Karlsruhe Institute of Technology

Bachelor-Thesis

Multilevel Hypergraph Partitioning with Vertex Weights Revisited

Nikolai Maas

Date of submission: 22.05.2020

Advisors: Prof. Dr. Peter Sanders
Dr. Sebastian Schlag
M. Sc. Tobias Heuer

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

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

Karlsruhe, den 22. Mai 2020

Abstract

We analyze the k-way partitioning problem for hypergraphs with vertex weights and specifically the task of finding a balanced partition within the multilevel context. Problems from practical applications such as VLSI design and sparse matrix-vector multiplication naturally incorporate weighted instances, but the topic is only partially covered by current state-of-the-art partitioners. We investigate existing definitions of the balance constraint for weighted hypergraphs and show that they either do not ensure the existence of a feasible solution or provide an unnecessarily relaxed bound. Instead, we propose a new definition which overcomes these problems. In order to construct a partitioning algorithm that guarantees a balanced solution, we examine different aspects of the multilevel paradigm. In particular, additional measures are necessary for k-way partitioning via recursive bisection. To address this, we develop an approach based on an assignment of heavy vertices as *fixed vertices* to both blocks of the bisection (which we call a *prepacking*) and use theoretical results to prove that the balance of the final partition can be guaranteed. Our algorithm is integrated into the hypergraph partitioner KaHyPar and is compared with other state-of-the-art partitioners in extensive computational experiments. Competing algorithms produce a proportion of imbalanced solutions of up to 48.5% ($\varepsilon = 0.01$), 27.7% ($\varepsilon = 0.03$) and 7.5% ($\varepsilon = 0.1$), while our configuration always computes a balanced partition on the tested instances. The solution quality is equivalent to the current version of KaHyPar while slightly improving the running time.

Zusammenfassung

Wir analysieren das k-Wege-Partitionierungsproblem für Hypergraphen mit Knotengewichten und insbesondere die Aufgabe, eine balancierte Partition zu finden, im Multilevel-Kontext. Viele Probleme, die aus praktischen Anwendungen wie VLSI-Design oder dem wissenschaftlichen Rechnen kommen, beinhalten natürlicherweise gewichtete Instanzen. Das Thema wird von Partitionierern auf dem gegewärtigen Stand der Technik jedoch nur teilweise abgedeckt. Wir untersuchen existierende Definitionen der Balance von Partitionen für gewichtete Hypergraphen und zeigen, dass diese entweder nicht die Existenz einer balancierten Lösung garantieren oder mehr Freiraum lassen als nötig. Stattdessen stellen wir eine neue Definition vor, die diese Probleme löst. Um einen Partitionierungsalgorithmus zu konstruieren, der eine balancierte Partition garantiert, betrachten wir verschiedene Aspekte des Multilevel-Paradigmas. Insbesondere sind zusätzliche Maßnahmen nötig, wenn die k-Wege-Partition über eine rekursive Zweiteilung berechnet wird. Zu diesem Zweck entwickeln wir einen Ansatz, der auf der Zuweisung von schweren Knoten als *fixed vertices* zu den Blöcken der Zweiteilung basiert (was wir als *Prepacking* bezeichnen) und nutzen theoretische Ergebnisse, um zu beweisen, das die Balance der resultierenden Partition garantiert werden kann. Unser Algorithmus ist in den Hypergraphpartitionierer KaHyPar integriert und wird in ausführlichen rechnerischen Experimenten mit anderen Partitionierern auf dem Stand der Technik verglichen. Konkurrierende Algorithmen erzeugen einen Anteil an imbalancierten Lösungen von bis zu 48.5% ($\varepsilon = 0.01$), 27.7% ($\varepsilon = 0.03$) und 7.5% ($\varepsilon = 0.1$), während unsere Konfiguration für die getesteten Instanzen immer eine balancierte Partition berechnet. Die Qualität der Lösungen ist gleichwertig zur aktuellen Version von KaHyPar mit leichter Verbesserung der Laufzeit.

Acknowledgments

This thesis would not be possible without my supervisors Dr. Sebastian Schlag and Tobias Heuer. They introduced me to hypergraph partitioning and always showed me new directions if I did not know how to continue. Tobias reliably found new tasks for me in our meetings, all of which were important to improve this work. I am very thankful for all their support. Additionally, I would like to thank Robert Wilbrandt for his extensive proof-reading. His annotations were sometimes annoying, but always helpful.

Finally, I would like to thank my family for their love and support. Although this is easy to forget, having such a family is not commonplace.

Contents

1. Introduction **6**
1.1. Problem Statement . . . 7
1.2. Contribution . . . 7
1.3. Outline . . . 7

2. Preliminaries **8**
2.1. Hypergraphs . . . 8
2.2. Hypergraph Partitioning . . . 8
2.3. Bin Packing . . . 9
2.4. Load Balancing . . . 9
2.5. Segment Trees . . . 10

3. Related Work **12**
3.1. Hypergraph Partitioning . . . 12
3.2. Bin Packing . . . 13
3.3. Load Balancing . . . 15

4. Generalizing the Balance Constraint **17**
4.1. Formal Description . . . 17
4.2. Proposal for a Generalized Definition . . . 20

5. Balanced k-way Partitioning **21**
5.1. Recursive Bisection and Deep Balance . . . 21
5.2. Balancing Strategy for Recursive Bisection . . . 23
5.3. The Balance Property . . . 26
5.4. An Exact Prepacking Algorithm . . . 30
5.5. Heuristic Approaches for Prepacking . . . 35
5.6. Bin Packing Initial Partitioner . . . 36

6. Experimental Results **38**
6.1. Setup and Methodology . . . 38
6.2. Balance Constraint Definitions . . . 40
6.3. Different Prepacking Approaches . . . 41
6.4. Bin Packing Algorithms . . . 42
6.5. Restarted Bisections . . . 43
6.6. Comparison with Other Hypergraph Partitioners . . . 44

7. Conclusion **48**
7.1. Future Work . . . 48

References **52**

A. Running Time Overhead of the Balancing Strategy **53**

B. Detailed Running Time Analysis of the Prepacking Algorithm **54**

C. Detailed Comparison with Other Partitioners **56**

1. Introduction

Hypergraphs are a generalization of graphs where each (hyper)edge can connect more than two vertices. A direct generalization of the graph partitioning problem is the k-way hypergraph partitioning problem: Find a partition of the vertex set of the hypergraph into k disjoint blocks of roughly equal weight while minimizing an objective function. Prominent examples for the objective function are the *cut-net* and the *connectivity* (or $\lambda - 1$) metrics. The cut-net metric aims to minimize the sum of the weights of all cut hyperedges, i.e. hyperedges that connect more than one block of the partition. The connectivity metric considers the actual number λ of blocks connected by a hyperedge and counts the weight of each hyperedge $\lambda - 1$ times. Both revert to the edge-cut metric for plain graphs.

There are multiple application areas for hypergraph partitioning. The goal in VLSI design is to partition a circuit into smaller clusters such that the number of connecting wires is minimized [28]. Hypergraphs are a fitting model for circuits as wires can connect more than two gates. Another area is scientific computing, where the total communication volume of parallel sparse matrix-vector multiplications is accurately modeled by the connectivity metric [36]. In SAT solving, hypergraph partitioning is used to decompose a formula into smaller subformulas [34].

Because hypergraph partitioning is NP-hard [22], heuristic algorithms are used in practice. The most successful approach used by state-of-the-art partitioners is the *multilevel paradigm* [36, 28]. Here, the hypergraph is coarsened with iteratively applied vertex contractions, preserving the basic structure. When the hypergraph is small enough, an initial partition is calculated. This partition is improved with different refinement techniques during the uncoarsening of the hypergraph, which happens in reverse order of the vertex contractions.

