# Scalable High-Quality Graph and 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

**Tobias Heuer**

aus Bietigheim (Baden)

Tag der mündlichen Prüfung: 26.10.2022

Erster Gutachter: Herr Prof. Dr. Peter Sanders
Zweiter Gutachter: Herr Prof. Dr. Ümit Çatalyürek

*In loving memories of my mother*

# Abstract

The *balanced hypergraph partitioning problem* (HGP) asks for a partition of the node set of a hypergraph into $k$ blocks of roughly equal size, such that an objective function defined on the hyperedges is minimized. In this work, we optimize the *connectivity metric*, which is the most prominent objective function for HGP.

The hypergraph partitioning problem is NP-hard and there exists no constant factor approximation. Thus, heuristic algorithms are used in practice with the *multilevel scheme* as the most successful approach to solve the problem: First, the input hypergraph is *coarsened* to obtain a hierarchy of successively smaller and structurally similar approximations. The smallest hypergraph is then *initially partitioned* into $k$ blocks, and subsequently, the contractions are reverted level-by-level, and, on each level, *local search* algorithms are used to improve the partition (*refinement phase*).

In recent years, several new techniques were developed for sequential multilevel partitioning that substantially improved solution quality at the cost of an increased running time. These developments divide the landscape of existing partitioning algorithms into systems that either aim for speed or high solution quality with the former often being more than an order of magnitude faster than the latter. Due to the high running times of the best sequential algorithms, it is currently not feasible to partition the largest real-world hypergraphs with the highest possible quality. Thus, it becomes increasingly important to parallelize the techniques used in these algorithms. However, existing state-of-the-art parallel partitioners currently do not achieve the same solution quality as their sequential counterparts because they use comparatively weak components that are easier to parallelize. Moreover, there has been a recent trend toward simpler methods for partitioning large hypergraphs that even omit the multilevel scheme.

In contrast to this development, we present two shared-memory multilevel hypergraph partitioners with parallel implementations of techniques used by the highest-quality sequential systems. Our first multilevel algorithm uses a parallel clustering-based coarsening scheme, which uses substantially fewer locking mechanisms than previous approaches. The contraction decisions are guided by the community structure of the input hypergraph obtained via a parallel community detection algorithm. For initial partitioning, we implement parallel multilevel recursive bipartitioning with a novel work-stealing approach and a portfolio of initial bipartitioning techniques to compute an initial solution. In the refinement phase, we use three different parallel improvement algorithms: label propagation refinement, a highly-localized direct $k$-way FM algorithm, and a novel parallelization of flow-based refinement. These algorithms build on our highly-engineered partition data structure, for which we propose several novel techniques to compute accurate gain values of node moves in the parallel setting.

Our second multilevel algorithm parallelizes the $n$-level partitioning scheme used in the highest-quality sequential partitioner KaHyPar. Here, only a single node is contracted on each level, leading to a hierarchy with approximately $n$ levels where $n$ is the number of nodes. Correspondingly, in each refinement step, only a single node is uncontracted, allowing a highly-localized search for improvements. We show that this approach, which seems inherently sequential, can be parallelized efficiently without compromises in solution quality. To this end, we design a forest-based representation of contractions from which we derive a *feasible* parallel schedule of the contraction operations that we apply on a novel dynamic hypergraph data structure *on-the-fly*. In the uncoarsening phase, we decompose the contraction forest into *batches*, each containing a fixed number of nodes. We then uncontract each batch in parallel and use highly-localized versions of our refinement algorithms to improve the partition around the uncontracted nodes.

We further show that existing sequential partitioning algorithms considerably struggle to find balanced partitions for *weighted* real-world hypergraphs. To this end, we present a technique that enables partitioners based on recursive bipartitioning to reliably compute balanced solutions. The idea is to preassign a small portion of the heaviest nodes to one of the two blocks of each bipartition and optimize the objective function on the remaining nodes. We integrated the approach into the sequential hypergraph partitioner KaHyPar and show that our new approach can compute balanced solutions for all tested instances without negatively affecting the solution quality and running time of KaHyPar.

In our experimental evaluation, we compare our new shared-memory (hyper)graph partitioner Mt-KaHyPar to 25 different graph and hypergraph partitioners on over 800 (hyper)graphs with up to two billion edges/pins. The results indicate that already our fastest configuration outperforms almost all existing hypergraph partitioners with regards to both solution quality and running time. Our highest-quality configuration ($n$-level with flow-based refinement) achieves the same solution quality as the currently best sequential partitioner KaHyPar, while being almost an order of magnitude faster with ten threads. In addition, we optimize our data structures for graph partitioning, which improves the running times of both multilevel partitioners by almost a factor of two for graphs. As a result, Mt-KaHyPar also outperforms most of the existing graph partitioning algorithms. While the shared-memory graph partitioner KaMinPar is still faster than Mt-KaHyPar, its produced solutions are worse by 10% in the median. The best sequential graph partitioner KaFFPa-StrongS computes slightly better partitions than Mt-KaHyPar (median improvement is 1%), but is more than an order of magnitude slower on average.

# Acknowledgements

This project would not have been possible without the many people who have supported me on this journey. Thus, I would like to use this chapter to express my deepest thanks to them.

First of all, I would like to thank my advisor Peter Sanders for welcoming me back from industry with open arms and for the many discussions that greatly influenced the outcome of this work. I would also like to thank Ümit Çatalyürek for writing one of the reviews for this dissertation.

Moreover, I would like to acknowledge the invaluable assistance of my co-authors Yaroslav Akhremtsev, Ümit Çatalyürek, Karen Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Nikolai Maas, Henning Meyerhenke, Peter Sanders, Daniel Seemaier, Sebastian Schlag, Christian Schulz, and Dorothea Wagner.

The coronavirus has posed many new challenges to our workplaces in recent years. Under these circumstances, I am more than grateful that our research group has managed to keep up our friendly and inspiring working atmosphere. Therefore, I would like to express my deepest thanks to my former and actual co-workers Michael Axtmann, Timo Bingmann, Daniel Funke, Simon Gog, Demian Hespe, Lukas Hübner, Markus Iser, Florian Kurpicz, Sebastian Lamm, Moritz Laupichler, Hans-Peter Lehmann, Tobias Maier, Matthias Schimek, Sebastian Schlag, Lorenz Hübschle-Schneider, Dominik Schreiber, Daniel Seemaier, Marvin Williams, and Sascha Witt.

Furthermore, I would also like to thank Florian Kurpicz, Moritz Laupichler, Nikolai Maas, Tobias Maier, Sebastian Schlag, Dominik Schreiber, and Daniel Seemaier for proof-reading part of this dissertation and their valuable feedback.

Over my academic career, I had the privilege to work with many bright students. I am very grateful to Patrick Firnkes, Tobias Fuchs, Manuel Haag, Cedrico Knoesel, Robert Krause, Moritz Laupichler, Nikolai Maas, and Lukas Reister for the endless hours they put into their projects from which I have gained many new ideas and insights. Especially, I would like to thank Nikolai Maas, who contributed to my work as a thesis student and research assistant over many years.

I cannot imagine how my academic career would have gone if I had not joined the research project of Sebastian Schlag on hypergraph partitioning seven years ago. I have always appreciated our countless and often endless discussions, in which I have learned so much. Thus, I would like to thank him for being the best mentor one could imagine. Moreover, this work would not have been possible without the collaboration with Lars Gottesbüren. I have always enjoyed our technical and non-technical discussions and highly value the energy he put into this project over the past years. Last, I would like to thank my wife Alessa Heuer. She always supported me with her love and motivated me when things went not as expected.

# Table of Contents


**1 Introduction** **1**
- 1.1 Contributions . . . 4
- 1.2 Outline . . . 8

**2 Preliminaries** **9**
- 2.1 Definitions and Notations . . . 9
- 2.2 The Balanced Hypergraph Partitioning Problem . . . 12
- 2.3 Shared-Memory Programming . . . 16
  - 2.3.1 The Theoretical Machine Model . . . 16
  - 2.3.2 The Parallelization Library . . . 18
- 2.4 Experimental Design . . . 21
  - 2.4.1 Benchmark Sets . . . 21
  - 2.4.2 Methodology . . . 24
  - 2.4.3 Visualizing Solution Quality . . . 25
  - 2.4.4 Visualizing Running Times and Speedups . . . 27
  - 2.4.5 Statistical Significance Tests . . . 28

**3 Related Work** **29**
- 3.1 Iterative Improvement Algorithms . . . 30
  - 3.1.1 Kernighan-Lin Algorithm . . . 30
  - 3.1.2 Fiduccia-Mattheyses Algorithm . . . 32
- 3.2 Flow-Based Refinement . . . 35
- 3.3 The Multilevel Scheme . . . 38
  - 3.3.1 The Label Propagation Algorithm . . . 41
  - 3.3.2 $n$-Level Hypergraph Partitioning . . . 41
  - 3.3.3 Algorithmic Components of Sequential Partitioners . . . 43
- 3.4 Parallel (Hyper)Graph Partitioning . . . 48
  - 3.4.1 Parallelization Challenges . . . 49
  - 3.4.2 Algorithmic Components of Parallel Partitioners . . . 52

**4 Parallel Improvement Algorithms** **59**
- 4.1 Parallel Gain Calculation . . . 60
  - 4.1.1 The Partition Data Structure . . . 60
  - 4.1.2 The Gain Table . . . 63
  - 4.1.3 Parallel Gain Recalculation . . . 67

4.2 Label Propagation Refinement . . . 68
4.3 Direct $k$-Way FM Local Search . . . 70
4.3.1 Multi-Try $k$-Way FM Algorithm . . . 70
4.3.2 Highly-Localized $k$-Way FM Search . . . 71
4.4 Flow-Based Refinement . . . 74
4.4.1 Parallel Active Block Scheduling . . . 74
4.4.2 Network Construction . . . 77
4.4.3 The FlowCutter Algorithm . . . 79
4.4.4 Parallel Maximum Flow Algorithm . . . 80
4.4.5 Intricacies with Preflows and FlowCutter . . . 82
4.4.6 Implementation Details . . . 82

**5 Parallel Multilevel Hypergraph Partitioning 85**
5.1 The Hypergraph Data Structure . . . 87
5.2 Coarsening . . . 87
5.2.1 Rating Function Evaluation . . . 88
5.2.2 Clustering Algorithm . . . 89
5.2.3 Contraction Limit . . . 91
5.2.4 Community Detection Enhancement . . . 92
5.3 Initial Partitioning . . . 94
5.3.1 Parallel Recursive Bipartitioning . . . 94
5.3.2 Flat Initial Bipartitioning . . . 95
5.4 Refinement . . . 96
5.5 Engineering Aspects . . . 97
5.6 Algorithm Configuration . . . 98
5.7 Insights into Multilevel Partitioning . . . 104
5.7.1 Analysis of Search Conflicts . . . 104
5.7.2 Effectiveness Tests . . . 109
5.7.3 Scalability . . . 113
5.7.4 Running Times of Components . . . 117

**6 Parallel $n$-level Hypergraph Partitioning 121**
6.1 The Dynamic Hypergraph Data Structure . . . 123
6.1.1 Remove and Restore Incident Nets . . . 125
6.1.2 Contraction Operation . . . 125
6.1.3 Uncontraction Operation . . . 126
6.2 Parallel $n$-level Coarsening . . . 127
6.2.1 Contraction Forest . . . 128
6.2.2 Handling Contraction Dependencies . . . 128
6.2.3 Removing Identical Nets . . . 129
6.3 Parallel $n$-level Uncoarsening . . . 129
6.3.1 Sibling Uncontraction Dependencies . . . 130
6.3.2 Batch Construction Algorithm . . . 131
6.3.3 Refinement . . . 131

6.3.4 Gain Table Maintenance . . . 132
6.4 Insights into $n$-Level Partitioning . . . 133
6.4.1 Algorithm Configuration . . . 134
6.4.2 Scalability . . . 135
6.4.3 Running Times of Components . . . 138
6.4.4 Comparison to Multilevel Partitioning . . . 140

**7 From Hypergraphs to Graphs 145**
7.1 Partition Data Structure . . . 146
7.1.1 The Gain Table . . . 146
7.1.2 Attributed Gains . . . 147
7.2 Multilevel Graph Partitioning . . . 148
7.2.1 The Graph Data Structure . . . 148
7.2.2 Peculiarities for Graph Partitioning . . . 149
7.3 $n$-Level Graph Partitioning . . . 150
7.3.1 Contraction and Uncontraction Operation . . . 150
7.3.2 Remove and Restore Selfloops and Identical Edges . . . 153
7.4 Experiments . . . 154

**8 A Comparison of Partitioning Algorithms 157**
8.1 Included Partitioning Algorithms . . . 157
8.2 Identifying Competitors . . . 160
8.3 Comparison to Other Systems . . . 167
8.3.1 Hypergraph Partitioning . . . 167
8.3.2 Graph Partitioning . . . 169
8.4 Summary . . . 172

**9 Multilevel Hypergraph Partitioning with Node Weights 177**
9.1 Definitions and Notations . . . 179
9.2 Balanced Recursive Bipartitioning . . . 180
9.2.1 Deeply Balanced Bipartitions . . . 180
9.2.2 Sufficiently Balanced Bipartitions . . . 182
9.2.3 The Prepacking Algorithm . . . 185
9.3 Experiments . . . 186
9.3.1 The LPT Balance Constraint . . . 188
9.3.2 Balanced Partitioning . . . 188
9.3.3 Solution Quality and Running Times . . . 190
9.4 Proof of Claim 9.6 . . . 193

**10 Conclusion 195**

**List of Algorithms 205**

**List of Figures 207**

**List of Tables** **211**

**Bibliography** **213**

**List of Publications** **239**

Chapter 1

# 1 Introduction

A widely used technique to accelerate parallel graph computations is to partition the node set of a graph into a predefined number of blocks of roughly the same size, such that the number of edges running between different blocks is minimized. This problem is called the *balanced graph partitioning* problem. The most prominent applications can be found in distributed scientific simulations [SKK00; Fie+12] and graph analytics [Low+12; Wan+14]. Here, graph partitioning distributes the node set of a graph to *machines* of a compute cluster, which are interconnected through a communication network. Each processor then performs some work on the nodes assigned to it and needs to exchange computational results with other processors that share common edges of the graph. Thus, a *good* partition evenly assigns the computational work across the machines and minimizes the communication costs. Other applications can be found in route planning [SWZ02; Del+11; HS18] and image segmentation [CG12; PZZ13]. In the former, graph partitioning finds natural separators in road networks (e.g., bridges, rivers, or highways) which can be used to accelerate shortest-path algorithms. In the latter, the goal is to partition an image into several objects of interest.

A graph can only model pairwise relationships between a set of objects. However, in many real-world problems, more complex relationships exist, which may result in a loss of information when modeled as a graph.

> Consider a set of authors and a set of scientific publications where each publication is written by a subset of the authors, as illustrated in Figure 1.1 (left). We can model these relationships with a graph by adding an edge between two authors if they are co-author of the same publication. However, with this graph model, we cannot answer queries that ask, e.g., for all publications of an author with more than three co-authors wihtout an additional labeling on the edges (this example is slightly adapted from Ref. [ZHS06]). Furthermore, a publication adds a clique between all its authors, which unnecessarily increases the size of our graph model (e.g., the largest number of authors on a paper is 5154 [Aad+15]).

For such relationships, a hypergraph – which consists of nodes and *hyperedges* – models the underlying application more accurately since hyperedges can connect more than two nodes.

The *hypergraph partitioning* problem is a natural generalization of the graph partitioning problem. The two most prominent objective functions are the cut-net and connectivity metric. The former sums up the weight of all hyperedges connecting more

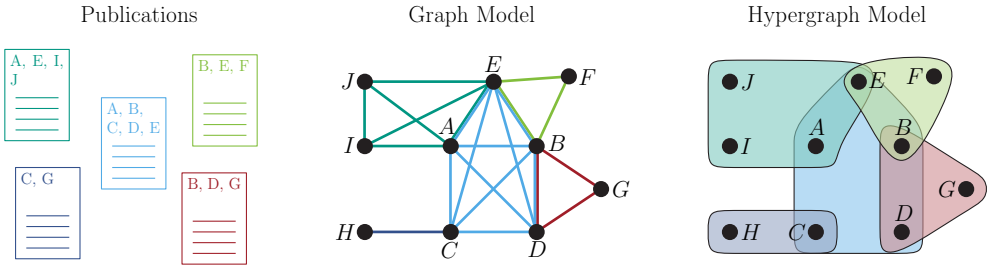



**Figure 1.1:** The graph and hypergraph model of the relationships between a set of authors and their publications. In both models, the authors are modeled as nodes. In the graph model, we add an edge between two authors if they are co-author of the same paper. In the hypergraph model, the authors of an publication form the hyperedges.

than one block, while the latter additionally considers the number of blocks connected by a hyperedge. Early work on hypergraph partitioning was mainly driven by the *very-large scale integration* (VLSI) design community [SK72; FM82; DK85; AK95; HB95; AHK97]. A logical circuit consists of gates connected by wires. A partition of a circuit into equally-sized blocks minimizing the number of wires between the different blocks divides it into more manageable components that are then mapped onto a chip in a top-down hierarchical design process. A *good* partition leads to a reduction of signal delays and wire lengths, and to a minimization of the layout area required by the circuit [AK95]. Since a wire can connect more than two gates, a hypergraph models the underlying problem more accurately than a graph.

VLSI design has long been one of the most prominent applications of hypergraph partitioning, but new applications have emerged with the rise of parallel scientific computing. For example, hypergraph partitioning is used to minimize the communication volume of the parallel sparse matrix-vector multiplication [CA99; ÇA01a; ÇA01b]. In the simplest model, a sparse matrix can be interpreted as a hypergraph where rows correspond to hyperedges and columns to nodes. Here, the connectivity metric more accurately models the communication volume than any graph-based model [CA99]. Other applications can be found in distributed quantum circuit simulations [AH19; GK21], storage sharding in distributed databases [Cur+10; Kum+14; YP15; Ser+16; Kab+17; Yan+18b], and variable reordering in satisfiability solvers [AMS04].

> The author of this dissertation also used hypergraph partitioning to assign guests to dining tables at his wedding. Here, the guests correspond to nodes, and the relationships between the guests form the hyperedges (e.g., families, couples, or friends). A good partition then corresponds to a feasible assignment of guests to the dining tables that minimizes the number of guests that know each other and are placed on different tables.

The hypergraph partitioning problem is NP-hard [GJS76; Len90], and it is unlikely that a constant factor approximation exists unless $P = NP$ [Fel13]. Thus, heuristic

solutions are used in practice where the multilevel method has emerged as the most successful approach to solve the problem [BS93; HL95]. The multilevel scheme consists of three phases. First, the hypergraph is *coarsened* to obtain a hierarchy of successively smaller and structurally similar approximations of the input hypergraph by *contracting* pairs or clusters of nodes. Once the coarsest hypergraph is small enough, an *initial partition* into $k$ blocks is computed. Subsequently, the contractions are reverted level-by-level, and, on each level, *local search* heuristics are used to improve the partition from the previous level (*refinement phase*).

The applications of (hyper)graph partitioning have different requirements on the solution quality and running time of partitioning algorithms. VLSI design is an example where even small improvements in partitioning quality significantly impact the performance of a chip and reduces its production costs [HB95]. If (hyper)graph partitioning is used as preprocessing technique in parallel computations, speed is more important than quality since the partitioning time should not dominate the overall execution time. Existing sequential partitioning algorithms can be categorized either into systems that aim for speed or high solution quality, while the latter are often more than an order of magnitude slower than the former [Sch20, p. 197].

With ever growing problem sizes, this tradeoff diminishes as the running time of the highest-quality sequential systems becomes prohibitive. Thus, it becomes increasingly important to parallelize the techniques used in these algorithms. Early work on parallel (hyper)graph partitioning has focussed on distributed-memory algorithms [KK96; WC00b]. The distributed-memory model is effective for trivially parallel problems but is "not well suited to fine-grained parallelism" [Lum+07] since computational results must be exchanged via expensive network communication. Therefore, these algorithms usually do not achieve the same solution quality as their sequential counterparts because they use comparatively weak components that are easier to parallelize.

As the capability of a single microprocessor converges to its physical limits, current technology continues to improve the performance of CPUs by increasing the number of its cores. The main benefit of such systems is that the different processors share access to the same global memory, which can be used for a fast exchange of computational results. In the last twenty years, the number of cores per socket increased from a single core to over one hundred [Bin18, p. 2]. At the same time, the costs of memory technologies have dropped dramatically[1]. Therefore, we believe that shared-memory machines have become a viable alternative for processing large problem instances.

Research has recently focused on shared-memory partitioning algorithms, which achieve good speedups and produce superior solutions compared to distributed-memory algorithms [LK16; ASS17]. However, the partitioning quality of these systems is still not competitive with the best sequential algorithms, as we will see throughout this work.

[1] `https://hblok.net/blog/storage/` (accessed May 2022)

## 1.1 Contributions

This dissertation closes the quality gap between sequential and parallel partitioning algorithms by presenting the *first* shared-memory multilevel hypergraph partitioner with parallel implementations of many techniques used by the highest-quality sequential algorithms. Our new system achieves the same solution quality as the best sequential codes for optimizing the connectivity metric, while being competitive to the fastest partitioning algorithms in terms of running time. As a result, it outperforms almost all publicly available (hyper)graph partitioners and enables the partitioning of extremely large (hyper)graphs with high solution quality.

This work is based on four conference publications [Got+21a; HMS21a; GHS22a; Got+22a] and two technical reports [Got+21c; GHS22c]. Furthermore, we describe unpublished optimizations that we implemented for graph partitioning. All contributions are presented in a consistent level of detail and with extensive experimental evaluations that go beyond the scope of the individual publications. We further compared 25 different partitioning algorithms on over 800 graphs and hypergraphs to our new shared-memory algorithm, which is, to the best of our knowledge, the most comprehensive comparison that can be found in the partitioning literature. In the following, we briefly outline the core contributions of this work.

> Most of the work presented in this dissertation was done in collaboration with Lars Gottesbüren. The conference publications [Got+21a; GHS22a; Got+22a] and technical reports [Got+21c; GHS22c] were written jointly by both authors. Peter Sanders and Sebastian Schlag provided valuable feedback and assisted in the editing process of the corresponding publications. Since Lars Gottesbüren and the author of this work use large parts of these publications in their dissertations, we include footnotes in the following and a short paragraph in each chapter that attributes the presented techniques and ideas to each author to highlight their individual contributions.

**Parallel Gain Computation Techniques[2].** Many local search algorithms greedily move nodes to other blocks according to a *gain* function. The gain value of a node move reflects the change in the objective function when performed on the partition. In the parallel setting, two concurrent node moves may degrade the solution quality even if both individual gain values suggest an improvement (due to race conditions during the gain computation). We present a technique that can detect move conflicts between concurrent node moves at the time they are performed on the partition based on synchronized data structure updates. We also use the technique to analyze how frequently such conflicts occur in local search algorithms, which provides valuable insights on the impact of such conflicts in practice. We further present a concurrent gain table data structure for storing and maintaining all gain values with a better asymptotic worst-case complexity than the one used in KaHyPar [Akh+17a].

[2] Lars Gottesbüren developed the gain table data structure, while the author of this dissertation implemented the partition data structure and the technique for detecting concurrent move conflicts.

**The First Fully Parallel Direct $k$-Way FM Algorithm[3].** The *Fiduccia-Mattheyses* (FM) algorithm [FM82] is the most widely used local search algorithm in multilevel partitioning algorithms. It performs the node move with the highest gain value in each step. Thereby, it also performs moves that intermediately worsen the solution quality and is therefore able to escape from local optima. A parallel version of the algorithm is already implemented in the shared-memory graph partitioner Mt-KaHIP [ASS17]. The algorithm repeatedly starts highly-localized FM searches in parallel. Each search is initialized with a few seed nodes and gradually expands around them by claiming neighbors of moved nodes. The implementation has two major shortcomings: (i) node moves are performed locally and are *not visible to other threads*, and (ii) the node moves performed by the different threads are concatenated to a global move sequence at the end of a refinement pass, for which *gains are recomputed sequentially*. Our implementation immediately applies a move sequence to the global partition once an improvement is found and other threads can see these changes. Moreover, we parallelize the sequential gain recomputation step. We further integrate our parallel gain computation techniques to accelerate the algorithm and to adapt gain values to changes made by other threads.

**A Novel Parallelization of Flow-Based Refinement[4].** We present a novel parallelization of the *flow-based refinement* algorithm used in the sequential hypergraph partitioner KaHyPar [SS11; HSS19a; Got+20]. The sequential algorithm operates on bipartitions (partition into two blocks), so the implementation schedules the bipartitioning routine on pairs of blocks to improve $k$-way partitions. Our parallelization of the algorithm uses two sources of parallelism: a parallel scheduling scheme and a parallel maximum flow algorithm. In addition, we propose several optimizations that substantially accelerate the algorithm in practice, enabling the use on extremely large hypergraphs.

**A Scalable Multilevel Hypergraph Partitioning Algorithm[5].** We present the first *shared-memory* multilevel hypergraph partitioning algorithm with parallel implementations of techniques used by the highest-quality sequential codes. We implement a parallel coarsening algorithm that finds a clustering of the nodes on each level, and subsequently contracts it in parallel to obtain the next coarser hypergraph. The parallel clustering algorithm detects and resolves conflicting cluster assignments *on-the-fly*, while previous algorithms use a postprocessing step or expensive locking mechanisms. To make better contraction decisions, we run a parallel community detection algorithm before coarsening and use the community structure to restrict contractions to nodes within the same community in the coarsening phase [HS17a].

[3] Lars Gottesbüren implemented the initial version of the parallel FM algorithm and the gain recomputation algorithm. The author of this dissertation has contributed several performance optimizations for the highly-localized FM searches.

[4] The author of this dissertation implemented the parallel scheduling scheme, while Lars Gottesbüren parallelized the flow computations.

[5] The parallel coarsening, initial partitioning, and label propagation algorithm was implemented by the author of this dissertation. Lars Gottesbüren developed the parallel community detection algorithm and was involved in the performance engineering process in many parts of the code.

Once the hypergraph is small enough, we compute an initial partition using parallel recursive bipartitioning. We further integrate a work-stealing approach to account for load imbalances within the recursive partitioning calls. In the uncoarsening phase, we use label propagation refinement, and our novel FM and flow-based refinement algorithms to improve the partition on each level.

We note that the core algorithmic ideas of the different components are already used in sequential and partly in parallel systems. However, the overall partitioning algorithm itself is one of the main contributions of this work as it already outperforms most of the existing partitioning algorithms even without flow-based refinement. Our fastest configuration can bipartition our largest hypergraph with more than two billion pins (= sum of the sizes of all hyperedges) in one minute with 64 threads, while cutting half as many hyperedges than the bipartitions produced by Zoltan (64 threads, three minutes) and PaToH-D (sequential, ten minutes). The self-relative speedup of our algorithm is 24.7 for larger problem instances with 64 threads on average.

**A Scalable *n*-Level Hypergraph Partitioning Algorithm[6].** The $n$-level partitioning scheme instantiates the multilevel paradigm in its most extreme version by contracting only a single node on each level. Correspondingly, in each refinement step, only a single node is uncontracted, allowing a highly-localized search for improvements. The $n$-level scheme is currently implemented in the highest-quality sequential partitioner KaHyPar [Sch+16a; Akh+17a; Sch20]. Although (un)contracting a single node on each level seems inherently sequential, we present a sophisticated parallelization of the approach that achieves the same solution quality as KaHyPar, while being an *order of magnitude faster* with ten threads. To this end, we implement a dynamic hypergraph data structure to perform concurrent (un)contractions. This is achieved by developing a representation of contractions by a forest which is used to determine a parallel execution order of the (un)contractions and to resolve conflicts *on-the-fly*. In the refinement phase, the contraction forest is decomposed into *batches*, each containing a fixed number of nodes. The batches are then uncontracted in parallel and used as a starting point for highly-localized local searches.

**A Simple Optimization for Graph Partitioning[7].** Graph (GP) and hypergraph partitioning (HGP) share many similarities, while HGP is considered "inherently more complicated" [Kay+12]. We found that the main differences between HGP and GP are in the representation of the (hyper)graph data structure and how gain values are computed. Hence, we implement optimized data structures for graph partitioning and use them as a drop-in replacement in our partitioning algorithms. As a result, we were able to accelerate our multilevel and $n$-level algorithm by a factor of 1.75 and 1.91 for graphs.

[6] The idea and implementation of the parallel $n$-level partitioning algorithm came from the author of this dissertation, while Lars Gottesbüren was involved in the performance engineering process.

[7] The idea of adapting the partition and hypergraph data structure for graph partitioning came from the author of this dissertation. The implementation was done by Nikolai Maas, who worked as a student research assistant in our group at the time.

**The Multi-Threaded Karlsruhe Hypergraph Partitioning Framework.** We make our new shared-memory (hyper)graph partitioner Mt-KaHyPar (**M**ulti-**T**hreaded **Ka**rlsruhe **Hy**pergraph **Par**titioning) publicly available under `https://github.com/kahypar/mt-kahypar`. The code contains our multilevel (Mt-KaHyPar-D) and $n$-level partitioning algorithm (Mt-KaHyPar-Q), as well as configurations extending them with flow-based refinement (Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F).

**An Extensive Comparison of Existing Partitioning Algorithms.** We propose multiple configurations of our algorithm, offering different tradeoffs in solution quality and running time. We have invested considerable effort in integrating as many publicly available partitioning algorithms as possible into our experimental evaluation to show to what extent Mt-KaHyPar is competitive. In total, we incorporated 25 different partitioning systems and evaluated them extensively on over 800 graphs and hypergraphs. The total running time of the experiment would have been 10.43 years when executed on a single machine. To the best of our knowledge, it is the most comprehensive study of partitioning algorithms that can be found in the literature.

For hypergraph partitioning, the results show that our fastest configuration Mt-KaHyPar-D already outperforms almost all partitioning algorithms in terms of solution quality and running time. On larger hypergraphs with up to one billion pins, it produces partitions better than those of the previously best parallel hypergraph partitioner Zoltan [Dev+06] (distributed-memory) by 23% in the median, while it is also faster by a factor of 2.72 on average. Our highest-quality configuration Mt-KaHyPar-Q-F achieves the same solution quality as the currently best sequential partitioner KaHyPar [Sch20] and is an order of magnitude faster with ten threads. For graph partitioning, we show that Mt-KaHyPar also outperforms almost all existing graph partitioning algorithms. However, the shared-memory partitioner KaMinPar [Got+21e] is still faster than Mt-KaHyPar-D, but its solutions are worse by 10% in the median. The best sequential graph partitioner KaFFPa-StrongS [SS11] computes slightly better partitions than Mt-KaHyPar-Q-F with ten threads (median improvement is 1%), but it is more than an order of magnitude slower on average.

There is a well-defined set of techniques for (hyper)graph partitioning providing different tradeoffs in solution quality and running time. Mt-KaHyPar completely evades this tradeoff for medium-sized instances as our highest-quality configuration achieves the same solution quality as the best sequential partitioner KaHyPar while being slightly faster than PaToH-Q [CA99] which is among the fastest sequential partitioning algorithms. This tradeoff is now shifted to larger problem instances for which we provide a new set of Pareto-optimal partitioning methods.

**Finding Balanced Partitions is Suprisingly Hard in Practice[8].** We show that existing sequential partitioning algorithms considerably struggle to find balanced partitions for weighted real-world hypergraphs under a tight balance constraint. To

[8]The idea of revisiting the balanced hypergraph partitioning problem for weighted instances came from Sebastian Schlag [Sch20, p. 218]. The theoretical foundations and implementation were done by Nikolai Maas as part of his bachelor thesis [Maa20a], which was supervised by Sebastian Schlag and the author of this dissertation.

this end, we present a technique enabling partitioners based on recursive bipartitioning to reliably compute balanced solutions. We integrated the balancing technique into the sequential hypergraph partitioner KaHyPar [Sch20]. In the experimental evaluation, we show that our new approach computes balanced partitions on all tested instances without negatively affecting the solution quality and running time of KaHyPar.

## 1.2 Outline

Chapter 2 introduces basic notations and definitions, and describes our benchmark sets and experimental methodology. In Chapter 3, we review existing literature closely related to the techniques used in this dissertation. We particularly focus on existing sequential and parallel partitioning algorithms to identify techniques essential for achieving high solution quality. Chapter 4 then presents our parallel gain computation techniques and three parallel refinement algorithms: a label propagation algorithm, a highly-localized direct $k$-way FM algorithm, and a novel parallelization of flow-based refinement. In Chapter 5, we discuss the algorithmic components of our parallel multilevel hypergraph partitioner and subsequently turn to our parallel $n$-level partitioning algorithm in Chapter 6. Chapter 7 then explains our optimizations for graph partitioning. In Chapter 8, we compare Mt-KaHyPar to 25 different partitioning algorithms and show that it offers an excellent tradeoff between solution quality and running time. In Chapter 9, we turn to sequential partitioning, for which we implemented a technique that enables partitioners based on recursive bipartitioning to reliably compute balanced partitions for weighted hypergraphs. Chapter 10 concludes the dissertation and presents directions for future research.

Chapter 2

# 2 Preliminaries

This chapter presents fundamental defintions and concepts, and introduces the experimental design used in this dissertation. In Section 2.1, we introduce basic notations and definitions. Section 2.2 then defines the balanced hypergraph partitioning problem, and discusses its computational complexity and several problem variations. In Section 2.3, we cover parallel programming concepts and present the parallelization library used in this work. We conclude this chapter with a description of our experimental design in Section 2.4, including the composition of our benchmark sets, methodology, and how we visualize running times, solution quality and speedups of different partitioning algorithms.

**References.** We closely worked with the authors of the sequential hypergraph partitioner KaHyPar [Sch+16a; Akh+17a; HS17a; Got+20] prior to our work on shared-memory hypergraph partitioning. Thus, we reuse large parts of their notations and definitions in our publications [Sch20, p. 11–14] and use a similar experimental design as described in the dissertation of Schlag [Sch20, p. 22–34]. We also contributed to a recent survey on (hyper)graph partitioning [Çat+22a] from which we copied some text passages that were written exclusively by the author of this dissertation.

## 2.1 Definitions and Notations

**Hypergraphs.** A *weighted hypergraph* $H = (V, E, c, \omega)$ is defined as a set of $n$ nodes $V$ and a set of $m$ hyperedges $E$ (also called *nets*) with node 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 node set $V$. The nodes of a net are called its *pins*. We extend $c$ and $\omega$ to sets in a natural way, i.e., $c(U) := \sum_{u \in U} c(u)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A node $u$ is *incident* to a net $e$ if $u \in e$. $\mathrm{I}(u) := \{e \mid u \in e\}$ are the set of all incident nets of $u$. The set $\Gamma(u) := \{v \mid \exists e \in E : \{u, v\} \subseteq e\}$ denotes the neighbors of $u$. Two nodes $u$ and $v$ are *adjacent* if $v \in \Gamma(u)$. The *degree* of a node $u$ is $d(u) := |\mathrm{I}(u)|$. The *size* $|e|$ of a net $e$ is the number of its pins. We denote the number of pins of a hypergraph with $p := \sum_{e \in E} |e| = \sum_{v \in V} d(v)$. The maximum node degree and net size is defined as $\Delta_v := \max_{v \in V} d(v)$ and $\Delta_e := \max_{e \in E} |e|$. We call two nets $e_i$ and $e_j$ *identical* if $e_i = e_j$. Given a subset $V' \subset V$, the *subhypergraph* $H_{V'}$ is defined as $H_{V'} := (V', \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\}, c, \omega')$ where $\omega'(e \cap V')$ is the weight of hyperedge $e$ in the original hypergraph. Figure 2.1 shows a hypergraph and illustrates the notations described in this paragraph.

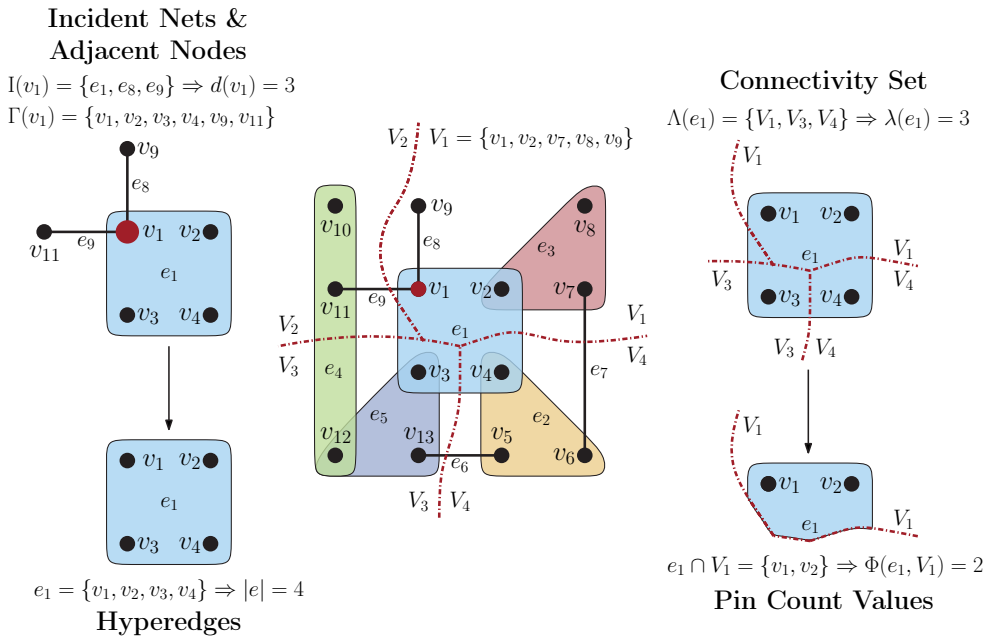



**Figure 2.1:** Illustration of a 4-way partition of a hypergraph with 13 nodes and 9 hyperedges (middle). Moreover, it shows the incident nets and adjacent nodes for node $v_1$ (left), and the connectivity set of hyperedge $e_1$ and its pin count value for block $V_1$ (right).

**Graphs.** A *directed* and *weighted graph* $G = (V, E, c, \omega)$ is defined as a set of $n$ nodes $V$ and a set of $m$ edges $E$ with node weights $c : V \to \mathbb{R}_{>0}$ and edge weights $\omega : E \to \mathbb{R}_{>0}$. We represent a directed edge $(u, v) \in E$ as ordered pair. We call an edge $(u, u)$ a *selfloop* and $(u, v)$ with $u \neq v$ a *regular* edge or simply an edge if we do not have to distingush between selfloops and regular edges. The set $\Gamma(u) := \{v \mid (u, v) \in E\}$ denotes the neighbors of $u$. A directed edge $(u, v) \in E$ implies that $u$ is adjacent to $v$, but $v$ is not adjacent to $u$ unless $(v, u) \in E$. An edge $\{u, v\}$ is *undirected* if $(u, v) \in E$ and $(v, u) \in E$. We use the set notation to represent an undirected edge $\{u, v\}$. A graph that only contains undirected edges is called an *undirected graph*. Each undirected edge $\{u, v\}$ can be interpreted as a hyperedge of size two. Thus, the definitions and notations for hypergraphs also apply to undirected graphs. We denote the weight of an undirected edge $e = \{u, v\}$ by $\omega(u, v) =: \omega(e)$. If $\{u, v\} \notin E$, then $\omega(u, v) = 0$. For a subset $V' \subseteq V$, $\omega(u, V') := \sum_{v \in V'} \omega(u, v)$ is the weight of all edges connecting node $u$ to subset $V'$.

The *bipartite graph representation* $G_x := (V \cup E, E_x)$ [SK72; HM85] of an unweighted hypergraph $H = (V, E)$ contains an undirected edge $\{u, e\} \in E_x$ if node $u$ is a pin of net $e$. More formally, $E_x := \{\{u, e\} \mid \exists e \in E : u \in e\}$. Figure 2.2 (middle) shows the bipartite graph representation of a hypergraph $H$.

**Partitions and Clusterings.** A *k-way partition* of a hypergraph $H = (V, E, c, \omega)$ is a partition of the node set $V$ into $k$ disjoint *blocks* $\Pi = \{V_1, \ldots, V_k\}$. A 2-way partition is also called a *bipartition*. We denote the block to which a node $u$ is assigned by $\Pi[u]$. 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 $\lambda(e) := |\Lambda(e)|$. A net is called a *cut net* if $\lambda(e) > 1$. A node $u$ that is incident to at least one cut net is called *boundary node*. The number of pins of a net $e$ in block $V_i$ is denoted by $\Phi(e, V_i) := |e \cap V_i|$. We refer to $\Phi(e, V_i)$ as the *pin count value* for a net $e$ and block $V_i$. Figure 2.1 (right) illustrates the connectivity set and pin count value of a hyperedge. The set $E(V_i, V_j) := \{e \in E \mid \{V_i, V_j\} \subseteq \Lambda(e)\}$ represents the cut nets connecting block $V_i$ and $V_j$. Two blocks $V_i$ and $V_j$ are *adjacent* if $E(V_i, V_j) \neq \emptyset$. The *quotient graph* $\mathcal{Q} := (\Pi, E_\Pi := \{(V_i, V_j) \mid E(V_i, V_j) \neq \emptyset\})$ contains an edge between all adjacent blocks.

We call a partition $\Pi$ *$\varepsilon$-balanced* if each block $V_i$ satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1 + \varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some *imbalance ratio* $\varepsilon \in (0, 1)$. A partition that fullfils $\forall V_i \in \Pi : V_i \leq \lceil \frac{c(V)}{k} \rceil$ is *perfectly balanced*. We also refer to an $\varepsilon$-balanced $k$-way partition as a *balanced* or *feasible* solution.[1]

A *clustering* $\mathcal{C} = \{C_1, \ldots, C_l\}$ of a hypergraph $H$ is a partition of the node set $V$ into disjoint blocks where the number of blocks is not given in advance. A cluster $C_i$ is called a *singleton* cluster if $|C_i| = 1$. A node contained in a singleton cluster is called *unclustered*.

**Contractions and Uncontractions.** Contracting a clustering $\mathcal{C} = \{C_1, \ldots, C_l\}$ of a hypergraph $H = (V, E, c, \omega)$ replaces each cluster $C_i$ with one supernode $u_i$ with weight $c(u_i) = \sum_{v \in C_i} c(v)$. For each net $e \in E$, we replace each pin $v \in e$ with the node $u_i$ representing the cluster $C_i$ in which $v$ is contained. After the replacement, multiple occurrences of the same supernode in a net are discarded. The contracted hypergraph may contain single-pin nets and nets identical to each other. For the hypergraph partitioning problem defined in the next section, we can discard single-pin nets and remove identical nets except for one representative at which we aggregate their weight as a performance optimization.

Contracting a node $v$ onto another node $u$ replaces $v$ with $u$ in all nets $e \in \mathrm{I}(v) \setminus \mathrm{I}(u)$ and removes $v$ from all nets $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$. The new weight of $u$ is then $c(u) + c(v)$. We call $u$ the *representative* of the contraction and $v$ its *contraction partner*. Uncontracting a node $v$ reverses the corresponding contraction operation.

**Flows.** A flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, \mathrm{c})$ is a directed graph with a dedicated source node $s \in \mathcal{V}$ and sink node $t \in \mathcal{V}$ in which each edge $e \in \mathcal{E}$ has capacity $\mathrm{c}(e) \geq 0$. An $(s, t)$-flow is a function $f : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ that satisfies the *capacity constraint* $\forall u, v \in \mathcal{V} : f(u, v) \leq \mathrm{c}(u, v)$, the *skew symmetry constraint* $\forall u, v \in \mathcal{V} : f(u, v) = -f(v, u)$ and the *flow conservation constraint* $\forall u \in \mathcal{V} \setminus \{s, t\} : \sum_{v \in \mathcal{V}} f(u, v) = 0$. The value

[1] We note that there is an ongoing discussion in our research group about the correct definition of the maximum allowed block weight $L_{\max}$. In the presence of node weights, there are situations where no feasible solution exists. We discuss several alternative definitions in Section 9.3.1 in more detail. However, except for the experimental evaluation in Section 9.3, we focus on unweighted (hyper)graphs, and thus use $L_{\max}$ as balance constraint.

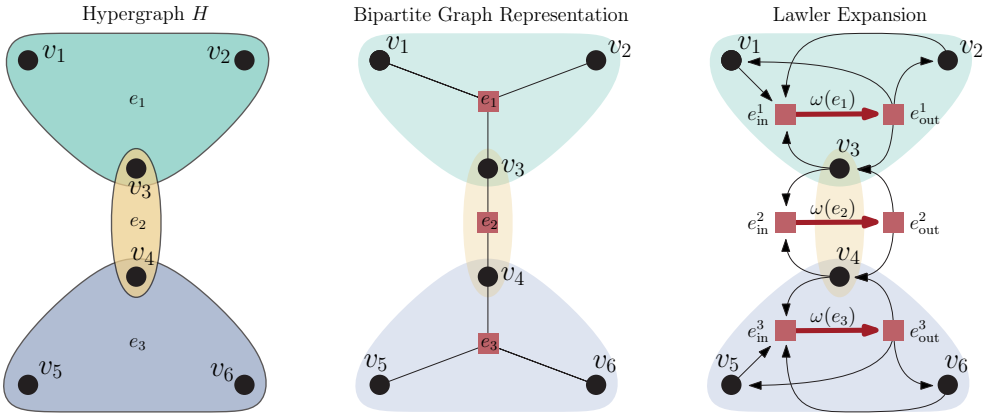



**Figure 2.2:** Illustration of the bipartite graph representation (middle) and Lawler expansion (right) of a hypergraph $H$ (left). Figure is taken from Ref. [Heu18a].

of a flow $|f| := \sum_{v \in \mathcal{V}} f(s,v) = \sum_{v \in \mathcal{V}} f(v,t)$ is defined as the total amout of flow transferred from $s$ to $t$. An $(s,t)$-flow $f$ is a maximum $(s,t)$-flow if there exists no other $(s,t)$-flow $f'$ with $|f| < |f'|$. The *residual capacity* is defined as $r_f(e) = \mathrm{c}(e) - f(e)$. An edge $e$ is *saturated* if $r_f(e) = 0$. The *residual network* $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f, r_f)$ with $\mathcal{E}_f := \{(u,v) \in \mathcal{V} \times \mathcal{V} \mid r_f(u,v) > 0\}$ contains all non-saturated edges. The max-flow min-cut theorem states that the value $|f|$ of a maximum $(s,t)$-flow equals the weight of a minimum cut that separates $s$ and $t$ [FF56]. This is also called a *minimum $(s,t)$-cut*. The minimum $(s,t)$-cut can be derived by exploring the nodes reachable from the source or sink via residual edges, which is also called the *source-side* or *sink-side cut*.

**Flows on Hypergraphs.** The *Lawler expansion* $\mathcal{N} = (\mathcal{V}, \mathcal{E}, \mathrm{c})$ [Law73] of a hypergraph $H = (V, E, c, \omega)$ is defined as follows: $\mathcal{V}$ contains all nodes $v \in V$. For each hyperedge $e \in E$, we add two nodes $e_{\text{in}}$ and $e_{\text{out}}$ to $\mathcal{V}$ and a *bridging edge* $(e_{\text{in}}, e_{\text{out}})$ with capacity $\mathrm{c}(e_{\text{in}}, e_{\text{out}}) = \omega(e)$ to $\mathcal{E}$. For each pin $v \in e$, we add two edges $(v, e_{\text{in}})$ and $(e_{\text{out}}, v)$ with infinite capacity to $\mathcal{E}$. The Lawler expansion is illustrated in Figure 2.2 (right). A minimum $(s,t)$-cut in the Lawler expansion directly corresponds to one in the hypergraph (since only bridging edges have finite capacity).

## 2.2 The Balanced Hypergraph Partitioning Problem

**Problem Definition.** Given parameters $\varepsilon$ and $k$, and a (hyper)graph $H$, the *balanced (hyper)graph partitioning problem* is to find an $\varepsilon$-balanced $k$-way partition $\Pi$ that minimizes an objective function defined on the (hyper)edges. For $k = 2$, we refer to the problem as the *bipartitioning* problem. The two most prominent objective functions are the *cut-net* metric $\mathfrak{f}_c(\Pi) := \sum_{e \in E'} \omega(e)$ (also called *edge cut* metric for graph partitioning) and *connectivity* metric $\mathfrak{f}_{\lambda-1}(\Pi) := \sum_{e \in E'} (\lambda(e) - 1) \cdot \omega(e)$ (also called

$(\lambda - 1)$-metric) where $E'$ denotes the set of all cut nets. The cut-net metric directly generalizes the edge cut metric from graphs to hypergraphs and minimizes the weight of all cut hyperedges. The connectivity metric additionally considers the number of blocks connected by a net and thus more accurately models the communication volume for parallel computations [CA99] (e.g., for the parallel sparse matrix-vector multiplication). The hypergraph partitioning problem is NP-hard for both objective functions [Len90]. The connectivity value of a net is bounded by its size and the number of blocks. Hence, the connectivity metric reverts to the cut-net metric for graph partitioning ($|e| = 2$ for all $e \in E$) and bipartitioning ($\lambda(e) \leq 2$ for all $e \in E$).

**Recursive Bipartitioning vs Direct *k*-Way Partitioning.** A $k$-way partition of a (hyper)graph can be obtained either by *recursive bipartitioning* or *direct k-way partitioning*. The former computes a bipartition of the input (hyper)graph and then calls the bipartitioning routine on both blocks recursively until the (hyper)graph is divided into the desired number of blocks. The latter partitions the (hyper)graph directly into $k$ blocks and applies $k$-way refinement algorithms to improve the partition.

**Concept of Gains.** Moving a node $u$ from its current block $\Pi[u]$ to a target block $V_j$ directly affects the underlying objective function. The gain value of a node move indicates how much the objective function would *change* when performed on the partition. It is central for many moved-based local search algorithms and is universally applicable to almost all objective functions. These algorithms move nodes greedily to other blocks according to a gain value. We denote the gain of moving a node $u$ to a block $V_j$ by $g_u(V_j)$. The actual definition of $g_u(V_j)$ depends on the objective function. For the connectivity metric, the gain of a node move is defined as follows:

$$g_{\lambda-1,u}(V_j) := \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi[u]) = 1\}) - \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, V_j) = 0\}).$$

Moving node $u$ to block $V_j$ decreases the connectivity of all nets by one for which $u$ is the last remaining pin in block $\Pi[u]$ (i.e., $\Phi(e, \Pi[u]) = 1$). Conversely, it increases the connectivity of all nets $e \in \mathrm{I}(u)$ by one for which the target block $V_j$ is not part of the connectivity set $\Lambda(e)$ (i.e., $\Phi(e, V_j) = 0$). If $g_{\lambda-1,u}(V_j)$ is positive, then moving $u$ to block $V_j$ improves $\mathfrak{f}_{\lambda-1}(\Pi)$ by $g_{\lambda-1,u}(V_j)$.

For the cut-net metric, we can define the gain of a node move as follows:

$$g_{c,u}(V_j) := \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, V_j) = |e| - 1\}) - \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi[u]) = |e|\}).$$

The first term is the weight of all cut nets incident to $u$ which we can remove from the cut by moving $u$ to block $V_j$. The second term is the weight of all non-cut nets incident to $u$, which become cut if $u$ is moved to block $V_j$. For graph partitioning, we can simplify the gain definition as follows (since $|e| = 2$ for all $e \in E$):

$$\begin{aligned} g_{c,u}(V_j) &= \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, V_j) = 1\}) - \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi[u]) = 2\}) \\ &= \omega(u, V_j) - \omega(u, \Pi[u]). \end{aligned}$$

The gain can be expressed as the weight of all edges connecting $u$ to the target block $V_j$ minus the weight of all edges connecting $u$ to its current block $\Pi[u]$.

Local search algorithms use gain values to identify node moves with positive gain [MSS14] or perform the move with the highest gain value in each step [KL70; FM82]. Krishnamurthy [Kri84] introduced the higher-level gain, which defines $l$ gain values for each node. The first level gain corresponds to the traditional gain definition, while subsequent levels capture the impact of the node move for future moves and can be used for tie-breaking decisions in local search algorithms.

**Other Objective Functions.** An objective function closely related to the connectivity metric is the *sum of external degree* metric (SOED) $\mathfrak{f}_s(\Pi) := \sum_{e \in E'} \lambda(e) \cdot \omega(e) = \mathfrak{f}_{\lambda-1}(\Pi) + \mathfrak{f}_c(\Pi)$ [KK00]. An interesting property of this metric is that removing a net $e$ from the cut reduces $\mathfrak{f}_s(\Pi)$ by $2\omega(e)$ while removing a single block from the connectivity set of a net $e$ with $\lambda(e) > 2$ only reduces it by $\omega(e)$. Thus, the objective function prefers removing nets from the cut and, as secondary criteria, optimizes their connectivity.

For bipartitioning, another popular objective function is the *ratio cut* $\mathfrak{f}_{rc}(\Pi) := \frac{\mathfrak{f}_c(\Pi)}{c(V_1) \cdot c(V_2)}$ [WC89; WC91]. It additionaly considers the block weights in the denominator and thus implicitly optimizes the balance of the bipartition. The *scaled cost* metric $\mathfrak{f}_{sc}(\Pi) := \frac{1}{n(k-1)} \sum_{i=1}^{k} \frac{E_i}{c(V_i)}$ with $E_i := \omega(\{e \in E' \mid e \cap V_i \neq \emptyset\})$ generalizes the ratio cut to $k$-way partitioning [CSZ93]. There are several other objective functions for which we refer the reader to surveys [AK95; Bul+16].

**Problem Variations.** Instead of partitioning the node set, the *(hyper)edge partitioning* problem asks for a partition of the (hyper)edge set $E$ of a (hyper)graph $H = (V, E, c, \omega)$ into $k$ disjoint and ($\varepsilon$-)balanced blocks $E_1, \ldots, E_k$ such that an objective function defined on the nodes is minimized. An widely used objective function for the (hyper)edge partitioning problem is the *vertex cut* $\sum_{u \in V} (\gamma(u) - 1) \cdot c(u)$, where $\gamma(u)$ denotes the number of blocks containing incident (hyper)edges of $u$ [Yan+18a]. The problem is NP-hard [BLV14] and can be used to minimize the number of node replicas for edge-centric computations on (hyper)graphs in the distributed-memory setting [Gon+12]. For the edge partitioning problem on graphs, existing approaches reduce it to traditional node-based graph partitioning [Li+17], which only implicitly optimizes the vertex cut. Alternatively, the input graph can be transformed into its dual hypergraph representation [Sch+19] where the edges represent the nodes and each node of the graph induces a hyperedge spanning its incident edges. Optimizing the connectivity metric of the corresponding hypergraph partitioning problem directly optimizes the vertex cut of the underlying edge partitioning problem.

Instead of optimizing a single objective function, there are also formulations optimizing multiple objective functions simultanously. Existing techniques either assign priorities to the different metrics [Çat+12b; Dev+15] or combine them to a single objective function [Aba+02; SK03]. Another problem variation is the multi-constaint partitioning formulation [KK98d; ACU08b; Slo+20]. Here, each node $u$ is associated with a weight vector $c(u) = (c_u^{(1)} \ldots c_u^{(l)})$ with $l$ entries, while there is also one individual balance constraint $L_{\max}^{(i)}$ for each entry. A feasible $k$-way partition $\Pi = \{V_1, \ldots, V_k\}$ satisfies $c^{(j)}(V_i) := \sum_{u \in V_i} c_u^{(j)} \leq L_{\max}^{(j)}$ for each block $V_i \in \Pi$ and entry $j \in \{1, \ldots, l\}$.

The (hyper)graph partitioning problem with *fixed nodes* introduces an additional constraint for the block assignment of some nodes. A fixed node is preassigned to one block prior to partitioning and is not allowed to change its block [ACU08b]. A formal definition of the problem follows in Chapter 9, where we use fixed nodes to reliably compute balanced $k$-way partitions for weighted hypergraphs under a tight balance constraint.

**Hardness Results and Approximations.** This paragraph discusses the computational complexity of the balanced (hyper)graph partitioning and the *minimum $k$-cut* problem. The latter asks for a set of (hyper)edges with minimum weight whose removal partitions the nodes into $k$ connected components (no balance constraint is enforced). We will refer to the minimum 2-cut problem as the minimum cut problem. The following results consider weighted (hyper)graphs and the cut-net metric as the underlying objective function. The main theoretical findings are that "the more balanced the partition we look for has to be, the harder the problem" [WW93] and hypergraphs are much harder to partition than graphs [RSS18; CL20].

The max-flow min-cut theorem relates the minimum $(s,t)$-cut in a weighted graph to the maximum $(s,t)$-flow in the corresponding flow network [FF56]. Thus, the minimum cut problem is solvable in polynomial time via $n-1$ maximum flow computations. A similar result also holds for hypergraphs since finding a minimum $(s,t)$-cut in a hypergraph can be reduced to finding a maximum $(s,t)$-flow in a flow network with $n+2m$ nodes [Law73] (see Lawler expansion in Section 2.1). The fastest known algorithms solve the minimum cut problem in $\mathcal{O}(nm\log(\frac{n^2}{m}))$ time for directed graphs [HO92] and $\mathcal{O}(nm+n^2\log n)$ time for undirected graphs [NI92; SW97]. The algorithm proposed by Stoer and Wagner [SW97] finds a minimum $(s,t)$-cut (where $s$ and $t$ are not given in advance) with a greedy graph traversal algorithm. The source $s$ and sink $t$ are then merged and the procedure is recursively applied to the contracted graph until only one node remains. The lightest minimum $(s,t)$-cut found within the recursive calls corresponds to a minimum cut. Klimmek and Wagner [KW96] generalized the algorithm to hypergraphs, resulting in an algorithm with a running time of $\mathcal{O}(n^2\log n+np)$.

The minimum $k$-cut problem is NP-hard when $k$ is part of the input [GH94]. For fixed $k$, the fastest known algorithm solves the problem in $\tilde{\mathcal{O}}(nm^{2k-3})$ time for graphs [Tho08] and $\tilde{\mathcal{O}}(nm^{2k-2})$ time for hypergraphs [FPZ19].[2] While the optimal solution for the minimum $k$-cut problem can be approximated within a factor of $2-\frac{2}{k}$ in polynomial time for arbitrary $k$ on graphs [SV95], there exists no constant factor approximation for hypergraphs unless $P=NP$ [CL20].

We now turn to the balanced (hyper)graph partitioning problem. The problem is NP-hard for $k=2$ and $\varepsilon=0$ [GJS76] (also known as the *(hyper)graph bisection problem*). However, an $\mathcal{O}(\log n)$-approximation for the graph bisection problem exists [Räc08], whereas the best approximation is within a factor of $\mathcal{O}(n\log^{5/4} n)$ for hypergraphs [RSS18]. Räcke et al. [RSS18] showed that no algorithm can achieve a

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

better approximation ratio than $\mathcal{O}(n^{1/4-\varepsilon})$ for the hypergraph bisection problem unless $P = NP$. Thus, the hypergraph bisection problem is much harder to approximate than the graph bisection problem. For $k > 2$, the existence of a polynomial time approximation algorithm for the graph partitioning problem is unlikely for $\varepsilon = 0$ [AR04], while there exists an $\mathcal{O}(\log n)$-approximation for $\varepsilon > 0$ [FF15]. In general, it is unlikely that a polynomial time algorithm exists approximating the optimal solution of the balanced (hyper)graph partitioning problem within a constant factor for arbitrary $k$ and $\varepsilon$ [Fel13].

## 2.3 Shared-Memory Programming

This section presents fundamental parallel programming concepts frequently used throughout this work. This includes the theoretical parallel machine model for analyzing parallel algorithms and the parallelization library `Intel TBB` used for implementing our shared-memory (hyper)graph partitioner.

### 2.3.1 The Theoretical Machine Model

**Random Access Machine (RAM).** Shepherdson and Sturgis [SS63] proposed an abstraction from the computer architecture introduced by John von Neumann [Neu45], which we still use today as an underlying model for analyzing the complexity of sequential computer programs. The *random access machine* (RAM) consists of a single processing unit and a main memory with an infinite number of cells $M[0], M[1], M[2], \ldots$. For this work, we assume that each memory cell $M[i]$ stores integer values which can be represented by 64 bits (also called a *machine word*). Additionally, there are also a limited number of *registers* $R[0], \ldots, R[k]$. The RAM model is illustrated in Figure 2.3 (left). The input for a computer program is stored in the main memory, and the processing unit executes predefined *instructions* step-by-step on values stored in the registers. The results of computations can be transferred between the registers and main memory via dedicated `load` and `store` instructions. In addition, there are also instructions for arithmetic (addition, subtraction, multiplication, division, ...) and logical operations (logical not, or, and, ...), and (un)conditional branching. The model assumes that each instruction takes one time unit. Hence, the total running time of a computer program is the total number of performed instructions.

However, the assumption that each instruction takes one time unit often does not hold in practice since the RAM model *oversimplifies* modern hardware architectures. For example, existing processors contain several arithmetic processing units which can process independent instructions in parallel (*instruction-level parallelism*). Moreover, today's machines also use a *cache hierarchy* to accelerate access to frequently used memory cells. Loading a value from the main memory can be more than orders of magnitude slower than accessing data from caches [Bin18, see Figure 3.11 on p. 73]. However, incorporating every single implementation detail of modern processors into the cost model "would end up with a complex model [*that is*] difficult to handle

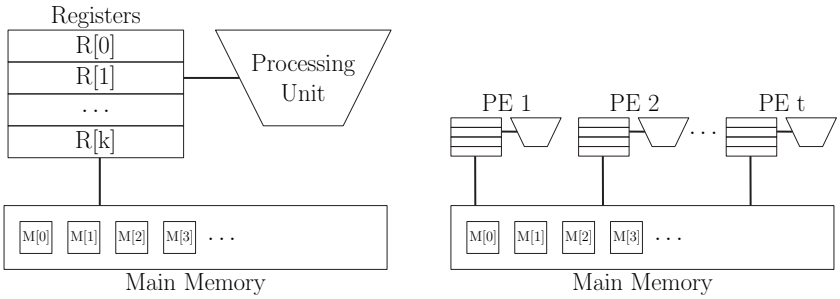



**Figure 2.3:** Visualization of the RAM and PRAM model. The RAM model consists of a processing unit that can load values from the main memory into registers and then perform a predefined set of instructions on them. In the PRAM model, several processing units have access to the same main memory, which requires synchronization mechanisms to handle concurrent read and write accesses.

[*and*] change with every new processor generation" [San+19]. Thus, the RAM model provides a simple tool to analyze algorithms in theory. However, algorithm engineers have to consider the architecture of modern processors in their algorithm design process as this is equally important in practice.

**Parallel Random Access Machine (PRAM).** The *parallel random access machine* (PRAM) consists of $t$ processing units/elements (PE) sharing a common main memory over which they can exchange data (see Figure 2.3 (right)).[3] The cost model of RAMs also apply to PRAMs. However, special care must be taken to handle concurrent reads (R) or writes (W) to the same memory cell. The access can be either *exclusive* (E) or *concurrent* (C). For concurrent write accesses, several models exist to resolve collisions. The *common* strategy only overwrites the content of a memory cell if each PE writes the same value, while the *abitrary* strategy chooses one value at random. Moreover, collisions can also be resolved by assigning *priorities* to PEs or using a *combine* operator aggregating values written to the same memory cell. The memory access model is often given as part of the name of the PRAM model to describe it sufficiently. For example, common CRCW-PRAM stands for the concurrent-read concurrent-write PRAM model with the common strategy.

The PRAM model assumes that the instructions of the different PEs are performed in globally synchronized time steps. This is not realistic in practice. A more practical PRAM model assumes that each PE processes its instructions *asynchronously* [Gib89] and concurrent write accesses are resolved by inserting the new value into a queue which is then written to the memory cell in *first-in first-out* fashion [GMR94] (aCRQW-

[3] The literature denotes the number of processing elements by $p$, which we already use for the number of pins of a hypergraph.

PRAM: asynchronous concurrent-read queue-write PRAM model). Thus, the write operation takes time proportional to the number of PEs that concurrently write to the same memory cell.

The outcome of concurrent write operations is often non-deterministic. Therefore, modern architectures provide *transactional* operations in their instruction sets. A transaction is a set of instructions executed step-by-step, while no other PE is allowed to write to memory cells accessed by the transaction. We also refer to a transaction as an *atomic* operation. In this work, we make use of the atomic `compare-and-swap`$(M[i], a, b)$ and `fetch-and-add`$(M[i], a)$ operation. The former writes value $b$ to memory cell $M[i]$ if $M[i]$ equals $a$ and subsequently returns true. Otherwise, it returns false. The latter returns the value stored in memory cell $M[i]$ and then adds the value $a$ to it. The atomic `test-and-set`$(M[i])$ operation is equivalent to `compare-and-swap`$(M[i], 0, 1)$. Although many architectures provide hardware support for a large number of atomic operations, it is often necessary to execute specific parts of a program as one transaction. This can be achieved via *synchronization* mechanisms. In this work, we use *spin-locks* which we implement by setting a bit via an atomic `test-and-set` instruction. The PE succeeding in setting the bit to one is allowed to execute the protected part of the program, while other PEs spin in a busy-waiting loop in which they further try to set the bit. However, this serializes the program and may cause contention on concurrently accessed memory cells, which can drastically reduce the performance of parallel algorithms.

The PRAM model is a simplified abstraction from parallel machines and allows us to analyze parallel algorithms. However, the theoretical results often do not match what we observe in practice since the model largely ignores the complex cache hierarchies and protocols to keep them synchronized (*cache coherence*). A limiting factor for the scalability of parallel algorithms is that the hardware enforces a consistent view on the shared memory among the private caches of processors. Consequently, overwriting a cache entry can invalidate cache lines stored in private caches of other PEs. Hence, achieving perfect speedups is difficult in practice.

### 2.3.2 The Parallelization Library

We implement all parallel algorithms using the *Intel Thread Building Block* library [Phe08] (`TBB`). `TBB` "serves two key roles: (1) it fills foundational voids in support for parallelism where the `C++` standard has not sufficiently evolved [...] and (2) it provides higher-level abstractions for parallelism [...]" [VAR19].

The library exploits task-based parallelism with *work-stealing*. Figure 2.4 illustrates its internal *task scheduler*. An *application thread* is responsible for spawning (sub)tasks of parallel computations in a *task arena*. A task arena contains several slots, each with a thread-local deque storing incoming tasks. The application thread and idle threads from the *global thread pool* then join a slot of a task arena and process the tasks. Once the local deque of a slot becomes empty, a thread can steal work from the tail of other deques. If no tasks are left, the threads return to the global thread pool, and the application thread continues with the execution of the program.

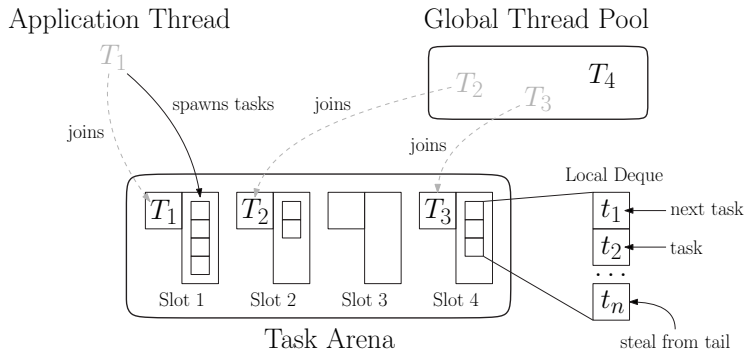



**Figure 2.4:** Visualization of the TBB task scheduler. The application thread spawns tasks in a task arena. A task arena consists of slots each storing tasks in a local deque. The threads can join a slot of a task arena and process its tasks. Once the local deque of a slot becomes empty, a thread can steal tasks from other slots in the task arena.

It is not deniable that the task scheduler introduces additional overheads. Thus, TBB might not be the best choice for algorithms with simple loop-based parallelization patterns. However, it provides many features supportive for implementing complex parallel algorithms. In this work, we use the concurrent vector and queue implementation as well as the thread-local storage feature and the scalable allocators. Moreover, we use low-level interfaces to implement parallel recursion patterns and the task arena observer pattern to pin threads to CPU cores.

In addition, we use the parallel sort, reduce, and prefix sum algorithms provided by TBB and our own parallel random shuffle implementation. In the following, we present the scaling behavior of these operations with an increasing number of threads. In the experiments, we generate an array $A$ of size $N$ containing 32-bit integer values and run the operations with $t \in \{1, 4, 16, 64\}$ threads on $A$. For each operation on a given array $A$, we execute ten repetitions and take the arithmetic mean as the total execution time. We run the experiments on arrays with up to $N = 10^9$ entries as this corresponds to the number of pins/edges of our largest (hyper)graph. The experiments are performed on machine $B$ (described in Section 2.4.2). Figure 2.5 summarizes the self-relative speedups $T_{\text{par}}(A, 1)/T_{\text{par}}(A, t)$ of the different operations, where $T_{\text{par}}(A, t)$ is the execution time of an operation with $t$ threads on a given array $A$.

**Parallel Sorting.** For these experiments, we sorted a random permutation of the numbers from 1 to $N$. The sort operation exhibits good speedups for $t = 4$ and becomes worthwhile on arrays with more than $10^4$ entries. For a larger number of threads ($t \geq 16$), the operation achieves mediocre speedups with 11.12 for $t = 64$ and $N = 10^9$. We note that there are better parallel sorting algorithms available [Axt+22]. However, we only use the operation to output basic properties of a (hyper)graph (e.g., median node degree and net size), which we do not need in experiments.

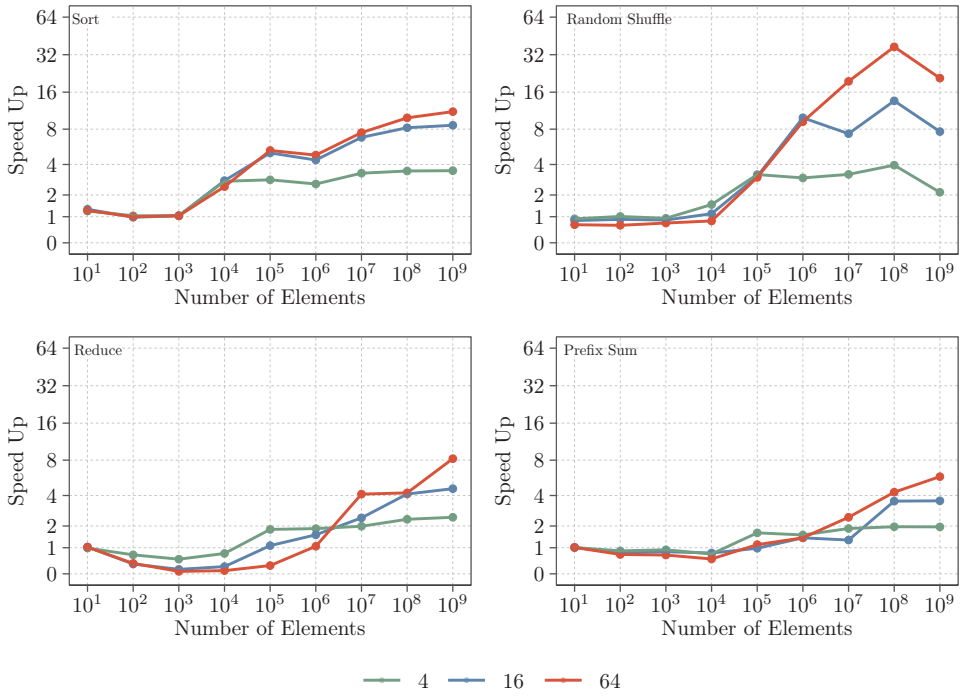



**Figure 2.5:** Speedups of different parallel algorithms used in this work.

**Parallel Random Shuffling.** We use a block-shuffling scheme that does not guarantee uniform randomness but suffices for our purposes. We divide the array into $2t$ equally-sized blocks. Each thread then swaps two random blocks and shuffles them sequentially. In the scalability experiments, we use the operation to obtain a random permutation of the numbers from 1 to $N$. The operation shows good speedups also for a larger number of threads. For $t = 64$ and $N = 10^8$, the speedup is 36.99. However, the speedups slightly decrease for $N = 10^9$. For $t = 64$ ($t = 16$) and $N = 10^9$, the total execution time of the parallel random shuffle operation is 0.79s (2.16s), which we consider negligible for instances with $10^9$ nodes.

**Parallel Reduce and Prefix Sum.** The parallel reduce operation applies an aggregation operator on the values of a given array. In the experiments, we compute the total sum of all entries of array $A$. The parallel prefix sum operation computes for each entry $j$ the prefix sum $\sum_{i=1}^{j} A[i]$. For both experiments, each entry of array $A$ contains either 0 or 1 chosen uniformly at random. As it can be seen in Figure 2.5 (bottom), both operations do not exhibit good speedups. For smaller array sizes, we also observe some slowdowns. However, both operations are extremely fast. The total execution time of the parallel reduce and prefix sum operation is 0.03s and 0.09s

for $t = 64$ and $N = 10^9$. Hence, their running times are negligible for most of the algorithms in which we use them.

## 2.4 Experimental Design

In this work, we extensively evaluate different configurations of our algorithm and present a large comparison of 25 partitioning algorithms. This section explains the experimental setup for these experiments. We start with a description of the different benchmark sets. We divide them into sets containing medium-sized and large (hyper)graph instances. We use the former to evaluate sequential partitioning algorithms, while the latter is used for comparisons to parallel systems. We then define the experimental setup, including the specification of the used machines and the parameters used as input for partitioning the (hyper)graphs of the different benchmarks sets (e.g., number of blocks $k$, imbalance factor $\varepsilon$, number of repetitions per instance). We further explain how we aggregate and visualize different performance indicators to compare partitioning algorithms (solution quality, running times, and speedups). We conclude this section with a note on statistical significance testing.

### 2.4.1 Benchmark Sets

In this dissertation, we evaluate sequential and parallel graph and hypergraph partitioners. Parallel algorithms are used when the running time of sequential codes becomes prohibitive. Thus, we assembled several benchmark sets divided into medium-sized (M) and large (L) instances. We abbreviate the name of a benchmark set, for example, with $\mathrm{L}_{\mathrm{HG}}$. The baseline denotes the size of the instances in the benchmark set, while the subscript indicates whether it contains hypergraphs (HG) or graphs (G). Furthermore, we evaluate different parameter choices of our new partitioner for which we assembled two parameter tuning benchmark sets $\mathrm{M}_{\mathrm{P}}$ and $\mathrm{L}_{\mathrm{P}}$, which are subsets of set $\mathrm{M}_{\mathrm{HG}}$ and $\mathrm{L}_{\mathrm{HG}}$ (i.e., $\mathrm{M}_{\mathrm{P}} \subset \mathrm{M}_{\mathrm{HG}}$ and $\mathrm{L}_{\mathrm{P}} \subset \mathrm{L}_{\mathrm{HG}}$). In the following, we describe the sources and composition of the different benchmark sets in more detail.[4] The properties of the different instances are summarized in Figure 2.6.

**Hypergraph Instances.** Our instances are derived from four sources encompassing three application domains: the ISPD98 VLSI Circuit Benchmark Suite [Alp98] (ISPD98), the DAC 2012 Routability-Driven Placement Contest [Vis+12] (DAC2012), the SuiteSparse Matrix Collection [DH11] (SPM), and the international SAT Competition 2014 [Bel+14] (SAT14).

We translate sparse matrices to hypergraphs using the row-net model [CA99]. Here, rows correspond to nets and columns to nodes. A non-zero entry in a cell $(i, j)$ means net $i$ contains node $j$ as a pin. We transform each satisfiability instance into three hypergraph representations: LITERAL, PRIMAL, and DUAL. In the LITERAL model,

[4] We made all benchmark sets and detailed statistics of their properties publicly available from `https://algo2.iti.kit.edu/heuer/dissertation/`.

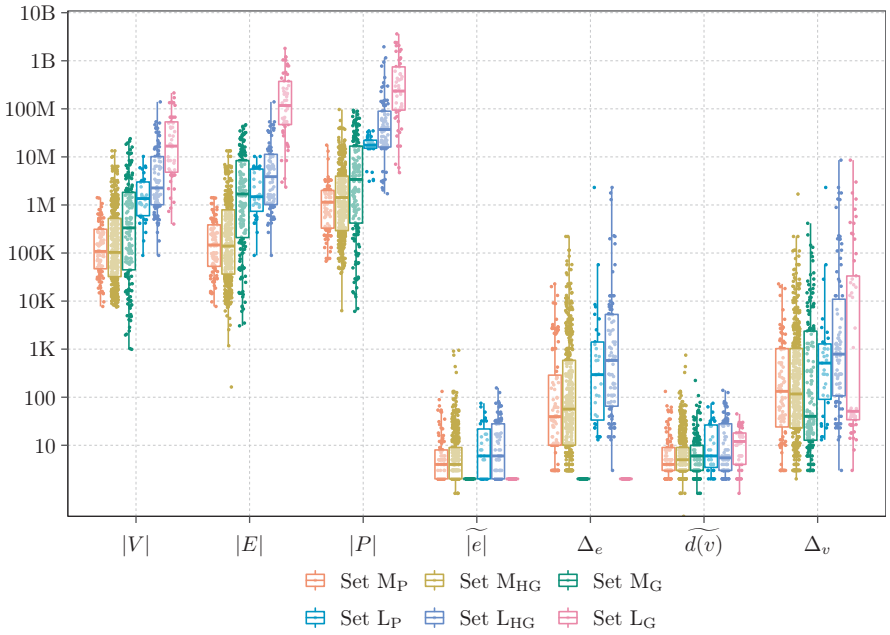



**Figure 2.6:** Summary of different properties for our benchmark sets. It shows for each (hyper)graph (points), the number of nodes $|V|$, nets $|E|$ and pins $|P|$ (for graphs the number of pins are $2|E|$), as well as the median and maximum net size ($\widetilde{|e|}$ and $\Delta_e$) and node degree ($\widetilde{d(v)}$ and $\Delta_v$).

each literal represents a node (each variable $a$ induces two nodes: $a$ and $\neg a$), and each clause corresponds to a hyperedge, while Primal instances represent variables as nodes [PM07]. In the Dual model, clauses correspond to nodes, while the variables form the hyperedges spanning the clauses in which they are contained [MP14].

For comparison with sequential partitioners, we use the well-established benchmark set of Heuer and Schlag [HS17a], which contains 488 hypergraphs (set $\mathrm{M_{HG}}$). In a later publication [ASS18], they proposed a subset $\mathrm{M_P}$ of set $\mathrm{M_{HG}}$ with 100 hypergraphs, which we use for parameter tuning experiments.

To evaluate parallel hypergraph partitioners, we assembled a new benchmark set composed of 94 large hypergraphs (set $\mathrm{L_{HG}}$). The benchmark set contains the 24 largest Sat14 instances from set $\mathrm{M_{HG}}$, which we enhanced with 18 even larger Sat14 instances from Ref. [Bel+14]. We also included 42 sparse matrices with at least 15 million non-zeros, randomly sampled from the SuiteSparse Matrix Collection [DH11]. Furthermore, we added all Dac2012 instances from set $\mathrm{M_{HG}}$. For parameter tuning experiments, we use a subset $\mathrm{L_P}$ of set $\mathrm{L_{HG}}$ composed of the 15 smallest Spm instances and the 5 smallest Dac2012, Literal, Primal and Dual instances, respectively.

Table 2.1 shows the number of instances included from the different sources and the largest hypergraph of each benchmark set. The largest instance of set $\mathrm{L_{HG}}$ (`sk-2005`)

**Table 2.1:** The number of hypergraphs included from the different sources and the largest instance of each benchmark set.

| Type | Set $M_{HG}$ | Set $L_{HG}$ | Set $M_P$ | Set $L_P$ | Source |
|---|---|---|---|---|---|
| SPM | 184 | 42 | 32 | 15 | [DH11] |
| ISPD98 | 18 | - | 10 | - | [Alp98] |
| DAC2012 | 10 | 10 | 4 | 5 | [Vis+12] |
| SAT14 - PRIMAL | 92 | 14 | 18 | 5 | [Bel+14] |
| SAT14 - LITERAL | 92 | 14 | 18 | 5 | [Bel+14] |
| SAT14 - DUAL | 92 | 14 | 18 | 5 | [Bel+14] |
| Total | 488 | 94 | 100 | 35 | |
| Largest Instance (Pins) | $\approx$ 97M | $\approx$ 2B | $\approx$ 17M | $\approx$ 34M | |

**Table 2.2:** The number of graphs included from the different sources and the largest instance of each benchmark set.

| Type | Set $M_G$ | Set $L_G$ | Source |
|---|---|---|---|
| DIMACS | 114 | 16 | [Bad+13] |
| SOCIAL NETWORKS | 30 | 16 | [LK14; Lab] |
| RANDOM GRAPHS | 15 | 15 | [KGB15; Fun+18] |
| SPM | 3 | 6 | [Wil+07; DH11] |
| DAC2012 | 10 | - | [Vis+12] |
| Total | 172 | 53 | |
| Largest Instance (Edges) | $\approx$ 47M | $\approx$ 1.8B | |

has roughly 2 billion pins. As can be seen in Figure 2.6, the median number of pins of the hypergraphs in set $M_{HG}$ is $\approx$ 1.4 million, while the instances of set $L_{HG}$ are more than an order of magnitude larger on average (median number of pins is $\approx$ 37 million).

**Graph Instances.** Our graph benchmark sets are composed of instances from the 10th DIMACS Implementation Challenge [Bad+13] (DIMACS), the Stanford Large Network Dataset Collection [LK14] and the Laboratory for Web Algorithms [Lab] (SOCIAL NETWORKS), the DAC 2012 Routability-Driven Placement Contest [Vis+12] (DAC2012), and the SuiteSparse Matrix Collection [Wil+07; DH11] (SPM). Moreover, we included several randomly generated graphs [KGB15; Fun+18] (RANDOM GRAPHS).

For comparison with sequential partitioners, we use the benchmark set of Gottesbüren et al. [Got+21e] (set $M_G$), which was initially assembled to evaluate the shared-memory graph partitioner KaMinPar. The benchmark set contains 195 graphs from which we excluded the 39 largest instances (considered too large for sequential algorithms). We additionally included 16 new graphs from the Stanford Large Network Dataset Collection [LK14] since we found that social networks were underrepresented

after excluding the largest graphs. In total, the benchmark set contains 172 graphs.

For comparison with parallel partitioners, we use the benchmark set of Ahkremtsev [Akh19] (set $L_G$), which was initially assembled to evaluate the shared-memory graph partitioner Mt-KaHIP. The benchmark set contains 42 large graphs from which we excluded 4 instances since they were used to evaluate external memory algorithms and are too large to fit into the main memory of a single machine. However, we enhanced the benchmark set with 15 out of the 39 excluded graphs from set $M_G$ (the other 24 insances were already contained in the benchmark set). In total, the benchmark set contains 53 large graphs.

Table 2.2 shows the number of instances included from the different sources. The largest instance (`sk-2005`) of set $L_G$ has roughly 1.8 billion edges. In Figure 2.6, we see that the medium-sized and large benchmark sets for graph and hypergraph partitioning have similar properties when comparing the number of edges and pins.

### 2.4.2 Methodology

**Systems.** We use a cluster of Intel Xeon Gold 6230 processors (2 sockets with 20 cores each) running at 2.1 GHz with 96GB RAM (machine A) for comparison with sequential partitioners on our medium-sized benchmark sets. Experiments running on our large benchmark sets are done on an AMD EPYC Rome 7702P (1 socket with 64 cores) running at 2.0–3.35 GHz with 1024GB RAM (machine B).

**Setup.** We implemented all algorithms presented in this work in the shared-memory hypergraph partitioner Mt-KaHyPar[5], which is implemented in `C++17`, parallelized using the `TBB` parallelization library [Phe08], and compiled using `g++9.2` with the flags `-O3 -mtune=native -march=native`.

Unless otherwise mentioned, we run experiments on our different benchmark sets using the partitioning setup shown in Table 2.3. We restrict the paramater space for experiments on our large benchmark set and machine B due to limited computational resources, which ensures that they run in a reasonable time frame. We run parallel partitioners with 10 threads when we evaluate them on our medium-sized benchmark sets and machine A. For experiments on machine $B$, we use all 64 available cores. For parallel partitioners we add a suffix to their name to indicate the number of threads used, e.g. Mt-KaHyPar 64 for 64 threads. We omit the suffix for sequential partitioners.

