A. Dasdan. „Graph and Hypergraph Partitioning“. Master Thesis. Bilkent University, 1993. [see pages 51, 95]

[DAT99] S. Dutt, H. Arslan, and H. Theny. „Partitioning Using Second-Order Information and Stochastic-Gainfunctions“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 18.4 (1999), pages 421–435. [see page 57]

[DD00] S. Dutt and W. Deng. „Probability-Based Approaches to VLSI Circuit Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 19.5 (2000), pages 534–549. [see page 60]

[DD02] S. Dutt and W. Deng. „Cluster-Aware Iterative Improvement Techniques for Partitioning Large VLSI Circuits“. In: *ACM Transactions on Design Automation of Electronic Systems* 7.1 (2002), pages 91–121. [see pages 54, 55, 122]

[DD96a] S. Dutt and W. Deng. „A Probability-Based Approach to VLSI Circuit Partitioning“. In: *33rd Conference on Design Automation (DAC)*. June 1996, pages 100–105. [see pages 36, 54–56]

[DD96b] S. Dutt and W. Deng. „VLSI Circuit Partitioning by Cluster-Removal Using Iterative Improvement Techniques“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1996, pages 194–200. [see pages 36, 54, 55, 68, 72, 77, 122, 148]

[DD96c] S. Dutt and W. Deng. „VLSI Circuit Partitioning by Cluster-Removal Using Iterative Improvement Techniques (Extended Version)“. In: 1996, pages 1–31. [see pages 54, 55, 122]

[DD99] S. Dutt and W. Deng. *Probability-Based Approaches to VLSI Circuit Partitioning*. Technical report. University of Illinois at Chicago, 1999. [see page 60]

[De 06] K. A. De Jong. *Evolutionary Computation - A Unified Approach*. MIT Press, 2006. [see page 172]

[Del+11] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. „Graph Partitioning with Natural Cuts“. In: *25th International Parallel and Distributed Processing Symposium (IPDPS)*. 2011, pages 1135–1146. [see pages 2, 174]

[Del+17] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. „Customizable Route Planning in Road Networks“. In: *Transportation Science* 51.2 (2017), pages 566–591. [see page 2]

[Dem06] J. Demsar. „Statistical Comparisons of Classifiers over Multiple Data Sets“. In: *Journal of Machine Learning Research* 7 (2006), pages 1–30. [see page 34]

[Dev+02] K. D. Devine, E. G. Boman, R. T. Heaphy, B. Hendrickson, and C. T. Vaughan. „Zoltan Data Management Services for Parallel Dynamic Applications“. In: *Computing in Science and Engineering* 4.2 (2002), pages 90–96. [see page 81]

[Dev+06] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek. „Parallel Hypergraph Partitioning for Scientific Computing“. In: *20th International Parallel and Distributed Processing Symposium (IPDPS)*. Apr. 2006. [see pages 81, 82, 85, 96, 112, 114, 182]

[Dev+09] K. D. Devine, E. G. Boman, L. A. Riesen, Ü. V. Çatalyürek, and C. Chevalier. „Getting Started with Zoltan: A Short Tutorial“. In: *Combinatorial Scientific Computing*. Feb. 2009. [see page 81]

[DH03a] D. E. Drake and S. Hougardy. „A Simple Approximation Algorithm for the Weighted Matching Problem“. In: *Information Processing Letters* 85.4 (2003), pages 211–213. [see page 85]

[DH03b] D. E. Drake and S. Hougardy. „Linear Time Local Improvements for Weighted Matchings in Graphs“. In: *2nd International Experimental and Efficient Algorithms Workshop*. 2003, pages 107–119. [see page 85]

[DH11] T. A. Davis and Y. Hu. „The University of Florida Sparse Matrix Collection“. In: *ACM Transactions on Mathematical Software* 38.1 (Nov. 2011), 1:1–1:25. [see page 23]

[DH72] W. E. Donath and A. J. Hoffman. „Algorithms for Partitioning Graphs and Computer Logic Based on Eigenvectors of Connection Matrices“. In: *IBM Technical Disclosure Bulletin* 15.3 (1972), pages 938–944. [see pages 20, 49]

[DH73] W. E. Donath and A. J. Hoffman. „Lower Bounds for the Partitioning of Graphs“. In: *IBM Journal of Research and Development* 17.5 (Sept. 1973), pages 420–425. [see pages 20, 49]

[Din70] E. A. Dinic. „Algorithm for solution of a problem of maximum flow in networks with power estimation“. In: *Soviet Mathematics Doklady* 11 (1970), pages 1277–1280. [see page 87]

[DK85] A. E. Dunlop and B. W. Kernighan. „A Procedure for Placement of Standard Cell VLSI Circuits“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 4.1 (1985), pages 92–98. [see pages 40, 44]

[DKÇ13] M. Deveci, K. Kaya, and Ü. V. Çatalyürek. „Hypergraph Sparsification and Its Application to Partitioning“. In: *42nd International Conference on Parallel Processing (ICPP)*. Oct. 2013, pages 200–209. [see pages 85, 107, 119]

[DM01] E. D. Dolan and J. J. Moré. „Benchmarking Optimization Software with Performance Profiles“. In: *CoRR* cs.MS/0102001 (2001). [see page 30]

[DM02] E. D. Dolan and J. J. Moré. „Benchmarking Optimization Software with Performance Profiles“. In: *Mathematical programming* 91.2 (2002), pages 201–213. [see pages 30, 31]

[Don88] W. E. Donath. „Logic partitioning“. In: *Physical Design Automation of VLSI Systems* (1988), pages 65–86. [see pages 4, 6, 14, 35, 37, 181]

[DT97] S. Dutt and H. Theny. „Partitioning Around Roadblocks: Tackling Constraints with Intermediate Relaxations“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1997, pages 350–355. [see pages 36, 56, 59]

[DT98] S. Dutt and H. Theny. „Partitioning Using Second-Order Information and Stochastic-Gain Functions“. In: *International Symposium on Physical Design (ISPD)*. Apr. 1998, pages 112–117. [see page 57]

[Dut93] S. Dutt. „New Faster Kernighan-Lin-Type Graph-Partitioning Algorithms“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1993, pages 370–377. [see pages 39, 40]

[DW19] M. Dietzfelbinger and S. Walzer. „Dense Peelable Random Uniform Hypergraphs“. In: *27th Annual European Symposium on Algorithms (ESA)*. 2019, 38:1–38:16. [see page 113]

[EHS99] M. Enos, S. Hauck, and M. Sarrafzadeh. „Evaluation and Optimization of Replication Algorithms for Logic Bipartitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 18.9 (Sept. 1999), pages 1237–1248. [see page 37]

[FB14] B. O. Fagginger Auer and R. H. Bisseling. „Efficient Matching for Column Intersection Graphs“. In: *ACM Journal of Experimental Algorithmics (JEA)* 19.1 (2014). [see page 85]

[Fel12] A. E. Feldmann. „Balanced partitioning of grids and related graphs: a theoretical study of data distribution in parallel finite element model simulations“. PhD thesis. ETH Zurich, 2012. [see page 19]

[Fel13] A. E. Feldmann. „Fast balanced partitioning is hard even on grids and trees“. In: *Theoretical Computer Science* 485 (2013), pages 61–68. [see page 19]

[FF12] A. E. Feldmann and L. Foschini. „Balanced Partitions of Trees and Applications“. In: *29th International Symposium on Theoretical Aspects of Computer Science (STACS)*. 2012, pages 100–111. [see page 19]

[FF15] A. E. Feldmann and L. Foschini. „Balanced Partitions of Trees and Applications“. In: *Algorithmica* 71.2 (2015), pages 354–376. [see page 19]

[FF56] L. R. Ford and D. R Fulkerson. „Maximal Flow through a Network“. In: *Canadian Journal of Mathematics* 8.3 (1956), pages 399–404. [see pages 13, 18, 142, 143]

[FF62] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962. [see pages 141, 144]

[FH16] S. Fortunato and D. Hric. „Community Detection in Networks: A User Guide“. In: *Physics Reports* 659 (2016), pages 1–44. [see pages 110, 148]

[Fie73] M. Fiedler. „Algebraic Connectivity of Graphs“. In: *Czechoslovak Mathematical Journal* 23.2 (1973), pages 298–305. [see page 20]

[Fie75a] M. Fiedler. „A Property of Eigenvectors of Nonnegative Symmetric Matrices and its Application to Graph Theory“. In: *Czechoslovak Mathematical Journal* 25.4 (1975), pages 619–633. [see pages 20, 49]

[Fie75b] M. Fiedler. „Eigenvectors of Acyclic Matrices“. In: *Czechoslovak Mathematical Journal* 25.4 (1975), pages 607–618. [see page 20]

[FJK96] A. S. Fukunaga, D. J.-H.Huang, and A. B. Kahng. „Large-Step Markov Chain Variants for VLSI Netlist Partitioning“. In: *International Symposium on Circuits and Systems (ISCAS)*. Volume 4. IEEE. 1996, pages 496–499. [see page 54]

[FM82] C. M. Fiduccia and R. M. Mattheyses. „A Linear-Time Heuristic for Improving Network Partitions“. In: *19th Conference on Design Automation (DAC)*. 1982, pages 175–181. [see pages 5, 9, 12, 35–37, 41–45, 52, 53, 55, 58, 67, 73, 79, 81, 82, 121, 154, 170]

[For+13] O. Fortmeier, H. M. Bücker, B. O. Fagginger Auer, and R. H. Bisseling. „A New Metric Enabling an Exact Hypergraph Model for the Communication Volume in Distributed-Memory Parallel Applications“. In: *Parallel Computing* 39.8 (2013), pages 319–335. [see page 16]

[For10] S. Fortunato. „Community Detection in Graphs“. In: *Physics Reports* 486.3–5 (2010), pages 75–174. [see pages 110, 112]

[Fou] The Apache Software Foundation. *Apache Giraph*. `http://giraph.apache.org`. [see pages 62, 92]

[FPZ19] K. Fox, D. Panigrahi, and F. Zhang. „Minimum Cut and Minimum k-Cut in Hypergraphs via Branching Contractions“. In: *30th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Jan. 2019, pages 881–896. [see page 18]

[FR89] T. A. Feo and M. G. C. Resende. „A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem“. In: *Operations Research Letters* 8.2 (Apr. 1989), pages 67–71. [see pages 58, 69, 80]

[Fri37] M. Friedman. „The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance“. In: *Journal of the American Statistical Association* 32.200 (1937), pages 675–701. [see page 34]

[Fri40] M. Friedman. „A Comparison of Alternative Tests of Significance for the Problem of m Rankings“. In: *The Annals of Mathematical Statistics* 11.1 (1940), pages 86–92. [see page 34]

[Fun+18] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. „Communication-Free Massively Distributed Graph Generation“. In: *32nd International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pages 336–347. [see page 26]

[FW86] P. J. Fleming and J. J. Wallace. „How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results“. In: *Communications of the ACM* 29.3 (1986), pages 218–221. [see page 30]

[Gar+10] S. Garcia, A. Fernández, J. Luengo, and F. Herrera. „Advanced Nonparametric Tests for Multiple Comparisons in the Design of Experiments in Computational Intelligence and Data Mining: Experimental Analysis of Power“. In: *Information Sciences* 180.10 (2010), pages 2044–2064. [see page 34]

[GB83] M. K. Goldberg and M. Burstein. „Heuristic Improvement Technique for Bisection of VLSI Networks“. In: *International Conference on Computer-Aided Design (ICCAD)*. 1983, pages 122–125. [see pages 23, 36, 44, 65, 66, 88, 94]