Many real-world applications require the enforcement of a tight balance of the block weights for instances with a high variance in vertex weights. In VLSI design, these weights correspond to the cell area of a component [9] and in parallel sparse-matrix vector multiplication, vertices can represent tasks with a specific computational weight [10]. This requirement is modeled with the *balance constraint*: Each block of the resulting partition of the hypergraph can have at most $1 + \epsilon$ times the average block weight. However, for weighted instances it is non-trivial to find a balanced partition as this task is similar to the NP-hard [22] bin packing problem. Especially if there are vertices with weights close to the average block weight, it becomes hard to find a valid solution. Additionally, even the definition of the balance constraint itself is problematic for some vertex weight distributions. There are instances where no balanced solution is possible, which is why it can be necessary to adjust the definition of the balance constraint for hypergraphs with a large variance in vertex weights.

Nonetheless, the problem is not an important topic in the partitioning community: Past research focused mostly on unweighted benchmark instances, we refer to [37] for a historical overview. While some partitioners apply measures to restrict the allowed weight for contractions during the coarsening process [1, 3], in general this is not sufficient to ensure a balanced partition without modifications to the initial partitioning. Also, to our knowledge there is no recent work addressing vertex weights in the context of the multilevel paradigm.

In order to analyze the implications for weighted instances it is sensible to revisit each phase of the multilevel paradigm. During coarsening, it is necessary to ensure that the weight variance of the resulting hypergraph is not too large for finding a balanced initial partition. In the initial partitioning phase, bin packing techniques might be used for the calculation of a balanced partition. If no valid initial partition could be obtained, we can consider active rebalancing of the partition in the refinement.

An additional difficulty is that most state-of-the-art partitioners calculate a k-way partition via

recursive bisection [23, 11, 29]. Here, the initial hypergraph is divided into two blocks which are subdivided further by recursive applications of the algorithm. This makes satisfying the balance constraint even harder as the recursive bisection algorithm has no global view on all k blocks, but is only aware of the current bisection. Because balancing the two blocks of the bisection is not sufficient to ensure the balance of the final partition, additional methods are required.

1.1. Problem Statement

In this thesis we analyze the problems that arise in the context of the multilevel paradigm when weighted input hypergraphs are used. It should be investigated whether it is preferable to use a modified definition of the balance constraint for ensuring the existence of a feasible solution. Based on this, the main task of this work is to revisit the multilevel paradigm in order to construct a multilevel partitioner that guarantees the balance of the final k-way partition within reasonable constraints (i.e. for most of the benchmark instances). The work should be integrated into the hypergraph partitioner *KaHyPar* (**Ka**rlsruhe **Hy**pergraph **Par**titioning). The goal is to significantly reduce the proportion of imbalanced results compared to the latest version of KaHyPar without compromising more than necessary on result quality and running time.

1.2. Contribution

We propose a generalized definition for the balance constraint and present multiple methods to ensure a balanced k-way partition via recursive bisection: First, we construct an initial partitioning algorithm based on bin packing. Second, we develop a theoretical framework which includes a novel property describing the imbalance of partitions. Using its results, we construct a bisection algorithm that can guarantee the existence of a balanced solution without affecting the quality of the partition more than necessary. To the best of our knowledge, no such algorithm has been proposed yet. We integrate our algorithm into the partitioner KaHyPar and compare it with other state-of-the-art partitioners using the generalized balance constraint. Our benchmark set consists mostly of real-world instances, but also includes a small set of artificially constructed hypergraphs. The results show that there are classes of hypergraphs where no state-of-the-art partitioner is capable of reliably finding a balanced solution, with some competitors producing a proportion of up to 48.5% ($\varepsilon = 0.01$), 27.7% ($\varepsilon = 0.03$) and 7.5% ($\varepsilon = 0.1$) imbalanced partitions on the full benchmark set. Meanwhile, our configuration achieves a feasible solution in every single run (provided a sufficient time limit) and computes the best partitions for 41% of the benchmark instances. Compared to the latest version of KaHyPar, our approach has equivalent solution quality and only negligible running time overhead.

1.3. Outline

We introduce the required notation and basic definitions in Section 2. In Section 3 we give a summary of related work in the area. Then, we introduce a generalized version of the balance constraint in Section 4. The algorithms and strategies used to ensure a balanced solution are described in Section 5 along with the theoretical foundation. In Section 6 the experimental evaluation is presented. The most important findings and directions for future work are summarized in Section 7.

2. Preliminaries

2.1. Hypergraphs

Hypergraphs are a generalization of graphs where an edge can consist of more than two nodes.

Definition 2.1 (Hypergraph)**.** *An undirected weighted hypergraph $H = (V, E, c, \omega)$ consists of a set of hypernodes V with a weight function $c\colon V \to \mathbb{R}_{\geq 0}$ and a set of hyperedges E with a weight function $\omega\colon E \to \mathbb{R}_{>0}$. Each hyperedge $e \in E$ is a non-empty subset of the hypernodes.*

Hypernodes are also called *vertices* and hyperedges are also called *nets*. A vertex contained in a net is called a *pin* of the net. The *size* of a net e is its cardinality $|e|$. A vertex v is *incident* to a net e if $v \in e$. $I(v)$ denotes the set of all incident nets of v and $d(v) \coloneqq |I(v)|$ the *degree* of v. Two vertices u and v are *adjacent* if there is a net e such that u and v are incident to e. For subsets $V' \subseteq V$ and $E' \subseteq E$ we define

$$c(V') \coloneqq \sum_{v \in V'} c(v)$$
$$\omega(E') \coloneqq \sum_{e \in E'} \omega(e)$$

Furthermore, we use $c_{max} \coloneqq \max\{c(v) \mid v \in V\}$ for the maximum weight of a vertex.

Definition 2.2 (Subhypergraph)**.** *Let $H = (V, E, c, \omega)$ a hypergraph and $V' \subseteq V$ a subset of hypernodes. $H_{V'} = (V', E_{V'}, c, \omega)$ with $E_{V'} \coloneqq \{e \cap V' \mid e \in E \text{ with } e \cap V' \neq \emptyset\}$ is called the subhypergraph of H induced by V'.*

2.2. Hypergraph Partitioning

Definition 2.3 (k-way partition)**.** *A k-way partition of a hypergraph H is a partition of its vertex set into k disjoint blocks $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$ and $V_i \neq \emptyset$ for $i = 1, \ldots, k$.*

Note that a 2-way partition is also called a *bisection*. A k-way partition $\Pi = \{V_1, \ldots, V_k\}$ is *ε-balanced* if every block $V_i \in \Pi$ satisfies the *balance constraint* $c(V_i) \leq L_{max} \coloneqq (1+\varepsilon)\left\lceil \frac{c(V)}{k} \right\rceil$ for some parameter ε. The *connectivity set* of a net e is denoted by $\Lambda(e, \Pi) \coloneqq \{V_i \in \Pi \mid e \cap V_i \neq \emptyset\}$ and the *connectivity* by $\lambda(e, \Pi) \coloneqq |\Lambda(e, \Pi)|$. e is *cut* if $\lambda(e, \Pi) > 1$. The set of all cut nets is denoted by $E(\Pi) \coloneqq \{e \in E \mid \lambda(e, \Pi) > 1\}$.