**Aggregating Performance Numbers.** The input for the balanced hypergraph partitioning problem consists of a hypergraph $H$, the number of blocks $k$, and an imbalance factor $\varepsilon$. We call a tuple $(H, k, \varepsilon)$ an *instance*. Each hypergraph partitioner optimizes the connectivity metric, and each graph partitioner optimizes the edge cut metric. We also refer to the connectivity or edge cut of a $k$-way partition as its solution quality.

For each instance, we aggregate running times and the solution quality using the arithmetic mean over all seeds. We use the geometric mean for absolute running times

[5] Mt-KaHyPar is publicly available from `https://github.com/kahypar/mt-kahypar`.

**Table 2.3:** Default partitioning setup for experiments conducted on our medium-sized (left) and large benchmark sets (right)

| | Set $\mathrm{M_{HG}}$/ $\mathrm{M_P}$/ $\mathrm{M_G}$ | Set $\mathrm{L_G}$/ $\mathrm{L_P}$/ $\mathrm{L_G}$ |
|---|---|---|
| Number of Blocks | $k \in \{2, 4, 8, 16, 32, 64, 128\}$ | $k \in \{2, 8, 16, 64\}$ |
| Imbalance | $\varepsilon = 3\%$ | $\varepsilon = 3\%$ |
| Repetitions per Instance/Seeds | 10 | 3 |
| Time Limit | 8 hours | 2 hours |
| Machine | A | B |
| Number of Used Cores (only for parallel partitioners) | 10 | 64 |

and self-relative speedups to further aggregate over multiple instances. If all runs of an algorithm produced imbalanced partitions or ran into the time limit on an instance, we consider the solution as infeasible. In plots, we mark imbalanced solutions with ✗ and similarly instances that timed out with ⏲. Runs with imbalanced partitions are not excluded from aggregated running times. For runs that exceeded the time limit, we use the time limit itself in the aggregates.

When we compare two algorithms $\mathcal{A}$ and $\mathcal{B}$ with geometric mean running times $\overline{t_\mathcal{A}}$ and $\overline{t_\mathcal{B}}$, we use the *relative slowdown* $x = \frac{\overline{t_\mathcal{B}}}{\overline{t_\mathcal{A}}}$ as a measure for running time improvements. If $x > 1$, we say that $\mathcal{A}$ is faster than $\mathcal{B}$ by a factor of $x$ on average. For comparing the solution quality of two algorithms $\mathcal{A}$ and $\mathcal{B}$, we use the median improvement of $\mathcal{A}$ over $\mathcal{B}$. Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be the set of test instances and $q_X(I)$ the solution quality of an algorithm $X$ produced for an instance $I \in \mathcal{I}$. Then, we compute the sequence $s = \langle \frac{q_\mathcal{B}(I_1)}{q_\mathcal{A}(I_1)}, \ldots, \frac{q_\mathcal{B}(I_n)}{q_\mathcal{A}(I_n)} \rangle$ and use the median improvement $x = (\mathrm{median}(s) - 1) \cdot 100$ (in percentage) as measure for improvements in solution quality. If $x$ is positive, we say that $\mathcal{A}$ produces better partitions than $\mathcal{B}$ by $x\%$ in the median. Note that the running times of different partitioning algorithms can vary by more than an order of magnitude, while improvements in solution quality typically differ only by a few percentages. Thus, we use the median improvement for comparing solution quality since it is less sensitive to outliers.

We use *self-relative* speedups $T_\mathrm{par}(I, 1)/T_\mathrm{par}(I, t)$ for analyzing the scalability of a parallel algorithm [San+19, p. 62]. Here, $T_\mathrm{par}(I, t)$ is the execution time of the algorithm with $t$ threads for a given input instance $I$.

### 2.4.3 Visualizing Solution Quality

**Performance Profiles.** *Performance profiles* can be used to compare the solution quality of different algorithms [DM02]. Let $\mathcal{X}$ be the set of all algorithms, $\mathcal{I}$ the set of instances, and $q_\mathcal{A}(I)$ the quality of algorithm $\mathcal{A} \in \mathcal{X}$ on instance $I \in \mathcal{I}$ ($q_\mathcal{A}(I)$ is the arithmetic mean over all seeds). For each algorithm $\mathcal{A}$, performance profiles show the fraction of instances ($y$-axis) for which $q_\mathcal{A}(I) \leq \tau \cdot \mathrm{Best}(I)$, where $\tau$ is on the $x$-axis

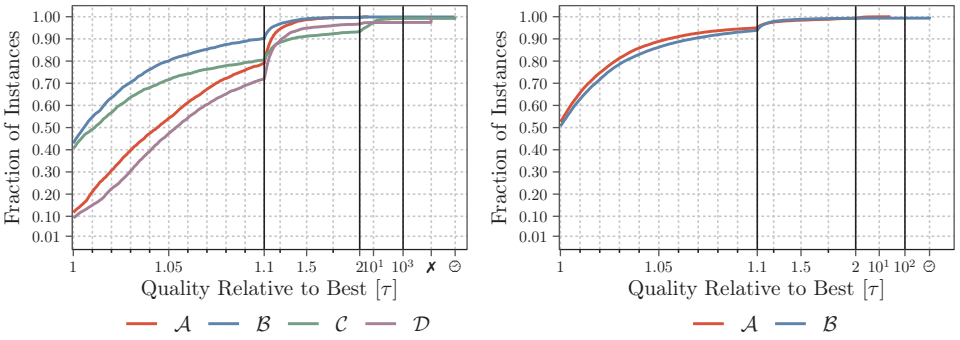



**Figure 2.7:** Performance profiles comparing four different algorithms (left) and the result of the effectiveness test between algorithm $\mathcal{A}$ and $\mathcal{B}$ (right).

and $\text{Best}(I) := \min_{\mathcal{A}' \in \mathcal{X}} q_{\mathcal{A}'}(I)$ is the best solution produced by an algorithm $\mathcal{A}' \in \mathcal{X}$ for an instance $I \in \mathcal{I}$. For $\tau = 1$, the $y$-value indicates the percentage of instances for which an algorithm $\mathcal{A} \in \mathcal{X}$ performs best. Achieving higher fractions at smaller $\tau$ values is considered better. The ✗- and ⏲-tick indicates the fraction of instances for which *all* runs of that algorithm produced an imbalanced solution or timed out. Note that these plots relate the quality of an algorithm to the best solution and thus do not permit a full ranking of three or more algorithms.

Figure 2.7 (left) compares the solution quality of four different algorithms using a performance profile. Algorithm $\mathcal{A}$ and $\mathcal{C}$ compute on roughly 40% of the instances the best solutions while algorithm $\mathcal{B}$ and $\mathcal{D}$ only on 10% (see $\tau = 1$). The solutions produced by algorithm $\mathcal{A}$ are worse than the best by $\approx 4\%$ in the median (intersection of $y = 0.5$ with the red line is at $\tau \approx 1.04$). If we compare algorithm $\mathcal{C}$ and $\mathcal{D}$ based on their geometric mean solution quality, we would observe that the solutions produced by algorithm $\mathcal{C}$ are better than those of $\mathcal{D}$ by 0.2% on average. Hence, we would probably conclude that there is no significant difference between both. If we look at the performance profile, we see that algorithm $\mathcal{C}$ is on most of the instances closer to the best solution than $\mathcal{D}$. However, it produces on $\approx 10\%$ of the instances solutions that are worse than the best by more than a factor of two, which has large influence on its geometric mean (see green line for $\tau \geq 2$).

**Effectiveness Tests.** The performance profile in Figure 2.7 (left) suggests that algorithm $\mathcal{B}$ produces solutions better than those of $\mathcal{A}$. Let us assume that algorithm $\mathcal{A}$ is more than an order of magnitude faster than $\mathcal{B}$ on average. Thus, algorithm $\mathcal{B}$ may have an unfair advantage due to its longer running time. Therefore, Ahkremtsev et al. [ASS17] introduces *effectiveness tests* to compare solution quality when two algorithms are given a similar running time by performing additional repetitions with the faster algorithm. To do so, we generate virtual instances that we compare using performance profiles. Consider two algorithms $\mathcal{A}$ and $\mathcal{B}$, and an instance $I$. We first

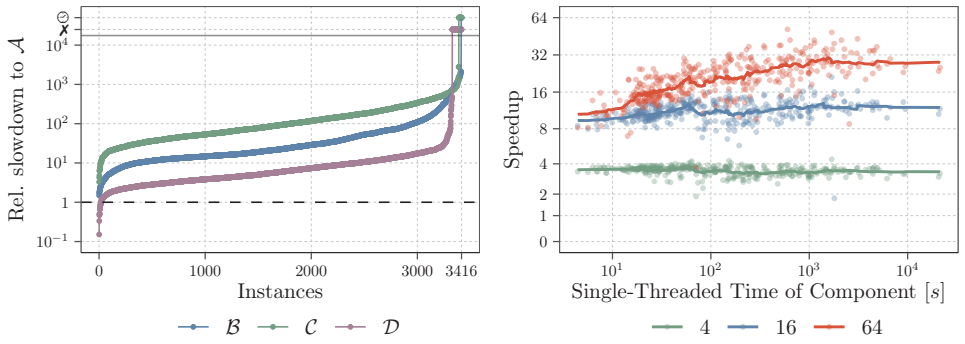



**Figure 2.8:** The relative running time plot compares the execution times of three algorithms $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ relative to a baseline algorithm $\mathcal{A}$ (left) and the speedup plot shows self-relative speedups of a parallel algorithm with an increasing number of threads (right).

sample one run of both algorithms for instance $I$. Let $t^1_{\mathcal{A}}, t^1_{\mathcal{B}}$ be their running times and assume that $t^1_{\mathcal{A}} \leq t^1_{\mathcal{B}}$. We then sample additional runs without replacement for $\mathcal{A}$ until their accumulated time exceeds $t^1_{\mathcal{B}}$ or all runs have been sampled. Let $t^2_{\mathcal{A}}, \ldots, t^l_{\mathcal{A}}$ denote their running times. We accept the last run with probability $(t^1_{\mathcal{B}} - \sum_{i=1}^{l-1} t^i_{\mathcal{A}})/t^l_{\mathcal{A}}$ so that the expected time for the sampled runs of $\mathcal{A}$ equals $t^1_{\mathcal{B}}$. The solution quality is the minimum out of the sampled runs. For each instance, we generate 10 virtual instances.

Figure 2.7 (right) shows the result of the effectiveness test for algorithm $\mathcal{A}$ and $\mathcal{B}$ using a performance profile. The plot shows that there is no significant difference between both algorithms when both are given the same amount of time to compute a solution for each instance.

### 2.4.4 Visualizing Running Times and Speedups

**Running Times.** Figure 2.8 (left) compares the execution times of three algorithms $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ relative to a baseline algorithm $\mathcal{A}$. The plot shows the slowdown of an algorithm relative to a baseline algorithm for each instance sorted in increasing order. We can see on how many instances the baseline algorithm is faster than another algorithm, and it also gives a good intuition for the magnitude of the improvements. Since the plot does not reveal any information on absolute execution times, we report geometric mean running times in the text when discussing these plots.

**Speedups.** Figure 2.8 (right) shows self-relative speedups of a parallel algorithm with an increasing number of threads. In the plot, we represent the speedup of each instance as a point and the centered rolling geometric mean with a window size of 25 as a line. The $x$-axis shows the single-threaded running time of the algorithm. In contrast

to many other publications in the parallel partitioning community, we do not correlate speedups to any of the common (hyper)graph metrics (such as the number of pins). We found that the running time often depends on a variety of different factors (e.g., pruning rules, sparsification techniques, or events that trigger repetitions). Fitting suitable parameters for a combination of the metrics seem much more complicated than plotting against sequential running time, which is often nicely correlated with speedups. Furthermore, the longer an algorithm runs sequentially, the more important is an efficient parallelization to achieve reasonable running times.

### 2.4.5 Statistical Significance Tests

In the performance profile in Figure 2.7 (left), it is not immediately obvious whether or not algorithm $\mathcal{B}$ performs better than $\mathcal{C}$ since both performance lines are close to each other. In this work, we use the Wilcoxon signed-rank test [Wil92] to determine whether the differences between partitioners with similar quality are statistically significant. At a 1% significance level ($p \leq 0.01$), a $Z$-score with $|Z| > 2.576$ is deemed significant [CS09, p. 180].

However, there is an ongoing controversy regarding statistical significance testing [Nuz14; WL16; MWS17; BD19; KW19]. These tests decide if the difference between two measurements is statistically significant but do not reveal any information on whether or not the difference is relevant in practice [Ang+19]. We therefore only use the Wilcoxon signed-rank test to show that the difference between two algorithms is *not* statistically significant.

Chapter 3

# 3 Related Work

This chapter gives an overview of existing algorithms for the (hyper)graph partitioning problem, focusing on techniques used in well-established sequential and parallel partitioning algorithms. Most of these algorithms are based on the multilevel paradigm as it provides an excellent tradeoff between solution quality and running time. We carefully analyze and describe the core components of these systems and identify techniques essential for achieving high solution quality. We also explain the main parallelization challenges and compare the best sequential and parallel partitioning systems in a short experimental evaluation demonstrating that parallel systems currently do not achieve the same solution quality as the highest-quality sequential algorithms. This will provide us with a starting point for our research, as the detailed analysis will reveal techniques for which there is no efficient parallelization or where compromises have been made.

**Excluded Topics.** The goal of this chapter is to provide an overview of techniques used in multilevel partitioning algorithms essential for achieving high solution quality in a reasonable amount of time. We therefore do not cover techniques that favor speed over solution quality, often omitting the multilevel paradigm and thus produce partitions of low quality. This excludes streaming partitioning algorithms [Tso+14; PGK19; FS21; JSA21], geometric partitioning algorithms [BB87; Sim91; Wil91; TN94; MK98; Dev+16; LTM18], and *flat* partitioning algorithms [Rah+13; ABM16; SMR16; Mar+17; Slo+17] that directly partition the input (hyper)graph into $k$ blocks. Furthermore, we are interested in algorithms that can be applied on (hyper)graphs with up to one billion pins/edges. Hence, we do not discuss techniques with high running times for instances at that scale, such as evolutionary algorithms [BM94; AY04; KKM04; SWC04; Arm+10; BH11a; SS12; ASS18], diffusion-based techniques [MMS08; MMS09; Mey12], or spectral partitioning [DH72; Bar82; BS93].

We refer the reader to survey articles [AK95; KAV04; PM07; BS11; Bul+16; Çat+22a] for a comprehensive overview on the (hyper)graph partitioning literature. Especially noteworthy is the dissertation of Schlag [Sch20], who presents the most recent and comprehensive overview of partitioning techniques from a historical perspective.

**Outline.** We start this chapter with a detailed discussion of the two most prominent local search algorithms in Section 3.1 – namely the Kernighan-Lin [KL70] and Fiduccia-Mattheyses algorithm [FM82]. We then turn to flow-based refinement in Section 3.2, which is considered the most powerful improvement heuristic for (hyper)graph partitioning. Section 3.3 introduces the multilevel paradigm and discusses the algorithmic components of existing sequential multilevel algorithms. We conclude this chapter

with an overview of the parallel partitioning literature in Section 3.4 with a particular focus on parallelization challenges and the building blocks of state-of-the-art parallel partitioning systems.

**References.** This chapter contains text passages from our conference publications [Got+21a; HMS21a; GHS22a; Got+22a], technical reports [Got+21c; GHS22c] and from a survey article [Çat+22a] in which we were involved as co-authors. However, most of it was written exclusively for this dissertation.

## 3.1 Iterative Improvement Algorithms

A widely-used algorithmic approach for optimization problems starts with an initial feasible solution and then *iteratively* improves it by exploring neighboring solutions until a local optimum is found. For the (hyper)graph partitioning problem, such algorithms move nodes across partition boundaries and accept the new solution if it improves the underlying objective function. Most early partitioning algorithms were based on this scheme until the multilevel paradigm was introduced in the mid-1990s. In their seminal work, Kernighan and Lin [KL70] proposed an iterative improvement algorithm "what is often described as the first *good* graph bisection heuristic" [AK95]. The algorithm exchanges node pairs between the two blocks based on their gain values. It also performs node moves that intermediately worsen the solution quality and therefore is able to escape from local optima. Fiduccia and Mattheyses [FM82] (FM algorithm) then further improved the algorithm by moving one node in each step instead of exchanging node pairs. In addition, the FM algorithm runs in linear time by using a novel data structure for maintaining the gain values of node moves. We describe both algorithms in more detail in the following since they are still used in many variations in today's partitioning systems.

### 3.1.1 Kernighan-Lin Algorithm

Kernighan and Lin [KL70] presented the first effective local search algorithm for the graph bisection problem that asks for a bipartition $\Pi = \{V_1, V_2\}$ with $|V_1| = |V_2|$. The general idea is that a bisection $\Pi = \{V_1, V_2\}$ can be transformed into an optimal solution $\Pi_{\mathrm{OPT}} = \{V_1^*, V_2^*\}$ by exchanging the nodes of $X := V_1 \setminus V_1^*$ and $Y := V_2 \setminus V_2^*$ between the two blocks of $\Pi$. However, identifying $X$ and $Y$ is NP-complete [KL70]. Thus, the algorithm approximates both sets by repeatedly pairing two nodes $u \in V_1$ and $v \in V_2$ that yield the largest edge cut reduction when both are moved to their opposite block. We refer to such an operation as an *exchange* operation.

**The Algorithm.** The Kernighan-Lin algorithm works in passes. At the beginning of each pass, the algorithm computes the gains $g_u(V_2)$ for all nodes $u \in V_1$ that corresponds to the reduction of the edge cut when $u$ is moved to block $V_2$ (for all nodes $v \in V_2$ analogously). Furthermore, each node is either *locked* or *unlocked*. Initially, all nodes are unlocked. In each pass, the algorithm repeatedly pairs two unlocked nodes

$u \in V_1$ and $v \in V_2$ leading to the largest edge cut reduction when both are exchanged. Subsequently, the algorithm performs the exchange operation, sets the state of $u$ and $v$ to locked, and updates the gains of the remaining unlocked nodes. A pass ends when all nodes are locked. The gain of exchanging $u$ and $v$ is $g_u(V_2) + g_v(V_1) - 2\omega(u, v)$. Since both individual gains assume that edge $\{u, v\}$ can be removed from the cut, we subtract $2\omega(u, v)$. The algorithm reverts to the best seen solution at the end of each pass by determining the index $l$ that maximizes the partial sum $\sum_{i=1}^{l} g_i$ where $g_i$ is the gain of the $i$-th exchange operation. The algorithm then continues with the next pass until it converges (i.e., $l = 0$). Note that it also performs moves that intermediately worsen the edge cut ($g_i < 0$). Thus, it can escape from local optima to some extent.

The algorithm can be extended to optimize the edge cut of $k$-way partitions by scheduling the bipartitioning algorithm on adjacent block pairs [KL70]. The algorithm performs several iterations over all block pairs. In later iterations, it only considers a block pair $(V_i, V_j)$ if either $V_i$ or $V_j$ changed in a previous iteration. Schweikert and Kernighan [SK72] generalized the algorithm to hypergraph partitioning.

**Running Time.** Kernighan and Lin [KL70] use an adjacency matrix to represent a graph. Thus, the initial gain computation scans all entries of the adjacency matrix, which can be done in $\mathcal{O}(n^2)$ time. In each pairing step, the algorithm sorts the gains of the nodes in block $V_1$ and $V_2$ separately in decreasing order. In general, finding the two nodes $u \in V_1$ and $v \in V_2$ that yield the largest edge cut reduction has a running time of $\mathcal{O}(n^2)$ (even if both lists are sorted). The operation can be implemented by fixing a node of block $V_1$ and then scanning down the sorted list of block $V_2$. However, Kernighan and Lin [KL70] noted that the scan could be aborted rapidly if it finds a pair whose exchange gain does not exceed the maximum gain seen so far. Hence, the running time of the scan is dominated by the sorting step, which is an $\mathcal{O}(n \log n)$ operation. After an exchange operation, the algorithm updates the gains of the remaining unlocked nodes. For each node $x \in V_1$, it adds $2\omega(x, u)$ and subtracts $2\omega(x, v)$ from $g_x(V_2)$. The gain updates for all nodes $y \in V_2$ can be done analogously. Therefore, the running time to update the gains of all unlocked nodes is $\mathcal{O}(n)$. Determining the maximum index $l$ that maximizes $\sum_{i=1}^{l} g_i$ at the end of each pass is a linear time operation. In each pass, the algorithm performs $n$ exchange operations. Thus, the overall running time of a pass is $\mathcal{O}(n^2 \log n)$. The number of edges bounds the number of passes. However, several experimental studies report that the algorithm converges within a constant number of passes [KL70; DK85; Dut93].

Dutt [Dut93] reduced the running time of the algorithm to $\mathcal{O}(m \max(\log(n), \Delta_u))$ per pass where $\Delta_u$ is the maximum node degree of the input graph. The implementation uses an adjacency list to represent the graph, allowing more efficient gain computations and updates. Furthermore, the pairing step only considers the first $d(u) + 1$ values of the sorted list of $V_2$ for a node $u \in V_1$ since we can abort the scan if we encounter the first non-adjacent node of $u$. The algorithm also uses an AVL tree [AHU74] to keep the gain values sorted.

### 3.1.2 Fiduccia-Mattheyses Algorithm

The Kernighan-Lin algorithm was the first effective local search heuristic for the graph bisection problem that was able to escape from local optima. However, it also has several disadvantages. In order to handle weighted instances, Kernighan and Lin [KL70] replaced each node $u \in V$ with $c(u)$ auxiliary nodes and connected them to a clique with edges of appropriately high cost such that it is unlikely that a good bisection will cut these edges. However, this transformation significantly increases the size of the graph, and a bisection can cut some of the high-cost edges. Moreover, the algorithm only performs moves that preserve the balance, while the definition of the traditional (hyper)graph partitioning problem allows some leeway in the block weights. Lastly, and most importantly, the main drawback is its high computational complexity.

To this end, Fiduccia and Mattheyses [FM82] presented the first linear time local search heuristic (*FM* algorithm) for the hypergraph partitioning problem that overcomes the drawbacks of the Kernighan-Lin algorithm. The FM algorithm proceeds also in passes, but it moves only one node at a time instead of exchanging nodes in each step. The algorithm achieves a running time of $\mathcal{O}(p)$ per pass by using a priority queue specifically tailored to the hypergraph partitioning problem. The algorithm is still used in many variations in existing partitioning algorithms. In this dissertation, we present the first fully-parallel direct $k$-way formulation of the FM algorithm. Therefore, we will explain the core algorithm that operates on bipartitions, extensions to $k$-way partitioning, and some variations of the algorithm that we use in our work.

**The Problem.** Unlike many other techniques in the partitioning context, the FM algorithm was originally developed for hypergraphs with node weights and unit hyperedge weights (adaption to non-uniform hyperedge weights is straightforward, as explained later). It improves the cut-net metric of a bipartition $\Pi = \{V_1, V_2\}$. A bipartition is considered balanced if block $V_1$ satisfies

$$r \cdot c(V) - \max_{u \in V} c(u) \leq V_1 \leq r \cdot c(V) + \max_{u \in V} c(u)$$

for some imbalance parameter $0 < r < 1$. Adding the maximum node weight to the balance definition ensures that a node can always be moved to either $V_1$ or $V_2$.

**The Algorithm.** The FM algorithm proceeds in passes similarly to the Kernighan-Lin algorithm. Initially, it computes the gains $g_u(V_2)$ and $g_v(V_1)$ of all nodes $u \in V_1$ and $v \in V_2$ and inserts them into two separate priority queues (PQ), each representing one block of the bipartition. The nodes contained in the PQs are *free* to move to their opposite block. Once a node is moved, it is *locked* in its current block until the end of the pass. In each pass, the algorithm repeatedly selects the move with the highest gain, performs it, and updates the gain of all free neighbors. A pass ends when all nodes are locked, or the balance constraint prevents further node moves. Subsequently, the algorithm reverts to the best seen solution and continues with the next pass until no further improvements are possible.

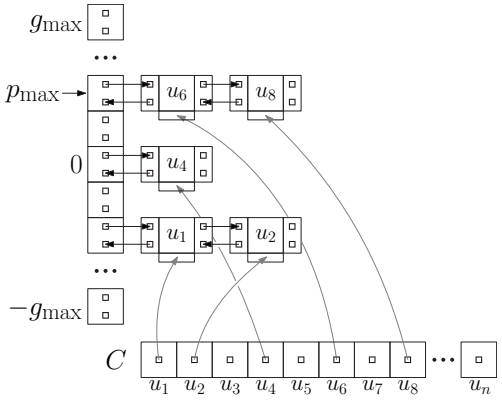



**Figure 3.1:** Illustration of the bucket priority queue data structure used in the FM algorithm.

Since the FM algorithm operates on hypergraphs with node weights, it must ensure that the move with the highest gain does not violate the balance constraint. Note that if both PQs are non-empty, there is always a node that can be moved to its opposite block due to the definition of the balance constraint. The algorithm first selects the node moves with the highest gain and then rejects moves that would result in an imbalanced bipartition. Among the remaining moves, it chooses the one leading to the best balance. A tie-breaking scheme can be used if there is still more than one candidate.

A time-consuming part of the algorithm is maintaining the gain values in a pass. Fiduccia and Mattheyses [FM82] use *delta gain updates* to update the gain values of all neighbors of a node $u$ that moved from a block $V_i$ to $V_j$. The update procedure iterates over all nets $e \in \mathrm{I}(u)$ and decides whether or not the gains of the remaining free pins of $e$ have to be updated based on the pin count values $\Phi(e, V_i)$ and $\Phi(e, V_j)$. They showed that only four cases trigger a gain update, and updates occur at most four times per net and pass. Since, as we will see in the next paragraph, the PQ provides amortized constant time operations for all required operations of the algorithm, the overall running time of a pass is $\mathcal{O}(p)$.

**Bucket Priority Queue.** The bucket priority queue data structure uses the observation that the maximum node degree $\Delta_u$ (assuming unit hyperedge weights) bounds the highest possible gain value $g_{\max}$ of a node move. The data structure then uses $2g_{\max} + 1$ gain *buckets*, each representing a gain value in the range $[-g_{\max}, \ldots, g_{\max}]$. Each bucket stores node moves with a particular gain value in a doubly-linked list. Furthermore, it uses an array $C$ of size $n$ storing handles to the entries in the doubly-linked lists for each node. Additionally, a pointer $p_{\max}$ points to the bucket containing the node moves with the highest gain. Figure 3.1 illustrates the data structure.

Inserting a node into the bucket priority queue entails inserting it at the end of the doubly-linked list of its corresponding gain bucket and storing a handle to it in

$C$. Updating a gain value of a node move can be done by identifying the node over its handle in $C$, removing it from its current doubly-linked list and inserting it into its new gain bucket. Removing a node can be done similarly. The node moves with the highest gain can be identified via $p_{\max}$. The pointer $p_{\max}$ is updated if a node is inserted with a gain value greater than $p_{\max}$. If the bucket to which $p_{\max}$ points to becomes empty, we linearly scan downwards to find the next non-empty bucket. Fiduccia and Mattheyses [FM82] showed that the time required to update $p_{\max}$ is in $\mathcal{O}(p)$ per pass.

This result does not hold for hypergraphs with hyperedge weights since the highest gain value is then bounded by $\Delta_u \cdot \max_{e \in E} \omega(e)$. More recent FM implementations therefore switched to binary heaps [OS10; SS11; Sch+16a]. Schlag [Sch20, p. 154] compared both approaches and concluded that binary heaps are slightly faster than bucket priority queues for weighted instances. In his experiments, he used a space-efficient bucket priority queue implementation for weighted hypergraphs that were proposed by Papa and Markov [PM07]. The implementation uses a binary search tree maintaining the distinct gain values and a hash table storing the gain buckets.

**Extensions to *k*-way Partitioning.** Sanchis [San93] and Hendrickson and Leland [HL95] proposed a variant of the FM algorithm for $k$-way partitioning using $k(k-1)$ bucket priority queues (one for each move direction). Additionally, it uses a binary heap storing the node moves with the highest gain of each bucket priority queue (PQ). Cong and Lim [CL98] showed that the $k$-way FM algorithm of Sanchis [San93] is outperformed by a 2-way FM implementation that obtains a $k$-way partition via recursive bipartitioning. They noted that due to the high degree of flexibility, the $k$-way FM algorithm tends to make wrong decision and increases the probability of getting stuck in a local minima. Moreover, the algorithm has a space complexity of $\mathcal{O}(nk(k-1))$, which makes it impractical for larger problem instances.

To this end, Cong and Lim [CL98] presented the K-PM algorithm that reduces the $k$-way partitioning problem to sets of concurrent bipartitioning problems. At the beginning of a FM pass, the algorithm computes a matching between the blocks of the partition and then only allows node moves between paired blocks. Thus, it requires only $k$ PQs. Zien et al. [ZCS97] proposed the Rotary KLFM algorithm that splits each FM pass into $k$ phases. In each phase, one block is chosen as the target block, and then it considers only moves to and out of that particular block. This results in $2(k-1)$ possible move directions. Osipov and Sanders [OS10] use $k$ PQs in their FM implementation. Each PQ represents one block and stores node moves to that particular block. Sanders and Schulz [SS10; SS11] reduced this to a single PQ considering only the move with the highest gain for a node.

**Stopping Rules.** The Kernighan-Lin and FM algorithm continue a refinement pass until all nodes are moved to their opposite block or the balance constraint prevents further node moves. Techniques reducing the complexity of a pass are based on the idea of aborting a pass early when further improvements become unlikely. Gilbert and Zmijewski [GZ87] abort a pass when the sum of the observed gain values is less than $-\Delta_u$ or when the algorithm performs $\Delta_u$ consecutive negative gain moves.

Hendrickson and Leland [HL95] terminate when the difference between the best solution seen so far and the current solution becomes too large. Karypis and Kumar [KK98c] abort when the last $x$ node moves have not improved the overall edge cut. Osipov and Sanders [OS10] use an adaptive stopping rule. They assume that the observed gain values of the last $p$ node moves follow a normal distribution with expectation $\mu$ and variance $\sigma^2$. Then they argue that it is unlikely to find further improvements if $p\mu^2 > \alpha\sigma^2 + \beta$ where $\alpha$ and $\beta$ are tuning parameters. The parameter $\beta$ $(= \log n)$ avoids that a local search stops after a few steps. Schlag [Sch20, p. 155] showed that his FM implementation with the adaptive stopping rule is an order of magnitude faster than the stopping rule of Karypis and Kumar [KK98c] with $x = 350$.

**Highly-Localized FM Searches.** The FM algorithm chooses the node move with the highest gain in each step. If there are several feasible moves with the same gain value, one node must be chosen using a tie-breaking scheme. Hagen et al. [HHK97] evaluated the following tie-breaking strategies to choose the best move from the highest gain bucket: pick a random node move, select the node move inserted last (LIFO) or first (FIFO) into the gain bucket. It seems that this is only a minor implementation detail, but the LIFO significantly outperformed the FIFO and random tie-breaking strategy. Hauck and Borriello [HB97] gave a possible explanation for this behavior. After a node is moved, the gains of some of its neighbors are updated. A gain update requires removing it from its current bucket and inserting it at the tail of the new bucket. This makes it more likely that the algorithm will move neighbors of an already moved node next. Thus, the search expands around moved nodes, which seems to be beneficial.

The graph partitioner KaFFPa [SS10; SS11] implements a *highly-localized* version of the FM algorithm using a single PQ initialized with a single node and its neighbors. The search then gradually expands around the seed nodes by inserting neighbors of moved nodes into the PQ. The algorithm uses the adaptive stopping rule of Osipov and Sanders [OS10] to abort a search early. Afterwards, a new seed node is used to start the next search. The expansion step only considers boundary nodes that were previously not touched by another search. A pass ends when there are no remaining seed nodes left. This procedure is then repeated for a constant number of passes. We present a parallel version of this algorithm in this work and will later refer to it as the *multi-try k-way FM* algorithm.

## 3.2 Flow-Based Refinement

The previously presented local search algorithms greedily move nodes to different blocks according to a gain value. The gain of a node move depends on its local neighborhood. However, there are situations where the optimal move sequence does not follow a greedy pattern.

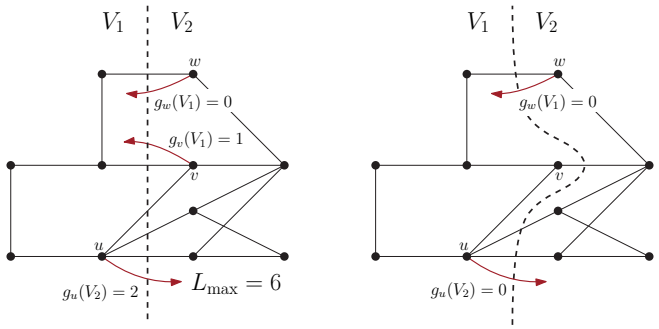



**Figure 3.2:** Illustration of a situation where the FM algorithm makes a suboptimal decision. The node $u$ can not be moved to block $V_2$ as this would result in an imbalanced bipartition.

> Figure 3.2 illustrates a situation where the FM algorithm makes a suboptimal decision. Clearly, moving node $u$ to block $V_2$ yields the largest edge cut reduction, but would result in an imbalanced bipartition ($c(V_2) + c(u) = 7 > 6 = L_{\max}$). Therefore, the FM algorithm moves the node with the highest gain of block $V_2$ to $V_1$, which is node $v$ with $g_v(V_1) = 1$. This locks $v$ in block $V_1$ and prevents further improvements in the current pass. However, moving $w$ instead of $v$ would enable the move of $u$ to $V_2$ and reduce the edge cut by 2 (instead of 1).

For hypergraphs, the presence of large nets makes it especially difficult to find meaningful moves [Saa95]. A net $e$ contributes positively to the gain of a node $u$, e.g., for the cut-net or connectivity metric, if and only if $u$ is the last remaining pin of $e$ in its current block. Thus, for hypergraphs with large nets, many of the moves have zero gain, and the outcome of local search algorithms depends mainly on tie-breaking decisions.

It seems natural to use the max-flow min-cut theorem [FF56] to overcome the limitations of existing move-based local search heuristics. The theorem relates the *global* minimum cut separating two nodes $s$ and $t$ of a graph $G$ to the maximum flow between $s$ and $t$ in the corresponding flow network of $G$. However, it was overlooked for a long time because it was unclear how to obtain balanced bipartitions and it was perceived as computationally expensive [KL70; YW96]. This perception changed over the last two decades since several publications demonstrate the effectiveness of flow-based refinement for balanced (hyper)graph partitioning. Today it is considered the most powerful refinement technique and is used in the highest-quality (hyper)graph partitioners KaFFPa [SS10; SS11] (optimizes the edge cut metric) and KaHyPar [HSS19a; Got+20] (optimizes the cut-net and connectivity metric). In this work, we present a parallel formulation of the flow-based refinement routine used in KaHyPar [Got+20] that generalized the approach of KaFFPa [SS10; SS11] from graphs to hypergraphs. Therefore, we outline the core ideas of these algorithms and explain them in more detail in Section 4.4.

**Flows on Hypergraphs.** Lawler [Law73] reduced the problem of finding a minimum $(s,t)$-cut in a hypergraph to computing a maximum $(s,t)$-flow in a directed graph (see Lawler expansion in Section 2.1). Liu and Wong reduced the size of the Lawler expansion by explicitly distinguishing between graph edges ($|e| = 2$) and hyperedges ($|e| > 2$), while Heuer et al. [HSS19a] additionally removed low-degree nodes ($d(u) \leq 3$).

There exist several algorithms computing maximum flows directly on hypergraphs. Li et al. [LLC95] presented a variant of the push-relabel algorithm of Goldberg and Tarjan [GT88], while Pistorius and Minoux [PM03] implemented the Edmonds-Karp [EK72] algorithm for hypergraphs. Recently, Gottesbüren et al. [GHW19] generalized Dinic's algorithm [Din70] to hypergraphs.

**Balanced Flow-Based Refinement.** Yang and Wong [YW96] proposed the FBB algorithm (Flow-Balanced-Bipartition) that finds a balanced bipartition of a hypergraph via incremental maximum flow computations. The algorithm computes a minimum $(s,t)$-cut, and if the induced bipartition is not balanced, the smaller side plus one additional node (also called *piercing* node) is contracted onto the corresponding terminal, i.e., $s$ or $t$ respectively. This ensures that the algorithm finds a different cut (possibly a larger cut) in the next iteration but with better balance. The algorithm then augments the previous flow again to a maximum flow. Li et al. [LLC95] improved the FBB algorithm with better piercing decisions. Hamann and Strasser [HS18] propose the FlowCutter algorithm that works similar to the FBB algorithm but computes several bipartitions with increasing cut sizes and better balance. Gottesbüren et al. [GHW19] generalized the FlowCutter algorithm to hypergraphs (HyperFlowCutter).

> Figure 3.3 illustrates one iteration of the FBB algorithm. We denote with $S$ and $T$ the sets of nodes that are already contracted onto the source $s$ and sink $t$. The sets $S_r$ and $T_r$ define the nodes reachable from the source $s$ and sink $t$ after a maximum flow computation. We assume that both bipartitions $(S \cup S_r, V \setminus (S \cup S_r))$ and $(T \cup T_r, V \setminus (T \cup T_r))$ induced by the first maximum flow computation are not balanced (see left side of Figure 3.3). Then the algorithm contracts $S_r$ plus the piercing node $u$ onto the source $s$ (assuming $|S \cup S_r| \leq |T \cup T_r|$) and augments the previous flow again to a maximum flow (see right side of Figure 3.3). This induces a new cut (potentially larger) with better balance.

Sanders and Schulz [SS10; SS11] integrated a flow-based refinement routine into their graph partitioner KaFFPa. The algorithm improves the edge cut of a given bipartition $\Pi = \{V_1, V_2\}$. The general idea is to grow a size-constrained region $B$ around the cut edges of $\Pi$. All nodes of $V_1 \setminus B$ and $V_2 \setminus B$ are then contracted to the source $s$ and sink $t$. The size of region $B$ is chosen such that each minimum $(s,t)$-cut yields a balanced bipartition in the original graph. If the algorithm improves $\Pi$ (better cut or balance), then the size of region $B$ is doubled in the next iteration. Otherwise, it is halved. The algorithm terminates if the size of region $B$ becomes smaller than a predefined threshold. The algorithm can be used to improve the edge

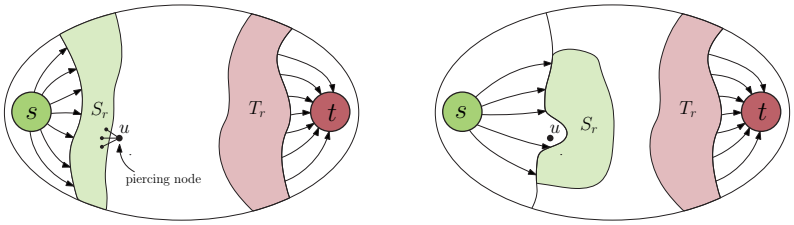



**Figure 3.3:** Example of one iteration of the FBB algorithm.

cut of a $k$-way partition by scheduling it on adjacent block pairs (similarly as already proposed for the Kernighan-Lin algorithm [KL70]). Heuer et al. [HSS19a] integrated this approach into their hypergraph partitioner KaHyPar, which was then further improved by Gottesbüren et al. [Got+20] by replacing the bipartitioning routine with the HyperFlowCutter algorithm [HS18; GHW19].

Liu and Wong [LW98] enhanced the FBB algorithm with the *most balanced minimum cut* heuristic. The heuristic uses the observation that there is usually more than one minimum $(s,t)$-cut. The algorithm then enumerates all minimum $(s,t)$-cuts and chooses the one with the best balance. The technique is also used in the (hyper)graph partitioner KaFFPa [SS10; SS11] and KaHyPar [HSS19a] and implicitly in the FlowCutter algorithm [HS18; GHW19] (piercing nodes that avoid augmenting paths in the next iteration). Piccard and Queyranne [PQ80] provided the theoretical foundations for this heuristic. Their theorem establishes a one-to-one correspondence between all minimum $(s,t)$-cuts and the strongly connected components (SCC) in the residual network of a maximum $(s,t)$-flow. One can then enumerate different minimum cuts by contracting all SCCs in the residual graph into a single node which yields a directed acyclic graph (DAG). Sweeping through a topological ordering of the DAG then induces different bipartitions with the same cut value. Note that there usually exists more than one topological ordering and finding the minimum $(s,t)$-cut with the best balance is still NP-complete [PQ80].

Notable other flow-based refinement algorithms are Improve [AL08] and MQI [LR04], which optimize the expansion or conductance metric of bipartitions. Delling et al. [Del+11] use flow techniques to find natural cuts in road networks. Schild and Sommer [SS15] use geometric information available in road networks to compute a linear embedding of the nodes, which is then utilized to construct the source and sink sets of a flow network.

## 3.3 The Multilevel Scheme

Iterative improvement algorithms were long used as the main approach for solving the hypergraph partitioning problem with the FM algorithm as the most successful technique. The original FM algorithm [FM82] starts from a randomly generated initial bipartition and then iteratively improve it by moving nodes between the blocks until

no further improvements are possible. A widely used technique enhancing its solution quality performs multiple restarts using different initial solutions and returns the best found bipartition. Alpert and Kahng [AK95, p. 61] analyzed the edge cuts produced by the FM algorithm when repeated executions are used. They noted that the solution quality of the bipartitions follows roughly a normal distribution, and the average quality of the bipartitions was significantly worse than the best found solution. Thus, the linear running time of the FM algorithm is outweighed by the fact that many repetitions are required to achieve high solution quality. Bui et al. [Bui+87; Bui+89] compared several iterative improvement algorithms for the graph bisection problem to the edge cut of an optimal bisection on randomly generated graphs. They observed that the Kernighan-Lin algorithm [KL70] finds near-optimal solutions for graphs with a high average node degree. However, the edge cuts deteriorate significantly for graphs with a low average node degree ($\leq 3$). For low-degree graphs, the gain values of node moves are relatively small (bounded by the degree of a node). Thus, the outcome of the algorithm mainly depends on tie-breaking decisions. The same behavior can be observed for hypergraphs with many large nets [Saa95; CA99; Kar+99] since a net $e$ only positively contributes to the gain value of a node $u$ when $u$ is the last remaining pin of $e$ in its corresponding block. The Kernighan-Lin and FM algorithm can break out of local optima by temporarily increasing the cut size of the solution. However, as the problem size increases, the number of local optima increases significantly reducing the likelihood of finding a solution close to the global optimum [CS93].

These observations suggest that it is beneficial to reduce the size of the input hypergraph while increasing its average node degree and reducing the size of large nets. Ishiga et al. [IKS75] presented the first algorithm that accomplishes this (even though many of the problems were not known at the time). The algorithm finds a clustering of the nodes and contracts them to obtain a coarser instance of the input hypergraph. Then the algorithm uses the Kernighan-Lin algorithm [KL70] to compute an initial partition on the coarser representation which is then projected back to the input hypergraph. They noted that the contraction step *buried* a large number of nets within the clusters, and therefore, the coarse hypergraph represents a simpler form that is easier to partition. These algorithms are known as *two-level* algorithms, i.e., the first level corresponds to the input (hyper)graph, while the second represents a coarser approximation obtained by a clustering algorithm.

Saab [Saa95] observed that densely-connected nodes are good cluster candidates since such nodes tend to stay in the same block of the partition after one pass of the FM algorithm. Due to the way contractions are performed, a partition on a coarse representation can be projected back to the input (hyper)graph with the same cut and balance properties. This allows refinement algorithms to operate on different scales [HL95]. Performing local search on coarser levels moves clusters across partition boundaries and therefore, it can explore a greater solution space than on the input (hyper)graph. Alpert et al. [AHK97] also noted that local search algorithms converge faster after the uncontraction step since they already start with a good solution.

The introduction of the two-level approach considerably improved the performance of move-based local search algorithms [Bui+89]. However, Alpert et al. [AHK97]

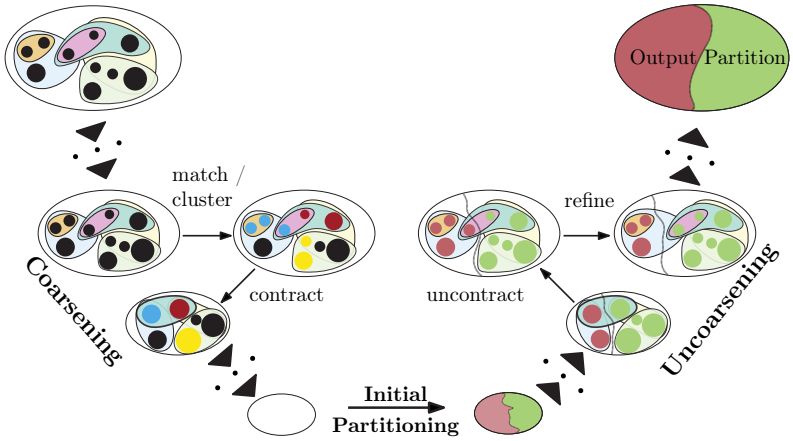



**Figure 3.4:** The multilevel scheme. Figure is taken from Ref. [Sch20].

noted that clustering techniques used in these algorithms reduce the size of the input (hyper)graph *too* aggressively, preventing refinement algorithms from finding good local optima. An alternative is to apply the clustering algorithm recursively and control the size of the coarser approximation on each level. This inspired several research groups [BS93; BJ93; CS93; HB95; HL95] to study the multilevel scheme as it is used in most of today's state-of-the-art partitioning algorithms.

Figure 3.4 illustrates the multilevel scheme that proceeds in three phases: First, the (hyper)graph is *coarsened* to obtain a hierarchy of successively smaller and structurally similar (hyper)graphs by *contracting* pairs or clusters of nodes. Once the coarsest (hyper)graph is sufficiently small, an *initial partition* into $k$ blocks is computed. Subsequently, the contractions are reverted level-by-level, and, on each level, *local search* heuristics are used to improve the partition from the previous level (*refinement phase*).

Hendrickson and Leland [HL95] showed that their multilevel partitioning method offers an excellent tradeoff between quality and running time. It computes partitions with comparable edge cuts to the multilevel spectral partitioning approach of Bernard and Simon [BS93] in a fraction of the time. Hauck and Borriello [HB97] showed in an extensive study that multilevel algorithms produce partitions with significantly better edge cuts than two-level approaches, while being only moderately slower.

This section discusses techniques used in sequential multilevel partitioning algorithms with the aim of identifying methods that produce high quality solutions. We start with introducing the label propagation algorithm in Section 3.3.1, which is a popular technique used in all phases of the multilevel scheme. Section 3.3.2 then explains the $n$-level scheme that instantiates the multilevel paradigm in its most extreme version by contracting only a single node on each level. We conclude this section by comparing the algorithmic components used in existing partitioning algorithms in Section 3.3.3.

### 3.3.1 The Label Propagation Algorithm

The label propagation algorithm is a greedy heuristic applicable in all phases of multilevel scheme. Furthermore, it is directly amenable to parallelization due to its simplicity. The algorithm was first mentioned in the context of semi-supervised learning to learn missing labels from existing graph data [ZG02] and later also used to detect communities in large-scale networks [RAK07]. Earlier work in the partitioning context refers to the same method as *greedy refinement* [KK98c; KK00].

The original algorithm associates with each node $u \in V$ a *label* $L[u]$. Initially, it assigns each node $u$ its own label $L[u] = u$. Then, the algorithm works in rounds. In each round, it visits all nodes in some order. If it visits a node $u$, it assigns $u$ the label that occurs most frequently in its neighborhood $\Gamma(u)$ (ties are broken randomly). The algorithm proceeds until a predefined number of rounds are reached, or none of the nodes changed their label in a round.

In the coarsening phase, the label propagation algorithm can be used to compute a clustering $\mathcal{C}$. All nodes with the same label belong to one common cluster. The algorithm often uses a size-constrained $U$ bounding the weight of the heaviest cluster to prevent shrinking the (hyper)graph too aggressively [MSS14]. In the refinement phase, it can be used as local search algorithm by initializing the labels with a $k$-way partition and $U = L_{\max}$ [KK98c; KK00; ACU08b; MSS14]. The algorithm then moves a node $u$ to the block $V_j$ with the highest gain $g_u(V_j)$. The algorithm only performs moves improving the objective function (positive gain moves), and therefore it cannot escape from local optima [BC09]. The algorithm has a running time of $\mathcal{O}(p)$ per round, and it usually converges within a few rounds [KK98c].

### 3.3.2 $n$-Level Hypergraph Partitioning

Traditional multilevel algorithms contract matchings or clusters on each level, inducing a multilevel hierarchy with approximately logarithmic depth. There is usually a correspondence between the number of levels and the tradeoff between solution quality and running time [Saa95; Sch20]. More levels provide "more opportunities to refine the current solution" [AHK97] but require highly-engineered algorithmic components to achieve reasonable running times. Osipov and Sanders [OS10] introduced the $n$-level scheme instantiating the multilevel paradigm in its most extreme version, contracting only a single node on each level. Correspondingly, in each uncoarsening step, only a single node is uncontracted, allowing a highly localized search for improvements. Schlag [Sch20] developed KaHyPar, the first hypergraph partitioner based on the $n$-level approach, in which he demonstrated that the scheme can be implemented efficiently in numerous publications [Sch+16a; Akh+17a; HS17a; HSS19a; Got+20] and produces partitions of high quality. In this work, we present a parallel formulation of the $n$-level technique that closely models the approach of KaHyPar. Thus, we explain the dynamic hypergraph data structure of KaHyPar in more detail and briefly discuss its algorithmic components.

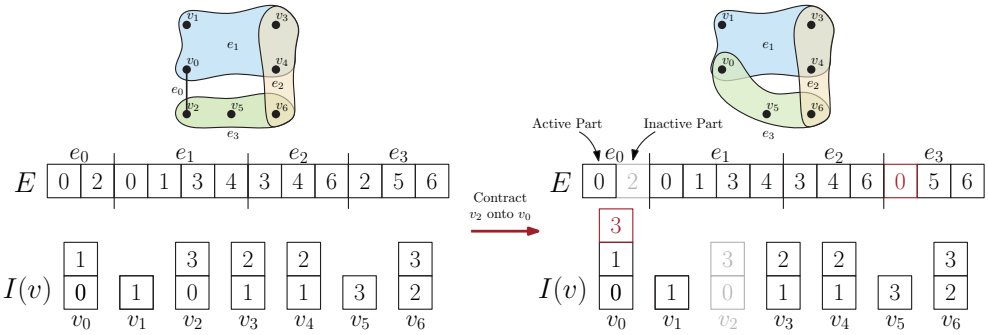



**Figure 3.5:** The dynamic hypergraph data structure used in KaHyPar.

**The Dynamic Hypergraph Data Structure of KaHyPar.** KaHyPar uses a dynamic hypergraph data structure to perform contractions *on-the-fly*, which is illustrated in Figure 3.5. The pins of nets are represented as an adjacency array $E$ (the sub-range storing the pins of a certain net is called its pin-list), whereas the incident nets $\mathrm{I}(v)$ of a node $v$ are represented using a adjacency-list. A contraction operation $(v, u)$ (contracting $v$ onto $u$) replaces $v$ with $u$ in each net $e \in \mathrm{I}(v) \setminus \mathrm{I}(u)$ and removes $v$ in each net $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$. Removing $v$ from a net $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$ swaps it to the end of the pin list of $e$ and decrements the size of $e$ by one, which divides $e$ into an *active* and *inactive* part (see right side of Figure 3.5). The incident nets of $u$ are updated by copying the nets $\mathrm{I}(v) \setminus \mathrm{I}(u)$ to the adjacency list of $u$. The contractions must be reverted in reverse order of contraction. To revert a contraction $(v, u)$, it considers all nets $e \in \mathrm{I}(v)$ to which $v$ was previously adjacent. If $v$ is the first pin in the inactive part of the pin list of $e$, we increment the size of the active part. Otherwise, we replace $u$ with $v$.

**Algorithmic Components of KaHyPar.** KaHyPar's coarsening algorithm proceeds in passes until the number of nodes drops below $160k$. In each pass, the nodes are visited in random order and whenever a node $u$ is visited, it is contracted onto the node $v$ that maximizes the heavy edge rating function $r(u, v) := \sum_{e \in \mathrm{I}(u) \cap \mathrm{I}(v)} \omega(e)/(|e| - 1)$. Initial partitioning uses $n$-level recursive bipartitioning with a portfolio of initial bipartitioning techniques [Heu15a; Sch+16a]. After reverting a contraction, KaHyPar starts a highly-localized version of the FM algorithm using the adaptive stopping rule of Osipov and Sanders [OS10]. The FM search is initialized with the uncontracted nodes and gradually expands around them (for more details, see Section 3.1.2). To improve performance, it uses a *gain table* storing the gain values for each possible node move. This prevents expensive recomputations during an FM pass. The gain table is initialized with the gain values of all possible node moves on the coarsest hypergraph. The FM algorithm maintains the gain values of the gain table using delta gain updates [Akh+17a; Sch20]. After an uncontraction, it recomputes the gain table entries of the uncontracted nodes. In addition to the FM algorithm, KaHyPar also uses flow-based refinement [Heu18a; HSS19a; Got+20].

### 3.3.3 Algorithmic Components of Sequential Partitioners

This section provides an overview on existing sequential partitioning algorithms with the aim to identify essential techniques for producing high quality solutions. For this purpose, we review the building blocks of (hyper)graph partitioners frequently appearing in the evaluations of recent publications. Table 3.1 summarizes the different partitioners and their algorithmic components discussed in this section.

We focus on techniques for the traditional (hyper)graph partitioning problem as defined in Section 2.2 (excluding, e.g., fixed nodes, multi-constraint or multi-objective formulations). Note that (meta)heuristics achieving even better solution quality than the multilevel scheme exist, e.g., V-cycles [Wal04], evolutionary algorithms [SS12; ASS18] or approaches based on integer linear programming [HNS20]. However, they have substantially longer running times, which are prohibitive for very large (hyper)graphs. Therefore, we limit the scope to techniques used within coarsening, initial partitioning, and refinement.

**Coarsening.** Early work on multilevel partitioning primarily used matching-based coarsening schemes [HL95; KK98a; KK98c; WC00a; VB05; MMS08; SS11]. A *matching* of a graph is a set of edges, no two of which are incident on the same node. A node is called *matched* if it is adjacent to one edge in the matching. Coarsening algorithms based on this scheme compute a maximal matching (matching where each edge of the graph is adjacent to at least one matched node) and subsequently contract all edges of the matching to obtain the next coarser graph. On hypergraphs, matching-based coarsening algorithms implicitly work on the clique expansion [Kar+99] (each hyperedge is replaced by a clique). A common approach iterates over all nodes in some order, and whenever an unmatched node $u$ is visited, it is matched with an unmatched neighbor $v \in \Gamma(u)$ that maximizes a rating function $r(u, v)$ [KK98a; KK98c; WC00a] (*local greedy matching*). Table 3.2 lists different rating functions. An alternative approach iterates over the edges in descending order of their weight and adds an edge to the matching if both adjacent nodes are unmatched (*global greedy matching*). KaFFPa [SS10; Sch13] uses the *Global Path Growing* (GPA) algorithm [DH03] to construct a matching. The algorithm works similarly to the global greedy matching algorithm but does not immediately build the matching. Instead, it constructs subgraphs composed of paths and cycles and uses dynamic programming to compute optimal solutions for the subgraphs. Holtgrewe et al. [HSS10] showed that the GPA algorithm with expansion$^2(u, v)$ as a rating function produces significantly better edge cuts than the local and global greedy matching technique with weight$(u, v)$ as a rating function.

Matching-based coarsening algorithms work well for (hyper)graphs with a regular structure, e.g., finite element meshes with uniform node degree distributions. However, complex networks with power-law node degree distributions are difficult to coarsen with matching-based approaches. Here, many low-degree nodes are adjacent to a few high-degree nodes. This leads to small matchings and consequently to a large number of levels [AK06]. Therefore, many partitioning algorithms switched to clustering-based coarsening schemes. On each level, densely-connected nodes are grouped and

**Table 3.1:** Algorithmic components of existing sequential partitioning algorithms.

| | | Chaco | Scotch | Metis | Jostle | KaSPar | KaFFPa | hMetis | PaToH | Mondriaan | KaHyPar |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Design | Type | GP | GP | GP | GP | GP | GP | HGP | HGP | HGP | HGP |
| | Direct $k$-Way | ◐[1] | ○ | ● | ● | ● | ● | ● | ◐[2] | ○ | ● |
| | Recursive Bipartitioning | ● | ● | ● | ○ | ○ | ○ | ● | ● | ● | ● |
| Coarsening | Community Detection | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| | $n$-Level | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ● |
| | Matching | ● | ? | ● | ● | ○ | ● | ● | ● | ● | ○ |
| | Clustering | ○ | ? | ◐[3] | ○ | ○ | ● | ● | ● | ○ | ○ |
| IP | Direct $k$-Way | ● | ● | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ |
| | Recursive Bipartitioning | ○ | ○ | ● | ○ | ○ | ● | ● | ● | ○ | ● |
| | Portfolio | ○ | ● | ● | ○ | ○ | ○ | ● | ● | ○ | ● |
| | Third-Party | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ |
| Refinement | LP | ○ | ○ | ● | ○ | ○ | ● | ● | ● | ○ | ○ |
| | Boundary FM | ● | ◐[4] | ◐[4] | ● | ○ | ● | ◐[4] | ◐[4] | ◐[4] | ○ |
| | Localized FM | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ | ● |
| | Flows | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● |
| | Open Source | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ● |
| | Publicly Available | ● | ● | ● | ○ | ○ | ● | ● | ● | ● | ● |

subsequently contracted into a single node. To prevent coarsening from reducing the size of the (hyper)graph too aggressively, many partitioners abort the clustering process when the size of the coarser approximations drops below half the size of the input (hyper)graph [KK00; AK06]. Moreover, high-degree nodes commonly appear in larger clusters. Thus, high-degree nodes become heavy nodes in the coarsest (hyper)graph, making it difficult or even impossible to find a feasible initial partition [MSS14]. Therefore, a common technique enforces a size-constrained on the weight of the heaviest cluster [MSS14; Sch+16a; Akh+17a] or penalizes the contraction of heavy nodes in the rating function [CA99; SS10; Sch13].

Clustering algorithms used in practice are based on size-constrained label propagation [MSS14] (see Section 3.3.1) or hierarchical agglomerative clustering [CA99; KK00; AK06; ACU08b]. The latter proceeds similarly to the previously presented matching

[1] The FM implementation of Chaco uses $k(k-1)$ PQs. To improve performance, Chaco restricts the number of blocks to at most 8 and uses octa-sectioning until the graph is partitioned into the desired number of blocks.

[2] There exists a direct $k$-way version of PaToH [ACU08b] but it is not publicly available.

[3] Several clustering algorithms are described in Ref. [AK06], but the techniques are not integrated into the publicly available version of Metis.

[4] only 2-way FM

**Table 3.2:** Listing of different rating functions used in coarsening algorithms.

| Partitioner | Rating Function | Type | Matching | Clustering | $n$-Level |
|---|---|---|---|---|---|
| Metis, Jostle | $\text{weight}(u,v) := \omega(u,v)$ | GP | ● | ○ | ○ |
| KaSPar | $\text{expansion}(u,v) := \frac{\omega(u,v)}{c(u)c(v)}$ | GP | ○ | ○ | ● |
| KaFFPa | $\text{expansion}^2(u,v) := \frac{\omega(u,v)^2}{c(u)c(v)}$ | GP | ● | ○ | ○ |
| | $\text{innerOuter}(u,v) := \frac{\omega(u,v)}{\text{Out}(u)+\text{Out}(v)-2\omega u,v}$ with $\text{Out}(u) := \sum_{w\in\Gamma(u)} \omega(u,w)$ | GP | ● | ○ | ○ |
| PaToH | $\text{absorption}(u,\mathcal{C}) := \sum_{e\in \text{I}(u)\cap \text{I}(C)} \frac{\lvert e\cap\mathcal{C}\rvert\omega(e)}{\lvert e\rvert-1}$ | HGP | ○ | ● | ○ |
| hMetis | $\text{heavyEdge}(u,v) := \sum_{e\in \text{I}(u)\cap \text{I}(v)} \frac{\omega(e)}{\lvert e\rvert-1}$ | HGP | ○ | ● | ○ |
| KaHyPar | $\text{heavyEdge}(u,v)$ | HGP | ○ | ○ | ● |

algorithms with the difference that unmatched nodes can join clusters consisting of already matched nodes. In contrast to label propagation, nodes cannot change their cluster after joining it. The algorithm can be either implemented in a localized or global greedy fashion. The localized version visits the nodes in some order, and whenever a node $u$ is visited, it is added to a cluster $C \in \mathcal{C}$ that maximizes the rating $r(u, C)$. The global approach determines the node-cluster pair with the highest rating $r(u, C)$ in each step. KaHyPar [Sch+16a] uses a PQ for this and performs lazy updates of the ratings (since $r(u, C)$ can change if an adjacent node of $u$ joins cluster $C$). Akhremtsev et al. [Akh+17a] (for the connectivity metric) and Abou-Rjeili et al. [AK06] (for the edge cut metric) independently showed that the localized version is significantly faster than the global approach, and both produce partitions with comparable quality. The $n$-level scheme can also be classified as a hierarchical agglomerative clustering technique.

The previously presented coarsening algorithms perform contraction decisions based on the local neighborhood of a node. KaHyPar [HS17a] additionally uses community detection as a preprocessing step enhancing the coarsening process with global information about the community structure of the hypergraph. To do so, KaHyPar transforms the input hypergraph into its bipartite graph representation and uses the Louvain algorithm [Blo+08], maximizing the modularity objective function. It then restricts contractions to nodes within the same community in the coarsening phase and thus preserves some of the global structure.

**Initial Partitioning.** The coarsening phase usually proceeds until only $c \cdot k$ nodes remain where $c$ is a tuning parameter. Typical parameter choices for $c$ are between 10 and 200 [KK98c; Kar+99; SS10; Akh+17a]. Direct $k$-way partitioning algorithms often use multilevel recursive bipartitioning (RB) to obtain an initial $k$-way partition of the coarsest (hyper)graph [KK98c; ACU08b; Sch13; Akh+17a]. Heuer [Heu15a] showed

that this leads to partitions with significantly better solution quality than using flat direct $k$-way partitioning methods. Another approach is taken by Jostle [WC00a] that continues coarsening until $k$ nodes remain and uses it as an initial $k$-way partition. However, this may induce a partition violating the balance constraint. Therefore, Jostle [WC00a] uses balancing techniques in the uncoarsening phase ensuring that the final $k$-way partition is balanced. Other partitioning algorithms run *third-party* partitioners for initial partitioning [OS10; SS10] (e.g., an initial version of KaFFPa [SS10] used Scotch [PR96] to obtain an initial $k$-way partition but later replaced it with a multilevel RB implementation [Sch13]).

Algorithms computing an initial bipartition within the recursive bipartitioning scheme include the following techniques: random initial partitioning [Kar+99; VB05; ÇA11; Sch+16a; Sch20], bin packing techniques [ÇA11], (greedy) graph growing [KK98a; CA99; Kar+99; ÇA11; Sch13; Sch+16a; Sch20], label propagation-based initial partitioning [Sch+16a; Sch20], and spectral methods [HL95]. Partitioners often assemble several methods in a *portfolio* [KK98a; ÇA11; Sch+16a; Sch20], run each algorithm multiple times followed by FM refinement [HL95; Kar+99; VB05; Sch13; Sch+16a], and take the best as initial solution. hMetis [Kar+99] refines all initial solutions in the uncoarsening phase simultaneously and evicts the partitions worse than the best by more than 10% on each level.

Graph growing techniques grow one block of a bipartition starting from a random seed node. The next node added to the block can be chosen either via breadth-first-search or with greedy techniques using a gain function, e.g., FM gain [KK98a; CA99; ÇA11; Sch13; Sch20] or the weight of the nets connecting a node to the growing block [ÇA11; Sch20]. The algorithm stops when the weight of all touched nodes would exceed $\frac{c(V)}{2}$. All remaining untouched nodes are then assigned to the second block.

**Uncoarsening.** Section 3.1 and 3.2 gave an extensive overview of existing refinement algorithms. This paragraph discusses how these techniques are integrated into multilevel (hyper)graph partitioners.

Although it is known that RB can produce $k$-way partitions that are arbitrarily far away from an optimal solution [ST97], early multilevel partitioners mostly use the RB scheme due to the lack of efficient direct $k$-way FM formulations. Hendrickson and Leland [HL95] integrated a direct $k$-way FM algorithm into their multilevel partitioner Chaco using $k(k-1)$ PQs. However, Cong and Smith [CS93] showed that Chaco could not outperform partitioners based on RB. Multilevel RB algorithms primarily use boundary 2-way FM implementations [PR96; KK98a; CA99; Kar+99; VB05; Sch+16a] (PQs are initialized with all boundary nodes).

Metis [KK98c], hMetis [KK00] and PaToH [ACU08b] implement generalizations of their RB-based systems to direct $k$-way partitioning. The systems all use the size-constrained label propagation algorithm [MSS14] for refinement (referred to as greedy refinement). Although the label propagation algorithm cannot escape from local optima, the authors argue that the hill-climbing ability of the FM algorithm becomes less critical in the multilevel context since moves on coarser levels correspond to moves of entire clusters in the original (hyper)graph. Karypis and Kumar [KK98c]

evaluated a direct $k$-way FM implementation using a single PQ storing the gain of a node $u$ as $\omega(u, V \setminus \Pi[u]) - \omega(u, \Pi[u])$ (weight of external minus internal edges). The algorithm then extracts the node with the largest key from the PQ and moves it to the block that yields the largest edge cut reduction (which may increase the edge cut). However, they report that the algorithm could not produce better edge cuts than label propagation refinement, while it was also slower by a factor of two.

KaSPar [OS10], KaFFPa [SS11] and KaHyPar [Akh+17a] use highly-localized versions of the direct $k$-way FM algorithm. The local search starts from a small number of seed nodes and then gradually expands the search around them (for more details, see Section 3.1.2). KaFFPa uses a single PQ storing the highest gain move for each node. KaSPar and KaHyPar use $k$ PQs each representing one block storing moves to that particular block. KaFFPa [SS11] and KaHyPar [HSS19a; Got+20] also implement quotient graph style refinement techniques that applies 2-way local search algorithms on adjacent block pairs. Both partitioners use flow-based refinement (see Section 3.2) and KaFFPa additionally runs boundary 2-way FM refinement. Schlag [Sch20] showed on a large benchmark set with 488 hypergraphs (set $\mathrm{M_{HG}}$) that the direct $k$-way version of KaHyPar produces partitions with better solution quality (for the cut-net and connectivity metric) than its corresponding RB version (both use the same algorithmic components).

**A Short Comparison of Sequential Partitioning Algorithms.** We now present a short comparison of existing sequential partitioning algorithms illustrating the impact of the presented techniques on the solution quality in practice. Figure 3.6 compares KaFFPa-StrongS to Metis-K on set $\mathrm{M_G}$ (left), and $k$KaHyPar to PaToH-D on set $\mathrm{M_{HG}}$ (right). As a result of our experimental evaluation in Chapter 8, we identified KaFFPa-StrongS and $k$KaHyPar as the highest-quality sequential graph and hypergraph partitioners, while Metis-K and PaToH-D offer a good tradeoff between solution quality and running time.

KaFFPa-StrongS (label propagation coarsening) and $k$KaHyPar (based on the $n$-level partitioning scheme) implement clustering-based coarsening algorithms. Both use a highly-localized direct $k$-way FM implementation and flow-based refinement. Metis-K uses a matching-based coarsening algorithm and label propagation refinement, while PaToH-D uses hierarchical agglomerative clustering and runs boundary 2-way FM refinement in the uncoarsening phase. All partitioners are based on the direct $k$-way partitioning scheme except for PaToH-D, which is based on recursive bipartitioning.

As shown in Figure 3.6, the edge cuts produced by KaFFPa-StrongS are better than those of Metis-K by 16% in the median, while the partitions computed by $k$KaHyPar are better than those of PaToH-D (for connectivity optimization) by 13% in the median. We point out that differences by a few percentages are considered significant in the partitioning literature, as we will see in Chapter 8. Moreover, KaFFPa-StrongS and $k$KaHyPar compute better partitions than Metis-K and PaToH-D on more than 90% of the instances. However, Metis-K (geometric mean running time 0.39s) and PaToH-D (1.17s) are more than an order of magnitude faster than KaFFPa-StrongS (201.99s) and $k$KaHyPar (48.97s).

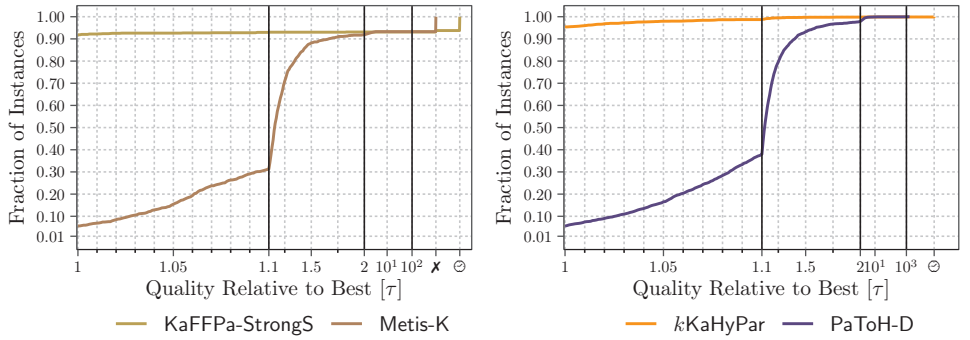



**Figure 3.6:** Performance profiles comparing KaFFPa-StrongS to Metis-K on set $M_G$ (left, cut-net optimization), and $k$KaHyPar to PaToH-D on set $M_{HG}$ (right, connectivity optimization).

The presented partitioners share many similarities, while KaFFPa-StrongS and $k$KaHyPar employ substantially stronger refinement techniques. These refinement algorithms lead to considerably better partitioning quality at the cost of increased execution times. Thus, parallelizing these techniques is essential for making high quality solutions on large instances feasible. However, we also want to point out that, unlike other partitioning systems, the codebases of KaFFPa (first publication in 2010) and KaHyPar (first commit in 2013) are actively maintained until today and each consists of about 40k lines of code. Therefore, it is a mistake to identify only the refinement algorithms as key components for their high quality. These systems combine techniques of more than a half-century of research in their particular field. However, the highly-localized $k$-way FM algorithm and flow-based refinement component are not used in any other partitioning algorithm.

## 3.4 Parallel (Hyper)Graph Partitioning

In the previous section, we discussed the core components of state-of-the-art sequential partitioning algorithms. We now turn to parallel techniques focusing on how the sequential algorithms are parallelized and used in modern parallel partitioning algorithms. We start with a discussion of the main parallelization challenges and then review the algorithmic components of existing parallel (hyper)graph partitioners.

**Distributed-Memory Machine Model.** In Section 2.3, we already presented the shared-memory machine model but omitted the distributed-memory model since it is only relevant in this section. We therefore briefly explain it in the following. In this model, several processors (PEs) are interconnected via a communication network, and each with its private memory inaccessible to others. Computational tasks on each PE operate independently only on local data usually representing a small subset of the

input. Intermediate computational results must be exchanged via dedicated network communication primitives.

Distributed (hyper)graph processing algorithms require that the nodes and (hyper)edges of the input are partitioned among the processors. Since many applications use balanced (hyper)graph partitioning to obtain a good initial assignment, much simpler techniques are used in distributed partitioning algorithms. There exist range-based [MSS17] and hash-based partitioning techniques [Slo+17; Slo+20]. The former split the node IDs into equidistant ranges, which are then assigned to the PEs. If geometric information is available, one can also use space-filling curves [ABM16; LTM18].

Each PE then stores the nodes assigned to it and the edges incident to them. The edges stored on a PE can be incident to local nodes or nodes stored on other PEs (also called *ghost* or *halo* nodes). We say that a node respectively PE is adjacent to another PE if they share a common edge. If we move a node to a different block, we have to propagate that change to adjacent PEs such that local search algorithms can work on accurate partition information. However, each communication operation introduces overheads that can limit the scalability of the system. Thus, the main challenge in distributed (hyper)graph partitioning is keeping the global partition information on each PE in some sense up to date while simultaneously minimizing the required communication.

### 3.4.1 Parallelization Challenges

Parallel refinement algorithms should allow several nodes to change their block simultaneously. This poses multiple challenges for parallel local search algorithms, which we discuss in this section. Furthermore, we explain how these problems are addressed in practice and why most of the refinement algorithms presented in the previous sections are considered difficult to parallelize.

**Move Conflicts.** Iterative improvement algorithms move nodes to other blocks according to a gain value. In the sequential setting, the gain $g_u(V_j)$ of moving a node $u$ to block $V_j$ accurately reflects the change in the objective function since only one node can change its block at a time. This does not hold in the parallel setting since concurrent moves of adjacent nodes can adversely affect each others gain. Figure 3.7 illustrates different types of concurrent *move conflicts* for the edge cut metric. The case where two adjacent nodes $u \in V_i$ and $v \in V_j$ with $i \neq j$, simultaneously move to different blocks can degrade the edge cut, even if both individual gains suggest an improvement (conflict type A and B). If, on the other hand, $u$ and $v$ are in the same block, the actual gain may even be better than suggested by the two individual gains (conflict type C). In the following, we discuss different solutions to prevent conflicts of type A and B.

Karypis and Kumar [KK96] use a node coloring of the graph (labeling of the nodes where adjacent nodes have different labels) and then split a refinement pass into sub-rounds where only nodes with the same color can change their block. Thus,

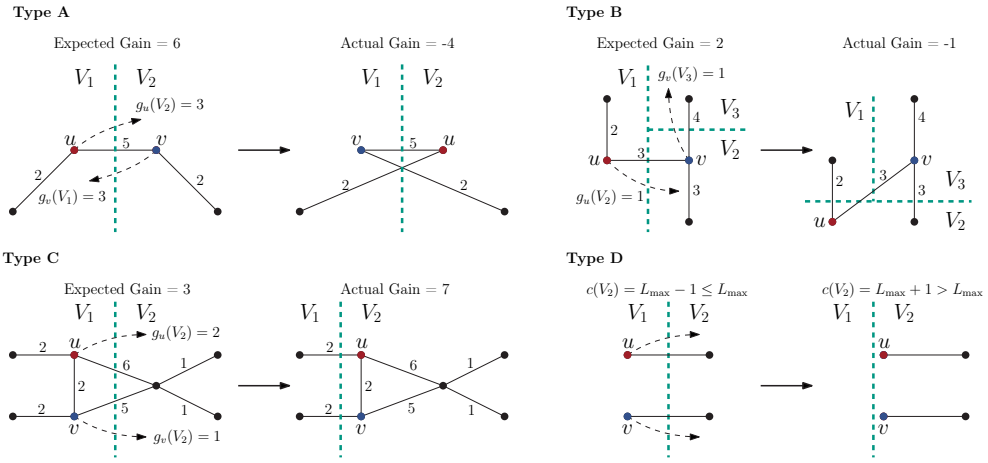



**Figure 3.7:** Illustration of different types of move conflicts for the cut-net metric that occur if two adjacent nodes change their block simultaneously (Type A, B and C). The conflict type D illustrates concurrent node moves that violate the balance constraint. Type A and B are taken from Ref. [KK96].

two adjacent nodes cannot change their block at the same time. Another approach computes an edge coloring of the quotient graph (labeling of the edges where incident edges have different labels) and then schedule sequential 2-way refinement algorithms on adjacent block pairs with the same color in parallel [Din+95; WC00b; HSS10]. Note that this restricts the available parallelism to at most $\frac{k}{2}$ processors/threads. A third technique is to split a refinement pass into two phases: an upward and downward phase [TK04a; LK13]. In the upward phase, a node can move from its current block $V_i$ to a target block $V_j$ if $i < j$ (vice versa in the downward phase). However, this scheme restricts the set of possible moves and only protects against conflicts of type A.

Walshaw et al. [WCE97] redefine the gain of moving a node $u \in V_i$ to a target block $V_j$ as $g_u(V_j) - \sum_{v \in \Gamma(u)} g_v(V_i)/|\Gamma(u)|$ (referred to as *relative gain*). More informally, the relative gain is the gain of $u$ minus the average gain of $u$'s neighbors. The gain function prefers node moves with a high gain value adjacent to nodes with a low gain value, which are good candidates to avoid collisions.

We note that many publications do not address the aforementioned types of conflicts. This can be seen as an optimistic strategy assuming that conflicts rarely happen in practice and are outweighed by node moves improving the objective function.

**Balance Constraint.** In the sequential setting, balance can be enforced when starting from a feasible solution by only applying node moves that do not violate the balance constraint. In the parallel setting, however, different processors can move several nodes to the same block at the same time. For each processor, it may appear that the resulting partition is balanced, but the combination of all concurrent node moves may result in an imbalanced solution (see conflict type D in Figure 3.7).

Shared-memory algorithms can use atomic instructions to modify the block weights [ASS17; Got+21e]. If a thread wants to move a node $u$ from its current block $V_i$ to $V_j$, it can use atomic `fetch-and-add` instructions to update $c(V_i)$ and $c(V_j)$. If the resulting block weight of $c(V_j)$ is smaller than or equal to $L_{\max}$, the thread can safely perform the move. Otherwise, it has to revert the block weight update of $c(V_i)$ and $c(V_j)$ and discard the move.

Maintaining the balance of a $k$-way partition is more complicated in the distributed-memory setting. Here, refinement algorithms often follow the *bulk synchronous parallel* model. In a computation phase, each processor moves its local nodes and updates the block weights locally. In the communication phase, updates are made visible to other PEs via a personalized `All-To-All` operation and the exact block weights are restored with an `AllReduce` operation [KK96; LK13; MSS17]. Note that this does not guarantee balance. However, frequent periodic updates adequately estimate the exact block weights in the computation phase. Some parallel partitioners additionally revert moves to restore balance [LK13]. Another approach is to send all node moves to a dedicated master processor, which selects a subset of them satisfying the balance constraint [TK04a; Dev+06; ASS17]. Slota et al. [SMR16; Slo+17; Slo+20] implement a scheme that alternates between refinement and balancing phases. In the balancing phase, the algorithm penalizes moves to overloaded blocks by incorporating a penalty term into the gain value of a node move. Consequently, moves to underloaded blocks become more attractive.

Recently, probabilistic methods were proposed that preserve the balance in expectation [Kab+17; Mar+17]. Kabiljo et al. [Kab+17] aggregates the number of nodes $S_{i,j}$ that want to move from block $V_i$ to $V_j$ after each computation phase at a dedicated master processor. Then, a node part of block $V_i$ is moved to its desired target block $V_j$ with probability $\frac{\min(S_{i,j},S_{j,i})}{S_{i,j}}$. This ensures that the expected number of nodes that move from block $V_i$ to $V_j$ and vice versa is the same and thus preserves the balance of the partition in expectation.

Many of the previously presented techniques do not guarantee balance. Therefore, some parallel partitioning algorithms use balancing algorithms restoring the balance by moving nodes out of overloaded blocks [WCE97; Got+21e; Mal+21].

**P-Completeness.** The complexity class `NC` describes the problems solvable in polylogarithmic time with a polynomial number of processors [Pip79]. Similar to the question $\mathtt{P} \stackrel{?}{=} \mathtt{NP}$, it is unknown whether $\mathtt{NC} \subset \mathtt{P}$ or $\mathtt{NC} = \mathtt{P}$. The former would imply that there exist *inherently sequential* problems [GHR+95]. A problem is `P`-hard if every problem in `P` can be reduced to it in polylogarithmic time with a polynomial number of processors. A problem is `P`-complete if it is `P`-hard and also in `P`. The existence of an efficient parallel algorithm for a `P`-complete problem would imply that $\mathtt{NC} = \mathtt{P}$. Thus, `P`-complete problems are considered difficult to parallelize efficiently.

Savage and Wloka [SW91] showed that the Kernighan-Lin and FM heuristic are `P`-complete. Furthermore, Goldschlager et al. [GSS82] proved that the maximum flow problem is `P`-complete. This means that most of the refinement algorithms presented in the previous sections are difficult to parallelize efficiently.

### 3.4.2 Algorithmic Components of Parallel Partitioners

Many authors of sequential partitioning algorithms have presented parallel formulations of their systems, e.g., ParJostle [WC00b] (parallel counterpart to Jostle), ParMetis [KK96] and Mt-Metis [LK13; LaS+15; LK16] (parallel counterpart to Metis), and ParHIP [MSS17] and Mt-KaHIP [ASS17] (parallel counterpart to KaFFPa). These algorithms translate the core ideas of their sequential counterparts into the parallel setting while more or less addressing the challenges discussed in the previous section. Historically, research focused on distributed-memory algorithms due to the lack of powerful multi-core machines at the time. This changed drastically over the last decade, which is why shared-memory partitioning algorithms have gained interest in recent years. The shared-memory model is usually more attractive because it offers more algorithmic options as changes on the global partition are immediately visible to all PEs without expensive network communication.

This section reviews the building blocks of parallel partitioning systems with a particular focus on multilevel algorithms. We are mainly interested to which extent the high quality techniques of sequential algorithms are used in parallel partitioners. This in turn is the foundation of our research. We structure the following discussion alongside the different phases of the multilevel scheme. The identified algorithmic components of parallel partitioning algorithms are listed in Table 3.3.

**Coarsening.** Parallel coarsening algorithms include matching-based approaches [KK96; Dev+06; CP08; Çat+12a; KR13; LK13; LaS+15; Akh19], hierarchical agglomerative clustering [TK04a; Çat+12a; Mal+21], and size-constrained label propagation [ASS17; Got+21e]. We structure the following discussion into techniques used in shared- and distributed-memory partitioners.

**Shared-Memory.** Mt-Metis [LK13], Mt-KaHIP [Akh19] and PaToH[5] [Çat+12a] implement all similar parallel matching-based coarsening algorithms based on unprotected writes to a shared matching vector $M$ of size $n$. The algorithm iterates in parallel over the nodes in some order, and whenever a node $u$ is visited, it computes a matching partner $v$ according to a rating function and sets $M[u] = v$ and $M[v] = u$. In a second parallel pass over the nodes, conflicts are resolved by setting $M[u] = u$ if $M[M[u]] \neq M[u]$. Mt-KaHIP [Akh19] sets $M[u] = v$ and matches two nodes $u$ and $v$ in a second pass if $M[u] = v$ and $M[v] = u$. The algorithm then continues with the remaining unmatched nodes. Mt-Metis [LK13] and PaToH [Çat+12a] also present a version that protects writes to $M$ by locking the two matched nodes. Lasalle and Karypis [LK13] report that the approach based on unprotected writes scales slightly better, while Catalyürek et al. [Çat+12a] observed the opposite. Catalyürek et al. [Çat+12a] noted that the number of conflicts increases with the number of threads, but the percentage of conflicts was smaller than 0.7% on all tested instances. Mt-Metis [LaS+15] computes a matching, and if this does not sufficiently reduce the

[5] PaToH implements a parallel version of its coarsening algorithm but no parallel initial partitioning and refinement. Therefore, we did not include it in Table 3.3.