[GG83] M. K. Goldberg and R. Gardner. „On the minimal cut problem“. In: *Silver Jubilee Conference on Combinatorics*. 1983. [see page 65]

[GH08] S. Garcia and F. Herrera. „An Extension on "Statistical Comparisons of Classifiers over Multiple Data Sets" for all Pairwise Comparisons“. In: *Journal of Machine Learning Research* 9 (2008), pages 2677–2694. [see page 34]

[GH94] O. Goldschmidt and D. S. Hochbaum. „A Polynomial Algorithm for the k-cut Problem for Fixed k“. In: *Mathematics of Operations Research* 19.1 (1994), pages 24–37. [see page 18]

[GHW19a] L. Gottesbüren, M. Hamann, and D. Wagner. „Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm“. In: *CoRR* (2019). arXiv: 1907.02053. [see pages 87, 182]

[GHW19b] L. Gottesbüren, M. Hamann, and D. Wagner. „Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm“. In: *27th Annual European Symposium on Algorithms (ESA)*. Volume 144. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Sept. 2019, 52:1–52:17. [see pages 87, 182, 253]

[GIM99] A. Gionis, P. Indyk, and R. Motwani. „Similarity Search in High Dimensions via Hashing“. In: *25th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers Inc., 1999, pages 518–529. [see pages 106, 107]

[GJS76] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. „Some Simplified NP-Complete Graph Problems“. In: *Theoretical Computer Science* 1.3 (1976), pages 237–267. [see pages 3, 9, 19]

[GL16] J. Giráldez-Cru and J. Levy. „Generating SAT instances with community structure“. In: *Artificial Intelligence* 238 (2016), pages 119–134. [see page 148]

[GLA19] P.-E. Gaillardon, W. Lau Neto, and M. Austin. *LSOracle - The Logic Synthesis Oracle*. `https://github.com/LNIS-Projects/LSOracle`. 2019. [see pages 6, 216]

[Glo89] F. Glover. „Tabu Search - Part I“. In: *INFORMS Journal on Computing* 1.3 (1989), pages 190–206. [see page 50]

[Glo90] F. Glover. „Tabu Search - Part II“. In: *INFORMS Journal on Computing* 2.1 (1990), pages 4–32. [see page 50]

[Gol+11] A. Goldberg, S. Hed, H. Kaplan, R. Tarjan, and R. Werneck. „Maximum Flows by Incremental Breadth-First Search“. In: *19th European Symposium on Algorithms (ESA)* (2011), pages 457–468. [see pages 145, 157, 158]

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989. [see page 22]

[Gon+12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. „PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs“. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Oct. 2012, pages 17–30. [see pages 16, 206, 218]

[Gon+14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. „GraphX: Graph Processing in a Distributed Dataflow Framework“. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2014, pages 599–613. [see page 218]

[Gon85] Te. F. Gonzalez. „Clustering to Minimize the Maximum Intercluster Distance“. In: *Theoretical Computer Science* 38 (1985), pages 293–306. [see page 51]

[Got19] L. Gottesbüren. *Interating HyperFlowCutter [GHW19b] as a refinement algorithm into KaHyPar*. Private Communication. 2019. [see pages 6, 216]

[GPS90] J. Garbers, H. J. Prömel, and A. Steger. „Finding Clusters in VLSI Circuits“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1990, pages 520–523. [see page 69]

[GS16] N. I. M. Gould and J. A. Scott. „A Note on Performance Profiles for Benchmarking Software“. In: *ACM Transactions on Mathematical Software* 43.2 (2016), 15:1–15:5. [see page 30]

[GSA07] R. Guimera, M. Sales-Pardo, and L. A. N. Amaral. „Module identification in bipartite and directed networks“. In: *Physical Review* 76 (3 Sept. 2007). [see page 111]

[GT86] A. V. Goldberg and R. E. Tarjan. „A New Approach to the Maximum Flow Problem“. In: *18th ACM Symposium on Theory of Computing (STOC)*. ACM, May 1986, pages 136–146. [see pages 18, 136]

[Had95] S. W. Hadley. „Approximation Techniques for Hypergraph Partitioning Problems“. In: *Discrete Applied Mathematics* 59.2 (1995), pages 115–127. [see page 14]

[Hag+92] L. W. Hagen, F. J. Kurdahi, C. Ramachandran, and A. B. Kahng. „On the Intrinsic Rent Parameter and Spectra-Based Partitioning Methodologies“. In: *European Design Automation Conference (EURO-DAC)*. Sept. 1992, pages 202–208. [see page 37]

[Hag+94] L. W. Hagen, A. B. Kahng, F. J. Kurdahi, and C. Ramachandran. „On the Intrinsic Rent Parameter and Spectra-Based Partitioning Methodologies“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 13.1 (1994), pages 27–37. [see page 37]

[Han+19] M Hanai, T. Suzumura, W. J. Tan, E. S. Liu, G. Theodoropoulos, and W. Cai. „Distributed Edge Partitioning for Trillion-edge Graphs“. In: *CoRR* abs/1908.05855 (2019). [see page 218]

[Hau95] S. A. Hauck. „Multi-FPGA Systems“. PhD thesis. 1995. [see page 69]

[HB15] T. Hoefler and R. Belli. „Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results“. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Nov. 2015, 73:1–73:12. [see page 29]

[HB95] S. Hauck and G. Borriello. „An Evaluation of Bipartitioning Techniques“. In: *16th Conference on Advanced Research in VLSI (ARVLSI)*. Mar. 1995, pages 383–403. [see pages 20, 37, 68, 69, 94, 204]

[HB97] S. Hauck and G. Borriello. „An Evaluation of Bipartitioning Techniques“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 16.8 (1997), pages 849–866. [see pages 6, 7, 36, 41, 44, 45, 68, 69, 94, 114, 181]

[HC14] B. Heintz and A. Chandra. „Beyond Graphs: Toward Scalable Hypergraph Analysis Systems“. In: *SIGMETRICS Performance Evaluation Review* 41.4 (2014), pages 94–97. [see pages 3, 4, 218]

[HC15] B. Heintz and A. Chandra. „Enabling Scalable Social Group Analytics via Hypergraph Analysis Systems“. In: *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. July 2015. [see pages 3, 218]

[HE92] L. J. Hwang and A. El Gamal. „Optimal Replication for Min-Cut Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1992, pages 432–435. [see page 17]

[Hei+19] B. Heintz, R. Hong, S. Singh, G. Khandelwal, C. Tesdahl, and A. Chandra. „MESH: A Flexible Distributed Hypergraph Processing System“. In: *IEEE International Conference on Cloud Engineering (IC2E)*. June 2019, pages 12–22. [see pages 3, 218]

[Hen+15a] V. Henne, H. Meyerhenke, P. Sanders, S. Schlag, and C. Schulz. „n-Level Hypergraph Partitioning“. In: *Computing Research Repository (CoRR)* (May 2015), pages 1–23. arXiv: 1505.00693. [see pages 4, 97]

[Hen15a] V. Henne. „Label Propagation for Hypergraph Partitioning“. Master Thesis. Karlsruhe Institute of Technology, May 2015. [see pages 97, 121]

[Hen98] B. Hendrickson. „Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes? (Extended Abstract)“. In: *5th International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR)*. Aug. 1998, pages 218–225. [see page 3]

[Heu15a] T. Heuer. „Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, Aug. 2015. [see pages 99, 120, 121]

[Heu18a] T. Heuer. „High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations“. Master Thesis. Karlsruhe Institute of Technology, Jan. 2018. [see pages 99, 145, 157, 158]

[HG95] L. J. Hwang and A. El Gamal. „Min-Cut Replication in Partitioned Networks“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 14.1 (1995), pages 96–106. [see page 17]

[HHK95a] L. W. Hagen, D. J.-H. Huang, and A. B. Kahng. „On Implementation Choices for Iterative Improvement Partitioning Algorithms“. In: *European Design Automation Conference (EURO-DAC)*. Sept. 1995, pages 144–149. [see pages 36, 40, 44, 45, 53, 68, 77]

[HHK95b] L. W. Hagen, D. J.-H. Huang, and A. B. Kahng. „Quantified Suboptimality of VLSI Layout Heuristics“. In: *32nd Conference on Design Automation (DAC)*. June 1995, pages 216–221. [see page 36]

[HHK97] L. W. Hagen, D. J.-H. Huang, and A. B. Kahng. „On Implementation Choices for Iterative Improvement Partitioning Algorithms“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 16.10 (1997), pages 1199–1205. [see pages 9, 44, 53]

[Hil+06] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. „Fast Point-to-Point Shortest Path Computations with Arc-Flags“. In: *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. 2006, pages 41–72. [see page 2]

[HK00] B. Hendrickson and T. G. Kolda. „Partitioning Rectangular and Structurally Unsymmetric Sparse Matrices for Parallel Processing“. In: *SIAM Journal on Scientific Computing* 21.6 (2000), pages 2048–2072. [see page 3]

[HK91a] L. W. Hagen and A. B. Kahng. „Fast Spectral Methods for Ratio Cut Partitioning and Clustering“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1991, pages 10–13. [see pages 35, 49, 50, 69, 70]

[HK91b] L. Hagen and A. Kahng. *New Spectral Methods for Ratio Cut Partitioning and Clustering*. Technical report TR-910073. UCLA, 1991. [see page 49]

[HK92a] L. W. Hagen and A. B. Kahng. „A New Approach to Effective Circuit Clustering“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1992, pages 422–427. [see pages 66, 68, 69, 88]

[HK92b] L. W. Hagen and A. B. Kahng. „New Spectral Methods for Ratio Cut Partitioning and Clustering“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 11.9 (1992), pages 1074–1085. [see pages 35, 44, 49]

[HK92c] L. Hagen and A. B. Kahng. „Improving the Quadratic Objective Function in Module Placement“. In: *5th Annual IEEE International ASIC Conference*. IEEE. 1992, pages 42–45. [see page 14]

[HK95] D. J.-H. Huang and A. B. Kahng. „Multi-Way System Partitioning into a Single Type or Multiple Types of FPGAs“. In: *3rd International ACM Symposium on Field-Programmable Gate Arrays (FPGA)*. Feb. 1995, pages 140–145. [see page 37]

[HK97] L. W. Hagen and A. B. Kahng. „Combining Problem Reduction and Adaptive Multistart: A New Technique for Superior Iterative Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 16.7 (July 1997), pages 709–717. [see pages 36, 71, 88]

[HL93] B. Hendrickson and R. Leland. *A Multilevel Algorithm for Partitioning Graphs*. Technical report SAND93-1301. Sandia National Laboratories, 1993. [see pages 20, 95]

[HL95] B. Hendrickson and R. Leland. „A Multi-Level Algorithm For Partitioning Graphs“. In: *Supercomputing*. 1995, page 28. [see pages 20, 37, 70, 71]

[HM85] T. C. Hu and K. Moerder. „Multiterminal Flows in a Hypergraph“. In: *VLSI Circuit Layout: Theory and Design*. Edited by T.C. Hu and E.S. Kuh. IEEE, 1985. Chapter 3, pages 87–93. [see pages 13, 139]

[HMV92] S. W. Hadley, B. L. Mark, and A. Vannelli. „An Efficient Eigenvector Approach for Finding Netlist Partitions“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 11.7 (1992), pages 885–892. [see pages 14, 35, 50]