Definition 2.4 (Hypergraph Partitioning Problem)**.** *The k-way hypergraph partitioning problem is to find an ε-balanced k-way partition of a hypergraph H that minimizes a certain objective function.*

Several different objective functions are used in hypergraph partitioning. Most popular is the *cut-net* metric, which is defined as

$$\text{cut}(\Pi) \coloneqq \sum_{e \in E(\Pi)} \omega(e)$$

Another important objective function is the (λ - 1)-metric or *connectivity* metric, which is defined as

$$(\lambda - 1)(\Pi) \coloneqq \sum_{e \in E} (\lambda(e, \Pi) - 1)\omega(e)$$

Both functions are equivalent for the 2-way case. Therefore, we refer to the objective function of a bisection also as the *cut* of the bisection.

2.3. Bin Packing

In the following, we use $[n] \coloneqq \{1, \dots, n\}$ to denote the set of the first n natural numbers. The classical bin packing problem consists of a sequence of elements L which are packed into bins of a certain capacity c. For a sequence $L = \langle a_1, \dots, a_n \rangle$ in $\mathbb{R}_{\geq 0}$ we call $i \in [n]$ an element and a_i the weight of i. For a subset of elements $B \subseteq [n]$ we denote the weight of B by $w(B) \coloneqq \sum_{i \in B} a_i$. The total weight of L is given by $w(L) \coloneqq \sum_{i=1}^{n} a_i$.

Definition 2.5 (Packing)**.** *A packing of a sequence L to k bins is a partition of $[n]$ into disjoint subsets $S = \{B_1, \dots, B_k\}$ such that $\bigcup_{j=1}^{k} B_j = [n]$.*

The *maximum bin weight* of a packing $S = \{B_1, \dots, B_k\}$ is defined as $\max(S) \coloneqq \max\{w(B_j) \mid j \in [k]\}$.

Definition 2.6 (Bin Packing)**.** *Given a sequence $L = \langle a_1, \dots, a_n \rangle$ and a maximum bin capacity c, the bin packing problem is to find a packing S of L to a minimum number of bins such that $\max(S) \leq c$.*

We refer to L and c as an *instance* $I = (L, c)$ of the bin packing problem. We use $w(I) \coloneqq w(L)$ and $a_{max} \coloneqq \max\{a_i \mid i \in [n]\}$ for the maximum weight of an element. Note that sometimes S is also called a *solution* of I. The corresponding decision problem, whether a packing into a given number of bins with capacity c exists, is known to be NP-complete [22].

The common quality metric for bin packing algorithms is the *asymptotic worst-case ratio*, where a small ratio corresponds to solutions close to the optimum. For a given optimization algorithm $\mathcal{A}$ for the bin packing problem, let $\mathcal{A}(I)$ denote the number of bins used by $\mathcal{A}$ and $OPT(I)$ the number of bins in an optimal solution. The *absolute* worst-case ratio of $\mathcal{A}$ is given by

$$R_A = \sup_I \left\{ \frac{\mathcal{A}(I)}{OPT(I)} \right\}.$$

The *asymptotic* worst-case ratio is given by

$$R_A^{\infty} = \lim_{k \to \infty} \sup_I \left\{ \frac{\mathcal{A}(I)}{OPT(I)} \mid OPT(I) \geq k \right\}.$$

2.4. Load Balancing

Load balancing, which is also referred to as *parallel machine scheduling* [21], is usually described as the scheduling of a set of jobs with assigned processing times to k parallel machines. However, we will from now on speak of elements and bins instead of jobs and machines, to ensure conformity to the bin packing notation. We use the same basic notations as in Section 2.3.

Definition 2.7 (Load Balancing)**.** *Given a sequence $L = \langle a_1, \dots, a_n \rangle$ and a number of bins k, the load balancing problem is to find a packing S of L to k bins that minimizes the maximum bin weight* $\max(S)$*.*

An instance of the load balancing problem is given by $I = (L, k)$. We use the additional notation $\overline{w} \coloneqq \overline{w}(I) \coloneqq \overline{w}(L) \coloneqq \frac{1}{k} w(L)$ for the *average bin weight* and $\mathrm{imb}(S) \coloneqq \max(S) - \overline{w}$ for the *imbalance* of a packing S. Note that in literature, $\max(S)$ is commonly called the *makespan* C_{max}. It is clear that the corresponding decision problem is equivalent to the bin packing problem and thus NP-complete [22]. However, the different structure of the approximation problem

requires other approximation algorithms than bin packing, although some of the algorithms bear a close resemblance.

As before, $\mathcal{A}(I)$ denotes the maximum bin weight used by an algorithm $\mathcal{A}$ on the instance I and $OPT(I)$ the maximum bin weight in an optimal solution. The common quality metric for an approximation algorithm is the *worst-case performance ratio* for a given number of bins k:

$$R_k(\mathcal{A}) = \sup \left\{ \frac{\mathcal{A}(I)}{OPT(I)} \mid I \text{ is an instance with } k \text{ bins} \right\}$$

In this thesis, we apply the problem to hypergraphs. For this, we translate a hypergraph to a load balancing instance, representing only the weight distribution of the hypernodes and not the hyperedge structure.

Definition 2.8 (Corresponding load balancing instance)**.** *Consider the k-way partitioning problem on a hypergraph $H = (V, E, c, \omega)$. The corresponding load balancing instance is defined as $I = (L, k)$ with elements $L := \langle c(v_1), \ldots, c(v_n) \rangle$ for some ordering of the hypernodes $V = \{v_1, \ldots, v_n\}$.*

Strictly speaking, the definition is ambiguous with respect to the ordering of the elements. However, the resulting instances are equivalent as the ordering of the elements is interchangeable in load balancing.

2.5. Segment Trees

A *segment tree* is a data structure providing efficient range queries over a sequence of elements. Suppose we have a sequence $L = \langle a_1, \ldots, a_n \rangle$ with n elements and an associative operation $\bigoplus$ that we want to apply to subsequences of L. Common examples for such an operation are the sum, product, minimum or maximum of the elements.

Definition 2.9 (Query result)**.** *The query result over the range $[i, j]$ is the value $\bigoplus_{l=i}^{j} a_l$ for $1 \leq i \leq j \leq n$.*

Each segment tree supports only one specific operation for range queries. It requires $\mathcal{O}(n)$ space and provides three operations:

- *build*, which is required to initialize the tree. It takes the sequence L as input and constructs the segment tree in $\mathcal{O}(n)$ time.
- *query* takes bounds i and j as input and calculates the query result over $[i, j]$ in $\mathcal{O}(\log(n))$ time.
- *update* takes an index i and a value x and updates the corresponding element of the sequence in $\mathcal{O}(\log(n))$ time.

A segment tree can be implemented as a binary tree where each node v is associated with a range $[i, j]$ and stores the query result for this range. The children of v are associated with the left and right half of the range, i.e. for $m := \left\lfloor \frac{1}{2}(i + j) \right\rfloor$ with the ranges $[i, m]$ and $[m + 1, j]$. Leafs are nodes with $i = j$ and are not stored explicitly, as their value is the value of the corresponding element of the underlying sequence. Each segment tree procedure can be implemented efficiently using recursion on the tree structure.

build constructs the segment tree bottom-up. For leafs, nothing needs be done. For a node v that is not a leaf, let l the value of the left child and r the value of the right child. Then v is initialized with $l \bigoplus r$.