**Table 3.3:** Algorithmic components of existing parallel partitioning algorithms.

| | | ParJostle | ParMetis | PT-Scotch | KaPPa | ParHIP | Mt-Metis | Mt-KaHIP | KaMinPar | Parkway | Zoltan | BiPart |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Design | Type | GP | GP | GP | GP | GP | GP | GP | GP | HGP | HGP | HGP |
| | Distributed-Memory | ● | ● | ● | ● | ● | ○ | ○ | ○ | ● | ● | ○ |
| | Shared-Memory | ○ | ○ | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ● |
| | Direct $k$-Way | ● | ○ | ○ | ● | ● | ● | ● | ○ | ● | ○ | ○ |
| | Recursive Bipartitioning | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● |
| | Deep Multilevel | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ |
| Coarsening | Community Detection | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| | $n$-Level | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| | Matching | ● | ● | ● | ● | ○ | ● | ● | ○ | ○ | ● | ○ |
| | Clustering | ○ | ○ | ○ | ○ | ● | ○ | ● | ● | ● | ○ | ● |
| IP | Direct $k$-Way | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| | Recursive Bipartitioning | ○ | ● | ○ | ○ | ○ | ● | ○ | ○ | ● | ○ | ○ |
| | Portfolio | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ |
| | Evolutionary | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ |
| | Sequential | ● | ● | ○ | ● | ○ | ○ | ● | ○ | ● | ● | ○ |
| Refinement | Label Propagation | ● | ● | ○ | ○ | ● | ○ | ● | ● | ● | ? | ● |
| | Band Refinement | ● | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| | Boundary FM | ○ | ○ | ○ | ○ | ○ | ◐[6] | ○ | ○ | ○ | ? | ○ |
| | Localized FM | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ | ○ |
| | Flows | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| Move Conflicts | Node Coloring | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| | Sequential 2-way FM | ● | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| | Up- and Downward Phase | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ● | ● | ○ |
| | Relative Gain | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| | Optimistic Strategy | ○ | ○ | ○ | ○ | ● | ○ | ● | ● | ○ | ○ | ● |
| Balance Constraint | Bulk Synchronous | ○ | ● | ○ | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ |
| | Master Process | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ● | ● | ○ |
| | Balancing Algorithm | ● | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ● |
| | Probabilistic Methods | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| | Atomic Instructions | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ |
| | Open Source | ○ | ○ | ○ | ○ | ● | ○ | ● | ● | ● | ○ | ● |
| | Publicly Available | ○ | ● | ● | ○ | ● | ● | ● | ● | ● | ● | ● |

size of the graph, it matches unmatched nodes that have a common neighbor (also known as 2-hop matching). Preferably unmatched nodes of degree one are matched. Afterwards, the algorithm considers unmatched nodes with the same neighbors (also called *twins*). Last, the algorithm matches the remaining unmatched nodes with at least one common neighbor.

PaToH [Çat+12a] also implements a parallel version of the hierarchical agglomerative

[6] performs only positive gain moves.

clustering scheme. The algorithm maintains the representatives of each cluster in an array rep of size $n$. If $\mathsf{rep}[u] = \perp$ then $u$ is considered as *unclustered*. If $\mathsf{rep}[u] = v$ then $u$ is *clustered* and $v$ is the representative of $u$'s cluster. The algorithm iterates in parallel over the nodes in some order, and whenever a node $u$ is visited, it is locked and checked if it is still unclustered. If $u$ is unclustered, the algorithm computes the rating $r(u, v) = \sum_{e \in I(u) \cap I(v)} \omega(e)$ for each neighbor $v \in \Gamma(u)$. Afterwards, it aggregates the ratings of the representatives of each cluster, i.e., $r(u, w) = \sum_{v \in \Gamma(v) \wedge \mathsf{rep}[v] = w} r(u, v)$. The algorithm then iterates over the aggregated ratings, and whenever a representative $w$ is found with a higher rating than the currently best, it is locked and checked if $c(u) + c(w)$ is smaller than a predefined size-constrained $U$. If fulfilled, $w$ is accepted as the new best representative for $u$. Otherwise, it releases the lock for $w$. In the end, the algorithm sets $\mathsf{rep}[u] = w$, updates the cluster weight of $c(w)$ to $c(u) + c(w)$, and releases the locks of $u$ and $w$.

Mt-KaHIP [ASS17] and KaMinPar [Got+21e] use clustering scheme based on parallel size-constrained label propagation. The algorithm can be parallelized straightforwardly: Iterate over the nodes in some order in parallel, and compute for each node $u$ the label to which $u$ has the strongest connection. Note that during the evaluation of the ratings, the labels of neighbors may change. However, this happens rarely in practice and therefore is acceptable [SM16] (often beneficial since it introduces random noise). The size-constrained on the weight of the heaviest cluster can be maintained by updating the cluster sizes via atomic `fetch-and-add` instructions.

**Distributed-Memory.** ParMetis [KK96], PT-Scotch [CP08], KaPPa [HSS10], ParJostle [WCE97; WC00b] and Zoltan [Dev+06] use parallel matching-based coarsening algorithms. In the distributed-memory setting, each processor iterates over its local nodes, and for each node $u$, it computes a matching partner $v$ according to a rating function. If $v$ is a local node, then $u$ and $v$ are matched immediately. If $v$ is a remote node, then a matching request is written into a send buffer for the corresponding PE [KK96; WCE97; CP08]. The send buffers are exchanged via a personalized `All-To-All` communication at the end of the computation phase. The matching requests are then accepted or rejected (e.g., accepted if $v$ chooses $u$ as a matching partner). ParMetis [KK96] uses a node coloring of the graph and considers nodes with the same color in each round. This scheme ensures that the matching partners of each node remain unmatched in the current round. If a node receives multiple matching requests in the communication phase (also done via an `All-To-All` operation), it chooses the node connected via the heaviest edge.

Parkway [TK04a] implements a parallel version of the hierarchical agglomerative clustering scheme. Each processor iterates over its local nodes, and for each node $u$, it computes the node $v$ with the strongest connection. If $v$ is a local node, then $u$ is added to $v$'s cluster. If $v$ is a remote node, then a matching request is written into a send buffer. The send buffers are then exchanged via a personalized `All-To-All` communication at the end of the computation phase. If $v$ is unmatched, matched locally or already matched remotely, the receiving processor grants the matching

request. If $v$ is currently also in a remote matching process, then the matching request of node $u$ is rejected.

ParHIP [MSS17] uses the size-constrained label propagation algorithm to construct a clustering. Each processor iterates over its local nodes and assigns them to the cluster with the strongest connection. Cluster updates to and from adjacent PEs are sent and received asynchronously. The size-constrained $U$ that bounds the weight of the heaviest cluster is only ensured locally. The authors argue that in the coarsening phase, the size-constrained is relatively soft, and maintaining exact cluster sizes would incur to much communication overhead.

**Initial Partitioning.** A widely-used technique to compute an initial partition in parallel partitioning algorithms replicates the coarsest (hyper)graph on each processor/thread and then runs a sequential partitioning algorithm on each copy [KK96; WC00b; TK04a; TK04b; Dev+06; HSS10; ASS17]. The best initial partition from all independent runs is projected onto the coarsest (hyper)graph. Another approach is based on parallel multilevel recursive bipartitioning [CP08; LaS+15]. The (hyper)graph is partitioned into two blocks using a parallel multilevel bipartitioning algorithm. Afterwards, the scheme splits the processors/threads into two disjoint groups that independently perform the two recursive partitioning calls. ParHIP [MSS17] partitions the coarsest graph using the distributed evolutionary partitioner KaFFPaE [SS12].

PT-Scotch [CP08] introduces the *fold-dup* technique that divides the processors into two disjoint subgroups each time the size of the original graph is halved in the coarsening phase. The two subgroups then recursively continue the coarsening process on two identical copies of the graph. If only one PE remains, it resorts to the sequential version of PT-Scotch [PR96]. The algorithm then uses the partition with the better edge cut from the two recursive partitioning calls to continue the uncoarsening process. The fold-dup technique is integrated into a parallel multilevel recursive bipartitioning scheme. Gottesbüren et al. [Got+21e] introduced the *deep multilevel scheme* generalizing the fold-dup technique to direct $k$-way partitioning, which is used in the shared-memory graph partitioner KaMinPar.

**Deep Multilevel Partitioning.** KaMinPar [Got+21e] is designed for partitioning graphs into a large number of blocks (e.g., $k \in \mathcal{O}(\sqrt{n})$). Here, the assumption that the coarsest graph is small does not hold since coarsening algorithms in the direct $k$-way setting terminate if the number of nodes drop below $c \cdot k$ where $c$ is a tuning parameter. Moreover, the running time of algorithms based on recursive bipartitioning is $\mathcal{O}(n \log k)$ assuming that the bipartitioning routine takes $\mathcal{O}(n)$ time. Thus, using recursive bipartitioning or sequential algorithms, can become a bottleneck for partitioning (hyper)graphs into a large number of blocks. Therefore, Gottesbüren et al. [Got+21e] introduced the deep multilevel scheme illustrated in Figure 3.8.

In the following, we assume that both the number of processors $t$ and the number of blocks $k$ are powers of two (we refer the reader to Ref. [Got+21e] for the general case). The algorithm continues parallel coarsening until the number of nodes equals $ct$ nodes. Then, similar to the fold-dup technique, the processors are split into two

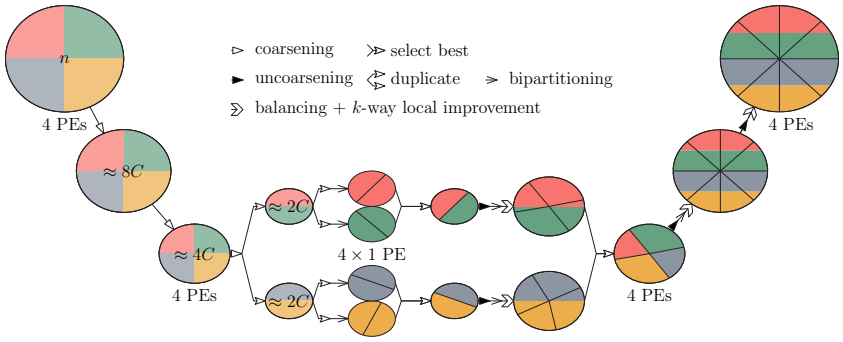



**Figure 3.8:** Example of the deep multilevel scheme that partitions a (hyper)graph into 8 blocks using 4 processors. The colors represent processors indicating on which copy of the (hyper)graph they work on. Figure is taken from Ref. [Got+21e].

disjoint groups and recursively continue parallel coarsening on two identical copies of the graph. On the coarsest level, one PE is responsible for bipartitioning a graph with roughly $2c$ nodes. During uncoarsening, the algorithm selects the best partition of the two recursive partitioning calls. Then, if the current number of blocks is still smaller than the desired number of blocks $k$, it further bipartitions each block independently in parallel. The scheme combines recursive bipartitioning and direct $k$-way partitioning techniques by recursively bipartitioning blocks and applying direct $k$-way local search algorithms in the uncoarsening phase.

Note that the actual implementation does not split the PEs into subgroups for processing the recursive calls. Instead, it uses task-based parallelism and a work-stealing approach.

**Uncoarsening.** In contrast to sequential partitioning algorithms, most of the parallel systems are based on direct $k$-way partitioning due to the lack of efficient and scalable 2-way FM implementations. The most widely-used parallel refinement technique is the label propagation algorithm [KK96; WCE97; TK04a; ASS17; MSS17; Got+21e; Mal+21]. The algorithm works similarly to what we described in the paragraph on parallel coarsening. Therefore, we omit a detailed description here.

KaPPa [HSS10] and PT-Scotch [CP08] implement a technique that they call *band refinement*. The *band graph* of a bipartition is a subgraph containing nodes within a certain distance of the cut. Band refinement extracts a band graph and broadcasts it to several PEs, which then perform sequential 2-way FM refinement. The best bipartition from all independent runs is projected onto the original graph. ParJostle [WC00b] implements the *interface optimization* technique. The algorithm defines the *interface* region $I_{ij}$ consisting of the nodes that prefer to move from block $V_i$ to $V_j$ and vice versa. For each interface region, one PE is responsible for performing sequential 2-way FM refinement (one PE may be responsible for several interface regions). The shared-memory graph partitioner Mt-Metis [LK13] parallelizes the greedy refinement

technique of Metis-K [KK98c]. Each thread inserts nodes into a thread-local PQ and repeatedly performs the highest positive gain move.

All presented parallel refinement techniques up to this point restrict the set of possible moves by either using sequential 2-way FM refinement on adjacent block pairs or performing only positive gain moves (limited ability to escape from local optima). Therefore, Mt-Metis [LK16] proposes the parallel hill-scanning algorithm extending its parallel greedy refinement algorithm [LK13]. If the next move from the thread-local PQ has negative gain, the algorithm attempts to find additional negative gain moves around this move, which yield overall positive gain if performed together. This technique is similar to the localized $k$-way FM algorithm of KaFFPa [SS11] presented in Section 3.1.2. However, it applies a set of moves to the partition as soon as it yields positive combined gain (instead of reverting to the best seen solution). Furthermore, the size of a hill is restricted to at most 16 nodes.

**Parallel Multi-Try $k$-Way FM.** Mt-KaHIP [ASS17] implements a shared-memory version of the multi-try $k$-way FM algorithm (see Section 3.1.2). The threads perform highly-localized FM searches that do not overlap on nodes. Each thread initializes its search with a different boundary node, and gradually expands around it by claiming neighbors of moved nodes. It performs node moves locally using a thread-local hash table to update block IDs of nodes and an array of size $k$ to maintain block weights. When all searches have terminated, the move sequences of each thread are concatenated, for which gains are recomputed sequentially. The prefix with the highest aggregated gain value of that combined sequence is then applied.

**A Short Comparison of Sequential and Parallel Partitioning Algorithms.** In our experimental evaluation in Chapter 8, we identified Mt-KaHIP (shared-memory) and Zoltan (distributed-memory) as the best parallel graph and hypergraph partitioners. We now compare them to KaFFPa-StrongS and $k$KaHyPar on set $\mathrm{M_G}$ and $\mathrm{M_{HG}}$ to demonstrate how much the solution quality of sequential and parallel partitioning algorithms differ at the moment.

As can be seen in Figure 3.9 (left), the solution quality of the partitions computed by KaFFPa-StrongS are better than those of Mt-KaHIP by 7.5% in the median. On all instances where KaFFPa-StrongS completed in the given time limit (1121), there are only 12 instances on which Mt-KaHIP produces partitions with a better edge cut than KaFFPa-StrongS. However, Mt-KaHIP produces better partitions than Metis-K on average (median improvement of KaFFPa-StrongS over Metis-K is 16%, see page 47). Furthermore, Mt-KaHIP (geometric mean running time 0.95s) is more than two orders of magnitude faster than KaFFPa-StrongS (201.99s) with only ten threads. This is a promising result since Mt-KaHIP is the first parallel partitioner that implements a parallel version of the multi-try $k$-way FM algorithm, which we have identified as an essential technique for achieving high solution quality.

For hypergraph partitioning (see right side of Figure 3.9), the partitions computed by $k$KaHyPar are better than those of Zoltan by 23% in the median. However, Zoltan (0.55s) is almost two orders of magnitude faster than $k$KaHyPar (48.97s) with ten threads.

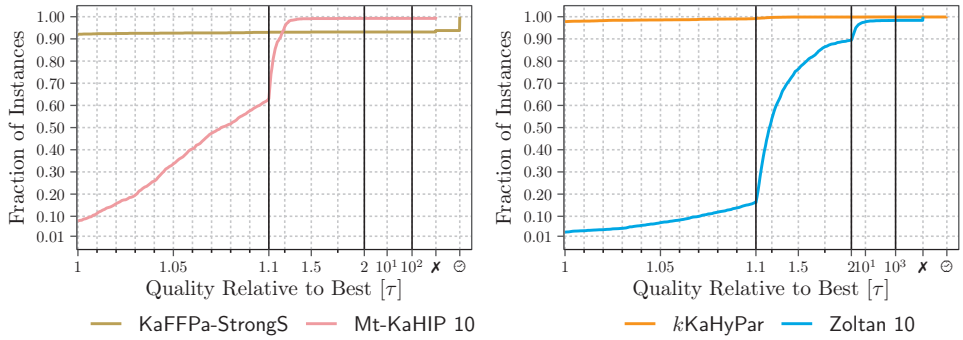



**Figure 3.9:** Performance profiles comparing Mt-KaHIP to KaFFPa-StrongS on set $M_G$ (left, cut-net optimization), and Zoltan to $k$KaHyPar on set $M_{HG}$ (right, connectivity optimization).

Zoltan implements the recursive bipartitioning scheme and uses a two-dimensional distribution of the input matrix (rows represent nodes and columns represent the hyperedges). Hence, "each processor knows only partial information about some nodes and some hyperedges" [Dev+06]. During refinement, the PEs associated with a range of nodes collaborate on computing their initial gain values. Subsequently, one PE of each distributed column is responsible for performing the node moves and updating the gain values based on its partial information of the hypergraph. This may lead to incorrect gain values and adversely affects the solution quality with an increasing number of PEs (also mentioned by Devine et al. [Dev+06]).

In the previous section, we have seen that using stronger refinement algorithms (e.g., highly-localized direct $k$-way FM search and flow-based refinement) leads to substantially better solution quality at the cost of an increased running time. Most of the existing parallel partitioning algorithms use comparatively weak components that are easier to parallelize (e.g., label propagation or 2-way FM refinement on adjacent block pairs). Both, the hill-scanning algorithm of Mt-Metis [LK16] and the multi-try $k$-way FM algorithm of Mt-KaHIP [ASS17] can be seen as the first steps in transferring higher quality techniques to the shared-memory setting. However, these systems are still inferior compared to the best sequential algorithms.

Chapter 4

# 4 Parallel Improvement Algorithms

In the previous chapter, we have seen that existing parallel (hyper)graph partitioners do not achieve the same solution quality as their sequential counterparts. One of the main reasons is that parallel systems use comparatively weak components such as the label propagation algorithm [KK96; WCE97; TK04a; ASS17; MSS17; Got+21e; Mal+21] or sequential 2-way FM refinement on adjacent block pairs [WC00b; HSS10]. The former cannot escape from local optima, and the latter unnecessarily restricts the set of possible moves. In contrast, the highest-quality sequential partitioners implement a diverse set of improvement heuristics such as the highly-localized $k$-way FM algorithm [SS11; Akh+17a] and flow-based refinement [SS11; HSS19a; Got+20].

Recently, Akhremtsev et al. [ASS17] presented the first parallel implementation of the multi-try $k$-way FM algorithm. Although the solution quality of their partitioner is still not competitive with the best sequential codes, it produces significantly better edge cuts than previous parallel systems and achieves good speedups. The results demonstrate that advanced local search heuristics can be parallelized efficiently and can be seen as a first step towards closing the quality gap. We therefore continue this line of work and propose parallel implementations of the most advanced improvement algorithms. Ultimately, this work aims to implement a parallel system that achieves the same solution quality as the highest-quality sequential partitioners.

In contrast to most publications, we start with parallel refinement algorithms instead of explaining them alongside the multilevel paradigm. There are two main reasons for this. First, the following parallel algorithms are one of the main contributions of this work. Second, and more importantly, we present two different multilevel partitioners that share large parts of our codebase. The main difference between both multilevel algorithms is how (un)contractions are performed, while the refinement algorithms presented in this chapter are reused with only minor modifications.

**Outline.** The previous chapter identified the label propagation, the multi-try $k$-way FM algorithm [SS11] and flow-based refinement [SS11; HSS19a; Got+20] as essential techniques for achieving high solution quality. We propose parallel formulations of these algorithms in Section 4.2 – 4.4. Our multilevel partitioners run these algorithms on each level. We use the label propagation algorithm to quickly converge to a local optimum such that the two more advanced techniques can focus on finding non-trivial improvements [ASS17]. However, all algorithms rely on up-to-date partition information which is inherently difficult in the parallel setting. We therefore present multiple techniques for parallel gain (re)computation based on our new partition data structure in Section 4.1.

**References and Contributors.** This chapter covers the gain computation techniques and parallel refinement algorithms presented in Refs. [Got+21a; Got+21c; GHS22a; GHS22c; Got+22a]. The text has been largely rewritten to provide a consistent level of detail throughout this work, except for large parts of Section 4.4 (flow-based refinement) that contain most of the text verbatim from Ref. [GHS22a]. We further improved the presentation with additional pseudocodes and illustrations.

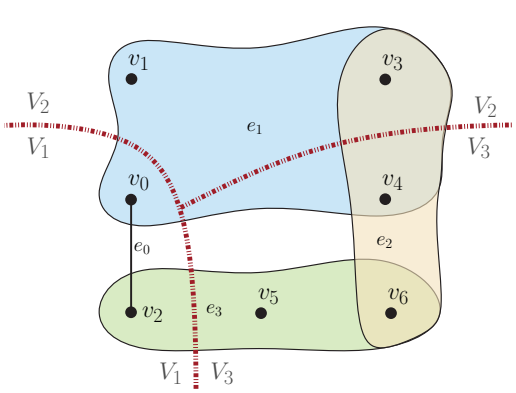
The author of this dissertation implemented the partition data structure and the attributed gain value technique presented in Section 4.1.1. The concurrent gain table and parallel gain recomputation algorithm discussed in Sections 4.1.2 and 4.1.3 was developed by Lars Gottesbüren. The author of this dissertation further improved the gain table for $n$-level partitioning [Got+22a]. We would also like to thank Michael Hamann, who was involved in the initial idea and discussion for the gain table. The parallel label propagation algorithm (see Section 4.2), and the parallel scheduling scheme and flow network construction algorithm for flow-based refinement (see Sections 4.4.1 and 4.4.2) came from the author of this dissertation. Lars Gottesbüren worked on the parallel multi-try $k$-way FM algorithm (see Section 4.3) and the parallelization of the FlowCutter algorithm (see Sections 4.4.3– 4.4.6). Moreover, both authors were involved in the performance engineering process of all components.

## 4.1 Parallel Gain Calculation

Because nodes can move simultaneously, it is inherently difficult for parallel partitioners to compute exact gain values. For example, the gain of a node move can change between its initial calculation and the actual execution. Consequently, two concurrent node moves can worsen the solution quality, even if each individual gain suggested an improvement as explained in Section 3.4.1. Our refinement algorithms have different requirements for calculating and verifying gains, depending on whether nodes are moved immediately as they are explored (label propagation) or not (FM and flow-based refinement). In the following, we describe a technique named *attributed gains* to track the overall improvement and double-check the gain of a move (detects move conflicts with neighbors and is used in all refinement algorithms), a parallel *gain table* to accelerate gain calculations, and a novel parallel algorithm for *recomputing exact gains* of a move sequence (both used in the FM algorithm). These techniques build on our concurrent partition data structure which we will describe in the next section in more detail.

### 4.1.1 The Partition Data Structure

Our partition data structure stores and maintains the partition assignments $\Pi$, the block weights $c(V_i)$, the pin count values $\Phi(e, V_i)$, and connectivity sets $\Lambda(e)$ for each net $e \in E$ and block $V_i \in \Pi$.




**Partition Assignments**

| $\Pi$ | 1 | 2 | 1 | 2 | 3 | 3 | 3 |
|---|---|---|---|---|---|---|---|
| | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ |

**Block Weights**

| $c(V_i)$ | 2 | 2 | 3 |
|---|---|---|---|
| | $V_1$ | $V_2$ | $V_3$ |

**Connectivity Sets**

| | $e_0$ | | | $e_1$ | | | $e_2$ | | | $e_3$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\Lambda(e)$ | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| | $V_1$ | $V_2$ | $V_3$ | $V_1$ | $V_2$ | $V_3$ | $V_1$ | $V_2$ | $V_3$ | $V_1$ | $V_2$ | $V_3$ |

**Pin Count Values**

| | $\Phi(e, V_1)$ | $\Phi(e, V_2)$ | $\Phi(e, V_3)$ |
|---|---|---|---|
| $e_0$ | 2 | 0 | 0 |
| $e_1$ | 1 | 2 | 1 |
| $e_2$ | 0 | 1 | 2 |
| $e_3$ | 1 | 0 | 2 |

**Figure 4.1:** The partition data structure.

Figure 4.1 shows our partition data structure for a 3-way partition $\Pi = \{\{v_0, v_2\}, \{v_1, v_3\}, \{v_4, v_5, v_6\}\}$. Hyperedge $e_1 = \{v_0, v_1, v_3, v_4\}$ has $\Phi(e_1, V_1) = 1$ pin in block $V_1$, $\Phi(e_1, V_2) = 2$ pins in block $V_2$ and $\Phi(e_1, V_3) = 1$ pin in block $V_3$. The connectivity set of hyperedge $e_3 = \{v_2, v_5, v_6\}$ is $\Lambda(e_3) = \{V_1, V_3\}$, since $v_2 \in V_1$ and $\{v_5, v_6\} \subset V_3$.

**Data Layout.** The size of a pin count value is bounded by the size of the largest hyperedge. To save memory, we use a packed representation with $\lceil \log(\max_{e \in E} |e|) \rceil$ bits per entry for the $\Phi(e, V_i)$ values. Furthermore, we use a bitset of size $k$ to store the connectivity set $\Lambda(e)$ of each hyperedge $e \in E$. We iterate over the connectivity set $\Lambda(e)$ by taking a snapshot of its bitset and then use *count-leading-zeroes* instructions. We compute the connectivity $\lambda(e) = |\Lambda(e)|$ of a hyperedge $e$ using *pop-count* instructions (counts the number of 1-bits in a machine word).

**The Move Node Operation.** Algorithm 4.1 shows the update operation performed on the partition data structure when moving a node $u$ from its current block $V_i$ to a target block $V_j$. It starts with an atomic `fetch-and-add` instruction adding the weight of node $u$ to the weight of block $V_j$. If the resulting partition is imbalanced, we revert the block weight update of $c(V_j)$ and reject the move (see Line 16). Otherwise, we move node $u$ to block $V_j$ and subtract the weight of node $u$ from the weight of its current block $V_i$. We do not perform an atomic operation to update the block ID of node $u$ since our refinement algorithms guarantee that only one thread moves a node at a time.

We then update the pin count values $\Phi(e, V_i)$ and $\Phi(e, V_j)$ of each net $e \in \mathrm{I}(u)$ using atomic `fetch-and-sub/add` instructions. Subsequently, we modify the connectivity set $\Lambda(e)$ by flipping a bit in the corresponding bitset if either $\Phi(e, V_i)$ decreases to zero or $\Phi(e, V_j)$ increases to one using an atomic `xor` operation with a bitmask containing a one at position $i$ respectively $j$.

**Algorithm 4.1:** Moves a node $u$ from block $V_i$ to $V_j$

**Input:** Node $u \in V_i$ and a target block $V_j$
**Output:** Attributed gain $\Delta_{\lambda-1}$ if the node was moved. Otherwise, $\perp$.

```
1  c_j ← fetch-and-add(c(V_j), c(u))
2  if c_j + c(u) ≤ L_max then                    // Check if partition is still balanced
3      Π[u] ← V_j
4      fetch-and-sub(c(V_i), c(u))
5      Δ_{λ−1} ← 0
6      for e ∈ I(u) do
           // Update pin count values and connectivity set of net e
7          Φ_i ← fetch-and-sub(Φ(e, V_i), 1) − 1
8          Φ_j ← fetch-and-add(Φ(e, V_j), 1) + 1
9          if Φ_i = 0 then Λ(e) ← Λ(e) \ {V_i}
10         if Φ_j = 1 then Λ(e) ← Λ(e) ∪ {V_j}
           // Compute attributed gain for hyperedge e
11         if Φ_i = 0 then Δ_{λ−1} ← Δ_{λ−1} + ω(e)
12         if Φ_j = 1 then Δ_{λ−1} ← Δ_{λ−1} − ω(e)
13         updateGainTable(e, Φ_i, Φ_j)                    // see Section 4.1.2
14     return Δ_{λ−1}
15 else
16     fetch-and-sub(c(V_j), c(u))                  // revert block weight update of V_j
17     return ⊥
```

> Consider a net $e$ with $\Phi(e, V_i) = 1$ and assume that the last remaining pin $u \in e$ in block $V_i$ moves out of block $V_i$, while another node $v \in e$ moves to block $V_i$ simultaneously. After applying both node moves, the connectivity set $\Lambda(e)$ should contain $V_i$ since $v \in e \cap V_i$. The pin count update operation either increases $\Phi(e, V_i)$ from 1 to 2 and then to 1 again, or decreases $\Phi(e, V_i)$ from 1 to 0 and then to 1 again, depending on their execution order. The former case does not trigger any connectivity set update. In the latter case, we may first process the connectivity set update that increases $\Phi(e, V_i)$ from 0 to 1 before we observe that $\Phi(e, V_i)$ decreases from 1 to 0. However, since we modify the bitset of the connectivity set with an atomic operation that flips a bit at position $i$, the first update operation removes $V_i$ from $\Lambda(e)$ (even though $\Phi(e, V_i)$ increases from 0 to 1) and the second adds $V_i$ to $\Lambda(e)$ again (even though $\Phi(e, V_i)$ decreases from 1 to 0). Hence, $V_i \in \Lambda(e)$ after applying both node moves regardless of their execution order.

After updating the pin count values and connectivity set, we compute an *attributed gain* value $\Delta_{\lambda-1}$ and update a *gain table* data structure described in the following paragraph and subsection in more detail.

The move node operation is lock-free. However, in our actual implementation, we use

a separate spin-lock for each net $e$ to synchronize writes to $\Phi(e, V_i)$ and $\Phi(e, V_j)$ since we cannot use atomic `fetch-and-add` operations due to their packed representation (one machine word can contain several $\Phi(e, V_i)$ values). Each thread holds at most one spin-lock at a time, and, in particular, we do not lock all incident nets of a node before moving it. Reads are not synchronized. We implement the spin-locks by setting a bit with an atomic `test-and-set` instruction (applies to all locks used in this work).[1] The running time of the move node operation is $\mathcal{O}(|\mathrm{I}(u)| \cdot T_{\mathrm{GC}})$ where $T_{\mathrm{GC}}$ is the complexity of the gain table update procedure which we discuss in Section 4.1.2.

**Attributed Gains.** In this work, we optimize the connectivity metric $\mathfrak{f}_{\lambda-1}(\Pi) = \sum_{e \in E}(\lambda(e) - 1) \cdot \omega(e)$. If we move a node $u$ from block $V_i$ to $V_j$ and $\Phi(e, V_i)$ decreases from one to zero for net $e$, $\mathfrak{f}_{\lambda-1}(\Pi)$ decreases by $\omega(e)$. If, on the other hand, the move increases $\Phi(e, V_j)$ from zero to one, $\mathfrak{f}_{\lambda-1}(\Pi)$ increases by $\omega(e)$.

As the gain value of a node move can change between its initial calculation and actual execution due to concurrent node moves in its neighborhood, we additionally compute an *attributed gain* value $\Delta_{\lambda-1}$ for each move based on the atomic updates of the pin count values $\Phi(e, V_i)$ and $\Phi(e, V_j)$ in Line 11 and 12 of Algorithm 4.1. We attribute a connectivity reduction by $\omega(e)$ to the move that reduces $\Phi(e, V_i)$ to zero (see Line 11) and an increase by $\omega(e)$ for increasing $\Phi(e, V_j)$ to one (see Line 12).

Since we do not lock all incident nets $e \in \mathrm{I}(u)$ before moving a node $u$, there is no guarantee on the order in which concurrent moves perform the pin count updates. Hence, this scheme may distribute the connectivity reductions to different threads, but the sum of the attributed gains equals the overall connectivity reduction. Thus, we use attributed gains as a secondary check to revert moves or move sequences that degrade $\mathfrak{f}_{\lambda-1}(\Pi)$ and correctly track the connectivity metric.

### 4.1.2 The Gain Table

Algorithm 4.2 computes the move with the highest gain for a node $u$. It uses the following reformulation of the gain value:

$$\begin{aligned} g_u(V_j) &= \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi[u]) = 1\}) - \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, V_j) = 0\}) \\ &= \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi[u]) = 1\}) - \omega(\mathrm{I}(u)) + \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, V_j) \geq 1\}) \\ &= b(u) - \omega(\mathrm{I}(u)) + p(u, V_j) \end{aligned}$$

We will refer to $b(u) := \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi[u]) = 1\})$ as the *benefit* term and to $p(u, V_j) := \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, V_j) \geq 1\})$ as the *penalty* term. The new gain formulation has the advantage that we can calculate $p(u, V_j)$ by iterating over the connectivity sets $\Lambda(e)$ of each net $e \in I(u)$ (instead of over all blocks $V_i \in \Pi \setminus \Lambda(e)$ which is usually much larger than $\Lambda(e)$). Algorithm 4.2 has a running time of

[1] We note that the atomic block weight updates can cause contention for smaller values of $k$. However, the work performed by the different refinement algorithms to identify move candidates usually dominates the running time of the move node operation.

**Algorithm 4.2:** Computes the highest gain move for a node $u$

**Input:** Node $u \in V$
**Output:** Pair $(g_u(V_j), V_j)$ where $g_u(V_j) := \max_{V_i \in \Pi} g_u(V_i)$.

```
1  b_u ← 0, p ← [0, ..., 0]                                  // p is array of size k
2  for e ∈ I(u) do
3  |  if Φ(e, Π[u]) = 1 then b_u ← b_u + ω(e)
4  |  for V_i ∈ Λ(e) \ {Π[u]} do p[i] ← p[i] + ω(e)
5  V_j ← Π[u], g_j ← 0
6  for i = 1 to k do
7  |  g_i ← b_u − ω(I(u)) + p[i]
8  |  if g_i ≥ g_j and c(V_i) + c(u) ≤ L_max then // breaking ties uniformly at random
9  |  |  V_j ← V_i, g_j ← g_i
10 return (g_j, V_j)
```

$\mathcal{O}((\sum_{e \in I(u)} |\Lambda(e)|) + k) = \mathcal{O}(|I(u)|k)$. In the following, we present a technique that reduces the running time to $\mathcal{O}(k)$. The idea is to explicitly store and maintain the benefit and penalty term in the partition data structure. We will refer to this data structure as the *gain table*.

**Motivation.** KaHyPar [Sch+16a; Akh+17a] uses a gain table in its highly-localized $k$-way FM algorithm (see Section 3.1.2). The authors report that the gain table reduces the running time of their FM implementation by 45% compared to a variant that computes the gain values each time from scratch. We also integrate the gain table into our parallel FM algorithm (see Section 4.3). Here, the gain table additionally enables fast recomputations to verify if the gain value of the expected best move had changed due to concurrent moves made by other threads.

**Data Layout.** Our gain table maintains the benefit term $b(u)$, the weighted node degree $\omega(\mathrm{I}(u))$, and the penalty term $p(u, V_j)$ for each node $u \in V$ and block $V_j \in \Pi$. The memory consumption of the data structure is $n(k+2)$. We initialize the gain table by iterating over each node in parallel, and use Algorithm 4.2 to compute the benefit and penalty terms.

Since we use only one memory location for $b(u)$, and not one for each block, the term can no longer be correctly updated after a node $u$ is moved to a different block $V_j$. We could recompute the new benefit term $b(u)$, which require locking all incident nets of $u$ to prevent other threads from modifying their pin count values. However, we decided to *invalidate* the gain table entry of a moved node $u$ since our refinement algorithms are organized in rounds where each node is moved at most once. Therefore, we recalculate $b(u)$ for every moved node $u$ after each round.

**Gain Table Update.** Algorithm 4.3 shows the `gainTableUpdate`$(e, \Phi_i, \Phi_j)$ procedure used in Line 13 of Algorithm 4.1. It implements the delta gain updates for the connectivity metric proposed by Schlag [Sch20, Algorithm 4.6 on p. 126], who proved

**Algorithm 4.3:** Gain Table Update

**Input:** A net $e$, and $\Phi_i$ and $\Phi_j$ that we obtained from the atomic updates of the pin count values $\Phi(e, V_i)$ and $\Phi(e, V_j)$ in Algorithm 4.1.

```
1  if Φ_i = 1 then                          // Case 1: Φ(e, V_i) decreased from 2 to 1
2      for u ∈ e do
           // Find last remaining pin of net e in block V_i
           // since moving it out of block V_i decreases λ(e) by one now.
3          if Π[u] = V_i then fetch-and-add(b(u), ω(e))
4  else if Φ_i = 0 then                     // Case 2: Φ(e, V_i) decreased from 1 to 0
       // Moving the pins of net e to block V_i increases λ(e) by one now.
5      for u ∈ e do fetch-and-add(p(u, V_i), −ω(e))
6  if Φ_j = 1 then                          // Case 3: Φ(e, V_j) increased from 0 to 1
       // Moving the pins of net e to block V_j do not increase λ(e) anymore.
7      for u ∈ e do fetch-and-add(p(u, V_j), ω(e))
8  else if Φ_j = 2 then                     // Case 4: Φ(e, V_j) increased from 1 to 2
9      for u ∈ e do
           // Find the previously last pin of net e in block V_j
           // since moving it out of block V_j does not decrease λ(e) anymore.
10         if Π[u] = V_j then fetch-and-add(b(u), −ω(e))
```

their correctness in the sequential setting [Sch20, Theorem 4.3 on p. 127]. We refrain from discussing the different cases in Algorithm 4.3 and refer to the comments in the pseudocode describing them in detail.

In the parallel setting, threads may see intermediate states of gain table updates since we do not synchronize reads. However, we can argue that the gain value $g_u(V_j)$ of each non-moved node $u \in V$ and block $V_j \in \Pi$ is correct after we have applied a sequence of moves in parallel.

We have to show that the order in which we apply the gain table updates is independent of the order of the pin count updates. The correctness of the gain table updates then follows from their correctness in the sequential setting. If we look at the different cases, we see that Case 1 and 4 modify the benefit term and Case 2 and 3 modify the penalty term. Thus, they are independent of each other. Furthermore, Case 3 performs the reverse operation of Case 2 (and vice versa). Thus, the order in which we apply the gain table updates is independent of the order of the pin count updates for both cases. The same holds for Case 1 and 4, which proves the correctness of the gain table updates in the parallel setting.

**Running Time.** We restrict the running time analysis of the gain table updates to the case where each node is moved exactly once. For $k = 2$, the different cases shown in Algorithm 4.3 corresponds to the cases of the delta gain updates proposed for the original FM algorithm [FM82]. Fiduccia and Mattheyses [FM82] showed that

updates are triggered at most four times per net and FM pass. The proof uses the observation that the pin count value $\Phi(e, V_i)$ of a net $e$ and block $V_i$ cannot decrease to zero anymore if one pin $u \in e$ moves to block $V_i$ (since each node is moved exactly once). Thus, we can trigger all cases of Algorithm 4.3 if we first move all pins of net $e$ out of block $V_i$ and then move two pins to block $V_i$ again. Hence, in our setting where each node is moved exactly once, the running time to maintain the gain table is $\sum_{e \in E} 4|e| = \mathcal{O}(p)$.

For general values of $k$, the worst case running time increases to $\mathcal{O}(pk)$. We can use a similar argument as in our previous discussion for $k = 2$. Each net $e$ has $k$ pin count values and each $\Phi(e, V_i)$ value triggers at most four updates per net $e$ and pass. Thus, the total running time sums up to $\sum_{e \in E} 4k|e| = \mathcal{O}(pk)$.

> In the following, we give an example showing that the $\mathcal{O}(pk)$ bound is tight. Let $e = \{u_1, \ldots, u_k\}$ be a net with $k$ pins and $\Phi(e, V_i) = 2$ for all blocks $V_i \in \{V_1, \ldots, V_{k/2}\}$. If we move each pin of net $e$ from its current block $V_i$ to a block $V_{i+k/2}$, then each move triggers exactly two cases of Algorithm 4.3. Thus, the total running time is $\mathcal{O}(k|e|)$.

However, many real-world instances contain only a small fraction of large hyperedges, and the connectivity $\lambda(e)$ of a net $e$ is usually small. Hence, the running time of the gain table updates is often closer to $\mathcal{O}(p)$ than to the $\mathcal{O}(pk)$ bound in practice.

**Differences to KaHyPar.** KaHyPar [Sch20] does not distinguish between a benefit and penalty term and directly stores the gain values $g_u(V_j)$ for each node $u \in V$ and block $V_j \in \Pi$. Therefore, gain table updates modifying the benefit term are more expensive (see Case 1 and 4 in Line 1 and 8).

> If $\Phi(e, V_i)$ decreases from 2 to 1 (Case 1 in Line 1), moving the last remaining pin $u \in e$ of block $V_i$ to a block $V_j \in \Lambda(e) \setminus \{V_i\}$ decreases $\lambda(e)$ now. Therefore, we have to add $\omega(e)$ to the gain value $g_u(V_j)$ for each block $V_j \in \Lambda(e) \setminus \{V_i\}$. If $\Phi(e, V_j)$ increases from 1 to 2 (Case 4 in Line 8), moving the previously last pin $u \in e$ of block $V_j$ to a block $V_i \in \Lambda(e) \setminus \{V_j\}$ does not decrease $\lambda(e)$ anymore. Therefore, we have to add $-\omega(e)$ to the gain value $g_u(V_i)$ for each block $V_i \in \Lambda(e) \setminus \{V_j\}$. For both cases, the update operation consists of identifying the (previously) last pin of net $e$ in block $V_i$ respectively $V_j$ and afterwards update $|\Lambda(e) \setminus \{V_i\}|$ respectively $|\Lambda(e) \setminus \{V_j\}| \leq k - 1$ gain values. Thus, the complexity of the operation is $\mathcal{O}(|e| + k - 1)$ per net.

Consequently, the running time of KaHyPar's gain table updates is

$$\sum_{e \in E} 2|e| + 2(|e| + k - 1) = \mathcal{O}(p + m(k - 1))$$

for $k = 2$, but it only requires to store $nk$ values.

### 4.1.3 Parallel Gain Recalculation

We now propose a parallel algorithm to recompute exact gain values of a sequence of node moves $M = \langle m_1, \ldots, m_t \rangle$ if they are supposed to be performed in this order. Each move $m_i \in M$ is of the form $m_i = (u, V_a, V_b)$, which means that node $u$ is moved from block $V_a$ to $V_b$. We further assume that each node is moved at most once, as it is in our parallel FM algorithm presented in Section 4.3, for which we will use the algorithm later. Recall that a move of a node $u$ from block $V_a$ to $V_b$ decreases the connectivity of a hyperedge $e$ (and also the connectivity metric by $\omega(e)$), if $\Phi(e, V_a)$ decreases to zero. Conversely, it increases the connectivity if $\Phi(e, V_b)$ increases to one. The idea of the following algorithm is to iterate over the hyperedges in parallel, and for each hyperedge, we identify the node moves in $M$ that increase or decrease its connectivity using the algorithm outlined in Algorithm 4.4.

Consider a hyperedge $e$ and a block $V_i \in \Pi$. The first observation is that if we move a pin $v \in e$ to $V_i$, then $\Phi(e, V_i)$ can not decrease to zero anymore since each node is moved at most once. In order to decrease $\Phi(e, V_i)$ to zero, we have to move all pins $u \in e \cap V_i$ out of block $V_i$ before we move the first pin $v \in e \setminus V_i$ to block $V_i$. In this case, the last pin $u \in e$ moved out of block $V_i$ decreases the connectivity of $e$ and the first pin $v \in e$ moved to block $V_i$ increases its connectivity again. Thus, we can decide whether or not a move increases or decreases the connectivity of a hyperedge by simply comparing the indices of the node moves in $M$, which were last moved out and first moved to a particular block. Additionally, we need to know if the move sequence $M$ moves all pins out of block $V_i$. To do so, we count the number of non-moved pins $v \in e$ in each block. If the number of non-moved pins is zero for a block $V_i$, then either $\Phi(e, V_i)$ was zero before, or the move sequence $M$ moved all nodes out of block $V_i$.

Algorithm 4.4 shows the pseudocode that identifies the node moves in $M$ that increase or decrease the connectivity of a hyperedge $e$. The algorithm uses two loops, both iterating over the pins of hyperedge $e$. The first loop computes the indices of the node moves that first moved to and last moved out of each block $V_i \in \Pi$ (see Line 7 and 8). For each non-moved pin $u \in e$ (moved nodes are marked in a bitset), we increment the number of non-moved pins in block $\Pi[u]$ (see Line 9).

The second loop then decides for each moved pin $u \in e$ whether or not it increases or decreases the connectivity of hyperedge $e$ by evaluating the conditions shown in Line 14 and 16. Let $m_i := (u, V_a, V_b)$ be the corresponding node move of pin $u \in e$ in $M$. If $M$ moves all nodes out of block $V_a$ ($\texttt{non\_moved\_pins}[V_a] = 0$) and $u$ is the last pin moved out of block $V_a$ ($\texttt{last\_out}[V_a] = i$), while the first move that moves a pin into block $V_a$ happens strictly after $m_i$ ($i < \texttt{first\_in}[V_a]$), then $m_i$ reduces the connectivity metric by $\omega(e)$. Conversely, if $M$ moves all nodes out of block $V_b$ ($\texttt{non\_moved\_pins}[V_b] = 0$) and $u$ is the first pin moved into block $V_b$ ($\texttt{first\_in}[V_b] = i$), while the last move that moves a pin out of block $V_b$ happens strictly before $m_i$ ($i > \texttt{last\_out}[V_b]$), then $m_i$ increases the connectivity metric by $\omega(e)$. Since we run the algorithm for each hyperedge in parallel, several threads can modify the gain value $g_i$ of a node move $m_i$ simultaneously. We therefore use atomic `fetch-and-add` instructions (see Line 15 and 17).

**Algorithm 4.4:** Parallel Gain Recalculation

**Input:** Hyperedge $e$, a sequence of node moves $M = \langle m_1, \ldots, m_t \rangle$ and a shared gain vector $\mathcal{G} = \langle g_1, \ldots, g_t \rangle$ representing the recalculated gain values

```
1  first_in ← [∞, ..., ∞], last_out ← [0, ..., 0]          // Arrays of size k
2  non_moved_pins ← [0, ..., 0]                             // Array of size k
3  for u ∈ e do
4      V_a ← Π[u]
5      if u was moved then                                  // moved nodes are marked in a bitset
6          m_i := (u, V_a, V_b) ← find corresponding move in M
7          last_out[V_a] ← max(i, last_out[V_a])
8          first_in[V_b] ← min(i, first_in[V_b])
9      else non_moved_pins[V_a] ← non_moved_pins[V_a] + 1
10
11 for u ∈ e do
12     if u was moved then
13         m_i := (u, V_a, V_b) ← find corresponding move in M
14         if last_out[V_a] = i ∧ i < first_in[V_a] ∧ non_moved_pins[V_a] = 0
            then
15             fetch-and-add(g_i, ω(e))
16         if first_in[V_b] = i ∧ i > last_out[V_b] ∧ non_moved_pins[V_b] = 0 then
17             fetch-and-add(g_i, −ω(e))
```

To further reduce the complexity of the algorithm, we only process hyperedges containing moved nodes. To do so, we iterate over the node moves in $M$ in parallel and run Algorithm 4.4 only for incident edges of moved nodes. We mark already processed hyperedges in a shared bitset using atomic `test-and-set` instructions.

## 4.2 Label Propagation Refinement

Our first refinement algorithm is a parallel version of the label propagation algorithm [KK96; WCE97; TK04a; ASS17; MSS17; Got+21e; Mal+21]. The algorithm only performs moves with positive gain and therefore cannot escape from local optima. However, we use it to find all *simple* node moves such that our parallel FM and flow-based refinement algorithm can focus on finding non-trivial improvements [ASS17]. In the following, we describe our parallel implementation of the algorithm outlined in Algorithm 4.5.

**Algorithm Overview.** The label propagation algorithm works in rounds. In each round, we iterate in parallel over all boundary nodes, and whenever we visit a node $u$, we compute the block $V_j$ that maximizes the gain $g_u(V_j)$ and satisfies the balance

**Algorithm 4.5:** The Parallel Label Propagation Algorithm

**Input:** Hypergraph $H = (V, E, c, \omega)$ and $k$-way partition $\Pi = \{V_1, \ldots, V_k\}$

```
1  i ← 0
2  b_u ← [0, ..., 0]                                   // bitset of size n
3  b_e ← [0, ..., 0]                                   // bitset of size m
4  V' ← find all border nodes u ∈ V of Π
5  while i < i_max and |V'| > 0 do
6      V' ← parallelRandomShuffle(V'), V'' ← ∅         // see Section 2.3
7      for u ∈ V' do in parallel
8          (g_j, V_j) ← compute move with highest gain for node u   // see Algorithm 4.2
9          if g_j > 0 or move improves balance then
10             Δ_{λ-1} ← move node u to block V_j       // see Algorithm 4.1
11             if Δ_{λ-1} ≥ 0 then                     // move improved f_{λ-1}(Π)
                   // Activate u and its neighbors for next round
12                 for e ∈ I(u) do                     // ignoring large nets
13                     if compare-and-swap(b_e[e], 0, 1) then
14                         for v ∈ e do
15                             if compare-and-swap(b_u[v], 0, 1) then
16                                 V'' ← V'' ∪ {u}
17             else if Δ_{λ-1} < 0 then revert move    // move has worsened f_{λ-1}(Π)
18     V' ← V'', i ← i + 1
19     reset all entries of b_u and b_e to 0           // see Section 2.3
```

constraint using Algorithm 4.2. We then move $u$ to block $V_j$ if it either has positive gain or, in case of a zero gain improvement, if it improves the balance. In Line 10, we apply the move on the partition data structure and obtain an attributed gain value $\Delta_{\lambda-1}$. If $\Delta_{\lambda-1}$ is negative, we assume that the move has worsened the connectivity metric due to concurrent moves made by other threads and revert it in Line 17. Otherwise, we activate $u$ and its neighbors for the next round in Line 12 – 15 (discussed in the next paragraph). The algorithm proceeds until a predefined number of rounds $i_{\max}$ is reached or none of the nodes changed its block in a round ($|V'| = 0$).

**Active Nodes.** We use the *active node strategy* to reduce the number of nodes visited in a round [MSS14]. A node $u$ is called *active* if either $u$ or one of its neighbors changed its block in the previous round. Initially, all boundary nodes of the partition are active. Only active nodes can have a positive gain. In later rounds, gain values of inactive nodes cannot change because none of their neighbors has changed its block in the previous round. We use two shared bitsets $b_e$ and $b_u$ to mark already activated nets and nodes. New active nodes are inserted into thread-local vectors and copied to the active node set $V'$ at the end of each round.

## 4.3 Direct $k$-Way FM Local Search

The classical FM algorithm inserts all boundary nodes into a PQ with the highest gain as key and repeatedly performs the best feasible move. In contrast, the multi-try $k$-way FM algorithm of Sanders and Schulz [SS11] initializes the PQ only with a single boundary node and its neighbors. The algorithm then gradually expands the search around the seed nodes by inserting neighbors of moved nodes into the PQ. A stopping rule is used to terminate the search early if it becomes unlikely to find further improvements [OS10]. Afterwards, the algorithm continues with the next highly-localized FM search until all nodes are moved at most once.

The multi-try $k$-way FM algorithm is highly amenable to parallelization since multiple searches starting from different seed nodes can run in parallel. A parallel version of the algorithm is already implemented in the shared-memory graph partitioner Mt-KaHIP [ASS17]. The threads perform localized searches that do not overlap on nodes. Each thread initializes its search with a different seed node, and gradually expands around it by claiming neighbors of moved nodes. The seed nodes are polled from a task queue initialized with all boundary nodes. The algorithm performs node moves locally using thread-local hash tables and therefore the *moves* performed by the different searches *are not visible to other threads*. Once the task queue is empty, the move sequences of the threads are concatenated to a global move sequence, for which *gains are recomputed sequentially*.

In the following, we present our parallel implementation of the algorithm that overcomes several shortcomings of the approach implemented in Mt-KaHIP. First, we can recompute the gains of the global move sequence in parallel using the algorithm presented in Section 4.1.3. Second, threads immediately apply a move sequence to the global partition once an improvement is found, and other threads can see these changes. Since this can invalidate gain values stored in the PQs of other searches, we propose techniques to adapt the gain values to changes on the global partition.

### 4.3.1 Multi-Try $k$-Way FM Algorithm

Algorithm 4.6 outlines the pseudocode of their parallel multi-try $k$-way FM algorithm. The algorithm proceeds in rounds, and each round starts with inserting all boundary nodes into a globally shared task queue $Q$. The threads then poll a fixed number of nodes from $Q$ which we use as seed nodes for the highly-localized FM searches in Line 9. The searches are non-overlapping, i.e., threads acquire exclusive ownership of nodes (see Line 7 in Algorithm 4.6), while hyperedges can touch multiple searches. Node moves performed by the different searches are not visible to other threads (performed locally using thread-local hash tables). However, once a thread finds an improvement, it immediately applies it to the global partition. We increment an atomic counter to assign global IDs to node moves at the time they are performed on the global partition, and store them in a vector that represents the global move sequence. We repeatedly start localized searches until the task queue is empty.

**Algorithm 4.6:** Multi-Try $k$-Way FM Algorithm

```
Input: Hypergraph H = (V, E, c, ω), k-way partition Π = {V_1, ..., V_k}
1  for i = 1 to i_max do
2      Q ← initialize task queue with all boundary nodes of Π
3      while Q is not empty do in parallel
4          V_seed ← ∅
5          while |V_seed| ≤ s_max and Q is not empty do
6              u ← Q.pop()
7              if try to acquire node u then
8                  V_seed ← V_seed ∪ {u}
9          localizedKWayFM(H, Π, V_seed)                    // see Algorithm 4.7
10     Π ← recomputeGainsAndRevertToBestGlobalPrefix()      // see Section 4.1.3
11     if no improvement then break
```

Once the task queue is empty, we proceed to the second phase, where we recalculate the gains of the global move sequence (see Section 4.1.3) and then use a parallel prefix sum and reduce operation on the recomputed gain values to revert to the best seen solution. The algorithm proceeds until a fixed number of rounds is reached or it fails to improve the connectivity metric in a round.

### 4.3.2 Highly-Localized $k$-Way FM Search

Algorithm 4.7 shows the pseudocode of our highly-localized $k$-way FM algorithm. It uses a single PQ storing the move with the highest gain for each inserted node. We initialize the PQ with several seed nodes and use the gain table to compute the initial best move for each node (see Line 2 – 4). Then, we repeatedly select the move with the highest gain and apply it to a thread-local partition $\Delta\Pi$. Changes on $\Delta\Pi$ are not visible to other threads. However, we immediately apply a sequence of moves to the global partition $\Pi$ once we find an improvement (see Line 15). When we move a node $u$ to a different block, we collect the nets $e \in \mathrm{I}(u)$ affected by a gain table update. We use them to update the gain values of nodes in the PQ and expand the search to neighbors of moved nodes. The localized search terminates when the PQ becomes empty or the adaptive stopping rule of Osipov and Sanders [OS10; Akh+17a] is triggered (see Section 3.1.2). We release the ownership of non-moved nodes at the end such that other searches can acquire them again. We do not release the ownership of moved nodes to ensure that each node is moved at most once during an FM pass.

**Thread-Local Partition.** The FM algorithm repeatedly performs the move with the highest gain, which can also intermediately worsen the solution quality. At the end of each search, the algorithm reverts to the best seen solution. If we immediately apply all moves found by the different searches to the global partition, the threads may

**Algorithm 4.7:** Highly-Localized $k$-Way FM Algorithm

**Input:** Hypergraph $H = (V, E, c, \omega)$, $k$-way partition $\Pi = \{V_1, \ldots, V_k\}$ and a set of seed nodes $V_{\text{seed}}$

```
1  ΔΠ ← Π
2  for u ∈ V_seed do                                   // initialize PQ with seed nodes
3      (g_j, V_j) ← computeMaxGainMove(u)              // using the gain table
4      PQ.insert(u, g_j, V_j)
5  cur_{λ−1} ← 0, M ← ∅
6  while PQ not empty and search should continue do
7      while true do                                   // extract move with highest gain from PQ
8          (u, g_j, V_j) ← PQ.maxGainMove()
9          (g_u, V_j) ← recomputeMaxGainMove(u)        // using the gain table
10         if g_u ≥ g_j then g_j ← g_u, PQ.pop(), break   // accept move
           // update gain of node u since its actual gain is smaller than expected
11         else PQ.update(u, g_u, V_j)
12     apply move of node u to block V_j on thread-local partition ΔΠ
13     cur_{λ−1} ← cur_{λ−1} + g_j, M ← M ∪ {(u, V_j)}
14     if cur_{λ−1} > 0 or (cur_{λ−1} = 0 and move improved balance) then
15         apply move sequence M to global partition Π
16         cur_{λ−1} ← 0, M ← ∅, ΔΠ ← Π
17     for all nets e ∈ I(u) affected by a gain table update do
18         for v ∈ e do
19             if v is not marked then
20                 if PQ.contains(v) then update gain of v in PQ
                   // expand search to neighbors of u
21                 else if try to acquire node v then
22                     (g_j, V_j) ← computeMaxGainMove(v)   // using the gain table
23                     PQ.insert(v, g_j, V_j)
24                 mark v
25     unmark all nodes
26 for u ∈ PQ do release ownership of u
```

see intermediate states of other searches that are later reverted because they do not improve the connectivity metric. This becomes a problem when other threads perform moves depending on these states since reverting a move sequence can invalidate their gain values. Therefore, we apply node moves to a thread-local partition $\Delta\Pi$ using hash tables and only perform them on the global partition if they lead to an improvement.

The thread-local partition $\Delta\Pi$ stores changes relative to the global partition. For example, we compute the weight of a block $V_i$ by calculating $c(V_i) + \Delta c(V_i)$ where

$c(V_i)$ is the weight of block $V_i$ stored in the global partition data structure and $\Delta c(V_i)$ is the weight of all nodes that locally moved to block $V_i$ minus the weight of nodes that moved out of block $V_i$. We maintain the pin count values, and benefit and penalty terms of the gain table analogously. Moving a node locally overrides its block ID in the global partition data structure.

Applying a move sequence to the global partition makes it immediately visible to the thread-local partitions of other threads. Since $\Delta\Pi$ stores local changes relative to the global partition, the block weights and pin count values are still correct. However, some gain values may be incorrect since the gain table updates on the global partition do not consider moves performed locally. In practice, as we will see in the experimental evaluation, the scheme drastically reduces conflicts since only a small fraction of the searches find an improvement.

**Apply Moves.** After extracting a node move from the PQ (see Line 8), we recompute its gain value using the gain table. If the recomputed gain is worse than the expected gain value, we update the PQ accordingly. Otherwise, we apply it to the thread-local partition. This way, we update the PQ to changes made by other threads.

If the observed gain values suggest an improvement, we immediately apply the corresponding move sequence to the global partition. During that, we compute an attributed gain value for each move and, if the actual does not match the expected improvement, we revert to the best seen solution based on the attributed gains.

**Gain Updates.** If we move a node $u$ from its current block $V_i$ to a target block $V_j$, the gain values of its neighbors $v \in \Gamma(u)$ may change. Therefore, we collect the nets $e \in \mathrm{I}(u)$ affected by a gain table update and iterate over the pins to update their gain values in the PQ. Let $v \in \Gamma(u)$ be a node in the PQ and $V_k$ its target block with the highest gain before we moved node $u$ from block $V_i$ to $V_j$. If $V_k \neq V_i$ and $V_k \neq V_j$, then only $g_v(V_i)$ and $g_v(V_j)$ can be greater than $g_v(V_k)$. Hence, we compare the gain values of moving $v$ to one of the three blocks using the gain table and update the gain of $v$ in the PQ accordingly. If $V_k = V_i$ or $V_k = V_j$, then the move may have reduced $g_v(V_k)$, and we recompute the highest gain block for node $v$ using the gain table.

If a node $v \in \Gamma(u)$ is not in the PQ, the thread tries to acquire its ownership and inserts it into the PQ (see Line 23). The original multi-try $k$-way FM algorithm [ASS17] inserts all nodes adjacent to the moved node into the PQ. Our implementation only considers adjacent nodes incident to nets affected by a gain table update. This way, we expand the search to boundary nodes whose gain values are affected by the current move and, subsequently, may improve the connectivity metric. Furthermore, we run the label propagation algorithm before the FM algorithm in our multilevel algorithms presented in Chapter 5 and 6. Thus, there are usually no positive gain moves left. Consequently, node moves have to trigger gain table updates to improve the connectivity metric.

**Implementation Details.** We use a binary heap-based priority queue implementation and store handles to identify elements in the PQ in a shared vector of size $n$. The thread-local partitions $\Delta\Pi$ use hash tables with linear probing to resolve collisions. If

the fill grade of a hash table becomes larger than $\frac{2}{5}$, we double its size and rehash the elements.

## 4.4 Flow-Based Refinement

The FM algorithm can escape from local optima, but its greedy nature can prevent it from finding some non-trivial improvements. In particular, if an improvement requires to violate the balance constraint intermediately, the FM algorithm may fail to discover it because it only performs moves preserving balance (see Figure 3.2). Furthermore, large hyperedges can make it difficult to find meaningful moves [Saa95]. They tend to have many pins in different blocks, leading to mostly zero-gain moves.

Maximum flows overcome the limitations of existing move-based local search heuristics because they find a minimum cut separating two nodes [FF56], and thus have a more global view on the partitioning problem. However, they were long overlooked due to their complexity [YW96], but have since enjoyed wide-spread adoption in the partitioning community [YW96; SS11; HS18; HSS19a; Got+20; GHW19]. Flow-based refinement is currently considered the most powerful improvement heuristic for (hyper)graph partitioning, but often substantially increases the running time of sequential partitioning algorithms, making them impractical for partitioning very large (hyper)graphs.

**Algorithm Overview.** In this section, we present the first parallel flow-based refinement algorithm for hypergraph partitioning. The high-level pseudocode of the algorithm is outlined in Algorithm 4.8. Flow-based refinement works on biparitions and can be scheduled on different block pairs to improve $k$-way partitions [SS11; HSS19a; Got+20]. We therefore start with a parallel scheduling scheme of adjacent block pairs based on the quotient graph in Section 4.4.1 (see Line 1 and 2). In Section 4.4.2, we describe the flow network construction algorithm that extracts a subhypergraph around the boundary nodes of two adjacent blocks, which then yields a flow network (see Line 3 and 4). On each network we run the FlowCutter algorithm, whose partition we convert into a set of moves $M$ and an expected connectivity reduction $\Delta_{\text{exp}}$. FlowCutter and its parallelization are discussed in Sections 4.4.3 and 4.4.4. If FlowCutter claims an improvement, i.e., $\Delta_{\text{exp}} \geq 0$, we apply the moves to the global partition and compute the exact reduction $\Delta_{\lambda-1}$, based on which we either mark the blocks for further refinement, or revert the moves (see Line 8 and 9).

### 4.4.1 Parallel Active Block Scheduling

Sanders and Schulz [SS11] propose the active block scheduling algorithm to apply their flow-based refinement algorithm for bipartitions on $k$-way partitions. Their algorithm proceeds in rounds. In each round, it schedules all pairs of adjacent blocks where at least one is marked as *active*. Initially, all blocks are marked as active. If a search on two blocks improves the edge cut, both are marked as active for the next round.

**Algorithm 4.8:** Parallel Flow-Based Refinement

**Input:** Hypergraph $H = (V, E, c, \omega)$ and $k$-way partition $\Pi$ of $H$

```
1 Q ← buildQuotientGraph(H, Π)                                  // see Section 4.4.1
2 while ∃ active (V_i, V_j) ∈ Q do in parallel                  // see Section 4.4.1
3     B :=← constructRegion(H, V_i, V_j)                        // see Section 4.4.2
4     (H, s, t) ← constructFlowNetwork(H, B)                    // see Section 4.4.2
5     (M, Δ_exp) ← FlowCutterRefinement(H, s, t)                // see Section 4.4.3 – 4.4.4
6     if Δ_exp ≥ 0 then                                         // potential improvement
7         Δ_{λ−1} ← applyMoves(H, Π, M)                         // see Section 4.4.1
8         if Δ_{λ−1} > 0 then mark V_i and V_j as active        // found improvement
9         else if Δ_{λ−1} < 0 then revertMoves(H, Π, M)         // no improvement
```

**Parallelization.** A simple scheme would be to schedule block pairs that form a maximum matching in the quotient graph $\mathcal{Q}$ in parallel [HSS10]. This allows searches to operate in independent regions of the hypergraph and thus avoids conflicting moves between different block pairs. However, this scheme restricts the available parallelism to at most $\frac{k}{2}$ threads. Thus, we do not enforce any constraints on the block pairs processed concurrently, e.g., there can be multiple threads running on the same block and they can also share some of their nodes as illustrated in Figure 4.2. We use $\min(t, \tau \cdot k)$ threads to process the active block pairs in parallel, where $t$ is the number of available threads in the system. The parameter $\tau$ controls the available parallelism in the scheduler. With higher values of $\tau$, more block pairs are scheduled in parallel. This can lead to interference between searches that operate on overlapping regions. Lower values for $\tau$ can reduce these conflicts but put more emphasis on good parallelization of 2-way refinement to achieve good speedups.

Our parallel active block scheduling algorithm uses one concurrent FIFO queue $A$ to schedule active block pairs. Each block pair is associated with a round and each round uses an array of size $k$ to mark blocks that become active in the next round. If a search finds an improvement on two blocks $(V_i, V_j)$ and $V_i$ or $V_j$ becomes active, we push all adjacent blocks into $A$ if they are not contained yet (marked using an atomic `test-and-set` instruction). If either $V_i$ or $V_j$ is already active, we insert $(V_i, V_j)$ into $A$ if not contained. Thus, active block pairs of different rounds are stored interleaved in $A$ and the end of a round does not induce a synchronization point as in the original algorithm [SS11]. For processing in the first round, we sort active block pairs in descending order of improvement they contributed in previous invocations (e.g., on previous levels in the multilevel context), with ties broken by larger cut size. A round ends when all of its block pairs have been processed and all prior rounds have ended. If the relative improvement at the end of a round is less than 0.1%, we immediately terminate the algorithm.

**Apply Moves.** When applying a move sequence $M$ to the global partition $\Pi$ (each move $m := (u, V_i, V_j) \in M$ moves a node $u$ from its current block $V_i$ to $V_j$), there are

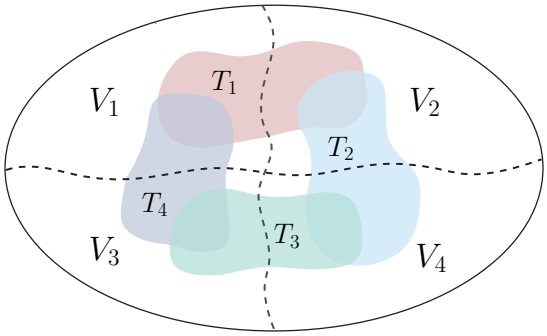



**Figure 4.2:** Illustration of the parallel scheduling scheme of flow computations on adjacent block pairs ($T_i$ denotes the flow network constructed by thread $i$).

three conflict types that can occur: balance constraint violations, $\Delta_{\lambda-1} \neq \Delta_{\text{exp}}$, i.e., the expected does not match the actual connectivity reduction, and nodes may no longer be in the block expected by $M$. These conflicts arise, because concurrently scheduled flow computations on block pairs can share some of their nodes and nets, which causes data races on the partition assignments $\Pi$, pin count values $\Phi(e, V_i)$ and connectivity sets $\Lambda(e)$. These are concurrently read by the network construction and modified when moves are applied. Updates after the construction are not observed, and thus the state at the time a refinement finishes may differ from the expected state.

Since the running time to apply moves is negligible compared to solving flow problems (see Figure 5.21 in Section 5.7.4), we can afford to use a lock so that only one thread applies moves at a time to address these conflicts. First, we remove all nodes from $M$ that are not in their expected block. Afterwards, we compute the block weights if all remaining moves were applied. If balanced, we perform the moves, during which we aggregate the attributed gains $\Delta_{\lambda-1}$ of each move. If $\Delta_{\lambda-1} < 0$, we revert all moves.

**Quotient Graph Construction.** For each block pair, we explicitly store the hyperedges connecting both. This information is required by the flow network construction algorithm to construct the region $B$. Block pairs that contain at least one hyperedge form the edges of the quotient graph. We construct this data structure by iterating over all hyperedges in parallel and add a hyperedge $e \in E$ to the block pairs contained in $\{\{V_i, V_j\} \subseteq \Lambda(e) \mid i < j\}$.

If we apply a move sequence on the partition, we add all hyperedges $e \in E$ where $\Phi(e, V_j)$ increases to one to all block pairs contained in $\{\{V_j, V_k\} \mid V_k \in \Lambda(e) \setminus \{V_j\}\}$. If $\Phi(e, V_i)$ decreases to zero, we remove $e$ lazily from the corresponding block pairs in the flow network construction algorithm.

**Implementation Details.** KaHyPar [HSS19a] established several pruning rules to skip unpromising flow computations that we also use in our parallel algorithm. The first rule skips refinement on two adjacent blocks if the cut between both is less than ten. The second rule modifies the active block scheduling algorithm such that after

the first round only block pairs are scheduled where at least one flow computation on them improved the solution quality on a previous level (only applicable in the multilevel context).

We additionally introduce a time limit to abort long-running flow computations. During scheduling, we track the average running time $\overline{t_f}$ required to solve a flow problem and set the time limit to $8 \cdot \overline{t_f}$. We activate the time limit once $k$ (number of blocks) block pairs have been processed.

### 4.4.2 Network Construction

To improve the cut of a bipartition $\Pi = \{V_1, V_2\}$, we grow a size-constrained region $B$ around the cut hyperedges of $\Pi$. We then contract all nodes in $V_1 \setminus B$ to the source $s$ and $V_2 \setminus B$ to the sink $t$ [SS11; GHW19] as illustrated in Figure 4.3 and obtain a coarser hypergraph $\mathcal{H}$. The flow network $\mathcal{N}$ is then given by the Lawler expansion of $\mathcal{H}$ described in Section 2.1. Note that reducing the hyperedge cut of a bipartition induced by two adjacent blocks of a $k$-way partition $\Pi_k$ optimizes the connectivity metric of $\Pi_k$ [HSS19a].

**Network Extraction Algorithm.** Sanders and Schulz [SS11] grow a region $B := B_1 \cup B_2$ with $B_1 \subseteq V_1$ and $B_2 \subseteq V_2$ around the cut hyperedges of $\Pi$ via two breadth-first-searches (BFS) as illustrated in Figure 4.3. The first BFS is initialized with all boundary nodes of block $V_1$ and continues to add nodes to $B_1$ as long as $c(B_1) \leq (1 + \alpha\varepsilon)\lceil \frac{c(V)}{2} \rceil - c(V_2)$, where $\alpha$ is an input parameter. The second BFS that constructs $B_2$ proceeds analogously. For $\alpha = 1$, each flow computation yields a balanced bipartition with a possibly smaller cut in the original hypergraph, since only nodes of $B$ can move to the opposite block ($c(B_1) + c(V_2) \leq (1 + \varepsilon)\lceil \frac{c(V)}{2} \rceil$ and vice versa for block $B_2$). Larger values for $\alpha$ lead to larger flow problems with potentially smaller minimum cuts, but also increase the likelihood of violating the balance constraint. However, this is not a problem since the flow-based refinement routine guarantees balance through incremental minimum cut computations (see Section 4.4.3). In practice, we use $\alpha = 16$ (also used in KaHyPar [HSS19a; Got+20]). We additionally restrict the distance of each node $v \in B$ to the cut hyperedges to be smaller than or equal to a parameter $\delta$ (= 2). We observed that it is unlikely that a node *far* way from the cut is moved to the opposite block by the flow-based refinement.

**Network Construction Algorithm.** We implemented two construction algorithms that are preferable in different situations. Both construct the hypergraph $\mathcal{H}$ as explained at the beginning of Section 4.4.2. In the following, we will denote with $E_B := \{e \in E \mid e \cap B \neq \emptyset\}$ the set of hyperedges that contain nodes of region $B$.

The first algorithm iterates over all nets $e \in E_B$. If a pin $p \in e$ is contained in $B$, we add $p$ to hyperedge $e$ in $\mathcal{H}$. Otherwise, we add the source $s$ or sink $t$ to $e$, if $p \in V_1$ or $p \in V_2$. The second algorithm iterates over all nodes $u \in B$ and for each net $e \in I(u)$, we insert a pair $(e, u)$ into a vector. Sorting the vector (lexicographically) yields the pin lists of the subhypergraph $H_B$. Afterwards, we insert each net $e$ in the

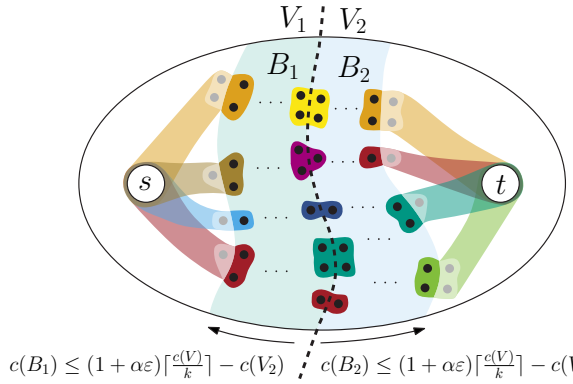



**Figure 4.3:** Illustration of the flow network construction algorithm.

pin list vector into $\mathcal{H}$ and add the source $s$ or sink $t$ to $e$, if $\Phi(e, B_1) < \Phi(e, V_1)$ or $\Phi(e, B_2) < \Phi(e, V_2)$.

The first algorithm has linear running time, but has to scan all hyperedges of $H$ in their entirety in the worst case even if most of their pins are not contained in $\mathcal{H}$. The complexity of the second algorithm only depends on the number of pins in $\mathcal{H}$, but requires to sort the pin lists in a temporary vector. We use the second algorithm for hypergraphs with a low density $d := |E|/|V|$ ($\leq 0.5$) or a large average hyperedge size $|e|$ ($\geq 100$). Note that both algorithms discard single-pin nets and nets that contain both the source and sink (such nets cannot be removed from the cut).

**Parallelization.** The first algorithm iterates over all nets $e \in E_B$ in parallel and each thread uses the sequential algorithm to construct a thread-local pin list vector. Afterwards, we use a prefix sum operation to copy the pin lists of each thread to $\mathcal{H}$.

The second algorithm iterates over all nodes $u \in B$ in parallel and then uses hashing to distribute the pairs $(e, u)$ to buckets. Afterwards, we process each bucket in parallel and apply the sequential algorithm to construct the pin list vector of each bucket. Finally, we use a prefix sum operation to copy the pin lists of each bucket to $\mathcal{H}$.

**Identical Net Removal.** Since some nets of $H$ are only partially contained in $\mathcal{H}$, some of them may become identical. Therefore, we further reduce the size of $\mathcal{H}$ by removing all identical nets except for one representative at which we aggregate their weight. We use the identical net detection algorithm of Aykanat et al. [ACU08b; DKÇ13]. It uses *fingerprints* $f_e := \sum_{v \in e} v^2$ to eliminate unnecessary pairwise comparisons between nets. Nets with different fingerprints or different sizes cannot be identical. If we insert a net $e$ into $\mathcal{H}$, we store the pair $(f_e, e)$ in a hash table with chaining to resolve collisions (uses concurrent vectors to handle parallel access). We can then use the hash table to perform pin-list comparisons between the nets with the same fingerprint for subsequent net insertions. Note that in the parallel scenario we may not be able to detect all identical nets due to simultaneous insertions. However, this does not affect correctness, as removing them is only a performance optimization.

**Algorithm 4.9:** The FlowCutter Algorithm

**Input:** Original hypergraph $H = (V, E, c, \omega)$, flow network $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and a source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$
**Output:** Balanced Bipartition of $\mathcal{H}$

```
1 S ← {s}, T ← {t}                                   // initialize source and sink set
2 initialize flow f : V × V → ℝ≥0 with ∀(u, v) ∈ V × V : f(u, v) = 0
3 while true do
4     f ← parallelMaxPreflow(H, S, T, f)      // augment f to a maximum preflow
5     (S_r, T_r) ← derive source- and sink-side cut S_r, T_r ⊂ V
6     if (S_r, V \ S_r) is balanced then return (S_r, V \ S_r)
7     else if (V \ T_r, T_r) is balanced then return (V \ T_r, T_r)
8     if c(S_r) ≤ c(T_r) then S ← S_r ∪ selectPiercingNode(S ∪ S_r)
9     else T ← T_r ∪ selectPiercingNode(T ∪ T_r)
```

### 4.4.3 The FlowCutter Algorithm

In this section we discuss the flow-based refinement on a bipartition. We introduce the aforementioned FlowCutter algorithm [YW96; HS18]. It is parallelized by plugging in a parallel maximum flow algorithm, which we discuss in the next section. To speed up convergence and make parallelism worthwhile, we propose an optimization named *bulk piercing*.

**Algorithm Overview.** FlowCutter solves a sequence of incremental maximum flow problems until a balanced bipartition is found. Algorithm 4.9 shows the pseudocode for the approach. In each iteration, first the previous flow (initially zero) is augmented to a maximum flow regarding the current source set $S$ and sink set $T$. Subsequently, the node sets $S_r, T_r \subset \mathcal{V}$ of the source- and sink-side cuts are derived. This is done via residual (parallel) BFS (forward from $S$ for $S_r$, backward from $T$ for $T_r$). The node sets induce two bipartitions $(S_r, \mathcal{V} \setminus S_r)$ and $(\mathcal{V} \setminus T_r, T_r)$. If neither is balanced, all nodes on the side with smaller weight are transformed to a source (if $c(S_r) \leq c(T_r)$) or a sink otherwise. To find a different cut in the next iteration, one additional node is added, called *piercing node*. Thus, the bipartitions contributed by the currently smaller side will be more balanced with a possibly larger cut in future iterations. Since the smaller side is grown, this process will converge to a balanced partition.

**Piercing.** For our purpose, there are two important piercing node selection heuristics: *avoid augmenting paths* [YW96; HS18] and *distance from cut* [Got+20]. Whenever possible, a node that is not reachable from the source or sink should be picked, i.e., $v \in \mathcal{V} \setminus (S_r \cup T_r)$. Such nodes do not increase the weight of the cut, while improving balance. As a secondary criterion, larger distances from the original cut are preferred, to reconstruct parts of it.

**Most Balanced Cut.** Once the partition is balanced, we continue to pierce as long as the cut does not increase. This is repeated with different random choices since it is

fast (no flow augmentation). More balance gives other refinement algorithms more leeway for improvement. An equivalent heuristic was already proposed in previous flow-based refinement algorithms, called the *most balanced minimum cut* heuristic [PQ80; SS11] (see Section 3.2).

**Bulk Piercing Optimization.** The complexity of FlowCutter is $O(\zeta m)$, where $\zeta$ is the final cut weight, and $m = |\mathcal{E}|$. This bound stems from a pessimistic implementation that augments one flow unit in $O(m)$ work [YW96; HS18]. For refinement, the performance is much better in practice, as the first cut is often close to the final cut. Only few augmenting iterations are needed and much less than $O(m)$ work is spent per flow unit [GHW19], with most work spent on the initial flow.

Still, the flow augmented per iteration is often small: at most the capacity of edges incident to the piercing node. On large instances, we observed that the number of required iterations increases substantially. We propose to accelerate convergence by piercing multiple nodes per iteration, as long as we cannot avoid augmenting paths and are far from balance. To ensure a poly-log iteration bound, we set a geometrically shrinking goal of weight to add to each side per iteration. The initial goal for the source side is set to $\beta(\frac{c(\mathcal{V})}{2} - c(s))$, where $\beta \in (0, 1)$ is the geometric shrinking factor that is multiplied with the term in each iteration, and $\frac{c(\mathcal{V})}{2} - c(s)$ is the weight to add for perfect balance.

If a goal is not met, its remainder is added to next iteration's goal. We track the average weight added per node and from this estimate the number of piercing nodes needed to match the current goal. To boost measurement accuracy, we pierce one node for the first few rounds. The sides have distinct measurements and goals, so that we do not pierce too aggressively when the smaller side flips.

### 4.4.4 Parallel Maximum Flow Algorithm

Maximum flow algorithms are notoriously difficult to parallelize efficiently [SV82; AS95; BBS15; KÖ19]. The synchronous push-relabel approach of Baumstark et al. [BBS15] is a recent algorithm that sticks closely to sequential FIFO and thus shows good results. We first describe the sequential push-relabel algorithm proposed by Goldberg and Tarjan [GT88] and then present its parallelization. Afterwards, we describe a so far undocumented bug of their algorithm followed by our fix, and conclude with implementation details and intricacies of using FlowCutter with preflows. Note that a maximum preflow already yields a minimum cut, which suffices for our purpose.

**Sequential Push-Relabel Algorithm.** The *push-relabel* algorithm [GT88] stores a distance label $d(u)$ and an excess value $\text{exc}(u) := \sum_{v \in \mathcal{V}} f(v, u)$ for each node. It maintains a *preflow* [Kar74] which is a flow where the conservation constraint is replaced by $\text{exc}(u) \geq 0$. The distance labels represent a lower bound for the distance of each node to the sink. A node $u \in \mathcal{V}$ is *active* if $\text{exc}(u) > 0$. An edge $(u, v) \in \mathcal{E}$ is *admissible* if $r_f(u, v) > 0$ and $d(u) = d(v) + 1$. A *push*$(u, v)$ operation sends $\delta = \min(\text{exc}(u), r_f(u, v))$ flow units over $(u, v)$. It is applicable if $u$ is active and $(u, v)$ is admissible. A *relabel*$(u)$ operation updates the distance label of $u$ to

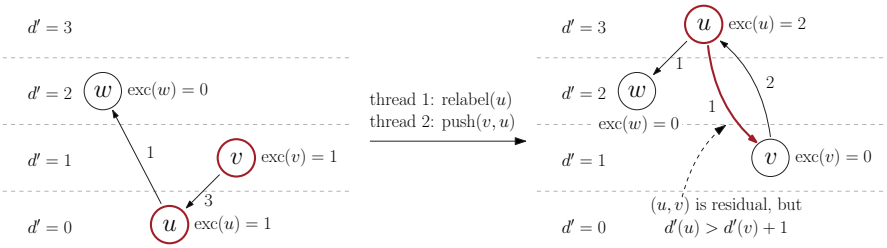



**Figure 4.4:** Example of a push-relabel conflict in the parallel discharge routine (adapted from Ref. [KÖ19]). The numbers on the arcs denote their residual capacities.

$\min(\{d(v) + 1 \mid r_f(u, v) > 0\})$, which is applicable if $u$ is active and has no admissible edges. The distance labels are initialized to $\forall u \in \mathcal{V} \setminus \{s\} : d(u) = 0$ and $d(s) = |V|$ and all source edges are saturated. Efficient variants use the *discharge* routine, which repeatedly scans the edges of an active node until its excess is zero. All admissible edges are pushed and at the end of a scan, the node is relabeled. Discharging active nodes in FIFO order results in an $\mathcal{O}(|\mathcal{V}|^3)$ time algorithm. The *global relabeling* heuristic [CG97] frequently assigns exact distance labels by performing a reverse BFS from the sink, to reduce relabel work in practice. Note that preflows already induce minimum sink-side cuts, so if only a minimum cut is required, the algorithm can already stop once no active nodes with distance label $< n$ exist.

**Synchronous Parallel Push-Relabel.** The parallel push-relabel algorithm of Baumstark et al. [BBS15] proceeds in rounds in which all active nodes are discharged in parallel. The flow is updated globally, the nodes are relabeled locally and the excess differences are aggregated in a second array using atomic instructions. After all nodes have been discharged, the distance labels $d$ are updated to the local labels $d'$ and the excess deltas are applied. The discharging operations thus use the labels and excesses from the previous round. This is repeated until there are no nodes with $\text{exc}(v) > 0$ and $d(v) < n$ left. To avoid concurrently pushing flow on residual arcs in both directions (race condition on flow values), a deterministic winning criterion on the old distance labels is used to determine which direction to push, if both nodes are active. If an arc cannot be pushed due to this, the discharge terminates after the current scan, as the node may not be relabeled in this round. The rounds are interleaved with global relabeling [CG97], after linear push and relabel work, using parallel reverse BFS in the residual network.

**A Bug in the Synchronous Algorithm.** The parallel discharge routine does not protect against push-relabel conflicts [KÖ19] as illustrated in Figure 4.4. In particular the winning criterion does not help. A node $u$ may be relabeled too high if it is concurrently pushed to through a residual arc $(v, u)$ with $d'(v) = d(u) + 1$. The arc $(u, v)$ may not be observed as residual yet, and thus $u$ may set its new label $d'(u) > d'(v) + 1$, violating label correctness. The bug becomes noticeable when

the algorithm terminates prematurely with incorrect distances. Our fix is to collect mislabeled excess nodes during global relabeling. When the algorithm would terminate, we run global relabeling, and restart the main loop if new active nodes are found.

### 4.4.5 Intricacies with Preflows and FlowCutter

In this section, we discuss challenges we faced during the implementation that arose from using FlowCutter with preflows. The major difference to a maximum flow is that there are nodes with positive excess left.

**Source-Side Cut.** A maximum preflow only yields a sink-side cut via the reverse residual BFS, but we also need the source-side cut. We can run flow decomposition [CG97] to push excess back to the source, to obtain an maximum flow. However, flow decomposition is difficult to parallelize [BBS15]. Instead, we initialize the forward residual BFS with all non-sink excess nodes. This finds the reverse paths that carry flow from the source to the excess nodes, which is what we need.

**Sink-Side Piercing.** Furthermore, when transforming a node with positive excess to a sink, its excess must be added to the flow value. This only happens when piercing, as sink-side nodes have no excess, if they are not sinks yet.