[HNS82] G. D. Hachtel, A. R. Newton, and A. L. Sangiovanni-Vincentelli. „An Algorithm for Optimal PLA Folding“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 1.2 (Apr. 1982), pages 63–77. [see page 48]

[Hof94] A. G. Hoffmann. „The Dynamic Locking Heuristic - A New Graph Partitioning Algorithm“. In: *International Symposium on Circuits and Systems (ISCAS)*. 1994, pages 173–176. [see pages 36, 51]

[Hol75a] J. H. Holland. *Adaptation in Natural and Artificial Systems*. second edition, 1992. University of Michigan Press, 1975. [see page 21]

[Hol75b] J. H. Holland. „Adaptation in Natural and Artificial Systems. An Introductory Analysis with Application to Biology, Control, and Artificial Intelligence“. In: (1975), pages 439–444. [see page 21]

[Hot+08] T. Hothorn, K. Hornik, M. A. van de Wiel, and A. Zeileis. „Implementing a Class of Permutation Tests: The coin Package“. In: *Journal of Statistical Software* 28.8 (2008), pages 1–23. [see page 34]

[HR55] T. E. Harris and F. S. Ross. *Fundamentals of a Method for Evaluating Rail Net Capacities*. Technical report. Oct. 1955. [see page 18]

[HR98] B. Hendrickson and E. Rothberg. „Improving the Run Time and Quality of Nested Dissection Ordering“. In: *SIAM Journal on Scientific Computing* 20.2 (1998), pages 468–489. [see page 118]

[HRW17] M. Henzinger, S. Rao, and D. Wang. „Local Flow Partitioning for Faster Edge Connectivity“. In: *28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Jan. 2017, pages 1919–1938. [see page 18]

[HS15] H. H. Hoos and T. Stützle. „Stochastic Local Search Algorithms: An Overview“. In: *Springer Handbook of Computational Intelligence*. 2015, pages 1085–1105. [see page 21]

[HS17a] T. Heuer and S. Schlag. „Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure“. In: *16th International Symposium on Experimental Algorithms (SEA)*. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2017, 21:1–21:19. [see pages 4, 11, 23, 24, 26, 97–99, 103, 111, 113, 146, 147, 150, 159, 170, 184]

[HS18a] M. Hamann and B. Strasser. „Graph Bisection with Pareto Optimization“. In: *ACM Journal of Experimental Algorithmics (JEA)* 23 (2018). [see page 87]

[HS18b] L. Hübschle-Schneider and P. Sanders. „Communication Efficient Checking of Big Data Operations“. In: *32nd International Parallel and Distributed Processing Symposium (IPDPS)*. May 2018, pages 650–659. [see page 119]

[HSS18a] T. Heuer, P. Sanders, and S. Schlag. „Network Flow-Based Refinement for Multilevel Hypergraph Partitioning“. In: *17th International Symposium on Experimental Algorithms (SEA)*. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2018, 1:1–1:19. [see pages 4, 11, 23, 24, 26, 97–99, 103, 146, 156, 159, 170, 175]

[HSS18b] T. Heuer, P. Sanders, and S. Schlag. „Network Flow-Based Refinement for Multilevel Hypergraph Partitioning“. In: *Computing Research Repository (CoRR)* (Feb. 2018), pages 1–28. arXiv: 1802.03587. [see pages 4, 98, 156]

[HSS19a] T. Heuer, P. Sanders, and S. Schlag. „Network Flow-Based Refinement for Multilevel Hypergraph Partitioning“. In: *ACM Journal of Experimental Algorithmics (JEA)* 24.1 (Sept. 2019), 2.3:1–2.3:36. [see pages 4, 11, 23, 97–99, 103, 138, 140–142, 144, 146, 156, 158, 159, 170, 171, 175, 181, 182, 184, 204, 208, 210]

[Hul90] M. Hulin. „Circuit Partitioning with Genetic Algorithms Using a Coding Scheme to Preserve the Structure of a Circuit“. In: *1st Parallel Problem Solving from Nature Workshop (PPSN)*. 1990, pages 75–79. [see pages 37, 172]

[Hul91] M. Hulin. „Analysis of Schema Distributions“. In: *4th International Conference on Genetic Algorithms (ICGA)*. July 1991, pages 204–209. [see page 37]

[HZY15] J. Huang, R. Zhang, and J. X. Yu. „Scalable Hypergraph Learning and Processing“. In: *International Conference on Data Mining (ICDM)*. Nov. 2015, pages 775–780. [see pages 3, 218]

[ID80] R. L. Iman and J. M. Davenport. „Approximations of the Critical Region of the Friedman Statistic“. In: *Communications in Statistics - Theory and Methods* 9.6 (1980), pages 571–595. [see page 34]

[IKS75] T. Ishiga, T. Kozawa, and S. Sato. „A Logic Partitioning Procedure by Interchanging Clusters“. In: *12th Conference on Design Automation (DAC)*. June 1975, pages 369–377. [see pages 16, 36, 65, 88]

[IM98] P. Indyk and R. Motwani. „Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality“. In: *29th ACM Symposium on Theory of Computing (STOC)*. ACM, 1998, pages 604–613. [see pages 5, 106]

[IWW93] E. Ihler, D. Wagner, and F. Wagner. „Modeling Hypergraphs by Graphs with the Same Mincut Properties“. In: *Information Processing Letters* 45.4 (1993), pages 171–175. [see pages 15, 59]

[Jal19a] J. Jalving. *KaHyPar.jl – A Julia interface to the KaHyPar hypergraph partitioning package*. 2019. URL: `https://github.com/jalving/KaHyPar.jl`. [see page 9]

[Jal19b] J. Jalving. *Partitioning hypergraphs derived from nonlinear optimization problems in the field of chemical engineering.* Private Communication. 2019. [see pages 6, 216]

[Jia+18] W. Jiang, J. Qi, J. X. Yu, J. Huang, and R. Zhang. „HyperX: A Scalable Hypergraph Framework“. In: *IEEE Transactions on Knowledge and Data Engineering* (2018). [see pages 3, 218]

[Joh+89] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. „Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning“. In: *Operations Research* 37.6 (1989), pages 865–892. [see page 46]

[Joh96] F. M. Johannes. „Partitioning of VLSI Circuits and Systems“. In: *33rd Conference on Design Automation (DAC)*. June 1996, pages 83–87. [see pages 4, 35, 37]

[Joh99] D. S. Johnson. „A Theoretician's Guide to the Experimental Analysis of Algorithms". In: *Data Structures, Near Neighbor Searches, and Methodology: 5th and 6th DIMACS Implementation Challenges*. 1999, pages 215–250. [see page 10]

[Kab+17] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, A. Shalita, Y. Akhremtsev, and A. Presta. „Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner". In: *Proceedings of the VLDB Endowment* 10.11 (2017), pages 1418–1429. [see pages 2, 4, 62, 93, 96, 182]

[Kah89] A. B. Kahng. „Fast Hypergraph Partition". In: *26th Conference on Design Automation (DAC)*. June 1989, pages 762–766. [see pages 36, 48, 49, 84]

[Kah98] A. B. Kahng. „Futures for Partitioning in Physical Design (Tutorial)". In: *International Symposium on Physical Design (ISPD)*. Apr. 1998, pages 190–193. [see pages 4, 37]

[Kam+91] Y. Kamidoi, S. Wakabayashi, J. Miyao, and N. Yoshida. „A Fast Heuristic Algorithm for Hypergraph Bisection". In: *International Symposium on Circuits and Systems (ISCAS)*. 1991, pages 1160–1163. [see page 49]

[Kar+97a] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. „Multilevel Hypergraph Partitioning: Application in VLSI Domain". In: *34th Conference on Design Automation (DAC)*. June 1997, pages 526–529. [see pages 7, 36, 72, 76, 77, 95]

[Kar+97b] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. *Multilevel Hypergraph Partitioning: Application in VLSI Domain*. Technical report. University of Minnesota, 1997. [see pages 72, 76, 77]

[Kar+99] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. „Multilevel Hypergraph Partitioning: Applications in VLSI Domain". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pages 69–79. [see pages 72, 76, 77, 114, 183]

[Kar02] G. Karypis. *Multilevel Hypergraph Partitioning*. Technical report 02-025. University of Minnesota, June 2002. [see page 94]

[Kar03] G. Karypis. „Multilevel Hypergraph Partitioning". In: *Multilevel Optimization in VLSICAD*. Edited by J. Cong and J. R. Shinnerl. Springer, 2003, pages 125–154. [see pages 36, 94, 95, 98, 110]

[Kar06] E. Karaca. „Parpatoh: A 2D-Parallel Hypergraph Partitioning Tool". Master Thesis. Bilkent University, 2006. [see page 82]

[Kar13] G. Karypis. *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 5.1.0*. Technical report. University of Minnesota, 2013. [see page 84]

[Kar19] G. Karypis. *hMETIS - Hypergraph & Circuit Partitioning*. 2019. URL: `http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview`. [see page 182]

[Kar96] G. Karypis. „Graph Partitioning and Its Applications to Scientific Computing“. PhD thesis. 1996. [see page 121]

[Kar99] G. Karypis. *Multilevel Algorithms for Multi-Constraint Hypergraph Partitioning*. Technical report. Minnesota University, 1999. [see page 17]

[KAS97] H. Krupnova, A. Abbara, and G. Saucier. „A Hierarchy-Driven FPGA Partitioning Method“. In: *34th Conference on Design Automation (DAC)*. ACM, 1997, pages 522–525. [see page 37]

[KAV04] D. Kucar, S. Areibi, and A. Vannelli. „Hypergraph Partitioning Techniques“. In: *Dynamics of Continuous, Discrete and Impulsive Systems Series A: Mathematical Analysis* 11.2-3 (2004), pages 339–367. [see pages 4, 37]

[Kay+10] E. Kayaaslan, A. Pinar, Ü. V. Çatalyürek, and C. Aykanat. *Hypergraph Partitioning through Vertex Separators on Graphs*. Technical report BU-CE-1017. Bilkent University, 2010. [see page 84]

[Kay+11] E. Kayaaslan, A. Pinar, Ü. V. Çatalyürek, and C. Aykanat. „Hypergraph Partitioning through Vertex Separators on Graphs“. In: *CoRR* (2011). arXiv: 1103.0106. [see page 84]

[Kay+12] E. Kayaaslan, A. Pinar, Ü. V. Çatalyürek, and C. Aykanat. „Partitioning Hypergraphs in Scientific Computing Applications through Vertex Separators on Graphs“. In: *SIAM Journal on Scientific Computing* 34.2 (2012). [see pages 4, 84]

[Kay13] E. Kayaaslan. „Hypergraph-Based Data Partitioning“. PhD thesis. Bilkent University, 2013. [see page 84]

[KB18] T. E. Knigge and R. H. Bisseling. „An Improved Exact Algorithm and an NP-Completeness Proof for Sparse Matrix Bipartitioning“. In: *CoRR* (2018). arXiv: 1811.02043. [see page 95]

[KB95] R. Kužnar and F. Brglez. „PROP: A Recursive Paradigm for Area-Efficient and Performance Oriented Partitioning of Large FPGA Netlists“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1995, pages 644–649. [see page 37]

[KBK93] R. Kužnar, F. Brglez, and K. Kozminski. „Cost Minimization of Partitions into Multiple Devices“. In: *30th Conference on Design Automation (DAC)*. June 1993, pages 315–320. [see page 37]