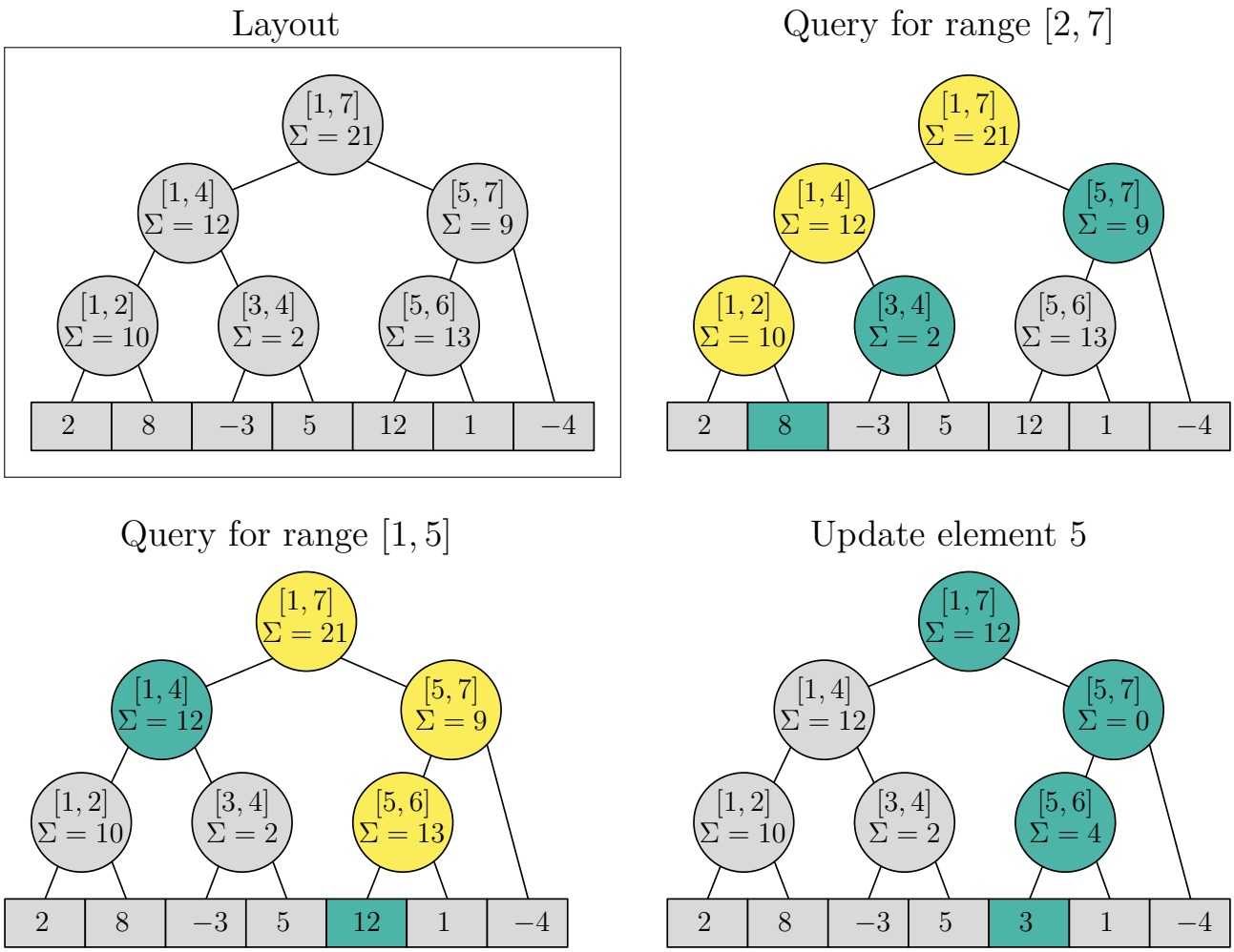

Figure 1: Illustration of a segment tree for sum calculation. Nodes where the value of the node is used or updated by the procedure are highlighted green. Nodes where the procedure is applied recursively without using the value are highlighted yellow.

Consider a *query* call with the range $[a, b]$. The segment tree is traversed starting with the root node (the node with range $[1, n]$). Let v the currently processed node with range $[i, j]$. There are three different cases. If $a = i$ and $b = j$, the value of v is returned. If the queried range is completely within the range of one child, i.e. $b \leq m$ or $a > m$, then *query* is called recursively on the respective child and the result is returned. In the last case, both children must be considered. Let l the query result over $[a, m]$ for the left child and r the query result over $[m+1, b]$ for the right child, both calculated with a recursive call. Then, the result is $l \bigoplus r$.

update assigns the new value to the corresponding element of the sequence. The updated query result for the parent node v is the new value combined with the value of the remaining child. This way, the update is propagated up to the root node. The *query* and *update* procedures are illustrated in Figure 1.

3. Related Work

3.1. Hypergraph Partitioning

Since the hypergraph partitioning problem is NP-hard [22] and it is even NP-hard to find approximate solutions for plain graphs [7], heuristic algorithms are used in practice for large hypergraphs. We present the most successful approaches in this section. Furthermore, the practical results of this thesis are integrated into the hypergraph partitioner *KaHyPar*. Therefore we give an overview over the implementation of KaHyPar in Section 3.1.2.

3.1.1. Multilevel Paradigm

The most successful approach for solving the hypergraph partitioning problem is the *multilevel paradigm* [36, 28, 37]. There are two main variants of the paradigm: In *direct k-way* partitioning, the algorithm has three phases (see Figure 2). First, the hypergraph is *coarsened*, i.e. a matching or clustering algorithm is used to compute vertex sets that are contracted. This process is repeated until the hypergraph has a predefined size, while still reflecting the basic structure of the input hypergraph. The coarsening thereby constructs a hierarchy of multiple levels corresponding to hypergraphs with decreasing size. Then, an *initial partitioning* algorithm calculates a k-way partition of the coarsest hypergraph. Due to the smaller size, the use of expensive algorithms for the partitioning is possible. During the *refinement* phase, the hypergraph is projected back to the original graph in reverse order of the contractions. At every level, refinement heuristics are used to further improve the quality of the solution. The most common technique is the Fiduccia-Mattheyses (FM) algorithm [16], based on local search.

In the *recursive bisection* variant of the paradigm, the algorithm calculates a bisection of the original hypergraph within the same three phases. Then, the algorithm is recursively applied to each of the two blocks to obtain the final k-way partition.

Main advantages of the multilevel paradigm are that it allows for expensive initial partitioning algorithms that operate globally while preserving near-linear computation time through the coarsening of the hypergraph, and that the refinement algorithms can escape local minima at the coarser levels while also improving the solution in detail at the finer levels.

3.1.2. The KaHyPar Partitioning Framework

KaHyPar is a multilevel hypergraph partitioner that uses a fine grained n-level approach, removing only one vertex for every level of the coarsening hierarchy. It is capable of finding high quality solutions for both the cut-net and the $(\lambda - 1)$ metric [25] while maintaining good performance compared to other widely-used hypergraph partitioners such as hMetis [1]. Both a direct k-way and a recursive bisection mode are supported. As our work is integrated into and tested with the direct k-way mode, we will describe it in more detail.

KaHyPar adds an initial *preprocessing* phase to the three phases of the multilevel paradigm. Here, a community detection algorithm is applied [6, 25]. This allows to use information about the global structure in the coarsening phase by restricting contractions to the detected communities. Contraction pairs are chosen using the *heavy-edge* rating function $r(u, v) := \sum_{e \in I(u) \cap I(v)} \frac{\omega(e)}{|e|-1}$. The coarsening algorithm works in passes, choosing per pass a contraction partner for each vertex according to the heavy-edge rating function. The passes are repeated until only $t = 160k$ hypernodes remain. To avoid imbalanced inputs to the initial partitioning phase, vertices that reach a certain weight threshold are not contracted further [1].