**Maintain Distance Labels.** Finally, we want to reuse the distance labels from the previous round to avoid re-initialization overheads. However, as the labels are a lower bound on the distance from the sink, piercing on the sink side invalidates the labels. Additionally, no new excess nodes are created. In this case, we run global relabeling to fix the labels and collect the existing excess nodes, before starting the main discharge loop. When piercing on the source side the labels remain valid and new excesses are created. These are added to the active nodes and we do not run an additional global relabeling. The existing excess nodes are collected during regular global relabel runs.

### 4.4.6 Implementation Details

To facilitate an efficient and practical code, we discuss several implementation details of our parallel version of the push-relabel algorithm. This covers techniques specific to the hypergraph setting, multi-source multi-sink settings and general techniques.

**Restricting Capacities.** Recall that only bridge edges $(e_{\text{in}}, e_{\text{out}})$ have finite capacity $(\omega(e))$ in the Lawler network, while all other edges $((u, e_{\text{in}})$ and $(e_{\text{out}}, u))$ have infinite capacity. Thus, a hypernode $u$ immediately relieves all its excess to one of its incident nets $e \in \mathrm{I}(u)$ during the discharge routine. However, only a small amount of this excess is pushed over the bridging edge $(e_{\text{in}}, e_{\text{out}})$, while the remainder is pushed back to $u$. A scheme that evenly distributes the excess of a hypernode to its incident nets would improve the running time and also the available parallelism.

Since $(e_{\text{in}}, e_{\text{out}})$ is the only outgoing edge of $e_{\text{in}}$ with non-zero capacity, the flow on edges $(u, e_{\text{in}})$ is also bounded by $\omega(e)$. Adding these capacities is a trivial optimization, but it reduces running time for one flow computation on our largest instance from

over two hours to 14 seconds, when using 16 cores. It also boosts the available parallel work, since hypernodes are not immediately relieved of all their excess. Without this optimization the minimum cut contains only bridge edges, but now may contain edges $(u, e_{\text{in}})$. This matters when tracking cut hyperedges (for collecting piercing candidates), which are detected by checking if $e_{\text{in}}$ and $e_{\text{out}}$ are on different sides. Therefore, we do not check the capacity and visit $e_{\text{in}}$ nodes during forward residual BFS.

**Avoid Pushing Flow Back.** Once the correct flow value is found, the algorithm could terminate in theory since this already yields a maximum preflow. This is often achieved in very few discharging rounds. At this point flow is only pushed back to the source. We terminate once all nodes with $\text{exc}(u) > 0$ have $d(u) \geq n$, which is most often detected by global relabeling. However, we observed that the number of active nodes follows a power law distribution. Due to little work in later rounds, it takes many rounds to trigger the global relabeling step. Therefore, we perform additional relabeling, if the flow value has not changed for some rounds (500), and only few active nodes ($< 1500$) were available in each.

**Active Nodes.** The set of active nodes is implemented as an array containing the nodes and an array of insertion timestamps that are atomically set to avoid duplicates. Nodes are accumulated in thread-local buffers that are frequently flushed to a shared array. During discharging, we build the array for the next round. We insert a node if it gets pushed to, or if it has excess left after its discharge operation. After a round of discharging, we swap the previous active nodes array with the newly built one, and increment the timestamp to reset the markers.

These arrays are reused for global relabeling as well as deriving the source-side and sink-side cuts. This enables computing the cut-side weights via a simple parallel reduction over the respective arrays, which we use to decide which side to grow.

**Flow Value.** We track the flow value to abort the refinement in case it exceeds the previous cut. Traditionally in push-relabel algorithms, the flow value is determined from the excess of the sink. Since we have many sinks, we do not want to repeatedly accumulate all of their excesses. Instead, we also insert sinks into the active nodes for the next round. This way, we can add their excess deltas to the flow value during the update phase, but we of course do not discharge them in the next round.

**Hypergraph Implementation.** For performance reasons we implement the flow algorithm directly on the hypergraph, simulating the Lawler expansion without actually constructing the graph flow network. We implement three separate discharge operations that scan the pins plus the bridge edge or the incident hyperedges of a hypernode and push the appropriate amount of flow.

Chapter 5

# 5 Parallel Multilevel Hypergraph Partitioning

The previous section has introduced parallel formulations for iterative improvement algorithms used in the highest-quality sequential partitioners. These systems also implement the mutlilevel partitioning scheme and run the refinement algorithms on each level. We will now present our first parallel multilevel algorithm that is part of the shared-memory hypergraph partitioning framework Mt-KaHyPar (**M**ulti-**T**hreaded **Ka**rlsruhe **Hy**pergraph **Par**titioning).

**Algorithm Overview.** Algorithm 5.1 shows the high-level pseudocode of our multilevel algorithm. In the coarsening phase, we contract clusters of nodes that we obtain with an algorithm based on hierarchical agglomerative clustering [CA99; KK00; AK06; ACU08b; Çat+12a]. The coarsening algorithm proceeds in passes and, in each pass, we iterate in parallel over the nodes. If we visit an *unclustered* node $u$, we add it to the cluster $C \in \mathcal{C}$ that maximizes the *heavy-edge* rating function [CA99; Kar+99; HS17a]. The function prefers clusters that have a large number of heavy nets with small size with $u$ in common. Subsequently, we contract the clusters and obtain a coarser hypergraph on which we recursively repeat the coarsening process until the number of nodes would become less than $160 \cdot k$. The main challenge is to perform consistent cluster update operations as several unclustered nodes may try to join each others cluster simultaneously.

We additionally enhance the coarsening process with global information about the community structure of the hypergraph. To do so, we run a parallel community detection algorithm before coarsening (to improve readability of the pseudocode we omit this componenent in Algorithm 5.1). Similar to KaHyPar [HS17a], we then use the community structure to restrict contractions to nodes that belong to the same community.

To obtain an initial $k$-way partition, we implement parallel multilevel recursive bipartitioning and use work-stealing to handle load imbalances within the recursive partitioning calls. For bipartitioning, we use a portfolio of *nine* bipartitioning techniques to compute initial solutions [Heu15a; Sch+16a]. We run each of them several times independently in parallel and continue uncoarsening with the best bipartition out of these runs.

In the uncoarsening phase, we project the partition $\Pi$ to the next level finer hypergraph and run the parallel label propagation, FM, and flow-based refinement algorithm on each level.

**Outline.** The remainder of the section discusses the different algorithmic components of our multilevel partitioner in more detail. Section 5.1 presents the hypergraph data

**Algorithm 5.1:** Parallel Multilevel Hypergraph Partitioning

**Input:** Hypergraph $H = (V, E)$, number of blocks $k$
**Output:** $k$-way partition $\Pi$ of $H$

```
1  H_1 ← H, 𝓗 ← ⟨H_1⟩, i ← 1
2  while |V_i| > 160 · k do
3      𝒞 ← V_i                                   // Initially, each node is in its own cluster
4      for u ∈ V_i in random order do in parallel
5          if state of u is Unclustered then
6              C ← arg max_{C∈𝒞} Σ_{e∈I(u)∩I(C)} ω(e)/(|e|−1) with c(u) + c(C) ≤ c_max
7              add u to cluster C and set state of u to Clustered   // see Algorithm 5.2
8      H_{i+1} ← H_i.contract(𝒞), 𝓗 ← 𝓗 ∪ ⟨H_{i+1}⟩, i ← i + 1      // see Section 5.1
9  Π ← initialPartition(H_i, k)                  // see Section 5.3
10 for l = i to 1 do
11     Π ← project Π onto H_l
12     Π ← labelPropagationRefinement(H_l, Π)    // see Section 4.2
13     Π ← fmLocalSearch(H_l, Π)                 // see Section 4.3
14     Π ← flowBasedRefinement(H_l, Π)           // see Section 4.4
15 return Π
```

structure and parallel contraction operation. In Section 5.2, we explain how we evaluate the heavy-edge rating function and discuss the clustering and community detection algorithm. The initial partitioning algorithm based on parallel recursive bipartitioning and the portfolio of initial bipartitioning techniques are described in Section 5.3. We then explain the integration of our parallel refinement algorithms in Section 5.4 and conclude the algorithmic description with implementation details in Section 5.5. Section 5.6 discusses the configuration of the algorithm and evaluates the impact of different tuning parameters on the running time and solution quality of the algorithm. Section 5.7 concludes the chapter with a detailed experimental evaluation analyzing search conflicts, the effectiveness of the different refinement algorithms, and the scalability and running times of the different algorithmic components.

**References and Contributors.** This chapter covers our multilevel partitioning algorithm presented in Ref. [Got+21a] and improvements for initial partitioning from Refs. [Got+21c; Got+22a]. The text contains some text passages verbatim from both publications, but large parts were rewritten substantially since Ref. [Got+21a] only provides a high-level overview of the overall algorithm. We further present detailed parameter tuning experiments and an evaluation of the multilevel partitioning algorithm that goes beyond the scope of the individual publications.

The author of this dissertation contributed the coarsening, initial partitioning, and rebalancing algorithm. Lars Gottesbüren implemented the parallel community detection algorithm and several performance optimizations.

## 5.1 The Hypergraph Data Structure

The hypergraph data structure stores the incident nets $I(u)$ of each node $u \in V$ and the pin-lists of each net $e \in E$ using two adjacency arrays as illustrated in Figure 5.1. Each node $u$ and net $e$ additionally stores its weight $c(u)$ and $\omega(e)$. In the following, we explain the parallel contraction operation. The algorithm takes as input a clustering $\mathcal{C}$ and each cluster $C \in \mathcal{C}$ has a unique representative $v \in C$.

**Contraction.** Contracting a clustering into a coarser hypergraph consists of several steps. First, we remap cluster IDs to a consecutive range from 0 to $|\mathcal{C}| - 1$ by computing a parallel prefix sum on an array of size $n$ that has a one at position $v$ if $v$ is a representative of a cluster and zero otherwise. Then, we accumulate the weights of nodes in each cluster using atomic `fetch-and-add` instructions. Subsequently, we iterate over the nets in parallel, map the pins of $e$ to their new cluster IDs, and remove duplicate pins by sorting the remapped pins of each net and keeping only the first occurrence of a node.

Furthermore, we remove nets with a single pin and replace multiple identical nets with a single net with aggregated weight. We parallelize the InrSrt algorithm of Aykanat et al. [ACU08a; DKÇ13] for identical net detection. It uses *fingerprints* $\sum_{v \in e} v^2$ to eliminate unnecessary pairwise comparisons between nets. Nets with different fingerprints or different sizes cannot be identical. We distribute the fingerprints and their associated nets to the threads using a hash function. Each thread sorts the nets by their fingerprint and size, and then performs pairwise comparisons on the subranges of potentially identical nets. We aggregate the weights of identical nets at a representative and mark the others as invalid.

A parallel prefix sum over an array of size $m$ storing a one at position $e$ if $e$ is a valid net and zero otherwise yields their IDs in the coarse hypergraph. Subsequently, we iterate in parallel over the nodes $u \in V$, map the incidents nets of $u$ to the hyperedge IDs of the coarse hypergraph, and remove invalid nets. Finally, we construct the two adjacency arrays of the coarse hypergraph by computing parallel prefix sums over the sizes of the remaining nets, respectively node degrees.

## 5.2 Coarsening

The purpose of the coarsening phase is to compute a sequence of structurally similar and successively smaller hypergraphs $\mathcal{H} = \langle H_1 = H, H_2, \ldots, H_r \rangle$, to enable fast improvements in the refinement phase. We obtain a coarser hypergraph $H_{i+1}$ from the finer $H_i$ by contracting a node clustering of $H_i$. To do so, we iterate over the nodes in parallel, and each node joins the cluster that maximizes the heavy-edge rating function. Additionally, we enforce a size constraint $c_{\max} := \frac{c(V)}{160 \cdot k}$ on the weight of the heaviest cluster to facilitate finding a balanced $k$-way partition in the initial partitioning phase.

We store the cluster IDs of the nodes in an array $\mathsf{rep}$ of size $n$. For each node $u \in V$, $\mathsf{rep}[u] = v$ stores the representative of $u$'s cluster. For each representative, we maintain

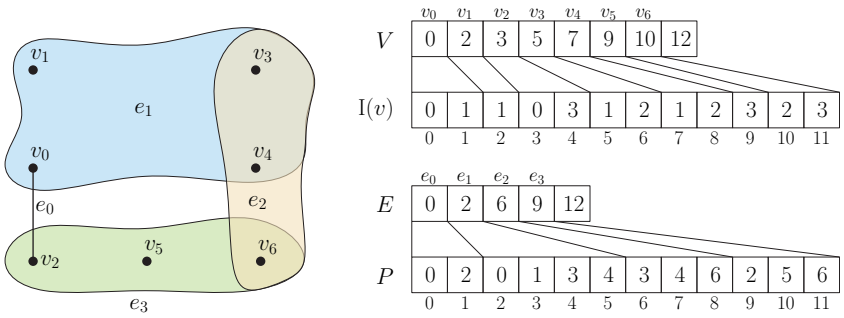



**Figure 5.1:** The hypergraph data structure used in our multilevel hypergraph partitioner.

the invariant that $\mathsf{rep}[v] = v$. Initially, each node is unclustered ($\forall u \in V : \mathsf{rep}[u] = u$). Only unclustered nodes are eligible to join multi-node clusters in the coarsening phase. Here, the main challenge is to apply consistent cluster assignments when several adjacent nodes try to join each others cluster simultaneously (e.g., when they try to cyclically join each other). A shared-memory implementation of this algorithm already exists [Çat+12a] using node locks while evaluating the rating function and updating the clustering $\mathcal{C}$ (see Section 3.4.2). Our implementation uses significantly less locking. More precisely, we do not use any locks during the evaluation of the rating function and only lock nodes when we detect that several nodes try to join each other.

In the following, we present a cache-friendly algorithm evaluating the heavy-edge rating function and a clustering algorithm resolving conflicts *on-the-fly*. Furthermore, we explain how we track the current number of contracted nodes and abort a coarsening pass once the contraction limit is reached. Finally, we present a preprocessing technique adapted from the sequential partitioner KaHyPar [HS17a] that we use to restrict contractions to *densely-connected* regions of the hypergraph.

### 5.2.1 Rating Function Evaluation

A node $u$ joins the cluster $C \in \mathcal{C}$ that maximizes the heavy-edge rating function

$$r(u, C) = \sum_{\mathrm{I}(u) \cap \mathrm{I}(C)} \frac{\omega(e)}{|e| - 1}.$$

The rating algorithm iterates over the incident nets $e \in \mathrm{I}(u)$ and aggregate the ratings to the representatives $\mathsf{rep}[v]$ of each pin $v \in e$ in a thread-local hash table. A net may contain several pins mapping to the same representative. Therefore, we use a Bloom filter, ensuring that the rating of each representative is considered at most once per net. The Bloom filter uses $10 \cdot \max_{e \in E} |e|$ bits, resulting in a false positive rate of less than 1% [Bon+06]. Afterwards, we iterate over the aggregated ratings and determine the representative $\mathsf{rep}[v]$ that maximizes $r(u, \mathsf{rep}[v])$ and fullfils the

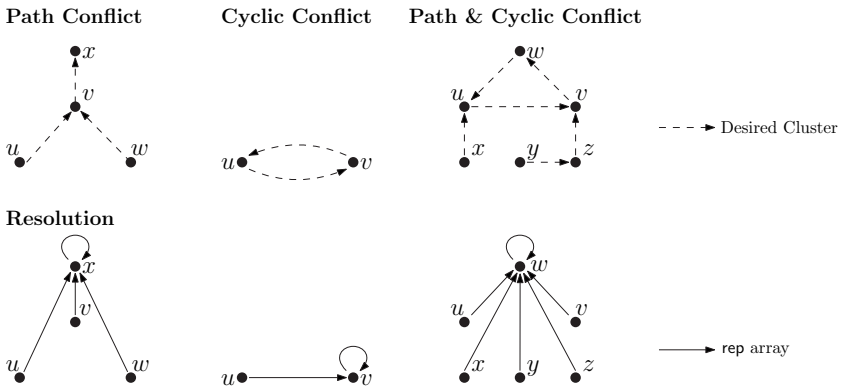



**Figure 5.2:** The different types of conflicts occuring when several nodes try to join a cluster simultaneously and their resolutions.

size-constrained $c(u) + c(\mathsf{rep}[v]) \leq c_{\max}$. In case of ties, we prefer singleton clusters. If two (un)clustered representatives with the same rating exist, we choose one uniformly at random. The algorithm then returns the node $\mathsf{rep}[v]$ that maximizes $r(u, \mathsf{rep}[v])$, which we also call the *contraction partner* of $u$. Note that the representative of a node can change during the evaluation of the rating function since we do not lock the nodes. We use an optimistic strategy assuming that conflicts happen rarely and are acceptable in practice.

Each thread uses a hash table to aggregate the ratings. Their size is fundamental for the scalability of our parallel coarsening algorithm since unnecessarily large hash tables can considerably increase the number of cache misses on shared caches. Therefore, we use fixed-capacity linear probing hash tables with $2^{15}$ entries and resort to a larger hash table if the expected fill ratio exceeds $1/3$ of the capacity. We conservatively estimate the fill grade as the sum of the sizes of incident nets. The larger hash table uses $6 \cdot 10^5$ entries and, if used, we sample the first $2 \cdot 10^5$ neighbors for the ratings.

### 5.2.2 Clustering Algorithm

Once a node $u$ chooses a contraction partner $v$ (representative of a cluster $C$), we have to set $\mathsf{rep}[u] = v$. Since several nodes can join clusters simultaneously, there may occur conflicts that must be resolved. As illustrated in Figure 5.2, there are two types of conflicts: *path* and *cyclic* conflicts . A path conflict involves several nodes $u_1, \ldots, u_l$ and occurs when each node $u_i$ tries to join $u_{i+1}$. In a cyclic conflict, the last node $u_l$ additionally tries to join $u_1$. It is also possible that a combination of both conflicts occurs, as illustrated in Figure 5.2 (right). We can resolve a path conflict when each node $u_i$ waits until $u_{i+1}$ has joined its desired target cluster. Afterwards, we can set $\mathsf{rep}[u_i] = \mathsf{rep}[u_{i+1}]$ to resolve the conflict. However, applying this resolution scheme to cyclic conflicts would result in a deadlock. Therefore, the threads must agree on a cluster join operation that breaks the cycle and reduces it to a path conflict. In

**Algorithm 5.2:** Clustering Algorithm

**Input:** A node $u$ and its contraction partner $v$ ($u$ joins the cluster of $v$'s representative)

```
1  Function addNodeToCluster(u, v)                                  // Helper Function
2      fetch-and-add(c(v), c(u))
3      if c(v) ≤ c_max then rep[u] ← v
4      else fetch-and-sub(c(v), c(u))
5  if c(u) + c(v) ≤ c_max and compare-and-swap(state[u], Unclustered, Joining) then
6      if state[v] = Clustered then
7          v ← rep[v], addNodeToCluster(u, v)
8      else if compare-and-swap(state[v], Unclustered, Joining) then
9          addNodeToCluster(u, v), state[v] ← Clustered
10     else
           // wait until state of v changes to Clustered
11         while state[v] = Joining do
12             if cyclic conflict detected and u is node with smallest ID in cycle then
13                 addNodeToCluster(u, v)                            // resolves cyclic conflict
14                 state[u] ← Clustered, state[v] ← Clustered, break
           // state of u may have changed due to resolving a cyclic conflict
15         if state[u] = Joining then v ← rep[v], addNodeToCluster(u, v)
16     state[u] ← Clustered
```

the following, we present the clustering algorithm outlined in Algorithm 5.2 resolving these conflicts on-the-fly.

We associate each node with one of the following three states: *unclustered*, currently *joining* a cluster, or *clustered*. We use `compare-and-swap` instructions to ensure consistency of node states. The clustering algorithm takes a node $u$ and its contraction partner $v$ as input, where $v$ is a representative of a cluster $C$. We note that the representative of cluster $C$ may have changed after the evaluation of the rating function, but its representative is stored in $\mathsf{rep}[v]$. The algorithm starts by setting the state of $u$ to joining. If $v$ is already clustered or we succeed in setting the state of $v$ to joining, we can safely set $\mathsf{rep}[u] = \mathsf{rep}[v]$ and set the state of $u$ and $v$ to clustered. In the former case, $v$ and its representative $\mathsf{rep}[v]$ are clustered and are not considered by the coarsening algorithm anymore. In the latter case, $v$ is unclustered and by setting its state to joining, the thread acquires exclusive ownership for modifying $\mathsf{rep}[u]$ and $\mathsf{rep}[v]$.

If $v$ is currently trying to join another cluster, we spin in a busy-waiting loop until the state of $v$ is updated to clustered by some other thread, and then join its new cluster (resolves path conflicts). In the busy-waiting loop, we additionally check if $u$ is part of a cycle of nodes trying to join each other. To detect a cyclic conflict, each node

writes its desired target cluster into a globally shared vector and checks if this induces a cycle. If so, the node with the smallest ID in the cycle gets to join its desired cluster, thus breaking the cycle.

We still have to show that no other cluster join operation can induce an additional cycle and may produce inconsistencies during conflict resolution. In the following, we say that a directed graph $G$ is *weakly connected* if the corresponding undirected graph (each directed edge is replaced with an undirected edge) is connected. A *weakly connected component* of $G$ is a weakly connected subgraph of $G$.

**Lemma 5.1**

*Let $G$ be the directed graph induced by all cluster join operations spinning in the busy-waiting loop. It holds that each weakly connected component of $G$ has at most one cycle.*

*Proof.* We assume that $G$ has at least one cycle and is *weakly connected*. If $G$ consists of more than one weakly connected component, then there are several independent conflicting states, and we can apply the following argument to each individually. Each node in $G$ has an out-degree of one since each node can join at most one cluster. Thus, $G$ has exactly $n$ edges. If we remove one edge of a cycle in $G$, $G$ is still weakly connected and has $n-1$ edges. A connected undirected graph with $n$ nodes and $n-1$ edges is a tree. Thus, a weakly connected graph with $n-1$ directed edges does not contain any cycles. Consequently, $G$ has at most one cycle. □

If we add a node $u$ to the cluster represented by $v$, we update the weight of $v$ to $c(v) + c(u)$ with an atomic `fetch-and-add` instruction. If $c(v) \leq c_{\max}$, we set $\mathsf{rep}[u] = v$. Otherwise, we reject the cluster join operation.

### 5.2.3 Contraction Limit

We stop coarsening once the number of nodes of the coarsest hypergraph drops below $160 \cdot k$. However, if we are already close to the contraction limit and perform an additional coarsening pass, the number of nodes may be significantly less than the desired $160 \cdot k$ nodes. Therefore, we additionally track the number of contracted nodes in a coarsening pass and abort if we reach the contraction limit. A simple solution to achieve this is to increment a globally shared counter $n'$ via a `fetch-and-add` instruction each time a node joins a cluster. However, this may cause contention on the shared counter $n'$. Therefore, we choose a different approach.

The idea is to update the globally shared counter $n'$ only if we are close to the contraction limit. Let $t$ be the number of threads, $n'_i$ be the number of locally contracted nodes of thread $i$, and $x$ the desired number of nodes of the coarsest hypergraph. Each thread $i$ uses a local contraction limit $s_i$, which we initially set to $\frac{n-x}{t}$. If $n'_i$ equals $s_i$, thread $i$ updates the globally shared counter $n'$ to $\sum_{1 \leq i \leq t} n'_i$ and sets its new local contraction limit to $s_i + \frac{n'-x}{t}$. Assume that thread $i$ is the first

thread that reaches its local contraction limit $s_i$. Then, it follows that

$$\sum_{1 \leq i \leq t} n'_i \leq \sum_{1 \leq i \leq t} s_i = \sum_{1 \leq i \leq t} \frac{n - x}{t} = n - x,$$

which is an upper bound for the current number of contracted nodes. Therefore, the coarsest hypergraph has at least $x$ nodes at this point. The new local contraction limit ensures that we update $n'$ more frequently if we are closer to the contraction limit $x$.

In addition to the global contraction limit that terminates the coarsening phase, we use a hierarchy contraction limit on each level to prevent coarsening from reducing the size of the hypergraph *too* aggressively. We abort a coarsening pass as soon as the clustering algorithm reduces the number of nodes by more than a factor of 2.5. Furthermore, the weight constraint $c_{\max}$ on coarse nodes can lead to passes in which only a few nodes are contracted. If a pass reduces the number of nodes by less than 1%, we still perform the remaining contraction and then proceed to initial partitioning, even if the $160 \cdot k$ node limit is not reached.

### 5.2.4 Community Detection Enhancement

A main design goal of the coarsening phase is that the coarser approximations should be *structurally similar* to the input hypergraph. Ideally, the coarsest hypergraph should have the same min-cut properties as the original hypergraph. Consequently, a coarsening algorithm should avoid contractions between nodes incident to nets of the minimum cut. However, identifying those nets is NP-complete. Therefore, the sequential partitioner KaHyPar [HS17a] uses community detection on the bipartite graph representation of the hypergraph [SK72] to identify *densely-connected* node clusters. Contractions are then only allowed between nodes in the same cluster. This preserves some of the global structure in coarsening phase and substantially improves the quality of both the initial and the final partition [HS17a].

We also integrate the approach of KaHyPar into our multilevel partitioner. We run the algorithm as a preprocessing step before the coarsening phase and then use the community structure to restrict contractions to nodes in the same community on each level. The community detection algorithm consists of two steps: transforming the hypergraph into its bipartite graph representation and then running the Louvain algorithm [Blo+08] for modularity maximization, a widely used objective function for community detection [NG04; Bra+08].

Note that the modularity objective function can also be directly defined on hypergraphs [Kam+19]. In a bachelor thesis [Kra21a], which we supervised, we implemented such an approach instead of using the bipartite graph representation. However, it was considerably slower and did not significantly improve the solution quality compared to the approach described in the following.

**Graph Transformation.** We use an adjacency array to represent the graph and transform a hypergraph into its bipartite graph representation by computing two parallel prefix sums over the node degrees and hyperedge sizes. Afterwards, we iterate

in parallel over all nodes and nets and copy their incident nets and pins to the adjacency array. Similar to KaHyPar [HS17a], we additionally redefine the weight of each edge based on the density $d := \frac{m}{n}$ of the input hypergraph. Let $(u, e)$ be an edge of the bipartite graph where $u$ is a pin of net $e$. Then, $\omega(u, e)$ is defined as follows:

$$\omega(u,e) := \begin{cases} \frac{d(u)}{|e|} & \text{if } d < 0.75 \\ 1 & \text{otherwise} \end{cases}$$

**Parallel Community Detection.** We implement the parallel Louvain method (PLM) of Staudt and Meyerhenke [Blo+08; SM16] for modularity maximization. Let $G = (V, E, \omega)$ be a weighted graph and $\mathcal{C}$ a clustering of $G$. Then, the modularity of $\mathcal{C}$ for a graph $G$ is defined as:

$$Q(G, \mathcal{C}) := \sum_{C \in \mathcal{C}} \frac{\omega(C)}{\omega(E)} - \frac{vol(C)^2}{4\omega(E)^2}$$

Here, $\omega(C)$ is the total weight of the internal edges of $C$ and $vol(C) := \sum_{u \in C} vol(u)$, where $vol(u) := \sum_{v \in \Gamma(u) \setminus \{u\}} \omega(u, v) + 2\omega(u, u)$ is the weighted degree of node $u$.

The PLM algorithm works in rounds (similar to the label propagation algorithm). Initially, each node is in its own cluster. In each round, we iterate in parallel over the nodes and move a node to the cluster in its neighborhood with the highest positive modularity improvement. The change in modularity when we move a node $u$ from its current cluster $C$ to a target cluster $D$ can be expressed as follows:

$$\begin{aligned} \Delta Q_u(C, D) &:= \frac{1}{\omega(E)} (\Delta Q(u, D) - \Delta Q(u, C)) \\ &= \frac{1}{\omega(E)} \left( \omega(u, D \setminus \{u\}) - \frac{vol(D \setminus \{u\}) vol(u)}{2\omega(E)} \right) \\ &\quad - \frac{1}{\omega(E)} \left( \omega(u, C \setminus \{u\}) - \frac{vol(C \setminus \{u\}) vol(u)}{2\omega(E)} \right) \end{aligned}$$

Here, $\omega(u, C) := \omega(\{(u, v) \in E \mid v \in C\})$ is the weight of the edges connecting $u$ with cluster $C$. Our PLM algorithm stores the weighted degrees $vol(u)$ of each node $u \in V$ and the cluster volumes $vol(C)$ of each cluster $C \in \mathcal{C}$. To find the cluster $D$ that maximizes $\Delta Q_u(C, D)$ for a node $u \in C$, we compute the $\omega(u, D \setminus \{u\})$ terms for all adjacent clusters of $u$ each time from scratch. To do so, we iterate over its neighbors $v \in \Gamma(u)$ and aggregate the edge weights to adjacent clusters using a thread-local hash table. If $\max_{D \in \mathcal{C}} \Delta Q(u, D) > \Delta Q(u, C)$, we move $u$ from its current cluster $C$ to $D$ and update the cluster volumes $vol(C)$ and $vol(D)$ using atomic `fetch-and-add` instructions. As in the coarsening phase, we use two hash tables with different sizes to aggregate ratings and resort to the larger hash table ($6 \cdot 10^5$ entries) if the number of neighbors $|\Gamma(u)|$ of a node $u$ exceeds 1/3 of the capacity of the smaller hash table ($2^{15}$ entries).

The algorithm proceeds for up to 5 rounds or until less than 1% of the nodes are moved in a round. If any node was moved, the process is repeated recursively on the graph obtained from contracting the clusters.

## 5.3 Initial Partitioning

In the parallel partitioning literature, many authors argue that the coarsest hypergraph is small enough to use sequential initial partitioning algorithms [KK96; WC00b; TK04a; TK04b; Dev+06; HSS10; ASS17]. This assumption holds for most of the instances, but not in general. The size-constrained $c_{\max}$ that bounds the weight of the heaviest node in the coarsest hypergraph may prevent coarsening algorithms from reaching the contraction limit. This happens mainly in complex networks with power-law node degree distributions. Here, the high-degree nodes reach the maximum allowed node weight quickly because they are attractive for many low-degree nodes. This can prevent further contractions in the coarsening phase. In the experimental evaluation of the shared-memory graph partitioner Mt-KaHIP, the initial partitioning phase has the largest share on the total partitioning time for many instances [Akh19, p. 134]. Furthermore, the number of hyperedges in a hypergraph is bounded by $|\mathcal{P}(V)| = 2^{|V|}$, unlike graphs, where the number of edges is bounded by $|V|^2$. Therefore, the coarsest hypergraph may contain many large nets [Kab+17] even if we reach the contraction limit. Hence, it is inevitable to parallelize the initial partitioning phase.

Instead of performing sequential initial partitioning, one can use parallel multilevel recursive bipartitioning. The approach evenly splits the thread pool into two subgroups which then independently process the two recursive partitioning calls in parallel [CP08; LaS+15]. However, this can lead to load imbalances when one block contains significantly more pins respectively edges than the other block. Complex networks are again affected by this. Such networks usually consist of a few densely-connected communities, while other components are only loosely connected to them. A good partition most likely places the denser parts of the hypergraph into one block inducing load imbalances in one of the recursive partitioning calls.

We therefore use parallel multilevel recursive bipartitioning with *work-stealing* to account for load imbalances. To obtain an initial bipartition within the recursive partitioning calls, we implemented the same portfolio of flat bipartitioning algorithms as used in KaHyPar [Sch+16a; Sch20]. In the following, we describe our recursive bipartitioning algorithm and portfolio approach in more detail.

### 5.3.1 Parallel Recursive Bipartitioning

We compute initial $k$-way partitions via parallel recursive bipartitioning using our multilevel partitioner shown in Algorithm 5.1 (initialized with $k = 2$). When calling our partitioning algorithm for $k = 2$, we replace the initial partitioning call with a portfolio of initial bipartitioning techniques to obtain an initial solution (described in the next section). Moreover, we do not use flow-based refinement as it did not improve the overall solution quality (see Section 5.6).

Once we obtain a bipartition $\Pi = \{V_1, V_2\}$ of the coarsest hypergraph $H$, we extract the subhypergraphs $H_{V_1}$ and $H_{V_2}$ and recurse on both in parallel by partitioning $H_{V_1}$ into $\lceil \frac{k}{2} \rceil$ and $H_{V_2}$ into $\lfloor \frac{k}{2} \rfloor$ blocks. We ensure that the final $k$-way partition obtained via recursive bipartitioning is balanced by adapting the imbalance ratio for

each bipartition individually [Sch+16a]. Let $H_{V'}$ be a subhypergraph that should be recursively partitioned into $k' \leq k$ blocks. Then,

$$\varepsilon' := \left( (1+\varepsilon) \frac{c(V)}{k} \cdot \frac{k'}{c(V')} \right)^{\frac{1}{\lceil \log_2 k' \rceil}} - 1 \tag{5.1}$$

is the imbalance ratio used for the bipartition of $H_{V'}$. The equation is based on the observation that the worst-case block weight of the resulting $k'$-way partition of $H_{V'}$ obtained via recursive bipartitioning is smaller than $(1+\varepsilon')^{\lceil \log_2(k') \rceil} \frac{c(V')}{k'}$ if $\varepsilon'$ is used for all further bipartitions. Requiring that this weight must be smaller or equal to $L_{\max} = (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ leads to the formula defined in Equation 5.1.

**Implementation Details.** We parallelize our algorithms with the shared-memory library `Intel TBB` [Phe08]. `TBB` provides an internal task scheduler implementing a work-stealing approach (see Section 2.3.2). Hence, we balance the computational work within the recursive partitioning calls by simply using the parallel primitives of `TBB`.

### 5.3.2 Flat Initial Bipartitioning

We use a portfolio of nine different flat bipartitioning algorithms to compute an initial bipartition, including (greedy) hypergraph growing [KK98a; CA99; Kar+99; ÇA11; Sch13; Sch+16a; Sch20], random assignment [Kar+99; VB05; ÇA11; Sch+16a; Sch20], and label propagation initial partitioning [Sch+16a; Sch20]. We run each algorithm at least 5 and at most 20 times. After 5 runs, we only run an algorithm again if it is likely to improve the best solution $\Pi^*$ found so far. We estimate this based on the arithmetic mean $\mu$ and standard deviation $\sigma$ of the connectivity values achieved by that algorithm so far, using the 95% rule. Assuming the connectivity values follow a normal distribution, roughly 95% of the runs will fall between $\mu - 2\sigma$ and $\mu + 2\sigma$. If $\mu - 2\sigma > \mathfrak{f}_{\lambda-1}(\Pi^*)$, we do not run the algorithm again. We run each algorithm independently in parallel, and each bipartition is refined using sequential 2-way FM refinement [FM82]. We continue uncoarsening using the bipartition with the best connectivity value. In case of ties, we choose the bipartition with the best balance. In the following, we will briefly outline the different flat bipartitioning algorithms used in the portfolio.

**Random Assignment.** Random initial bipartitioning iterates over the nodes and assigns each node to a block chosen uniformly at random. If the assignment of a node violates the balance constraint, we try to add it to the opposite block. If this also fails, the node is randomly assigned to one of the blocks.

**Seed Nodes.** The following initial bipartitioning algorithms grow the two blocks of a bipartition starting from two seed nodes that are *far* away from each other [GL81]. To compute them, we start a breadth-first search (BFS) from a node chosen uniformly at random. The last node $u$ touched by the search is our first seed node. Afterwards, we perform a second BFS starting from $u$, and the last node touched by this search is our second seed node. We will call such nodes *pseudo-peripheral seed nodes*.

**BFS-based Partitioning.** The algorithm grows the two blocks of the bipartition using two breadth-first searches. For each block, we use one queue and initialize it with one pseudo-peripheral seed node. We then alternate between the two queues and assign the next node $u$ in the queue to its corresponding block if it does not violate the balance constraint. Afterwards, we push the neighbors of $u$ into the queue of the block to which $u$ was assigned. If the queue of a block becomes empty, we assign a random node to the block (ignoring the balance constraint) and insert its unassigned neighbors into the corresponding queue.

**Greedy Hypergraph Growing.** The algorithm maintains a priority queue (PQ) for each block, storing adjacent nodes of the growing block according to a gain function. We initialize the PQs with two pseudo-peripheral seed nodes. The algorithm then assigns the node with the highest gain from one of the two PQs to its corresponding block and either pushes its neighbors into the PQ or updates their gain values if already contained.

We use the FM gain [FM82] and the *max-net* gain definition [ÇA11]. The FM gain prefers node moves with the least increase in cut size if assigned to the growing block. The max-net gain of a node $u$ is the weight of the nets $e \in \mathrm{I}(u)$ incident to the growing block. Furthermore, we use three different policies to select the next move from the PQs. The *sequential* policy first grows block $V_1$ and then block $V_2$. The *global* policy assigns the node with the highest gain to one of the blocks in each step. The *round-robin* policy alternates between the PQs and assigns the node with the highest gain from the currently considered PQ to its corresponding block. In total, we use two different gain definitions and three PQ selection policies, resulting in *six* variants of the greedy hypergraph growing algorithm.

**Label Propagation.** The algorithm initially selects two pseudo-peripheral seed nodes and five of their neighbors and assigns them to one of the two blocks. Afterwards, it runs the size-constrained label propagation algorithm [MSS14; Heu15a] until none of the nodes changed their block in a round or a predefined number of rounds is reached. If a node is visited, it is moved to the block with the highest FM gain. If the node is not adjacent to any block, it stays unassigned in the current round. If there are still unassigned nodes at the end, they are assigned one-by-one to the block with minimum weight.

## 5.4 Refinement

In the uncoarsening phase, we revert the contractions level-by-level by projecting the partition to the next level finer hypergraph. To do so, we iterate over the nodes of the finer hypergraph in parallel and assign them to the block of their corresponding constituent in the coarse graph. Afterwards, we run the parallel label propagation, FM, and flow-based refinement algorithms described in Sections 4.2 – 4.4.

While our refinement algorithms are guaranteed to produce balanced solutions, it turns out that intermediate balance violations in the FM algorithm improve solution

quality. The intuition behind this is that localized FM searches may each find good improvements, but their combination is barely infeasible. Hence, we relax the balance constraint for the second FM phase (the global rollback step) by using the imbalance ratio $\varepsilon' = 1.25 \cdot \varepsilon$. If the partition is still imbalanced on the finest level, we rebalance it using an approach that is similar to label propagation. We iterate over the nodes in parallel. If a node is in an overloaded block and can be moved with non-negative gain, we perform the move immediately. Negative-gain moves are collected in thread-local priority queues. If the partition is still imbalanced after performing all non-negative gain moves, the threads perform the moves with the smallest increase in connectivity from their priority queues in a second step.

In the experiments on benchmark set $\mathrm{M_{HG}}$, the rebalancing component was triggered on 2494 out of the 34160 instances (7.3%). The algorithm did not worsen the connectivity metric on 25.7% of the rebalanced instances, while the relative deterioration of the solution quality was less than 0.25% on 92.9% of all runs. The rebalancing component worsened the solution quality on five instances by more than 20%. However, this did not occur for all repetitions (we perform ten repetitions per instance). Hence, the impact of the rebalancing step on the overall solution quality is negligible.

## 5.5 Engineering Aspects

In the following, we outline engineering aspects that are necessary to enhance the performance of our system.

**Memory Allocation.** Memory allocations are a significant bottleneck, which is why we implement a custom memory pool. We estimate the memory needed by our data structures, categorized by the phases of the multilevel scheme, and then allocate for the peak memory usage across all phases. At the end of a phase, we pass the memory along to the next phase, and initialize data structures in parallel. Further, we use the `TBB` scalable allocator [Phe08] for concurrent memory allocations.

**Large Nets.** We often iterate over the incident nets of a node and then over the pins of each incident net, e.g., to aggregate ratings for coarsening, to make neighbors eligible to move in the next round of label propagation, or to insert them into a PQ for FM. To improve performance, we skip nets that exceed a size threshold – 100 for label propagation, and 1000 for FM and coarsening. Furthermore, we completely remove nets with size greater than $\max(0.01 \cdot |V|, 5 \cdot 10^4)$ before partitioning.

**Fast I/O.** While evaluating other parallel partitioning algorithms, we found that the time required to read the input file and construct the (hyper)graph data structure was often significantly larger than the actual partitioning time. The main issue is that some tools perform these steps sequentially and use expensive streaming primitives of `C++` to parse the input file. Although we do not measure the time for I/O operations in our experiments, we also parallelize and optimize this step to improve the usability of our system.

The input for our partitioning algorithm is expected in hMetis file format [KK98b]. The first line contains the number of nodes $n$ and hyperedges $m$, followed by $m$ lines containing the pins of each hyperedge. We use `nmap` to map the input file to the main memory. Afterwards, we perform a sequential pass over the input and divide it into several chunks that are then processed by the different threads to construct the hypergraph data structure in parallel.

## 5.6 Algorithm Configuration

We provide two configurations of our multilevel partitioning algorithm: Mt-KaHyPar-D (-**D**efault) and Mt-KaHyPar-D-F (-**D**efault-**F**lows). Mt-KaHyPar-D-F extends Mt-KaHyPar-D with flow-based refinement. Mt-KaHyPar-D aims to compute good partitions very fast, while Mt-KaHyPar-D-F aims for high solution quality.

This section discusses the different parameter choices in our algorithm which we summarize in Table 5.1. Note that each multilevel partitioner has many different configuration options, with some parameters depending on each other. Our partitioner is no exception. We therefore refrain from a detailed evaluation of each parameter and restrict ourselves to algorithmic choices having a significant impact on the performance of our multilevel partitioner. Some non-evaluated parameters were introduced for handling special cases on some instances and had no measurable impact on the overall performance (e.g., sampling threshold, minimum shrink factor, bulk piercing), while others were taken from KaHyPar's configuration [Sch20] (e.g, maximum allowed node weight, contraction limit, rating function).

The following experiments are conducted on benchmark set $M_P$ using 10 threads of machine A. In each experiment, we vary one parameter and use the default values depicted in Table 5.1 for all others.

**Table 5.1:** Algorithm configuration of Mt-KaHyPar-D(-F).

| Parameter | Value | Description | Reference |
|---|---|---|---|
| **Community Detection** | | | Section 5.2.4 |
| Louvain Rounds | 5 | Maximum number of rounds per level | |
| Mininum Number of Moved Nodes | 1% | Minimum number of moved nodes per round relative to the total number of nodes | |
| Sampling Threshold | 200.000 | We sample the neighbors with a degree larger than this threshold when aggregating ratings | |
| **Coarsening** | | | Section 5.2 |
| Rating Function | Heavy-Edge | | Line 6 in Algorithm 5.1 |

| | | | |
|---|---|---|---|
| Contraction Limit | $160 \cdot k$ | We abort coarsening when the number of nodes drops below this threshold | Section 5.2.3 |
| Maximum Node Weight | $\frac{c(V)}{160 \cdot k}$ | Maximum allowed weight of a node in the coarsest hypergraph | Line 3 in Algorithm 5.2 |
| Hierarchy Contraction Limit | 2.5 | We abort a coarsening pass when we shrink the hypergraph by a factor of 2.5 | Section 5.2.3 |
| Minimum Shrink Factor | 1% | If we reduce the size of a hypergraph only by 1% on a level, we abort coarsening | Section 5.2.3 |
| Sampling Threshold | 200.000 | We sample the neighbors with a degree larger than this threshold when aggregating ratings | Section 5.2.1 |
| **Initial Partitioning** | | | Section 5.3 |
| Minimum IP Runs | 5 | We run each flat bipartitioning algorithm at least 5 times | Section 5.3.2 |
| Maximum IP Runs | 20 | We run each flat bipartitioning algorithm at most 20 times | Section 5.3.2 |
| IP Refinement | (+LP,+FM,-F) | During uncoarsening, we use the parallel label propagation (+LP) and FM (+FM) algorithm but no flow-based refinement (-F) | Section 5.3 |
| **Refinement** | | | Section 5.4 |
| **Label Propagation** | | | Section 4.2 |
| LP Rounds | 5 | Maximum number of rounds | Line 5 in Algorithm 4.5 |
| **Multi-Try FM Algorithm** | | | Section 4.3 |
| Multi-Try FM Rounds | 10 | Number of Multi-Try FM Rounds | Line 1 in Algorithm 4.6 |
| Seed Nodes | 25 | We initialize each localized FM search with 25 seed nodes | Line 5 in Algorithm 4.6 |
| Balance Violation Factor | 1.25 | During global rollback, we relax the imbalance ratio to $\varepsilon' = 1.25 \cdot \varepsilon$ | Section 5.4 |
| **Flow-Based Refinement** (only for Mt-KaHyPar-D-F) | | | Section 4.4 |
| Flow Region Scaling Factor | 16 | If we grow a region $B = B_1 \cup B_2$ around the cut of a bipartition then $c(B_1) \leq (1 + 16 \cdot \varepsilon)\lceil \frac{c(V)}{2} \rceil - c(V_2)$ (analogously for $B_2$) | Section 4.4.2 |
| $\tau$ | 1 | We use $\min(t, \tau \cdot k) = \min(t, k)$ threads to process the active block pairs of the quotient graph in parallel | Section 4.4.1 |
| Maximum BFS Distance | 2 | We restrict the distance of each node in the region $B$ to the cut nets of a bipartition to at most 2 | Section 4.4.2 |
| Bulk Piercing | on | We use the bulk piercing feature | Section 4.4.3 |

| | | | |
|---|---|---|---|
| Time Limit Factor | 8 | We abort a flow problem when it takes longer than $8\overline{t_f}$ seconds where $\overline{t_f}$ is the average running time of all previously solved flow problems | Section 4.4.1 |

**Maximum Number of Flat Initial Biartitioning Runs (Mt-KaHyPar-D).** We run each flat bipartitioning algorithm at least 5 and at most 20 times. After 5 runs, we only run an algorithm again if it is likely to improve the best solution found so far. In this experiment, we evaluate the impact of the second parameter, the maximum number of runs per flat initial bipartitioning algorithm, on the solution quality and running time of Mt-KaHyPar-D.

Figure 5.3 shows the performance profiles comparing the solution quality for different choices of the parameter. The partitions produced by Mt-KaHyPar-D with at most 20 runs are better than those with 2, 5 and 10 runs by 2.1% (using 2 runs is faster by a factor of 1.5 on average), 0.9% (1.3) and 0.4% (1.16) in the median, respectively. If we increase the number of initial bipartitioning runs from at most 20 to 50 runs, we can still improve the connectivity metric by 0.4% in the median, but it slows down the running time of Mt-KaHyPar-D by a factor of 1.33 on average. We therefore run each flat bipartitioning algorithm at most 20 times as it provides a good tradeoff between quality and running time.

**Initial Partitioning Refinement Configuration (Mt-KaHyPar-D).** In the initial partitioning phase, we use multilevel recursive bipartitioning and run our parallel label propagation (LP), FM, and flow-based refinement (F) algorithms to improve a bipartition in the uncoarsening phase. We will now evaluate whether it is necessary to run all refinement algorithms or if a subset of them leads to the same solution quality. In the following, we will denote a refinement configuration with (+LP,+FM,-F) where $+$ or $-$ indicates whether or not we run the corresponding refinement algorithm.

The performance profiles in Figure 5.4 show that the differences in the connectivity metric produced by the different refinement configurations are only marginal. The running times of the configurations (+LP,-FM,-F) and (+LP,+FM,-F) are comparable ((+LP,+FM,-F) is 1% slower on average). However, enabling flow-based refinement (+LP,+FM,+F) slows down the overall partitioning time compared to (+LP,+FM,-F) by 10% on average. We therefore use the parallel label propagation and FM algorithm in the initial partitioning phase but no flow-based refinement.

**Multi-Try FM Rounds (Mt-KaHyPar-D).** Our parallel FM algorithm works in rounds, and each round starts highly-localized FM searches around a few seed nodes until each node is moved at most once. This experiment determines the number of rounds required to achieve a good tradeoff between solution quality and running time.

Figure 5.5 (right) shows the performance profile comparing the solution quality of Mt-KaHyPar-D with an increasing number of multi-try FM rounds. We can see that the solution quality consistently increases until 10 rounds. Using more than 10 rounds only leads to minor improvements (increasing the number of rounds from 10 to 20 improves the connectivity metric only by 0.3% in the median). The configuration

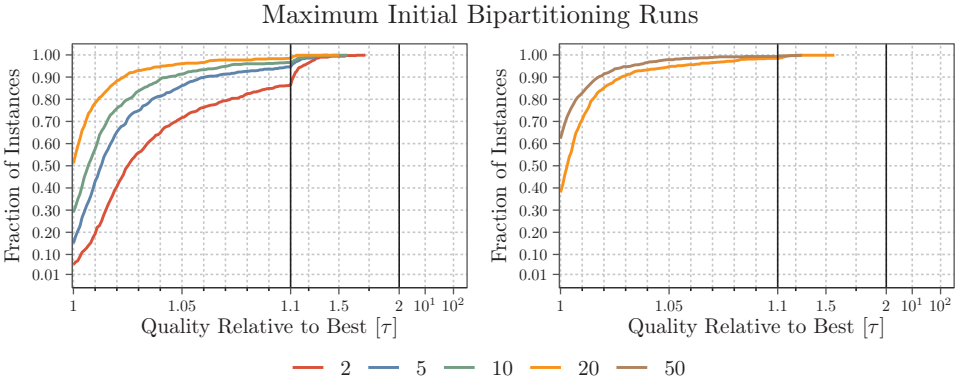



**Figure 5.3:** Performance profile comparing the solution quality of Mt-KaHyPar-D with at most $\leq 20$ (left) and $\geq 20$ (right) runs per flat initial bipartitioning algorithm.

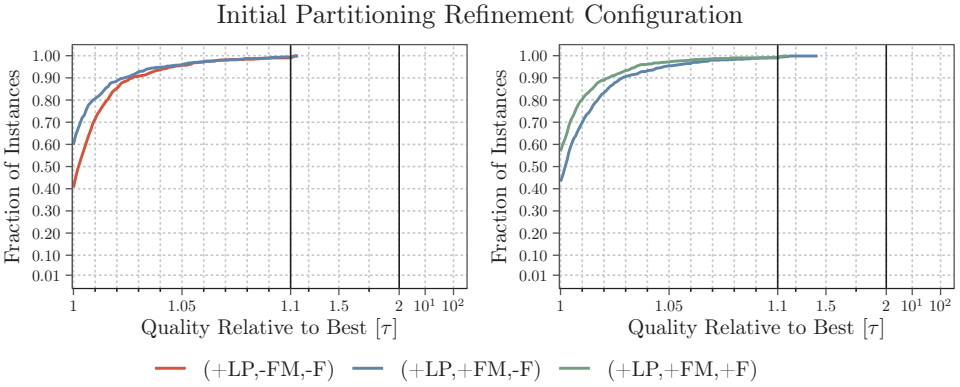



**Figure 5.4:** Performance profile comparing the solution quality of Mt-KaHyPar-D with different refinement configurations in the initial partitioning phase.

with 10 rounds is slower than with 1 and 5 round(s) by a factor of 1.18 and 1.07, but faster than 20 rounds by a factor of 1.07 on average. We therefore set the number of multi-try FM rounds to 10.

Our parallel label propagation algorithm also proceeds in rounds. As it can be seen in Figure 5.5 (left), we do not observe any further improvements with 5 or more rounds. This can be explained by the fact that the FM algorithm can escape from local optima by also performing negative gain moves. Since we move each node at most once in an FM round, we may not find all possible improvements due to already moved nodes in the neighborhood of the search region. The regions in which the label propagation algorithm can find improvements in subsequent rounds are restricted to

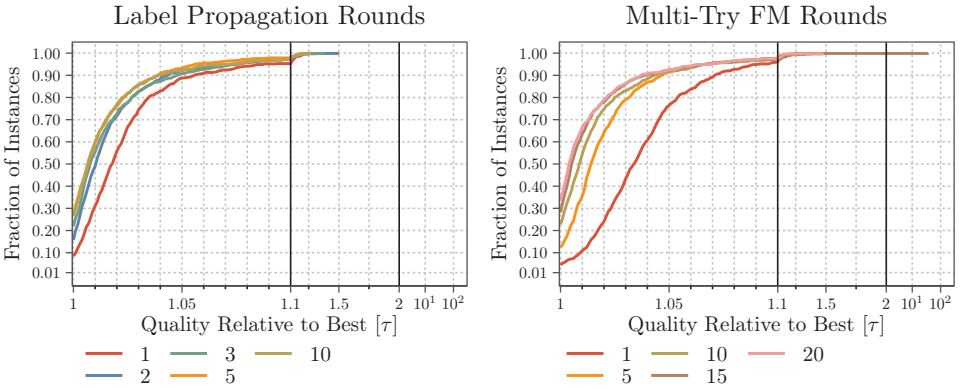



**Figure 5.5:** Performance profile comparing the solution quality of Mt-KaHyPar-D with different numbers of label propagation rounds (left) and multi-try FM rounds (right).

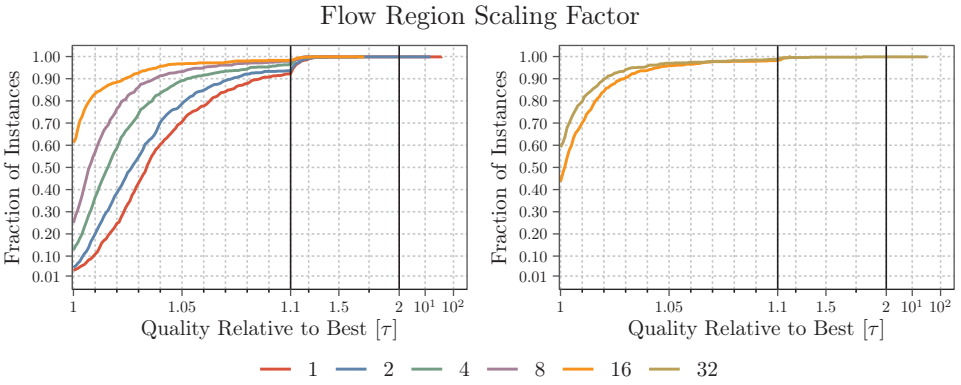



**Figure 5.6:** Performance profile comparing the solution quality of Mt-KaHyPar-D-F with flow region scaling factor $\alpha \leq 16$ (left) and $\alpha \geq 16$ (right).

neighbors of moved nodes (since it only performs positive gain moves). Therefore, the FM algorithm profits more from multiple restarts since different execution orders allow the localized FM searches to expand to regions of previously locked nodes.

**Flow Region Scaling Factor (Mt-KaHyPar-D-F).** The flow region scaling factor $\alpha$ determines the size of the region $B$ containing the nodes part of the flow network. Larger values for $\alpha$ lead to larger flow problems with potentially smaller minimum cuts but also increase the likelihood of violating the balance contraint.

Figure 5.6 illustrates the impact of the flow region scaling factor on the solution quality of Mt-KaHyPar-D-F. The partitions produced by Mt-KaHyPar-D-F with $\alpha = 16$

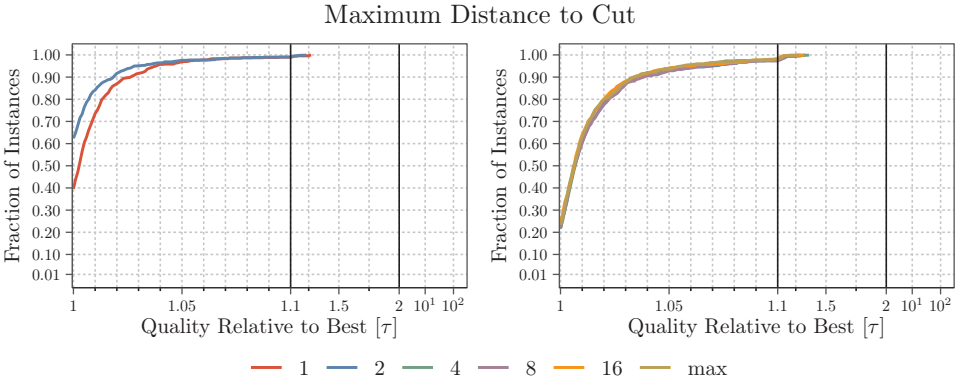



**Figure 5.7:** Performance profile comparing the solution quality of Mt-KaHyPar-D-F when we restrict the distance of each node to the cut hyperedges of a bipartition to at most $\delta \leq 2$ (left) and $\delta \geq 2$ (right).

are better than those with $\alpha = 1, 2, 4$, and 8 by 3%, 2.3%, 1.3% and 0.5% in the median, respectively. Mt-KaHyPar-D-F with $\alpha = 16$ is slower than the configurations with $\alpha = 1$ and $\alpha = 8$ by a factor of 1.84 and 1.29 on average. If we increase $\alpha$ from 16 to 32, we still see a small improvement by 0.2% in the median but it slows down the overall partitioning time by 35% on average. Since Mt-KaHyPar-D-F aims for high solution quality, we set $\alpha$ to 16.

**Maximum BFS Distance (Mt-KaHyPar-D-F).** We restrict the distance of each node contained in the region $B$ to the cut hyperedges of the bipartition to at most $\delta$. The rationale behind this is that larger flow problems potentially lead to smaller minimum cuts but do not necessarily induce a balanced bipartition. The distance parameter $\delta$ restricts the region $B$ to promising minimum cuts that may induce a balanced bipartition and lead to a faster convergence of the FlowCutter algorithm.

Figure 5.7 shows the performance profiles that compare the solution quality of Mt-KaHyPar-D-F with different parameter choices for $\delta$. We see that the solution quality of Mt-KaHyPar-D-F is slightly better if we use $\delta = 2$ instead of $\delta = 1$. Larger values for $\delta$ do not improve the connectivity metric further (see Figure 5.7 right). However, Mt-KaHyPar-D-F with $\delta = 2$ is faster than the configuration that does not restrict the distances ($\delta = \max$) by a factor of 1.23 on average.

We conclude that restricting the distance of each node to the cut hyperedges of the bipartition is an effective technique to improve the running time of flow-based refinement without comprises in solution quality. Larger flow problems induce smaller minimum cuts, but balanced bipartitions are only induced by node moves close to the original cut.

## 5.7 Insights into Multilevel Partitioning

We now evaluate the different algorithmic components of Mt-KaHyPar-D(-F) in more detail. We start by analyzing how frequently search conflicts occur in our parallel refinement algorithm due to concurrent node moves. We did a similar analysis in Ref. [GHS22a] for flow-based refinement, and extend it to the label propagation and FM algorithm in Section 5.7.1. Section 5.7.2 then answers the question whether using stronger refinement algorithms leads to higher solution quality or if multiple repetitions of weaker and faster configurations achieve similar results. We conclude the experimental evaluation by studying the scalability and running times of the different algorithmic components of Mt-KaHyPar-D(-F) in Sections 5.7.3 and 5.7.4.

### 5.7.1 Analysis of Search Conflicts

In the parallel setting, several nodes can change their block simultaneously, which poses additional challenges for implementing parallel refinement algorithms. For example, the gain of a node move may change between its initial calculation and actual execution due to concurrent moves in its neighborhood (referred to as move conflicts, see Figure 3.7 in Section 3.4.1). Existing partitioners either employ techniques that restrict the set of possible moves during a refinement pass to prevent move conflicts (see Section 3.4.1) or follow an optimistic strategy assuming that conflicts happen rarely.

However, to the best of our knowledge, there is no experimental evaluation on the frequency of such conflicts in practice. Our refinement algorithms also follow an optimistic strategy but can detect move conflicts using attributed gain values and moves that violate the balance constraint using atomic `fetch-and-add` instructions to update block weights. If we detect such a conflict, we immediately revert the corresponding move. We will now use these techniques to measure how often these conflicts occur in practice.

We run Mt-KaHyPar-D-F on our parameter tuning benchmark set $\mathrm{M_P}$ using all 64 threads of machine $B$ (with $k \in \{2, 8, 16, 64\}$). The benchmark set $\mathrm{M_P}$ contains mainly small hypergraphs, and by using 64 threads, we want to force as many conflicts as possible. For each instance (hypergraph and number of blocks $k$), we count the total number of performed searches and the number of searches that found a potential improvement ($\Delta_{\text{exp}} \geq 0$). In the label propagation algorithm, a search consists of computing the highest gain move for a boundary node and then either performing it or not. In the FM and flow-based refinement algorithm, a search returns a sequence of node moves with a potential improvement based on the observed gains during a highly-localized FM search or the minimum cut obtained from a flow computation. For each potential improvement, we count the number of move sequences that violate the balance constraint ($c(V_i) \geq L_{\max}$), or degrade ($\Delta_{\lambda-1} < 0$) or improve the connectivity metric ($\Delta_{\lambda-1} \geq 0$). For move sequences with $\Delta_{\lambda-1} \geq 0$, we count if the actual improvement equals the expected ($\Delta_{\lambda-1} = \Delta_{\text{exp}}$) as well as zero-gain ($\Delta_{\lambda-1} = 0$) and positive-gain improvements ($\Delta_{\lambda-1} > 0$).

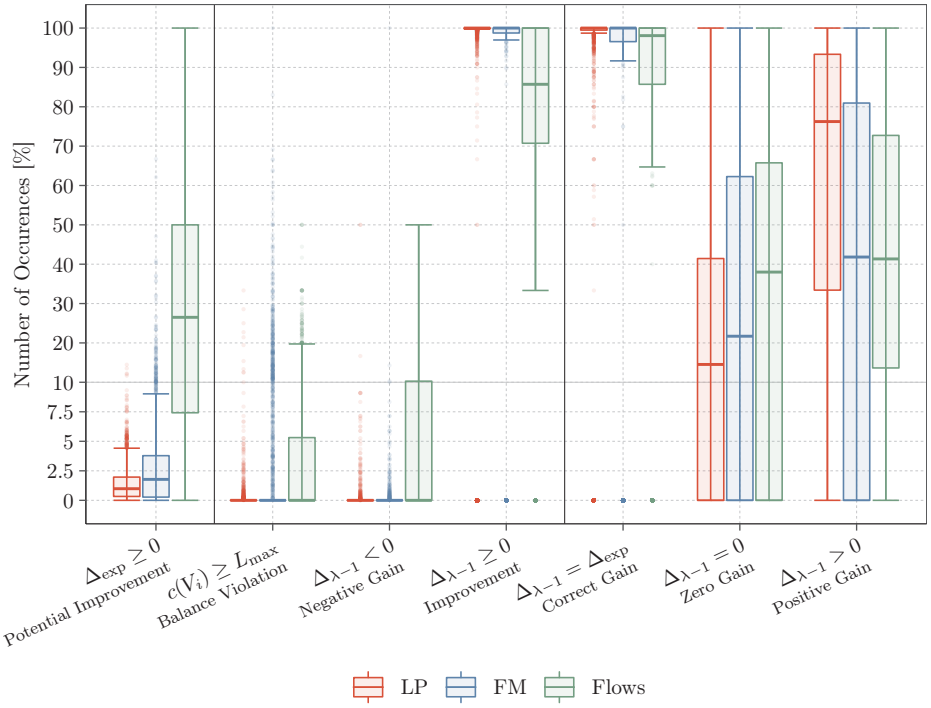



**Figure 5.8:** Frequency of search conflicts in the different parallel refinement algorithms due to concurrent node moves. Our refinement algorithms find move sequences with a potential improvement $\Delta_{\text{exp}}$. Applying a move sequence to the global partition yields an attributed gain value $\Delta_{\lambda-1}$ which is the actual improvement.

Figure 5.8 summarizes the experimental results for each instance using a box plot (outliers are represented with a point). The plot is divided into three parts. In the first part, we see the percentage of searches that found a potential improvement ($\Delta_{\text{exp}} \geq 0$). The second part shows the percentage of move sequences for which $c(V_i) \geq L_{\max}$, $\Delta_{\lambda-1} < 0$ or $\Delta_{\lambda-1} \geq 0$ relative to the number of potential improvements ($\Delta_{\text{exp}} \geq 0$). The last part shows the percentage of move sequences for which $\Delta_{\lambda-1} = \Delta_{\text{exp}}$, $\Delta_{\lambda-1} = 0$ or $\Delta_{\lambda-1} > 0$ relative to the number of move sequences that improved the connectivity metric ($\Delta_{\lambda-1} \geq 0$). Table 5.2 shows average percentages for the different statistics shown in Figure 5.8. We now analyze the conflict rates of the different refinement algorithms in more detail.

**Label Propagation Refinement.** On average, the label propagation algorithm finds a potential improvement for 1.4% of the boundary nodes from which 99.6% are

**Table 5.2:** Average percentages for the different statistics shown in Figure 5.8.

| | LP Avg [%] | FM Avg [%] | Flows Avg [%] |
|---|---|---|---|
| $\Delta_{\text{exp}} \geq 0$ | 1.4 | 3 | 31.9 |
| $c(V_i) \geq L_{\max}$ | 0.2 | 4.5 | 4.4 |
| $\Delta_{\lambda-1} < 0$ | 0.2 | 0.2 | 7.1 |
| $\Delta_{\lambda-1} \geq 0$ | 99.6 | 99.8 | 88.6 |
| $\Delta_{\lambda-1} = \Delta_{\text{exp}}$ | 99.1 | 99.6 | 96.2 |
| $\Delta_{\lambda-1} = 0$ | 29.4 | 44.2 | 45 |
| $\Delta_{\lambda-1} > 0$ | 70.6 | 55.8 | 55 |

applied on the partition ($\Delta_{\lambda-1} \geq 0$). For the applied moves, the expected equals the attributed gain value in 99.1% of all cases on average. Only 0.4% of the potential improvements violate the balance constraint (0.2%) or degrade the connectivity metric (0.2%). We also see that most of the performed moves have positive gain ($\Delta_{\lambda-1} > 0$). If we only look at SPM instances (which have larger node degrees and net sizes), we see that 0.7% of the potential improvements violate the balance constraint (0.3%) or degrade the connectivity metric (0.4%) on average. Thus, larger neighborhoods lead to more conflicts. However, the conflict rates remain low on average, suggesting that move conflicts are negligible in practice.

**FM Refinement.** On average, 3% of the localized FM searches find a potential improvement. The conflict rates are comparable to those of the label propagation algorithm. We note that the sum of moves that violate the balance constraint or degrade or improve the connectivity metric is larger than the number of potential improvements. The reason is that we relax the balance constraint in the localized FM searches (see Section 5.4) but use the original balance constraint to count the number of balance violations.

The results again suggest that conflicts are negligible, and the FM algorithm may also work well without the different techniques that we used to protect against move conflicts. However, one of the techniques to reduce conflicts is to perform moves on a thread-local partition and only make improvements visible to other threads. We now illustrate the impact of this technique on the solution quality and running time by running Mt-KaHyPar-D with and without thread-local partitions. If we run Mt-KaHyPar-D without thread-local partitions, we perform all moves on the global partition, meaning that threads can see intermediate states of other searches.

Figure 5.9 show the performance profiles comparing the solution quality of the two approaches (left) and the slowdown of the FM algorithm without thread-local partitions relative to the running time with thread-local partitions (right). We see that the solution quality of the partitions produced with thread-local partitions compared to performing all moves on the global partition is 1% better in the median with 64 threads.

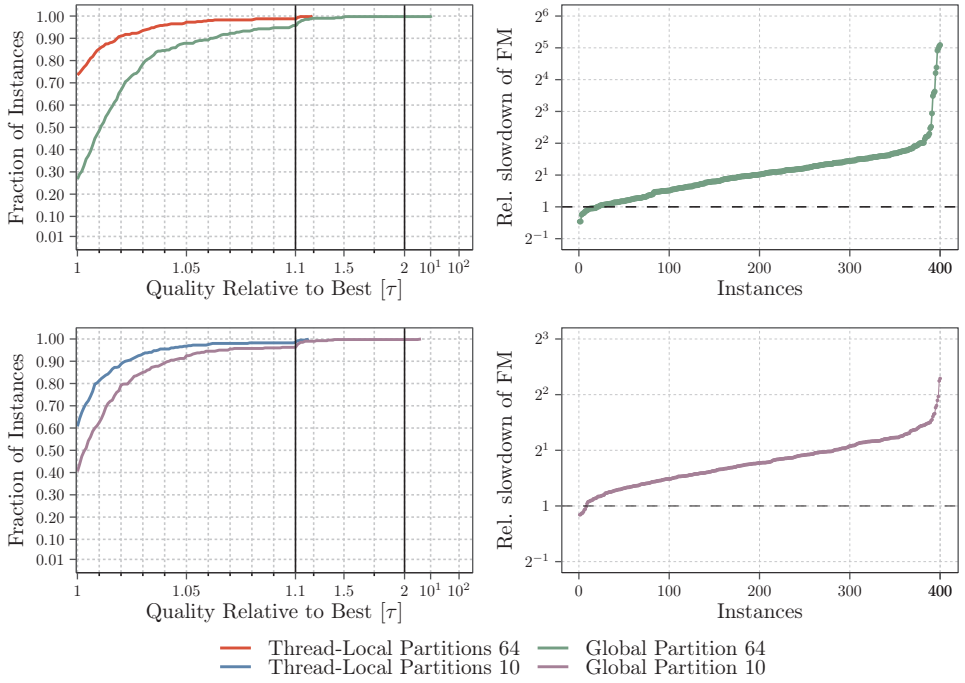



**Figure 5.9:** Performance profile comparing the solution quality (left) and running times (right) of Mt-KaHyPar-D with and without thread-local partitions.

The difference in solution quality is less pronounced when we use only ten threads. This demonstrates that move conflicts can have a significant impact on solution quality with an increasing number of threads. Surprisingly, the FM implementation that uses thread-local partitions is 2.06 times faster on average with 64 threads, even though it uses hash tables to store changes on the local partition. We assume this is due to contention on the hyperedge locks, which we use to update the pin count values when we perform a node move. Again, the slowdowns are less pronounced with ten threads.

In summary, the FM algorithm has similar conflict rates to the label propagation algorithm. However, thread-local partitions are required to keep them low.

**Flow-Based Refinement.** On average, 31.9% of the flow computations on adjacent block pairs find a potential improvement. Hence, it has significantly higher success rates as the other two refinement algorithms. We also measured the number of moved nodes per search and found out that flow-based refinement moves 48.6 nodes on average, while the FM algorithm moves only 3.3 nodes per localized FM search. Additionally, flows are computationally expensive, meaning that it can take several (milli)seconds to solve a flow problem. Changes made by other threads are not visible

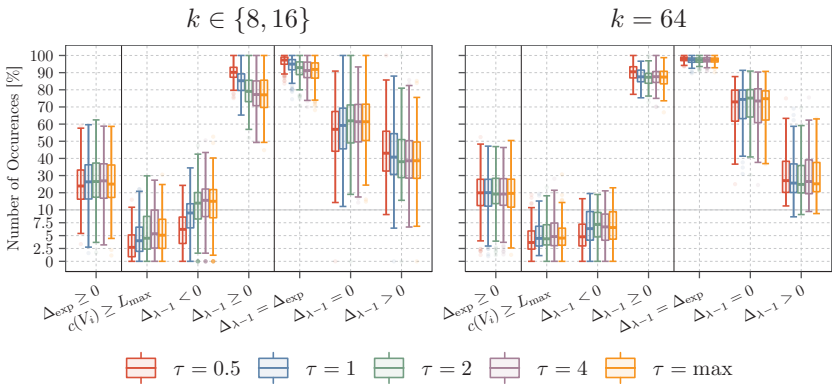



**Figure 5.10:** Frequency of search conflicts in the flow-based refinement algorithm for different values of $\tau$.

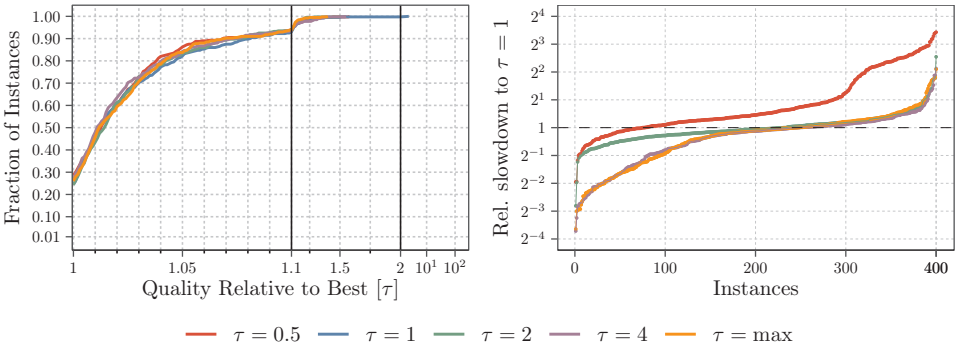



**Figure 5.11:** Performance profile comparing the solution quality (left) and running times (right, relative to $\tau = 1$) of Mt-KaHyPar-D-F for different values $\tau$.

when calculating a maximum flow. This can lead to conflicts when applying the corresponding move sequence. The high success rates, the large number of moved nodes per flow computation, and the high running times also explain the higher conflict rates. On average, 4.4% (5.5% for $k > 2$) of the potential improvements violate the balance constraint, and 7.1% (8.9% for $k > 2$) degrade the connectivity metric. However, as we will see in Section 5.7.3, increasing the number of threads does not adversely affect the solution quality of Mt-KaHyPar-D-F.

Our scheduler uses $\min(k, \tau \cdot t)$ threads to process the active block pairs of the quotient graph in parallel (and one thread for $k = 2$). Figure 5.10 shows the conflict rates of the flow-based refinement algorithm for $k \in \{8, 16\}$ (left) and $k = 64$ (right) for different values of $\tau$. We can see that we can almost half the number of conflicts when

we use $\tau = 0.5$ instead of $\tau = \max$ for $k \in \{8, 16\}$. Figure 5.11 compares the solution quality of Mt-KaHyPar-D-F for different values of $\tau$ (left) and their running times relative to $\tau = 1$ (right). The performance profile indicates that all configurations produce partitions with comparable solution quality. However, $\tau = 0.5$ is slower than $\tau = 1$, and all other configurations have similar running times. Therefore, we choose $\tau = 1$, providing a good tradeoff between running time and conflict rates.

### 5.7.2 Effectiveness Tests

We presented several techniques that improve the solution quality of partitions at the cost of an increased running time. This includes the community detection algorithm (CD) and the parallel label propagation, FM, and flow-based refinement (F) algorithm. We now analyze whether enabling each component leads to better solution quality or if multiple repetitions of weaker configurations can produce similar results. To do so, we run different configurations of our multilevel algorithm on benchmark set $M_{HG}$ using ten threads. In order to compare them, we use the effectiveness tests presented in Section 2.4.3. Here, we compare two algorithms by giving the faster algorithm more time to perform additional repetitions until its expected running time equals the running time of the slower algorithm. We abbreviate a configuration with (+CD,-FM,-F), where + or - indicates whether or not the corresponding component is used. We enable label propagation refinement in all configurations. Note that (+CD,+FM,-F) and (+CD,+FM,+F) correspond to Mt-KaHyPar-D and Mt-KaHyPar-D-F.

Figure 5.12 shows the effectiveness tests on virtual instances comparing different configurations of Mt-KaHyPar. On the left side, we compare configurations where we successively disable our refinement algorithms except for the last plot that compares our weakest and strongest configuration (bottom-left). On the right side, we compare variants using the same set of refinement algorithms but with and without the community-aware coarsening algorithm.

We see that using stronger refinement algorithms leads to significantly better solution quality. Enabling the FM algorithm improves the connectivity metric by 4.8% in the median compared to the configuration that only uses the label propagation algorithm. If we additionally enable flow-based refinement, we can further improve the solution quality by 2.3% in the median. The partitions computed by our strongest configuration are better than those of our weakest configuration by 12% in the median. All configurations using the community-aware coarsening algorithm (+CD) outperform their corresponding weaker configurations (-CD). However, the differences are less pronounced when we use stronger refinement algorithms.

**Comparison of Mt-KaHyPar-D and Mt-KaHyPar-D-F.** The effectiveness tests have shown that each component improves the solution quality, even when faster configurations are given more time to perform additional repetitions. We now compare our two partitioners, Mt-KaHyPar-D and Mt-KaHyPar-D-F, when both perform the same number of repetitions on our medium-sized (set $M_{HG}$, 10 threads) and large benchmark set (set $L_{HG}$, 64 threads). The results are shown in Figure 5.13 and 5.14.

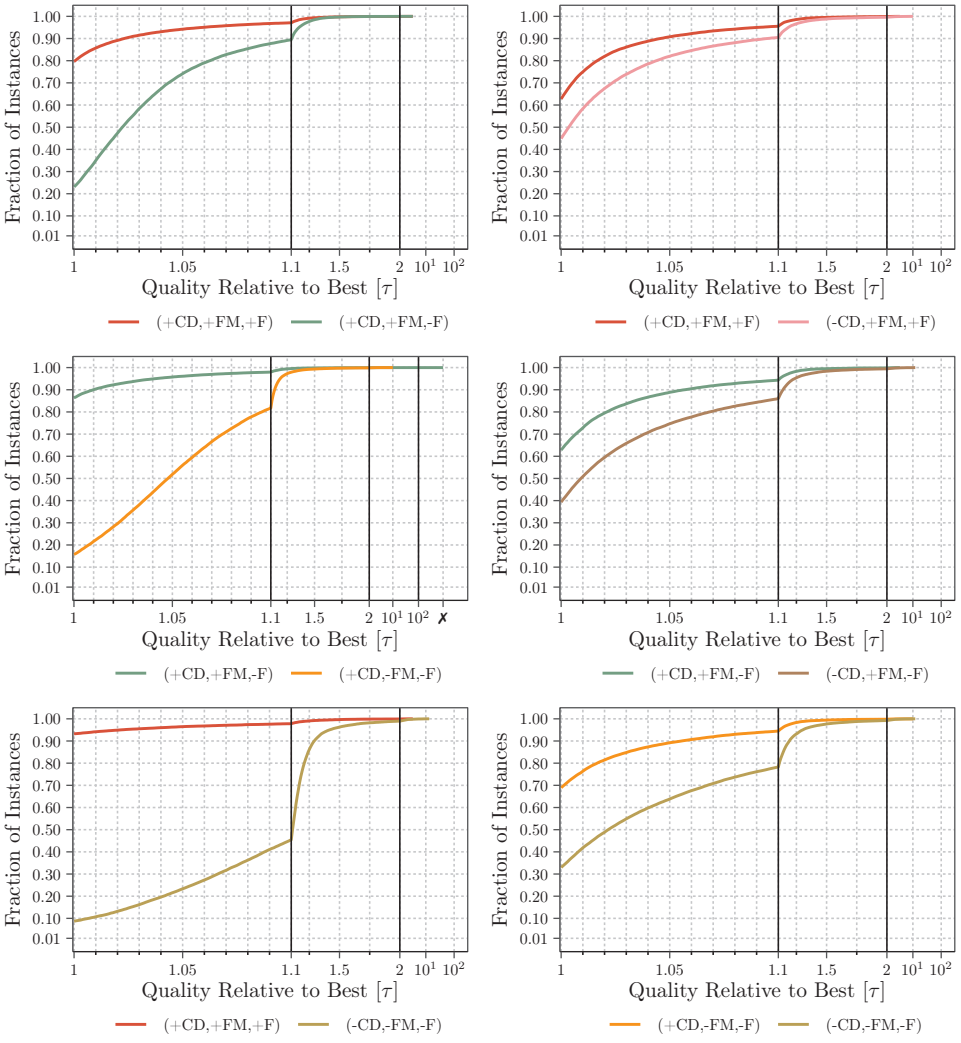



**Figure 5.12:** Component effectiveness tests for our multilevel partitioner on benchmark set $M_{HG}$. The performance profiles compare the solution quality of different configurations with (+) and without (-) community-aware coarsening (CD) and FM, and flow-based refinement (F).

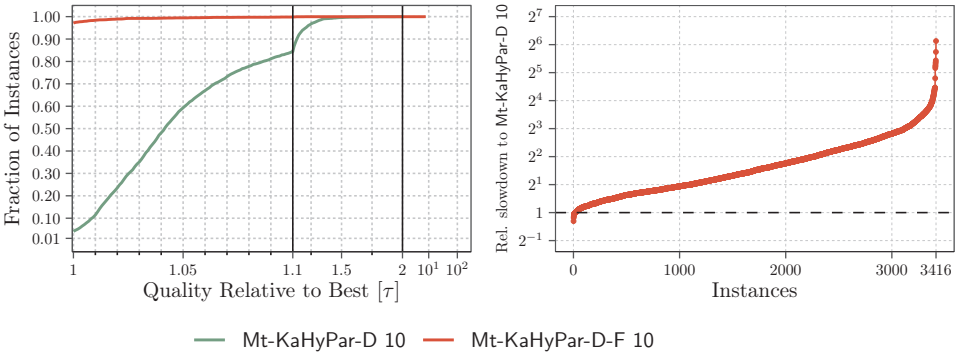



**Figure 5.13:** Performance profiles and running times comparing Mt-KaHyPar-D and Mt-KaHyPar-D-F on set $\mathrm{M_{HG}}$.

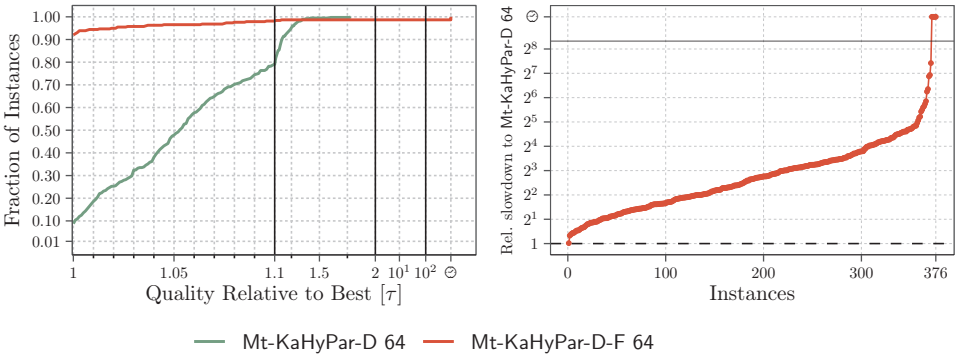



**Figure 5.14:** Performance profiles and running times comparing Mt-KaHyPar-D and Mt-KaHyPar-D-F on set $\mathrm{L_{HG}}$.

On set $\mathrm{M_{HG}}$, Mt-KaHyPar-D-F computes partitions with better connectivity than Mt-KaHyPar-D on 97.1% of the instances. The median improvement of Mt-KaHyPar-D-F compared to Mt-KaHyPar-D is 4.2% while only incurring a slowdown by a factor of 3.08 on average (geometric mean running time 2.73s vs 0.89s). On set $\mathrm{L_{HG}}$, both the improvements (5.3%) and slowdowns (geometric mean running time 30.38s vs 4.64s) are more pronounced. The slowdowns are expected since the size of the flow problems scales linearly with instance sizes, while the complexity of the flow-based refinement routine does not. We note that Mt-KaHyPar-D-F was not able to partition our largest instance into $k \in \{8, 16, 64\}$ blocks (`sk-2005`) and our second and third largest instance into $k = 64$ blocks (`it-2004` and `uk-2005`) in under two hours.

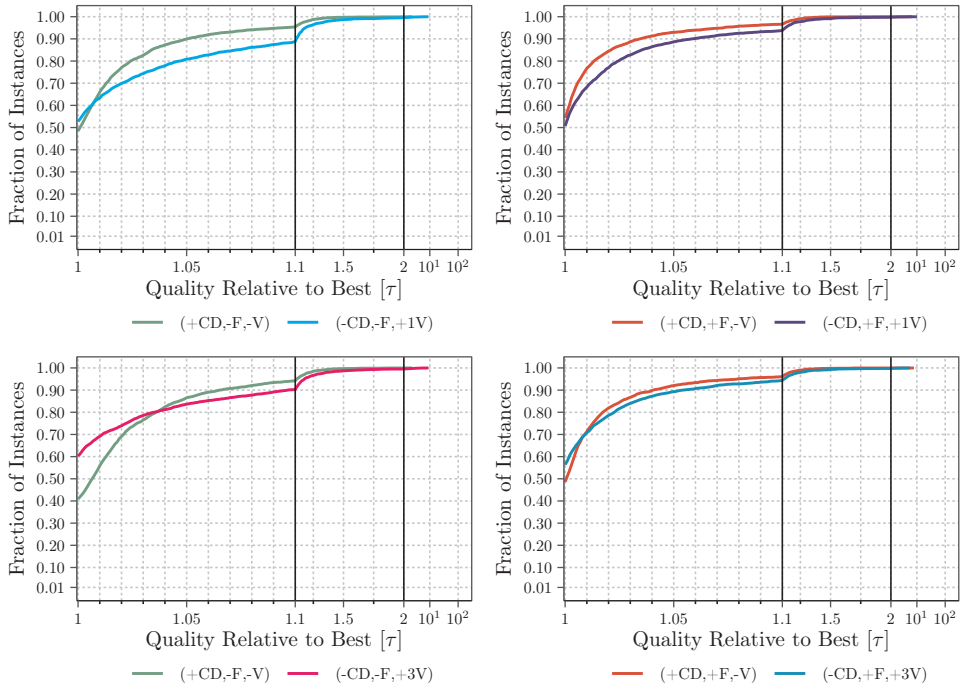



**Figure 5.15:** Performance profiles comparing the solution quality of different configurations with (+) and without (-) community-aware coarsening (CD), flow-based refinement (F), and with one (+1V) and three V-cycles (+3V).