[KBZ94] R. Kužnar, F. Brglez, and B. Zajc. „Multi-way Netlist Partitioning into Heterogeneous FPGAs and Minimization of Total Device Cost and Interconnect“. In: *31st Conference on Design Automation (DAC)*. ACM, 1994, pages 238–243. [see page 37]

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. „Optimization by Simulated Annealing“. In: *Science* 220.4598 (1983), pages 671–680. [see pages 36, 46, 63]

[Kim+11] J. Kim, I. Hwang, Y. H. Kim, and B. R. Moon. „Genetic Approaches for Graph Partitioning: A Survey“. In: *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2011, pages 473–480. [see pages 22, 51]

[KK00] G. Karypis and V. Kumar. „Multilevel k-way Hypergraph Partitioning“. In: *VLSI Design* 3 (2000), pages 285–300. [see pages 6, 16, 74, 75, 77, 79, 82, 114, 118, 121, 124, 136, 185]

[KK95a] G. Karypis and V. Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. Technical report 95-035. University of Minnesota, July 1995. [see pages 70, 94]

[KK95b] G. Karypis and V. Kumar. „Multilevel Graph Partitioning Schemes“. In: *24th International Conference on Parallel Processing (ICPP)*. Aug. 1995, pages 113–122. [see page 71]

[KK95c] G. Karypis and V. Kumar. *Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Technical report. University of Minnesota, 1995. [see pages 70, 76]

[KK98a] G. Karypis and V. Kumar. „A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs“. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pages 359–392. [see pages 70, 94, 182]

[KK98b] G. Karypis and V. Kumar. „hMETIS: A Hypergraph Partitioning Package, Version 1.5.3“. In: *user manual* 23 (1998). [see pages 93, 183]

[KK98c] G. Karypis and V. Kumar. *Multilevel k-way Hypergraph Partitioning*. Technical report 98-036. University of Minnesota, 1998. [see pages 74, 75, 77, 79, 82, 185]

[KK99] G. Karypis and V. Kumar. „Multilevel k-way Hypergraph Partitioning“. In: *36th Conference on Design Automation (DAC)*. 1999, pages 343–348. [see pages 7, 36, 74, 75, 77, 79, 82, 94–96, 116, 136, 171, 173, 183, 185]

[KKM04] J.-P. Kim, Y.-H. Kim, and B. R. Moon. „A Hybrid Genetic Approach for Circuit Bipartitioning“. In: *Genetic and Evolutionary Computation Conference (GECCO)*. June 2004, pages 1054–1064. [see pages 61, 171, 172, 175]

[KKS96] N. Kim, C. Kim, and H. Shin. „An Improved Partitioning Method Using Clustering Refinement [VLSI design]“. In: *Asia Pacific Conference on Circuits and Systems (APCCAS)*. 1996, pages 302–305. [see pages 67, 88]

[KL70] B. W. Kernighan and S. Lin. „An Efficient Heuristic Procedure for Partitioning Graphs“. In: *The Bell System Technical Journal* 49.2 (Feb. 1970), pages 291–307. [see pages 35, 37–40, 47, 57, 62, 70, 91, 95]

[KLK93] J.-U. Kim, C.-H. Lee, and M. Kim. „Efficient Multiple-Way Network-Partitioning Algorithm“. In: *Computer-Aided Design* 25.5 (1993), pages 269–280. [see pages 15, 50]

[KM04] Y.-H. Kim and B.-R. Moon. „Lock-Gain Based Graph Partitioning“. In: *Journal of Heuristics* 10.1 (Jan. 2004), pages 37–57. [see page 61]

[KN91] C. Kring and A. R. Newton. „A Cell-Replicating Approach to Minicut-Based Circuit Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1991, pages 2–5. [see page 17]

[KNS09] R. Krauthgamer, J. Naor, and R. Schwartz. „Partitioning Graphs into Balanced Components“. In: *20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, Jan. 2009, pages 942–949. [see page 19]

[Kod72] U. R. Kodres. „Partitioning and Card Selection“. In: *Design Automation of Digital Systems* (1972), pages 173–212. [see pages 4, 35, 37]

[Kri84] B. Krishnamurthy. „An Improved Min-Cut Algorithm for Partitioning VLSI Networks“. In: *IEEE Transactions on Computers* 33.5 (1984), pages 438–446. [see pages 36, 46–48, 51, 53–55, 69, 124]

[KRU11] K. Kaya, F.-H. Rouet, and B. Uçar. „On Partitioning Problems with Complex Objectives“. In: *European Conference on Parallel Processing (Euro-Par)*. 2011, pages 334–344. [see page 17]

[Kuc08] D. Kucar. „Partitioning and Clustering“. In: *Handbook of Algorithms for Physical Design Automation*. 2008. [see pages 4, 37]

[KUÇ14] K. Kaya, B. Uçar, and Ü. V. Çatalyürek. „Analysis of Partitioning Models and Metrics in Parallel Sparse Matrix-Vector Multiplication“. In: *Parallel Processing and Applied Mathematics*. Edited by R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski. Springer, 2014, pages 174–184. [see page 3]

[Kuh62] T. Kuhn. *The Structure of Scientific Revolutions*. The University of Chicago Press, 1962. [see page 35]

[Kum+14] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller. „SWORD: Workload-Aware Data Placement and Replica Selection for Cloud Data Management Systems“. In: *The VLDB Journal* 23.6 (Dec. 2014), pages 845–870. [see page 2]

[KW19] T. A. Kuffner and S. G. Walker. „Why are p-Values Controversial?“ In: *The American Statistician* 73.1 (2019), pages 1–3. [see page 34]

[KW96] R. Klimmek and F. Wagner. *A Simple Hypergraph Min Cut Algorithm*. Technical report B 96-02. FU Berlin, 1996. [see page 18]

[KY15] Y.-H. Kim and Y. Yoon. „Accounting for Recent Changes of Gain in Dealing with Ties in Iterative Methods for Circuit Partitioning.“ In: *Discrete Dynamics in Nature & Society* (2015), pages 1–8. [see page 45]

[KZB94] R. Kužnar, B. Zajc, and F. Brglez. „A Unified Cost Model for Min-Cut Partitioning with Replication Applied to Pptimization of Large Heterogeneous FPGA Partitions“. In: *European Design Automation Conference (EURO-DAC)*. Sept. 1994, pages 271–276. [see page 37]

[Law73] E. L. Lawler. „Cutsets and Partitions of Hypergraphs“. In: *Networks* 3.3 (1973), pages 275–285. [see pages 18, 53, 139, 141]

[Laz+09] D. Lazer, A. Pentland, L. Adamic, S. Aral, A.-L. Barabási, D. Brewer, N Christakis, N. Contractor, J. Fowler, M. Gutmann, T. Jebara, G. King, M. Macy, D. Roy, and M. Van Alstyne. „Computational Social Science“. In: *Science* 323.5915 (2009), pages 721–723. [see page 3]

[LB06a] J. Li and L. Behjat. „A Connectivity Based Clustering Algorithm with Application to VLSI Circuit Partitioning“. In: *IEEE Transactions on Circuits and Systems* 53-II.5 (2006), pages 384–388. [see pages 82, 88]

[LB06b] J. Li and L. Behjat. „Net Cluster: A Net-Reduction Based Clustering Preprocessing Algorithm“. In: *International Symposium on Physical Design (ISPD)*. Apr. 2006, pages 200–205. [see page 83]

[LBK07] J. Li, L. Behjat, and A. A. Kennings. „Net Cluster: A Net-Reduction-Based Clustering Preprocessing Algorithm for Partitioning and Placement“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 26.4 (2007), pages 669–679. [see page 83]

[LBR07] J. Li, L. Behjat, and L. Rakai. „Clustering Algorithms for Circuit Partitioning and Placement Problems“. In: *18th European Conference on Circuit Theory and Design (ECCTD)*. Aug. 2007, pages 547–550. [see page 117]

[LD17] S. Locke and L. D'Agostino McGowan. *The Datasaurus Dozen*. `https://itsalocke.com/datasaurus/`. 2017. [see pages 29, 30]

[Lee06] T. van Leeuwen. „Expanding Mondriaan's Palette: A Study to Improve Mondriaan, a Hypergraph-Based Matrix Partitioner“. Master Thesis. 2006. [see pages 83, 92]

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990. [see pages 3, 9, 14, 15, 36, 40, 42, 44, 66, 95]

[LF09] A. Lancichinetti and S. Fortunato. „Community detection algorithms: A comparative analysis“. In: *Physical Review* 80 (5 Nov. 2009). [see page 112]

[Li+17] L. Li, R. Geda, A. B. Hayes, Y.-H. Chen, P. Chaudhari, E. Z. Zhang, and M. Szegedy. „A Simple Yet Effective Balanced Edge Partition Model for Parallel Computing“. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)* 1.1 (2017), 14:1–14:21. [see pages 16, 26, 207]

[Lie97] J. Lienig. „Physical Design of VLSI Circuits and the Application of Genetic Algorithms“. In: *Evolutionary Algorithms in Engineering Applications*. Springer, 1997, pages 277–292. [see pages 4, 37]

[Lim00] S. K. Lim. „Performance Driven Circuit Partitioning“. PhD thesis. University of California, Los Angeles, 2000. [see pages 4, 37, 96]

[Lim97] S. K. Lim. „Large Scale Circuit Partitioning With Loose/Stable Net Removal And Signal Flow Based Hierarchical Clustering“. Master Thesis. University of California, 1997. [see pages 55, 58]

[Liu+95a] L.-T. Liu, M.-T. Kuo, C.-K. Cheng, and T. C. Hu. „A Replication Cut for Two-Way Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 14.5 (1995), pages 623–630. [see page 17]

[Liu+95b] L.-T. Liu, M.-T. Kuo, C.-K. Cheng, and T. C. Hu. „Performance-Driven Partitioning Using a Replication Graph Approach“. In: *32nd Conference on Design Automation (DAC)*. June 1995, pages 206–210. [see page 17]

[Liu+95c] L.-T. Liu, M.-T. Kuo, S.-C. Huang, and C.-K. Cheng. „A Gradient Method on the Initial Partition of Fiduccia-Mattheyses Algorithm“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1995, pages 229–234. [see page 54]

[LJ15] F. Lotfifar and M. Johnson. „A Multi-Level Hypergraph Partitioning Algorithm Using Rough Set Clustering“. In: *European Conference on Parallel Processing (Euro-Par)*. Springer, Aug. 2015, pages 159–170. [see page 85]

[LJ16] F. Lotfifar and M. Johnson. „A Serial Multilevel Hypergraph Partitioning Algorithm“. In: *CoRR* (2016). arXiv: `1601.01336`. [see page 85]

[LK10] P. Li and A. C. König. „b-Bit Minwise Hashing“. In: *19th International Conference on World Wide Web (WWW)*. Apr. 2010, pages 671–680. [see page 107]

[LLC95] J. Li, J. Lillis, and C.-K. Cheng. „Linear Decomposition Algorithm for VLSI Design Applications“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1995, pages 223–228. [see pages 18, 53]

[Lot16] F. Lotfifar. „Hypergraph Partitioning in the Cloud“. PhD thesis. Durham University, UK, 2016. [see pages 4, 37]

[LR04] K. J. Lang and S. Rao. „A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts“. In: *10th International Integer Programming and Combinatorial Optimization Conference (IPCO)*. 2004, pages 325–337. [see page 136]

[LRU14] J. Leskovec, A. Rajaraman, and J. D. Ullman. „Finding Similar Items“. In: *Mining of Massive Datasets*. 2nd edition. Cambridge University Press, 2014, pages 68–122. [see page 107]