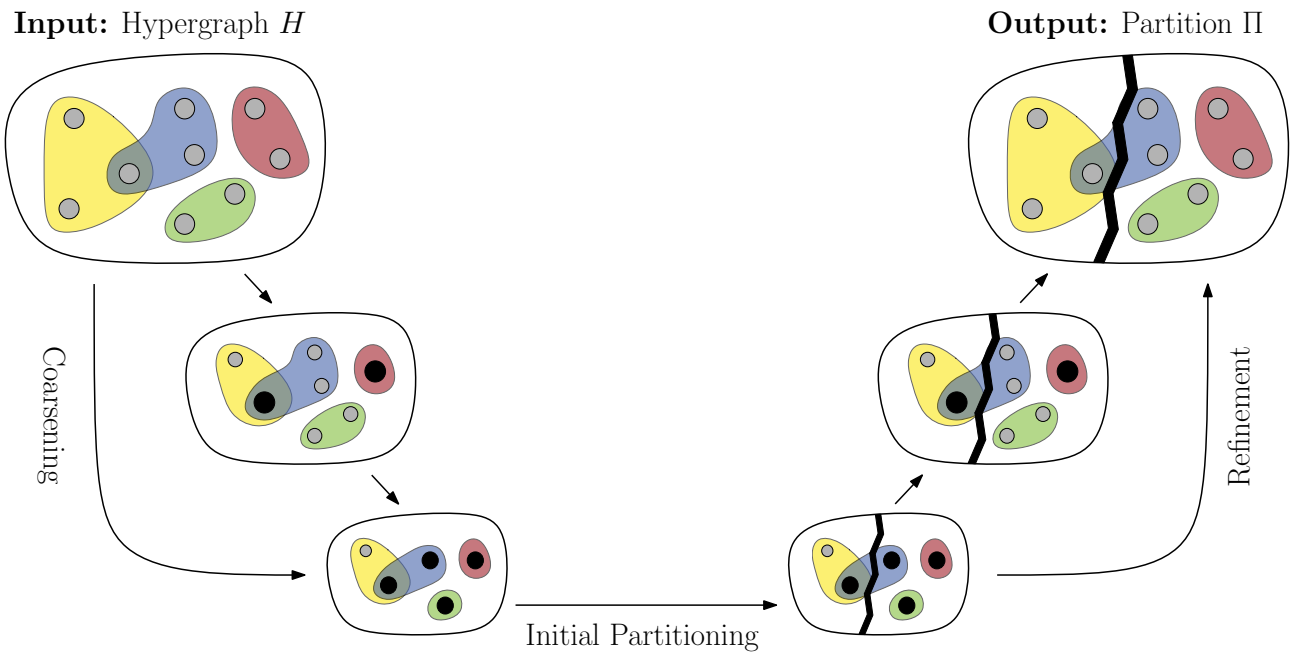

Figure 2: Hypergraph partitioning with the multilevel paradigm

Even in direct k-way mode, the initial partitioning phase uses a recursive bisection algorithm internally, as this produced better results compared to flat approaches [23]. In every recursive step, a bisection of the current subhypergraph is calculated in the following way: The hypergraph is coarsened further until it reaches a size of 300 hypernodes. Then, a bisection is calculated using a portfolio of different partitioning algorithms and selecting a balanced result with the best objective function [23]. The resulting blocks are uncoarsened using a 2-way FM local search algorithm.

To ensure that the resulting partition is balanced KaHyPar uses an *adaptive imbalance parameter* for the bisection steps which is defined as follows: Let H_{V_i} the current subhypergraph for which a k'-way partition should be calculated. Then

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

is used as an imbalance parameter for the bisection of V_i [38].

In the refinement phase, the resulting k-way partition is uncoarsened while optimizing the objective function via a k-way local search and a *maximum flow* algorithm. The FM local search algorithm maintains a *priority queue* for each block of the partition according to the *gain* of a hypernode and updates the gains of all hypernodes adjacent to a move with a *delta-gain* update strategy. To improve the running time a sophisticated *gain cache* [1, 38] is used that prevents expensive recalculations and an *adaptive stopping rule* which terminates the local search when a further improvement becomes unlikely [1]. This technique is combined with a refinement algorithm based on *Max-Flow-Min-Cut* calculations on a sparsified hypergraph flow network, further improving the solution quality [24].

3.2. Bin Packing

As *bin packing* has many real-world applications and calculating an optimal solution is usually not feasible due to the problem being NP-hard, approximation algorithms have been studied extensively [33]. We will describe some of the most relevant known algorithms.

3.2.1. On-line Algorithms

A bin packing algorithm is called *on-line* if it assigns each element only based on its weight and the current bin weights, i.e. without information about subsequent elements. Such algorithms are completely specified by a packing rule. Best known are the *first fit* (FF) and the *best fit* (BF) rules. The FF rule assigns each elements to the first fitting bin, while BF assigns each element to a fitting bin with the least space available. Both algorithms have been shown to have an asymptotic worst-case ratio of $\frac{17}{10}$ [27]. Another possible strategy is *worst fit* (WF), which assigns elements to the non-empty bin with the most space available.

However, the performance ratios of on-line algorithms are limited due to their inherent restrictions. Balogh et al. gave a lower bound of approximately 1.54278 on the asymptotic worst-case ratio of any on-line algorithm [4].

The performance ratio is much better if the elements are sorted in decreasing order before the packing. For this case, Johnson showed an upper bound on the asymptotic worst-case ratio of $\frac{5}{4}$ and a lower bound of $\frac{11}{9}$ for a whole class of algorithms [27]: A bin packing algorithm is an *any fit* algorithm if elements are never assigned to a new bin as long as there is a non-empty bin with enough space available. Note that FF, BF and WF (or respective FFD, BFD and WFD for the variants with decreasing element order) fall into this category. For the FFD algorithm it has been shown that the upper bound is exactly $\frac{11}{9}$ [26]. Furthermore, all of these algorithms can be implemented with runtime $\mathcal{O}(n \log(n))$ using appropriate data structures [27].

3.2.2. Off-line Algorithms

General off-line algorithms have a wider range of possible behaviors, specifically they can inspect the full sequence in advance and reassign elements. There are known linear-time off-line algorithms with good asymptotic worst-case ratios. Békési, Galambos and Kellerer showed an upper bound of $\frac{5}{4}$ for the so-called H_7-algorithm [5]. The basic idea is to split the elements in 7 classes based on their weight. Then the elements are assigned based on these classes with a rather complicated set of rules which include splitting the sets and pairing of elements. The linear runtime is possible because the number of classes is constant.

Another approach uses bin-oriented heuristics. The basic idea is to consider a complete bin in every step, choosing a subset of elements for the bin that minimizes the remaining (slack) capacity. One example is the *minimum bin slack* (MBS) algorithm presented by Gupta and Ho [20]. MBS considers all possible element subsets that fit into a bin, choosing the one which minimizes the slack capacity. The algorithm can be implemented in a way that terminates when an optimal solution is found, without considering the remaining possibilities.

MBS has different characteristics compared to the classical any fit algorithms. In particular, it is good when the optimal solution requires near-completely filled bins [20]. On the other hand, it has exponential worst-case runtime, although the time is polynomial for instances where the maximum number of elements per bin is limited. Also, bin-oriented heuristics tend to use the small elements for the first bins, which can provide poor solutions for later bins in some cases [17].