**Comparison of Community Detection and Iterated Multilevel Cycles.** We run a community detection algorithm to obtain the community structure of the input hypergraph and use it to restrict contractions to nodes within the same community in the coarsening phase. This approach is similar to the popular iterated multilevel cycle technique [Wal04] (also called *V-cycle* technique). The technique uses an existing $k$-way partition to restrict contractions to nodes within the same block in the coarsening phase. In the initial partitioning phase, the input partition is projected to the coarsest hypergraph, which then induces the same cut and balance as on the input hypergraph. This can be repeated several times, each time using the partition found in previous multilevel cycle as input.

In the following, we compare the solution quality of Mt-KaHyPar-D and Mt-KaHyPar-D-F with our community detection algorithm (CD) and the V-cycle technique (V) on set $M_{HG}$. We abbreviate a configuration, e.g., with (-CD,+F,+1V), where + or - indicates whether or not the corresponding component is used (F stands for flow-based refinement, and +1V indicates that *one* V-cycle is used). Note that (+CD,-F,-V) and (+CD,+F,-V) corresponds to Mt-KaHyPar-D and Mt-KaHyPar-D-F.

**Table 5.3:** Geometric mean speedups over all instances and instances with a single threaded running time ≥ 100s for total partition time (T), community detection (CD), coarsening (C), initial partitioning (IP), label propagation (LP), FM, and flow-based refinement. The last row shows the percentage of instances with a single-threaded running time ≥ 100 seconds.

| | | Mt-KaHyPar-D | | | | | | Mt-KaHyPar-D-F | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Flow-Based Refinement | | | |
| Number of Threads | | T | CD | C | IP | LP | FM | T | $k = 2$ | $k \in \{8, 16\}$ | $k = 64$ |
| All | 4 | 3.5 | 3.3 | 3.6 | 3.7 | 3.4 | 3.5 | 3.1 | 2.7 | 2.5 | 3.4 |
| | 16 | 11.1 | 9.9 | 11.2 | 11.1 | 6.8 | 10.3 | 7.4 | 3 | 4.9 | 10.7 |
| | 64 | 19.7 | 20.1 | 23.3 | 11.6 | 7.7 | 14.4 | 10.6 | 2.9 | 6.2 | 18.5 |
| ≥ 100s | 4 | 3.4 | 3 | 3.3 | 3.4 | 3.8 | 3.6 | 3.1 | 6 | 2.5 | 3.2 |
| | 16 | 11.5 | 10.7 | 11.6 | 14.2 | 12.2 | 12.6 | 8.4 | 9.9 | 6.3 | 10.1 |
| | 64 | 24.7 | 27.6 | 27.4 | 34 | 26.8 | 29.4 | 14.5 | 13.3 | 11.4 | 17.6 |
| Inst. ≥ 100s [%] | | 47.6 | 13.8 | 17 | 19.4 | 1.3 | 27.9 | 54.3 | 25 | 47.4 | 51.3 |

Figure 5.15 shows the solution quality of Mt-KaHyPar-D (left side) and Mt-KaHyPar-D-F (right side) compared to configurations that replace the community detection algorithm with *one* and *three* V-cycles. As can be seen, using one V-cycle produces partitions with similar solution quality to our community detection approach. Increasing the number of V-cycles from one to three only moderately increases the solution quality for Mt-KaHyPar-D, while we see almost no improvement for Mt-KaHyPar-D-F. This indicates that Mt-KaHyPar-D-F already finds good local optima with little room for improvements. However, the advantage of the community detection approach becomes obvious when we look at the running times. Mt-KaHyPar-D (geometric mean running time 0.88s) with community detection is on 93.4% (99.6%) of the instances faster than the configuration with one (three) V-cycle(s) (one V-cycle: 1.09s, three V-cycles: 1.56s). The running time improvements look similar for Mt-KaHyPar-D-F (2.73s, one V-cycle: 3.5s, three V-cycles: 4.82s).

### 5.7.3 Scalability

In Figure 5.16 and 5.18 as well as Table 5.3, we show self-relative speedups for several algorithmic components of Mt-KaHyPar-D and Mt-KaHyPar-D-F with varying number of threads $t \in \{4, 16, 64\}$. We run the scalability experiments for Mt-KaHyPar-D on set $L_{HG}$ and machine B. For Mt-KaHyPar-D-F, we used a subset of set $L_{HG}$ (76 out of 94 hypergraphs) containing all hypergraphs on which Mt-KaHyPar-D-F 64 was able to finish in under 600 seconds for all $k \in \{2, 8, 16, 64\}$. This experimental evaluation is based on the data from the corresponding publications of Mt-KaHyPar-D [Got+21a] and Mt-KaHyPar-D-F [GHS22a]. We did not rerun these experiments since one run took roughly six weeks on machine B.

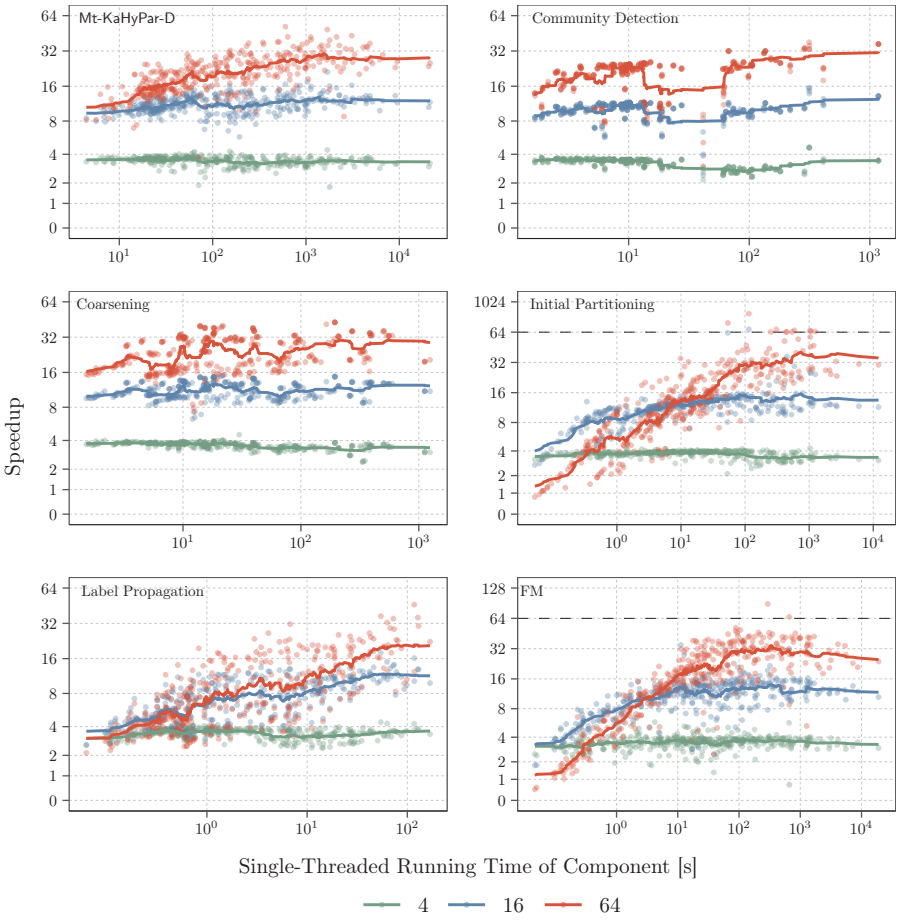



**Figure 5.16:** Speedups of Mt-KaHyPar-D and its different algorithmic components.

**Scalability of Mt-KaHyPar-D.** The overall geometric mean speedup of Mt-KaHyPar-D is 3.5 for $t = 4$, 11.1 for $t = 16$ and 19.7 for $t = 64$. If we only consider instances with a single-threaded running time $\geq 100$s, we achieve a geometric mean speedup of 24.7 for $t = 64$. For $t = 4$, the speedup is at least 3 on 92.3% of the instances.

Community detection and coarsening share many similarities in their implementation and both show reliable speedups for an increasing number of threads. For the remaining three components, we observe that longer single-threaded execution leads to substantially better speedups. For initial partitioning, increasing the number of threads from 16 to 64 can even be harmful for instances with a single-threaded running time

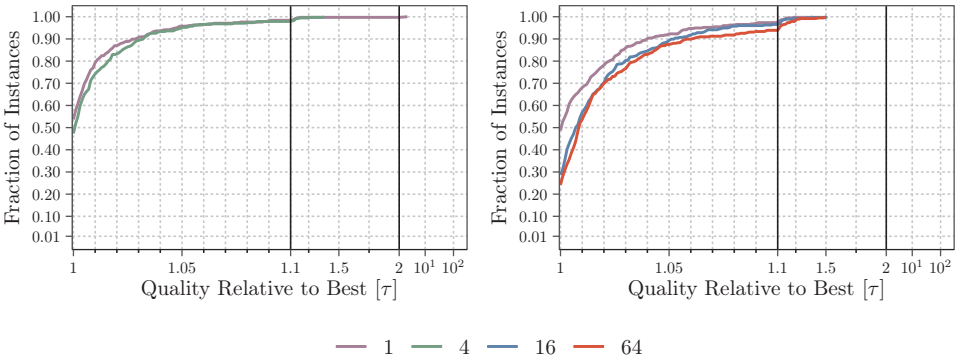



**Figure 5.17:** Performance profiles comparing the solution quality of Mt-KaHyPar-D with an increasing number of threads on set $L_{HG}$.

of one second or less. While label propagation refinement yields the least promising speedups, it is substantially faster than the other components, taking less than 10% of the overall running time on over 95% of the instances for $t = 64$.

The FM algorithm is the most time-consuming component of Mt-KaHyPar-D (see Figure 5.20 in Section 5.7.4) and therefore its scalability is crucial for the overall algorithm. The geometric mean speedup of the FM algorithm is 3.5 for $t = 4$, 10.3 for $t = 16$ and 14.4 for $t = 64$. However, the speedup increases to 29.4 for $t = 64$ when we only consider instances with a single-threaded running time $\geq$ 100s.

Figure 5.17 compares the solution quality of Mt-KaHyPar-D with increasing number of threads. The solution quality of the partitions with a moderate number of threads ($\leq 4$) is comparable. However, using more than 16 threads slightly affects the solution quality of Mt-KaHyPar-D.

**Scalability of Mt-KaHyPar-D-F.** The overall geometric mean speedup of Mt-KaHyPar-D-F is 3.1 for $t = 4$, 7.4 for $t = 16$ and 10.6 for $t = 64$. If we only consider instances with a single-threaded running time $\geq$ 100s, we achieve a geometric mean speedup of 14.5 for $t = 64$. Since Mt-KaHyPar-D-F extends Mt-KaHyPar-D with flow-based refinement, we focus on the scalability of this component. Recall that we use $\min(t, k)$ threads to process the active block pairs of the quotient graph in parallel (and only one thread for $k = 2$).

For $k = 2$, the scalability of the flow-based refinement routine largely depends on the FlowCutter algorithm as the only parallelism source. With 4 threads, the geometric mean speedup is 2.7. For $t \in \{16, 64\}$, the parallelization overheads are only outweighed for longer running instances, with more threads becoming worthwhile at about 100 seconds of sequential time. Unfortunately, we even experience some minor slowdowns and the speedups are strongly scattered. The geometric mean speedup for all instances with a single-threaded running time $\geq$ 100s is 9.9 for $t = 16$, and 13.3 for

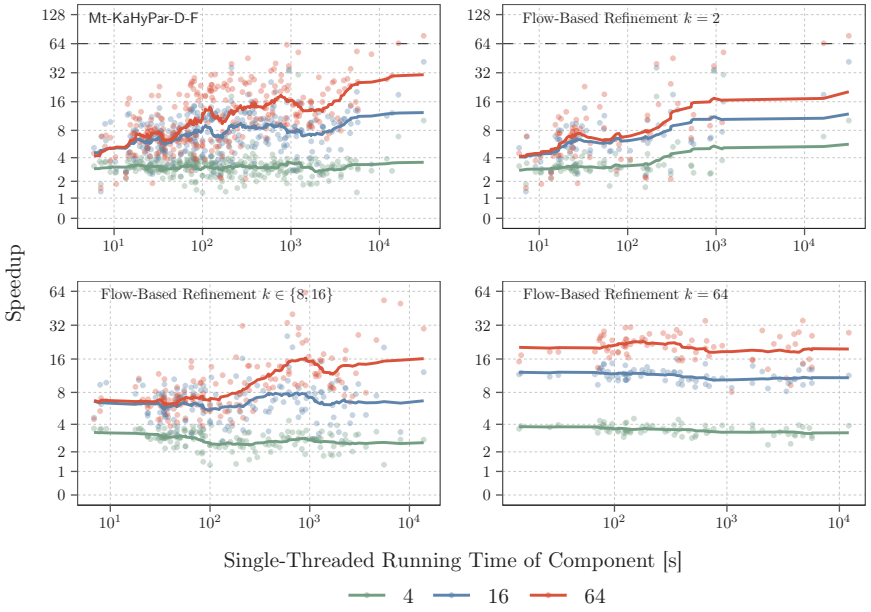



**Figure 5.18:** Speedups of Mt-KaHyPar-D-F and the flow-based refinement routine for different values of $k$.

$t = 64$. However, the speedups match the results of the parallel flow algorithm from Ref. [BBS15] that we integrated into the FlowCutter algorithm.

For $k = 8$ ($= 16$) and $t = 64$, we use 8 (16) threads to process the active block pairs in parallel, while the remaining threads are used for parallel flow computations. Thus, both parallelism sources are used. The speedups are slightly better than for $k = 2$. Note that we use sequential implementations of the flow network construction and maximum flow algorithm when the number of flow problems processed in parallel equals the number of available threads (e.g., for $k = 16$ and $t = 16$). Therefore, the poor speedups for instances with short single-threaded running times ($\leq$ 100s) are caused by parallelization overheads.

For $k = 64$ and $t = 64$, we achieve a geometric mean speedup of 18.5. In this case, we use all threads to process the active block pairs in parallel. Thus, all parallelism is leveraged in the scheduler, and none in the FlowCutter algorithm, which explains why we obtain more reliable speedups than for all other $k$.

Figure 5.19 shows that increasing the number of threads does not adversely affect the solution quality of Mt-KaHyPar-D-F.

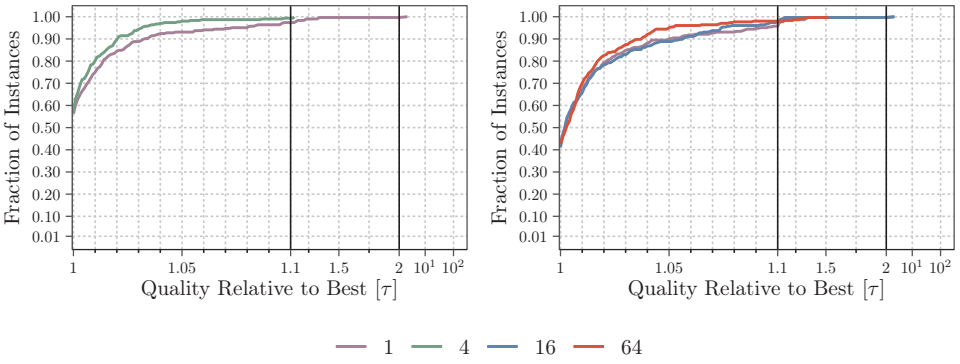



**Figure 5.19:** Performance profiles comparing the solution quality of Mt-KaHyPar-D-F with increasing number of threads on set $L_{HG}$.

### 5.7.4 Running Times of Components

We now analyze the running times of the different components of Mt-KaHyPar-D and Mt-KaHyPar-D-F on set $L_{HG}$ in more detail (using 64 threads). Figure 5.20 shows two different types of plots illustrating the share of each component on the total partitioning time. The first plot (left) uses a bar plot to visualize the running times of each component for each instance. In these plots, we take the execution time of the most time-consuming component and sort the instances according to it. The second plot (right) is similar to the performance profiles. It shows the percentage of instances (y-axis) for which the share of a component on the total partitioning time is $\geq x\%$.

**Mt-KaHyPar-D.** The most time-consuming parts of Mt-KaHyPar-D are community detection, coarsening, and the FM algorithm. These components have approximately the same share on the total partitioning time, which is between 21% and 23% in the median. However, there are a few instances for which the FM algorithm dominates the running time. A closer look reveals that these are mainly Dual instances derived from satisfiability problems [Bel+14]. Hypergraphs from this domain usually have large hyperedges with a low average node degree. We observed that the localized FM searches perform many zero-gain moves for these instances, making our adaptive stopping rule less effective. Thus, the searches move more nodes on average, requiring more memory to store intermediate states in the thread-local partitions. This makes the FM algorithm less cache-efficient and increases its running time.

The share of the initial partitioning phase on the total partitioning time is 8.3% in the median. For $k = 2$, the share of initial partitioning on the total running time is less than 10% on 88% of the instances. Larger running times can be observed for Literal and Primal instances also derived from satisfiability problems. These instances have many graph edges and highly-skewed node degree distributions (similar to complex networks). Here, the coarsest hypergraph tends to be larger and causes longer running

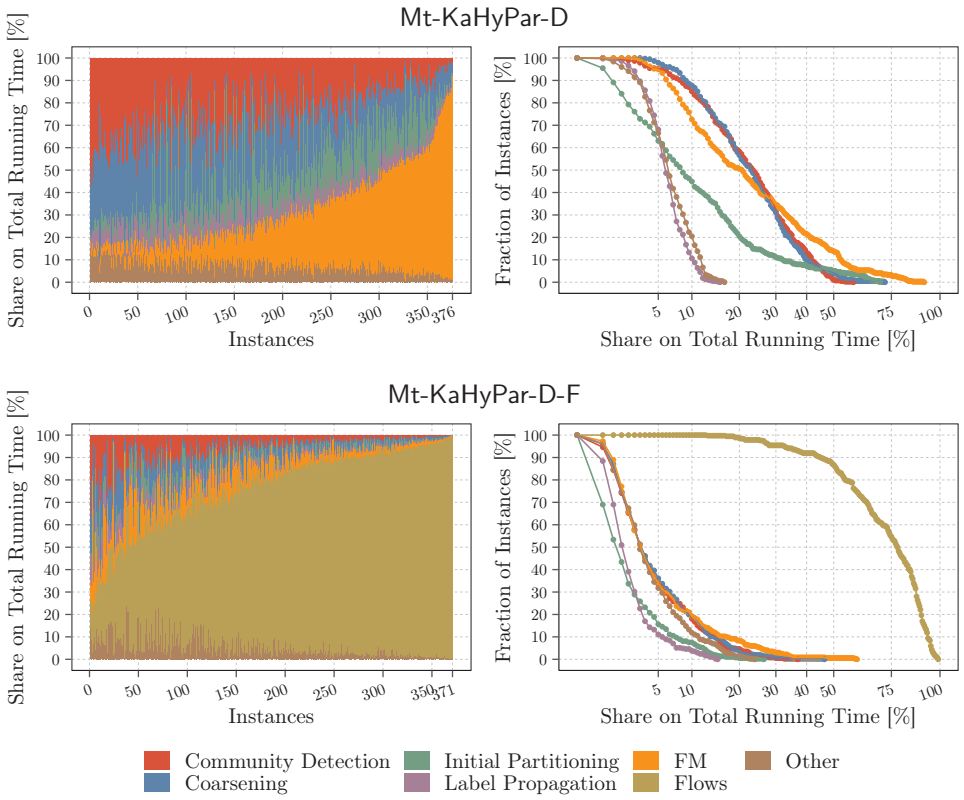



**Figure 5.20:** Running time shares of different components on the total execution time of Mt-KaHyPar-D (top) and Mt-KaHyPar-D-F (bottom).

times. The running time of the label propagation algorithm is negligible on most of the instances.

**Mt-KaHyPar-D-F.** As shown in Figure 5.20 (bottom), the flow-based refinement routine dominates the running time of Mt-KaHyPar-D-F on most of the instances. We therefore analyze this component in more detail. Since the flow-based refinement algorithm uses nested parallelism, we can only measure the exact running time shares of the different components for $k = 2$. For $k > 2$, we measure the running times of the different phases in the nested calls and take their sum as the running time of a component.

Figure 5.21 shows the shares of different components on the total running time of the flow-based refinement routine for different values of $k$. For $k = 2$, solving the flow problems with the FlowCutter algorithm takes the most time. The results look similar for $k \in \{8, 16\}$ and are therefore omitted in the plot. For $k = 64$, the flow network construction and FlowCutter algorithm both have similar shares on total partitioning

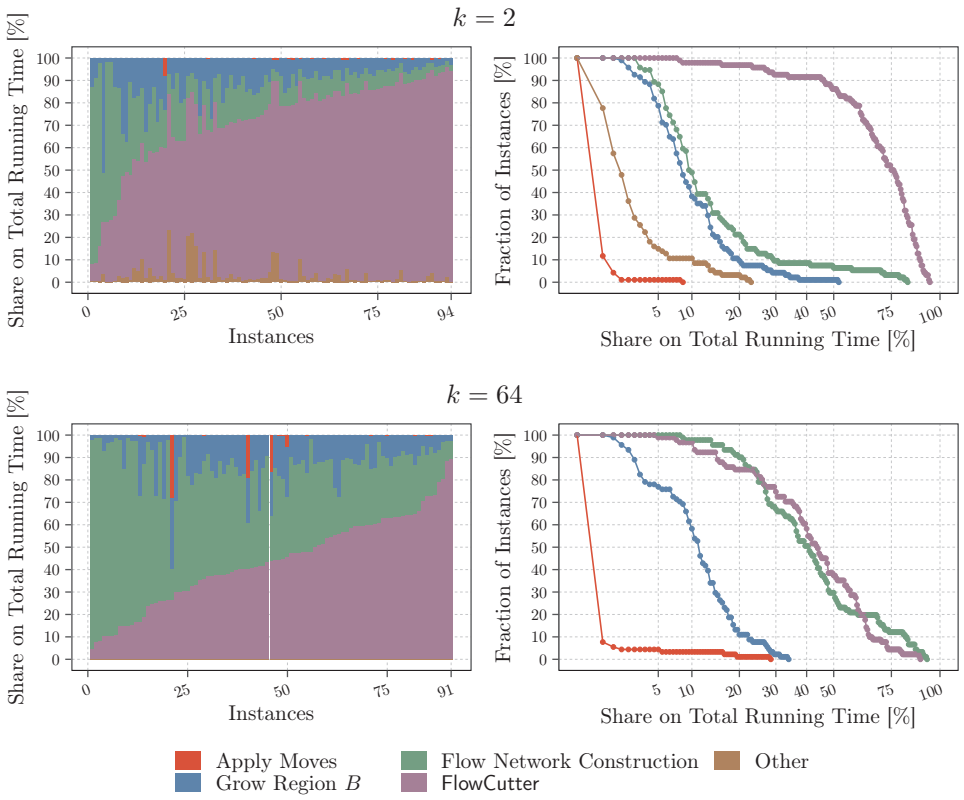



**Figure 5.21:** Running time shares of different components on the total execution time of the flow-based refinement routine.

time. Recall that the size of a flow problem is proportional to the balance constraint $L_{\max} = (1 + \varepsilon)\lceil \frac{c(V)}{k} \rceil$. Larger values for $k$ lead to smaller flow problems reducing the time for solving them.

Chapter 6

# 6 Parallel $n$-level Hypergraph Partitioning

Traditional multilevel partitioners contract matchings or clusterings on each level, leading to a multilevel hierarchy with approximately a logarithmic number of levels. However, there is usually a correspondence between the number of levels and the tradeoff between solution quality and running time [Saa95; Sch20]. More levels provide "more opportunities to refine the current solution" [AHK97] at different granularities but using more local search also increases the running time. The $n$-level partitioning scheme completely evicts this tradeoff by instantiating the multilevel paradigm in its most extreme version, contracting only a single node on each level. Correspondingly, in each refinement step, only a single node is uncontracted, allowing a highly-localized search for improvements. The $n$-level scheme is currently implemented by the sequential hypergraph partitioner KaHyPar [Sch+16a; Akh+17a]. KaHyPar is the method of choice for partitioning hypergraphs with high solution quality but is often substantially slower than other partitioning systems – prohibitively so for large hypergraphs. Although contracting a single node on each level seems inherently sequential, we show that under certain relaxations the $n$-level scheme can be parallelized efficiently and without compromises in solution quality.

**Algorithm Overview.** Algorithm 6.1 shows the high-level pseudocode of our parallel $n$-level hypergraph partitioner, which also serves to outline the structure of this chapter. Contracting and uncontracting nodes in a strict order is inherently sequential, which is why we have to relax the $n$-level paradigm. For the coarsening phase, we iterate in parallel over the nodes and select a contraction partner for each node. Contractions are performed on-the-fly as in the sequential $n$-level scheme, completely asynchronously with the contraction partner selection. To this end, we propose a new low-overhead hypergraph data structure in Section 6.1 and describe how to implement contractions and uncontractions on it. This will reveal certain conditions and intricacies regarding parallelization that must be addressed by the coarsening and uncoarsening algorithms in Sections 6.2 and 6.3. The challenge addressed in Section 6.2 is to keep the contractions compatible (see Line 5, a formal definition follows) as well as to determine a schedule for the contraction operations. For uncoarsening, we construct a sequence of batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$ of contracted nodes, such that $|B_i| \approx b_{\max}$ where $b_{\max}$ is an input parameter. Batches are processed one after another, enabling the uncontraction of nodes in subsequent batches. Nodes in the same batch are uncontracted in parallel. The challenge of identifying which nodes can or even must appear in the same batch is addressed in Section 6.3. After uncontracting each batch, we apply highly-localized refinement algorithms around the batched nodes.

**Algorithm 6.1:** Parallel $n$-level Hypergraph Partitioning

**Input:** Hypergraph $H = (V, E)$, number of blocks $k$
**Output:** $k$-way partition $\Pi$ of $H$

```
1  ∀v ∈ V : rep[v] ← v                                   // initialize empty forest F
2  while |V| > 160 · k do
3      for u ∈ V in random order do in parallel
4          v ← arg max_v Σ_{e∈I(u)∩I(v)} ω(e)/(|e|−1)    // see Section 5.2.1
5          if (u, v) can be safely added to F then       // see Section 6.2
6              rep[v] ← u and contract v onto u          // see Section 6.1.2
7      remove single-pin and identical nets              // see Section 6.1.1
8  Π ← initialPartition(H, k)                            // see Section 5.3
9  B = ⟨B_1, ..., B_l⟩ ← constructBatches(F)             // see Section 6.3
10 for B ∈ B do                                          // |B| ≈ b_max
11     if B ≠ ∅ then
12         for v ∈ B do in parallel
13             uncontract v from rep[v]                  // see Section 6.1.3
14             Π[v] ← Π[rep[v]]
           // Localized refinement around the boundary nodes of the current batch
15         while improvement found do
16             Π ← labelPropagationRefinement(H, Π, B)   // see Section 4.2
17             Π ← fmLocalSearch(H, Π, B)                // see Section 4.3
18     else
19         restore single-pin and identical nets         // see Section 6.1.1
           // Global refinement using all boundary nodes of the partition
20         while relative connectivity improvement ≥ 0.25% do
21             Π ← fmLocalSearch(H, Π)                   // see Section 4.3
22             Π ← flowBasedRefinement(H, Π)             // see Section 4.4
```

**Similarities to Mt-KaHyPar-D(-F).** Our $n$-level code shares most of the algorithmic components with Mt-KaHyPar-D(-F). For coarsening, we use the heavy-edge rating function and restrict contractions to densely-connected regions of the hypergraph by using the community detection algorithm presented in Section 5.2.4. In the initial partitioning phase, we use multilevel recursive bipartitioning with the same portfolio of flat bipartitioning techniques as explained in Section 5.3 but use the $n$-level coarsening and uncoarsening algorithm. Uncoarsening uses the same set of refinement algorithms, but after uncontracting a batch of nodes, we initialize them only with the boundary nodes of the current batch (see Line 16 and 17). After reverting all contractions from a coarsening pass, we additionally use a global refinement step that corresponds to refinement used in Mt-KaHyPar-D(-F) (see Line 21 and 22).

**References and Contributors.** This chapter covers our $n$-level partitioning algorithm presented in Refs. [Got+21c; Got+22a]. Large parts of the content were copied verbatim from the corresponding conference publication [Got+22a], but most of the experimental evaluation was rewritten and enhanced with additional experiments. The idea and implementation of the parallel $n$-level partitioning algorithm came from the author of this dissertation, while Lars Gottesbüren was involved in the performance engineering process.

## 6.1 The Dynamic Hypergraph Data Structure

To support concurrent on-the-fly contractions and uncontractions, we use a dynamic hypergraph data structure which is a space-efficient version of the one used in KaHyPar [Sch20, p. 100]. In KaHyPar, the pins of the nets are represented as an adjacency array (the sub-range storing the pins of a certain net is called its pin-list), whereas the incident nets of the nodes are represented using adjacency-lists, i.e., as a separate vector for each node (see Figure 3.5 in Section 3.3.2). Contracting a node $v$ onto another node $u$ replaces $v$ with $u$ in all nets $e \in \mathrm{I}(v) \setminus \mathrm{I}(u)$ and removes $v$ from all nets $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$. Because of the adjacency-lists in KaHyPar, a contraction $(u, v)$ entails copying $\mathrm{I}(v) \setminus \mathrm{I}(u)$ to $\mathrm{I}(u)$. In the worst case, this can lead to quadratic memory usage and is therefore not practical for large hypergraphs or hypergraphs with a highly-skewed degree distribution. Instead, we propose a new data structure for storing incident nets that enables concurrent (un)contractions without allocating additional memory. The data structure is slightly slower than the approach used in KaHyPar (see Section 6.4.3), but allows us to handle larger and highly skewed instances.

The key idea is to remove $\mathrm{I}(u) \cap \mathrm{I}(v)$ from $\mathrm{I}(v)$ (instead of adding $\mathrm{I}(v) \setminus \mathrm{I}(u)$ to $\mathrm{I}(u)$). $\mathrm{I}(u)$ is obtained by iterating over both the representation of the current $\mathrm{I}(u)$ and the remaining entries of $\mathrm{I}(v)$. For each node $u \in V$, we store an array $I_u$ which is initialized with the incident nets of $u$ on the input hypergraph. We organize all nodes contracted onto $u$ as well as $u$ itself in a doubly-linked list $L_u$, so that the *current state* for $\mathrm{I}(u)$ is obtained by iterating over all $I_w$ arrays for $w \in L_u$. When contracting $v$ onto $u$, we remove any incident net of $u$ from the arrays $I_w$ for $w \in L_v$ and append $L_v$ to $L_u$. For storing the pin-lists of nets, we use an adjacency array as in KaHyPar. Each pin-list is split into an *active* and *inactive* part. The active part represents the pin-list of the net, and the inactive part contains pins that were previously part of the net but have already been contracted. The data structure and all operations, which we describe in more detail in the following, are illustrated with an example of multiple contraction and uncontraction steps in Figure 6.1. In each step, the top part shows the current state of the pin-lists, the bottom part shows the incident net arrays $I_w$ and lists $L_w$.

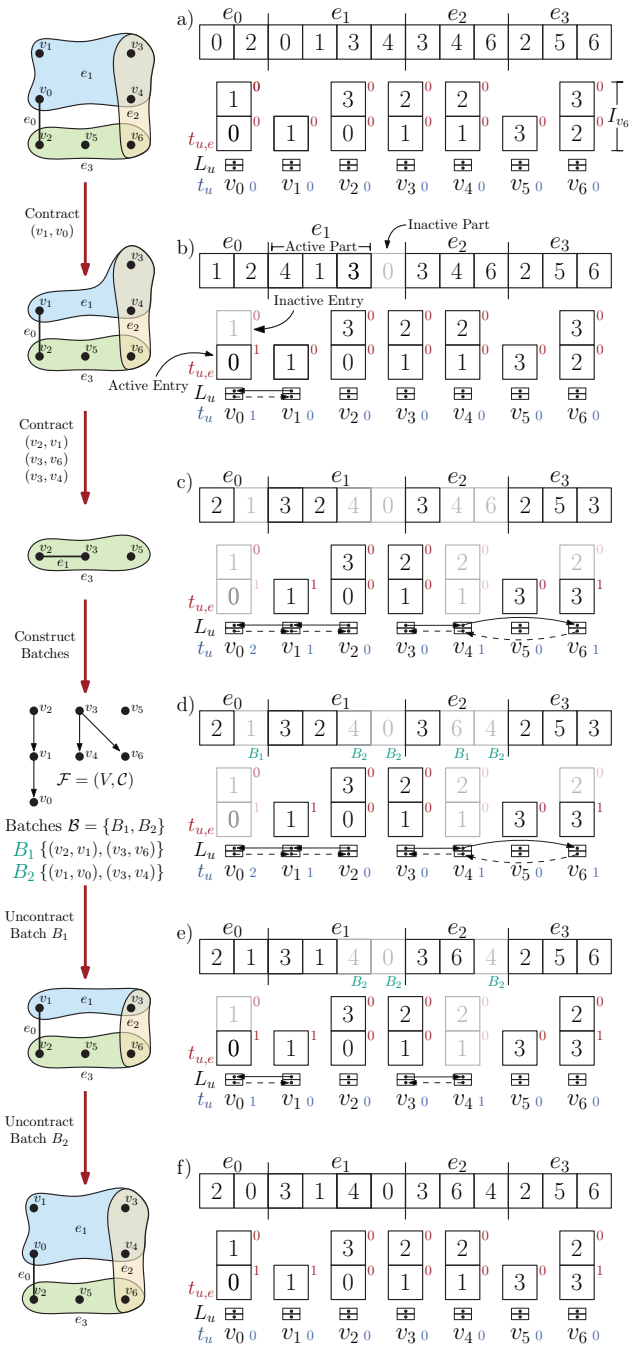



**Figure 6.1:** Contraction and uncontraction operations applied on the dynamic hypergraph data structure.

### 6.1.1 Remove and Restore Incident Nets

To remove and later restore entries from an incident net array $I_w$, we additionally store a counter $t_w$ which counts in how many contractions $I_w$ was modified, as well as a marker $t_{w,e}$ for each entry of $I_w$. The counter and markers are initially set to zero. Entries with markers $\geq t_w$ are *active*, i.e., were not removed yet. To remove a set $X$ of entries from $I_w$, we increment $t_w$ and iterate over the previously active entries of $I_w$ (now marked with $t_w - 1$). If the entry is not in $X$, we set its marker to $t_w$. Otherwise, we swap the entry and its marker to the end of the active part but keep its marker at $t_w - 1$, thereby marking the entry inactive. This maintains the invariant that the entries of $I_w$ are sorted by decreasing markers so that iterating over active entries of $I_w$ has no overhead. In particular, the iterator for $\mathrm{I}(u)$ has a time complexity of $\mathcal{O}(|\mathrm{I}(u)| + |L_u|)$.

To restore entries, we decrement $t_w$ so that we consider entries marked with $t_{w,e} = t_w - 1$ as active again. The restore operations must be performed in reverse order of the remove operations to restore the correct entries. This is one of the dependencies addressed in Sections 6.2 and 6.3.

While the description references a separate incident net array for each node, they are actually organized as an adjacency array. To speed up iteration over incident nets, we store a second doubly-linked list where nodes $w$ without active entries in $I_w$ are removed. This improves the iterator's complexity to $\mathcal{O}(|\mathrm{I}(u)|)$.

> In Figure 6.1 b we contract $v_0$ onto $v_1$. We remove $e_1$ from the incident net array of $v_0$ but keep $e_0$. Therefore, we increment $t_0$ to 1 and set $t_{0,0}$ to 1 while leaving $t_{0,1}$ at 0. In step f (which uncontracts $v_1$), we decrement $t_0$ to 0 and thus mark $e_1$ as active again in the incident net array of $v_0$.

### 6.1.2 Contraction Operation

Algorithm 6.2 shows the pseudocode for contracting a node $v$ onto another node $u$. To edit the pin-lists, we iterate over the incident nets $e \in \mathrm{I}(v)$ and search for the position of $u$ in $e$ (see Line 4). If we do not find $u$, we replace $v$ with $u$ in the pin-list of $e$ in Line 8. Otherwise, we swap $v$ to the end of the active part of $e$ and decrement the current size of $e$ (see Line 5), as well as mark $e$ in a bitset $X$. We use this bitset to remove all nets $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$ from the incident net arrays $I_w$ for all $w \in L_v$ in Line 10.

> In Figure 6.1 b, the contraction $(v_1, v_0)$ replaces $v_0$ with $v_1$ in net $e_0$ and removes $v_0$ from net $e_1$. Afterwards, $v_1$ is incident to net $e_0$ and $e_1$.

To enable concurrent contractions, we use a separate lock for each net to synchronize edits to the pin-lists. In Line 12, the set $\mathrm{I}(u)$ may change due to concurrent contractions onto $u$, which is why it is not thread-safe to initialize $X$ by iterating over $\mathrm{I}(u)$. Therefore, we use the synchronized edits of the pin-lists to mark the nets $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$ in $X$. If multiple nodes contracted concurrently onto $u$ share a net $e$, only the first pin-list edit of $e$ can do the replacement (if $u$ was not already in $e$). All subsequent edits of $e$

**Algorithm 6.2:** Contraction Operation

```
   Input: Contraction (u, v)
 1 c(u) ← c(u) + c(v)
 2 for e ∈ I(v) do                                   // edit pin-lists
 3     lock e
 4     if u in pin-list of e then                    // e ∈ I(u) ∩ I(v)
 5         remove v from pin-list of e
 6         mark e in bitset X
 7     else                                          // e ∈ I(v) \ I(u)
 8         replace v by u in the pin-list of e
 9 for w ∈ L_v do                                    // edit incident net arrays
10     remove active entries in X from I_w
11 lock u
12     append L_v to L_u
```

correctly remove their pins and mark $e$ in their thread-local bitset $X$.

We use a separate lock for each node $u \in V$ to synchronize modifications to $L_u$ and $c(u)$. If $c(u)+c(v)$ exceeds the maximum node weight $c_{\max}$, we discard the contraction. Operations on the incident net arrays $I_w$ for $w \in L_v$ are not synchronized (see Line 10) since $I_w$ is only modified by contracting nodes in $L_v$. These must be finished before the contraction of $v$ starts, and our algorithm in Section 6.2 guarantees this.

### 6.1.3 Uncontraction Operation

Algorithm 6.3 shows the pseudocode for uncontracting a node $v$ that is contracted onto a node $u$. To restore $L_v$ from $L_u$ in Line 3, we additionally store the last node in $L_v$ at the time $v$ is contracted. To restore the incident nets of $v$ that were removed, we iterate over all nodes $w \in L_v$ and decrement the counter $t_w$ in Line 5. This reactivates all entries of $I_w$ that became inactive due to contracting $v$, i.e., had marker $t_{w,e} = t_w$. The other active nets are marked with $t_{w,e} > t_w$, which were not incident to $u$ at the time of contraction and thus not removed. Since we modified the incident nets $I_w$ based on synchronized edits of the pin-lists, the markers encode information on how we have to revert the contraction operation for each net $e \in \mathrm{I}(v)$. To restore the pin-lists, we iterate over all active nets $e \in I_w$, and if $t_{w,e} = t_w$ ($e \in \mathrm{I}(u) \cap \mathrm{I}(v)$), we restore $v$ from the inactive part of the pin-list of $e$ in Line 9 (see $v_4$ in $e_2$ in Figure 6.1 f). Otherwise, if $t_{w,e} > t_w$ ($e \in \mathrm{I}(v) \setminus \mathrm{I}(u)$), we replace $u$ by $v$ in the pin-list of $e$ in Line 15 (see $v_0$ replacing $v_1$ in $e_0$ in Figure 6.1 f). In Line 11 and 16, we update the gain table which we describe in more detail in Section 6.3.4.

In the sequential setting, contractions are undone in the reverse order in which they were performed, so if $v$ was removed, it is the first entry in the inactive part of $e$. In

**Algorithm 6.3:** Uncontraction Operation

```
   Input: Contraction (u, v)
 1 Π[v] ← Π[u]
 2 lock u
 3 |  restore the sublist L_v from L_u
 4 for w ∈ L_v do
 5 |  t_w ← t_w − 1
 6 |  for active entries e ∈ I_w do
 7 |  |  if t_{w,e} = t_w then                          // e ∈ I(u) ∩ I(v)
 8 |  |  |  if test-and-set(e) then
 9 |  |  |  |  restore all pins in the inactive part of e of the current batch B
10 |  |  |  lock e
11 |  |  |  |  updateGainTable(u, v, e)                // see Algorithm 6.4
12 |  |  else                                           // e ∈ I(v) \ I(u)
13 |  |  |  find u in e
14 |  |  |  lock e
15 |  |  |  |  replace u by v in pin-list of e
16 |  |  |  |  updateGainTable(u, v, e)                // see Algorithm 6.4
17 c(u) ← c(u) − c(v)
```

this case, it suffices to increment the current size of $e$ to restore $v$ in Line 9. In our parallel implementation, we uncontract the nodes of a batch $B$ in parallel so that $v$ can be anywhere in the inactive part of $e$'s pin-list. However, after constructing the batches, we sort each pin-list (in particular the inactive entries) by the batches in which the pins are uncontracted (see net $e_2$ in Figure 6.1 d). Then, all pins of $e$ that have to be restored in the current batch can be activated simultaneously by appropriately incrementing the current size of $e$ (as seen in parts e and f of Figure 6.1). Only one thread that triggers the restore case on a net performs the restore operation, which we ensure with an atomic `test-and-set` instruction.

## 6.2 Parallel $n$-level Coarsening

The previous section introduced a dynamic hypergraph data structure to handle concurrent (un)contractions. However, we made several assumptions on the order in which we perform these (un)contractions. For example, we can not contract a node $w$ onto another node $v$ while simultaneously contracting $v$ onto a node $u$. Moreover, we have to revert the contractions in reverse order to restore the incident nets of each uncontracted node correctly. We designed our hypergraph data structure to support any *valid* sequence of (un)contractions. In the following, we will formalize

this by introducing the *contraction forest* from which we derive a parallel schedule of (un)contractions.

### 6.2.1 Contraction Forest

Let us assume that we know the contractions we want to perform in advance, i.e., for each node $v \in V$, we know its representative $\mathsf{rep}[v] = u$ (meaning that $v$ is contracted onto $\mathsf{rep}[v]$). We call the contractions *compatible* if the directed graph $\mathcal{F} = (V, \{(v, \mathsf{rep}[v]) \mid v \in V, \mathsf{rep}[v] \neq v\})$ with edges pointing from the contracted node to its representative is acyclic (underlying undirected edges). If so, we call $\mathcal{F}$ the *contraction forest*. If $v$ does not get contracted, we have $\mathsf{rep}[v] = v$. These are the roots. The *ancestors* of $v$ are the nodes on the unique path towards the root of its tree. The *children* of $v$ are all nodes $w \in V \setminus \{v\}$ with $\mathsf{rep}[w] = v$, and the *descendants* of $v$ are the nodes in the subtree rooted at $v$. Two nodes $v_1$ and $v_2$ are *siblings* if they are children of the same node, i.e., $v_1 \neq \mathsf{rep}[v_1] = \mathsf{rep}[v_2] \neq v_2$.

A node can be contracted as soon as all of its children in $\mathcal{F}$ have been contracted. To obtain parallelism, different subtrees and siblings can be contracted independently, i.e., we traverse $\mathcal{F}$ in a bottom-up fashion in parallel. The contracted hypergraph will be the same regardless of the exact execution order. Only the data structure representation may differ, i.e., which pin was replaced by its representative and which pin was removed.

In the actual algorithm, we do not know $\mathcal{F}$ in advance but start with $\mathsf{rep}[v] = v$ for all $v \in V$. In the following, we describe how we dynamically extend $\mathcal{F}$ in a thread-safe manner that still enables this parallelization scheme, while simultaneously performing the contractions.

### 6.2.2 Handling Contraction Dependencies

Contracting a node $v$ onto another node $u$ does not break compatibility with existing contractions if it satisfies the following two conditions: (i) $\mathcal{F}$ must remain a rooted forest and (ii) the contraction operation of $u$ onto its parent $\mathsf{rep}[u]$ must not have started yet (see Algorithm 6.2). More precisely, adding edge $(v, u)$ to $\mathcal{F}$ must not induce a cycle, and $v$ must still be a root, i.e., $\mathsf{rep}[v] = v$. If $u$'s contraction has started, contracting $v$ onto $u$ would introduce inconsistencies as $u$ may have been replaced by $\mathsf{rep}[u]$ in some of its incident nets. This has two consequences: First, if $u$'s contraction has started, we instead contract $v$ onto a suitable ancestor of $u$. Secondly, if there are unfinished contractions onto $v$, we cannot contract $v$ right away. Note that we explicitly allow multiple concurrent contractions onto the same node.

We use a zero-initialized array $\mathsf{pending}$, where $\mathsf{pending}[x]$ stores the number of nodes $y$ with $\mathsf{rep}[y] = x$ whose contraction is not finished. If $\mathsf{pending}[x] = 0$, it is safe to contract $x$. If additionally $\mathsf{rep}[x] \neq x$, we assume that the contraction of $x$ onto $\mathsf{rep}[x]$ has started. The entries $\mathsf{rep}[x]$ and $\mathsf{pending}[x]$ are only modified while holding a node-specific lock for $x$. The following procedure takes a contraction $(u, v)$ as input

and ensures at some point that $v$ is contracted onto a node $w$, where $w$ is either $u$ (most common case) or an ancestor of $u$.

First, $v$ is locked so that no other thread can write to $\mathsf{rep}[v]$. If $\mathsf{rep}[v] \neq v$, we discard the contraction $(u, v)$, as another thread has already selected a representative for $v$ and will run this algorithm. Otherwise, we walk the path towards the root of $u$'s tree in $\mathcal{F}$ by chasing the $\mathsf{rep}$ entries to find the lowest ancestor $w$ of $u$, for which either $\mathsf{rep}[w] = w$ or $\mathsf{pending}[w] > 0$, i.e., the contraction of $w$ has not started. If $v$ is found on this path, the contraction is discarded, as it would add a cycle to $\mathcal{F}$.

If no cycle is found, $w$ is locked, and we check $\mathsf{rep}[w]$ and $\mathsf{pending}[w]$ again. If they changed, we release $w$ and keep walking up to find an ancestor of $w$. Otherwise, $w$ is the desired candidate. We still finish the walk up to the root to check for cycles. If no cycles are found, we set $\mathsf{rep}[v] = w$, increment $\mathsf{pending}[w]$ by 1, and unlock $v$ and $w$.

The thread that reduces $\mathsf{pending}[v]$ to 0 is responsible for contracting $(w, v)$. If $\mathsf{pending}[v] = 0$ already, we contract $v$ onto $w$ using Algorithm 6.2, and subsequently decrement $\mathsf{pending}[w]$ by one. If this reduces $\mathsf{pending}[w]$ to 0 and $\mathsf{rep}[w] \neq w$, we recursively apply this process to $(\mathsf{rep}[w], w)$. If instead $\mathsf{pending}[w] > 0$, we are done.

### 6.2.3 Removing Identical Nets

There is one last detail left for the coarsening phase. We remove single-pin and identical nets after each coarsening pass (see Line 7 in Algorithm 6.1) using the algorithm described in Section 5.1 (same as in Mt-KaHyPar-D(-F)) but adapt it to the dynamic hypergraph data structure. Removing this redundant information speeds up the other algorithmic components. KaHyPar [Sch+16a; Sch20] removes these nets directly after each contraction operation. Doing this on-the-fly in the parallel setting would introduce additional dependencies for the batches in the uncoarsening phase since we have to restore these nets at the time at which they become identical. Furthermore, the pin-list of a net may change while we simultaneously try to identify identical nets. Thus, we would have to lock all incident nets of a node to prevent concurrent modifications of the pin-lists, which is why we decided against it.

## 6.3 Parallel $n$-level Uncoarsening

For the uncoarsening phase, our goal is to create a sequence of batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$, where $\mathcal{B}$ is a partition of the contracted nodes into disjoint sets such that $\forall B \in \mathcal{B} : |B| \approx b_{\max}$. The batch size $b_{\max}$ is an input parameter that interpolates between scalability (high values) and the traditional $n$-level scheme that uncontracts only a single node on each level ($b_{\max} = 1$) which is inherently sequential. Each batch $B_i$ will be chosen such that we can uncontract the nodes $v \in B_i$ in parallel. Refinement is applied after each batch. Processing $B_i$ will resolve the last dependencies required to uncontract the next batch $B_{i+1}$. Clearly, the uncontraction of a node $v$ can only start once the uncontraction of $\mathsf{rep}[v]$ is finished, i.e., all of its ancestors are uncontracted. Therefore, we construct the batches via a top-down traversal of $\mathcal{F}$. However, we have

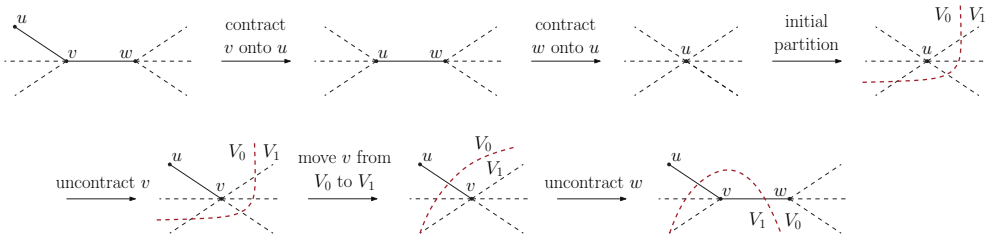



**Figure 6.2:** Example of an uncontraction that increases the cut size (uncontracting $w$ increases the cut by one). This occurs because we did not uncontract the siblings $v$ and $w$ of $u$ in reverse order of contraction.

introduced additional ordering dependencies between siblings in $\mathcal{F}$ due to the way we perform the replacement edits of the pin-lists.

### 6.3.1 Sibling Uncontraction Dependencies

Figure 6.2 illustrates a situation demonstrating that it is necessary to uncontract siblings in the contraction forest $\mathcal{F}$ in reverse order of contraction. Consider a hyperedge $e = \{v, w\}$. If we contract $v$ onto a node $u$, then $u$ replaces $v$ in $e$ ($e = \{u, w\}$). If we subsequently contract $w$ onto $u$, then $w$ is moved to the inactive part of $e$ ($e = \{u\}$). We now uncontract node $v$ before $w$, which replaces $u$ in $e$ with $v$ again ($e = \{v\}$). If $u$ and $v$ are in different blocks of the partition, uncontracting $w$ would make hyperedge $e$ a cut net since $w$ is assigned to the block of $u$ and $\Pi[u] \neq \Pi[v]$. This would violate a fundamental property of the multilevel paradigm that we can project a partition to a finer hypergraph in the hierarchy with the same solution quality. Thus, we require that all sibling contractions are uncontracted in reverse order of contraction. Since we perform contractions in parallel, we additionally require that contractions that happen *at the same time* must be reverted in the same batch.

To detect time overlaps, we atomically increment a counter before starting and after finishing a contraction operation. For each contracted node $v$, this yields an interval $[s_v, e_v]$ with start time $s_v$ and end time $e_v$. If the intervals of two nodes overlap, we assume they were contracted at the same time, otherwise one is strictly earlier than the other. Among siblings, we need to compute the transitive closure of nodes with overlapping intervals and order them decreasingly if one is strictly earlier than the other. Since comparing for equality with interval overlaps is not transitive, we sort them in decreasing order of $e_v$. Then, the nodes in a transitive closure are ordered consecutively and can be found with a rightward sweep from the first interval by checking whether the next interval overlaps with the union of intervals in the closure so far and extending the union interval if they overlap.

### 6.3.2 Batch Construction Algorithm

After the coarsening phase, we need to construct batches of contracted nodes to uncontract in parallel during the uncoarsening phase. We traverse $\mathcal{F}$ top-down in breadth-first search (BFS) order, using two FIFO queues: $Q$ for the current BFS layer and $Q'$ for the next layer. Additionally, we have a batch $B_{\text{cur}}$ that we are currently adding contracted nodes to. If $|B_{\text{cur}}| \geq b_{\max}$ or we get to the next BFS level, we append $B_{\text{cur}}$ to $\mathcal{B}$, and proceed with a new empty batch. We additionally add a sentinel batch $B = \emptyset$ to $\mathcal{B}$ whenever all contractions of a coarsening pass are proccesed. This signalizes the uncontraction algorithm to restore single-pin and identical nets (see Line 11 and 18 in Algorithm 6.1).

$Q$ and $Q'$ store elements $(u, v_i)$, where $u$ is a node and $v_i$ is the $i$-th child of $u$ in the sorted order described in Section 6.3.1 (sorted by finish time of the contraction operation). For elements in $Q$, we maintain the invariant that $u$ is uncontracted in a batch before $B_{\text{cur}}$, so that its children can be added to $B_{\text{cur}}$. Furthermore, $v_i$ is the first child of $u$ that has not been added to any batch yet. To initialize $Q$, we insert all entries $(r, w_1)$, where $r$ is a root of $\mathcal{F}$ and $w_1$ is the first child of $r$.

We pop elements from $Q$ until it is empty, and then swap it with $Q'$. Now, let $(u, v_i)$ be the current element we popped from $Q$, and let $T$ denote the transitive closure of $v_i$, as described in the previous section. For each $v \in T$, we add $v$ to $B_{\text{cur}}$, and push $(v, w_1)$ to $Q'$, where $w_1$ is the first child of $v$. Additionally, we push $(u, v_j)$ to the end of $Q$, where $v_j$ is the first child of $u$ outside $T$, if any. The reason for this reinsertion is to minimize the number of contractions in the same batch which have the same representative. This reduces synchronization overheads during uncontraction operations and reduces the overlap of local searches. The complexity of this algorithm is $\mathcal{O}(|V|)$.

We obtain parallelism by traversing the different trees of $\mathcal{F}$ concurrently. $\mathcal{F}$ has as many trees as nodes left in the coarsest hypergraph, so at least around $160 \cdot k$. To approximate a BFS order across trees, we perform one BFS per thread, which is initialized with the different roots (and first children) assigned to the thread. The threads collaborate on filling batches, and we keep multiple batches open, as threads may progress at different rates. The parallelization is work-efficient in the theoretical sense, as the work is still $\mathcal{O}(|V|)$. The span of the algorithm is linear in the maximum tree size of $\mathcal{F}$.

### 6.3.3 Refinement

We run our parallel label propagation and FM algorithm after uncontracting a batch of nodes. Mt-KaHyPar-D(-F) initializes the local searches with the boundary nodes of the partition. However, this would incur too much overhead with $\mathcal{O}(|V|)$ levels since using all boundary nodes can lead to quadratic running times. Hence, the refinement should be *localized*, i.e., focus on areas close to the uncontracted nodes. Instead of considering all boundary nodes, we only use the boundary nodes from the current batch. Note that the searches may expand to nodes not in the batch. We do not run

our flow-based refinement algorithm here since it solves large flow problems around the cut of two adjacent block pairs and therefore can not be used as a localized refinement algorithm.

We complement the localized refinement with a refinement pass on the entire hypergraph after restoring single-pin and identical nets. Here, we run our parallel FM (initialized with all boundary nodes) and flow-based refinement algorithm. We do not run the label propagation algorithm since it gave no quality benefits in preliminary experiments. The *global* refinement step is performed on approximately a logarithmic number of levels similar to the refinement in Mt-KaHyPar-D(-F).

### 6.3.4 Gain Table Maintenance

The FM algorithm moves nodes greedily according to a gain value. Our parallel implementation uses a gain table to calculate the highest gain move for each node. Recall that the gain $g_u(V_j)$ of moving a node $u$ from its current block $\Pi[u]$ to a target block $V_j$ can be formulated as follows (for the connectivity metric):

$$\begin{aligned} b(u) &= \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi[u]) = 1\}) \\ p(u, V_j) &= \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, V_j) \geq 1\}) \\ g_u(V_j) &= b(u) - \omega(\mathrm{I}(u)) + p(u, V_j) \end{aligned}$$

The gain table described in Section 4.1.2 stores the benefit term $b(u)$ and penalty term $p(u, V_j)$ for each node $u \in V$ and block $V_j \in \Pi$. We use delta-gain updates to maintain both terms during an FM pass (see Algorithm 4.3). In the multilevel algorithm, the gains are initialized from scratch on each level, which incurs too much overhead with $O(n)$ levels. Instead, we update the entries based on the uncontraction operations described in Section 6.1.

Algorithm 6.4 shows the pseudocode of the gain table update operation after reverting a contraction $(u, v)$ for a net $e \in \mathrm{I}(v)$. The algorithm implements the `updateGainTable`$(u, v, e)$ procedure shown in Algorithm 6.3. The values $b(v)$ and $p(v, V_j)$ of the uncontracted node $v$ are initialized to zero. The corresponding values for $u$ are still correct from the previous levels. Updates of the benefit and penalty terms of $v$ are done exclusively by one thread, and updates to $u$ can happen from multiple threads, which is why we use atomic `fetch-and-add` instructions. Analogously to the uncontraction operation, on each net $e \in \mathrm{I}(v)$, we distinguish two cases for the gain updates due to uncontracting $v$: (i) whether $v$ replaces $u$ in $e$ (see Line 2), or (ii) whether $u$ and $v$ are both incident to $e$ after the uncontraction (see Line 8).

If $v$ replaces $u$, we subtract $\omega(e)$ from $p(u, V_j)$ and add it to $p(v, V_j)$ for all $V_j \in \Lambda(e)$. Additionally, if $\Phi(e, \Pi[u]) = 1$, we subtract $\omega(e)$ from $b(u)$ and add it to $b(v)$. In this case, $\Phi(e, \Pi[u])$ does not change, since $v$ is now a pin of $e$, but $u$ no longer is.

In the second case, where $u$ and $v$ are both incident to $e$ after the uncontraction, we increase $\Phi(e, \Pi[u])$ by one since $v$ is assigned to $\Pi[u]$ and now both $u$ and $v$ are pins of $e$ in $\Pi[u]$. If this increased the value to 2, we have to reduce $b(u)$ by $\omega(e)$, since

**Algorithm 6.4:** Uncontraction Gain Table Update

**Input:** Contraction $(u, v)$ and a net $e$ where the uncontraction operation of $(u, v)$ is already applied to its pin-list.

```
// We call this function while holding a net-specific lock for e (see Algorithm 6.3)
1  V_i ← Π[u]
2  if u ∉ e and v ∈ e then                                   // v replaces u in e
3      if Φ(e, V_i) = 1 then
4          fetch-and-add(b(u), −ω(e))
5          fetch-and-add(b(v), ω(e))
6      unlock(e)                                             // see Algorithm 6.3
7      for V_j ∈ Λ(e) do fetch-and-add(p(u, V_j), −ω(e))
8  else                                                      // u and v are both incident to e now
9      Φ(e, V_i) ← Φ(e, V_i) + 1
10     if Φ(e, V_i) = 2 then
           // u may be already replaced in net e by another node w
11         w ← find pin (≠ v) of net e part of block V_i
12         fetch-and-add(b(w), −ω(e))
13     unlock(e)                                             // see Algorithm 6.3
14 for V_j ∈ Λ(e) do fetch-and-add(p(v, V_j), ω(e))
```

moving $u$ out of $\Pi[u]$ would no longer remove $\Pi[u]$ from $\Lambda(e)$. Additionally, we add $\omega(e)$ to each $p(v, V_j)$ for $V_j \in \Lambda(e)$.

The connectivity set $\Lambda(e)$ of a net $e$ does not change during uncontractions, since $\Phi(e, V_i)$ is only modified in the second case, where $\Phi(e, V_i) \geq 1$. Therefore, iteration over $\Lambda(e)$ is thread-safe. Modification and reads on $\Phi(e, \Pi[u])$ are thread-safe because they are protected by a net-specific lock. The parallel setting introduces one more intricacy. In the second case, $u$ might have been replaced in the active part of $e$ by some $w \neq v$ due to a concurrent uncontraction $(u, w)$. Therefore, we search for a pin $w$ in the active part of $e$ with $\Pi[w] = \Pi[u]$ and $w \neq v$, and then update $b(w)$ instead of $b(u)$. If $u$ was not replaced, we simply find $w = u$. Since we do this while holding the lock for $e$, we can guarantee that if $\Phi(e, \Pi[u])$ is incremented to 2, there are only two nodes of $\Pi[u]$ in the active part of $e$.

## 6.4 Insights into $n$-Level Partitioning

We provide two configurations of the $n$-level partitioning algorithm: Mt-KaHyPar-Q (-**Q**uality) and Mt-KaHyPar-Q-F (-**Q**uality-**F**lows). Mt-KaHyPar-Q-F extends Mt-KaHyPar-Q with flow-based refinement. Both partitioners aim for high solution quality.

We now evaluate the different algorithmic components of Mt-KaHyPar-Q(-F) in more detail. Section 6.4.1 discusses the configuration of the algorithm. Most important will

be the choice of the maximum batch size parameter $b_{\max}$, as it interpolates between scalability and the inherently sequential $n$-level scheme. We then analyze the scalability and running times of the different components in Section 6.4.2 and 6.4.3. Section 6.4.4 concludes the experimental evaluation by comparing the traditional multilevel and $n$-level partitioning scheme.

### 6.4.1 Algorithm Configuration

Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F) share large parts of our codebase, and therefore both use similar parameter settings as depicted in Table 5.1 in Section 5.6. The parameters for community detection, label propagation, flow-based refinement, and initial partitioning (uses the $n$-level (un)coarsening scheme) were adopted without modifications.

Mt-KaHyPar-Q(-F) implements a different coarsening algorithm but still uses the heavy-edge rating function, the same contraction limit ($160k$ nodes) and maximum allowed node weight ($\frac{c(V)}{160 \cdot k}$). The refinement algorithms are reused but are used in a different context as in Mt-KaHyPar-D. After uncontracting a batch of nodes, we run a localized version of the label propagation and FM algorithm only initialized with the boundary nodes of the current batch. Since significantly fewer boundary nodes are available here, we set the number of seed nodes for the localized FM searches to 5 instead of 25 (all other parameters are the same as in Mt-KaHyPar-D). To improve scalability, we uncontract multiple batches until the number of restored boundary nodes exceeds a threshold $\beta = \max(b_{\max}, 50 \cdot t)$, where $t$ is the number of threads, and only then perform localized refinement. For initial partitioning, we set $\beta = 0$ since the nested parallelism available from recursive bipartitioning suffices.

We use a *global* refinement pass performed on the entire hypergraph after reverting all contractions from a coarsening pass (using the FM algorithm and flow-based refinement). Since Mt-KaHyPar-Q aims for high solution quality, we run the localized and global refinement step multiple times on each level until the relative improvement made in a pass is less than 0.25%.

**Maximum Batch Size.** In Figure 6.3, we compare the impact of different batch size values $b_{\max} \in \{1, 100, 200, 1000, 10000\}$ on the solution quality and running time on our parameter tuning benchmark set $\mathrm{L_P}$ on machine A. For this experiment, the minimum number of boundary nodes $\beta$ is set to 0, so that localized refinement is performed after each batch, and thus scalability depends only on $b_{\max}$. Each run uses 20 threads.

We see that the different values yield roughly the same performance. Increasing the batch size from $b_{\max} = 100$ to $b_{\max} = 200$ slightly decreases the solution quality. However, the differences are not statiscally significant (Wilcoxon signed rank test: $Z = -1.2267$ and $p = 0.2199$). The fastest configuration is $b_{\max} = 1000$ (geometric mean running time 14.81s), which is 1.64 times faster than $b_{\max} = 100$ (24.34s) and 11.42 times faster than $b_{\max} = 1$ (169.16s) on average. We therefore choose $b_{\max} = 1000$.

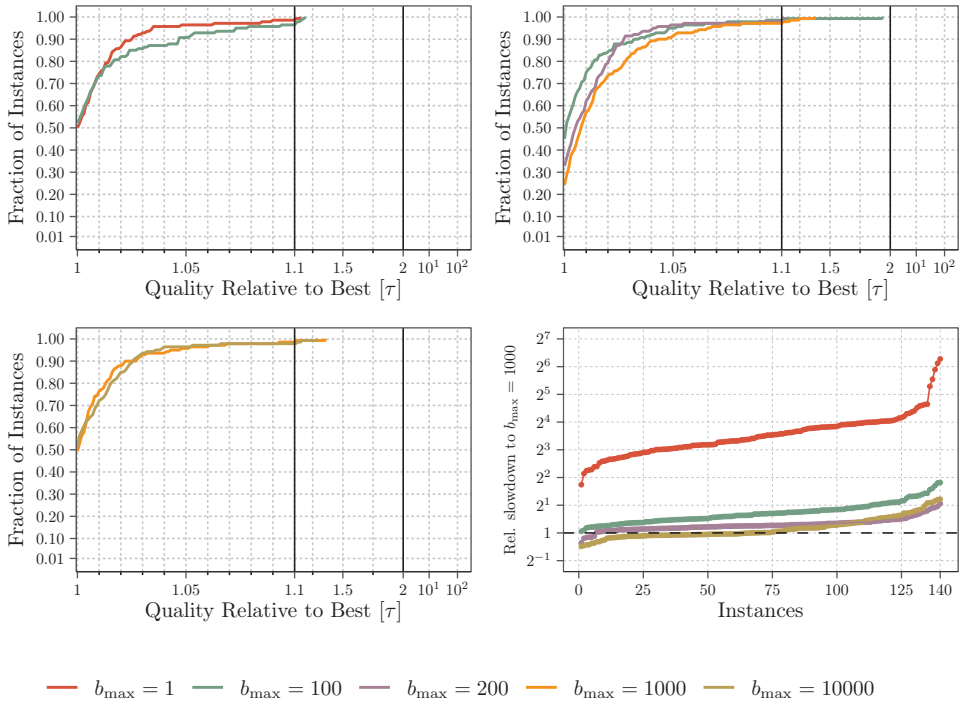



**Figure 6.3:** Performance profiles and running times comparing Mt-KaHyPar-Q with different batch size values $b_{\max}$.

## 6.4.2 Scalability

In Figure 6.4 and Table 6.1, we show self-relative speedups for several algorithmic components of Mt-KaHyPar-Q with varying number of threads $t \in \{4, 16, 64\}$. We run the scalability experiments for Mt-KaHyPar-Q on a subset of set $\mathrm{L_{HG}}$ (77 out of 94 hypergraphs) that contains all hypergraphs on which Mt-KaHyPar-Q 64 was able to finish in under 800 seconds for all $k \in \{2, 8, 16, 64\}$. This experimental evaluation is based on the data from the corresponding publications of Mt-KaHyPar-Q [Got+22a]. We did not rerun these experiments since it took roughly six weeks on machine B. We omit scalability experiments with Mt-KaHyPar-Q-F due to the long time requirements and because flow-based refinement is used in the same context in Mt-KaHyPar-D-F (see Section 5.7.3).

The overall geometric mean speedup of Mt-KaHyPar-Q is 3.7 for $t = 4$, 11.9 for $t = 16$ and 23.7 for $t = 64$. If we only consider instances with a single-threaded running time $\geq 100$s, we achieve a geometric mean speed up of 25.9 for $t = 64$.

Coarsening and batch uncontractions both have similar speedups (i.e., both 11.2 for

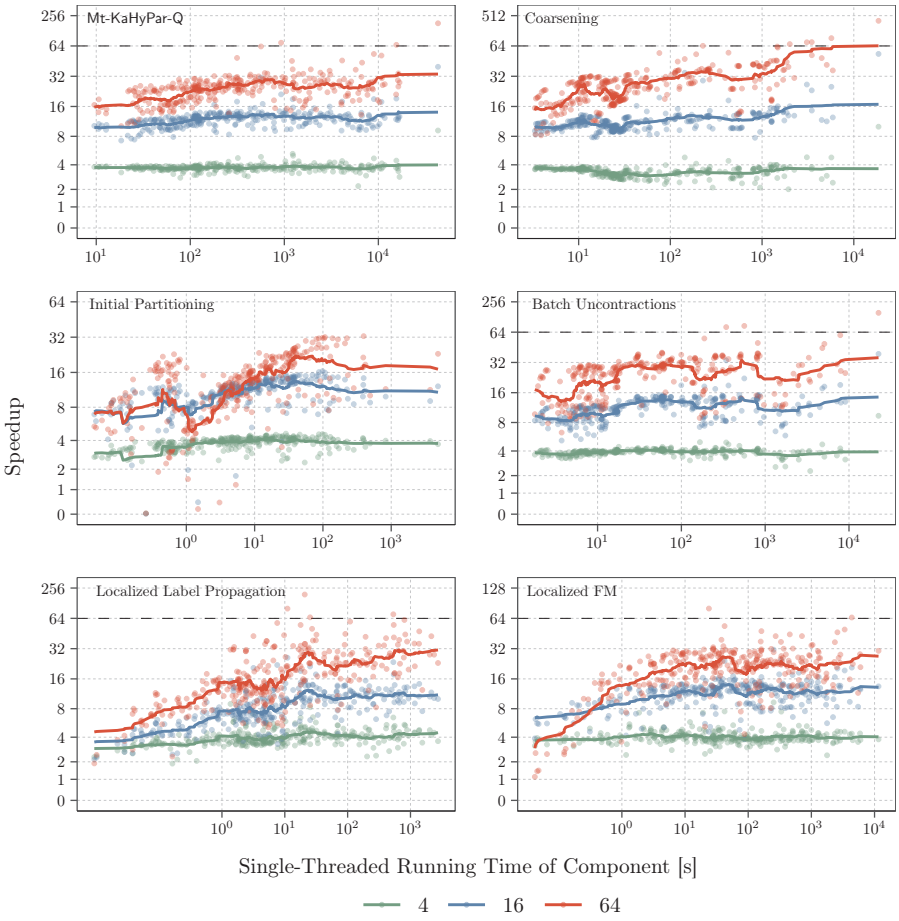



**Figure 6.4:** Speedups of Mt-KaHyPar-Q and its different algorithmic components.

$t = 16$), whereas coarsening performs better for $t = 64$ (25.4 vs 23.2). Both localized refinement algorithms yield reliable speedups for an increasing number of threads (both $\geq 20$ for $t = 64$ and instances with sequential time $\geq 100$s). Initial partitioning shows the least promising speedups of all components, but is substantially faster, running in less than 100 seconds on 92.2% of the instances for $t = 1$. On some instances we obtain super-linear speedups (up to 412), which are caused by (un)coarsening in which non-deterministic behavior causes varying running times.

Figure 6.5 shows that increasing the number of threads adversely affects the solution quality of Mt-KaHyPar-Q, but only by a very small amount. In a master thesis [Lau21a], which we supervised, we implemented an *asynchronous uncoarsening* scheme in which

**Table 6.1:** Geometric mean speedups over all instances and instances with a single threaded running time $\geq$ 100s for total partition time (T), coarsening (C), initial partitioning (IP), batch uncontractions (BU), localized label propagation (LP) and FM. The last row shows the percentage of instances with a single-threaded running time $\geq$ 100 seconds.

| Number of Threads | | T | C | IP | BU | LP | FM |
|---|---|---|---|---|---|---|---|
| All | 4 | 3.7 | 3.3 | 3.7 | 3.9 | 3.9 | 4 |
| | 16 | 11.9 | 11.2 | 10.1 | 11.2 | 8.1 | 11.2 |
| | 64 | 23.7 | 25.4 | 11.6 | 23.2 | 15.9 | 19 |
| $\geq$ 100s | 4 | 3.7 | 3.4 | 3.8 | 3.9 | 4.1 | 4 |
| | 16 | 12.5 | 13 | 11.2 | 12.1 | 10.8 | 12 |
| | 64 | 25.9 | 36.4 | 18.2 | 25.6 | 26.5 | 21.8 |
| Inst. $\geq$ 100s [%] | | 71.8 | 31.5 | 7.8 | 34.4 | 18.5 | 39 |

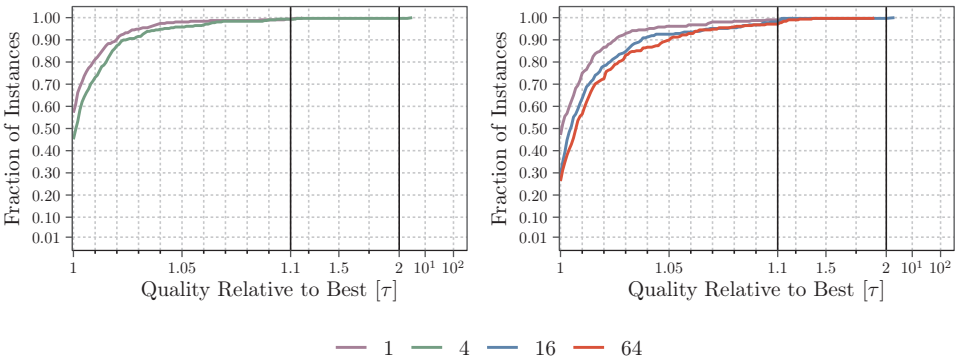



**Figure 6.5:** Performance profiles comparing the solution quality of Mt-KaHyPar-Q with an increasing number of threads on set $L_{HG}$.

uncontractions and localized refinement happen concurrently (no batches required). Here, the differences in solution quality were even more pronounced with an increasing number of threads. We observed that the uncontraction gain table updates increased interference with the localized searches. Diversifying the search by trying to keep the neighborhoods of concurrently uncontracted nodes disjoint has significantly improved the solution quality. We assume that such an approach may also work well for Mt-KaHyPar-Q by assembling uncontractions of independent regions of the hypergraph in a batch.

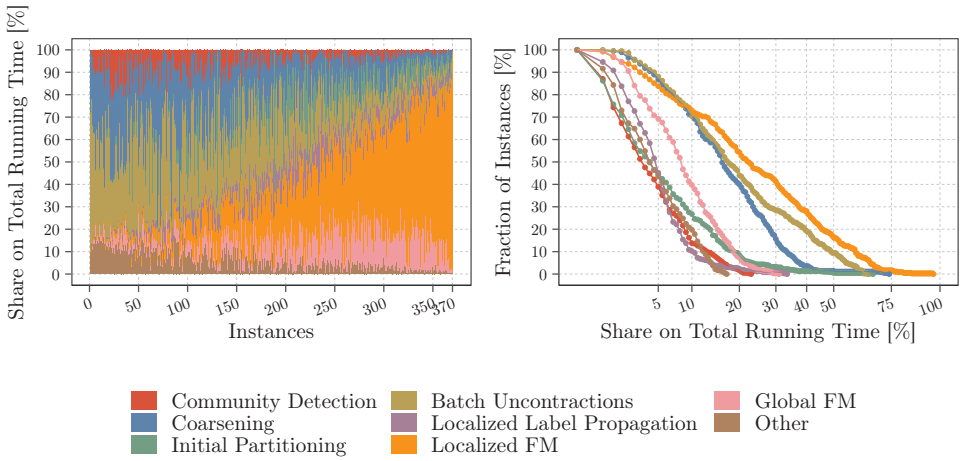



**Figure 6.6:** Running time shares of different components on the total execution time of Mt-KaHyPar-Q.

### 6.4.3 Running Times of Components

We now analyze the running times of the different algorithmic components of Mt-KaHyPar-Q on set $L_{HG}$ in more detail (using 64 threads). Figure 6.6 shows two different types of plots illustrating the share of each component on the total partitioning time. The first plot (left) uses a bar plot to visualize the running times of each component for each instance. In these plots, we take the execution time of the most time-consuming component and sort the instances according to it. The second plot (right) is similar to the performance profiles. It shows the percentage of instances (y-axis) for which the share of a component on the total partitioning time is $\geq x\%$.

As we can see, the coarsening phase (share on the total partitioning time is 16% in the median), the batch uncontraction operations (17.3%), and the localized FM algorithm (22.1%) are the most time-consuming components of Mt-KaHyPar-Q. Outliers in the running time can be explained when we analyze specific instance classes in more detail. The largest share on total execution time for Dual instances is attributed to localized FM (28.4%) where large hyperedges slow down the algorithm (explained in Section 5.7.4). The running time of batch uncontractions is most pronounced on Primal and Literal instances (38.1%). The share of coarsening is surprisingly low on these instances (16.2%), even though the computational complexity of contractions and uncontractions is the same. However, batch uncontractions additionally require updating the gain table, and Primal and Literal instances have many small nets that trigger uncontraction gain table updates more often.

The share of the community detection (3.1%) and global FM algorithm (8.2%), which made up a large fraction of the total partitioning time in Mt-KaHyPar-D, is less pronounced in Mt-KaHyPar-Q. The running time of initial partitioning (3.7%) and localized label propagation (6.7%) is negligible on most of the instances.

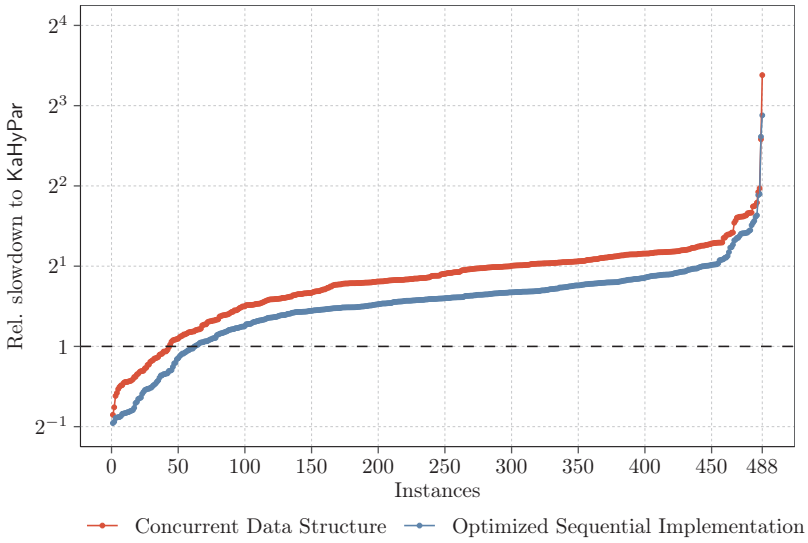



**Figure 6.7:** The plot shows the slowdowns of our concurrent hypergraph data structure relative to KaHyPar's hypergraph data structure when we apply a sequence of contractions sequentially on them for all instances of benchmark set $M_{HG}$.

**Overhead of the Concurrent Hypergraph Data Structure.** We presented a dynamic hypergraph data structure with low memory overheads that can perform concurrent (un)contractions. It uses doubly-linked lists to link the incident nets of two contracted nodes instead of copying them. This requires no additional memory but may add some overheads since the incident nets of a node are no longer stored in a consecutive memory region. We therefore study the overhead of the data structure by comparing it to the hypergraph data structure used in KaHyPar [Sch+16a; Akh+17a] (explained in Section 3.3.2).

In this experiment, we use our $n$-level coarsening algorithm to derive a schedule of contractions, which we then apply to our new and KaHyPar's data structure sequentially. In addition to our proposed concurrent hypergraph data structure, we also evaluate an optimized sequential implementation that is lock-free and does not use the contraction forest. We run the experiment on benchmark set $M_{HG}$ and perform three repetitions per hypergraph. We measure the time required to perform all contractions on the corresponding data structure and use the arithmetic mean over all repetitions as the total time.

Figure 6.7 shows the slowdowns of applying all contractions to our new hypergraph data structure relative to KaHyPar's hypergraph data structure. As we can see, the slowdown of our concurrent data structure is between 0.74 and 3.09 for 95%

of the instances. There are only two instances with a slowdown $\geq 4$. On average, KaHyPar's data structure is faster by a factor of 1.74. The contraction forest and locking mechanisms slow down our hypergraph data structure by a factor of 1.21 on average (compared to the optimized sequential implementation).

Initially, we started with a concurrent implementation of KaHyPar's data structure but were not able to partition some of our largest hypergraphs since we ran out of memory on a machine with 1TB RAM. After replacing it with our low memory data structure, we observed a slowdown of the overall partitioning algorithm by 5% on average. However, we are unable to repeat the experiment since we made changes incompatible with the previous implementation. Moreover, we achieve near-optimal speedups with a moderate number of threads (see Figure 6.4) that outweigh its sequential overheads.

### 6.4.4 Comparison to Multilevel Partitioning

Our multilevel and $n$-level partitioning algorithms differ in how contractions and uncontractions are performed. Most of the algorithmic components are shared between both with only minor adjustments. This allows us to compare both partitioning schemes directly and answer whether or not more levels lead to better partitioning quality in practice.

**Solution Quality and Running Times.** Figure 6.8 – 6.10 compare the solution quality and running times of Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F) on set $\mathrm{M_{HG}}$ and set $\mathrm{L_{HG}}$.

On set $\mathrm{M_{HG}}$, Mt-KaHyPar-Q computes partitions with better connectivity than Mt-KaHyPar-D on 87.5% of the instances (median improvement is 1.9%) while it is slower by a factor of 3.37 on average (geometric mean running time 2.99s vs 0.89s). However, the differences become less pronounced when both partitioners use flow-based refinement. The median improvement of Mt-KaHyPar-Q-F over Mt-KaHyPar-D-F is 0.6%, while it is slower by a factor of 1.86 on average (geometric mean running time 5.08s vs 2.73s). We note that Mt-KaHyPar-D-F (2.73s) is faster than Mt-KaHyPar-Q (2.99s) and computes better partitions on 87.3% of the instances (median improvement is 2%). However, as we will see in Chapter 8, Mt-KaHyPar-Q(-F) produces comparable partitions to the sequential $n$-level partitioner KaHyPar [HS17a; Got+20] (uses similar algorithmic components), while being an order of magnitude faster with ten threads.

If we compare the solution quality of the partitioners on set $\mathrm{L_{HG}}$, we see that the results are similar to those on set $\mathrm{M_{HG}}$. However, the differences in the running times are more pronounced on set $\mathrm{L_{HG}}$. Here, Mt-KaHyPar-Q is slower than Mt-KaHyPar-D by a factor of 6.48 on average (geometric mean running time 30.1s vs 4.64s). Mt-KaHyPar-D-F (30.44s) has a comparable running time to Mt-KaHyPar-Q (30.1s) and is faster than Mt-KaHyPar-Q-F (58.5s).

**Effectiveness Tests.** The experimental results suggest that traditional multilevel partitioners can achieve the same solution quality as $n$-level partitioners when flow-based refinement is used. However, our $n$-level partitioner Mt-KaHyPar-Q produces

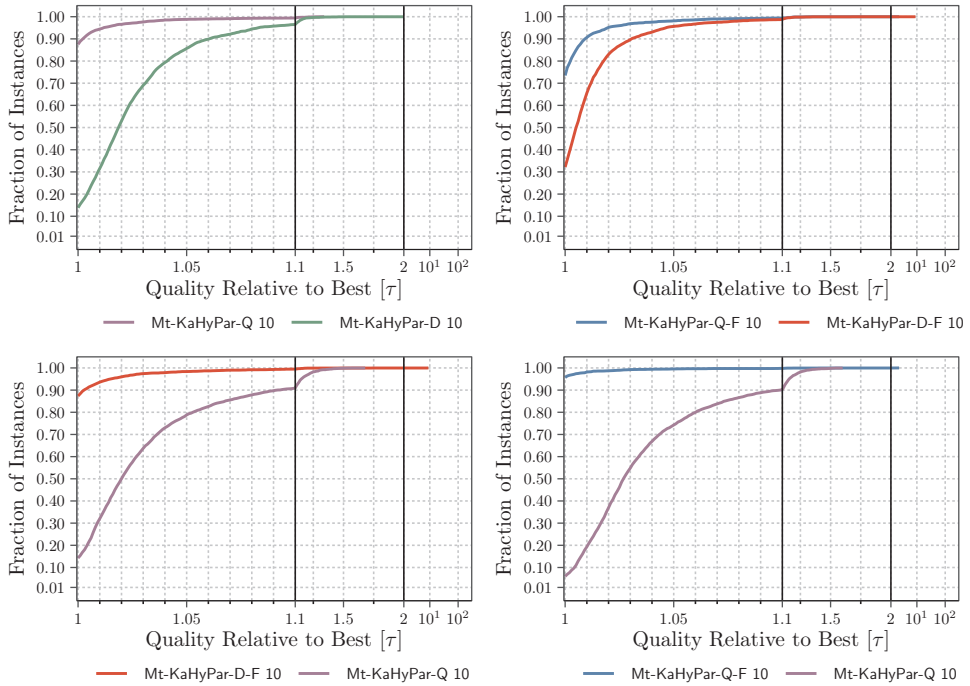



**Figure 6.8:** Performance profiles comparing Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F) on set $M_{HG}$.

partitions with better solution quality than our multilevel partitioner Mt-KaHyPar-D without flow-based refinement. Since Mt-KaHyPar-D is faster than Mt-KaHyPar-Q, we now compare both using the effectiveness tests presented in Section 2.4.3. Here, we give the faster algorithm more time to perform additional repetitions until its expected running time equals the running time of the slower algorithm.