[LS14] D. Luxen and D. Schieferdecker. „Candidate Sets for Alternative Routes in Road Networks“. In: *ACM Journal of Experimental Algorithmics* 19.1 (2014). [see page 2]

[LW98] H. Liu and M. D. F. Wong. „Network-Flow-Based Multiway Partitioning with Area and Pin Constraints“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 17.1 (Jan. 1998), pages 50–59. [see pages 23, 37, 56, 136, 140, 141]

[Mad99] P. H. Madden. „Partitioning by Iterative Deletion“. In: *International Symposium on Physical Design (ISPD)*. Apr. 1999, pages 83–89. [see pages 58, 154]

[Mäk90] E. Mäkinen. „How to Draw a Hypergraph“. In: *International Journal of Computer Mathematics* 34.3–4 (1990), pages 177–185. [see page 12]

[Mal+10] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. „Pregel: A System for Large-Scale Graph Processing“. In: *ACM SIGMOD International Conference on Management of Data*. ACM, June 2010, pages 135–146. [see pages 3, 62, 218]

[Man16] S. S. Mangiafico. *Summary and Analysis of Extension Program Evaluation in R*. `http://www.rcompanion.org/handbook/`. 2016. [see page 31]

[Mat93] D. W. Matula. „A Linear Time $2+\epsilon$ Approximation Algorithm for Edge Connectivity“. In: *4th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Jan. 1993, pages 500–504. [see page 18]

[May+18] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, and K. Rothermel. „HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion“. In: *International Conference on Big Data*. 2018, pages 458–467. [see pages 7, 62, 63, 92, 93, 182, 184, 218]

[MC10] P. Moscato and C. Cotta. „A Modern Introduction to Memetic Algorithms“. In: *Handbook of Metaheuristics*. Springer, 2010, pages 141–183. [see page 22]

[McG12] C. C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012. [see pages 7, 29, 31]

[ME19] R. Mayer and L. Epple. *HYPE*. 2019. URL: `https://github.com/mayerrn/HYPE`. [see page 182]

[Mey08] H. Meyerhenke. „Disturbed Diffusive Processes for Solving Partitioning Problems on Graphs“. PhD thesis. Universität Parderborn, 2008. [see page 182]

[MF17] J. Matejka and G. W. Fitzmaurice. „Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing“. In: *CHI Conference on Human Factors in Computing Systems*. May 2017, pages 1290–1294. [see pages 29, 30]

[Mil+93] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. „Automatic Mesh Partitioning“. In: *Graph Theory and Sparse Matrix Computation*. Springer, 1993, pages 57–84. [see page 2]

[Moh91] B. Mohar. „The Laplacian spectrum of graphs“. In: *Graph Theory, Combinatorics, and Applications* 2 (1991), pages 871–898. [see page 20]

[Mos89] P. Moscato. *On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms*. Technical report C3P Report 826. California Institute of Technology, 1989. [see page 22]

[MP14] Z. Á. Mann and P. A. Papp. „Formula Partitioning Revisited“. In: *5th Pragmatics of SAT workshop*. 2014, pages 41–56. [see pages 23, 25, 96, 136, 147]

[MP17] Z. Á. Mann and P. A. Papp. „Guiding SAT Solving by Formula Partitioning“. In: *International Journal on Artificial Intelligence Tools* 26 (2017), pages 1–37. [see page 25]

[MS08] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008. [see page 19]

[MS10] M. Müller-Hannemann and S. Schirra, editors. *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*. Volume 5971. Lecture Notes in Computer Science. Springer, 2010. [see page 7]

[MS12] H. Meyerhenke and T. Sauerwald. „Beyond Good Partition Shapes: An Analysis of Diffusive Graph Partitioning“. In: *Algorithmica* 64.3 (2012), pages 329–361. [see page 20]

[MS15] D. W. Margo and M.x I. Seltzer. „A Scalable Distributed Graph Partitioner“. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pages 1478–1489. [see page 218]

[MSS14] H. Meyerhenke, P. Sanders, and C. Schulz. „Partitioning Complex Networks via Size-Constrained Clustering“. In: *13th International Symposium on Experimental Algorithms (SEA)*. 2014, pages 351–363. [see pages 5, 24, 27, 94, 121]

[MSS16] H. Meyerhenke, P. Sanders, and C. Schulz. „Partitioning (Hierarchically Clustered) Complex Networks via Size-Constrained Graph Clustering“. In: *Journal of Heuristics* 22.5 (2016), pages 759–782. [see pages 94, 121]

[Mur10] T. Murata. „Modularity for Bipartite Networks“. In: *Data Mining for Social Network Data*. Edited by N. Memon, J. J. Xu, D. L. Hicks, and H. Chen. Springer, 2010. [see page 111]

[MW00] W.-K. Mak and D. F. Wong. „A Fast Hypergraph Min-Cut Algorithm for Circuit Partitioning“. In: *Integration: The VLSI Journal* 30.1 (Nov. 2000), pages 1–11. [see page 18]

[MWM15] R. R. McCune, T. Weninger, and G. Madey. „Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing“. In: *ACM Computing Surveys (CSUR)* 48.2 (Nov. 2015), 25:1–25:39. [see page 206]

[MWS17] R. Matthews, R. Wasserstein, and D. Spiegelhalter. „The ASA's p-value statement, one year on“. In: *Significance* 14.2 (Apr. 2017), pages 38–41. [see page 34]

[MWW95] R. H. Möhring, D. Wagner, and F. Wagner. „Chapter 8 VLSI Network Design“. In: *Network Routing*. Volume 8. Elsevier, 1995, pages 625–712. [see pages 4, 37]

[Net19] W. Lau Neto. *Partitioning hypergraphs for logic synthesis in the field of electronic design automation tools for VLSI*. Private Communication. 2019. [see pages 6, 216]

[New04] M. E. J. Newman. „Analysis of weighted networks“. In: *Physical Review* 70 (5 Nov. 2004). [see page 111]

[NG04] M. E. J. Newman and M. Girvan. „Finding and Evaluating Community Structure in Networks“. In: *Physical Review* 69 (2 Feb. 2004). [see page 110]

[NI00] H. Nagamochi and T. Ibaraki. „A Fast Algorithm for Computing Minimum 3-Way and 4-Way Cuts“. In: *Mathematical Programming* 88.3 (2000), pages 507–520. [see page 18]

[NI92] H. Nagamochi and T. Ibaraki. „Computing Edge-Connectivity in Multigraphs and Capacitated Graphs“. In: *SIAM Journal on Discrete Mathematics* 5.1 (1992), pages 54–66. [see pages 17, 78]

[NK07] H. Nagamochi and Y. Kamidoi. „Minimum Cost Subpartitions in Graphs“. In: *Information Processing Letters* 102.2-3 (2007), pages 79–84. [see page 18]

[NO09] N. Neubauer and K. Obermayer. „Towards Community Detection in k-partite k-Uniform Hypergraphs“. In: *Neural Information Processing Systems (NIPS)*. 2009, pages 1–9. [see page 111]

[NOP87] T.-K. Ng, J. Oldfield, and V. Pitchumani. „Improvements of a Mincut Partition Algorithm“. In: *International Conference on Computer-Aided Design (ICCAD)*. 1987, pages 470–473. [see pages 36, 37]

[NR01] J. Naor and Y. Rabani. „Tree Packing and Approximating k-Cuts“. In: *12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Jan. 2001, pages 26–27. [see page 18]

[Nuz14] R. Nuzzo. „Scientific method: Statistical errors“. In: *Nature* 506.7487 (2014), pages 150–152. [see page 34]

[Oor17] M. van Oort. „Accelerating the Mondriaan Sparse Matrix Partitioning Package“. Master Thesis. 2017. [see pages 86, 92]

[OS10a] V. Osipov and P. Sanders. „n-Level Graph Partitioning“. In: *18th European Symposium on Algorithms (ESA)*. Sept. 2010, pages 278–289. [see pages 5, 97, 98, 102, 103, 114, 116, 117, 121, 122, 125, 135, 154, 170]

[OS10b] V. Osipov and P. Sanders. „n-Level Graph Partitioning“. In: *CoRR* (2010). arXiv: 1004.4024. [see pages 5, 97, 98, 121, 122, 125, 135]

[Osi14] V. Osipov. „Algorithm Engineering for fundamental Sorting and Graph Problems“. PhD thesis. Karlsruhe Institute of Technology, 2014. [see page 170]

[Ouy+00] M. Ouyang, M. Toulouse, K. Thulasiraman, F. Glover, and J. Deogun. „Multilevel Cooperative Search: Application to the Circuit/Hypergraph Partitioning Problem“. In: *International Symposium on Physical Design (ISPD)*. ACM, 2000, pages 192–198. [see page 78]

[Paw92] Z. Pawlak. *Rough Sets: Theoretical Aspects of Reasoning About Data*. Kluwer Academic Publishers, 1992. [see page 85]

[PB14] D. M. Pelt and R. H. Bisseling. „A Medium-Grain Method for Fast 2D Bipartitioning of Sparse Matrices“. In: *28th International Parallel and Distributed Processing Symposium (IPDPS)*. May 2014, pages 529–539. [see page 3]

[Pel19] F. Pellegrini. *SCOTCH Home Page*. 2019. URL: `https://www.labri.fr/perso/pelegrin/scotch/`. [see page 182]

[Pet+15] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. „HDRF: Stream-Based Partitioning for Power-Law Graphs“. In: *24th International Conference on Information and Knowledge Management (CIKM)*. Oct. 2015, pages 243–252. [see page 218]

[PH01] A. Pinar and B. Hendrickson. „Partitioning for Complex Objectives“. In: *15th International Parallel and Distributed Processing Symposium (IPDPS)*. Apr. 2001, page 121. [see page 17]

[PM03] J. Pistorius and M. Minoux. „An improved direct labeling method for the max–flow min–cut computation in large hypergraphs and applications“. In: *International Transactions in Operational Research* 10.1 (2003), pages 1–11. [see page 87]

[PM06] D. A. Papa and I. L. Markov. *Illustration of Partitioning Formats and Partitioner Performance Comparison*. `http://vlsicad.eecs.umich.edu/BK/PART/illustrations/`. 2006. [see page 182]

[PM07] D. A. Papa and I. L. Markov. „Hypergraph Partitioning and Clustering“. In: *Handbook of Approximation Algorithms and Metaheuristics*. 2007. [see pages 4, 14, 23, 25, 37, 42–45, 122, 123, 136, 148]

[PP93] C.-I. Park and Y.-B. Park. „An Efficient Algorithm for VLSI Network Partitioning Problem Using a Cost Function with Balancing Factor“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 12.11 (1993), pages 1686–1694. [see pages 50, 64]

[PQ80] J.-C. Picard and M. Queyranne. „On the Structure of all Minimum Cuts in a Network and Applications“. In: *Combinatorial Optimization II* (1980), pages 8–16. [see page 138]

[Pra59] J. Pratt. „Remarks on Zeros and Ties in the Wilcoxon Signed Rank Procedures“. In: *Journal of the American Statistical Association* 54.287 (1959), pages 655–667. [see page 34]

[PS18] R. J. Preen and J. Smith. „Evolutionary n-Level Hypergraph Partitioning with Adaptive Coarsening“. In: (Mar. 2018). arXiv: 1803.09258. [see page 86]

[PS19] R. J. Preen and J. Smith. „Evolutionary n-level Hypergraph Partitioning with Adaptive Coarsening“. In: *IEEE Transactions on Evolutionary Computation* (2019). [see pages 6, 86, 216]