A number of improvements for MBS have been proposed. Fleszar and Hindi [18] presented multiple different approaches. One of them is the MBS' algorithm that uses a seed element for every bin before applying MBS. Furthermore, they introduced the *variable neighborhood search* algorithm, which is a local search algorithm designed to improve the solution given by a previously applied bin packing algorithm. Fleszar and Charalambous introduce the *sufficient average weight* heuristic that aims to improve existing bin-oriented algorithms by enforcing the use of element sets with sufficient average element weight [17].

The quality of the MBS heuristic and the mentioned improvements have been studied in computational experiments. However, to our knowledge there is yet no comprehensive analysis of worst-case ratio bounds for bin-oriented heuristics.

3.3. Load Balancing

Many classical approximation algorithms for *load balancing* use methods similar to common techniques for bin packing. We give a summary in Section 3.3.1 and present more recent approaches based on *metaheuristics* in Section 3.3.2.

3.3.1. Basic Algorithms

Probably best known is the *longest processing time* (LPT) algorithm proposed by Graham [19]. In its first step, LPT sorts the elements in non-increasing order. Then, every job is iteratively assigned to the bin with the lowest current weight. Note the similarity to the WFD algorithm for bin packing. The algorithm can be implemented with runtime $\mathcal{O}(n \log(n))$ and has been shown to have a worst-case performance ratio of $\frac{4}{3} - \frac{1}{3k}$ [19].

Coffman, Garey and Johnson proposed the MULTIFIT algorithm [12]. MULTIFIT uses the relation between the load balancing problem and the bin packing problem to apply the FFD bin packing algorithm to a load balancing instance by using a binary search for the minimum makespan value. It starts with an upper bound u and a lower bound l and works iteratively. In each iteration, the FFD algorithm is applied to the mean value $c = \frac{1}{2}(u + l)$. If the FFD result uses at most k bins, the upper bound is set to c. Otherwise, the lower bound is set to c. If b is the number of iterations, the runtime of MULTIFIT is $\mathcal{O}(n \log(n) + bn \log(k))$. Thus, a fixed value should be used for b if low computational time is required. Yue showed that the performance ratio is $\frac{11}{9} + (\frac{1}{2})^b$ [40].

An extension of the MULTIFIT algorithm is the LISTFIT heuristic proposed by Gupta and Ruiz-Torres [21]. It consists of 4 rounds where in each round n different lists (orderings of the elements) are generated. Each list is obtained by concatenating two sublists A and B. A and B are sorted increasingly or decreasingly respectively where each of the 4 rounds of the algorithm uses a different combination of orderings for A and B. A is initialized with all elements and B empty. In every of the n steps of a round, the last element of A is removed and inserted into B (according to the ordering of B). Then, MULTIFIT is applied to the resulting list, i.e. the list is used as input for the FF algorithm used by MULTIFIT. The output is the best obtained result for those $4n$ generated lists.

The runtime is $\mathcal{O}(n^2 \log(n) + bn^2 \log(k))$, where b is the number of iterations used for MULTIFIT. It has the same worst-case performance ratio as MULTIFIT, but has been shown to outperform the previously mentioned algorithms in computational experiments [21]. However, this comes at the cost of a factor $4n$ increase in computational costs compared to MULTIFIT.

3.3.2. Metaheuristic Approaches

A metaheuristic is a high-level procedure for searching optimum solutions that is applicable to a wider range of optimization problems. Metaheuristic approaches used for the load balancing problem include a *simulated annealing* approach presented by Lee, Wu and Chen [31]. It uses LPT for a baseline configuration and tries to optimize it by performing *neighborhood moves*, i.e. exchanging two elements where the first is chosen randomly from the bin with the highest weight and the second from the remaining bins. If it reduces the makespan, the move is accepted. If the

makespan increases, the move is accepted with a certain probability that depends on the delta of the makespan. The probability of accepting an increasing move is lowered as the algorithm progresses. An advantage of simulated annealing is that it can escape local optima, as there is still a probability for accepting moves that worsen the objective. Lee, Wu and Chen showed that the approach performs very well in computational experiments [31].

More recently, *swarm intelligence* based approaches for a number of combinatorial optimization problems, including makespan minimization, have been explored [35]. Swarm intelligence here describes a large field of algorithms that are inspired by nature, such as *evolutionary algorithms* and *particle swarm optimization*. For example, Kashan and Karimi present an application of particle swarm optimization to the load balancing problem with good experimental results [30].

4. Generalizing the Balance Constraint

Strongly related to the vertex weight distribution of a hypergraph is the notion of the balance of a partition. As explained in Section 2.2, a partition is ε-balanced if it satisfies the *balance constraint* $c(V_i) \leq L_{max} := (1+\varepsilon)\left\lceil \frac{c(V)}{k} \right\rceil$ for every block V_i. Usually, rather small values are chosen for ε, e.g. $\varepsilon = 0.1$ or $\varepsilon = 0.02$ [2]. The problem with this definition is that there are vertex weight distributions where no balanced partition is possible, specifically but not exclusively for small values of ε. The simplest example is probably a hypergraph H which contains a single vertex v with high weight, i.e. $c_{max} = c(v) > (1+\varepsilon)\left\lceil \frac{c(V)}{k} \right\rceil$. Obviously, no balanced partition exists as one block of the partition must contain v and is thus not balanced.

One may argue that in the case of a vertex with higher weight than average block weight, a partition is probably not very meaningful anyway and choosing a lower value for k is a more adequate solution. However, we will show that there are problem instances with no feasible solution even for much smaller maximum vertex weights, i.e. in the order of $\varepsilon\left\lceil \frac{c(V)}{k} \right\rceil$. Even if there is a balanced solution, the best possible balance might be close to L_{max} and thus the solution space could be very restricted. In our opinion this does not fit the intent of the definition well, as the value for ε is usually chosen to allow for additional optimization possibilities. Also, there are examples from practical applications with vertices with high weight, e.g. in circuit design [2]. In conclusion, it is desirable to formulate a generalized version of the balance constraint which ensures the existence of a balanced solution and is also practically applicable.

Of course, we seek a generalization that is as close as possible to the original definition (and, intuitively, equivalent for hypergraphs with unit vertex weights). Thus, we propose to use a formulation that remains the same except for the value of L_{max}. There are two intuitive approaches we considered at first, both using the maximum vertex weight c_{max} of H:

$$L_{max}^{(1)} := (1+\varepsilon)\max\left\{\left\lceil \frac{c(V)}{k} \right\rceil, c_{max}\right\} \tag{2}$$

$$L_{max}^{(2)} := (1+\varepsilon)\left(\left\lceil \frac{c(V)}{k} \right\rceil + c_{max}\right) \tag{3}$$

Note that Definition 2 is equivalent to the original definition if $c_{max} \leq \left\lceil \frac{c(V)}{k} \right\rceil$ and that for high values of c_{max} the value of $L_{max}^{(2)}$ is much higher than that of $L_{max}^{(1)}$ or the original definition. While those definitions are quite intuitive, both are not well suited for our purpose. This is because Definition 2 is not sufficient to guarantee the existence of a balanced partition while Definition 3 results in a much higher value for L_{max} than necessary for most instances. We will provide a more formal description of these issues in Section 4.1 and propose an alternative solution in Section 4.2. Additionally, we present a statistical comparison of possible definitions in Section 6.2.

4.1. Formal Description

To enable an analysis of the properties of possible definitions, we use a formalization of such a definition as a function of the vertex weight distribution. Obviously, the structure of a hypergraph (i.e. the hyperedges) is not relevant to the possible balance of a problem instance. Whether a balanced partition exists depends only on the weight distribution. This means we can omit the edge structure for our analysis, which is achieved by considering the corresponding load balancing instance I as defined in Section 2.4 instead of the hypergraph.