Figure 6.11 shows the effectiveness tests on virtual instances comparing both partitioners with (right) and without flow-based refinement (left) on set $M_{HG}$. We see that the difference in solution quality between Mt-KaHyPar-D and Mt-KaHyPar-Q is only marginal. Hence, if multiple restarts are used, our multilevel achieves the same solution quality as our $n$-level partitioner.

**The Future of $n$-Level Partitioning.** These negative results are disappointing but are still an important scientific result. The complexity of today's partitioning systems makes it difficult to compare different techniques due to the "lack of documented key implementation details in the literature" [CKM00a]. The $n$-level partitioning scheme is used in the highest-quality sequential partitioner KaHyPar [Sch20] and requires high engineering efforts to implement efficiently. Our experimental results show that the

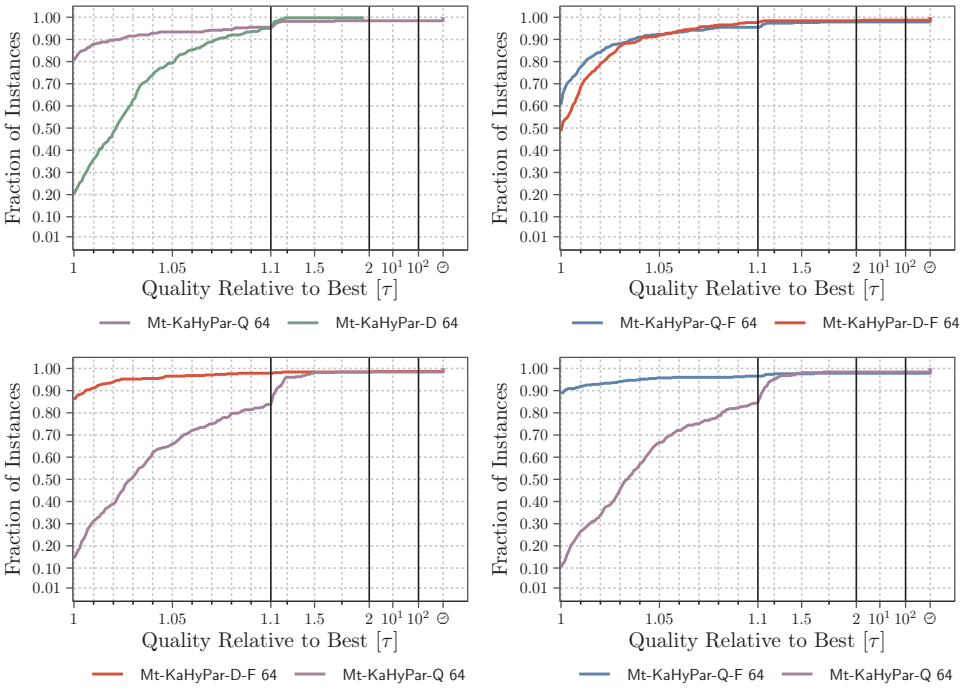



**Figure 6.9:** Performance profiles comparing Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F) on set $\mathrm{L_{HG}}$.

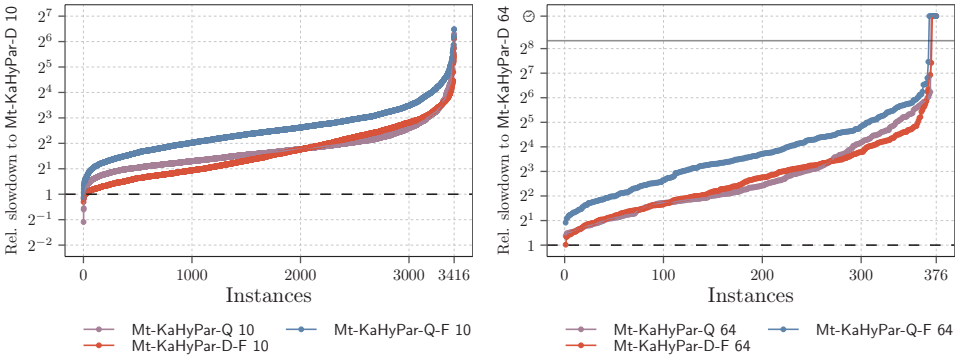



**Figure 6.10:** Running times of Mt-KaHyPar-D-F and Mt-KaHyPar-Q(-F) relative to Mt-KaHyPar-D on set $\mathrm{M_{HG}}$ (left) and set $\mathrm{L_{HG}}$ (right).

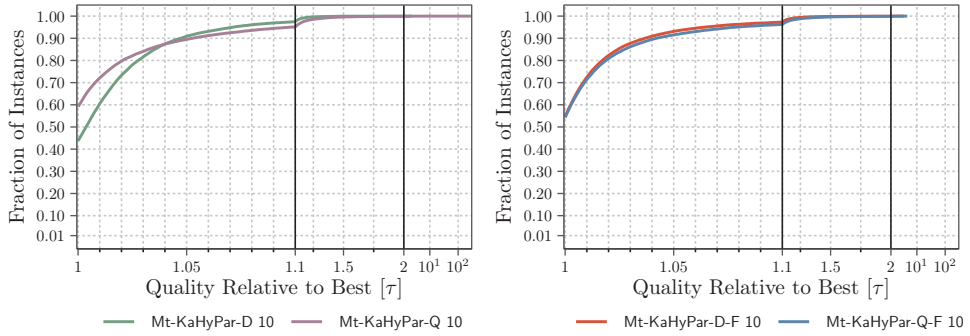



**Figure 6.11:** Effectiveness tests comparing Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F)) on set $M_{HG}$.

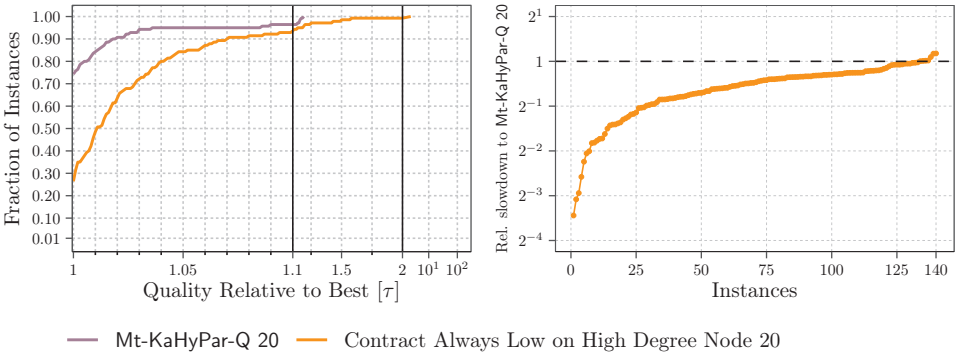



**Figure 6.12:** Performance profiles and running times comparing Mt-KaHyPar-Q to an optimization that always chooses the node with larger degree as the representative of a contraction in the coarsening phase.

technique is not essential to achieve high solution quality and may lead to simpler systems in the future.

Nevertheless, we still want to point out some future challenges that we could not entirely address within the scope of this work. One way to improve the effectiveness of the $n$-level partitioner is to implement faster (un)contraction operations since they account for a large fraction of total partitioning time. The coarsening algorithm iterates over the nodes in parallel, and whenever we visit a node $u$, we search for a contraction partner $v$ and subsequently contract $v$ onto $u$. The complexity of the contraction depends on the number of incident nets of the contraction partner $v$. Thus, a simple optimization is to contract $u$ onto $v$ if $d(u) < d(v)$, and vice versa otherwise.

Figure 6.12 compares Mt-KaHyPar-Q to a version implementing the optimization

mentioned above on our parameter tuning benchmark set $L_P$. The optimized implementation is 60% faster than Mt-KaHyPar-Q, but we also see that the solution quality deteriorates significantly. From a theoretical perspective, the resulting hypergraph should be the same regardless of whether we contract $v$ onto $u$ or $u$ onto $v$. This behavior was also independently discovered by the authors of KaHyPar (not published). Understanding this observation may lead to a faster implementation or an approach that achieves even better solution quality.

Furthermore, we do not use flow-based refinement on each level of the $n$-level hierarchy. It might be interesting to implement a localized version of the flow-based refinement algorithm that only solves small flow problems around the uncontracted nodes of each batch.

Chapter 7

# 7 From Hypergraphs to Graphs

A hypergraph partitioner can also be used to partition graphs. We are aware of two publications [AB12; Sch20] comparing hypergraph (HGP) and graph partitioning (GP) algorithms. Both conclude that both types of systems perform equally well on graph instances. However, Auer and Bisseling [AB12] report that Mondriaan (HGP) is an order of magnitude slower than Metis and Scotch (GP). This is not surprising since HGP and GP systems often use similar techniques, but HGP is considered "inherently more complicated" [Kay+12] and thus more complex "in terms of implementation and running time" [Bul+16].

These claims seem to be widely accepted, but what makes HGP so much more complicated than GP? The description of partitioning algorithms is often generic. For example, label propagation refinement iterates over the nodes in some order, and whenever we visit a node, we move it to the block maximizing its move gain. The high-level structure of the algorithm reveals no difference between HGP and GP but becomes noticeable when we take a closer look at how the gain function is evaluated.

For the cut-net metric of a given bipartition, a graph partitioner computes the move gain of a node $u$ by calculating the weight of the edges connecting $u$ to the target block minus the weight of the edges connecting it to its current block. To decide whether or not a hyperedge $e \in \mathrm{I}(u)$ can be removed from the cut, we have to know if $u$ is the last remaining pin of $e$ in block $\Pi[u]$. Thus, we either have to scan the entire pin-list of $e$ or store and maintain the pin count values explicitly. This makes the gain computation for hypergraphs much more complex than for graphs.

Furthermore, many algorithms iterate over the neighbors $\Gamma(u)$ of a node $u$, e.g., to aggregate ratings in the coarsening phase. For hypergraphs, this requires scanning the pin-lists of all incident nets of $u$. The incident nets and pin-lists are often stored in two separate adjacency arrays [CA99; Sch+16a], which introduces an additional indirection and random access to enumerate the neighbors of a node. Graph data structures maintain neighbors in a contiguous memory region, making access to them more cache-friendly (since only one adjacency array is required).

We have seen that the generic description of partitioning techniques often do not reveal any differences between GP and HGP. However, the concrete implementation of these algorithms mainly differ in the design of the partition (gain computation) and (hyper)graph data structure (representation of neighbors), which is "inherently more complicated" [Kay+12] for hypergraphs. In this chapter, we want to transform Mt-KaHyPar into a graph partitioner by implementing simplified data structures that take advantage of graph properties. From a software engineering perspective, we design

a graph data structure implementing the interface of our hypergraph data structure such that it can be used as a drop-in replacement in all our partitioning algorithms. From a research perspective, we study the performance overhead of a hypergraph data structure for graph partitioning.

**Outline.** We start this chapter with the partition data structure in Section 7.1, which covers a simplified gain table and a different algorithm to compute attributed gain values for graphs. Section 7.2 introduces the graph data structure for multilevel partitioning and discusses several peculiarities of using the new graph and partition data structure in our multilevel algorithm. In Section 7.3, we present a dynamic graph data structure for $n$-level partitioning. Section 7.4 then concludes this chapter by comparing the running times of the different components of our multilevel and $n$-level partitioning algorithm with and without the new partition and graph data structure.

**Contributors.** This chapter is based on unpublished work written exclusively for this dissertation. The idea of adapting the partition and hypergraph data structure for graph partitioning came from the author of this dissertation. The implementation was done by Nikolai Maas, who worked as a student research assistant in our group at the time.

## 7.1 Partition Data Structure

For hypergraphs, our partition data structure stores the partition assignments $\Pi$, the block weights $c(V_i)$, the pin count values $\Phi(e, V_i)$, and connectivity sets $\Lambda(e)$ for each net $e \in E$ and block $V_i \in \Pi$. Since a graph edge connects only two nodes, we can remove the pin count values and connectivity sets as we can calculate them on-the-fly. The move node operation is shown in Algorithm 7.1. It uses a simplified gain table and a different approach to compute attributed gains (compare with Algorithm 4.1 in Section 4.1.1).

### 7.1.1 The Gain Table

For graphs, the connectivity metric reverts to the cut metric since the connectivity $\lambda(e)$ of a graph edge $e$ is either one (internal edge) or two (cut edge). If we move a node $u$ to another block $V_j$, we remove all incident edges $\{u, v\} \in \mathrm{I}(u)$ with $\Pi[v] = V_j$ from the cut. Conversely, edges $\{u, v\} \in \mathrm{I}(u)$ with $\Pi[v] = \Pi[u]$ become cut edges. Hence, we can calculate the gain of moving a node $u$ to another block $V_j$ as

$$g_u(V_j) := \omega(u, V_j) - \omega(u, \Pi[u]).$$

Here, $\omega(u, V_i)$ denotes the weight of all edges connecting $u$ with block $V_i$. Thus, the gain table for graphs stores and maintains the $\omega(u, V_i)$ values for each node $u \in V$ and block $V_i \in \Pi$. After moving a node $u$, we update the gain table by adding $\omega(u, v)$ to $\omega(v, V_j)$ and $-\omega(u, v)$ to $\omega(v, V_i)$ for each $v \in \Gamma(u)$ using atomic `fetch-and-add` instructions (see Line 10 and 11 in Algorithm 7.1). Each node move updates exactly

**Algorithm 7.1:** Moving a node $u$ from block $V_i$ to $V_j$

**Input:** Node $u \in V_i$ and a target block $V_j$
**Output:** If $u$ was moved to $V_j$, then the function returns an attributed gain value $\Delta_{\lambda-1}$. Otherwise, it returns $\perp$.

```
1  B ← [⊥, ..., ⊥]                                          // Array of size m
2  c_j ← fetch-and-add(c(V_j), c(u))
3  if c_j + c(u) ≤ L_max then                                // Check if partition is still balanced
4      fetch-and-sub(c(V_i), c(u)), Δ_{λ−1} ← 0
5      for e = {u, v} ∈ I(u) do
           // Compute Attributed Gain
6          V_l ← Π[v]
7          if not compare-and-swap(B[e], ⊥, V_j) then V_l ← B[e]
8          if V_j = V_l then Δ_{λ−1} ← Δ_{λ−1} + ω(u, v)     // e becomes an internal edge
9          if V_i = V_l then Δ_{λ−1} ← Δ_{λ−1} − ω(u, v)     // e becomes a cut edge
           // Update Gain Table
10         fetch-and-add(ω(v, V_i), −ω(u, v))
11         fetch-and-add(ω(v, V_j), ω(u, v))
12     Π[u] ← V_j
13     return Δ_{λ−1}
14 else
15     fetch-and-sub(c(V_j), c(u))                           // revert block weight update of V_j
16     return ⊥
```

$2|\Gamma(u)|$ gain table entries. Hence, the computational complexity of the gain table updates when each node is moved at most once is $\sum_{u \in V} 2|\Gamma(u)| = \mathcal{O}(m)$.

### 7.1.2 Attributed Gains

We use attributed gains to track the overall improvement and double-check the gain of a node move. For hypergraphs, we compute this value based on synchronized writes to the pin count values. For a node move from block $V_i$ to $V_j$, we attribute a connectivity reduction by $\omega(e)$ to the move that reduces $\Phi(e, V_i)$ to zero and an increase by $\omega(e)$ for increasing $\Phi(e, V_j)$ to one. Since we removed the pin count values from the partition data structure, we need another synchronization mechanism to calculate attributed gains for graphs.

If we move a node $u$, we need to know whether or not it changes the state of an edge $\{u, v\} \in \mathrm{I}(u)$ (making it an internal or cut edge), and subsequently attribute a reduction or an increase by $\omega(u, v)$. In the parallel setting, another thread may move an adjacent node $v \in \Gamma(u)$ at the same time, making it difficult to decide which move removes or adds an edge to the cut. The following algorithm assumes that each node is moved at most once, as is in our parallel refinement algorithms. The idea is that we

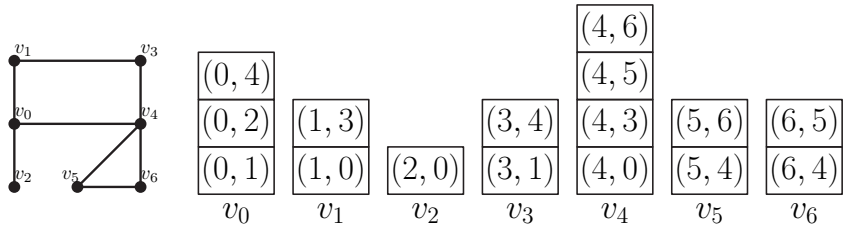



**Figure 7.1:** The graph data structure used in our multilevel graph partitioner.

use an array $B$ of size $m$ (initialized with $\perp$) to synchronize the node moves for each edge (instead of the pin count values). If a thread moves a node $u$ to block $V_j$, we set the $B[e]$ entries to $V_j$ for all edges $e \in \mathrm{I}(u)$ using `compare-and-swap` instructions. If the operation succeeds for an edge $e = \{u, v\}$, no other thread has moved node $v$ yet. In this case, we compare $V_j$ and $\Pi[v]$ to compute the attributed gain value for edge $e$. If $B[e] \neq \perp$, another thread has already moved or is simultaneously moving node $v$ to another block. In both cases, $B[e]$ is the new block of $v$, and we can compute the attributed gain value by comparing $V_j$ and $B[e]$.

The label propagation and FM algorithm are organized in rounds, where each node is moved at most once. Here, we reset array $B$ after each round. The flow-based refinement routine resets $B$ after applying a move sequence to the global partition.

## 7.2 Multilevel Graph Partitioning

Our multilevel algorithm contracts a node clustering on each level and runs parallel refinement algorithms to improve the solution quality of a partition in the uncoarsening phase. The coarsening algorithm builds on our (hyper)graph data structure, and the refinement algorithms use the partition data structure discussed in the previous section. This section presents our graph data structure with a simplified parallel contraction algorithm and discusses peculiarities of using our coarsening and refinement algorithms for graph partitioning.

### 7.2.1 The Graph Data Structure

We use *one* adjacency array to represent an undirected graph as illustrated in Figure 7.1. The adjacency list of each node $u$ stores the directed edges $(u, v) \in \mathrm{I}(u)$. We note that we could reduce the memory overhead of the data structure by storing only the neighbors $v \in \Gamma(u)$ instead of the ordered pair $(u, v) \in \mathrm{I}(u)$ for each node $u \in V$. However, our graph data structure implements the interface of our hypergraph data structure such that we do not have to adapt our partitioning algorithms. These algorithms often iterate over the incident edges of a node and then request the pin-list of an edge using its edge ID (position in the adjacency array). If we only store the neighbors in the adjacency lists, we would not be able to efficiently determine the two

nodes connected by an edge. Thus, our data structure requires twice as much memory as traditional graph data structures.

We now outline the parallel contraction operation used to contract a node clustering on each level. Recall that the coarsening algorithm stores the clustering in an array rep where $\mathsf{rep}[u] = v$ stores the representative of $u$'s cluster. For each representative $v$, we maintain the invariant that $\mathsf{rep}[v] = v$.

**Contraction.** We first remap cluster IDs to a consecutive range by computing a parallel prefix sum on an array of size $n$ that has a one at position $v$ if $v$ is a representative of a cluster and zero otherwise. Then, we accumulate the weights and degrees of nodes in each cluster using atomic `fetch-and-add` instructions. Afterwards, we copy the incident edges of each cluster to a consecutive range in a temporary adjacency array by computing a parallel prefix sum over the cluster degrees.

We then iterate over the adjacency lists of each cluster in parallel, sort it, and remove selfloops and identical edges except for one representative at which we aggregate their weight. Finally, we construct the adjacency array of the coarse graph by computing a parallel prefix sum over the remaining cluster degrees.

### 7.2.2 Peculiarities for Graph Partitioning

This section discusses implementation details and interesting properties of using the multilevel partitioning algorithm with the new partition and graph data structure.

**Community Detection.** In preliminary experiments, we found that community detection significantly improves partitioning for complex networks (highly-skewed node degree distributions) but degrades the solution quality for mesh-like networks (uniform node degree distributions). We therefore classify a graph based on the average $\mu$ and standard deviation $\sigma$ of its node degrees. If $\sigma \leq \frac{\mu}{2}$, we assume that the graph represents a mesh network and deactivate the community detection preprocessing step. We note that better classification algorithms exist, but the implemented scheme suffices for our purpose. Moreover, we omit the transformation into the bipartite graph representation and run the community detection algorithm directly on the graph representation.

**Coarsening.** The coarsening algorithm presented in Section 5.2 contracts a node clustering on each level. To do so, it iterates in random order over the nodes, and whenever we visit an unclustered node $u$, we add it to the cluster $C$ maximizing the heavy-edge rating function $r(u, C) = \sum_{\mathrm{I}(u) \cap \mathrm{I}(C)} \frac{\omega(e)}{|e|-1}$ [CA99; Kar+99; HS17a]. For graphs, the function reverts to $r(u, C) := \omega(u, C)$. Thus, a node $u$ joins the cluster $C$ to which it has the strongest connection. This corresponds to the rating function used in clustering algorithms based on label propagation [MSS14; ASS17; MSS17; Got+21e].

**Refinement.** We use all refinement algorithms without modifications in our multilevel graph partitioner. We note that flow-based refinement requires further engineering efforts to handle large graphs efficiently. The flow network construction algorithm

transforms a graph into a hypergraph, while the FlowCutter algorithm runs a parallel maximum flow algorithm implicitly on the Lawler expansion [Law73]. This causes much more overhead than necessary, as an optimized version would run the FlowCutter algorithm directly on the graph representation. Since the development of the graph partitioning algorithm was done at the same time as the write-up of this thesis, we were not able to address this performance issue. However, we will implement an optimized version in a future release of Mt-KaHyPar.

The FM algorithm applies moves to a thread-local partition. The moves are immediately performed on the global partition, once their individual gains suggest an improvement. The thread-local partitions store changes relative to the global partition. For example, we can calculate the local weight of a block $V_i$ by computing $c(V_i) + \Delta c(V_i)$, where $c(V_i)$ is the weight of block $V_i$ stored in the global partition and $\Delta c(V_i)$ is the weight of all nodes that locally moved to minus the weight of all nodes that moved out of block $V_i$. The gain table is maintained analogously.

For hypergraphs, applying a move sequence to the global partition can invalidate gain table entries stored in the thread-local partitions of other searches. A gain table update is only triggered when specific pin count values are observed, and applying moves to the global partition does not consider moves performed locally. For graphs, the gain table stores the $\omega(u, V_i)$ values for each node $u \in V$ and block $V_i \in \Pi$. Moving a node $u$ from block $V_i$ to $V_j$ requires updating the $\omega(v, V_i)$ and $\omega(v, V_j)$ values of all its neighbors $v \in \Gamma(u)$. The correctness of the gain table update does not depend on any moves performed locally by other threads, and therefore the gain table entries stored in the thread-local partitions are still correct.

## 7.3 $n$-Level Graph Partitioning

Instead of contracting a node clustering, the $n$-level algorithm contracts only a single node on each level. In the uncoarsening phase, we assemble independent contractions in a batch and uncontract them in parallel. We use the contraction forest to derive a parallel schedule of contractions and to construct the batches. The concept of the contraction forest is universally applicable to graph and hypergraph partitioning. Thus, we only have to redesign the dynamic hypergraph data structure, which we will describe in the following in more detail.

### 7.3.1 Contraction and Uncontraction Operation

We use an adjacency array to store an undirected graph. The graph is represented as a directed graph, meaning that each undirected edge $e = \{u, v\}$ induces two directed edges $(u, v)$ and $(v, u)$ in the data structure. For an edge $(u, v)$, we call $(v, u)$ its corresponding *backward edge*. The ID of an edge is its position in the adjacency array. For each edge $(u, v)$, we store the ID of its corresponding backward edge $(v, u)$ and its weight $\omega(u, v)$. The *unique ID* of an undirected edge $\{u, v\}$ is the smaller ID of

the two directed edges $(u, v)$ and $(v, u)$. Figure 7.2 illustrates the dynamic graph data structure with multiple (un)contraction operations performed on it.

**Contraction.** Contracting a node $v$ onto another node $u$ entails replacing $v$ with $u$ in each edge $(v, w) \in \mathrm{I}(v)$ (and also in the corresponding backward edge) and copying the modified adjacency list $\mathrm{I}(v)$ to $\mathrm{I}(u)$. For each node $u$, we store its adjacency list in an array $I_u$ (initialized with the adjacency list $I(u)$ of the input graph), and all nodes contracted onto $u$ in a doubly-linked list $L_u$. Once we have performed the contraction operation for each edge $e \in I(v)$, we lock $u$ and append $L_v$ to $L_u$. We can then obtain the adjacency list $I(u)$ of a node $u$ by iterating over all $I_w$ arrays for $w \in L_u$.

A contraction can transform an edge into a selfloop (see Figure 7.2 a) – b) and node $v_0$). A selfloop can be removed from the graph since it does not contribute to the cut metric. However, we do not remove them immediately due to reasons explained in the next section. Instead, we disable selfloops contained in the adjacency list of a node $v$ when we contract $v$ onto another node (see Figure 7.2 d)). We ignore disabled edges in further contractions. If we would consider them, we have to transfer selfloops to the representative of subsequent contractions (contracting $v$ onto $u$ requires to change $(v, v)$ to $(u, u)$). Consequently, the uncontraction operation must distinguish between uncontractions transferring selfloops and transforming them to regular edges again. If we disable selfloops when we contract a node, selfloops are not transferred, and thus the uncontraction operation can perform the reverse contraction operation.

The contraction operation is thread-safe since our coarsening algorithm ensures that all contractions onto $v$ are finished before we contract $v$ onto $u$. Therefore, $L_v$ is not modified by another thread. Furthermore, a contraction replaces $v$ with $u$ in all edges $e \in \mathrm{I}(v)$ and also in the corresponding backward edges. There may be two threads processing the same edge, but only one thread replaces $v$ at a time.

**Uncontraction.** Uncontracting $v$ from $u$ requires restoring $L_v$ from $L_u$. To do this, we additionally store the last node of $L_v$ at the time $v$ is contracted. Since multiple nodes can have the same representative in a batch, we lock $u$ before restoring $L_v$. Furthermore, we enable all selfloops of $v$ again. Afterwards, we iterate over each edge $e \in \mathrm{I}(v)$ and replace $u$ with $v$ (also in the corresponding backward edge).

Our graph data structure uses significantly less locking than our hypergraph data structure. We only use a lock when we append $L_v$ to $L_u$ (and restore $L_v$ from $L_u$), while our hypergraph data structure locks hyperedges to edit the pin-lists and to update the gain table.

**Uncontraction Gain Table Update.** Recall that the gain table stores the $\omega(u, V_i)$ values for each node $u \in V$ and block $V_i \in \Pi$. An uncontraction $(u, v)$ assigns $v$ to the block of $u$. Recomputing the gain table entries each time from scratch after each uncontraction incurs too much overhead with $\mathcal{O}(n)$ levels, which is why we update the $\omega(u, V_i)$ values based on the uncontraction of $v$. The gain table update operation distinguishes between two cases for each edge $e \in \mathrm{I}(v)$: (i) the uncontraction transforms a selfloop into regular edge (e.g., $(u, u)$ to $(u, v)$) or (ii) the uncontraction replaces $u$ with $v$ in a regular edge (e.g., $(u, w)$ to $(v, w)$). Note that the weight of a selfloop

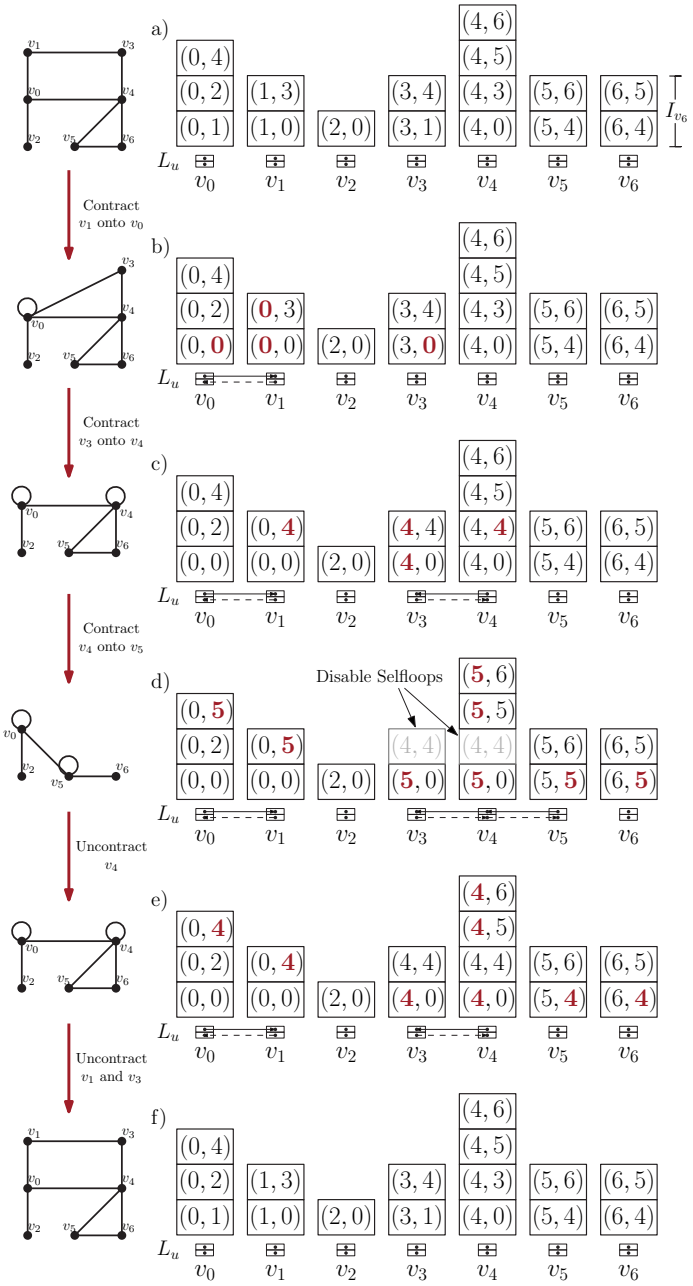



**Figure 7.2:** Contraction and uncontraction operations applied on the dynamic graph data structure. Entries modified by an (un)contraction operation are highlighted red.

$(u, u)$ is not considered in the gain table entry $\omega(u, \Pi[u])$. In the first case, we have to add $\omega(u, v)$ to $\omega(u, \Pi[u])$ and $\omega(v, \Pi[u])$. In the second case, we have to add $-\omega(v, w)$ to $\omega(u, \Pi[u])$ and $\omega(v, w)$ to $\omega(v, \Pi[u])$. In both cases, we use atomic `fetch-and-add` instructions.

In the parallel setting, it is not immediately apparent whether an uncontraction transforms a selfloop into a regular edge or replaces its representative: Consider two nodes $v$ and $w$, both adjacent before they were contracted onto the same node $u$ simultaneously. Contracting $v$ and $w$ onto $u$ induces a selfloop $(u, u)$. The batch uncontraction operation uncontracts both nodes in the same batch. If $v$ and $w$ revert their contraction for $(u, u)$ at the same time, we need a synchronization mechanism to decide which uncontraction transforms the selfloop into a regular edge.

We therefore use a bitset $X$ of size $m$ to synchronize the gain table updates. Before we replace $u$ with $v$ in an edge $e \in \mathrm{I}(v)$, we check if $e$ is a selfloop. If so, we raise a bit in $X$ at the position of $e$'s unique ID using an atomic `test-and-set` instruction. If the operation succeeds, uncontracting $v$ transforms the selfloop $(u, u)$ into a regular edge $(u, v)$. Otherwise, it replaces $u$.

**Differences to KaSPar.** KaSPar [OS10] is a graph partitioner based on the $n$-level scheme. The graph data structure also uses an adjacency array. A contraction of two nodes $u$ and $v$ marks both as deleted and appends a new node $w$ to the end of the adjacency array that contains $\Gamma(u) \cup \Gamma(v)$. Moreover, edges pointing to $u$ and $v$ are redirected to $w$. The approach suffers from a worst-case quadratic memory consumption and may destroy some locality contained in the input data since new nodes are added to the end of the adjacency array. However, a direct comparison between our and the approach used in KaSPar is missing in this work since it is not publicly available.

### 7.3.2 Remove and Restore Selfloops and Identical Edges

A contraction can transform edges into selfloops or make them identical to others. Selfloops can be removed from the graph since they cannot become cut edges. We can also remove identical edges except for one at which we aggregate their weight. To remove these edges, we divide each $I_u$ array into an *inactive* and *active* part. The inactive part contains selfloops and edges that became identical to others and are not considered anymore. The active part stores the current incident edges of $u$.

We can remove selfloops on-the-fly by swapping them to the inactive part of the corresponding $I_u$ array. This may lead to race conditions with other threads performing a contraction on the swapped edge (swapping an edge requires updating the IDs of the backward edges). However, removing these edges is only a performance optimization and does not affect the correctness of the algorithm. We therefore follow the same approach as in our $n$-level hypergraph partitioning algorithm and remove them after each coarsening pass.

**Removing Edges.** To remove selfloops and identical edges, we iterate over the nodes in parallel and process their adjacency lists sequentially. Since incident edges of

a node $u$ are not stored in a continuous memory region (obtained by iterating over the $I_w$ arrays for $w \in L_u$), we first copy them to a temporary vector and sort them. Identical edges are now next to each other in the sorted vector. We then add the weight of all identical edges to one representative (first occurrence of the edge in the sorted vector) and mark these edges in a globally shared bitset. Selfloops are also marked in the bitset.

In a second pass, we iterate over the $I_u$ arrays in parallel and swap marked edges to the inactive part. Note that this changes the edge IDs since they correspond to the positions in the adjacency array. Thus, we have to adapt the IDs of the backward edges. To do so, we maintain the permutation induced by the swap operations. In an additional pass over all edges, we apply the permutation to the IDs of the backward edges.

**Restoring Edges.** To restore selfloops and identical edges, we reconstruct the order of the adjacency array at the time before we removed them using the previously computed permutation. For each removed identical edge, we additionally store the edge ID of its representative. We then subtract the weight of each restored identical edge from its representative.

## 7.4 Experiments

The presented graph data structures replaces the internal hypergraph representation of our multilevel and $n$-level partitioning algorithms. The coarsening, initial partitioning, and refinement algorithms are reused without any modifications since the graph data structure implements the interface of our hypergraph data structure. We now analyze to which extent the running times of the different algorithmic components of Mt-KaHyPar-D and -Q benefit from the new graph layout. We run the experiments on our large graph benchmark set $L_G$ using all 64 cores of machine B and optimize the edge cut metric. We were not able to run the configurations with flow-based refinement due to the reasons explained in Section 7.2.2.

As we can see in Figure 7.3, replacing the internal hypergraph data structure with our new graph representation has no impact on the solution quality of Mt-KaHyPar-D and -Q (the $n$-level graph partitioner even produces slightly better partitions than our hypergraph partitioning code). We therefore analyze the running time improvements of the different algorithmic components in more detail. Figure 7.4 shows the speedups of each component of Mt-KaHyPar-D and -Q for each instance using a box plot.

**Mt-KaHyPar-D.** The coarsening algorithm has the largest running time improvement out of all components (geometric mean speedup is 2.48). The algorithm iterates in random order over all nodes and then over the pin-lists of all incident edges to aggregate ratings to adjacent clusters. This leads to a large number of random accesses when the incident edges and pin-lists are stored separately, as is in our hypergraph data structure. The graph data structure uses one adjacency array which stores neighbors of a node in a continuous memory region. The simplified representation also leads to

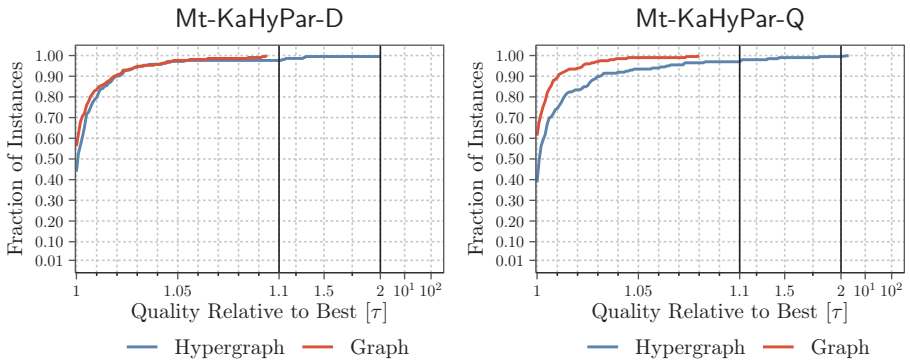



**Figure 7.3:** Performance profiles comparing the solution quality Mt-KaHyPar-D/-Q with and without our new graph data structure.

a faster contraction algorithm since only one adjacency array must be contracted, and identical edges can be removed more efficiently.

For community detection, we observe little to no speedups. This is expected since the algorithm uses an auxiliary graph data structure on which we then perform the Louvain algorithm. In a future version, we want to omit the transformation for graph partitioning.

The FM algorithm shows the least promising speedups (1.29). One of the most time-consuming parts of the algorithm is retrieving and updating entries from the gain table. The graph data structure implements a simplified gain table but still uses $k$ entries per node (instead of $k + 2$). Thus, retrieving and updating gain table values still suffer from a poor cache utlization (many main memory accesses), which may explain that the speedups are less pronounced here. The speedups of the label propagation algorithm are slightly better compared to the FM algorithm (1.53). The algorithm computes the gain values each time from scratch but still updates the gain table when moving a node. Thus, it profits more from the optimized memory layout when iterating over the neighbors of a node, but its speedup is still limited by the gain table updates.

The initial partitioning phase (1.8) has better speedups than both refinement algorithms but slightly worse speedups than the coarsening algorithm. This can be explained by the fact that initial partitioning uses coarsening and both refinement algorithms for multilevel recursive bipartitioning.

We also observe minor slowdowns on a few instances. A closer look reveals that these are mainly graph instances with a regular structure (uniform node degree distributions), e.g., road networks or finite element meshes. We believe that the input data of these instances already encode some locality and thus profits less from the optimized data layout. Furthermore, different random tie-breaking decisions can also significantly impact the execution time.

In total, the graph data structure improves the running time of Mt-KaHyPar-D by a factor of 1.75 on average (geometric mean running time 10.8s vs 18.94s).

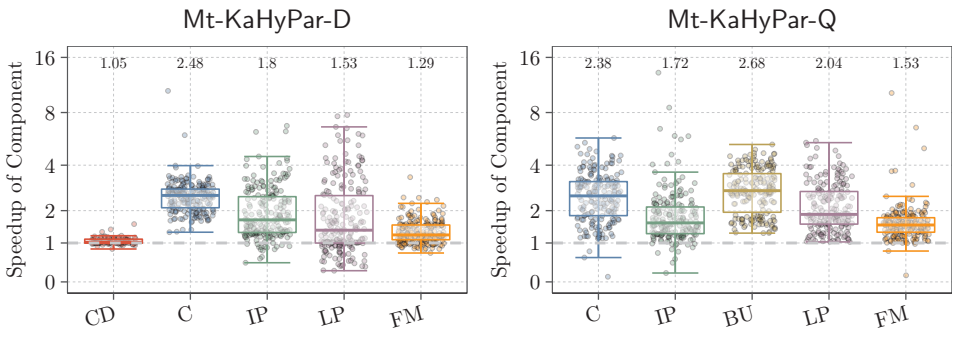



**Figure 7.4:** Running time improvements of the community detection (CD), coarsening (C), initial partitioning (IP), (localized) label propagation (LP), (localized) FM, and batch uncontraction (BU) algorithm for each instance when we use the graph instead of the hypergraph data structure.

**Mt-KaHyPar-Q.** Figure 7.4 (right) shows the speedups of the different algorithmic components of Mt-KaHyPar-Q. We omit the community detection algorithm and global refinement step since they are used in the same context as in Mt-KaHyPar-D.

The speedups of coarsening (2.38) and initial partitioning (1.72) are similar to what we observed for Mt-KaHyPar-D, while the speedups of the localized label propagation (2.04) and FM algorithm (1.53) are more pronounced in Mt-KaHyPar-Q. The localized searches only expand to a small number of nodes around the uncontracted nodes of each batch. Thus, we assume that most of the gain table values are still contained in shared caches after the batch uncontraction operation so that the overhead of retrieving and updating the gain table becomes less dominant.

The batch uncontraction operation (2.68) profits most from the new graph layout. It reverts the contraction operations and also benefits from the optimized data layout. Moreover, the uncontraction gain table update operation does not lock edges as in the hypergraph data structure and does not distinguish between a benefit and penalty term.

In total, the $n$-level graph data structure improves the running time of Mt-KaHyPar-Q by a factor of 1.91 on average (geometric mean running time 97.45s vs 186.32s).

Chapter 8

# 8 A Comparison of Partitioning Algorithms

In the previous chapters, we have seen that Mt-KaHyPar provides multiple configurations offering different tradeoffs in terms of speed and solution quality. We now compare them extensively to existing partitioning algorithms to see if Mt-KaHyPar can improve the state-of-the-art. This experimental evaluation compares 25 different partitioning algorithms (excluding Mt-KaHyPar), which we list and describe in Section 8.1. Section 8.2 then excludes systems that are outperformed by others. We then compare the remaining partitioning tools to Mt-KaHyPar in Section 8.3 and summarize the experimental results in Section 8.4.

**References.** This chapter summarizes the experimental results from our conference publications [Got+21a; GHS22a; Got+22a] and technical reports [Got+21c; GHS22c] and extends them with an extensive comparison of graph partitioning algorithms. We rerun all configurations of Mt-KaHyPar for this evaluation. The results of all evaluated hypergraph partitioners were taken from our latest publication [GHS22a]. We made all experimental results publicly available from `https://algo2.iti.kit.edu/heuer/dissertation/`.

## 8.1 Included Partitioning Algorithms

We did an extensive research on existing partitioning tools and integrate most of them into this evaluation. This is, to the best of our knowledge, the largest comparison of partitioning algorithms in the literature. We start this section by explaining which types of systems are excluded from this evaluation, and then describe the included partitioning algorithms listed in Table 8.1.

**Excluded Partitioning Algorithms.** In this experimental evaluation, we partition (hyper)graphs in up to 128 blocks. This already excludes algorithms restricted to bipartitioning (e.g., MLPart [CKM00a; CRX03] and ReBaHFC [GHW19]).

We further exclude systems from which we know that they perform considerably worse than the best sequential codes. This mainly excludes flat partitioning algorithms (e.g., Xtra-Pulp [Slo+17; Slo+20], JA-BE-JA [Rah+13], Spinner [Mar+17], HYPE [May+18], SHP [Kab+17]). It is known that these algorithms are inferior to multilevel methods [HB97]. For example, HYPE [May+18] (uses greedy hypergraph growing) computes partitions that are worse than those of the best sequential codes by more than an order of magnitude for connectivity optimization on average [Sch20].

**Table 8.1:** Listing of partitioning algorithms included in the experimental evaluation. For partitioners publicly available on GitHub, we show the first seven characters of the corresponding commit hash indicating the version used for the experiments.

| Partitioner | Version/Hash | Release Date |
|---|---|---|
| Sequential Graph Partitioner | | |
| Metis | 5.1.0 | Mar 30, 2013 |
| KaFFPa | `f239f7a` | Jan 13, 2022 |
| Scotch | 6.1.3 | Jan 1, 2022 |
| Parallel Graph Partitioner | | |
| KaMinPar | `29101f6` | Dec 15, 2021 |
| Mt-Metis | 0.6.0 | Oct 30, 2016 |
| ParMetis | 4.0.3 | Mar 30, 2013 |
| Mt-KaHIP | `30de737` | Mar 22, 2021 |
| ParHIP | `f239f7a` | Jan 13, 2022 |
| Sequential Hypergraph Partitioner | | |
| PaToH | 3.3 | July 15, 2020 |
| Mondriaan | 4.2.1 | Aug, 2019 |
| hMetis | 2.0pre1 | May 25, 2007 |
| KaHyPar | `876b776` | Aug 18, 2020 |
| Parallel Hypergraph Partitioner | | |
| Zoltan | 3.83 | Jan, 2016 |
| BiPart | `49a59a6` | Feb 22, 2021 |

We also exclude partitioners from which we do not expect that they run in a reasonable time frame on our benchmark sets. For example, we include the strongsocial configuration of KaFFPa [SS11] (KaFFPa-StrongS), which took roughly one month to complete on set $M_G$ using a cluster with 50 machines of type A. KaFFPa provides even stronger configurations, e.g., an evolutionary algorithm [SS12] and an approach based on integer linear programming [HNS20]. We expect that these algorithms are slower than KaFFPa-StrongS and thus exclude them. For similar reasons, we do not consider diffusion-based partitioning algorithms [MMS08; MMS09; Mey12], evolutionary techniques [SS12; ASS18], approaches based on tabu search [AV93; BH11b], integer linear programming [HNS20], and spectral methods [BS93; ZCS97].

Unfortunately, we are not able to include the publicly available version of Parkway [TK04a] (distributed hypergraph partitioner), PT-Scotch [CP08] (distributed graph partitioner), and Chaco [HL95] (sequential graph partitioner). These partitioners mostly crash with a segmentation fault on our benchmark instances.

**Included Hypergraph Partitioners.** We include the following sequential hypergraph partitioners: the default (PaToH-D) and quality preset (PaToH-Q) of PaToH

3.3 [CA99], the recursive bipartitioning (hMetis-R) and direct $k$-way version (hMetis-K) of hMetis 2.0 [Kar+99; KK00], Mondriaan 4.2.1 [VB05], and the recursive bipartitioning ($r$KaHyPar) and direct $k$-way version ($k$KaHyPar, uses similar algorithmic components than Mt-KaHyPar-Q-F) of KaHyPar [Sch+16a; Akh+17a; HS17a; HSS19a; Got+20]. We additionally include a version of KaHyPar that does not use flow-based refinement (KaHyPar-CA, uses similar algorithmic components than Mt-KaHyPar-Q). On large instances, we compare Mt-KaHyPar to the distributed-memory partitioner Zoltan 3.83 [Dev+06] and the deterministic shared-memory partitioner BiPart [Mal+21].

**Included Graph Partitioners.** We include the following sequential graph partitioners: the recursive bipartitioning (Metis-R) and direct $k$-way version (Metis-K) of Metis 5.1.0 [KK98a; KK98c], Scotch 6.1.3 [PR96], and the fast (KaFFPa-Fast), eco (KaFFPa-Eco), and strong configuration (KaFFPa-Strong) of KaFFPa [SS12; Sch13]. We additionally include the *social* configurations (KaFFPa-FastS, KaFFPa-EcoS, and KaFFPa-StrongS) of KaFFPa which use a clustering-based instead of a matching-based coarsening algorithm.

On large instances, we compare Mt-KaHyPar to the distributed-memory partitioners ParMetis 4.0.3 [KK96] and the fast and eco configuration of ParHIP [MSS17] as well as the shared-memory partitioners Mt-Metis 0.6.0 [LK13; LaS+15; LK16], Mt-KaHIP [ASS17; Akh19] and KaMinPar [Got+21e].

**Algorithm Configurations.** Almost all included partitioning algorithms have a considerable number of configuration options that influence their behavior. We therefore refrain from tuning them and use the default settings mentioned in the corresponding publications and user manuals. However, we manually adjusted some parameters to ensure a fair comparison between all algorithms.

Partitioners based on recursive bipartitioning must restrict the input imbalance ratio $\varepsilon$ to facilitate finding a balanced $k$-way partition (see Section 5.3 for a detailed explanation). KaHyPar [Sch+16a] (and also our initial partitioning algorithm) adjusts the imbalance ratio for each bipartition individually based on Equation 5.1. For partitioners based on recursive bipartitioning, where we observe that most of the solutions are imbalanced when $\varepsilon$ is used, we adjust $\varepsilon$ to

$$\varepsilon' := (1+\varepsilon)^{\frac{1}{\lceil \log_2(k) \rceil}}$$

which we obtain by applying Equation 5.1 to the first bipartitioning step. This applies to Metis-R, hMetis-R, and BiPart.

Furthermore, hMetis does not directly optimize the connectivity metric. Instead, it optimizes the sum-of-external-degree (SOED) metric $f_s(\Pi) := \sum_{e \in E} \lambda(e) \cdot \omega(e) = f_{\lambda-1}(\Pi) + f_c(\Pi)$ (connectivity plus cut-net metric). We therefore configure both hMetis versions to optimize the SOED metric and compute the connectivity metric accordingly. We additionally configure Mt-Metis to use its hill-scanning refinement algorithm [LK16] that produces partitions with better edge cuts than a previous version using greedy refinement [LK13]. Moreover, Scotch and BiPart do not provide an command line parameter to set an initial seed value. Thus, we do not perform multiple repetitions when running them.

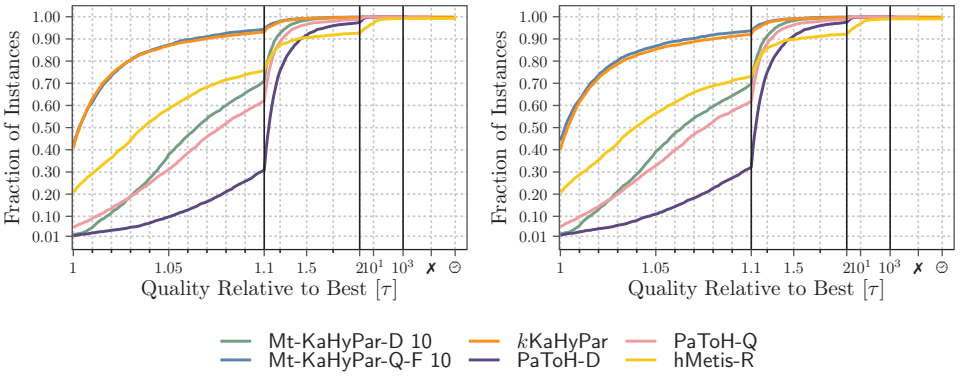



**Figure 8.1:** Performance profile comparing Mt-KaHyPar to different sequential partitioning algorithms on set $M_{HG}$ (left, 488 hypergraphs) and on the same benchmark set where we excluded the instances of our parameter tuning benchmark set $M_P$ (right, 388 hypergraphs).

We made the repository with which we conducted the following experiments publicly available.[1] It contains build scripts and the partitioning calls for each included algorithm.

**A Note on our Benchmark Sets.** In Section 5.6, we evaluated different parameter configurations of our algorithm on our parameter tuning benchmark set $M_P$, which is a subset of set $M_{HG}$. In this experimental evaluation, we compare Mt-KaHyPar to different sequential partitioning algorithms on set $M_{HG}$. Thus, one might argue that Mt-KaHyPar might have an unfair advantage since it has been optimized for a subset of set $M_{HG}$. A possible solution could be to exclude all instances of set $M_P$ from set $M_{HG}$ for the following comparison. However, as it can be seen in Figure 8.1, it makes no difference whether or not we include or exclude these instances. Both performance profiles comparing different sequential partitioning algorithms to Mt-KaHyPar look exactly the same. Thus, we decided to include the hypergraphs of set $M_P$ to increase the evidence of the following experimental results.

## 8.2 Identifying Competitors

In this experimental evaluation, we study the performance of 25 different partitioning algorithms (14 distinct partitioners). In the following, we reduce this set by excluding systems that are outperformed by others. For the experiments, we used the partitioning setup shown in Table 2.3 in Section 2.4.2. Recall that we optimize connectivity metric for hypergraph partitioning, and the cut-net metric for graph partitioning. Moreover,

[1] `https://github.com/kittobi1992/hypergraph_partitioner`

we add a suffix to the name of parallel partitioners to indicate the number of threads used, e.g. Mt-KaHyPar 64 for 64 threads. We omit the suffix for sequential partitioners.

**A Note on Improvements.** A partitioner $X$ *outperforms* another partitioner $Y$ if (i) $X$ produces partitions with significantly better solution quality and is at least as fast as $Y$, *or* (ii) $X$ produces partitions with comparable solution quality than $Y$ and is significantly faster.

In the following evaluation, no partitioner consistently outperforms another partitioner on all tested instances. Thus, we conclude whether or not an algorithm is better than another based on metrics extracted from the performance profiles and relative running time plots. More precisely, we look at the percentage of instances where an algorithm produces better partitions and is faster than another. Furthermore, we use the median improvement for solution quality and the average slowdown for running time to quantify improvements. However, several other metrics could be considered. For example, we might be interested that the quality of the partitions produced by an algorithm are not too far away from the best found solutions. To do so, we can look at the intersection of $\tau = 1.1$ with the line of an algorithm in the performance profiles. This denotes the fraction of instances where an algorithm performs worse than the best by at most 10%. However, to simplify the evaluation, we restrict ourselves to the metrics mentioned above and leave it to the reader to extract further performance indicators from the plots.

The question remains of what we consider a significant improvement. From an application perspective, an improvement in solution quality or running time should have a considerable impact on its application. For VLSI design, even small improvements in solution quality are considered critical [HB95], whereas speed is more important than quality for sparse matrix-vector multiplications [CA99]. Hence, evaluating improvements depends on the application. However, setting up application-specific benchmarks requires a strong understanding of the underlying domain and is not within the scope of this work.

We therefore took another approach and looked at improvements that were considered significant in the past. On benchmark set $\mathrm{M_{HG}}$, the quality preset of PaToH (PaToH-Q) improves the default preset (PaToH-D) by 5.3% in the median and is a factor of 4.99 slower on average. The median improvement of hMetis-R compared to PaToH-Q is 2.6%, while it is a factor of 15.9 slower on average. We can see that improvements by a few percentages can be considered significant, while the running times of partitioning algorithms can differ by a multiple.

**Sequential Hypergraph Partitioners.** Figure 8.2 and 8.3 compare the solution quality and running times of different sequential hypergraph partitioners on set $\mathrm{M_{HG}}$.

In an individual comparison, PaToH-D and Mondriaan compute better partitions than the other on roughly 50% of the instances, but there are a few instances where Mondriaan performs significantly worse than PaToH-D. Moreover, Mondriaan (geometric mean running time 6.62s) is slower than PaToH-D (1.17s) on almost all instances.

Figure 8.3 (top) compares the performance of KaHyPar-CA to hMetis-R and hMetis-K. KaHyPar-CA (28.14s) computes better solutions than hMetis-R (93.2s) and hMetis-K

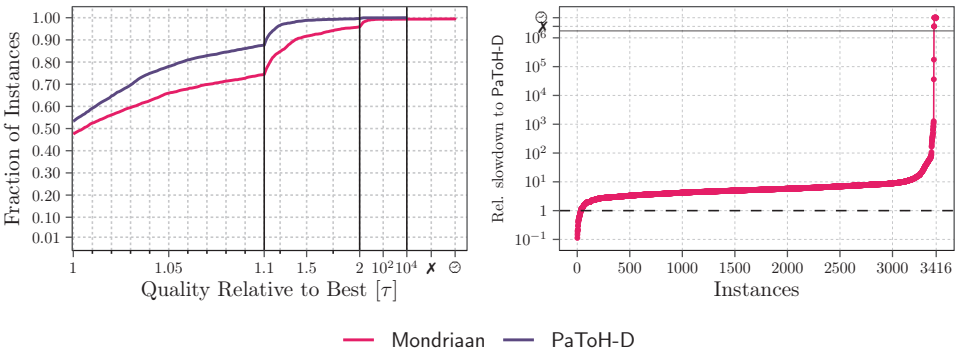



**Figure 8.2:** Performance profile and running times comparing PaToH-D and Mondriaan on set $M_{HG}$.

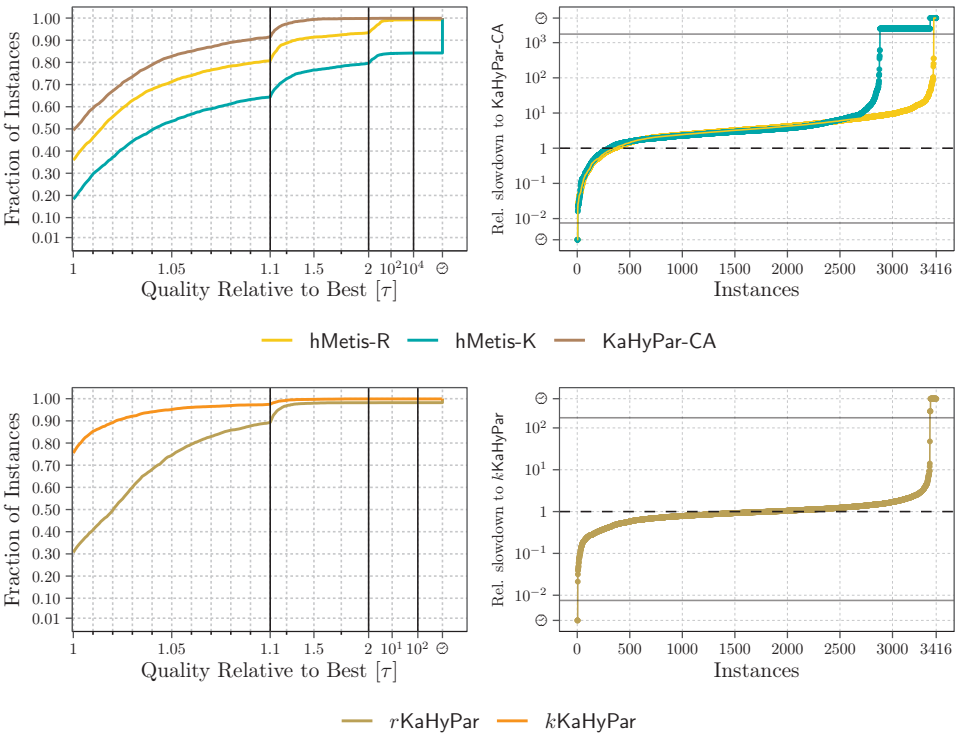



**Figure 8.3:** Performance profile and running times comparing different configurations of KaHyPar and hMetis on set $M_{HG}$.

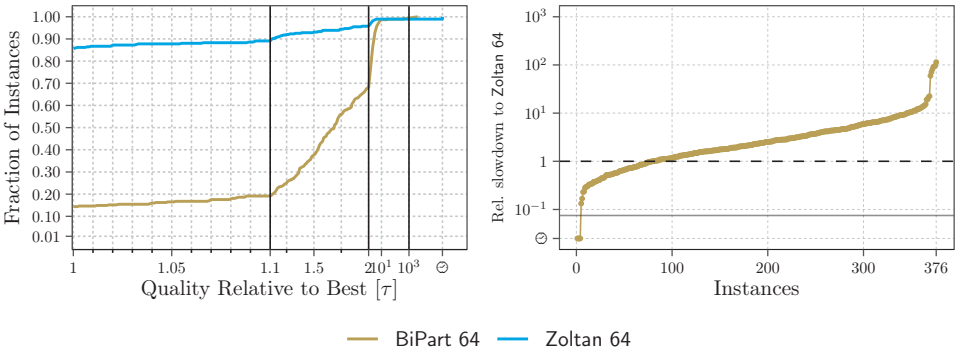



**Figure 8.4:** Performance profile and running times comparing Zoltan and BiPart on set $L_{HG}$.

(73.74s) on 56.2% and 68% of the instances (median improvement is 0.5% and 2.6%) and is faster by a factor of 3.31 and 2.62 on average. In Figure 8.3 (bottom), we compare the recursive bipartitioning and direct $k$-way version of the highest-quality configuration of KaHyPar. The partitions produced by $k$KaHyPar are better than those of $r$KaHyPar by 2.1% in the median, while the running times of both are comparable (48.97s vs 46.09s).

In summary, we exclude Mondriaan (outperformed by PaToH-D), hMetis-R, and hMetis-K (outperformed by KaHyPar-CA), and $r$KaHyPar (outperformed by $k$KaHyPar) from further experiments.

**Parallel Hypergraph Partitioners.** Figure 8.4 compares the performance of Zoltan and BiPart on set $L_{HG}$. Zoltan (12.82s) is faster and produces partitions with better solution quality than BiPart (29.18s) on 85.6% of the instances (median improvement is 69%). We therefore exclude BiPart in further experiments.

**Sequential Graph Partitioners.** Figure 8.5 shows that Metis-K (geometric mean running time 0.39s) outperforms Metis-R (0.55s), KaFFPa-Fast (1.69s), KaFFPa-FastS (1.88s), and Scotch (1.83s) on set $M_G$. In an individual comparison, Metis-K computes partitions with better edge cuts than Metis-R, KaFFPa-Fast, KaFFPa-FastS, and Scotch on 75.41%, 80.31%, 58.3%, and 59.55% of the instances, respectively (median improvement is 2.9%, 5.8%, 2.2%, and 2.5%, respectively). We note that the partitions produced by Metis-R are perfectly balanced on 46.3% of the instances, even though we set the imbalance ratio to $\varepsilon = 3\%$ as described in the user manual [Kar13]. Thus, we believe that Metis-R could compute better partitions than Metis-K if it would use the leeway provided by the balance constraint.

In Figure 8.6, we compare the *social* configurations of KaFFPa (KaFFPa-EcoS and KaFFPa-StrongS, clustering-based coarsening) to their non-social counterparts (KaFFPa-Eco and KaFFPa-Strong, matching-based coarsening). The running times of

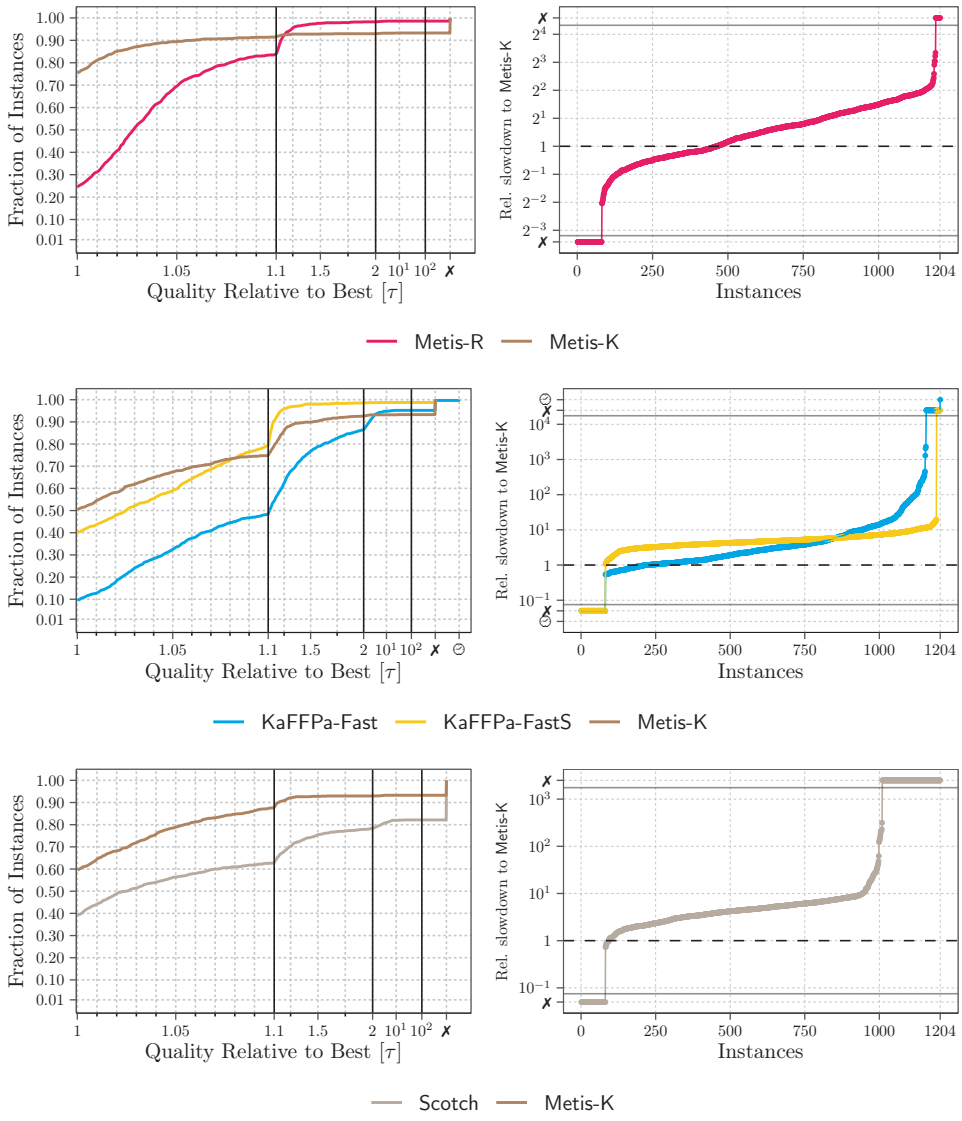



**Figure 8.5:** Performance profile and running times comparing Metis-K to Metis-R, KaFFPa and Scotch on set $M_G$.

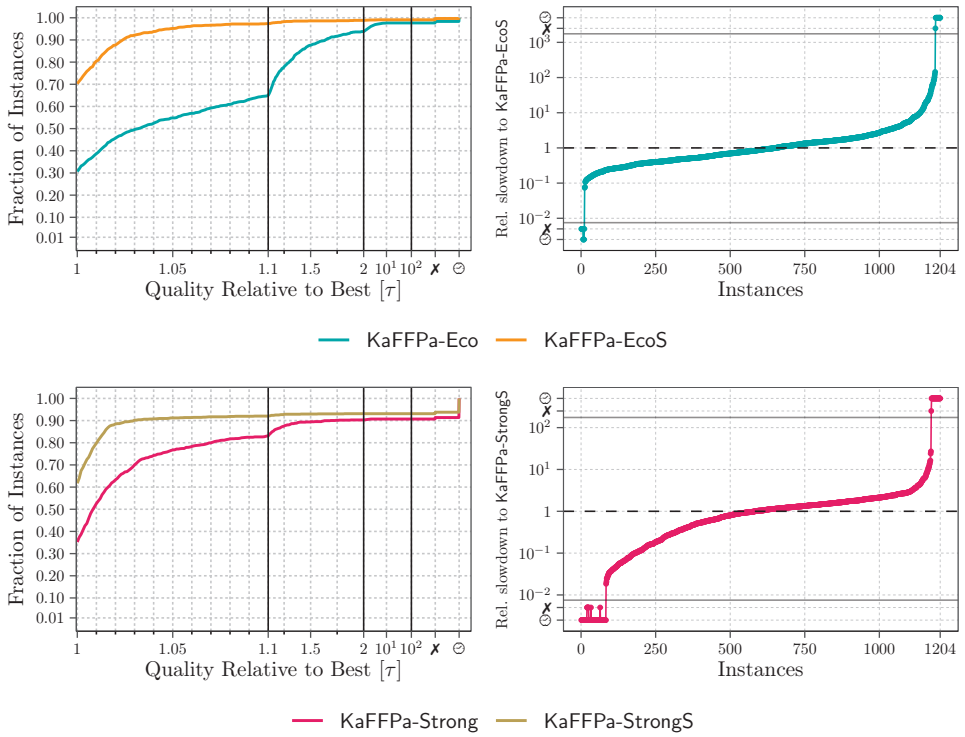



**Figure 8.6:** Performance profile and running times comparing the social configurations of KaFFPa to their non-social counterparts on set $M_G$.

KaFFPa-Eco (10.94s) and KaFFPa-EcoS (10.51s) are comparable, while KaFFPa-Strong (162.83s) is faster than KaFFPa-StrongS (201.99s). If we look at the performance profiles in Figure 8.6 (left), we see that the social configurations produce partitions with better edge cuts than their non-social counterparts.

We exclude Metis-R, KaFFPa-Fast, KaFFPa-FastS, Scotch (outperformed by Metis-K), and KaFFPa-Eco (outperformed by KaFFPa-EcoS) from further experiments.

**Parallel Graph Partitioners.** Figure 8.7 (top and middle) shows that KaMinPar (geometric mean running time 2.67s) outperforms ParHIP-Fast (21.85s), Mt-Metis (24.33s), and ParMetis (564.24s) on set $L_G$. On almost all instances ($\geq 99\%$), KaMinPar is faster than ParMetis and ParHIP-Fast. The edge cuts of the partitions produced by KaMinPar are better than those of ParHIP-Fast by 2.8% in the median. A comparison to ParMetis and Mt-Metis turns out to be more difficult since both compute imbalanced solutions on 14.15% and 32.54% of the tested instances. If we include them into the comparison, then KaMinPar still computes better partitions than ParMetis (median improvement is 4.4%) and comparable partitions to Mt-Metis (both compute partitions

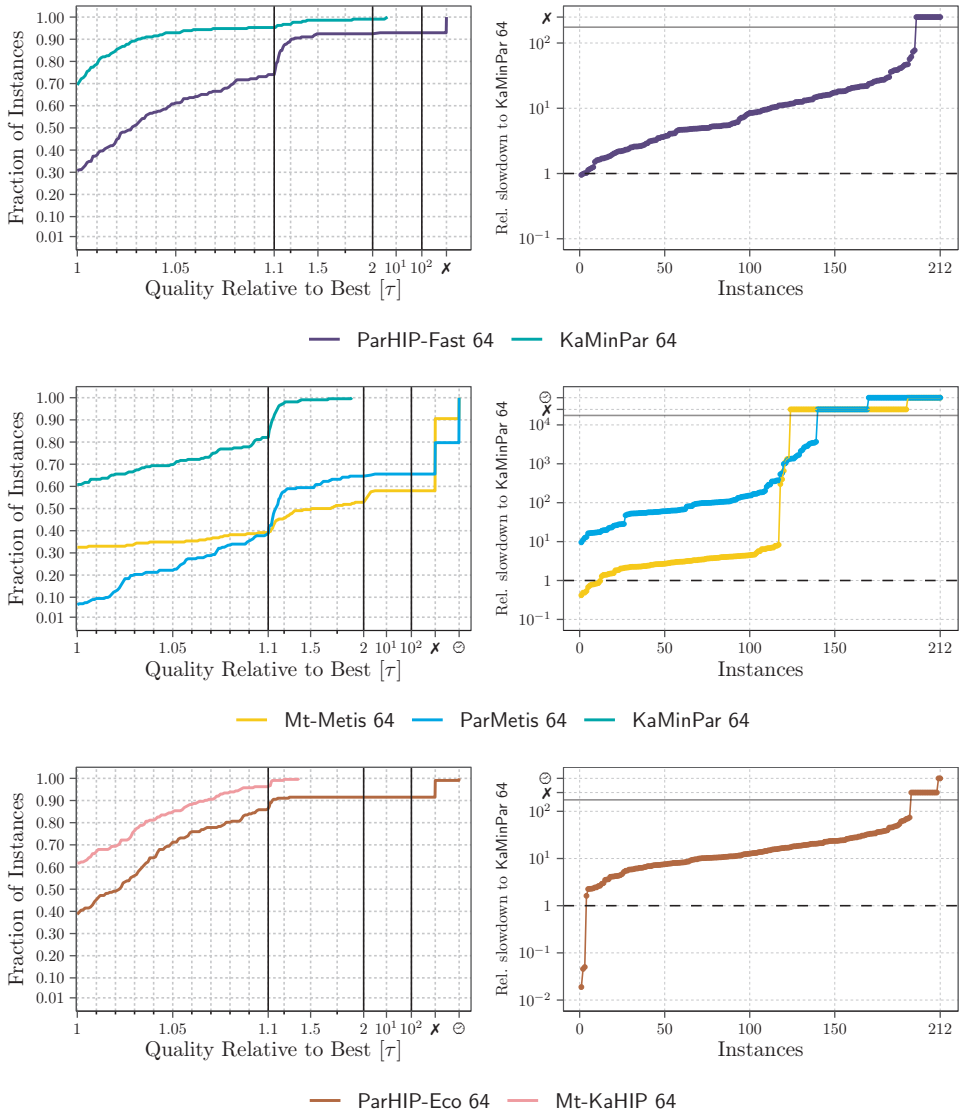



**Figure 8.7:** Performance profile and running times comparing different parallel graph partitioners on set $L_G$.

with a better edge cut than the other on 50% of the instances). Moreover, as illustrated in Figure 8.7 (bottom), Mt-KaHIP (13.69s) outperforms ParHIP-Eco (159.09s, median improvement is 2.2%).

We therefore exclude ParHIP-Fast, Mt-Metis, and ParMetis (outperformed by KaMinPar) as well as ParHIP-Eco (outperformed by Mt-KaHIP) in further experiments.

## 8.3 Comparison to Other Systems

We now compare Mt-KaHyPar to the partitioning algorithms identified as the main competitors in the previous section. Mt-KaHyPar provides a multilevel (Mt-KaHyPar-D) and $n$-level partitioning algorithm (Mt-KaHyPar-Q), as well as configurations extending them with flow-based refinement (Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F). The partitioning algorithms to which we compare Mt-KaHyPar in the following can be divided into systems aiming for speed (e.g., PaToH-D and Metis-K) or high solution quality (e.g., $k$KaHyPar and KaFFPa-StrongS). To simplify the evaluation, we compare the fast partitioners to Mt-KaHyPar-D (fastest configuration) and the high quality partitioners to Mt-KaHyPar-Q-F (highest-quality configuration). However, we recommend using Mt-KaHyPar-D-F in practice, as using multiple restarts produces comparable solutions to Mt-KaHyPar-Q-F in the same amount of time (see effectiveness tests in Section 6.4.4). For graph partitioning, we use the partition and graph data structure presented in Chapter 7. We run Mt-KaHyPar using *ten* threads of machine A when comparing it to sequential partitioning algorithms. Note that we do not use all available cores of machine A (2 sockets with 20 cores each) since we want to simulate the performance of Mt-KaHyPar that can be expected on commodity workstations.

**A Note on Effectiveness Tests.** In previous chapters, we used effectiveness tests to compare two algorithms by giving the faster algorithm more time to perform additional repetitions until its expected running time equals the running time of the slower algorithm. This turns out to be more difficult when comparing a sequential and parallel algorithm. To ensure a fair comparison, we can run the sequential algorithm independently in parallel using the same number of threads as the parallel algorithm and take the best solution out of these runs as the final partition. However, this requires substantially more memory, and we do not know in advance whether or not the main memory of the target machine suffices to conduct these experiments on each instance. We therefore do not perform effectiveness tests in this section.

### 8.3.1 Hypergraph Partitioning

**Medium-Sized Instances.** Figure 8.8 and 8.9 compares Mt-KaHyPar to the sequential partitioners PaToH and KaHyPar on set $\mathrm{M_{HG}}$. In an individual comparison, Mt-KaHyPar-D (geometric mean running time 0.88s) computes better partitions than PaToH-D (1.17s) and PaToH-Q (5.85s) on 82.9% and 58.34% of the instances (median improvement is 6.6% and 1.2%), while it achieves a speedup of 1.32 w.r.t. PaToH-D

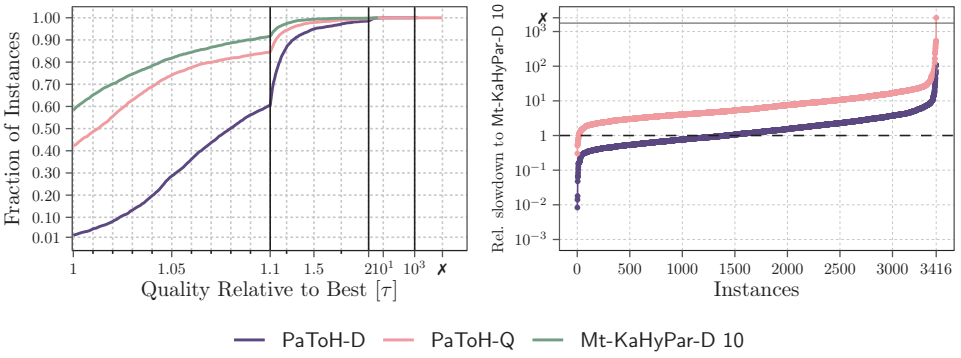



**Figure 8.8:** Performance profile and running times comparing Mt-KaHyPar-D and PaToH on set $M_{HG}$.

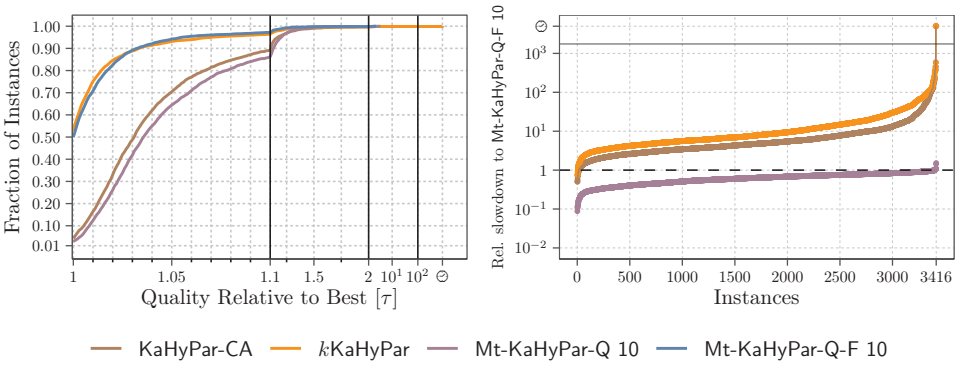



**Figure 8.9:** Performance profile and running times comparing Mt-KaHyPar-Q-F and KaHyPar on set $M_{HG}$.

and 6.6 w.r.t. PaToH-Q with ten threads on average. Thus, Mt-KaHyPar-D outperforms PaToH-D and PaToH-Q.

Figure 8.9 (left) shows that the solution quality of the partitions produced by Mt-KaHyPar-Q-F and *k*KaHyPar are on par. We also see that the differences in solution quality between Mt-KaHyPar-Q and KaHyPar-CA is not statistically significant (Wilcoxon signed-ranked test: $Z = -2.4073$ and $p = 0.01607$). Mt-KaHyPar-Q-F (5.08s) is faster than KaHyPar-CA (28.14s) and *k*KaHyPar (48.97s) on almost all instances with ten threads ($\geq 99\%$). This shows that we achieved the same solution quality as the currently highest-quality sequential partitioner *k*KaHyPar, while being almost an order of magnitude faster with only ten threads. Moreover, Mt-KaHyPar-Q-F is also slightly faster than PaToH-Q, while it computes better partitions than PaToH-Q on 87.7% of the instances (median improvement is 6.4%).

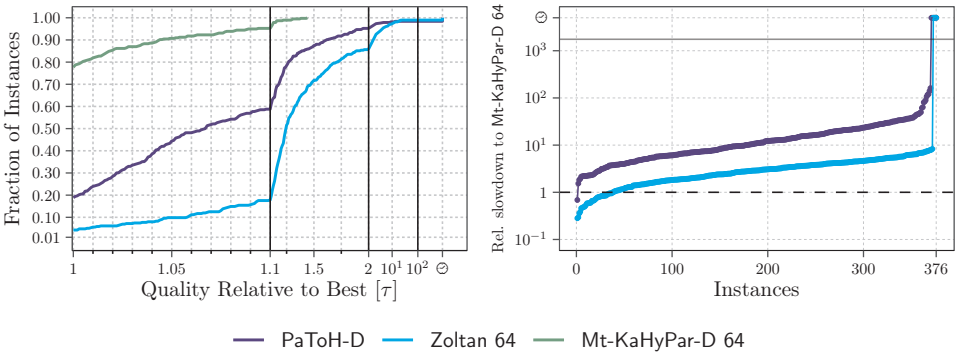



**Figure 8.10:** Performance profile and running times comparing Mt-KaHyPar-D to PaToH-D and Zoltan on set $L_{HG}$.

**Large Instances.** Figure 8.10 compares Mt-KaHyPar-D to distributed-memory partitioner Zoltan and the sequential partitioner PaToH-D on set $L_{HG}$. Note that PaToH-D is fast enough to conduct the experiments on set $L_{HG}$ in a reasonable time frame, while this is not the case for any of the other sequential partitioners.

In an individual comparison, Mt-KaHyPar-D (geometric mean running time 4.64s) computes better partitions than PaToH-D (51.2s) and Zoltan (12.63s) on 79.78% and 95.21% of the instances (median improvement is 6.6% and 23%), while it is also faster by a factor of 2.72 than Zoltan and it achieves a speedup of 11.03 w.r.t. PaToH-D with 64 threads on average. Our flow-based refinement configuration Mt-KaHyPar-D-F (30.44s) computes partitions better than those of Zoltan by 34% in the median, while it is only slower by a factor of 2.41 on average. This can be considered a major breakthrough as it enables partitioning of extremely large hypergraphs with high solution quality, which was previously only possible with sequential codes on medium-sized instances.

## 8.3.2 Graph Partitioning

**Medium-Sized Instances.** Figure 8.11 and 8.12 compares Mt-KaHyPar to the sequential partitioners Metis-K and different configurations of KaFFPa on set $M_G$. Mt-KaHyPar-D (geometric mean running time 0.55s) is slightly slower but produces considerably better edge cuts than Metis-K (0.39s, edge cuts are better by 5.9% in the median) when using ten threads. We therefore compare Metis-K to a weaker configuration by disabling the FM algorithm (referred to as Mt-KaHyPar-S). The Wilcoxon signed-ranked test reveals that there is no statistically significant difference between the edge cuts produced by Mt-KaHyPar-S (0.37s) and Metis-K ($Z = -1.5126$ and $p = 0.1304$), while the execution times of both are comparable when running Mt-KaHyPar-S with ten threads.

KaFFPa-StrongS computes slightly better edge cuts than Mt-KaHyPar-Q-F (median

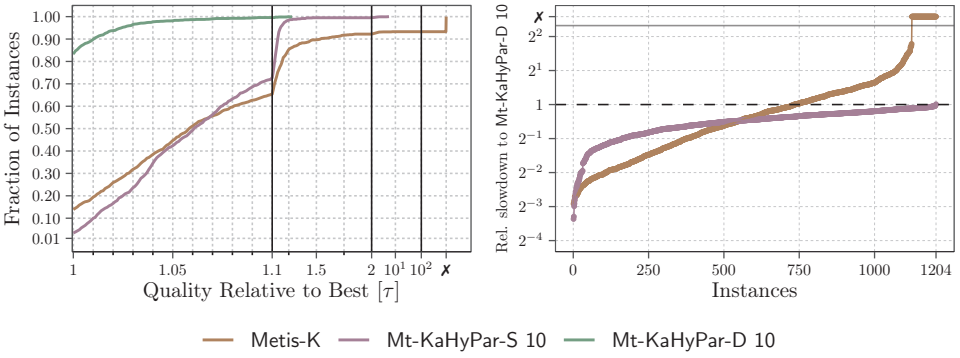



**Figure 8.11:** Performance profile and running times comparing Mt-KaHyPar-D and Metis-K on set $M_G$.

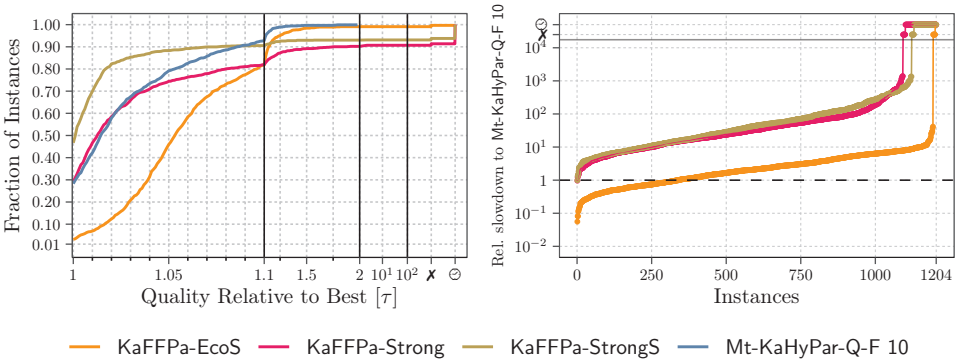



**Figure 8.12:** Performance profile and running times comparing Mt-KaHyPar-Q-F and KaFFPa on set $M_G$.

improvement is 1%). However, Mt-KaHyPar-Q-F (5.22s) achieves a speedup over KaFFPa-StrongS (201.99s) by a factor 38.66 with ten threads on average, making the quality improvement questionable in practice. The differences between the edge cuts produced by Mt-KaHyPar-Q-F and KaFFPa-Strong (162.83s) are not statistically significant ($Z = -2.3101$ and $p = 0.02088$). Moreover, Mt-KaHyPar-Q-F outperforms KaFFPa-EcoS (10.51s, edge cuts are better by 2.9% in the median).