[PSL90] A. Pothen, H. D. Simon, and K.-P. Liou. „Partitioning Sparse Matrices with Eigenvectors of Graphs“. In: *SIAM Journal on Matrix Analysis and Applications* 11.3 (July 1990), pages 430–452. [see page 20]

[QKD13] A. Quamar, K. A. Kumar, and A. Deshpande. „SWORD: scalable workload-aware data placement for transactional workloads“. In: *16th Conference on Extending Database Technology (EDBT)*. Mar. 2013, pages 430–441. [see page 2]

[Räc08] H. Räcke. „Optimal Hierarchical Decompositions for Congestion Minimization in Networks“. In: *40th ACM Symposium on Theory of Computing (STOC)*. May 2008, pages 255–264. [see page 19]

[RB12] S. Rajamanickam and E. G. Boman. „Parallel Partitioning with Zoltan: Is Hypergraph Partitioning Worth It?“ In: *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*. 2012, pages 37–52. [see page 92]

[RB87] K. Roberts and Preas B. *MCNC*. Technical report. Physical Design Workshop, 1987. [see pages 45, 60]

[RDJ94] B. M. Riess, K. Doll, and F. M. Johannes. „Partitioning Very Large Circuits Using Analytical Placement Techniques“. In: *31st Conference on Design Automation (DAC)*. June 1994, pages 646–651. [see page 52]

[Riy03] S. Riyavong. *Experiments on Sparse Matrix Partitioning*. Technical report CERFACS Working Note WN/PA/03/32. CERFACS, 2003. [see page 184]

[RM03] A. Ramani and I. L. Markov. „Combining Two Local Search Approaches to Hypergraph Partitioning“. In: *18th International Joint Conference on Artificial Intelligence (IJCAI)*. 2003. [see page 61]

[RS93] K. Roy and C. Sechen. „A Timing Driven N-Way Chip and Multi-Chip Partitioner“. In: *International Conference on Computer-Aided Design (ICCAD)*. IEEE. 1993, pages 240–247. [see pages 37, 69]

[RS94] N. J. Radcliffe and P. D. Surry. „Formal Memetic Algorithms“. In: *AISB Workshop On Evolutionary Computing*. 1994, pages 1–16. [see page 22]

[RSS18] H. Räcke, R. Schwartz, and R. Stotz. „Trees for Vertex Cuts, Hypergraph Cuts and Minimum Hypergraph Bisection“. In: *30th Symposium on Parallelism in Algorithms and Architectures (SPAA)*. July 2018, pages 23–32. [see page 19]

[Rut64] R. A. Rutman. „An Algorithm for Placement of Interconnected Elements Based on Minimum Wire Length“. In: *Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS)*. ACM, Apr. 1964, pages 477–491. [see page 2]

[Saa00a] Y. Saab. „A New 2-Way Multi-Level Partitioning Algorithm“. In: *VLSI Design* 3 (2000), pages 301–310. [see pages 77, 80]

[Saa00b] Y. Saab. „A New Effective And Efficient Multi-Level Partitioning Algorithm“. In: *Design, Automation and Test in Europe (DATE)*. Mar. 2000, pages 112–116. [see pages 77, 80]

[Saa02] Y. Saab. „A Contraction-Based Ratio-Cut Partitioning Algorithm“. In: *VLSI Design* 2 (2002), pages 485–489. [see pages 77, 88]

[Saa04] Y. Saab. „An Effective Multilevel Algorithm for Bisecting Graphs and Hypergraphs“. In: *IEEE Transactions on Computers* 53.6 (2004), pages 641–652. [see page 80]

[SAA17] R. O. Selvitopi, S. Acer, and C. Aykanat. „A Recursive Hypergraph Bipartitioning Framework for Reducing Bandwidth and Latency Costs Simultaneously“. In: *IEEE Transactions on Parallel and Distributed Systems* 28.2 (2017), pages 345–358. [see page 3]

[Saa95] Y. Saab. „A Fast and Robust Network Bisection Algorithm“. In: *IEEE Transactions on Computers* 44.7 (1995), pages 903–913. [see pages 36, 68, 80, 88, 98]

[Saa99] Y. Saab. „A Ratio-Cut Partitioning Algorithm Using Node Contraction“. In: *42nd Midwest Symposium on Circuits and Systems*. Aug. 1999, pages 210–213. [see pages 77, 88]

[San09] P. Sanders. „Algorithm Engineering - An Attempt at a Definition“. In: *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. 2009, pages 321–340. [see pages 7, 9]

[San86] L. A. Sanchis. *Multiple-Way Network Partitioning*. Technical report. University of Rochester, Department of Computer Science, 1986. [see pages 47, 49, 50, 53, 57, 75, 78]

[San89] L. A. Sanchis. „Multiple-Way Network Partitioning“. In: *IEEE Transactions on Computers* 38.1 (1989), pages 62–81. [see pages 36, 44, 47, 49, 50, 53, 57, 70, 81, 96, 124]

[San93] L. A. Sanchis. „Multiple-Way Network Partitioning with Different Cost Functions“. In: *IEEE Transactions on Computers* 42.12 (1993), pages 1500–1504. [see pages 51, 57, 72, 124]

[SC88] C. Sechen and D. Chen. „An Improved Objective Function for Mincut Circuit Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. 1988, pages 502–505. [see pages 36, 37]

[Sch+15a] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. „k-way Hypergraph Partitioning via n-Level Recursive Bisection“. In: *Computing Research Repository (CoRR)* (Nov. 2015), pages 1–21. arXiv: 1511.03137. [see pages 4, 97]

[Sch+16a] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. „k-way Hypergraph Partitioning via n-Level Recursive Bisection“. In: *18th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Jan. 2016, pages 53–67. [see pages 4, 11, 23, 25, 28, 97–99, 102–105, 133, 151, 155, 170, 181, 231]

[Sch+18a] S. Schlag, C. Schulz, D. Seemaier, and D. Strash. „Scalable Edge Partitioning“. In: *Computing Research Repository (CoRR)* (Oct. 2018), pages 1–17. arXiv: 1808.06411. [see page 182]

[Sch+19a] S. Schlag, C. Schulz, D. Seemaier, and D. Strash. „Scalable Edge Partitioning“. In: *21st Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2019, pages 211–225. [see pages 6, 11, 24, 26, 182, 207, 208, 216–218]

[Sch07] S. E. Schaeffer. „Graph Clustering“. In: *Computer Science Review* 1.1 (Aug. 2007), pages 27–64. [see page 110]

[Sch13b] C. Schulz. „High Quality Graph Partitioning“. PhD thesis. Karlsruhe Institute of Technology, 2013. [see pages 27, 28, 104, 121–123, 185]

[Sch19] S. Schlag. *Benchmark Sets used in the Dissertation of Sebastian Schlag*. KITopen. 2019. DOI: 10.5445/IR/1000098881. [see pages 9, 23]

[Sco05] A. D. Scott. „Judicious Partitions and Related Problems“. In: *20th British Combinatorial Conference*. July 2005, pages 95–117. [see page 16]

[SDJ91] G. Sigl, K. Doll, and F. M. Johannes. „Analytical Placement: A Linear or a Quadratic Objective Function?“ In: *28th Conference on Design Automation (DAC)*. June 1991, pages 427–432. [see page 52]

[SG02a] A. Strehl and J. Ghosh. „Cluster Ensembles - A Knowledge Reuse Framework for Combining Multiple Partitions“. In: *Journal of Machine Learning Research* 3 (2002), pages 583–617. [see page 16]

[SG02b] A. Strehl and J. Ghosh. „Cluster Ensembles - A Knowledge Reuse Framework for Combining Partitionings“. In: *18th National Conference on Artificial Intelligence (AAAI) / 14th Conference on Innovative Applications of Artificial Intelligence (IAAI)*. 2002, pages 93–99. [see page 16]

[Sha19] R. Shaydulin. *Aggregative Coarsening for Multilevel Hypergraph Partitioning*. 2019. URL: https://github.com/rsln-s/aggregative-coarsening-for-multilevel-hypergraph-partitioning. [see page 182]

[Sim91] H. D. Simon. „Partitioning of Unstructured Problems for Parallel Processing“. In: *Computing Systems in Engineering* 2.2 (1991), pages 135–148. [see page 20]

[SK03] N. Selvakkumaran and G. Karypis. „Multi-Objective Hypergraph Partitioning Algorithms for Cut and Maximum Subdomain Degree Minimization“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 2003, pages 726–733. [see page 17]

[SK06] N. Selvakkumaran and G. Karypis. „Multiobjective Hypergraph-Partitioning Algorithms for Cut and Maximum Subdomain-Degree Minimization“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 25.3 (2006), pages 504–517. [see page 17]

[SK72] D. G. Schweikert and B. W. Kernighan. „A Proper Model for the Partitioning of Electrical Circuits“. In: *9th Conference on Design Automation (DAC)*. June 1972, pages 57–62. [see pages 2, 3, 14, 35, 40, 41, 44, 62, 65, 112, 113]

[SK93] H. Shin and C. Kim. „A Simple Yet Effective Technique for Partitioning“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 1.3 (1993), pages 380–386. [see pages 67, 88]

[SK94] M. Shih and E. S. Kuh. „Circuit Partitioning Under Capacity and I/O Constraints“. In: *Custom Integrated Circuits Conference (CICC)*. IEEE, May 1994, pages 659–662. [see page 37]

[SKK00] K. Schloegel, G. Karypis, and V. Kumar. „Graph partitioning for high performance scientific simulations“. In: *Sourcebook of Parallel Computing*. Morgan Kaufmann, 2000, pages 491–541. [see page 2]

[Slo+17] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri. „Partitioning Trillion-Edge Graphs in Minutes“. In: *31nd International Parallel and Distributed Processing Symposium (IPDPS)*. June 2017, pages 646–655. [see page 2]

[SM16] C. L. Staudt and H. Meyerhenke. „Engineering Parallel Algorithms for Community Detection in Massive Networks“. In: *IEEE Transactions on Parallel and Distributed Systems* 27.1 (Jan. 2016), pages 171–184. [see page 217]

[SMR14] G. M. Slota, K. Madduri, and S. Rajamanickam. „PuLP: Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks“. In: *International Conference on Big Data*. Oct. 2014, pages 481–490. [see pages 2, 62]

[SNK95] T. Shibuya, I. Nitta, and K. Kawamura. „SMINCUT: VLSI Placement Tool Using Min-Cut“. In: *Fujitsu Scientific & Technical Journal* 31.2 (1995), pages 197–207. [see page 56]

[Spi07] D. A. Spielman. „Spectral Graph Theory and its Applications“. In: *48th Symposium on Foundations of Computer Science (FOCS)*. Oct. 2007, pages 29–38. [see page 20]

[Spi12] D. Spielman. „Spectral Graph Theory“. In: *Combinatorial Scientific Computing*. Edited by U. Naumann and O. Schenk. Chapman & Hall/CRC, 2012. Chapter 18, pages 495–517. [see page 20]

[SR89a] Y. Saab and V. B. Rao. „A Stochastic Algorithm for Circuit Bi-Partitioning“. In: *First Great Lakes Computer Science Conference*. Oct. 1989, pages 313–321. [see pages 36, 48]

[SR89b] Y. Saab and V. B. Rao. „An Evolution-Based Approach to Partitioning ASIC Systems“. In: *26th Conference on Design Automation (DAC)*. ACM, June 1989, pages 767–770. [see pages 36, 48]