The weight of a bin is the block weight of the corresponding block in the partition. Thus, a balanced partition, i.e. a partition with $c(V_i) \leq L_{max}$ for every block V_i, exists if and only if

$OPT(I) \leq L_{max}$. Therefore we can define the balance constraint in terms of the load balancing problem by using an estimator for the maximum weight.

Definition 4.1 (Estimator)**.** *A bin weight estimator is a function γ that maps a load balancing instance I to an estimated maximum bin weight b.*

Given such a function γ, we will derive a definition for the balance constraint by setting $L_{max} \coloneqq (1+\varepsilon)\gamma(I)$, where I is the instance corresponding to H and k. Note that both proposals for a definition from the previous section can be formulated within these terms. For Definition 2 we have $\gamma^{(1)}(I) \coloneqq \max\{\left\lceil \frac{w(I)}{k} \right\rceil, a_{max}\}$. Definition 3 is obtained from $\gamma^{(2)}(I) \coloneqq \left\lceil \frac{w(I)}{k} \right\rceil + a_{max}$.
The next step is to define which properties are desirable for an estimator. First, the existence of a balanced partition should be guaranteed.

Definition 4.2 (Feasibility)**.** *An estimator γ is feasible if $\gamma(I) \geq OPT(I)$ for every instance I.*

Second, an estimator should produce results reasonably close to the optimum solution. In the original definition of the balance constraint, the ε parameter defines the margin compared to a perfectly balanced partition. Ideally, the same principle should apply to the generalized definition: ε defines the margin in relation to an optimal partition in terms of maximum bin weight.

Definition 4.3 (α-accuracy)**.** *An estimator γ is α-accurate if $\gamma(I) \leq (1+\alpha)OPT(I)$ for every instance I.*

Informally, this means that γ is close to the optimum by a proportion of α. We think a generalized definition should be accurate for a rather small value of α.
Considering $\gamma^{(1)}$, it is indeed 0-accurate as $OPT(I) \geq a_{max}$ and $OPT(I) \geq \left\lceil \frac{w(I)}{k} \right\rceil$ for any instance I. However, it rather obvious that it is not feasible. A minimal counterexample is an instance I with $L = \langle 2, 2, 2 \rangle$ and $k = 2$. Then $OPT(I) = 4$ but $\gamma^{(1)}(I) = 3 < OPT(I)$. Moreover, the bound given by $\gamma^{(2)}$ is already minimal in the sense that there are instances that come arbitrarily close to it even for a small maximum element weight, as we show in the following lemma.

Lemma 4.4 (Infeasible instances)**.** *For any $\varepsilon \in (0,1)$ there is an instance I with a_{max} in the order of $\varepsilon \left\lceil \frac{w(I)}{k} \right\rceil$ and $OPT(I)$ close to $\left\lceil \frac{w(I)}{k} \right\rceil + a_{max}$.*

More precisely, for every $\delta > 0$ there is an instance I such that $\frac{1}{3}\varepsilon \left\lceil \frac{w(I)}{k} \right\rceil \leq a_{max} \leq \varepsilon \left\lceil \frac{w(I)}{k} \right\rceil$ and $OPT(I) > (1-\delta)(\left\lceil \frac{w(I)}{k} \right\rceil + a_{max})$.

Proof. Let k the number of bins and choose an $m \in \mathbb{N}$ with $\frac{1}{2}\varepsilon \leq \frac{1}{m} \leq \varepsilon$. We construct an instance I with $n = km + 1$ elements by choosing uniform weight k for each element. Then $\left\lceil \frac{w(I)}{k} \right\rceil = \left\lceil \frac{k^2 m + k}{k} \right\rceil = km + 1$ and a_{max} is within the stated bounds:

$$a_{max} = k \leq k + \frac{1}{m} = \frac{1}{m} \left\lceil \frac{w(I)}{k} \right\rceil \leq \varepsilon \left\lceil \frac{w(I)}{k} \right\rceil$$

$$\frac{1}{3}\varepsilon \left\lceil \frac{w(I)}{k} \right\rceil = \frac{2}{3} \cdot \frac{1}{2}\varepsilon \left\lceil \frac{w(I)}{k} \right\rceil \leq \frac{2}{3m} \left\lceil \frac{w(I)}{k} \right\rceil = \frac{2}{3}(k + \frac{1}{m}) \leq k = a_{max}$$

.

Now consider a packing S for I. Because we have $km+1$ elements, S must contain a bin B_i with at least $m+1$ elements and thus $w(B_i) = \sum_{i \in B_i} a_i = k|B_i| \geq km + k$. Therefore $OPT(I) \geq km + k$. Using

$$\lim_{k\to\infty} \frac{OPT(I)}{\left\lceil \frac{w(I)}{k} \right\rceil + a_{max}} \geq \lim_{k\to\infty} \frac{km+k}{km+k+1} = 1$$

we find the desired instance I by choosing k sufficiently large according to δ. □

Furthermore, $\gamma^{(2)}$ is feasible. We omit a formal proof, but the basic idea is to apply the LPT algorithm (see Section 3.3.1) and observe that the smallest bin weight always remains below $\left\lceil \frac{w(I)}{k} \right\rceil$. As the bound of $\gamma^{(2)}$ is precise for some cases and it is feasible, does this mean it is suitable for a generalized definition?

Unfortunately, there are also instances where the bound is much higher than necessary. Indeed, this is probably true for most weighted hypergraphs (specifically those appearing in practical applications). A very simple example is the following construction: Let I an instance with k bins and k elements of uniform weight a. Then, $\gamma^{(2)}$ gives the bound $\gamma^{(2)}(I) = \left\lceil \frac{w(I)}{k} \right\rceil + a_{max} = 2a$. But $OPT(I) = a$ because we can just assign one element to each bin. That means $\gamma^{(2)}$ is at most 1-accurate. Indeed, we will show a more general statement: There is no estimator that is feasible and accurate at the same time without using a deeper consideration of the weight distribution, i.e. there is no elementary definition that is a good generalization. This is not too surprising considering that giving an upper bound for the balance constraint is computationally equivalent to calculating an approximation for the load balancing problem, which is NP-hard. However, we think it is useful to provide a formal reason for why it is necessary to use a more complex definition.

Definition 4.5 (Elementary)**.** *An estimator γ is* elementary *if it depends only on the basic parameters n, k, a_{max} and $w(I)$. Or more formally, if there is a function f s.t. $\gamma(I) = f(n, k, a_{max}, w(I))$ for every instance I.*

Theorem 4.6 (No elementary estimator)**.** *There is no estimator that is elementary, feasible and α-accurate for any $\alpha < 1$.*

Specifically, for any $\alpha < 1$ there are instances I and J of equal size n, number of bins k and maximum weight a_{max} with $w(I) = w(J)$ and $OPT(I) > (1+\alpha)OPT(J)$.

Proof. Let $\alpha < 1$, choose k sufficiently large s.t. $\frac{2k}{k+3} > 1 + \alpha$. We construct instances I and J with $n = 2k$ elements and k bins in the following way: For I, we choose the elements $a_1 = \cdots = a_{k+1} = k+1$ and the remaining elements with unit weight. For J, we choose elements $b_1 = \cdots = b_k = k+1$ and the remaining elements with weight 2. Both instances have maximum element weight $k+1$ and identical total weight:

$$w(I) = (k+1)(k+1) + (k-1) = k(k+3)$$

$$w(J) = k(k+1) + 2k = k(k+3)$$

Claim. $OPT(I) \geq 2k$ and $OPT(J) = \frac{w(J)}{k} = k+3$.