This demonstrates that Mt-KaHyPar also outperforms most of the existing sequential graph partitioners. KaFFPa-StrongS still computes slightly better edge cuts than Mt-KaHyPar-Q-F, but is more than an order of magnitude slower.

**Large Instances.** Figure 8.13 compares Mt-KaHyPar-D to the shared-memory partitioners Mt-KaHIP (also implements a parallel version of the FM algorithm) and

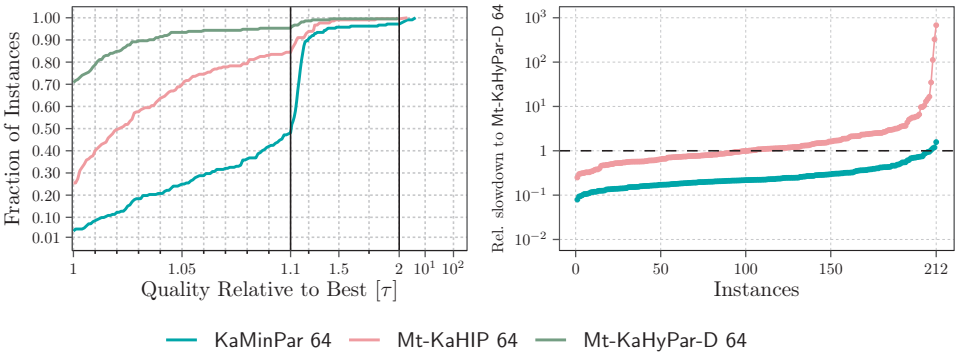



**Figure 8.13:** Performance profile and running times comparing Mt-KaHyPar-D to KaMinPar and Mt-KaHIP on set $L_G$.

KaMinPar on set $L_G$. We see that Mt-KaHyPar-D (geometric mean running time 10.8s) computes better edge cuts than Mt-KaHIP (13.69s) on 74.52% of the instances (median improvement is 2.1%), while it is also slightly faster. Thus, Mt-KaHyPar-D outperforms Mt-KaHIP.

The median speedup of KaMinPar (2.69s) over Mt-KaHyPar-D is 4.51 on average, but its edge cuts are worse than to those of Mt-KaHyPar-D by 9.9% in the median. Thus, KaMinPar is the method of choice when speed is more important than quality, and Mt-KaHyPar-D should be used if one aims for high solution quality.

**Comparison to KaSPar.** KaSPar [OS10] is the first (graph) partitioning algorithm based on the $n$-level scheme and can be considered as a seminal work for KaHyPar. The algorithm uses highly-localized FM searches and so-called trial trees: after reducing the size of the graph by a factor of $c$ during coarsening, the graph is copied, and multilevel partitioning is continued on both copies of the graph recursively. Unfortunately, KaSPar is not publicly available, but we were able to obtain experimental results for the strong configuration of KaSPar from the author. There also exists a fast configuration, which is compared to the strong configuration, a factor of 5 – 6 faster, but it computes partitions that are 2% – 3% worse on average [OS10].

In the following, we compare Mt-KaHyPar-Q-F (using ten threads) to KaSPar strong on the same benchmark set used in their publication with the same experimental setup ($k \in \{2, 4, 8, 16, 32, 64\}$, $\varepsilon = 3\%$, and ten repetitions per instance). The benchmark set contains 37 graphs composed of random geometric graphs, Delaunay triangulations, road networks, sparse matrices [DH11], social networks [LK14], and instances from the Walshaw benchmark archive [SWC04] (for detailed properties of the graphs, see Ref. [OS10]). However, we excluded three graphs as our version of these instances did not match the properties reported in their publication (`cnr2000`, `citationCiteseer`, and `nld`). The experiments run on one of our oldest machines to have at least some comparable running times (Intel Xeon E5-2670 v3 processor, 2 sockets with 12 cores

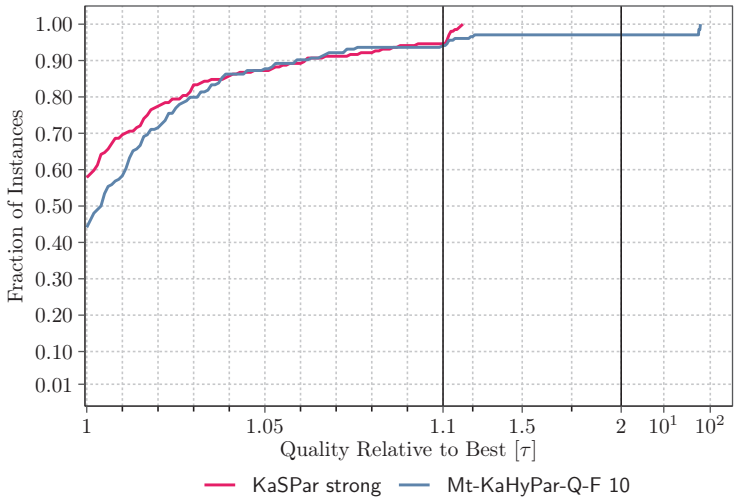



**Figure 8.14:** Performance profile comparing Mt-KaHyPar-Q-F and KaSPar strong on the benchmark set from Ref. [OS10].

each, 2.3 GHz, 128GB RAM).

As can be seen in Figure 8.14, the difference in solution quality between Mt-KaHyPar-Q-F and KaSPar strong is marginal. KaSPar strong computes on slightly more instances better partitions than Mt-KaHyPar-Q-F (56% vs 44%). Looking closer at the results reveals that Mt-KaHyPar-Q-F computes better partitions on social networks, while KaSPar strong performs better on mesh and road networks. Especially on the Belgium road network, we observed that Mt-KaHyPar-Q-F cannot find a small separator that naturally exists in such networks [Del+11] (for $k = 2$, KaSPar strong cut: 81 vs Mt-KaHyPar-Q-F cut: 4452). Since the benchmark set contains only three social networks, we assume that the results would look much better if we had run the experiments on set $M_G$. The geometric mean running times of Mt-KaHyPar-Q-F (using ten threads) and KaSPar strong are 1.81s and 24.1s. However, the running times are not comparable since the experiments run on different machines.

## 8.4 Summary

This experimental evaluation compared 17 sequential and 8 parallel partitioning algorithms to our new shared-memory (hyper)graph partitioner Mt-KaHyPar. We evaluated the partitioners on *four* different benchmarks sets consisting of 804 graphs and hypergraphs. The total running time of the experiment would have been 10.43 years when executed on a single machine. Thus, it is the most comprehensive study of partitioning algorithms using the largest benchmark set that can be found in the

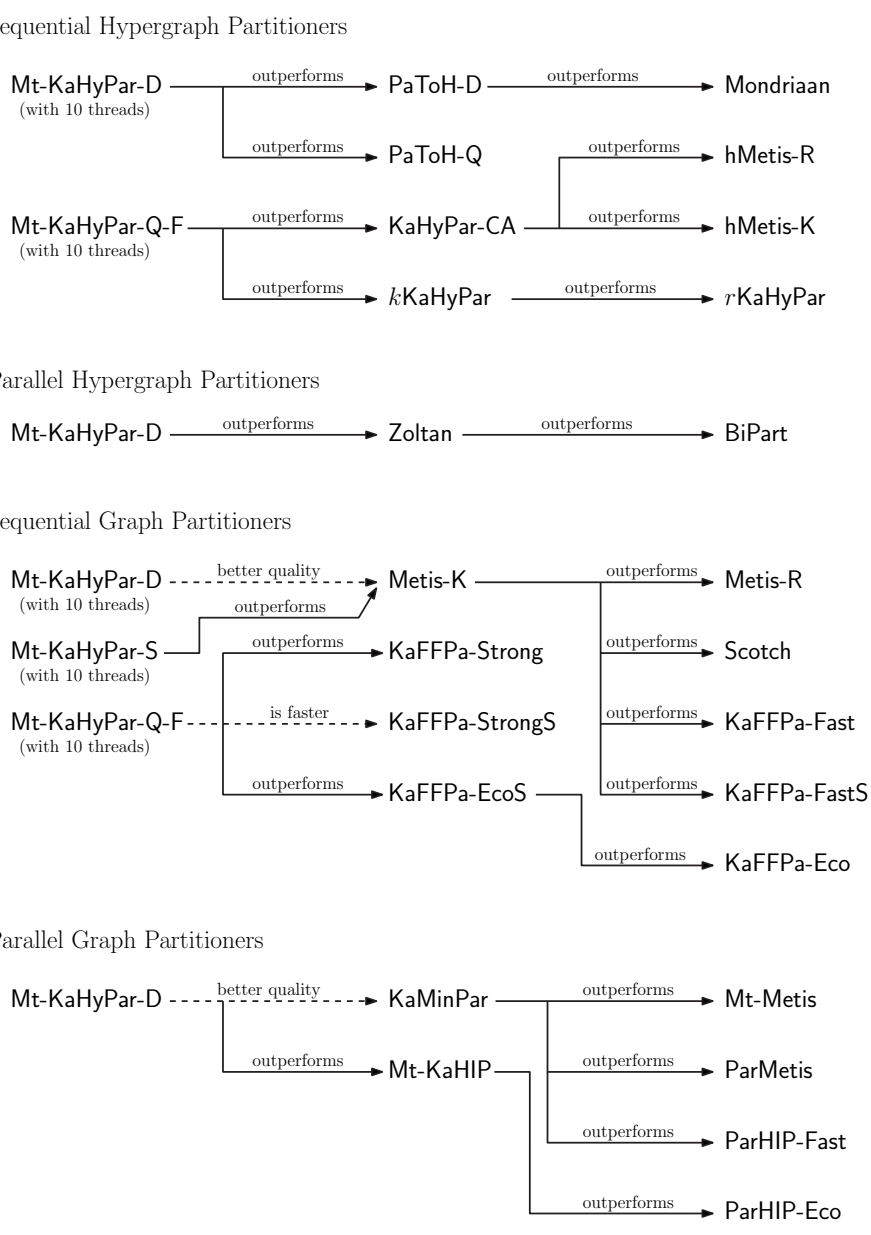



**Figure 8.15:** Summary of the experimental results.

literature.

Figure 8.15 summarizes our main results. We saw that sequential partitioning algorithms can be divided into systems aiming for speed or high solution quality. The latter are often more than an order of magnitude slower than the fast partitioning methods. For hypergraph partitioning, Mt-KaHyPar outperforms all existing sequential algorithms with only ten threads. It achieves the same solution quality as the currently highest-quality partitioner $k$KaHyPar, while it is an order of magnitude faster on average. Our fastest configuration Mt-KaHyPar-D is faster and produces significantly better partitions than the default preset of PaToH, which was the previous method of choice when speed is more important than quality. Furthermore, our highest-quality configuration Mt-KaHyPar-Q-F is slightly faster than the quality preset of PaToH, which can also be classified as a fast partitioning method. Thus, Mt-KaHyPar can be seen as a new type of system that is able to produce high quality solutions very fast.

Since we also optimized Mt-KaHyPar for graph partitioning, we extensively compared it to sequential graph partitioning algorithms. Here, Mt-KaHyPar also outperforms all existing systems, except for the highest-quality configuration of KaFFPa. KaFFPa-StrongS produces edge cuts that are better than those of Mt-KaHyPar-Q-F by 1% in the median, but it is 38.66 times slower on average. Moreover, it is more than two orders of magnitudes slower than our fastest configuration Mt-KaHyPar-D. Hence, Mt-KaHyPar is also the method of choice for graph partitioning unless one has unlimited time for computing the best possible solution.

On larger (hyper)graphs, where the running time of sequential algorithms becomes prohibitive, the distributed-memory hypergraph partitioner Zoltan produces partitions worse than those of our fastest configuration Mt-KaHyPar-D by 23% in the median, while it is also slower by a factor of 2.72 on average. Our flow-based refinement configuration Mt-KaHyPar-D-F produces partitions better than those of Zoltan by 34% in the median. This demonstrates that Mt-KaHyPar can compute high quality solutions for extremely large hypergraphs, which was previously possible only with sequential codes on medium-sized instances. The comparison to parallel graph partitioners reveals that Mt-KaHyPar also computes the best edge cuts for large graph instances. However, the shared-memory graph partitioner KaMinPar is still faster than Mt-KaHyPar, but its edge cuts are worse by almost 10% in the median. Hence, there is still a tradeoff for partitioning graphs: KaMinPar is the method of choice when speed is more important than quality, and Mt-KaHyPar should be used when one aims for high solution quality.

This study showed that Mt-KaHyPar produces high quality solutions in a fraction of the time needed by the best sequential systems. As a result, Mt-KaHyPar outperforms most publicly available partitioning algorithms and enables partitioning (hyper)graphs with more than *one* billion pins/edges with previously unattained quality.

**Limits of the Study.** In this experimental evaluation, we partitioned hypergraphs with up to *one* billion pins/edges in up to 128 blocks with an imbalance ratio $\varepsilon = 3\%$. We want to point out that there are still settings where the results of this evaluation do not apply.

For example, when $k$ is large ($k \in \mathcal{O}(\sqrt{n})$), then KaMinPar is the method of choice.

For this case, other partitioners often struggle to find balanced solutions or do not complete in a reasonable time frame [Got+21e]. The main performance issue in Mt-KaHyPar is that the memory required for storing the gain table depends on $k$. During the write-up of this thesis, we worked on a configuration to handle the large $k$ case. Here, we disable the FM algorithm and implement the deep multilevel scheme for initial partitioning [Got+21e]. However, the latter needs further engineering efforts to achieve comparable speed to KaMinPar. Moreover, external or distributed-memory algorithms are the only way to partition instances that do not fit into the main memory of a single machine.

Hypergraph partitioning with a tight balance constraint (e.g., $\varepsilon \approx 0\%$) poses additional challenges for refinement algorithms. In this setting, the set of possible node moves is drastically reduced. Here, KaFFPa implements some effective techniques based on negative cycle detection in the quotient graph [SS13] (an edge $(V_i, V_j)$ is labeled with the highest gain move from block $V_i$ to $V_j$).

Chapter 9

# 9 Multilevel Hypergraph Partitioning with Node Weights

Many real-world applications of hypergraph partitioning (HGP) use node weights to accurately model the underlying problem domain. The two most prominent examples are very large scale integration (VLSI) design [AK95; Kar+99] and the parallel computation of the sparse matrix-vector product [ÇA96]. In the former, HGP is used to divide a circuit into two or more blocks such that the number of external wires interconnecting circuit elements in different blocks is minimized. In this setting, each node is associated with a weight equal to the area of the respective circuit element [Alp98] and tightly-balanced partitions minimize the total area required by the physical circuit [DT97]. In the latter, HGP is used to optimize the communication volume for parallel computations of sparse matrix-vector products [ÇA96]. In the simplest hypergraph model, nodes correspond to rows and hyperedges to columns of the matrix (or vice versa) and a partition of the hypergraphs yields an assignment of matrix entries to processors [ÇA96]. The work of a processor (which can be measured in terms of the number of non-zero entries [Bis+12]) is integrated into the model by assigning each node a weight equal to its degree [ÇA96]. Tightly-balanced partitions hence ensure that the work is distributed evenly among the processors.

Despite the importance of weighted instances for real-world applications, the HGP research community mainly uses unweighted hypergraphs in experimental evaluations [Sch20]. These instances become weighted implicitly due to contractions in the coarsening phase. Many partitioners therefore incorporate techniques that prevent the formation of heavy nodes [Hau95; CA99; HS17a] during coarsening to facilitate finding a feasible solution during the initial partitioning phase [Sch20]. However, in practice, many weighted hypergraphs derived from real-world applications already contain heavy nodes – rendering the mitigation strategies of today's multilevel hypergraph partitioners ineffective. The popular ISPD98 VLSI benchmark set [Alp98], for example, includes instances in which nodes can weigh up to 10% of the total weight of the hypergraph.

**Contributions and Outline.** In this chapter, we revisit the hypergraph partitioning problem with node weights in the multilevel context and show that current state-of-the-art partitioners often struggle considerably to find feasible solutions for weighted instances under a tight balance constraint. After introducing basic notation in Section 9.1, we present a technique enabling partitioners based on the recursive bipartitioning scheme to reliably compute balanced $k$-way partitions in Section 9.2.

The proposed method balances the partition by pre-assigning a small subset of the heaviest nodes to the two blocks of each bipartition (i.e., treating them as *fixed nodes*) and optimizes the actual partitioning objective on the remaining nodes. We further establish a balance property for the pre-assigned nodes that is verifiable in polynomial time and, if fulfilled, leads to provable balance guarantees for the resulting $k$-way partition obtained via recursive bipartitioning. In the experimental evaluation in Section 9.3, we describe the integration of our algorithm into the sequential hypergraph partitioner KaHyPar and show that our approach is able to compute balanced partitions of high quality on a diverse set of weighted real-world instances.

**Remarks.** There are usually two approaches to compute balanced $k$-way partitions under a tight balance constraint in a multilevel partitioner: (i) we either ensure that initial partitioning finds a balanced partition and refinement only applies changes on the partition that satisfies the balance constraint, or (ii) we allow intermediate balance violations [DT97; CKM00a; CKM00b; Got+21e] and use rebalancing techniques to ensure that the final $k$-way partition is balanced [WCE97; Got+21e; Mal+21]. Given that finding a balanced $k$-way partition is a NP-hard problem [GJ79], both approaches do not guarantee balance. The method proposed in this work falls into the first category. We may also not be able to compute for all weighted instances a feasible solution. However, we established several theoretical results that leads to provable balance guarantees under certain assumptions and turned out to be extremly effective in practice. Furthermore, the approach can be integrated into almost all partitioning systems with minor changes (only support for fixed nodes is required).

A comparison to systems that follow the second approach is missing in this work. Most of the proposed techniques were developed for non-multilevel algorithms and are not integrated into existing state-of-the-art partitioners. We also believe that there is a large design space that is far from being exhausted. Thus, the integration of these techniques into multilevel partitioner is an interesting avenue for future research. Thus, the approach presented in this work might be only a first step providing theoretical foundations and raises attention on the research topic.

**References and Contributors.** This chapter is based on a conference publication [HMS21a] from which most text passages are copied verbatim. The paper was written by the author of this dissertation, except for the experimental evaluation that was done by Nikolai Maas. Sebastian Schlag assisted in the editing process of the publication. The idea of preassigning a small portion of the heaviest nodes to one of the blocks of each bipartition came from the author of this dissertation. The theoretical foundations and implementation was done by Nikolai Maas as part of his bachelor thesis [Maa20a], which was supervised by us. In our weekly meetings with Nikolai Maas, we identified challenges, discussed new algorithmic ideas, and defined the experimental setup for evaluating the new techniques.

## 9.1 Definitions and Notations

We will now provide some basic terminology required in the subsequent sections. In the following, we will refer to a bipartition with $\Pi_2 := \{V_1, V_2\}$ and to a $k$-way partition with $\Pi_k := \{V_1, \ldots, V_k\}$. We call a $k$-way partition $\Pi_k$ *$\varepsilon$-balanced* if each block $V_i \in \Pi_k$ satisfies the *balance constraint*: $c(V_i) \leq L_k := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some parameter $\varepsilon \in (0, 1)$. For bipartitions, we will denote the maximum allowed block weight with $L_2$ instead of $L_k$.

**Hypergraph Partitioning with Fixed Nodes.** Let $H = (V, E, c, \omega)$ be a weighted hypergraph. We refer to a $k$-way partition $\Psi_k = \{P_1, \ldots, P_k\}$ of a subset $P \subseteq V$ as a *$k$-way prepacking*. We call a node $v \in P$ a *fixed* node and a node $v \in V \setminus P$ an *ordinary* node. During partitioning, fixed nodes are not allowed to be moved to a different block of the partition. The *$k$-way hypergraph partitioning problem* initialized with a $k$-way prepacking $\Psi_k = \{P_1, \ldots, P_k\}$ is to find an $\varepsilon$-balanced $k$-way partition $\Pi_k = \{V_1, \ldots, V_k\}$ of a hypergraph $H$ that minimizes an objective function defined on the hyperedges and satisfies $\forall i \in \{1, \ldots, k\} : P_i \subseteq V_i$. In this chapter, we optimize the connectivity metric $\mathfrak{f}_{\lambda-1}(\Pi) := \sum_{e \in E} (\lambda(e) - 1) \cdot \omega(e)$.

**Most Balanced Partition Problem.** The *most balanced partition problem* is to find a $k$-way partition $\Pi_k$ of a weighted hypergraph $H = (V, E, c, \omega)$ such that $\max(\Pi_k) := \max_{V' \in \Pi_k} c(V')$ is minimized. For an optimal solution $\Pi_{\mathrm{OPT}}$ it holds that there exists no other $k$-way partition $\Pi'_k$ with $\max(\Pi'_k) < \max(\Pi_{\mathrm{OPT}})$. We use $\mathrm{OPT}(H, k) := \max(\Pi_{\mathrm{OPT}})$ to denote the weight of the heaviest block of an optimal solution.

Note that the problem is equivalent to the most common version of the *job scheduling* problem: Given a sequence $J = \langle j_1, \ldots, j_n \rangle$ of $n$ computing jobs each associated with a *processing time* $p_i$ for $i \in [1, n]$, the task is to find an assignment of the $n$ jobs to $k$ identical machines (each job $j_i$ runs exclusively on a machine for exactly $p_i$ time units) such that the latest completion time of a job is minimized. The job scheduling problem is NP-hard [GJ79] and we refer the reader to existing literature [Gra+79; Pin12] for a comprehensive overview of the research topic.

**Longest Processing Time Algorithm.** In this work, we make use of the *longest processing time* (LPT) algorithm proposed by Graham [Gra69]. We will explain the algorithm in the context of the most balanced partition problem: For a weighted hypergraph $H = (V, E, c, \omega)$, the algorithm iterates over the nodes of $V$ sorted in decreasing node-weight order and assigns each node to the block of the $k$-way partition with the lowest weight. The algorithm can be implemented to run in $\mathcal{O}(n \log n)$ time, and for a $k$-way partition $\Pi_k$ produced by the algorithm it holds that $\max(\Pi_k) \leq (\frac{4}{3} - \frac{1}{3k})\mathrm{OPT}(H, k)$.

**Adaptive Imbalance Ratio.** The hypergraph partitioner KaHyPar [Sch+16a] ensures that a $k$-way partition computed via recursive bipartitioning (RB) is balanced by adapting the imbalance ratio for each bipartition individually. Let $H_{V'}$ be the

subhypergraph of the current bipartition that should be partitioned recursively into $k' \leq k$ blocks. Then,

$$\varepsilon' := \left( (1+\varepsilon) \frac{c(V)}{k} \cdot \frac{k'}{c(V')} \right)^{\frac{1}{\lceil \log_2(k') \rceil}} - 1 \tag{9.1}$$

is the imbalance ratio used for the bipartition of $H_{V'}$. The equation is based on the observation that the worst-case block weight of the resulting $k'$-way partition of $H_{V'}$ obtained via RB is smaller than $(1+\varepsilon')^{\lceil \log_2(k') \rceil} \frac{c(V')}{k'}$, if $\varepsilon'$ is used for all further bipartitions. Requiring that this weight must be smaller or equal to $L_k = (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ leads to the formula defined in Equation 9.1.

## 9.2 Balanced Recursive Bipartitioning

Most multilevel hypergraph partitioners either employ recursive bipartitioning directly [CA99; Kar+99; VB05; Dev+06; Sch+16a] or use RB-based algorithms in the initial partitioning phase to compute an initial $k$-way partition of the coarsest hypergraph [KK00; ACU08b; Akh+17a]. In both settings, a $k$-way partition is derived by first computing a bipartition $\Pi_2 = \{V_1, V_2\}$ of the (input/coarse) hypergraph $H$ and then recursing on the subhypergraphs $H_{V_1}$ and $H_{V_2}$ by partitioning $V_1$ into $\lceil \frac{k}{2} \rceil$ and $V_2$ into $\lfloor \frac{k}{2} \rfloor$ blocks. Although KaHyPar adaptively adjusts the allowed imbalance at each bipartitioning step (using the imbalance factor $\varepsilon'$ as defined in Equation 9.1), an *unfortunate* distribution of the nodes in some bipartitions $\Pi_2$ can easily lead to instances for which it is impossible to find a balanced solution during the recursive partitioning calls – even though the current bipartition $\Pi_2$ satisfies the adjusted balance constraint.

> An example is shown in Figure 9.1 (left): Although the current bipartition (indicated by the red line) is perfectly balanced, it will not be possible to recursively partition the subhypergraph induced by the nodes of $V_2$ into two blocks of equal weight, because each of the three nodes has a weight of four.

### 9.2.1 Deeply Balanced Bipartitions

To capture this problem, we introduce the notion of *deep balance*:

**Definition 9.1 (Deep Balance)**
*Let $H = (V, E, c, \omega)$ be a weighted hypergraph for which we want to compute an $\varepsilon$-balanced $k$-way partition, and let $H_{V'}$ be a subhypergraph of $H$ which should be partitioned into $k' \leq k$ blocks via recursive bipartitioning. A subhypergraph $H_{V'}$ is deeply balanced w.r.t. $k'$, if there exists a $k'$-way partition $\Pi_{k'}$ of $H_{V'}$ such that $\max(\Pi_{k'}) \leq L_k := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$. A bipartition $\Pi_2 = \{V_1, V_2\}$ of $H_{V'}$ is deeply balanced*

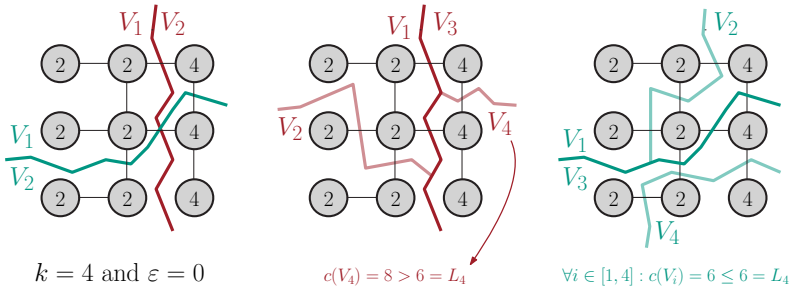



**Figure 9.1:** Illustration of a deeply (left, green line) and a non-deeply balanced bipartition (left, red line). The numbers in each circle denotes the node weights. In both cases, the hypergraph is partitioned into $k = 4$ blocks with $\varepsilon = 0$ via recursive bipartitioning. Thus, the weight of heaviest block must be smaller or equal to $L_4 = 6$ and for the first bipartition, we use $L_2 = 12$ as an upper bound.

*w.r.t.* $k'$*, if the subhypergraphs* $H_{V_1}$ *and* $H_{V_2}$ *are deeply balanced with respect to* $\lceil \frac{k'}{2} \rceil$ *resp.* $\lfloor \frac{k'}{2} \rfloor$*.*

If a subhypergraph $H_{V'}$ is deeply balanced with respect to $k'$, there always exists a $k'$-way partition $\Pi_{k'}$ of $H_{V'}$ such that weight of the heaviest block satisfies the original balance constraint $L_k$ imposed on the partition of the input hypergraph $H$. Moreover, there also always exists a deeply balanced bipartition $\Pi_2 := \{V_1, V_2\}$ ($V_1$ is the union of the first $\lceil \frac{k'}{2} \rceil$ and $V_2$ of the last $\lfloor \frac{k'}{2} \rfloor$ blocks of $\Pi_{k'}$). Hence, a RB-based partitioning algorithm that is able to compute deeply balanced bipartitions on deeply balanced subhypergraphs will always compute $\varepsilon$-balanced $k$-way partitions (assuming the input hypergraph is deeply balanced).

**Deep Balance and Adaptive Imbalance Adjustments.** Computing deeply balanced bipartitions in the RB setting guarantees that the resulting $k$-way partition is $\varepsilon$-balanced. Thus, the concept of deep balance could replace the adaptive imbalance ratio $\varepsilon'$ employed in KaHyPar [Sch+16a] (see Equation 9.1). However, as we will see in the following example, combining both approaches gives the partitioner more flexibility (in terms of feasible node moves during refinement). Assume that we want to compute a 4-way partition via recursive bipartitioning and that the first bipartition $\Pi_2 := \{V_1, V_2\}$ is deeply balanced with $c(V_1) = (1 + \varepsilon)\lceil \frac{c(V)}{2} \rceil$. The deep-balance property ensures that we can further partition $V_1$ into two blocks such that the weight of the heavier block is smaller than $L_4$. However, this bipartition has to be perfectly balanced:

$$L_2 = (1 + \bar{\varepsilon})\left\lceil \frac{c(V_1)}{2} \right\rceil = (1 + \bar{\varepsilon})\left\lceil \frac{(1 + \varepsilon)\lceil \frac{c(V)}{2} \rceil}{2} \right\rceil \leq (1 + \varepsilon)\left\lceil \frac{c(V)}{4} \right\rceil = L_4 \Rightarrow \bar{\varepsilon} \approx 0.$$

If we would have computed the first bipartition with an adjusted imbalance ratio $\varepsilon'$, then $\max(\Pi_2) \leq (1 + \varepsilon')\lceil \frac{c(V)}{2} \rceil = \sqrt{1 + \varepsilon}\lceil \frac{c(V)}{2} \rceil$ – providing more flexibility for

subsequent bipartitions. In the following, we therefore focus on computing deeply $\varepsilon'$-balanced bipartitions.

### 9.2.2 Sufficiently Balanced Bipartitions

In general, computing a deeply balanced bipartition $\Pi_2 := \{V_1, V_2\}$ w.r.t. $k$ is NP-hard, as we must show that there exists a $k$-way partition $\Pi_k$ of $H$ with $\max(\Pi_k) \leq L_k$, which can be reduced to the most balanced partition problem presented in Section 9.1. However, we can first compute a $k$-way partition $\Pi_k := \{V'_1, \ldots, V'_k\}$ using the LPT algorithm, thereby approximating an optimal solution. If $\max(\Pi_k) \leq L_k$, we can then construct a deeply balanced bipartition $\Pi_2 = \{V_1, V_2\}$ by choosing $V_1 := V'_1 \cup \ldots \cup V'_{\lceil \frac{k}{2} \rceil}$ and $V_2 := V'_{\lceil \frac{k}{2} \rceil + 1} \cup \ldots \cup V'_k$. Unfortunately, this approach completely ignores the optimization of the objective function – yielding balanced partitions of low quality. If such a bipartition were to be used as initial solution in the multilevel setting, the objective could still be optimized during the refinement phase. However, this would necessitate that refinement algorithms are aware of the concept of deep balance and that they only perform node moves that do not destroy the deep-balance property of the starting solution. Since this is infeasible in practice, we propose a different approach that involves fixed nodes.

The key idea of our approach is to compute a prepacking $\Psi = \{P_1, P_2\}$ of the $|P_1| + |P_2|$ heaviest nodes of the hypergraph and to show that this prepacking suffices to ensure that each $\varepsilon'$-balanced bipartition $\Pi_2 = \{V_1, V_2\}$ with $P_1 \subseteq V_1$ and $P_2 \subseteq V_2$ is deeply balanced. Note that the upcoming definitions and theorems are formulated from the perspective of the first bipartition of the input hypergraph $H$ to simplify notation. They can be generalized to subhypergraphs $H_{V'}$ in a similar fashion as was done in Definition 9.1. Furthermore, we say that the bipartition $\Pi_2 = \{V_1, V_2\}$ respects a prepacking $\Psi = \{P_1, P_2\}$, if $P_1 \subseteq V_1$ and $P_2 \subseteq V_2$, and that the bipartition is balanced, if $\max(\Pi_2) \leq L_2 := (1 + \varepsilon')\lceil \frac{c(V_1 \cup V_2)}{2} \rceil$ (with $\varepsilon'$ as defined in Equation 9.1). The following definition formalizes our idea.

**Definition 9.2 (Sufficiently Balanced Prepacking)**
*Let $H = (V, E, c, \omega)$ be a hypergraph for which we want to compute an $\varepsilon$-balanced $k$-way partition via recursive bipartitioning. We call a prepacking $\Psi$ of $H$ sufficiently balanced if every balanced bipartition $\Pi_2$ respecting $\Psi$ is deeply balanced with respect to $k$.*

Our approach to compute $\varepsilon$-balanced $k$-way partitions is outlined in Algorithm 9.1. We first compute a bipartition $\Pi_2$. Before recursing on each of the two induced subhypergraphs, we check if $\Pi_2$ is deeply balanced using the LPT algorithm in a similar fashion as described in the beginning of this section. If it is not deeply balanced, we compute a sufficiently balanced prepacking $\Psi$ and re-compute $\Pi_2$ – treating the nodes of the prepacking as fixed nodes. If this second bipartitioning call was able to compute a balanced bipartition, we found a deeply balanced partition and proceed to partition the subhypergraphs recursively.

Note that, in general, we may not detect that $\Pi_2$ is deeply balanced or fail to find a sufficiently balanced prepacking $\Psi$ or a balanced bipartition $\Pi_2$, since all involved problems are NP-hard. However, as we will see in the experimental evaluation, Algorithm 9.1 computes balanced partitions for all instances of our large real-world benchmark set. This indicates that the above-mentioned problems only happen rarely in practice.

**Algorithm 9.1:** Recursive Bipartitioning Algorithm

**Input:** Hypergraph $H$ for which we seek an $\varepsilon$-balanced $k$-way partition and subhypergraph $H_{V'}$ of $H$ which is to be bipartitioned recursively into $k' \leq k$ blocks.

**Output:** $k'$-way partition $\Pi_{k'}$ of $H_{V'}$

```
1 L_2 ← (1 + ε')⌈c(V')/2⌉                          // with ε' as defined in Equation 9.1
2 Π_2 := {V_1, V_2} ← multilevelBipartitioning(H_V', L_2)
3 if k' = 2 then return Π_2
4 else if Π_2 is not deeply balanced w.r.t. k' then
5     Ψ ← sufficientlyBalancedPrepacking(H, k, ε, H_V', k')   // see Algorithm 9.2
6     Π_2 ← multilevelBipartitioning(H_V', L_2, Ψ)    // treating Ψ as fixed nodes
7 Π_{k_1} ← recursiveBipartitioning(H, k, ε, H_{V_1}, k_1) with k_1 := ⌈k'/2⌉
8 Π_{k_2} ← recursiveBipartitioning(H, k, ε, H_{V_2}, k_2) with k_2 := ⌊k'/2⌋
9 return Π_{k'} ← Π_{k_1} ∪ Π_{k_2}
```

The prepacking $\Psi$ is constructed by incrementally assigning nodes to $\Psi$ in decreasing order of weight and checking a property $\mathcal{P}$ after each assignment that, if satisfied, implies that the current prepacking is sufficiently balanced. In the proof of property $\mathcal{P}$, we will extend a $k$-way prepacking $\Psi_k$ to an $\varepsilon$-balanced $k$-way partition $\Pi_k$ using the LPT algorithm and use the following upper bound on the weight of the heaviest block of $\Pi_k$.

**Lemma 9.3 (The LPT Bound)**

*Let $H = (V, E, c, \omega)$ be a weighted hypergraph, $\Psi_k$ be a $k$-way prepacking for a set of fixed nodes $P \subseteq V$, and let $O := \langle v_1, \ldots, v_m \mid v_i \in V \setminus P \rangle$ be the sequence of all ordinary nodes of $V \setminus P$ sorted in decreasing order of weight. If we assign the remaining nodes $O$ to the blocks of $\Psi_k$ by using the* LPT *algorithm, we can extend $\Psi_k$ to a $k$-way partition $\Pi_k$ of $H$ such that the weight of the heaviest block is bounded by:*

$$\max(\Pi_k) \leq \max\{\frac{1}{k}c(P) + h_k(O), \max(\Psi_k)\}$$

$$\text{with } h_k(O) := \max_{i \in \{1,\ldots,m\}} c(v_i) + \frac{1}{k}\sum_{j=1}^{i-1} c(v_j).$$

*Proof.* We define $\Psi_k := \{P_1, \ldots, P_k\}$ and $\Pi_k := \{V_1, \ldots, V_k\}$. Let assume that the LPT algorithm assigned the $i$-th node $v_i$ of $O$ to block $V_j \in \Pi_k$. We define $V_j^{(i)}$ as

a subset of block $V_j$ that only contains nodes of $\langle v_1, \ldots, v_i \rangle \subseteq O$ and $P$. Since the LPT algorithm always assigns a node to a block with the smallest weight, the weight of $V_j^{(i-1)}$ must be smaller or equal to $\frac{1}{k}(c(P) + \sum_{j=1}^{i-1} c(v_j))$ (average weight of all previously assigned nodes), otherwise $V_j^{(i-1)}$ would be not the block with the smallest weight.

$$\Rightarrow c(V_j^{(i)}) = c(V_j^{(i-1)}) + c(v_i) \leq \frac{1}{k}(c(P) + \sum_{j=1}^{i-1} c(v_j)) + c(v_i) \leq \frac{1}{k}c(P) + h_k(O)$$

We can establish an upper bound on the weight of all blocks to which the LPT algorithm assigns a node to with $\frac{1}{k}c(P) + h_k(O)$. If the LPT algorithm does not assign any node to a block $V_j \in \Pi_k$, its weight equals $c(P_j) \leq \max(\Psi_k)$.

$$\Rightarrow \max(\Pi_k) \leq \max\{\frac{1}{k}c(P) + h_k(O), \max(\Psi_k)\}$$ □

$O$ is sorted in decreasing order of weight because for any permutation $O'$ of $O$, it holds that $h_k(O) \leq h_k(O')$ – resulting in the tightest bound for $\max(\Pi_k)$.

Assuming that the number $k$ of blocks is even (i.e., $k_1 = k_2 = k/2$) to simplify notation, the balance property $\mathcal{P}$ is defined as follows (for the generalized version we refer the reader to Ref. [HMS21a]):

**Definition 9.4 (Balance Property $\mathcal{P}$)**

*Let $H = (V, E, c, \omega)$ be a hypergraph for which we want to compute an $\varepsilon$-balanced $k$-way partition and let $\Psi$ be a prepacking of $H$ for a set of fixed nodes $P \subseteq V$. Furthermore, let $O_t := \langle v_1, \ldots, v_t \rangle$ be the sequence of the $t$ heaviest ordinary nodes of $V \setminus P$ sorted in decreasing order of weight such that $t$ is the smallest number that satisfies $\max(\Psi) + c(O_t) \geq L_2$. We say that the prepacking $\Psi$ satisfies the balance property $\mathcal{P}$ if the following two conditions hold:*

*(i) the prepacking $\Psi$ is deeply balanced*

*(ii)* $\frac{1}{k/2}\max(\Psi) + h_{k/2}(O_t) \leq L_k$.

In the following, we will show that the LPT algorithm can be used to construct a $k/2$-way partition $\Pi_{k/2}$ for both blocks of any balanced bipartition $\Pi_2 = \{V_1, V_2\}$ that respects $\Psi$, such that the weight of the heaviest block can be bound by the left term of Condition (ii). This implies that $\max(\Pi_{k/2}) \leq L_k$ (right term of Condition (ii)) and thus proofs that any balanced bipartition $\Pi_2$ respecting $\Psi$ is deeply balanced. Note that choosing $t$ as the smallest number that satisfies $\max(\Psi) + c(O_t) \geq L_2$ minimizes the left term of Condition (ii) (since $h_k(O_t) \leq h_k(O_{t+1})$).

**Theorem 9.5**

*A prepacking $\Psi$ of a hypergraph $H = (V, E, c, \omega)$ that satisfies the balance property $\mathcal{P}$ is sufficiently balanced with respect to $k$.*

*Proof.* For convenience, we use $k' := k/2$. Let $\Pi_2 = \{V_1, V_2\}$ be an abitrary balanced bipartition that respects the prepacking $\Psi = \{P_1, P_2\}$ with $\max(\Pi_2) \leq L_2$. Since $\Psi$ is deeply balanced (see Definition 9.4(i)), there exists a $k'$-way prepacking $\Psi_{k'}$ of $P_1$ such that $\max(\Psi_{k'}) \leq L_k$. We define the sequence of the ordinary nodes of block $V_1$ sorted in decreasing weight order with $O_1 := \langle v_1, \ldots, v_m \mid v_i \in V_1 \setminus P_1 \rangle$. We can extend $\Psi_{k'}$ to a $k'$-way partition $\Pi_{k'}$ of $V_1$ by assigning the nodes of $O_1$ to the blocks in $\Psi_{k'}$ using the LPT algorithm. Lemma 9.3 then establishes an upper bound on the weight of the heaviest block.

$$\begin{aligned}\max(\Pi_{k'}) &\overset{\text{Lemma 9.3}}{\leq} \max\{\frac{1}{k'}c(P_1) + h_{k'}(O_1), \max(\Psi_{k'})\} \\ &\overset{\max(\Psi_{k'}) \leq L_k}{\leq} \max\{\frac{1}{k'}c(P_1) + h_{k'}(O_1), L_k\}\end{aligned}$$

Let $O_t$ be the sequence of the $t$ heaviest ordinary nodes of $V \setminus P$ with $P := P_1 \cup P_2$ as defined in Definition 9.4.

**Claim 9.6**

*It holds that:* $\frac{1}{k'}c(P_1) + h_{k'}(O_1) \leq \frac{1}{k'}\max(\Psi) + h_{k'}(O_t)$.

*Proof.* see Section 9.4 □

We can conclude that

$$\frac{1}{k'}c(P_1) + h_{k'}(O_1) \overset{\text{Claim 9.6}}{\leq} \frac{1}{k'}\max(\Psi) + h_{k'}(O_t) \overset{\text{Definition 9.4(ii)}}{\leq} L_k.$$

This proves that the subhypergraph $H_{V_1}$ is deeply balanced. The proof for block $V_2$ can be done analogously, which then implies that $\Pi_2$ is deeply balanced. Since $\Pi_2$ is an abitrary balanced bipartition respecting $\Psi$, it follows that $\Psi$ is sufficiently balanced. □

### 9.2.3 The Prepacking Algorithm

Algorithm 9.2 outlines our approach to efficiently compute a sufficiently balanced prepacking $\Psi$. In Line 5, we compute a $k'$-way prepacking $\Psi_{k'}$ of the $i$ heaviest nodes with the LPT algorithm and if $\Psi_{k'}$ satisfies $\max(\Psi_{k'}) \leq L_k$, then Line 6 constructs a deeply balanced prepacking $\Psi$ (which fullfils Condition (i) of Definition 9.4). We store the blocks $P'_j$ of $\Psi_{k'}$ together with their weights $c(P'_j)$ as key in an addressable priority queue such that we can determine and update the block with the smallest weight in time $\mathcal{O}(\log k')$ (Line 5). In Line 8, we compute the smallest $t$ that satisfies $\max(\Psi) + c(O_t) \geq L_2$ via a binary search in logarithmic time over an array containing the node weight prefix sums of the sequence $O$, which can be precomputed in linear time. Furthermore, we construct a range maximum query data structure over the array $H_{k'/2} = \langle c(v_1), c(v_2) + \frac{1}{k'/2}c(v_1), \ldots, c(v_n) + \frac{1}{k'/2}\sum_{j=1}^{n-1} c(v_j) \rangle$. Caculating $h_{k'/2}(O_t)$ then corresponds to a range maximum query in the interval $[i+1, i+t]$ in $H_{k'/2}$, which can be answered in constant time after $H_{k'/2}$ has been precomputed in time $\mathcal{O}(n)$ [BF00].

**Algorithm 9.2:** Prepacking Algorithm

**Input:** Hypergraph $H = (V, E, c, \omega)$ for which we seek an $\varepsilon$-balanced $k$-way partition and subhypergraph $H_{V'} = (V', E', c, \omega)$ of $H$ which is to be to bipartitioned recursively into $k' \leq k$ blocks.

**Output:** Sufficiently Balanced Prepacking $\Psi$ of $V'$

1 $\Psi = \langle P_1, P_2 \rangle \leftarrow \langle \emptyset, \emptyset \rangle$ and $\Psi_{k'} = \langle P'_1, \ldots, P'_{k'} \rangle \leftarrow \langle \emptyset, \ldots, \emptyset \rangle$ // *Initialization*

2 $L_2 \leftarrow (1+\varepsilon')\lceil \frac{c(V')}{2} \rceil$ and $L_k \leftarrow (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ // *with $\varepsilon'$ as defined in Equation 9.1*

3 $O \leftarrow \langle v_1, \ldots, v_n \mid v_i \in V' \rangle$ // *$V'$ sorted in decreasing order of weight $\Rightarrow \mathcal{O}(n \log n)$*

4 **for** $i = 1$ to $n$ **do**

5 Add $v_i \in O$ to bin $P'_j \in \Psi_{k'}$ with smallest weight // LPT *algorithm*

6 $\Psi \leftarrow \{P'_1 \cup \ldots \cup P'_x, P'_{x+1} \cup \ldots \cup P'_{k'}\}$ with $x := \lceil \frac{k'}{2} \rceil$

7 **if** $\max(\Psi) \leq L_2$ and $\max(\Psi_{k'}) \leq L_k$ **then** // *$\Rightarrow \Psi$ is deeply ($\varepsilon'$-)balanced*

8 $t \leftarrow \min(\{t \mid \max(\Psi) + c(O_t) \geq L_2\})$ // *$O_t := \langle v_{i+1}, \ldots, v_{i+t} \rangle$*

9 **if** $\frac{2}{k'} \max(\Psi) + h_{k'/2}(O_t) \leq L_k$ **then** // *Condition (ii) of Definition 9.4*

10 **return** $\Psi$ // *$\Rightarrow \Psi$ is sufficiently balanced (Theorem 9.5)*

11 **return** $\Psi$ // *No sufficiently balanced prepacking found $\Rightarrow$ treat all nodes as fixed nodes*

In total, the running time of the algorithm is $\mathcal{O}(n(\log k' + \log n))$. Note that if the algorithm reaches Line 11, we could not prove that any of the intermediate constructed prepackings were sufficiently balanced, in which case $\Psi$ represents a bipartition of $H_{V'}$ computed by the LPT algorithm.

## 9.3 Experiments

We integrated the prepacking technique (see Algorithms 9.1 and 9.2) into the recursive bipartitioning algorithm of the sequential hypergraph partitioner KaHyPar. Our implementation is available from `http://www.kahypar.org`. The code is written in `C++17` and compiled using `g++9.2` with the flags `-mtune=native -O3 -march=native`. Since KaHyPar offers both a recursive bipartitioning and direct $k$-way partitioning algorithm (which uses the RB algorithm in the initial partitioning phase), we refer to the RB-version using our improvements as $r$KaHyPar-BP and to the direct $k$-way version as $k$KaHyPar-BP (`BP` = **B**alanced **P**artitioning).

**Instances.** The following experimental evaluation is based on a benchmark set that consists of 50 hypergraphs originating from the VLSI design and scientific computing domain. It contains instances from the ISPD98 VLSI Circuit Benchmark Suite [Alp98] (18 instances), the DAC 2012 Routability-Driven Placement Benchmark Suite [Vis+12] (9 instances), 16 instances from the Stanford Network Analysis (SNAP) Platform [LK14], and 7 highly asymmetric matrices of Davis et al. [DDN20] (referred to as ASM). For VLSI instances (ISPD98 and DAC), we use the area of a circuit

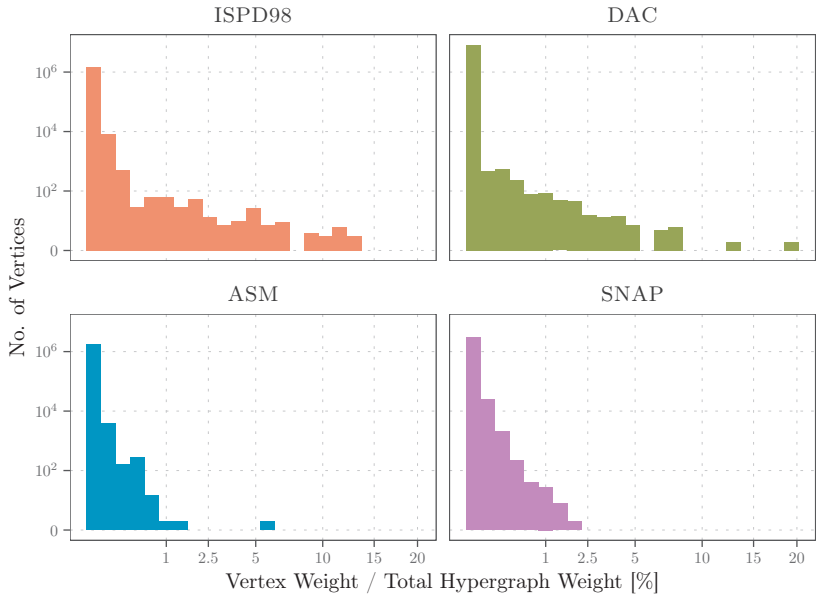



**Figure 9.2:** Distributions of node weights for different instance types in our benchmark sets. Each bucket of a histogram shows the number of nodes (y-axis) that contribute $x\%$ to the total weight of the corresponding hypergraph.

element as the weight of its corresponding node. We translate sparse matrices (SNAP and ASM instances) to hypergraphs using the row-net model [CA99] and use the degree of a node as its weight. The node weight distributions of the individual instance types are depicted in Figure 9.2.

**System and Methodology.** All experiments are performed on a single core of a cluster with Intel Xeon Gold 6230 processors running at 2.1 GHz with 96GB RAM. We partition each hypergraph into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks, and use $\varepsilon \in \{0.01, 0.03, 0.1\}$ and ten repetitions using different seeds for each combination of $k$ and $\varepsilon$, and a time limit of eight hours. We call a combination of a hypergraph $H = (V, E, c, \omega)$, $k$, and $\varepsilon$ an *instance*.

We compare $r$KaHyPar-BP and $k$KaHyPar-BP with the latest recursive bipartitioning ($r$KaHyPar) and direct $k$-way version ($k$KaHyPar) of KaHyPar [Sch+16a; Akh+17a; Got+20], the default (PaToH-D) and quality preset (PaToH-Q) of PaToH 3.3 [CA99], as well as with the recursive bipartitioning (hMetis-R) and direct $k$-way version (hMetis-K) of hMetis 2.0 [Kar+99; KK00]. We also include the default (Mt-KaHyPar-D) and quality preset (Mt-KaHyPar-Q-F) of our new shared-memory hypergraph partitioner Mt-KaHyPar (using 10 threads). Each partitioner minimizes the connectivity metric.

### 9.3.1 The LPT Balance Constraint

A $k$-way partition of a weighted hypergraph $H = (V, E, c, \omega)$ is balanced, if the weight of each block is below some predefined upper bound. In the literature, the most commonly used bounds are $L_k := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ (standard definition) and $L_k^{\max} := L_k + \max_{v \in V} c(v)$ [FM82; Sch13; Sch20]. The latter was initially proposed by Fiduccia and Mattheyses [FM82] for bipartitioning to ensure that the highest gain node can always be moved to the opposite block.

Both definitions exhibit shortcomings in the presence of heavy nodes: As soon as the hypergraph contains even a single node with $c(v) > L_k$, no feasible solution exists when the block weights are constrained by $L_k$, while for $L_k^{\max}$ it follows that $L_k^{\max} > 2L_k$ – allowing large variations in block weights even if $\varepsilon$ is small. In the following, we therefore propose a new balance constraint that (i) guarantees the existence of an $\varepsilon$-balanced $k$-way partition and (ii) avoids unnecessarily large imbalances.

While the optimal solution of the most balanced partition problem would yield a partition with the best possible balance, it is not feasible in practice to use $L_k^{\mathrm{OPT}} := (1+\varepsilon)\mathrm{OPT}(H,k)$ as balance constraint, because finding such a $k$-way partition is NP-hard [GJ79]. Hence, we propose to use the bound provided by the LPT algorithm instead:

$$L_k^{\mathrm{LPT}} := (1+\varepsilon)\ \mathrm{LPT}(H,k) \leq \left(\frac{4}{3} - \frac{1}{3k}\right) L_k^{\mathrm{OPT}}. \tag{9.2}$$

If the hypergraph is unweighted, the LPT algorithm will always find an optimal solution with $\mathrm{OPT}(H,k) = \lceil \frac{|V|}{k} \rceil$ and thus, $L_k^{\mathrm{LPT}}$ equals $L_k$.

Note that since all evaluated partitioners internally employ $L_k$ as balance constraint, we initialize each partitioner with a modified imbalance factor $\hat{\varepsilon}$ instead of $\varepsilon$ which is calculated as follows:

$$L_k = (1+\hat{\varepsilon})\left\lceil \frac{c(V)}{k} \right\rceil = (1+\varepsilon)\mathrm{LPT}(H,k) = L_k^{\mathrm{LPT}} \Rightarrow \hat{\varepsilon} = \frac{L_k^{\mathrm{LPT}}}{\lceil \frac{c(V)}{k} \rceil} - 1.$$

We consider the resulting $k$-way partition $\Pi_k$ to be imbalanced, if it is not $\hat{\varepsilon}$-balanced. Furthermore, we remove all nodes $v \in V$ from $H$ with a weight greater than $L_k = (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ as proposed by Caldwell et al. [CKM00b] and adapt $k$ to $k' := k - |V_R|$, where $V_R$ represents the set of removed nodes. We repeat that step recursively until there is no node with a weight greater than $L_{k'} := (1+\varepsilon)\lceil \frac{c(V \setminus V_R)}{k'} \rceil$. The input for each partitioner is the subhypergraph $H_{V \setminus V_R}$ of $H$ for which we compute a $k'$-way partition with $L_{k'}^{\mathrm{LPT}}$ as maximum allowed block weight.

### 9.3.2 Balanced Partitioning

In Table 9.1, we report the percentage of imbalanced instances produced by each partitioner for each instance type and $\varepsilon$. Both $k$KaHyPar-BP and $r$KaHyPar-BP compute balanced partitions for all tested instances. For the remaining partitioners, the number of imbalanced solutions increases as the balance constraint becomes

**Table 9.1:** Percentage of imbalanced instances produced by each partitioner for each combination of instance type and $\varepsilon$.

| | ISPD98 | | | DAC | | | ASM | | | SNAP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\varepsilon$ | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 |
| $k$KaHyPar-BP | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| $r$KaHyPar-BP | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| $k$KaHyPar | 6.3 | 5.6 | 0.8 | 9.5 | 7.9 | 6.3 | 4.1 | 4.1 | 2.0 | 0.9 | 0.9 | 0.0 |
| $r$KaHyPar | 10.3 | 8.7 | 7.1 | 19.0 | 19.0 | 14.3 | 6.1 | 4.1 | 4.1 | 6.2 | 2.7 | 0.9 |
| Mt-KaHyPar-D | 4.0 | 7.1 | 1.6 | 11.1 | 9.5 | 4.8 | 4.1 | 4.1 | 2.0 | 3.6 | 2.7 | 0.0 |
| Mt-KaHyPar-Q-F | 5.6 | 5.6 | 1.6 | 11.1 | 12.7 | 4.8 | 4.1 | 4.1 | 2.0 | 3.6 | 1.8 | 0.0 |
| hMetis-R | 17.5 | 15.1 | 7.1 | 20.6 | 15.9 | 12.7 | 8.2 | 6.1 | 4.1 | 15.2 | 10.7 | 4.5 |
| hMetis-K | 43.7 | 22.2 | 9.5 | 33.3 | 22.2 | 11.1 | 67.3 | 32.7 | 4.1 | 33.9 | 20.5 | 3.6 |
| PaToH-Q | 15.9 | 11.1 | 5.6 | 23.8 | 17.5 | 9.5 | 24.5 | 6.1 | 4.1 | 33.9 | 8.0 | 1.8 |
| PaToH-D | 9.5 | 7.9 | 3.2 | 20.6 | 17.5 | 9.5 | 28.6 | 6.1 | 4.1 | 22.3 | 11.6 | 2.7 |

**Table 9.2:** Percentage of imbalanced instances produced by each partitioner for each combination of $k$ and $\varepsilon$.

| | $k \in \{2,4,8\}$ | | | $k \in \{16,32\}$ | | | $k \in \{64,128\}$ | | | All | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\varepsilon$ | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 |
| $k$KaHyPar-BP | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| $r$KaHyPar-BP | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| $k$KaHyPar | 0.7 | 0.0 | 0.0 | 5.0 | 6.0 | 2.0 | 11.1 | 9.1 | 4.0 | 4.9 | 4.3 | 1.7 |
| $r$KaHyPar | 2.0 | 0.7 | 0.7 | 11.0 | 9.0 | 7.0 | 21.0 | 18.0 | 13.0 | 10.0 | 8.0 | 6.0 |
| Mt-KaHyPar-D | 0.7 | 0.7 | 0.0 | 6.0 | 8.0 | 3.0 | 11.0 | 11.0 | 3.0 | 5.1 | 5.7 | 1.7 |
| Mt-KaHyPar-Q-F | 0.0 | 0.0 | 0.0 | 7.0 | 8.0 | 3.0 | 13.0 | 11.0 | 3.0 | 5.7 | 5.4 | 1.7 |
| hMetis-R | 2.7 | 2.0 | 0.0 | 18.0 | 14.0 | 7.0 | 34.0 | 27.0 | 17.0 | 16.0 | 12.6 | 6.9 |
| hMetis-K | 12.0 | 2.0 | 0.0 | 53.0 | 21.0 | 11.0 | 76.0 | 57.0 | 14.0 | 42.0 | 23.1 | 7.1 |
| PaToH-Q | 15.3 | 2.7 | 0.7 | 28.0 | 11.0 | 5.0 | 34.0 | 22.0 | 11.0 | 24.3 | 10.6 | 4.9 |
| PaToH-D | 9.3 | 2.7 | 0.7 | 18.0 | 11.0 | 4.0 | 32.0 | 22.0 | 10.0 | 18.3 | 10.6 | 4.3 |

tighter. For the previous KaHyPar versions, the number of imbalanced partitions is most pronounced on VLSI instances: For $\varepsilon = 0.01$, $k$KaHyPar and $r$KaHyPar compute infeasible solutions for 6.3% (10.3%) of the ISPD98 and for 9.5% (19.0%) of the DAC instances. Comparing the distribution of node weights reveals that these instances tend to have a larger proportion of *heavier* nodes compared to the ASM and SNAP instances (see Figure 9.2).

We also see that the number of imblanced partitions of both variants of Mt-KaHyPar is comparable to that of $k$KaHyPar, while both version of PaToH and hMetis-R produce a similar number of infeasible solutions than $r$KaHyPar with a notable exceptions: PaToH and hMetis-K compute significantly fewer feasible solutions on sparse matrix instances (ASM and SNAP) for $\varepsilon = 0.01$. Out of all partitioners, hMetis-K yields the most imbalanced instances across all instances types. As can be seen in Table 9.2, the number of imbalanced partitions produced by each competing partitioner increases with deceasing $\varepsilon$ and increasing $k$.

**Table 9.3:** Occurrence of prepacked nodes (i.e., nodes that are fixed to a specific block during partitioning) for each combination of $k$ and $\varepsilon$ when using $r$KaHyPar-BP and $k$KaHyPar-BP: Minimum/average/maximum percentage of prepacked nodes (left), and percentage of instances for which the prepacking is executed at least once (right).

| | | $\varepsilon = 0.01$ | | | $\varepsilon = 0.03$ | | | $\varepsilon = 0.1$ | | | Prepacking Triggered | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k$ | Min | Avg | Max | Min | Avg | Max | Min | Avg | Max | $\varepsilon = 0.01$ | $\varepsilon = 0.03$ | $\varepsilon = 0.1$ |
| $r$KaHyPar-BP | 2 | - | - | - | - | - | - | - | - | - | 0.0 | 0.0 | 0.0 |
| | 4 | - | - | - | - | - | - | - | - | - | 0.0 | 0.0 | 0.0 |
| | 8 | $\leq 0.1$ | $\leq 0.1$ | 0.2 | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | - | - | - | 5.0 | 1.7 | 0.0 |
| | 16 | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | 8.3 | 5.0 | 3.3 |
| | 32 | $\leq 0.1$ | 8.1 | 59.0 | $\leq 0.1$ | 6.2 | 68.4 | $\leq 0.1$ | 1.9 | 14.7 | 20.0 | 18.3 | 10.0 |
| | 64 | $\leq 0.1$ | 23.2 | 87.7 | $\leq 0.1$ | 17.3 | 90.9 | $\leq 0.1$ | 2.7 | 35.9 | 18.3 | 13.3 | 10.0 |
| | 128 | $\leq 0.1$ | 67.9 | 100.0 | $\leq 0.1$ | 42.0 | 96.3 | $\leq 0.1$ | 15.4 | 97.0 | 26.7 | 20.0 | 15.0 |
| $k$KaHyPar-BP | 2 | - | - | - | - | - | - | - | - | - | 0.0 | 0.0 | 0.0 |
| | 4 | - | - | - | - | - | - | - | - | - | 0.0 | 0.0 | 0.0 |
| | 8 | 6.7 | 17.1 | 41.6 | 0.4 | 0.5 | 0.6 | 2.3 | 2.3 | 2.3 | 5.0 | 3.3 | 1.7 |
| | 16 | 3.1 | 15.6 | 34.0 | 0.2 | 2.0 | 7.2 | 1.9 | 2.1 | 2.3 | 8.3 | 6.7 | 3.3 |
| | 32 | 0.3 | 29.9 | 56.0 | 0.1 | 11.7 | 42.3 | 0.2 | 3.4 | 26.3 | 13.3 | 15.0 | 6.7 |
| | 64 | 0.2 | 54.4 | 94.3 | 0.3 | 23.0 | 69.3 | 0.4 | 6.6 | 94.7 | 21.7 | 10.0 | 8.3 |
| | 128 | 0.5 | 76.5 | 100.0 | 0.4 | 42.4 | 91.0 | 0.3 | 15.7 | 59.8 | 28.3 | 21.7 | 11.7 |

Table 9.3 shows (i) how often our prepacking algorithm is triggered at least once in $r$KaHyPar-BP and $k$KaHyPar-BP (see Line 4 in Algorithm 9.1) and (ii) the percentage of nodes that are treated as fixed nodes. In $r$KaHyPar-BP, except for $k = 128$, on average less than 25% of the nodes are treated as fixed nodes (even less than 10% for $k < 64$), which provides sufficient flexibility to optimize the connectivity metric on the remaining ordinary nodes. However, in a few cases there are also runs where almost all nodes are added to the prepacking. As expected, the triggering frequency and the percentage of fixed nodes increases for larger values of $k$ and smaller $\varepsilon$. The percentage of prepacked nodes in $k$KaHyPar-BP is larger than in $r$KaHyPar-BP on average. Here, we use the prepacking algorithm in the initial partitioning phase where node weights tend to be larger than on the input hypergraph.

### 9.3.3 Solution Quality and Running Times

In Figure 9.3, we compare the solution quality of KaHyPar-BP to the different partitioners for $\varepsilon = 0.01$. The performance profiles look similar for $\varepsilon \in \{0.03, 0.1\}$ and are therefore omitted. Comparing the different KaHyPar configurations in Figure 9.3 (top-left), we can see that our new configurations provide the same solution quality as their non-prepacking counterparts. Furthermore, we see that, in general, the direct $k$-way algorithm still performs better than its RB counterpart (on par with Ref. [Sch20]). The other performance profiles in Figure 9.3 therefore compare the strongest configuration $k$KaHyPar-BP with Mt-KaHyPar (top-right), hMetis (bottom-left) and PaToH (bottom-right). We see that $k$KaHyPar-BP performs considerably better than the

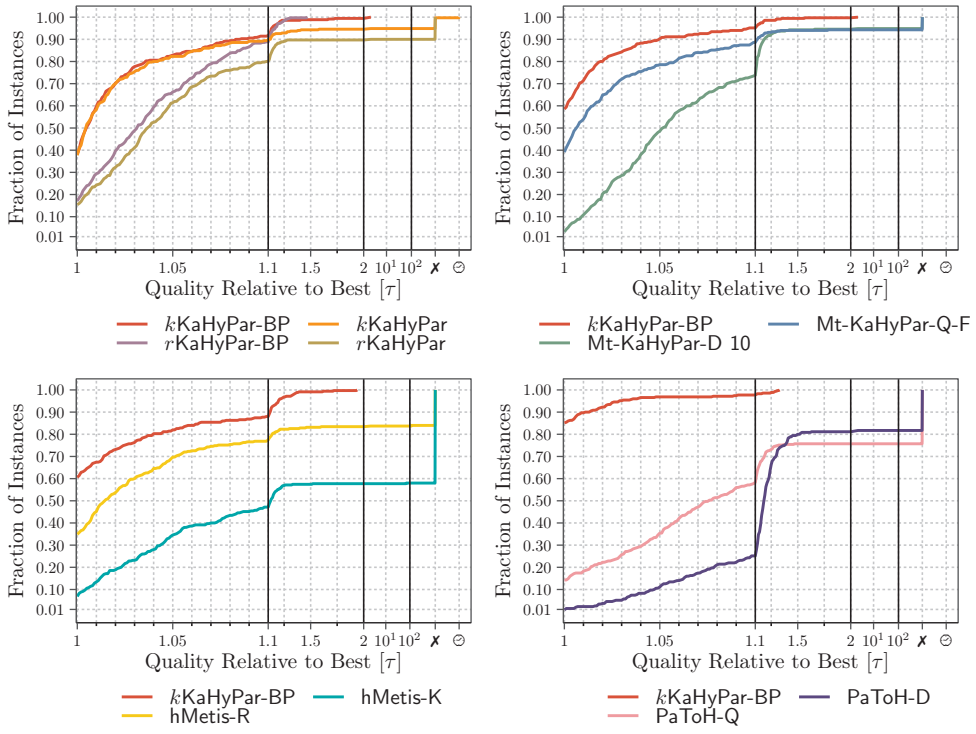



**Figure 9.3:** Performance profiles comparing the solution quality of KaHyPar-BP to KaHyPar (top-left), Mt-KaHyPar (top-right), hMetis (bottom-left), and PaToH (bottom-right) for $\varepsilon = 0.01$.

competitors with the notable exception of Mt-KaHyPar-Q-F that only produces slightly worse partitions than $k$KaHyPar-BP. For unweighted instances, Mt-KaHyPar-Q-F and $k$KaHyPar compute partitions with comparable solution quality (see Figure 8.9). We assume that the loss of quality is due to the rebalancing component that we run on the input hypergraph in Mt-KaHyPar when we detect that the final partition is imbalanced, which is more often triggered for weighted instances and smaller imbalance ratios. If we compare $k$KaHyPar-BP with each partitioner individually (for $\varepsilon = 0.01$), $k$KaHyPar-BP produces on a majority of the instances partitions with higher quality than $r$KaHyPar-BP (on 70.3% of the instances), $k$KaHyPar (54%), $r$KaHyPar (73.1%), Mt-KaHyPar-D (85.7%), Mt-KaHypar-Q-F (59.4%), hMetis-R (61.7%), hMetis-K (82%), PaToH-Q (85.7%) and PaToH-D (96.9%).

The running time plots in Figure 9.4 (top-left) show that our new approach does not impose any additional overheads in KaHyPar. For $\varepsilon = 0.01$ (and also for $\varepsilon \in \{0.03, 0.1\}$), $k$KaHyPar-BP (geometric mean running time is 42.89s) and $r$KaHyPar-BP (29.61s) are slightly faster than their counterparts $k$KaHyPar (46.91s) and $r$KaHyPar (30.09s) as our

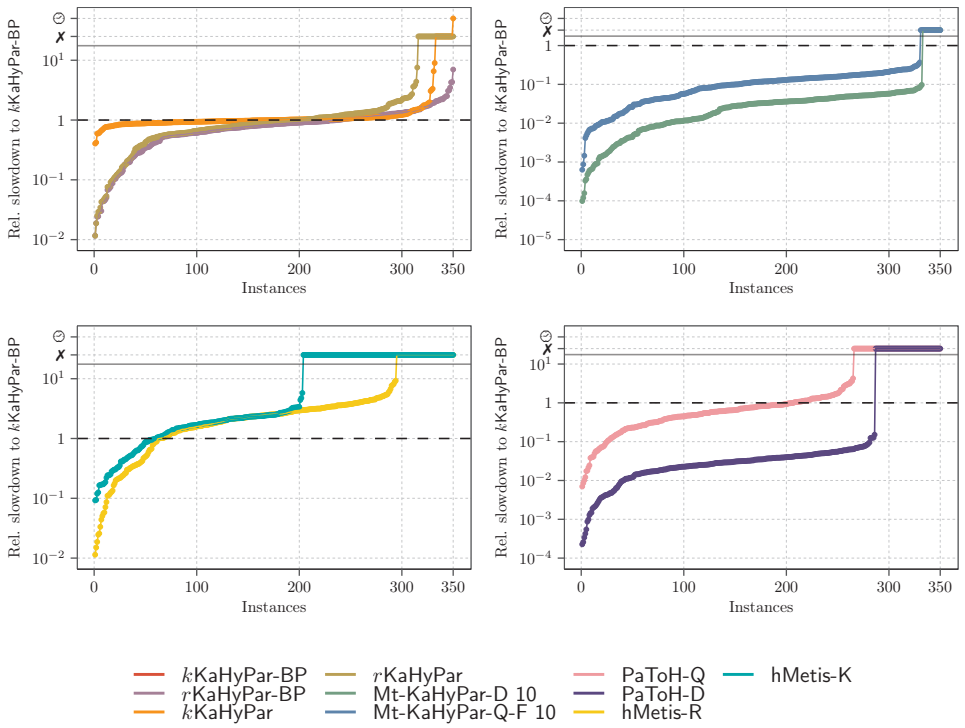



**Figure 9.4:** Running times of the different configurations of KaHyPar (left), hMetis (right), and PaToH (right) relative to $k$KaHyPar-BP for $\varepsilon = 0.01$.

new algorithm has replaced the previous balancing strategy in KaHyPar (restarting the bipartition with an tighter bound on the weight of the heaviest block if the bipartition is imbalanced). The running time difference is less pronounced for $r$KaHyPar-BP and $r$KaHyPar. This can be explained by the fact that, in $r$KaHyPar-BP, our prepacking algorithm is executed on the input hypergraph, whereas it is executed on the coarsest hypergraph in $k$KaHyPar-BP. The running time of all KaHyPar configurations increases with larger imbalance ratios ($k$KaHyPar: 56.69s for $\varepsilon = 0.03$ and 75.26s for $\varepsilon = 0.1$). This is expected since the running time of the flow-based refinement algorithm used in KaHyPar [HSS19a; Got+20] depends on $\varepsilon$. The running times of all other partitioners do not differ significantly with varying $\varepsilon$ except for Mt-KaHyPar-Q-F (3.25s for $\varepsilon = 0.01$, 3.88s for $\varepsilon = 0.03$, and 4.39s for $\varepsilon = 0.1$) that also uses flow-based refinement. For $\varepsilon = 0.01$, $k$KaHyPar-BP (42.89s) is faster than hMetis-R (62.07s) and hMetis-K (54.04s) but slower than PaToH-Q (18.65s), Mt-KaHyPar-Q-F (3.25s), PaToH-D (0.88s), and Mt-KaHyPar-D (0.75s).

## 9.4 Proof of Claim 9.6

**Lemma 9.7**

*Let $L = \langle a_1, \ldots, a_n \rangle$ be a sequence of elements sorted in decreasing weight order with respect to a weight function $c : L \to \mathbb{R}_{\geq 0}$ (for a subsequence $A := \langle a_1, \ldots, a_l \rangle$ of $L$, we define $c(A) := \sum_{i=1}^{l} c(a_i)$), $L'$ be an abitrary subsequence of $L$ sorted in decreasing weight order and $L_m = \langle a_1, \ldots, a_m \rangle$ the subsequence of the $m \leq n$ heaviest elements in $L$. Then the following conditions hold:*

*(i) If $c(L') \leq c(L_m)$, then $h_k(L') \leq h_k(L_m)$*

*(ii) If $c(L') > c(L_m)$, then $h_k(L') - \frac{1}{k}c(L') \leq h_k(L_m) - \frac{1}{k}c(L_m)$*

*Proof.* For convenience, we define $L' := \langle b_1, \ldots, b_l \rangle$. Note that $\forall i \in \{1, \ldots, \min(m, l)\} : c(a_i) \geq c(b_i)$, since $L_m$ contains the $m$ heaviest elements in decreasing order. We define $i := \arg\max_{i \in \{1,\ldots,l\}} c(b_i) + \frac{1}{k} \sum_{j=1}^{i-1} c(b_j)$ (index that maximizes $h_k(L')$).

(i) + (ii): If $i \leq m$, then

$$h_k(L') = c(b_i) + \frac{1}{k} \sum_{j=1}^{i-1} c(b_j) \overset{\substack{\forall j \in [1,i]: \\ c(b_j) \leq c(a_j)}}{\leq} c(a_i) + \frac{1}{k} \sum_{j=1}^{i-1} c(a_j) \leq h_k(L_m)$$

(i): If $m < i \leq l$, then

$$h_k(L') = c(b_i) + \frac{1}{k} \sum_{j=1}^{i-1} c(b_j) = c(b_i) - \frac{1}{k} \sum_{j=i}^{n} c(b_j) + \frac{1}{k} c(L') \leq \left(1 - \frac{1}{k}\right) c(b_i) + \frac{1}{k} c(L')$$
$$\overset{\substack{c(b_i) \leq c(a_m) \\ c(L') \leq c(L_m)}}{\leq} \left(1 - \frac{1}{k}\right) c(a_m) + \frac{1}{k} c(L_m) = c(a_m) + \frac{1}{k} \sum_{j=1}^{m-1} c(a_j) \leq h_k(L_m)$$

(ii): If $m < i \leq l$, then

$$h_k(L') - \frac{1}{k} c(L') = c(b_i) + \frac{1}{k} \sum_{j=1}^{i-1} c(b_j) - \frac{1}{k} c(L') = c(b_i) - \frac{1}{k} \sum_{l=i}^{n} c(b_l) \leq \left(1 - \frac{1}{k}\right) c(b_i)$$
$$\overset{c(b_i) \leq c(a_m)}{\leq} \left(1 - \frac{1}{k}\right) c(a_m) = c(a_m) + \frac{1}{k} \sum_{j=1}^{m-1} c(a_j) - \frac{1}{k} c(L_m) \leq h_k(L_m) - \frac{1}{k} c(L_m) \qquad \square$$

**Claim (9.6)**

*It holds that: $\frac{1}{k'} c(P_1) + h_{k'}(O_1) \leq \frac{1}{k'} \max(\Psi) + h_{k'}(O_t)$.*

*Proof.* Remember, $\Psi = \{P_1, P_2\}$, $\Pi_2 = \{V_1, V_2\}$ with $P_1 \subseteq V_1$ and $P_2 \subseteq V_2$, $O_1$ is equal to $V_1 \setminus P_1$ and $O_t$ represents the $t$ heaviest nodes of $(V_1 \cup V_2) \setminus (P_1 \cup P_2)$ with

$\max(\Psi) + c(O_t) \geq L_2$ as defined in Definition 9.4. The following proof distingush two cases based on Lemma 9.7.

If $c(O_1) \leq c(O_t)$, then

$$\frac{1}{k'}c(P_1) + h_{k'}(O_1) \overset{\text{Lemma 9.7(i)}}{\leq} \frac{1}{k'}c(P_1) + h_{k'}(O_t) \overset{c(P_1)\leq\max(\Psi)}{\leq} \frac{1}{k'}\max(\Psi) + h_{k'}(O_t)$$

If $c(O_1) > c(O_t)$, then

$$\begin{aligned}
\frac{1}{k'}c(P_1) + h_{k'}(O_1) &= \frac{1}{k'}c(P_1) + h_{k'}(O_1) - \frac{1}{k'}c(O_1) + \frac{1}{k'}c(O_1) \\
&\overset{\text{Lemma 9.7(ii)}}{\leq} \frac{1}{k'}(c(P_1) + c(O_1)) + h_{k'}(O_t) - \frac{1}{k'}c(O_t) \\
&\overset{c(P_1)+c(O_1)=c(V_1)}{=} \frac{1}{k'}(c(V_1) - c(O_t)) + h_{k'}(O_t) \\
\overset{c(V_1)\leq L_2}{\leq} \frac{1}{k'}(L_2 - c(O_t)) + h_{k'}(O_t) &\overset{\max(\Psi)+c(O_t)\geq L_2}{\leq} \frac{1}{k'}\max(\Psi) + h_{k'}(O_t) \qquad \square
\end{aligned}$$

Chapter 10

# 10 Conclusion

In this dissertation, we implemented the first shared-memory multilevel hypergraph partitioner that achieves the same solution quality as the highest-quality sequential algorithms while being an order of magnitude faster with ten threads. This was made feasible by carefully analyzing the core components of the best sequential systems and translating techniques essential for high solution quality into the parallel context. We engineered different scalable refinement algorithms and presented several novel concurrent gain computation techniques. We integrated the refinement algorithms into two parallel multilevel algorithms: a traditional multilevel partitioner that contracts clusters of nodes on each level and a novel parallelization of the $n$-level partitioning scheme that contracts only a single node on each level. We further presented several optimizations for graph partitioning that accelerated our overall partitioning algorithm by almost a factor of two for graphs. In a large experimental evaluation with 25 different sequential and parallel partitioning algorithms, we showed that our new shared-memory (hyper)graph partitioner Mt-KaHyPar outperforms most existing partitioning algorithms for optimizing the connectivity metric and can partition extremely large (hyper)graphs very fast with high solution quality.

We started this work with fundamental techniques to accurately compute gain values. This is especially challenging in the parallel setting as the gain of a node move can change between its initial calculation and actual execution due to concurrent node moves in its neighborhood. To this end, we introduced the *attributed gain value* technique that recomputes the gain of a node move based on synchronized data structure updates. We further presented a concurrent data structure for *caching* gain values that enables faster retrieval of the best possible move for a node. For maintaining the gain values in the setting where each node is moved at most once, we showed that our proposed gain table provides a better asymptotic worst-case complexity than the gain table used in KaHyPar [Akh+17a].

Based on this, we have implemented three parallel refinement algorithms: a label propagation refinement and highly-localized direct $k$-way FM algorithm, and a novel parallelization of flow-based refinement. In our multilevel algorithms, we first run the label propagation algorithm to find all *simple* node moves and then use the FM algorithm and flow-based refinement focussing on non-trivial improvements. Our highly-localized direct $k$-way FM algorithm improves an existing implementation used in the shared-memory graph partitioner Mt-KaHIP [ASS17] in several ways. In the previous algorithm of Mt-KaHIP, the threads perform node moves only locally and at the end of a refinement pass, the move sequences of the different threads are

concatenated to a global move sequence, for which gains are recomputed sequentially. In our implementation, the threads search for improvements in small non-overlapping regions of the (hyper)graph, and once they find an improvement, they immediately apply it to the global partition such that other threads can see these changes. We further implemented an algorithm to recompute the gain values of all node moves in the global move sequence in parallel.

We also presented a novel parallelization of flow-based refinement, which is considered the most powerful improvement heuristic for (hyper)graph partitioning at the moment. It parallelizes the approach used in the sequential hypergraph partitioner KaHyPar [HSS19a; Got+20]. The original algorithm works on pairs of blocks, for which we implemented an algorithm that schedules the bipartitioning routine on adjacent block pairs of the partition in parallel. We additionally integrated a parallel maximum flow algorithm to increase the scalability of flow-based refinement in situations where the number of adjacent block pairs does not suffice to achieve reasonable speedups. Furthermore, we proposed several optimizations that substantially accelerate the algorithm in practice.

We then integrated the refinement algorithms into a parallel multilevel partitioning algorithm. The coarsening algorithm finds and contracts clusters of nodes on each level in parallel, and is guided by a parallel community detection algorithm [HS17a]. For initial partitioning, we use parallel recursive bipartitioning with a work-stealing approach to account for load imbalances within the recursive partitioning calls. We further use a portfolio of different bipartitioning techniques to find an initial solution of the coarsest (hyper)graph [Heu15a; Sch+16a]. Each algorithm in this portfolio is executed several times independently in parallel, while we adjust the number of repetitions dynamically, meaning that algorithms that produce better bipartitions than others run more often. We then use the best bipartition out of all runs as initial solution. In the uncoarsening phase, we use our parallel refinement algorithms to improve the solution on each level. The overall parallel algorithm achieves good speedups with 24.7 for 64 threads on average (without flow-based refinement).

Moveover, we presented a novel parallelization of the $n$-level partitioning scheme used in the highest-quality sequential hypergraph partitioner KaHyPar [Sch20]. This scheme takes the multilevel paradigm to its most extreme version by only contracting a single node on each level. Correspondingly, in the uncoarsening phase, only a single node is uncontracted, allowing a highly-localized search for improvements. Although this approach seems inherently sequential, we showed that the $n$-level scheme can be parallelized efficiently under certain relaxations without comprises in solution quality. A key observation for our parallel implementation was that any valid sequence of contractions forms a forest (contracting a node $v$ onto another node $u$ induces an arc $(v, u)$ in the forest). We then presented a concurrent construction algorithm of the *contraction forest* that we used to derive a parallel schedule of contractions in the coarsening phase. These contractions are then applied to our new dynamic hypergraph data structure in parallel. In contrast to the hypergraph data structure used in KaHyPar, contracting two nodes does not require to allocate additional memory, which makes the approach applicable for very large hypergraphs. In the uncoarsening

phase, we assemble independent contractions in *batches*, each having roughly the same size. Each batch is uncontracted in parallel, and uncontracting a batch resolves the last dependencies required to uncontract the next batch. After uncontracting a batch, we run a highly-localized version of label propagation and FM refinement around the nodes contained in the batch. Our $n$-level partitioning algorithm achieves similar speedups to our multilevel partitioner (25.9 for 64 threads on average) and produces better partitions without flow-based refinement (median improvement is 2%). However, in the setting where two algorithms are given the same amount of time to find a solution or when flow-based refinement is used, our multilevel and $n$-level partitioner produces partition with comparable solution quality.

In contrast to the prevalent perception that hypergraph partitioning is "inherently more complicated" [Kay+12] than graph partitioning and thus more complex "in terms of implementation and running time" [Bul+16], we showed that the main differences between both are in the implementation of the (hyper)graph data structure and how gain values for node moves are computed. We then implemented simplified data structures for graph partitioning that we used as drop-in replacement in our partitioning algorithms. As a result, the optimizations accelerated our multilevel and $n$-level partitioning algorithms by almost a factor of two for graphs.

We also revisited the balanced hypergraph partitioning problem for weighted hypergraphs in the sequential setting. We showed that existing sequential partitioning algorithms considerably struggle to find balanced solution for weighted real-world instances. To this end, we presented a technique that enables partitioners based on recursive bipartitioning to reliably compute balanced partitions. The proposed method balances the partition by pre-assigning a small portion of the heaviest nodes to the two blocks of each bipartition and optimizes the objective function on the remaining nodes. We further established a balance property for the pre-assigned nodes that is verifiable in polynomial time and, if fulfilled, leads to provable balance guarantees for the resulting partition obtained via recursive bipartitioning. We integrated the balancing technique into the sequential hypergraph partitioner KaHyPar [Sch20]. In the experimental evaluation, we showed that our new approach computes balanced partitions on all tested instances without negatively affecting the solution quality and running time of KaHyPar.