[SR90] Y. Saab and V. B. Rao. „Stochastic Evolution: a Fast Effective Heuristic for Some Generic Layout Problems“. In: *27th Conference on Design Automation (DAC)*. June 1990, pages 26–31. [see page 48]

[SS11] P. Sanders and C. Schulz. „Engineering Multilevel Graph Partitioning Algorithms“. In: *19th European Symposium on Algorithms (ESA)*. Springer, 2011, pages 469–480. [see pages 6, 7, 99, 122, 125, 136, 144, 170, 177, 181, 182]

[SS12] P. Sanders and C. Schulz. „Distributed Evolutionary Graph Partitioning“. In: *12th Workshop on Algorithm Engineering and Experimentation, (ALENEX)*. 2012, pages 16–29. [see pages 6, 31, 172–174, 176]

[SS13] P. Sanders and C. Schulz. „Think Locally, Act Globally: Highly Balanced Graph Partitioning“. In: *12th International Symposium on Experimental Algorithms (SEA)*. Springer, June 2013, pages 164–175. [see pages 154, 182]

[SS18a] R. Shaydulin and I. Safro. „Aggregative Coarsening for Multilevel Hypergraph Partitioning“. In: *CoRR* (2018). arXiv: 1802.09610. [see pages 86, 182]

[SS18b] R. Shaydulin and I. Safro. „Aggregative Coarsening for Multilevel Hypergraph Partitioning“. In: *17th International Symposium on Experimental Algorithms (SEA)*. Volume 103. 2018, 2:1–2:15. [see pages 7, 86, 182]

[SS95] W.-J. Sun and C. Sechen. „Efficient and Effective Placement for Very Large Circuits“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 14.3 (1995), pages 349–359. [see page 16]

[SSC19] R. Shaydulin, I. Safro, and J. Chen. „Relaxation-Based Coarsening for Multilevel Hypergraph Partitioning“. In: *Multiscale Modeling & Simulation* 17.1 (2019), pages 482–506. [see pages 86, 182]

[SSS19b] J. Sybrandt, R. Shaydulin, and I. Safro. „Hypergraph Partitioning With Embeddings“. In: *arXiv e-prints* (Sept. 2019), arXiv:1909.04016. eprint: 1909.04016. [see pages 6, 216, 218]

[ST97] H. D. Simon and S.-H. Teng. „How Good is Recursive Bisection?“ In: *SIAM Journal on Scientific Computing* 18.5 (1997), pages 1436–1445. [see pages 15, 75, 95, 161]

[STA12] R. O. Selvitopi, A. Turk, and C. Aykanat. „Replicated Partitioning for Undirected Hypergraphs“. In: *Journal of Parallel and Distributed Computing* 72.4 (2012), pages 547–563. [see page 17]

[SU72] D. M. Schuler and E. G. Ulrich. „Clustering and Linear Placement“. In: *9th Conference on Design Automation (DAC)*. June 1972, pages 50–56. [see page 69]

[SV95] H. Saran and V. V. Vazirani. „Finding k Cuts within Twice the Optimal“. In: *SIAM Journal on Computing* 24.1 (1995), pages 101–108. [see page 18]

[SW09] K. Suzuki and K. Wakita. „Extracting Multi-facet Community Structure from Bipartite Networks“. In: *12th International Conference on Computational Science and Engineering (CSE)*. 2009, pages 312–319. [see page 111]

[SW11] P. Sanders and D. Wagner. „Algorithm Engineering“. In: *it - Information Technology* 53.6 (2011), pages 263–265. [see page 7]

[SW13] P. Sanders and D. Wagner. „Algorithm Engineering“. In: *Informatik Spektrum* 36.2 (2013), pages 187–190. [see page 7]

[SW97] M. Stoer and F. Wagner. „A Simple Min-Cut Algorithm“. In: *Journal of the ACM (JACM)* 44.4 (1997), pages 585–591. [see page 18]

[SWC04] A. J. Soper, C. Walshaw, and M. Cross. „A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning“. In: *J. Global Optimization* 29.2 (2004), pages 225–241. [see page 26]

[Tal09] E.-G. Talbi. *Metaheuristics - From Design to Implementation*. Wiley, 2009. [see page 21]

[Tan+17] T. Tan, J. Gui, S. Wang, S. Gao, and W. Yang. „An Efficient Algorithm for Judicious Partition of Hypergraphs“. In: *11th International Conference on Combinatorial Optimization and Applications (COCOA)*. Dec. 2017, pages 466–474. [see page 16]

[Tao02] L. Tao. *An Efficient Multiway Hypergraph Partitioning Algorithm for VLSI Layout*. Technical report. Pace University, 2002. [see page 61]

[Tho08] M. Thorup. „Minimum k-Way Cuts Via Deterministic Greedy Tree Packing“. In: *40th ACM Symposium on Theory of Computing (STOC)*. May 2008, pages 159–166. [see page 18]

[TK04a] A. Trifunović and W. J. Knottenbelt. „A Parallel Algorithm for Multilevel k-Way Hypergraph Partitioning“. In: *3rd International Symposium on Parallel and Distributed Computing (ISPDC)*. July 2004, pages 114–121. [see pages 79, 81, 84, 114, 124]

[TK04b] A. Trifunović and W. J. Knottenbelt. „Par kway 2.0: A Parallel Multilevel Hypergraph Partitioning Tool“. In: *19th International Symposium on Computer and Information Sciences (ISCIS)*. Oct. 2004, pages 789–800. [see pages 79, 81, 84]

[TK04c] A. Trifunović and W. J. Knottenbelt. „Towards a Parallel Disk-Based Algorithm for Multilevel k-way Hypergraph Partitioning“. In: *18th International Parallel and Distributed Processing Symposium (IPDPS)*. Apr. 2004. [see page 79]

[TK08] A. Trifunović and W. J. Knottenbelt. „Parallel Multilevel Algorithms for Hypergraph Partitioning“. In: *Journal of Parallel and Distributed Computing* 68.5 (2008), pages 563–581. [see pages 6, 84, 114, 124]

[Tri06] A. Trifunović. „Parallel Algorithms for Hypergraph Partitioning“. PhD thesis. Imperial College London, UK, 2006. [see pages 4, 37, 79, 95, 183]

[Tuk57] J. W. Tukey. „On the Comparative Anatomy of Transformations“. In: *The Annals of Mathematical Statistics* 28.3 (Sept. 1957), pages 602–632. [see page 31]

[Tuk77a] J. W. Tukey. „Box-and-whisker plots“. In: *Exploratory data analysis* (1977), pages 39–43. [see page 33]

[Tuk77b] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977. [see page 31]

[TVV85] L. Törnqvist, P. Vartia, and Y. O. Vartia. „How Should Relative Changes be Measured?“ In: *The American Statistician* 39.1 (1985), pages 43–46. [see page 29]

[UA04] B. Uçar and C. Aykanat. „Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies“. In: *SIAM Journal on Scientific Computing* 25.6 (2004), pages 1837–1859. [see pages 3, 80, 81, 95, 124, 136]

[UA07] B. Uçar and C. Aykanat. „Revisiting Hypergraph Models for Sparse Matrix Partitioning“. In: *SIAM Review* 49.4 (2007), pages 595–603. [see page 3]

[UÇ10] B. Uçar and Ü. V. Çatalyürek. „On the Scalability of Hypergraph Models for Sparse Matrix Partitioning“. In: *18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*. Feb. 2010, pages 593–600. [see page 3]

[Uça99] B. Uçar. „Partitioning Sparse Rectangular Matrices for Parallel Computing of AA^TX“. Master Thesis. Bilkent University, 1999. [see page 81]

[VB05] B. Vastenhouw and R. H. Bisseling. „A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication“. In: *SIAM Review* 47.1 (2005), pages 67–95. [see pages 3, 7, 81, 96, 114]

[VH90] A. Vannelli and S. W. Hadley. „A Gomory-Hu Cut Tree Representation of a Netlist Partitioning Problem“. In: *IEEE Transactions on Circuits and Systems* 37.9 (Sept. 1990), pages 1133–1139. [see page 14]

[Vij90] G. Vijayan. „Partitioning Logic on Graph Structures to Minimize Routing Cost“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 9.12 (1990), pages 1326–1334. [see page 49]

[Vis+12] N. Viswanathan, C. J. Alpert, C. C. N. Sze, Z. Li, and Y. Wei. „The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite“. In: *49th Conference on Design Automation (DAC)*. ACM, June 2012, pages 774–782. [see pages 23, 147]

[WA98] S. Wichlund and E. Aas. „On Multilevel Circuit Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1998, pages 505–511. [see pages 7, 73, 174, 176]

[Wal03] C. Walshaw. „An Exploration of Multilevel Combinatorial Optimisation“. In: *Multilevel Optimization in VLSICAD*. Edited by J. Cong and J. R. Shinnerl. Springer, 2003, pages 71–124. [see pages 93, 94]

[Wal04] C. Walshaw. „Multilevel Refinement for Combinatorial Optimisation Problems“. In: *Annals of Operations Research* 131.1–4 (2004), pages 325–372. [see pages 172, 174]

[Wan+00] M. Wang, S. Lim, J. Cong, and M. Sarrafzadeh. „Multi-Way Partitioning Using Bi-Partition Heuristics“. In: *Asia South Pacific Design Automation Conference (ASP-DAC)*. 2000, pages 667–672. [see page 124]

[Wan+14] L. Wang, Y. Xiao, B. Shao, and H. Wang. „How to partition a billion-node graph“. In: *30th International Conference on Data Engineering (ICDE)*. Apr. 2014, pages 568–579. [see page 2]

[WC89] Y.-C. A. Wei and C.-K. Cheng. „Towards Efficient Hierarchical Designs by Ratio Cut Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1989, pages 298–301. [see pages 16, 44, 49, 69, 70, 74]

[WC90] Y.-C. Wei and C.-K. Cheng. „A Two-Level Two-Way Partitioning Algorithm“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1990, pages 516–519. [see pages 66, 88]

[WC91] Y.-C. A. Wei and C.-K. Cheng. „Ratio Cut Partitioning for Hierarchical Designs“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 10.7 (1991), pages 911–921. [see pages 16, 49]

[WCE95] C. H. Walshaw, M. Cross, and M. G. Everett. „A Localized Algorithm for Optimizing Unstructured Mesh Partitions“. In: *The International Journal of Supercomputer Applications and High Performance Computing* 9.4 (1995), pages 280–295. [see page 70]

[Wil16] S. Wild. „Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential“. PhD thesis. Technische Universität Kaiserslautern, 2016, pages 1–377. [see page 10]

[Wil45] F. Wilcoxon. „Individual Comparisons by Ranking Methods“. In: *Biometrics Bulletin* 1.6 (1945), pages 80–83. [see page 34]

[WK93] N. S. Woo and J. Kim. „An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementation.“ In: *30th Conference on Design Automation (DAC)*. June 1993, pages 202–207. [see page 37]

[WL16] R. Wasserstein and N. Lazar. „The ASA Statement on p-Values: Context, Process, and Purpose“. In: *The American Statistician* 70.2 (2016), pages 129–133. [see page 34]

[WS02] L. Wang and H. Selvaraj. „Performance Driven Circuit Clustering and Partitioning“. In: *Information Technology: Coding and Computing (ITCC)*. IEEE, 2002, pages 352–354. [see page 37]

[Wu+03] Y.-L. Wu, C.-C. Cheung, D. I. Cheng, and H. Fan. „Further Improve Circuit Partitioning Using GBAW Logic Perturbation Techniques“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11.3 (2003), pages 451–460. [see page 37]