In our experimental evaluation, we evaluated 25 different sequential and parallel (hyper)graph partitioning algorithms on over 800 graphs and hypergraphs. We compared our parallel multilevel partitioner without flow-based refinement (Mt-KaHyPar-D) to fast sequential and parallel partitioners. For comparison to the highest-quality sequential partitioner, we used our parallel $n$-level partitioner with flow-based refinement (Mt-KaHyPar-Q-F). Mt-KaHyPar-D was already able to outperform almost all existing partitioning algorithms in terms of solution quality and running time. On large hypergraphs, the partitions produced by Mt-KaHyPar-D are better than those of the state-of-the-art distributed-memory hypergraph partitioner Zoltan [Dev+06] by 23% in the median, while it is also faster by a factor of 2.72 on average. Out of all evaluated partitioners, only the shared-memory graph partitioner KaMinPar [Got+21e] was faster than Mt-KaHyPar-D, but the computed partitions of KaMinPar are 10%

worse in the median. Our highest-quality configuration Mt-KaHyPar-Q-F achieves the same solution quality as the best sequential hypergraph partitioner KaHyPar, while being an order of magnitude faster with ten threads. For graph partitioning, only KaFFPa-StrongS was able to produce better partitions than Mt-KaHyPar-Q-F (median improvement is 1%), but it is more than an order of magnitude slower.

The main contribution of this work is a partitioning algorithm that can compute high quality solutions extremely fast for very large graphs and hypergraphs. This opens up new opportunities for applications that previously relied on fast and medium-quality partitioning methods. For applications where quality is more important than speed, we provide a system that can cope with ever growing problem sizes, as Mt-KaHyPar has proven effective in partitioning (hyper)graphs with more than one billion edges/pins.

**Future Challenges.** In our experimental evaluation in Chapter 8, we briefly discussed what we consider a significant improvement in running time and solution quality. Our approach was to look at improvements that were considered as significant in the past, which were a few percent for solution quality and an order of magnitude for running times. A more natural way to assess improvements is to evaluate their impact for applications of hypergraph partitioning. However, due to the versatility of (hyper)graph partitioning, researchers often do not have the required domain-specific knowledge to setup appropriate benchmarks. Conversely, practitioners do not have a comprehensive overview of the (hyper)graph partitioner landscape and often stick to tools that worked well in the past [Cur+10; Kum+14] or develop their own solutions because existing algorithms do not support some constraints required by the application [ABM16; Kab+17]. A study evaluating the impact of different partitioning algorithms for several applications would provide valuable insights for both sides.

Although our shared-memory partitioner can partition very large (hyper)graphs, there exist instances that do not fit into the main memory of a single machine. Thus, future research should focus on translating the techniques presented in this work into the distributed-memory setting. A possible implementation of the multi-try $k$-way FM algorithm could compute small regions around a predefined set of seed nodes in a preprocessing step and distribute them to the PEs that then run highly-localized FM searches on them sequentially. The parallel scheduling scheme of our flow-based refinement algorithm can be integrated into distributed-memory partitioners similarly as already done for 2-way FM refinement [CP08; HSS10]. The PEs can collaborate in extracting a flow network, which is then copied to one PE performing flow-based refinement sequentially.

Since the number of available machines in data centers and high-performance computing clusters increases, partitioning (hyper)graphs into a large number of blocks (e.g., $k \geq 1000$) becomes increasingly important. However, since the complexity and memory overhead of some techniques used in Mt-KaHyPar depends on $k$ (e.g., the gain table stores $nk$ entries), it is currently not well-suited for the large $k$ setting. During our work on Mt-KaHyPar, we were also involved in developing the shared-memory graph partitioner KaMinPar [Got+21e], specifically tailored for partitioning graphs into a large number of blocks. In an ongoing bachelor thesis, we study the performance

of several advanced refinement techniques in this setting. Here, it turns out that FM refinement only leads to marginal improvements compared to label propagation refinement, while flow-based refinement can still improve the solution quality by a few percentages. However, the improvements are not as pronounced as for smaller values of $k$. Thus, further research is required to understand why these techniques are less effective in this case.

Our experimental results suggest that traditional multilevel partitioners can achieve the same solution quality as $n$-level partitioners when both are given the same amount of time to compute a solution or when flow-based refinement is used. We have presented a simple optimization improving the running time of the overall $n$-level partitioning algorithm by almost 60% by always selecting the node with the lower degree as the contraction partner. However, this also negatively affected the solution quality, which we could not explain. Moreover, a version of flow-based refinement running on each level might be attractive to further improve the solution quality (currently, it only runs on an approximately logarithmic number of levels). A concrete implementation could construct small flow problems around the nodes contained in each batch and then solve them in parallel. An alternative could be to perform the uncontraction directly on the flow network and use the flow from the previous level to augment it again to a maximum flow on the current level.

We found some instances where we could show that multilevel partitioning algorithms produce partitions of poor quality (not published). Here, we compared several existing multilevel partitioners to a simple heuristic: sort the nodes in increasing order of their node degrees and then cut the sorted order into two equally-sized blocks to obtain a bipartition. Surprisingly, this algorithm produced bipartitions with better edge cuts than all evaluated multilevel algorithms by more than a factor of two for some graphs (we found ten instances in set $\mathrm{M_G}$). These graphs have the property that they consist of a small and dense core of highly connected nodes, while other nodes are only loosely connected to them. A good bipartition places the dense core (or the high-degree nodes) into one block. In multilevel algorithms, the low-degree nodes are contracted onto the high-degree nodes. This increases their node weight, forcing initial partitioning to cut the dense core due to the balance constraint. As one might assume that these are just some obscure instances not relevant in practice, we found that the `Twitter` graph ($\approx 1.2$ billion edges) actually suffers from this problem. In an ongoing master thesis, we currently work on adapting coarsening and initial partitioning to deal with such graphs. The idea is to remove or ignore low-degree nodes in the coarsening phase and then use a specialized packing algorithm to assign them to the blocks after initial partitioning.

# Appendix

# List of Algorithms


**4 Parallel Improvement Algorithms**

4.1 Moves a node $u$ from block $V_i$ to $V_j$ . . . 62
4.2 Computes the highest gain move for a node $u$ . . . 64
4.3 Gain Table Update . . . 65
4.4 Parallel Gain Recalculation . . . 68
4.5 The Parallel Label Propagation Algorithm . . . 69
4.6 Multi-Try $k$-Way FM Algorithm . . . 71
4.7 Highly-Localized $k$-Way FM Algorithm . . . 72
4.8 Parallel Flow-Based Refinement . . . 75
4.9 The FlowCutter Algorithm . . . 79

**5 Parallel Multilevel Hypergraph Partitioning**

5.1 Parallel Multilevel Hypergraph Partitioning . . . 86
5.2 Clustering Algorithm . . . 90

**6 Parallel $n$-level Hypergraph Partitioning**

6.1 Parallel $n$-level Hypergraph Partitioning . . . 122
6.2 Contraction Operation . . . 126
6.3 Uncontraction Operation . . . 127
6.4 Uncontraction Gain Table Update . . . 133

**7 From Hypergraphs to Graphs**

7.1 Moving a node $u$ from block $V_i$ to $V_j$ . . . 147

**9 Multilevel Hypergraph Partitioning with Node Weights**

9.1 Recursive Bipartitioning Algorithm . . . 183
9.2 Prepacking Algorithm . . . 186

# List of Figures


**1 Introduction**

1.1 The graph and hypergraph model of the relationships between a set of authors and their publications. . . . . 2

**2 Preliminaries**

2.1 Example of a 4-way partition of a hypergraph. . . . . 10
2.2 The bipartite graph representation and Lawler expansion. . . . . 12
2.3 The RAM and PRAM model. . . . . 17
2.4 The TBB task scheduler. . . . . 19
2.5 Speedups of different parallel algorithms used in this work. . . . . 20
2.6 Basic properties of the graphs and hypergraphs in our benchmark sets. 22
2.7 Example of a performance profile plot and effectiveness tests. . . . . 26
2.8 Example of a relative running time and speedup plot. . . . . 27

**3 Related Work**

3.1 The bucked priority queue used in the FM algorithm. . . . . 33
3.2 Illustration of a situation where the FM algorithm makes a suboptimal decision. . . . . 36
3.3 Example of one iteration of the FBB algorithm. . . . . 38
3.4 The multilevel scheme. . . . . 40
3.5 The dynamic hypergraph data structure used in KaHyPar. . . . . 42
3.6 Performance profiles comparing KaFFPa-StrongS to Metis-K, and $k$KaHyPar to PaToH-D. . . . . 48
3.7 Different types of concurrent move conflicts. . . . . 50
3.8 The deep multilevel scheme. . . . . 56
3.9 Performance profiles comparing KaFFPa-StrongS to Mt-KaHIP, and $k$KaHyPar to Zoltan. . . . . 58

**4 Parallel Improvement Algorithms**

4.1 The partition data structure. . . . . 61
4.2 Example of our parallel scheduling scheme for flow computations on adjacent block pairs. . . . . 76
4.3 Illustration of the flow network construction algorithm. . . . . 78
4.4 Example of a push-relabel conflict in the parallel discharge routine. . . 81

**5 Parallel Multilevel Hypergraph Partitioning**

5.1 The hypergraph data structure used in our multilevel hypergraph partitioner. . . . 88
5.2 The different types of conflicts occuring in our parallel clustering algorithm and their resolutions. . . . 89
5.3 Effects of the maximum number of runs per flat initial bipartitioning algorithm on the solution quality. . . . 101
5.4 Performance profiles comparing different refinement configurations in the initial partitioning phase. . . . 101
5.5 Performance profiles comparing the effects of repeated executions of the label propagation and FM algorithm on the solution quality. . . . 102
5.6 Effects of the flow region scaling factor on the solution quality. . . . 102
5.7 Effects of restricting the distance of each node to the cut hyperedges in our flow-based refinement algorithm on the solution quality. . . . 103
5.8 Frequency of search conflicts in our different parallel refinement algorithms. . . . 105
5.9 Performance profile comparing the solution quality and running times of Mt-KaHyPar-D with and without thread-local partitions. . . . 107
5.10 Frequency of search conflicts in the flow-based refinement algorithm for different values of $\tau$. . . . 108
5.11 Performance profile comparing the solution quality and running times of Mt-KaHyPar-D-F for different values $\tau$. . . . 108
5.12 Component effectiveness tests for our multilevel partitioner. . . . 110
5.13 Performance profiles and running times comparing Mt-KaHyPar-D and Mt-KaHyPar-D-F on set $M_{HG}$. . . . 111
5.14 Performance profiles and running times comparing Mt-KaHyPar-D and Mt-KaHyPar-D-F on set $L_{HG}$. . . . 111
5.15 Performance profiles comparing the solution quality of different configurations with (+) and without (-) community-aware coarsening (CD), flow-based refinement (F), and with one (+1V) and three V-cycles (+3V). 112
5.16 Speedups of Mt-KaHyPar-D and its different algorithmic components. . 114
5.17 Performance profiles comparing the solution quality of Mt-KaHyPar-D with an increasing number of threads on set $L_{HG}$. . . . 115
5.18 Speedups of Mt-KaHyPar-D-F and the flow-based refinement routine for different values of $k$. . . . 116
5.19 Performance profiles comparing the solution quality of Mt-KaHyPar-D-F with increasing number of threads on set $L_{HG}$. . . . 117
5.20 Running time shares of different components on the total execution time of Mt-KaHyPar-D and Mt-KaHyPar-D-F. . . . 118
5.21 Running time shares of different components on the total execution time of the flow-based refinement routine. . . . 119

**6 Parallel $n$-level Hypergraph Partitioning**

6.1 Contraction and uncontraction operations applied on the dynamic hypergraph data structure. . . . 124
6.2 Example of an uncontraction that increases the cut size. . . . 130
6.3 Performance profiles and running times comparing Mt-KaHyPar-Q with different batch size values $b_{\max}$. . . . 135
6.4 Speedups of Mt-KaHyPar-Q and its different algorithmic components. . 136
6.5 Performance profiles comparing the solution quality of Mt-KaHyPar-Q with an increasing number of threads on set $\mathrm{L_{HG}}$. . . . 137
6.6 Running time shares of different components on the total execution time of Mt-KaHyPar-Q. . . . 138
6.7 Slowdowns of our new dynamic hypergraph data structure compared to KaHyPar's hypergraph data structure. . . . 139
6.8 Performance profiles comparing Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F) on set $\mathrm{M_{HG}}$. . . . 141
6.9 Performance profiles comparing Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F) on set $\mathrm{L_{HG}}$. . . . 142
6.10 Running times of Mt-KaHyPar-D-F and Mt-KaHyPar-Q(-F) relative to Mt-KaHyPar-D on set $\mathrm{M_{HG}}$ and set $\mathrm{L_{HG}}$. . . . 142
6.11 Effectiveness tests comparing Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F)) on set $\mathrm{M_{HG}}$. . . . 143
6.12 Effects of always choosing the node with larger degree as the representative of a contraction on the solution quality and running time of Mt-KaHyPar-Q. . . . 143

**7 From Hypergraphs to Graphs**

7.1 The graph data structure used in our multilevel graph partitioner. . . . 148
7.2 Contraction and uncontraction operations applied on the dynamic graph data structure. . . . 152
7.3 Performance profiles comparing the solution quality Mt-KaHyPar-D/-Q with and without our new graph data structure. . . . 155
7.4 Running time improvements of different algorithmic components for Mt-KaHyPar-D/-Q when we use our new graph data structure. . . . 156

**8 A Comparison of Partitioning Algorithms**

8.1 Performance profile comparing Mt-KaHyPar to different sequential partitioning algorithms on set $\mathrm{M_{HG}}$ and on the same benchmark set where we excluded the instances of our parameter tuning benchmark set $\mathrm{M_P}$. 160
8.2 Performance profile and running times comparing PaToH-D and Mondriaan on set $\mathrm{M_{HG}}$. . . . 162
8.3 Performance profile and running times comparing different configurations of KaHyPar and hMetis on set $\mathrm{M_{HG}}$. . . . 162
8.4 Performance profile and running times comparing Zoltan and BiPart on set $\mathrm{L_{HG}}$. . . . 163

8.5 Performance profile and running times comparing Metis-K to Metis-R, KaFFPa and Scotch on set $M_G$. . . . . 164
8.6 Performance profile and running times comparing the social configurations of KaFFPa to their non-social counterparts on set $M_G$. . . . . 165
8.7 Performance profile and running times comparing different parallel graph partitioners on set $L_G$. . . . . 166
8.8 Performance profile and running times comparing Mt-KaHyPar-D and PaToH on set $M_{HG}$. . . . . 168
8.9 Performance profile and running times comparing Mt-KaHyPar-Q-F and KaHyPar on set $M_{HG}$. . . . . 168
8.10 Performance profile and running times comparing Mt-KaHyPar-D to PaToH-D and Zoltan on set $L_{HG}$. . . . . 169
8.11 Performance profile and running times comparing Mt-KaHyPar-D and Metis-K on set $M_G$. . . . . 170
8.12 Performance profile and running times comparing Mt-KaHyPar-Q-F and KaFFPa on set $M_G$. . . . . 170
8.13 Performance profile and running times comparing Mt-KaHyPar-D to KaMinPar and Mt-KaHIP on set $L_G$. . . . . 171
8.14 Performance profile comparing Mt-KaHyPar-Q-F and KaSPar strong on the benchmark set from Ref. [OS10]. . . . . 172
8.15 Summary of the experimental results. . . . . 173

**9 Multilevel Hypergraph Partitioning with Node Weights**

9.1 Example of a deeply and non-deeply balanced bipartition. . . . . 181
9.2 Distributions of node weights for different instance types in our benchmark sets. . . . . 187
9.3 Performance profiles comparing the solution quality of KaHyPar-BP to KaHyPar, Mt-KaHyPar, hMetis, and PaToH for $\varepsilon = 0.01$. . . . . 191
9.4 Running times of the different configurations of KaHyPar, hMetis, and PaToH relative to $k$KaHyPar-BP for $\varepsilon = 0.01$. . . . . 192

# List of Tables


**2 Preliminaries**

2.1 The number of hypergraphs included from the different sources and the largest instance of each benchmark set. . . . . 23
2.2 The number of graphs included from the different sources and the largest instance of each benchmark set. . . . . 23
2.3 Default partitioning setup for experiments. . . . . 25

**3 Related Work**

3.1 Algorithmic components of existing sequential partitioning algorithms. 44
3.2 Listing of different rating functions used in coarsening algorithms. . . 45
3.3 Algorithmic components of existing parallel partitioning algorithms. . 53

**5 Parallel Multilevel Hypergraph Partitioning**

5.1 Algorithm configuration of Mt-KaHyPar-D(-F). . . . . 98
5.2 Average percentages of different search conflicts occuring in our parallel refinement algorithms. . . . . 106
5.3 Geometric mean speedups for different algorithmic components of Mt-KaHyPar-D(-F). . . . . 113

**6 Parallel $n$-level Hypergraph Partitioning**

6.1 Geometric mean speedups for different algorithmic components of Mt-KaHyPar-Q. . . . . 137

**8 A Comparison of Partitioning Algorithms**

8.1 Listing of partitioning algorithms included in the experimental evaluation. 158

**9 Multilevel Hypergraph Partitioning with Node Weights**

9.1 Percentage of imbalanced instances produced by each partitioner for each combination of instance type and $\varepsilon$. . . . . 189
9.2 Percentage of imbalanced instances produced by each partitioner for each combination of $k$ and $\varepsilon$. . . . . 189
9.3 Occurrence of prepacked nodes for each combination of $k$ and $\varepsilon$ when using $r$KaHyPar-BP and $k$KaHyPar-BP. . . . . 190

# Bibliography


[Aad+15] Georges Aad et al. „Combined Measurement of the Higgs Boson Mass in *pp* Collisions at $\sqrt{s} = 7$ and 8 TeV with the ATLAS and CMS Experiments“. In: *Physical Review Letters* 114.19 (2015). [see page 1]

[AB12] Bas Fagginger Auer and Rob H. Bisseling. „Abusing a Hypergraph Partitioner for Unweighted Graph Partitioning“. In: *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*. Volume 588. Contemporary Mathematics. American Mathematical Society, 2012, pages 19–36. [see page 145]

[Aba+02] Cristinel Ababei, Navaratnasothie Selvakkumaran, Kia Bazargan, and George Karypis. „Multi-Objective Circuit Partitioning for Cutsize and Path-Based Delay Minimization“. In: *International Conference on Computer-Aided Design (ICCAD)*. 2002, pages 181–185. DOI: 10.1145/774572.774599. [see page 14]

[ABM16] Kevin Aydin, Mohammad H. Bateni, and Vahab S. Mirrokni. „Distributed Balanced Partitioning via Linear Embedding“. In: *9th ACM International Conference on Web Search and Data Mining (WSDM)*. ACM, 2016, pages 387–396. DOI: 10.1145/2835776.2835829. [see pages 29, 49, 198]

[ACU08a] Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. „Multi-level Direct *k*-Way Hypergraph Partitioning With Multiple Constraints and Fixed Vertices“. In: *Journal of Parallel Distributed Computing* 68.5 (2008), pages 609–625. DOI: 10.1016/j.jpdc.2007.09.006. [see page 87]

[ACU08b] Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora 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. DOI: 10.1016/j.jpdc.2007.09.006. [see pages 14, 15, 41, 44–46, 78, 85, 180]

[AH19] Pablo Andres-Martinez and Chris Heunen. „Automated Distribution of Quantum Circuits via Hypergraph Partitioning“. In: *Physical Review A* 100.3 (2019), pages 1–11. [see page 2]

[AHK97] Charles J. Alpert, Jsen-Hsin Huang, and Andrew B. Kahng. „Multilevel Circuit Partitioning“. In: *34th Conference on Design Automation (DAC)*. June 1997, pages 530–533. DOI: 10.1145/266021.266275. [see pages 2, 39, 41, 121]

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. ISBN: 0-201-00029-6. [see page 31]

[AK06] Amine Abou-Rjeili and George Karypis. „Multilevel Algorithms for Partitioning Power-Law Graphs“. In: *20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006. DOI: 10.1109/IPDPS.2006.1639360. [see pages 43–45, 85]

[AK95] Charles J. Alpert and Andrew B. Kahng. „Recent Directions in Netlist Partitioning: A Survey“. In: *Integration: The VLSI Journal* 19.1-2 (1995), pages 1–81. DOI: 10.1016/0167-9260(95)00008-4. [see pages 2, 14, 29, 30, 39, 177]

[Akh+17a] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. „Engineering a Direct $k$-way Hypergraph Partitioning Algorithm“. In: *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2017, pages 28–42. DOI: 10.1137/1.9781611974768.3. [see pages 4, 6, 9, 41, 42, 44, 45, 47, 59, 64, 71, 121, 139, 159, 180, 187, 195]

[Akh19] Yaroslav Akhremtsev. „Parallel and External High Quality Graph Partitioning“. Dissertation. Karlsruhe Institute of Technology, 2019. [see pages 24, 52, 94, 159]

[AL08] Reid Andersen. and Kevin J. Lang. „An Algorithm for Improving Graph Partitions“. In: *Proceedings of the 19th SIAM Journal on Discrete Mathematics*. SIAM. 2008, pages 651–660. DOI: 10.5555/1347082.1347154. [see page 38]

[Alp98] Charles J. Alpert. „The ISPD98 Circuit Benchmark Suite“. In: *International Symposium on Physical Design (ISPD)*. Apr. 1998, pages 80–85. DOI: 10.1145/274535.274546. [see pages 21, 23, 177, 186]

[AMS04] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. „MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation“. In: *The International Journal of Universal Computer Science* 10.12 (2004), pages 1562–1596. DOI: 10.3217/jucs-010-12-1562. [see page 2]

[Ang+19] Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. „Guidelines for Experimental Algorithmics: A Case Study in Network Analysis“. In: *Algorithms* 12.7 (2019), page 127. DOI: 10.3390/a12070127. [see page 28]

[AR04] Konstantin Andreev and Harald Räcke. „Balanced Graph Partitioning“. In: *16th Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, June 2004, pages 120–124. DOI: 10.1145/1007912.1007931. [see page 16]

[Arm+10] Ed Armstrong, Gary William Grewal, Shawki Areibi, and Gerarda A. Darlington. „An Investigation of Parallel Memetic Algorithms for VLSI Circuit Partitioning on Multi-Core Computers“. In: *Proceedings of the 23rd Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2010, pages 1–6. DOI: 10.1109/CCECE.2010.5575207. [see page 29]

[AS95] Richard J. Anderson and João C. Setubal. „A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem“. In: *Journal of Parallel and Distributed Computing* 29.1 (1995), pages 17–26. DOI: 10.1006/jpdc.1995.1103. [see page 80]

[ASS17] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. „High-Quality Shared-Memory Graph Partitioning“. In: *European Conference on Parallel Processing (Euro-Par)*. Springer, Aug. 2017, pages 659–671. DOI: 10.1007/978-3-319-96983-1_47.
[see pages 3, 5, 26, 51, 52, 54–59, 68, 70, 73, 94, 149, 159, 195]

[ASS18] Robin Andre, Sebastian Schlag, and Christian Schulz. „Memetic Multilevel Hypergraph Partitioning“. In: *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, July 2018, pages 347–354. DOI: 10.1145/3205455.3205475. [see pages 22, 29, 43, 158]

[AV93] Shawki Areibi and Anthony Vannelli. „Circuit Partitioning Using a Tabu Search Approach“. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1993, pages 1643–1646. [see page 158]

[Axt+22] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. „Engineering In-place (Shared-memory) Sorting Algorithms“. In: *ACM Transaction on Parallel Computing* 9.1 (2022), 2:1–2:62. DOI: 10.1145/3505286. [see page 19]

[AY04] Shawki Areibi and Zhen Yang. „Effective Memetic Algorithms for VLSI Design = Genetic Algorithms + Local Search + Multi-Level Clustering“. In: *Evolutionary Computation* 12.3 (2004), pages 327–353. DOI: 10.1162/1063656041774947. [see page 29]

[Bad+13] David Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*. Volume 588. Contemporary Mathematics. American Mathematical Society, Feb. 2013. [see page 23]

[Bar82] Earl Barnes. „An Algorithm for Partitioning the Nodes of a Graph“. In: *SIAM Journal on Algebraic Discrete Methods* 3.4 (1982), pages 541–550. [see page 29]

[BB87] Marsha J. Berger and Shahid H. Bokhari. „A Partitioning Strategy for Nonuniform Problems on Multiprocessors“. In: *IEEE Transactions on Computers* 100.5 (1987), pages 570–580. [see page 29]

[BBS15] Niklas Baumstark, Guy E. Blelloch, and Julian Shun. „Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm“. In: *23rd European Symposium on Algorithms (ESA)*. Volume 9294. Springer, 2015, pages 106–117. DOI: 10.1007/978-3-662-48350-3_10. [see pages 80–82, 116]

[BC09] Michael J Barber and John W Clark. „Detecting Network Communities by Propagating Labels Under Constraints“. In: *Physical Review E* 80.2 (2009). [see page 41]

[BD19] Daniel Berrar and Werner 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 28]

[Bel+14] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. *The SAT Competition 2014*. http://www.satcompetition.org/2014/. 2014. [see pages 21–23, 117]

[BF00] Michael A. Bender and Martin Farach-Colton. „The LCA Problem Revisited“. In: *Latin American Symposium on Theoretical Informatics*. Springer. 2000, pages 88–94. [see page 185]

[BH11a] Una Benlic and Jin-Kao Hao. „A Multilevel Memetic Approach for Improving Graph $k$-Partitions“. In: *IEEE Transactions on Evolutionary Computation* 15.5 (2011), pages 624–642. [see page 29]

[BH11b] Una Benlic and Jin-Kao Hao. „An Effective Multilevel Tabu Search Approach for Balanced Graph Partitioning“. In: *Computers & Operation Research* 38.7 (2011), pages 1066–1075. DOI: 10.1016/j.cor.2010.10.007. [see page 158]

[Bin18] Timo Bingmann. „Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools“. PhD thesis. Karlsruhe Institute of Technology, 2018. [see pages 3, 16]

[Bis+12] Rob H. Bisseling, Bas O. Auer Fagginger, A. N. Yzelman, Tristan van Leeuwen, and Ümit V. Çatalyürek. „Two-Dimensional Approaches to Sparse Matrix Partitioning“. In: *Combinatorial Scientific Computing* (2012), pages 321–349. [see page 177]

[BJ93] Thang N. Bui and Curt 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. URL: https://www.osti.gov/biblio/54439. [see page 40]

[Blo+08] Vincent D. Blondel, Jean Guillaume, Renaud Lambiotte, and Etienne Lefebvre. „Fast Unfolding of Communities in Large Networks“. In: *Journal of Statistical Mechanics: Theory and Experiment* 10 (2008). [see pages 45, 92, 93]

[BLV14] Florian Bourse, Marc Lelarge, and Milan Vojnovic. „Balanced Graph Edge Partition“. In: *20th International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2014, pages 1456–1465. DOI: 10.1145/2623330.2623660. [see page 14]

[BM94] Thang Nguyen Bui and Byung Ro Moon. „A Fast and Stable Hybrid Genetic Algorithm for the Ratio-Cut Partitioning Problem on Hypergraphs“. In: *31st Conference on Design Automation (DAC)*. IEEE, 1994, pages 664–669. DOI: 10.1145/196244.196607. [see page 29]

[Bon+06] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. „An Improved Construction for Counting Bloom Filters“. In: *14th European Symposium on Algorithms (ESA)*. Volume 4168. Springer, 2006, pages 684–695. DOI: 10.1007/11841036_61. [see page 88]

[Bra+08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. „On Modularity Clustering“. In: *IEEE Transactions on Knowledge and Data Engineering* 20.2 (2008), pages 172–188. DOI: 10.1109/TKDE.2007.190689. [see page 92]

[BS11] Charles-Edmond Bichot and Patrick Siarry. *Graph Partitioning*. John Wiley & Sons, 2011. [see page 29]

[BS93] Stephen T. Barnard and Horst 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 3, 29, 40, 158]

[Bui+87] Thang Nguyen Bui, Soma Chaudhuri, Frank Thomson Leighton, and Michael Sipser. „Graph Bisection Algorithms with Good Average Case Behavior“. In: *Combinatorica* 7.2 (1987), pages 171–191. DOI: 10.1007/BF02579448. [see page 39]

[Bui+89] Thang Nguyen Bui, C. Heigham, Curt Jones, and Frank Thomson 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. DOI: 10.1145/74382.74527. [see page 39]

[Bul+16] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. „Recent Advances in Graph Partitioning“. In: *Algorithm Engineering - Selected Results and Surveys*. Volume 9220. 2016, pages 117–158. DOI: 10.1007/978-3-319-49487-6_4. [see pages 14, 29, 145, 197]

[ÇA01a] Ümit V. Çatalyürek and Cevdet Aykanat. „A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices“. In: *15th International Parallel and Distributed Processing Symposium (IPDPS)*. 2001, page 118. DOI: 10.1109/IPDPS.2001.925093. [see page 2]

[ÇA01b] Ümit V. Çatalyürek and Cevdet Aykanat. „A Hypergraph-Partitioning Approach for Coarse-Grain Decomposition“. In: *ACM/IEEE Conference on Supercomputing*. ACM, 2001, page 28. DOI: 10.1145/582034.582062. [see page 2]

[ÇA11] Ümit V. Çatalyürek and Cevdet Aykanat. *PaToH: Partitioning Tool for Hypergraphs*. 2011. [see pages 46, 95, 96]

[ÇA96] Ümit V Çatalyürek and Cevdet Aykanat. „Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication“. In: *International Workshop on Parallel Algorithms for Irregularly Structured Problems*. Springer. 1996, pages 75–86. DOI: 10.1007/BFb0030098. [see page 177]

[CA99] Ümit V. Catalyurek 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. DOI: 10.1109/71.780863. [see pages 2, 7, 13, 21, 39, 44, 46, 85, 95, 145, 149, 159, 161, 177, 180, 187]

[Çat+12a] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. „Multithreaded Clustering for Multi-level Hypergraph Partitioning“. In: *26th International Parallel and Distributed Processing Symposium (IPDPS)*. 2012, pages 848–859. DOI: 10.1109/IPDPS.2012.81. [see pages 52, 53, 85, 88]

[Çat+12b] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora 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. URL: http://www.ams.org/books/conm/588/11704. [see page 14]

[Çat+22a] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. „More Recent Advances in (Hyper)Graph Partitioning“. In: *Computing Research Repository (CoRR)* abs/2205.13202 (2022). arXiv: 2205.13202. [see pages 9, 29, 30]

[CG12] K Santle Camilus and VK Govindan. „A Review on Graph Based Segmentation“. In: *International Journal of Image, Graphics and Signal Processing* 4.5 (2012), page 1. [see page 1]

[CG97] Boris V. Cherkassky and Andrew V. Goldberg. „On Implementing the Push-Relabel Method for the Maximum Flow Problem“. In: *Algorithmica* 19.4 (1997), pages 390–410. DOI: 10.1007/PL00009180. [see pages 81, 82]

[CKM00a] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. „Improved Algorithms for Hypergraph Bipartitioning“. In: *Asia South Pacific Design Automation Conference (ASP-DAC)*. 2000, pages 661–666. DOI: 10.1145/368434.368864. [see pages 141, 157, 178]

[CKM00b] Andrew. E. Caldwell, Andrew B. Kahng, and Igor 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. DOI: 10.1109/43.892854. [see pages 178, 188]

[CL20] Chandra Chekuri and Shi Li. „On the Hardness of Approximating the $k$-Way Hypergraph Cut Problem“. In: *Theory of Computing* 16 (2020), pages 1–8. DOI: 10.4086/toc.2020.v016a014. [see page 15]

[CL98] Jason Cong and Sung Kyu Lim. „Multiway Partitioning with Pairwise Movement“. In: *International Conference on Computer-Aided Design (ICCAD)*. 1998, pages 512–516. DOI: 10.1145/288548.289079. [see page 34]

[CP08] Cédric Chevalier and François Pellegrini. „PT-Scotch: A Tool for Efficient Parallel Graph Ordering“. In: *Parallel Computing* 34.6-8 (2008), pages 318–331. DOI: 10.1016/j.parco.2007.12.001. [see pages 52, 54–56, 94, 158, 198]

[CRX03] Jason Cong, Michail Romesis, and Min Xie. „Optimality, Scalability and Stability Study of Partitioning and Placement Algorithms“. In: *International Symposium on Physical Design (ISPD)*. Apr. 2003, pages 88–94. DOI: 10.1145/640000.640021. [see page 157]

[CS09] Michael J. Campbell and Thomas D.V. Swinscow. *Statistics at Square One*. BMJ Publishing Group, 2009. [see page 28]

[CS93] Jason Cong and M'Lissa 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. DOI: 10.1145/157485.165119. [see pages 39, 40, 46]

[CSZ93] Pak K. Chan, Martine D. F. Schlag, and Jason Y. Zien. „Spectral $k$-Way Ratio-Cut Partitioning and Clustering“. In: *30th Conference on Design Automation (DAC)*. June 1993, pages 749–754. DOI: 10.1109/43.310898. [see page 14]

[Cur+10] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. „Schism: A Workload-Driven Approach to Database Replication and Partitioning“. In: *Proceedings of the VLDB Endowment* 3.1 (2010), pages 48–57. DOI: 10.14778/1920841.1920853. [see pages 2, 198]

[DDN20] Timothy A. Davis, Iain S. Duff, and Stojce Nakov. „Design and Implementation of a Parallel Markowitz Threshold Algorithm“. In: *SIAM Journal on Matrix Analysis and Applications* 41.2 (Apr. 2020), pages 573–590. DOI: 10.1137/19M1245815. [see page 186]

[Del+11] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. „Graph Partitioning with Natural Cuts“. In: *25th International Parallel and Distributed Processing Symposium (IPDPS)*. 2011, pages 1135–1146. DOI: 10.1109/IPDPS.2011.108. [see pages 1, 38, 172]

[Dev+06] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Çatalyürek. „Parallel Hypergraph Partitioning for Scientific Computing“. In: *20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006. DOI: 10.1109/IPDPS.2006.1639359. [see pages 7, 51, 52, 54, 55, 58, 94, 159, 180, 197]

[Dev+15] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit V. Çatalyürek. „Hypergraph Partitioning for Multiple Communication Cost Metrics: Model and Methods“. In: *Journal of Parallel and Distributed Computing* 77 (2015), pages 69–83. DOI: 10.1016/j.jpdc.2014.12.002. [see page 14]

[Dev+16] Mehmet Deveci, Sivasankaran Rajamanickam, Karen D. Devine, and Ümit V. Çatalyürek. „Multi-Jagged: A Scalable Parallel Spatial Partitioning Algorithm“. In: *IEEE Transactions on Parallel and Distributed Systems* 27.3 (2016), pages 803–817. DOI: 10.1109/TPDS.2015.2412545. [see page 29]

[DH03] Doratha E. Drake and Stefan Hougardy. „A Simple Approximation Algorithm for the Weighted Matching Problem“. In: *Information Processing Letters* 85.4 (2003), pages 211–213. DOI: 10.1016/S0020-0190(02)00393-9. [see page 43]

[DH11] Timothy A. Davis and Yifan Hu. „The University of Florida Sparse Matrix Collection“. In: *ACM Transactions on Mathematical Software* 38.1 (Nov. 2011), 1:1–1:25. DOI: 10.1145/2049662.2049663. [see pages 21–23, 171]

[DH72] William E. Donath and Alan 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 page 29]

[Din+95] Pedro C. Diniz, Steve Plimpton, Bruce Hendrickson, and Robert W. Leland. „Parallel Algorithms for Dynamically Partitioning Unstructured Grids“. In: *7th SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*. SIAM, 1995, pages 615–620. [see page 50]

[Din70] Yefim Dinitz. „Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation“. In: *Soviet Mathematics Doklady* 11.5 (Sept. 1970), pages 1277–1280. [see page 37]

[DK85] Alfred E. Dunlop and Brian 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. DOI: 10.1109/TCAD.1985.1270101. [see pages 2, 31]

[DKÇ13] Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. „Hypergraph Sparsification and Its Application to Partitioning“. In: *42nd International Conference on Parallel Processing (ICPP)*. 2013, pages 200–209. DOI: 10.1109/ICPP.2013.29. [see pages 78, 87]

[DM02] Elizabeth D. Dolan and Jorge J. Moré. „Benchmarking Optimization Software with Performance Profiles“. In: *Mathematical Programming* 91.2 (2002), pages 201–213. DOI: 10.1007/s101070100263. [see page 25]

[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. DOI: 10.1109/ICCAD.1997.643546. [see pages 177, 178]

[Dut93] Shantanu Dutt. „New Faster Kernighan-Lin-Type Graph-Partitioning Algorithms“. In: *International Conference on Computer-Aided Design (ICCAD)*. 1993, pages 370–377. DOI: 10.1109/ICCAD.1993.580083. [see page 31]

[EK72] Jack R. Edmonds and Richard M. Karp. „Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems“. In: *Journal of the ACM (JACM)* 19.2 (1972), pages 248–264. DOI: 10.1145/321694.321699. [see page 37]

[Fel13] Andreas E. Feldmann. „Fast Balanced Partitioning is Hard Even On Grids and Trees“. In: *Theoretical Computer Science* 485 (2013), pages 61–68. DOI: 10.1016/j.tcs.2013.03.014. [see pages 2, 16]

[FF15] Andreas E. Feldmann and Luca Foschini. „Balanced Partitions of Trees and Applications“. In: volume 71. 2. 2015, pages 354–376. DOI: 10.1007/s00453-013-9802-3. [see page 16]

[FF56] Lester Randolph Ford and Delbert R Fulkerson. „Maximal Flow Through a Network“. In: *Canadian Journal of Mathematics* 8 (1956), pages 399–404. DOI: 10.4153/CJM-1956-045-5. [see pages 12, 15, 36, 74]

[Fie+12] Jonas Fietz, Mathias J. Krause, Christian Schulz, Peter Sanders, and Vincent Heuveline. „Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries“. In: *European Conference on Parallel Processing (Euro-Par)*. Volume 7484. Springer, 2012, pages 818–829. DOI: 10.1007/978-3-642-32820-6_81. [see page 1]

[FM82] Charles M. Fiduccia and Robert M. Mattheyses. „A Linear-Time Heuristic for Improving Network Partitions“. In: *19th Conference on Design Automation (DAC)*. 1982, pages 175–181. DOI: 10.1145/800263.809204. [see pages 2, 5, 14, 29, 30, 32–34, 38, 65, 95, 96, 188]

[FPZ19] Kyle Fox, Debmalya Panigrahi, and Fred Zhang. „Minimum Cut and Minimum $k$-Cut in Hypergraphs via Branching Contractions“. In: *30th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2019, pages 881–896. DOI: 10.1137/1.9781611975482.54. [see page 15]

[FS21] Marcelo Fonseca Faraj and Christian Schulz. *Buffered Streaming Graph Partitioning*. Technical report. 2021. arXiv: 2102.09384. [see page 29]

[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. DOI: 10.1109/IPDPS.2018.00043. [see page 23]

[GH94] Olivier Goldschmidt and Dorit S. Hochbaum. „A Polynomial Algorithm for the $k$-Cut Problem for Fixed k“. In: *Mathematics of Operations Research* 19.1 (1994), pages 24–37. DOI: 10.1287/moor.19.1.24. [see page 15]

[GHR+95] Raymond Greenlaw, H James Hoover, Walter L Ruzzo, et al. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995. [see page 51]

[GHS22a] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. „Parallel Flow-Based Hypergraph Partitioning“. In: *20th International Symposium on Experimental Algorithms (SEA)*. Volume 233. LIPIcs. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 5:1–5:21. DOI: 10.4230/LIPIcs.SEA.2022.5. [see pages 4, 30, 60, 104, 113, 157]

[GHS22c] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. *Parallel Flow-Based Hypergraph Partitioning*. Technical report. 2022. arXiv: 2201.01556. [see pages 4, 30, 60, 157]

[GHW19] Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. „Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm“. In: *27th European Symposium on Algorithms (ESA)*. 2019, 52:1–52:17. DOI: 10.4230/LIPIcs.ESA.2019.52. [see pages 37, 38, 74, 77, 80, 157]

[Gib89] Phillip B. Gibbons. „A More Practical PRAM Model“. In: *1st Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 1989, pages 158–168. DOI: 10.1145/72935.72953. [see page 17]

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Volume 174. W. H. Freeman, 1979. [see pages 178, 179, 188]

[GJS76] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. „Some Simplified NP-Complete Graph Problems“. In: *Theoretical Computer Science* 1.3 (1976), pages 237–267. DOI: 10.1016/0304-3975(76)90059-1. [see pages 2, 15]

[GK21] Johnnie Gray and Stefanos Kourtis. „Hyper-Optimized Tensor Network Contraction“. In: *Quantum* 5 (2021), page 410. DOI: 10.22331/q-2021-03-15-410. [see page 2]

[GL81] Alan George and Joseph W Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981. [see page 95]

[GMR94] P Gibbons, Y Matias, and V Ramachandran. „The QRQW PRAM: Accounting for Contentionin Parallel Algorithms“. In: *6th Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1994, pages 638–648. [see page 17]

[Gon+12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. „PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs“. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012, pages 17–30. [see page 14]

[Got+20] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. „Advanced Flow-Based Multilevel Hypergraph Partitioning“. In: *18th International Symposium on Experimental Algorithms (SEA)* (2020). DOI: 10.4230/LIPIcs.SEA.2020.11. [see pages 5, 9, 36, 38, 41, 42, 47, 59, 74, 77, 79, 140, 159, 187, 192, 196]

[Got+21a] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. „Scalable Shared-Memory Hypergraph Partitioning“. In: *23rd Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 2021, pages 16–30. DOI: 10.1137/1.9781611976472.2. [see pages 4, 30, 60, 86, 113, 157]

[Got+21c] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. *Shared-Memory n-level Hypergraph Partitioning*. Technical report. 2021. arXiv: 2104.08107. [see pages 4, 30, 60, 86, 123, 157]

[Got+21e] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. „Deep Multilevel Graph Partitioning“. In: *29th European Symposium on Algorithms (ESA)*. Volume 204. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 48:1–48:17. DOI: 10.4230/LIPIcs.ESA.2021.48. [see pages 7, 23, 51, 52, 54–56, 59, 68, 149, 159, 175, 178, 197, 198]

[Got+22a] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. „Shared-Memory $n$-level Hypergraph Partitioning“. In: *24th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2022, pages 131–144. DOI: 10.1137/1.9781611977042.11. [see pages 4, 30, 60, 86, 123, 135, 157]

[Gra+79] Ronald L. Graham, Eugene L. Lawler, Jan K. Lenstra, and Rinnooy Kan. „Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey“. In: *Annals of Discrete Mathematics*. Volume 5. Elsevier, 1979, pages 287–326. [see page 179]

[Gra69] Ronald L. Graham. „Bounds on Multiprocessing Timing Anomalies“. In: *SIAM Journal on Applied Mathematics* 17.2 (1969), pages 416–429. [see page 179]

[GSS82] Leslie M. Goldschlager, Ralph A. Shaw, and John Staples. „The Maximum Flow Problem is Log Space Complete for P“. In: *Theoretical Computer Science* 21 (1982), pages 105–111. DOI: 10.1016/0304-3975(82)90092-5. [see page 51]

[GT88] Andrew V. Goldberg and Robert Endre Tarjan. „A New Approach to the Maximum-Flow Problem“. In: *Journal of the ACM (JACM)* 35.4 (1988), pages 921–940. DOI: 10.1145/48014.61051. [see pages 37, 80]

[GZ87] John R. Gilbert and Earl Zmijewski. „A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor“. In: *International Journal of Parallel Programming* 16.6 (1987), pages 427–449. DOI: 10.1007/BF01388998. [see page 34]

[Hau95] S. A. Hauck. „Multi-FPGA Systems“. PhD thesis. 1995. [see page 177]

[HB95] Scott Hauck and Gaetano Borriello. „An Evaluation of Bipartitioning Techniques“. In: *16th Conference on Advanced Research in VLSI (ARVLSI)*. Mar. 1995, pages 383–403. [see pages 2, 3, 40, 161]

[HB97] Scott Hauck and Gaetano Borriello. „An Evaluation of Bipartitioning Techniques“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 16.8 (1997), pages 849–866. DOI: 10.1109/43.644609. [see pages 35, 40, 157]

[Heu15a] Tobias Heuer. „Engineering Initial Partitioning Algorithms for direct $k$-way Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, Aug. 2015. [see pages 42, 45, 85, 96, 196]

[Heu18a] Tobias Heuer. „High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations“. Master Thesis. Karlsruhe Institute of Technology, Jan. 2018. [see pages 12, 42]

[HHK97] Lars W. Hagen, Dennis J.-H. Huang, and Andrew 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. DOI: 10.1109/43.662682. [see page 35]

[HL95] Bruce Hendrickson and Robert W. Leland. „A Multi-Level Algorithm For Partitioning Graphs“. In: *Supercomputing*. ACM, 1995, page 28. DOI: 10.1145/224170.224228. [see pages 3, 34, 35, 39, 40, 43, 46, 158]

[HM85] T. C. Hu and K. Moerder. „Multiterminal Flows in a Hypergraph“. In: *VLSI Circuit Layout: Theory and Design*. IEEE, 1985. Chapter 3, pages 87–93. [see page 10]

[HMS21a] Tobias Heuer, Nikolai Maas, and Sebastian Schlag. „Multilevel Hypergraph Partitioning with Vertex Weights Revisited“. In: *19th International Symposium on Experimental Algorithms (SEA)*. Volume 190. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 8:1–8:20. DOI: 10.4230/LIPIcs.SEA.2021.8. [see pages 4, 30, 178, 184]

[HNS20] Alexandra Henzinger, Alexander Noe, and Christian Schulz. „ILP-Based Local Search for Graph Partitioning“. In: *ACM Journal of Experimental Algorithmics (JEA)* 25 (2020), pages 1–26. DOI: 10.1145/3398634. [see pages 43, 158]

[HO92] Jianxiu Hao and James B. Orlin. „A Faster Algorithm for Finding the Minimum Cut in a Graph“. In: *4th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. ACM/SIAM, 1992, pages 165–174. URL: http://dl.acm.org/citation.cfm?id=139404.139439. [see page 15]

[HS17a] Tobias Heuer and Sebastian Schlag. „Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure“. In: *16th International Symposium on Experimental Algorithms (SEA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2017, 21:1–21:19. DOI: 10.4230/LIPIcs.SEA.2017.21. [see pages 5, 9, 22, 41, 45, 85, 88, 92, 93, 140, 149, 159, 177, 196]

[HS18] Michael Hamann and Ben Strasser. „Graph Bisection with Pareto Optimization“. In: *ACM Journal of Experimental Algorithmics (JEA)* 23 (2018). DOI: 10.1145/3173045. [see pages 1, 37, 38, 74, 79, 80]

[HSS10] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. „Engineering a Scalable High Quality Graph Partitioner“. In: *24th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2010, pages 1–12. DOI: 10.1109/IPDPS.2010.5470485. [see pages 43, 50, 54–56, 59, 75, 94, 198]

[HSS19a] Tobias Heuer, Peter Sanders, and Sebastian 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. DOI: 10.1145/3329872. [see pages 5, 36–38, 41, 42, 47, 59, 74, 76, 77, 159, 192, 196]

[IKS75] Tadakatsu Ishiga, Tokinori Kozawa, and Shoji Sato. „A Logic Partitioning Procedure by Interchanging Clusters“. In: *12th Conference on Design Automation (DAC)*. June 1975, pages 369–377. URL: http://dl.acm.org/citation.cfm?id=809089. [see page 39]

[JSA21] Nazanin Jafari, Oguz Selvitopi, and Cevdet Aykanat. „Fast Shared-Memory Streaming Multilevel Graph Partitioning“. In: *Journal of Parallel and Distributed Computing* 147 (2021), pages 140–151. DOI: 10.1016/j.jpdc.2020.09.004. [see page 29]

[Kab+17] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. „Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner“. In: *Proceedings of the VLDB Endowment* 10.11 (2017), pages 1418–1429. DOI: 10.14778/3137628.3137650. [see pages 2, 51, 94, 157, 198]

[Kam+19] Bogumi Kamiski, Valérie Poulin, Pawe Praat, Przemysaw Szufel, and François Théberge. „Clustering via Hypergraph Modularity“. In: *PLOS One* 14.11 (2019), pages 1–15. DOI: 10.1371/journal.pone.0224307. [see page 92]

[Kar+99] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. „Multilevel Hypergraph Partitioning: Applications in VLSI Domain“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pages 69–79. DOI: 10.1109/92.748202. [see pages 39, 43, 45, 46, 85, 95, 149, 159, 177, 180, 187]

[Kar13] George Karypis. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* 2013. [see page 163]

[Kar74] Alexander V. Karzanov. „Determining the Maximal Flow in a Network by the Method of Preflows“. In: *Soviet Mathematics Doklady.* Volume 15. 1974, pages 434–437. [see page 80]

[KAV04] Dorothy Kucar, Shawki Areibi, and Anthony Vannelli. „Hypergraph Partitioning Techniques“. In: *Dynamics of Continuous, Discrete and Impulsive Systems Series A: Mathematical Analysis* 11.2-3 (2004), pages 339–367. [see page 29]

[Kay+12] Enver Kayaaslan, Ali Pinar, Ümit V. Çatalyürek, and Cevdet Aykanat. „Partitioning Hypergraphs in Scientific Computing Applications through Vertex Separators on Graphs“. In: *SIAM Journal on Scientific Computing* 34.2 (2012). DOI: 10.1137/100810022. [see pages 6, 145, 197]

[KGB15] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. „Scalable SIMD-Efficient Graph Processing on GPUs“. In: *International Conference on Parallel Architectures and Compilation (PACT).* 2015, pages 39–50. DOI: 10.1109/PACT.2015.15. [see page 23]

[KK00] George Karypis and Vipin Kumar. „Multilevel $k$-way Hypergraph Partitioning“. In: *VLSI Design* 2000.3 (2000), pages 285–300. DOI: 10.1155/2000/19436. [see pages 14, 41, 44, 46, 85, 159, 180, 187]

[KK96] George Karypis and Vipin Kumar. „Parallel Multilevel $k$-way Partitioning Scheme for Irregular Graphs“. In: *ACM/IEEE Conference on Supercomputing.* 1996, page 35. DOI: 10.1109/SC.1996.32. [see pages 3, 49–52, 54–56, 59, 68, 94, 159]

[KK98a] George Karypis and Vipin Kumar. „A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs“. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pages 359–392. DOI: 10.1137/S1064827595287997. [see pages 43, 46, 95, 159]

[KK98b] George Karypis and Vipin Kumar. *hMETIS: A Hypergraph Partitioning Package, Version 1.5.3.* 1998. [see page 98]

[KK98c] George Karypis and Vipin Kumar. „Multilevel $k$-way Partitioning Scheme for Irregular Graphs“. In: *Journal of Parallel and Distributed Computing* 48.1 (1998), pages 96–129. DOI: 10.1006/jpdc.1997.1404. [see pages 35, 41, 43, 45, 46, 57, 159]

[KK98d] George Karypis and Vipin Kumar. „Multilevel Algorithms for Multi-Constraint Graph Partitioning“. In: *ACM/IEEE Conference on Supercomputing*. 1998, page 28. DOI: 10.1109/SC.1998.10018. [see page 14]

[KKM04] Jong-Pil Kim, Yong-Hyuk Kim, and Byung Ro Moon. „A Hybrid Genetic Approach for Circuit Bipartitioning“. In: *Genetic and Evolutionary Computation Conference (GECCO)*. Volume 3103. Springer, 2004, pages 1054–1064. DOI: 10.1007/978-3-540-24855-2_116. [see page 29]

[KL70] Brian W. Kernighan and Shen Lin. „An Efficient Heuristic Procedure for Partitioning Graphs“. In: *The Bell System Technical Journal* 49.2 (Feb. 1970), pages 291–307. DOI: 10.1002/j.1538-7305.1970.tb01770.x. [see pages 14, 29–32, 36, 38, 39]

[KÖ19] Gökçehan Kara and Can C. Özturan. „Graph Coloring Based Parallel Push-relabel Algorithm for the Maximum Flow Problem“. In: *ACM Transactions on Mathematical Software* 45.4 (2019), 46:1–46:28. DOI: 10.1145/3330481. [see pages 80, 81]

[KR13] Shad Kirmani and Padma Raghavan. „Scalable Parallel Graph Partitioning“. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2013, 51:1–51:10. DOI: 10.1145/2503210.2503280. [see page 52]

[Kra21a] Robert Krause. „Community Detection in Hypergraphs with Application to Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, Apr. 2021. [see page 92]

[Kri84] Balakrishnan Krishnamurthy. „An Improved Min-Cut Algorithm for Partitioning VLSI Networks“. In: *IEEE Transactions on Computers* 33.5 (1984), pages 438–446. DOI: 10.1109/TC.1984.1676460. [see page 14]

[Kum+14] K. Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. „SWORD: Workload-Aware Data Placement and Replica Selection for Cloud Data Management Systems“. In: *The VLDB Journal* 23.6 (2014), pages 845–870. DOI: 10.1007/s00778-014-0362-1. [see pages 2, 198]

[KW19] Todd A. Kuffner and Stephan G. Walker. „Why are p-Values Controversial?“ In: *The American Statistician* 73.1 (2019), pages 1–3. [see page 28]

[KW96] R. Klimmek and F. Wagner. *A Simple Hypergraph Min Cut Algorithm*. Technical report B 96-02. FU Berlin, 1996. [see page 15]

[Lab] University of Milano Laboratory of Web Algorithms. *Datasets*. URL: http://law.di.unimi.it/datasets.php. [see page 23]

[LaS+15] Dominique LaSalle, Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. „Improving Graph Partitioning For Modern Graphs and Architectures“. In: *5th Workshop on Irregular Applications - Architectures and Algorithms IA3*. 2015, 14:1–14:4. DOI: 10.1145/2833179.2833188. [see pages 52, 55, 94, 159]

[Lau21a] Moritz Laupichler. „Asynchronous $n$-Level Hypergraph Partitioning“. Master Thesis. Karlsruhe Institute of Technology, Nov. 2021. [see page 136]

[Law73] Eugene L. Lawler. „Cutsets and Partitions of Hypergraphs“. In: *Networks* 3.3 (1973), pages 275–285. DOI: 10.1002/net.3230030306. [see pages 12, 15, 37, 150]

[Len90] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, 1990. DOI: 10.1017/S0263574700015691. [see pages 2, 13]

[Li+17] Lingda Li, Robel Geda, Ari B. Hayes, Yan-Hao Chen, Pranav Chaudhari, Eddy Z. Zhang, and Mario 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. DOI: 10.1145/3084451. [see page 14]

[LK13] Dominique Lasalle and George Karypis. „Multi-Threaded Graph Partitioning“. In: *27th International Parallel and Distributed Processing Symposium (IPDPS)*. 2013, pages 225–236. DOI: 10.1109/IPDPS.2013.50. [see pages 50–52, 56, 57, 159]

[LK14] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. 2014. [see pages 23, 171, 186]

[LK16] Dominique LaSalle and George Karypis. „A Parallel Hill-Climbing Refinement Algorithm for Graph Partitioning“. In: *45th International Conference on Parallel Processing (ICPP)*. 2016, pages 236–241. DOI: 10.1109/ICPP.2016.34. [see pages 3, 52, 57, 58, 159]

[LLC95] Jianmin Li, John Lillis, and Chung-Kuan Cheng. „Linear Decomposition Algorithm for VLSI Design Applications“. In: *International Conference on Computer-Aided Design (ICCAD)* (Nov. 1995), pages 223–228. DOI: 10.1109/ICCAD.1995.480016. [see page 37]

[Low+12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. „Distributed GraphLab: A Framework for Machine Learning in the Cloud“. In: *Proceedings of the VLDB Endowment* 5.8 (2012), pages 716–727. DOI: 10.14778/2212351.2212354. [see page 1]

[LR04] Kevin J. Lang and Satish Rao. „A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts“. In: *10th International Integer Programming and Combinatorial Optimization Conference (IPCO)*. Volume 3064. 2004, pages 325–337. DOI: 10.1007/978-3-540-25960-2_25. [see page 38]

[LTM18] Moritz von Looz, Charilaos Tzovas, and Henning Meyerhenke. „Balanced $k$-means for Parallel Geometric Partitioning“. In: *47th International Conference on Parallel Processing (ICPP)*. ACM, 2018, 52:1–52:10. DOI: 10.1145/3225058.3225148. [see pages 29, 49]

[Lum+07] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. „Challenges in Parallel Graph Processing“. In: *Parallel Processing Letters* 17.1 (2007), pages 5–20. DOI: 10.1142/S0129626407002843. [see page 3]

[LW98] Huiqun Liu and 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 (1998), pages 50–59. DOI: 10.1109/43.673632. [see page 38]

[Maa20a] Nikolai Maas. „Multilevel Hypergraph Partitioning with Vertex Weights Revisited“. Bachelor Thesis. Karlsruhe Institute of Technology, 2020. [see pages 7, 178]

[Mal+21] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. „BiPart: A Parallel and Deterministic Hypergraph Partitioner“. In: *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2021, pages 161–174. DOI: 10.1145/3437801.3441611. [see pages 51, 52, 56, 59, 68, 159, 178]

[Mar+17] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. „Spinner: Scalable Graph Partitioning in the Cloud“. In: *33rd IEEE International Conference on Data Engineering ICDE*. 2017, pages 1083–1094. DOI: 10.1109/ICDE.2017.153. URL: https://doi.org/10.1109/ICDE.2017.153. [see pages 29, 51, 157]

[May+18] Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. „HYPE: Massive Hypergraph Partitioning With Neighborhood Expansion“. In: *IEEE International Conference on Big Data*. IEEE Computer Society, 2018, pages 458–467. DOI: 10.1109/BigData.2018.8621968. [see page 157]

[Mey12] Henning Meyerhenke. „Shape Optimizing Load Balancing for MPI-Parallel Adaptive Numerical Simulations“. In: *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*. 2012, pages 67–82. URL: http://www.ams.org/books/conm/588/11699. [see pages 29, 158]

[MK98] T Minyard and Y Kallinderis. „Octree Partitioning of Hybrid Grids for Parallel Adaptive Viscous Flow Simulations“. In: *International Journal for Numerical Methods in Fluids* 26.1 (1998), pages 57–78. [see page 29]

[MMS08] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. „A New Diffusion-Based Multilevel Algorithm for Computing Graph Partitions of Very High Quality“. In: *22nd International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2008, pages 1–13. DOI: 10.1109/IPDPS.2008.4536237. [see pages 29, 43, 158]

[MMS09] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. „A New Diffusion-Based Multilevel Algorithm for Computing Graph Partitions“. In: *Journal of Parallel Distributed Computing* 69.9 (2009), pages 750–761. DOI: 10.1016/j.jpdc.2009.04.005. [see pages 29, 158]

[MP14] Zoltán Á. Mann and Pál A. Papp. „Formula Partitioning Revisited“. In: *5th Pragmatics of SAT Workshop*. 2014, pages 41–56. DOI: 10.29007/9skn. [see page 22]

[MSS14] Henning Meyerhenke, Peter Sanders, and Christian Schulz. „Partitioning Complex Networks via Size-Constrained Clustering“. In: *13th International Symposium on Experimental Algorithms (SEA)*. Volume 8504. Springer, 2014, pages 351–363. DOI: 10.1007/978-3-319-07959-2_30. [see pages 14, 41, 44, 46, 69, 96, 149]

[MSS17] Henning Meyerhenke, Peter Sanders, and Christian Schulz. „Parallel Graph Partitioning for Complex Networks“. In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (2017), pages 2625–2638. DOI: 10.1109/TPDS.2017.2671868. [see pages 49, 51, 52, 55, 56, 59, 68, 149, 159]

[MWS17] Robert Matthews, Ron Wasserstein, and David Spiegelhalter. „The ASA's p-value statement, one year on“. In: *Significance* 14.2 (Apr. 2017), pages 38–41. [see page 28]

[Neu45] John von Neumann. *First Draft of a Report on the EDVAC*. 1945. [see page 16]

[NG04] Mark E. J. Newman and Michelle Girvan. „Finding and Evaluating Community Structure in Networks“. In: *Physical Review* 69 (2 Feb. 2004). [see page 92]

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. „Computing Edge-Connectivity in Multigraphs and Capacitated Graphs“. In: *SIAM Journal on Discrete Mathematics* 5.1 (1992), pages 54–66. DOI: 10.1137/0405004. [see page 15]

[Nuz14] Regina Nuzzo. „Scientific Method: Statistical Errors“. In: *Nature* 506.7487 (2014), pages 150–152. DOI: 10.1038/506150a. [see page 28]

[OS10] Vitaly Osipov and Peter Sanders. „$n$-Level Graph Partitioning“. In: *18th European Symposium on Algorithms (ESA)*. Springer. 2010, pages 278–289. DOI: 10.1007/978-3-642-15775-2_24. [see pages 34, 35, 41, 42, 46, 47, 70, 71, 153, 171, 172]

[PGK19] Md Anwarul K. Patwary, Saurabh K. Garg, and Byeong Kang. „Window-based Streaming Graph Partitioning Algorithm“. In: *Proceedings of the Australasian Computer Science Week Multiconference ACSW*. ACM, 2019, 51:1–51:10. DOI: 10.1145/3290688.3290711. [see page 29]

[Phe08] Chuck Pheatt. „Intel Threading Building Blocks“. In: *Journal of Computing Sciences in Colleges* 23.4 (2008), pages 298–298. [see pages 18, 24, 95, 97]

[Pin12] Michael Pinedo. *Scheduling*. Volume 29. Springer, 2012. [see page 179]

[Pip79] Nicholas Pippenger. „On Simultaneous Resource Bounds“. In: *20th Annual Symposium on Foundations of Computer Science*. 1979, pages 307–311. DOI: 10.1109/SFCS.1979.29. [see page 51]

[PM03] Joachim Pistorius and Michel Minoux. „An Improved Direct Labeling Method for the MaxFlow MinCut Computation in Large Hypergraphs and Applications“. In: *International Transactions in Operational Research* 10.1 (2003), pages 1–11. DOI: 10.1111/1475-3995.00389. [see page 37]

[PM07] David A. Papa and Igor L. Markov. „Hypergraph Partitioning and Clustering“. In: *Handbook of Approximation Algorithms and Metaheuristics*. 2007. DOI: 10.1201/9781420010749.ch61. [see pages 22, 29, 34]

[PQ80] Jean-Claude Picard and Maurice Queyranne. „On the Structure of All Minimum Cuts in a Network and Applications“. In: *Combinatorial Optimization II* (1980), pages 8–16. DOI: 10.1007/BF01581031. [see pages 38, 80]

[PR96] François Pellegrini and Jean Roman. „SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs“. In: *High-Performance Computing and Networking (HPCN)*. Volume 1067. Springer, 1996, pages 493–498. DOI: 10.1007/3-540-61142-8_588. [see pages 46, 55, 159]

[PZZ13] Bo Peng, Lei Zhang, and David Zhang. „A Survey of Graph Theoretical Approaches to Image Segmentation“. In: *Pattern Recognition* 46.3 (2013), pages 1020–1038. DOI: 10.1016/j.patcog.2012.09.015. [see page 1]

[Räc08] Harald Räcke. „Optimal Hierarchical Decompositions for Congestion Minimization in Networks“. In: *40th ACM Symposium on Theory of Computing (STOC)*. ACM, 2008, pages 255–264. DOI: 10.1145/1374376.1374415. [see page 15]

[Rah+13] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Márk Jelasity, and Seif Haridi. „JA-BE-JA: A Distributed Algorithm for Balanced Graph Partitioning“. In: *7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 2013, pages 51–60. DOI: 10.1109/SASO.2013.13. URL: https://doi.org/10.1109/SASO.2013.13. [see pages 29, 157]

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. „Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks“. In: *Physical Review E* 76.3 (2007), page 036106. DOI: 10.1103/PhysRevE.76.036106. [see page 41]

[RSS18] Harald Räcke, Roy Schwartz, and Richard Stotz. „Trees for Vertex Cuts, Hypergraph Cuts and Minimum Hypergraph Bisection“. In: *30th Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2018, pages 23–32. DOI: 10.1145/3210377.3210398. [see page 15]

[Saa95] Youssef Saab. „A Fast and Robust Network Bisection Algorithm“. In: *IEEE Transactions on Computers* 44.7 (1995), pages 903–913. DOI: 10.1109/12.392848. [see pages 36, 39, 41, 74, 121]

[San+19] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0. [see pages 17, 25]

[San93] Laura A. Sanchis. „Multiple-Way Network Partitioning with Different Cost Functions“. In: *IEEE Transactions on Computers* 42.12 (1993), pages 1500–1504. DOI: 10.1109/12.260640. [see page 34]

[Sch+16a] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. „$k$-way Hypergraph Partitioning via $n$-Level Recursive Bisection“. In: *18th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM. 2016, pages 53–67. DOI: 10.1137/1.9781611974317.5. [see pages 6, 9, 34, 41, 42, 44–46, 64, 85, 94, 95, 121, 129, 139, 145, 159, 179–181, 187, 196]

[Sch+19] Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Darren Strash. „Scalable Edge Partitioning“. In: *21st Workshop on Algorithm Engineering & Experiments (ALENEX)*. 2019, pages 211–225. DOI: 10.1137/1.9781611975499.17. [see page 14]

[Sch13] C. Schulz. „High Quality Graph Partitioning“. PhD thesis. Karlsruhe Institute of Technology, 2013. [see pages 43–46, 95, 159, 188]

[Sch20] Sebastian Schlag. „High-Quality Hypergraph Partitioning“. PhD thesis. Karlsruhe Institute of Technology, 2020. DOI: 10.5445/IR/1000105953. [see pages 3, 6–9, 29, 34, 35, 40–42, 46, 47, 64–66, 94, 95, 98, 121, 123, 129, 141, 145, 157, 177, 188, 190, 196, 197]

[Ser+16] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. „Clay: Fine-Grained Adaptive Partitioning for General Database Schemas“. In: *Proceedings of the VLDB Endowment* 10.4 (2016), pages 445–456. DOI: 10.14778/3025111.3025125. [see page 2]

[Sim91] Horst D. Simon. „Partitioning of Unstructured Problems for Parallel Processing“. In: *Computing Systems in Engineering* 2.2 (1991), pages 135–148. [see page 29]

[SK03] Navaratnasothie Selvakkumaran and George Karypis. „Multi-Objective Hypergraph Partitioning Algorithms for Cut and Maximum Subdomain Degree Minimization“. In: *International Conference on Computer-Aided Design (ICCAD)*. 2003. DOI: 10.1109/ICCAD.2003.1257889. [see page 14]

[SK72] Daniel G. Schweikert and Brian W. Kernighan. „A Proper Model for the Partitioning of Electrical Circuits“. In: *9th Conference on Design Automation (DAC)*. ACM, 1972, pages 57–62. DOI: 10.1145/800153.804930. [see pages 2, 10, 31, 92]

[SKK00] Kirk Schloegel, George Karypis, and Vipin Kumar. „Graph Partitioning for High-Performance Scientific Simulations“. In: *Sourcebook of Parallel Computing*. Morgan Kaufmann, 2000, pages 491–541. [see page 1]

[Slo+17] George M. Slota, Sivasankaran Rajamanickam, Karen D. Devine, and Kamesh Madduri. „Partitioning Trillion-Edge Graphs in Minutes“. In: *31nd International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pages 646–655. DOI: 10.1109/IPDPS.2017.95. URL: https://doi.org/10.1109/IPDPS.2017.95. [see pages 29, 49, 51, 157]

[Slo+20] George M. Slota, Cameron Root, Karen D. Devine, Kamesh Madduri, and Sivasankaran Rajamanickam. „Scalable, Multi-Constraint, Complex-Objective Graph Partitioning“. In: *IEEE Transactions on Parallel and Distributed Systems* 31.12 (2020), pages 2789–2801. DOI: 10.1109/TPDS.2020.3002150. [see pages 14, 49, 51, 157]

[SM16] Christian L. Staudt and Henning 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. DOI: 10.1109/TPDS.2015.2390633. [see pages 54, 93]

[SMR16] George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. „Complex Network Partitioning Using Label Propagation“. In: *SIAM Journal of Scientific Computing* 38.5 (2016). DOI: 10.1137/15M1026183. [see pages 29, 51]

[SS10] Peter Sanders and Christian Schulz. *Engineering Multilevel Graph Partitioning Algorithms*. Technical report. 2010. arXiv: 1012.0006. [see pages 34–38, 43–46]

[SS11] Peter Sanders and Christian Schulz. „Engineering Multilevel Graph Partitioning Algorithms“. In: *19th European Symposium on Algorithms (ESA)*. Springer, 2011, pages 469–480. DOI: 10.1007/978-3-642-23719-5_40. [see pages 5, 7, 34–38, 43, 47, 57, 59, 70, 74, 75, 77, 80, 158]

[SS12] Peter Sanders and Christian Schulz. „Distributed Evolutionary Graph Partitioning“. In: *12th Workshop on Algorithm Engineering & Experiments (ALENEX)*. 2012, pages 16–29. DOI: 10.1137/1.9781611972924.2. [see pages 29, 43, 55, 158, 159]

[SS13] Peter Sanders and Christian Schulz. „Think Locally, Act Globally: Highly Balanced Graph Partitioning“. In: *12th International Symposium on Experimental Algorithms (SEA)*. Volume 7933. Springer, 2013, pages 164–175. DOI: 10.1007/978-3-642-38527-8_16. [see page 175]

[SS15] Aaron Schild and Christian Sommer. „On Balanced Separators in Road Networks“. In: *14th International Symposium on Experimental Algorithms (SEA)*. Springer. 2015, pages 286–297. [see page 38]

[SS63] John C. Shepherdson and Howard E. Sturgis. „Computability of Recursive Functions“. In: *Journal of the ACM (JACM)* 10.2 (1963), pages 217–255. [see page 16]

[ST97] Horst D. Simon and Shang-Hua Teng. „How Good is Recursive Bisection?“ In: *SIAM J. Sci. Comput.* 18.5 (1997), pages 1436–1445. DOI: 10.1137/S1064827593255135. [see page 46]

[SV82] Yossi Shiloach and Uzi Vishkin. „An $\mathcal{O}(n^2 \log n)$ Parallel Max-Flow Algorithm“. In: *Journal of Algorithms* 3.2 (1982), pages 128–146. DOI: 10.1016/0196-6774(82)90013-X. [see page 80]

[SV95] Huzur Saran and Vijay V. Vazirani. „Finding $k$ Cuts within Twice the Optimal“. In: *SIAM Journal on Computing* 24.1 (1995), pages 101–108. DOI: 10.1137/S0097539792251730. [see page 15]

[SW91] John E. Savage and Markus G. Wloka. „Parallelism in Graph-Partitioning“. In: *Journal of Parallel and Distributed Computing* 13.3 (1991), pages 257–272. DOI: 10.1016/0743-7315(91)90074-J. [see page 51]

[SW97] Mechthild Stoer and Frank Wagner. „A Simple Min-Cut Algorithm“. In: *Journal of the ACM (JACM)* 44.4 (1997), pages 585–591. DOI: 10.1145/263867.263872. [see page 15]

[SWC04] Alan J. Soper, Chris Walshaw, and Mark Cross. „A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning“. In: *Journal of Global Optimization* 29.2 (2004), pages 225–241. DOI: 10.1023/B:JOGO.0000042115.44455.f3. [see pages 29, 171]

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. „Using Multilevel Graphs for Timetable Information in Railway Systems“. In: *4th Workshop on Algorithm Engineering & Experiments (ALENEX)*. Volume 2409. Springer, 2002, pages 43–59. DOI: 10.1007/3-540-45643-0_4. [see page 1]

[Tho08] Mikkel Thorup. „Minimum $k$-Way Cuts via Deterministic Greedy Tree Packing“. In: *40th ACM Symposium on Theory of Computing (STOC)*. ACM, 2008, pages 159–166. DOI: 10.1145/1374376.1374402. [see page 15]

[TK04a] Aleksandar Trifunovic and William J. Knottenbelt. „Parkway 2.0: A Parallel Multilevel Hypergraph Partitioning Tool“. In: *19th International Symposium on Computer and Information Sciences (ISCIS)*. Volume 3280. Springer, 2004, pages 789–800. DOI: 10.1007/978-3-540-30182-0_79. [see pages 50–52, 54–56, 59, 68, 94, 158]

[TK04b] Aleksandar Trifunovic and William J. Knottenbelt. „Towards a Parallel Disk-Based Algorithm for Multilevel $k$-way Hypergraph Partitioning“. In: *18th International Parallel and Distributed Processing Symposium (IPDPS)*. 2004. DOI: 10.1109/IPDPS.2004.1303286. [see pages 55, 94]

[TN94] Valerie E. Taylor and Bahram Nour-Omid. „A Study of the Factorization Fill-In for a Parallel Implementation of the Finite Element Method“. In: *International Journal for Numerical Methods in Engineering* 37.22 (1994), pages 3809–3823. [see page 29]

[Tso+14] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. „FENNEL: Streaming Graph Partitioning for Massive Scale Graphs“. In: *7th ACM International Conference on Web Search and Data Mining (WSDM)*. ACM, 2014, pages 333–342. DOI: 10.1145/2556195.2556213. [see page 29]

[VAR19] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Springer, 2019. [see page 18]

[VB05] Brendan Vastenhouw and Rob H. Bisseling. „A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication“. In: *SIAM Review* 47.1 (2005), pages 67–95. DOI: 10.1137/S0036144502409019. [see pages 43, 46, 95, 159, 180]

[Vis+12] Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. „The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite“. In: *49th Conference on Design Automation (DAC)*. ACM, June 2012, pages 774–782. DOI: 10.1145/2228360.2228500. [see pages 21, 23, 186]

[Wal04] C. Walshaw. „Multilevel Refinement for Combinatorial Optimisation Problems“. In: *Annals of Operations Research* 131.1–4 (2004), pages 325–372. DOI: 10.1023/B:ANOR.0000039525.80601.15. [see pages 43, 112]

[Wan+14] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. „How to Partition a Billion-Node Graph“. In: *30th International Conference on Data Engineering (ICDE)*. 2014, pages 568–579. DOI: 10.1109/ICDE.2014.6816682. [see page 1]

[WC00a] Chris Walshaw and Mark Cross. „Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm“. In: *SIAM Journal on Scientific Computing* 22.1 (2000), pages 63–80. DOI: 10.1137/S1064827598337373. [see pages 43, 46]

[WC00b] Chris Walshaw and Mark Cross. „Parallel Optimisation Algorithms for Multilevel Mesh Partitioning“. In: *Parallel Computing* 26.12 (2000), pages 1635–1660. DOI: 10.1016/S0167-8191(00)00046-6. [see pages 3, 50, 52, 54–56, 59, 94]

[WC89] Yen-Chuen A. Wei and Chung-Kuan Cheng. „Towards Efficient Hierarchical Designs by Ratio Cut Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. 1989, pages 298–301. DOI: 10.1109/ICCAD.1989.76957. [see page 14]

[WC91] Yen-Chuen A. Wei and Chung-Kuan Cheng. „Ratio Cut Partitioning for Hierarchical Designs“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 10.7 (1991), pages 911–921. DOI: 10.1109/43.87601. [see page 14]

[WCE97] Chris Walshaw, Mark Cross, and Martin G. Everett. „Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes“. In: *Journal of Parallel and Distributed Computing* 47.2 (1997), pages 102–108. DOI: 10.1006/jpdc.1997.1407. [see pages 50, 51, 54, 56, 59, 68, 178]

[Wil+07] Samuel Williams, Leonid Oliker, Richard W. Vuduc, John Shalf, Katherine A. Yelick, and James Demmel. „Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms“. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM Press, 2007, page 38. DOI: 10.1145/1362622.1362674. [see page 23]

[Wil91] Roy D. Williams. „Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations“. In: *Concurrency: Practice and Experience* 3.5 (1991), pages 457–481. DOI: 10.1002/cpe.4330030502. [see page 29]

[Wil92] Frank Wilcoxon. „Individual Comparisons by Ranking Methods“. In: *Breakthroughs in Statistics*. Springer, 1992, pages 196–202. DOI: 10.1007/978-1-4612-4380-9_16. [see page 28]

[WL16] Ronald Wasserstein and Nicole Lazar. „The ASA Statement on p-Values: Context, Process, and Purpose“. In: *The American Statistician* 70.2 (2016), pages 129–133. [see page 28]

[WW93] Dorothea Wagner and Frank Wagner. „Between Min Cut and Graph Bisection“. In: *18th International Symposium on Mathematical Foundations of Computer Science (MFCS)*. Volume 711. Springer, 1993, pages 744–750. DOI: 10.1007/3-540-57182-5_65. [see page 15]

[Yan+18a] Wenyin Yang, Li Ma, Ruchun Cui, and Guojun Wang. „Hypergraph Partitioning for Big Data Applications“. In: *IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBD-*

*Com/IOP/SCI)*. IEEE, 2018, pages 1705–1710. DOI: 10.1109/SmartWorld.2018.00289. [see page 14]

[Yan+18b] Wenyin Yang, Guojun Wang, Kim-Kwang Raymond Choo, and Shuhong Chen. „HEPart: A Balanced Hypergraph Partitioning Algorithm for Big Data Applications“. In: *Future Generation Computer Systems* 83 (2018), pages 250–268. DOI: 10.1016/j.future.2018.01.009. [see page 2]

[YP15] Boyang Yu and Jianping Pan. „Location-Aware Associated Data Placement for Geo-Distributed Data-Intensive Applications“. In: *IEEE Conference on Computer Communications (INFOCOM)*. IEEE. 2015, pages 603–611. DOI: 10.1109/INFOCOM.2015.7218428. [see page 2]

[YW96] Hannah H. Yang and D. F. Wong. „Efficient Network Flow Based Min-Cut Balanced Partitioning“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 15.12 (1996), pages 1533–1540. DOI: 10.1007/978-1-4615-0292-0_41. [see pages 36, 37, 74, 79, 80]

[ZCS97] Jason Y. Zien, Pak K. Chan, and Martine D. F. Schlag. „Hybrid Spectral/Iterative Partitioning“. In: *International Conference on Computer-Aided Design (ICCAD)*. 1997, pages 436–440. DOI: 10.1109/ICCAD.1997.643572. [see pages 34, 158]

[ZG02] Xiaojin Zhu and Zoubin Ghahramani. *Learning from Labeled and Unlabeled Data with Label Propagation*. Technical report. Carnegie Mellon University, 2002. [see page 41]

[ZHS06] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. „Learning with Hypergraphs: Clustering, Classification, and Embedding“. In: *20th Annual Conference on Neural Information Processing Systems*. MIT Press, 2006, pages 1601–1608. [see page 1]

# List of Publications

## Journal Articles

[1] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. „High-Quality Hypergraph Partitioning“. In: *ACM Journal of Experimental Algorithmics (JEA)* (Mar. 2022). Just Accepted. DOI: 10.1145/3529090.

[2] Tobias Heuer, Peter Sanders, and Sebastian 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. DOI: 10.1145/3329872.

## Conference Publications

[1] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. „Parallel Flow-Based Hypergraph Partitioning“. In: *20th International Symposium on Experimental Algorithms (SEA)*. Volume 233. LIPIcs. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 5:1–5:21. DOI: 10.4230/LIPIcs.SEA.2022.5.

[2] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. „Shared-Memory $n$-level Hypergraph Partitioning“. In: *24th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2022, pages 131–144. DOI: 10.1137/1.9781611977042.11.

[3] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. „Deep Multilevel Graph Partitioning“. In: *29th European Symposium on Algorithms (ESA)*. Volume 204. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 48:1–48:17. DOI: 10.4230/LIPIcs.ESA.2021.48.

[4] Tobias Heuer, Nikolai Maas, and Sebastian Schlag. „Multilevel Hypergraph Partitioning with Vertex Weights Revisited“. In: *19th International Symposium on Experimental Algorithms (SEA)*. Volume 190. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 8:1–8:20. DOI: 10.4230/LIPIcs.SEA.2021.8.

[5] Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. „PACE Solver Description: The KaPoCE Exact Cluster Editing Algorithm“. In: *16th International Symposium on Parameterized and Exact Computation (IPEC)*. Volume 214. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 27:1–27:3. DOI: 10.4230/LIPIcs.IPEC.2021.27.

[6] Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. „PACE Solver Description: KaPoCE: A Heuristic Cluster Editing Algorithm“. In: *16th International Symposium on Parameterized and Exact Computation (IPEC)*. Volume 214. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 31:1–31:4. DOI: 10.4230/LIPIcs.IPEC.2021.31.

[7] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. „Scalable Shared-Memory Hypergraph Partitioning“. In: *23rd Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 2021, pages 16–30. DOI: 10.1137/1.9781611976472.2.

[8] Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. „The Quantile Index - Succinct Self-Index for Top-k Document Retrieval“. In: *16th International Symposium on Experimental Algorithms (SEA)*. Volume 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 15:1–15:14. DOI: 10.4230/LIPIcs.SEA.2017.15.

[9] Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. „Practical Range Minimum Queries Revisited“. In: *16th International Symposium on Experimental Algorithms (SEA)*. Volume 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 12:1–12:16. DOI: 10.4230/LIPIcs.SEA.2017.12.

[10] Tobias Heuer, Peter Sanders, and Sebastian Schlag. „Network Flow-Based Refinement for Multilevel Hypergraph Partitioning“. In: *17th International Symposium on Experimental Algorithms (SEA)*. Volume 103. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 1:1–1:19. DOI: 10.4230/LIPIcs.SEA.2018.1.

[11] Tobias Heuer and Sebastian Schlag. „Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure“. In: *16th International Symposium on Experimental Algorithms (SEA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2017, 21:1–21:19. DOI: 10.4230/LIPIcs.SEA.2017.21.

[12] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. „Engineering a Direct $k$-way Hypergraph Partitioning Algorithm“. In: *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2017, pages 28–42. DOI: 10.1137/1.9781611974768.3.

[13] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. „$k$-way Hypergraph Partitioning via n-Level Recursive Bisection“. In: *18th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM. 2016, pages 53–67. DOI: 10.1137/1.9781611974317.5.

## Survey Articles

[1] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. „More Recent Advances in (Hyper)Graph Partitioning“. In: *Computing Research Repository (CoRR)* abs/2205.13202 (2022). arXiv: 2205.13202.

## Technical Reports

[1] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. *Parallel Flow-Based Hypergraph Partitioning*. Technical report. 2022. arXiv: 2201.01556.

[2] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. *Shared-Memory n-level Hypergraph Partitioning*. Technical report. 2021. arXiv: 2104.08107.

[3] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. *Deep Multilevel Graph Partitioning*. Technical report. 2021. arXiv: 2105.02022.

[4] Tobias Heuer, Nikolai Maas, and Sebastian Schlag. *Multilevel Hypergraph Partitioning with Vertex Weights Revisited*. Technical report. 2021. arXiv: 2102.01378.

[5] Tobias Heuer, Peter Sanders, and Sebastian Schlag. *Network Flow-Based Refinement for Multilevel Hypergraph Partitioning*. Technical report. 2018. arXiv: 1802.03587.

[6] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. *k-way Hypergraph Partitioning via n-Level Recursive Bisection*. Technical report. 2015. arXiv: 1511.03137.

## Theses

[1] Tobias Heuer. „High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations“. Master Thesis. Karlsruhe Institute of Technology, Jan. 2018.

[2] Tobias Heuer. „Engineering Initial Partitioning Algorithms for direct $k$-way Hypergraph Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, Aug. 2015.

## Supervised Theses

[1] Moritz Laupichler. „Asynchronous n-Level Hypergraph Partitioning“. Master Thesis. Karlsruhe Institute of Technology, 2021.

[2] Manuel Haag. „Engineering of Algorithms for Very Large k Partitioning“. Master Thesis. Karlsruhe Institute of Technology, 2021.

[3] Robert Krause. „Community Detection in Hypergraphs with Application to Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, 2021.

[4] Lukas Reister. „A Parallel Network Flow-Based Refinement Techinque for Multilevel Hypergraph Partitioning“. Master Thesis. Karlsruhe Institute of Technology, 2020.

[5] Tobias Fuchs. „Machine-Learning based Hypergraph Pruning for Partitioning“. Bachelor Thesis. Karlsruhe Institute of Technology, 2020.

[6] Nikolai Maas. „Multilevel Hypergraph Partitioning with Vertex Weights Revisited“. Bachelor Thesis. Karlsruhe Institute of Technology, 2020.

[7] Patrick Firnkes. „Throughput Optimization in a Distributed Database System via Hypergraph Partitioning“. Master Thesis. Karlsruhe Institute of Technology, 2019.