[WW93] D. Wagner and F. Wagner. „Between Min Cut and Graph Bisection“. In: *18th International Symposium on Mathematical Foundations of Computer Science (MFCS)*. 1993, pages 744–750. [see page 19]

[Xia08] M. Xiao. „Finding Minimum 3-Way Cuts in Hypergraphs“. In: *5th International Conference on Theory and Applications of Models of Computation (TAMC)*. Springer, Apr. 2008, pages 270–281. [see page 18]

[Xia10] M. Xiao. „Finding Minimum 3-Way Cuts in Hypergraphs“. In: *Information Processing Letters* 110.14-15 (2010), pages 554–558. [see page 18]

[YA14] V. Yazici and C. Aykanat. „Constrained Min-Cut Replication for k-Way Hypergraph Partitioning“. In: *INFORMS Journal on Computing* 26.2 (2014), pages 303–320. [see page 17]

[Yan+16] W. Yang, G. Wang, L. Ma, and S. Wu. „A Distributed Algorithm for Balanced Hypergraph Partitioning“. In: *10th Asia-Pacific Services Computing Conference (APSCC)*. Nov. 2016, pages 477–490. [see page 16]

[Yan+18a] W. Yang, L. Ma, R. Cui, and G. Wang. „Hypergraph Partitioning for Big Data Applications“. In: *SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*. Oct. 2018, pages 1705–1710. [see page 16]

[Yan+18b] W. Yang, G. Wang, K.-K. R. Choo, and S. Chen. „HEPart: A Balanced Hypergraph Partitioning Algorithm for Big Data Applications“. In: *Future Generation Computer Systems* 83 (2018), pages 250–268. [see page 16]

[Yan+94] W.-P. Yang, J.-S. Ker, Y.-H. Kuo, and Y.-K. Ho. „Two-Level Partitioning Algorithm with Stable Performance“. In: *IEE Proceedings - Circuits, Devices and Systems* 141.3 (June 1994), pages 197–202. [see pages 68, 88]

[YB09] A. N. Yzelman and R. H. Bisseling. „Cache-Oblivious Sparse Matrix–Vector Multiplication by Using Sparse Matrix Partitioning Methods“. In: *SIAM Journal on Scientific Computing* 31.4 (2009), pages 3128–3154. [see page 3]

[YC00] E. Yarack and J. Carletta. „An Evaluation of Move-Based Multi-Way Partitioning Algorithms“. In: *International Conference on Computer Design*. Sept. 2000, pages 363–369. [see page 61]

[YCL91a] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. „A General Purpose Multiple Way Partitioning Algorithm“. In: *28th Conference on Design Automation (DAC)*. ACM, 1991, pages 421–426. [see pages 66, 69, 73, 88]

[YCL91b] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. „An Experimental Evaluation of Partitioning Algorithms“. In: *4th Annual IEEE International ASIC Conference*. Sept. 1991. [see page 66]

[YCL92] C.-W. Yeh, C.-K. Cheng, and T. Y. Lin. „A Probabilistic Multicommodity-Flow Solution to Circuit Clustering Problems“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1992, pages 428–431. [see page 37]

[YCL94] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. „A General Purpose, Multiple-Way Partitioning Algorithm“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 13.12 (1994), pages 1480–1488. [see pages 66, 69, 73, 88]

[YCL95] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. „Optimization by Iterative Improvement: An Experimental Evaluation on Two-Way Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 14.2 (1995), pages 145–153. [see page 66]

[YI13] J. R. Yaros and T. Imielinski. „Imbalanced Hypergraph Partitioning and Improvements for Consensus Clustering“. In: *IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI)*. Nov. 2013, pages 358–365. [see page 16]

[YM90] J. S. Yih and P. Mazumder. „A Neural Network Design for Circuit Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 9.12 (Dec. 1990), pages 1265–1271. [see page 2]

[YW08] H. H. Yang and M. D. F. Wong. „Circuit Partitioning: A Network-Flow-Based Balanced Min-Cut Approach“. In: *Encyclopedia of Algorithms*. Edited by M.-Y. Kao. Springer, 2008. [see pages 52, 87]

[YW94] H. H. Yang and M. D. F. Wong. „Efficient Network Flow Based Min-Cut Balanced Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. 1994, pages 50–55. [see pages 52, 87]

[YW95] H. H. Yang and M. D. F. Wong. „New Algorithms for Min-Cut Replication in Partitioned Circuits“. In: *International Conference on Computer-Aided Design (ICCAD)*. Nov. 1995, pages 216–222. [see page 17]

[YW96] H. H. Yang and M. D. F. Wong. „Efficient Network Flow Based Min-Cut Balanced Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 15.12 (Dec. 1996), pages 1533–1540. [see pages 52, 87, 136]

[Zha+17] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. „Graph Edge Partitioning via Neighborhood Heuristic“. In: *23rd International Conference on Knowledge Discovery and Data Mining (KDD)*. Aug. 2017, pages 605–614. [see pages 62, 218]

[Zha00] Z. Z. Zhao. „An Effective Algorithm for Multiway Hypergraph Partitioning“. Master Thesis. Concordia University, 2000. [see page 61]

[Zie12] F. Ziegler. „n-Level Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, July 2012. [see page 97]

[ZTY15] Y. Zhang, Y. C. Tang, and G. Y. Yan. „On Judicious Partitions of Hypergraphs with Edges of Size at Most 3“. In: *European Journal of Combinatorics* 49 (2015), pages 232–239. [see page 16]

[ZTZ02] Z. Zhao, L. Tao, and Y. Zhao. „An Effective Algorithm for Multiway Hypergraph Partitioning“. In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 49.8 (2002), pages 1079–1092. [see page 61]

Publications and Supervised Theses

Journal Articles

[1] T. Heuer, P. Sanders, and S. Schlag. „Network Flow-Based Refinement for Multilevel Hypergraph Partitioning“. In: *ACM Journal of Experimental Algorithmics (JEA)* 24.1 (Sept. 2019), 2.3:1–2.3:36.

Articles in Conference Proceedings

[1] I. Baar, L. Hübner, P. Oettig, A. Zapletal, S. Schlag, A. Stamatakis, and B. Morel. „Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning“. In: *18th International Workshop on High Performance Computational Biology (HiCOMB)*. IEEE, 2019.

[2] S. Schlag, C. Schulz, D. Seemaier, and D. Strash. „Scalable Edge Partitioning“. In: *21st Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2019, pages 211–225.

[3] S. Schlag, M. Schmitt, and C. Schulz. „Faster Support Vector Machines“. In: *21st Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2019, pages 199–210.

[4] T. Heuer, P. Sanders, and S. Schlag. „Network Flow-Based Refinement for Multilevel Hypergraph Partitioning“. In: *17th International Symposium on Experimental Algorithms (SEA)*. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2018, 1:1–1:19.

[5] R. Andre, S. Schlag, and C. Schulz. „Memetic Multilevel Hypergraph Partitioning“. In: *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, July 2018, pages 347–354.

[6] T. Heuer and S. Schlag. „Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure“. In: *16th International Symposium on Experimental Algorithms (SEA)*. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2017, 21:1–21:19.

[7] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. „Engineering a Direct k-way Hypergraph Partitioning Algorithm“. In: *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Jan. 2017, pages 28–42.

[8] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. „Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++“. In: *International Conference on Big Data*. IEEE, Dec. 2016, pages 172–183.

[9] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. „k-way Hypergraph Partitioning via n-Level Recursive Bisection“. In: *18th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Jan. 2016, pages 53–67.

[10] P. Sanders, S. Schlag, and I. Müller. „Communication Efficient Algorithms for Fundamental Big Data Problems“. In: *IEEE International Conference on Big Data*. IEEE, Oct. 2013, pages 15–23.

Technical Reports

[1] I. Baar, L. Hübner, P. Oettig, A. Zapletal, S. Schlag, A. Stamatakis, and B. Morel. „Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning“. In: *The Preprint Server for Biology* (2019), pages 1–10. bioRxiv: `579318`.

[2] S. Schlag, C. Schulz, D. Seemaier, and D. Strash. „Scalable Edge Partitioning“. In: *Computing Research Repository (CoRR)* (Oct. 2018), pages 1–17. arXiv: `1808.06411`.

[3] S. Schlag, M. Schmitt, and C. Schulz. „Faster Support Vector Machines“. In: *Computing Research Repository (CoRR)* (Oct. 2018), pages 1–12. arXiv: `1808.06394`.

[4] T. Heuer, P. Sanders, and S. Schlag. „Network Flow-Based Refinement for Multilevel Hypergraph Partitioning“. In: *Computing Research Repository (CoRR)* (Feb. 2018), pages 1–28. arXiv: `1802.03587`.

[5] R. Andre, S. Schlag, and C. Schulz. „Memetic Multilevel Hypergraph Partitioning“. In: *Computing Research Repository (CoRR)* (Feb. 2018), pages 1–16. arXiv: `1710.01968`.

[6] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. „Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++“. In: *Computing Research Repository (CoRR)* (Aug. 2016), pages 1–15. arXiv: `1608.05634`.

[7] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. „k-way Hypergraph Partitioning via n-Level Recursive Bisection“. In: *Computing Research Repository (CoRR)* (Nov. 2015), pages 1–21. arXiv: `1511.03137`.

[8] V. Henne, H. Meyerhenke, P. Sanders, S. Schlag, and C. Schulz. „n-Level Hypergraph Partitioning“. In: *Computing Research Repository (CoRR)* (May 2015), pages 1–23. arXiv: `1505.00693`.

Theses

[1] S. Schlag. „Distributed Duplicate Removal“. Master Thesis. Karlsruhe Institute of Technology, July 2013.

[2] S. Schlag. „Transportation Management in the Cloud – A Prototype for Tendering Scenarios“. Bachelor Thesis. Baden-Württemberg Cooperative State University, Karlsruhe, Sept. 2010.

Supervised Theses

[1] T. Ribizel. „Communication Optimization by Data Replication for Distributed Graph Algorithms“. Master Thesis. Karlsruhe Institute of Technology, Sept. 2019.

[2] P. Firnkes. „Throughput Optimization in a Distributed Database System via Hypergraph Partitioning“. Master Thesis. Karlsruhe Institute of Technology, Apr. 2019.

[3] C. Mercatoris. „Combining Recursive Bisection and k-way Local Search for Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, Nov. 2018.

[4] M. Schmitt. „Support Vector Machines via Multilevel Label Propagation“. Bachelor Thesis. Karlsruhe Institute of Technology, June 2018.

[5] C. Öhl. „Algorithm Configuration for Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, May 2018.

[6] T. Heuer. „High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations“. Master Thesis. Karlsruhe Institute of Technology, Jan. 2018.

[7] R. Andre. „Evolutionary Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, Nov. 2017.

[8] Y. Kolev. „Better Recursive Graph Bisection“. Bachelor Thesis. Karlsruhe Institute of Technology, Nov. 2017.

[9] D. Seemaier. „Engineering Graph Partitioning Algorithms to Minimize Communication Volume“. Bachelor Thesis. Karlsruhe Institute of Technology, Nov. 2017.

[10] O. Kolev. „Smart Local Search in Hypergraph Partitioning“. Diploma Thesis. Karlsruhe Institute of Technology, Oct. 2017.

[11] T. Heuer. „Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, Aug. 2015.

[12] V. Henne. „Label Propagation for Hypergraph Partitioning“. Master Thesis. Karlsruhe Institute of Technology, May 2015.