Consider a solution S of I. S has k bins, thus there is a bin B_i that contains at least two of the first $k+1$ elements. Therefore $w(B_i) \geq 2(k+1) = 2k+2$ which is even more than the claimed bound $OPT(I) \geq 2k$. We construct an optimal solution $S' = \{B'_1, \ldots, B'_k\}$ for J by distributing the elements uniformly among the bins, or formally $B'_j := \{j, j+k\}$. Then

$$w(B'_j) = a_j + a_{j+k} = (k+1) + 2 = k+3$$

which proves the claim. We conclude

$$OPT(I) \geq 2k = \frac{2k}{k+3}(k+3) = \frac{2k}{k+3}OPT(J) > (1+\alpha)OPT(J)$$

□

4.2. Proposal for a Generalized Definition

Given Theorem 4.6, we have to consider a different approach. As already explained, finding a precise upper bound for the block weight is in principal equivalent to the solution of the corresponding load balancing instance. Therefore, it is intuitive to make use of an approximation algorithm for load balancing. Let $\mathcal{A}$ such an algorithm, i.e. $\mathcal{A}$ takes a load balancing instance I as input and outputs the maximum bin weight of the computed approximation. We propose the following generalized definition:

Definition 4.7 (Generalized balance)**.** *Let H be a hypergraph, Π be a k-way partition of H, ε be an imbalance parameter and I be the load balancing instance with k bins corresponding to H. We call Π ε-balanced with respect to $\mathcal{A}$ if every block $V_i \in \Pi$ satisfies the balance constraint w.r.t. $\mathcal{A}$:*

$$c(V_i) \leq L_{max} \coloneqq (1+\varepsilon)\mathcal{A}(I)$$

Note that the definition is equivalent to the original definition in the case of unit weights as $\mathcal{A}$ computes a perfectly balanced partition where the maximum bin is of weight $\left\lceil \frac{w(I)}{k} \right\rceil$ (at least for a reasonable choice of $\mathcal{A}$). Using the notation of Section 4.1, this corresponds to an estimator with $\gamma(I) \coloneqq \mathcal{A}(I)$. Obviously, this estimator is feasible as the packing calculated by $\mathcal{A}$ corresponds to a partition with maximum block weight $\mathcal{A}(I)$. The accuracy depends on the exact algorithm used for $\mathcal{A}$. Theoretically, the definition would allow for the use of an exact algorithm. However, this approach would be quite impractical as the balance constraint is not very useful if it can not be verified in an efficient way.

Instead, we propose to use the LPT algorithm as a baseline for practical applications. It has a worst-case performance ratio of $R_m(LPT) = \frac{4}{3} - \frac{1}{3m}$ [19], i.e. the estimator is $\frac{1}{3}$-accurate. While not being the best available algorithm, we think it is reasonable to not use a too restrictive bound for hypergraphs with hard to pack weight distributions. After all, hypergraph partitioners are designed to optimize an objective function and not to find the best possible packing (a good approximation is nevertheless desirable). Furthermore, LPT is very easy to implement and runs in $\mathcal{O}(n \log(n))$. In practice it is very fast, as the constant factor is small, and for most practical instances it already finds a near-optimal solution. In conclusion, LPT provides a good baseline for the balance with very low overhead and will be used to calculate the imbalance parameter in our experiments.

Hypergraph partitioners currently do not support a balance constraint dependent on the weight distribution. To enable a fair comparison it is necessary to translate the generalized definition to the original definition. For an input parameter ε we calculate a modified parameter $\hat{\varepsilon}$ corresponding to the original definition with

$$(1+\hat{\varepsilon})\left\lceil \frac{c(V)}{k} \right\rceil = (1+\varepsilon)\mathcal{A}(I) \tag{4}$$

Then, $\hat{\varepsilon}$ is given as parameter to the hypergraph partitioner. Thereby all partitioners can use the original definition with the resulting maximum block weight of the generalized version.

5. Balanced k-way Partitioning

While we presented a definition for the balance constraint in Section 4.2 which ensures that a balanced solution is possible, this does not imply that partitioners are capable of finding one for uncommon vertex weight distributions. Specifically, as shown in our experimental results we can construct instances where no state-of-the-art partitioner finds a feasible solution. In order to analyze the source of the problems, we examine different aspects of the multilevel paradigm in this section. We will explain that a key challenge is to ensure the balance of a k-way partition that is calculated via *recursive bisection*. To solve this, we develop a high-level strategy that uses a *prepacking* approach. For the concrete algorithms used by this strategy, we develop a formal property which describes the balance of bisections with respect to the final k-way partition. This is used to construct an algorithm which guarantees a balanced solution and, in addition, heuristic algorithms. Finally, we analyze how to efficiently apply bin packing methods for recursive bisection, e.g. for an initial partitioning algorithm.

We start with a closer look into the different phases of the multilevel paradigm for a *direct k-way* approach (see Section 3.1.1). Consider the *coarsening* phase, where the input hypergraph H is transformed to a coarse hypergraph H' by a series of vertex contractions. If two vertices v and w are contracted, the resulting vertex has weight $c(v) + c(w)$. This could cause vertices of H' to have a much higher weight than vertices of H (obviously, the average vertex weight of H' is higher than that of H by a factor of $\frac{|V|}{|V'|}$, where V and V' are the vertex sets). Single elements with high weight are problematic for the balance and thus unfortunate coarsening might make it impossible to find a balanced partition of H'. However, we will argue that any problems with coarsening are simple to solve: As a rule of thumb, vertices can impact the balance of a k-way partition critically if their weight is in the order of $\varepsilon \left\lceil \frac{c(V)}{k} \right\rceil$ (see Lemma 4.4). For all usual choices of ε and the size of H' this is much higher than the average vertex weight of H'. Therefore it is sufficient if the coarsening algorithm asserts that no vertices with too high weight are created. A simple solution for this is to forbid further contractions of vertices above a certain weight threshold, which is already implemented by some state-of-the-art partitioners [1, 3].

The *initial partitioning*, where a k-way partition Π' of H' is calculated, is clearly a key step for the balance. If Π' is ε-balanced, it already induces an ε-balanced partition Π of H. And if Π' is imbalanced, it is unlikely that the final partition will be balanced. Thus this section will be dedicated to the task of ensuring a balanced initial partition. It is important to note that it is rather easy to find a balanced partition Π' for an algorithm that calculates the initial partition directly as it has a global view on all blocks (in the worst case a bin packing approach could be used to guarantee a balanced result). However, many partitioners use a recursive bisection approach for initial partitioning (see Section 3.1.2, [23, 11, 29]).

The *refinement* phase does not introduce additional problems. Refinement algorithms usually adhere to constraints ensuring that the balance of a partition is not destroyed [16], i.e. it is sufficient that Π' is balanced to get a balanced partition Π of H. An interesting idea in this area is that refinement could even try to fix an imbalanced partition by applying some kind of *rebalancing* technique. Though, this is unnecessary if Π' is already balanced and left for future work, as initial partitioning is of primary importance. This view is also supported by our experimental results presented in Section 6.6: The changes we introduced to initial partitioning are sufficient to find a balanced solution in every run.

5.1. Recursive Bisection and Deep Balance

A common method to calculate a k-way partition in the multilevel paradigm partition is recursive bisection. It can be applied either to the whole partitioning algorithm or to implement the

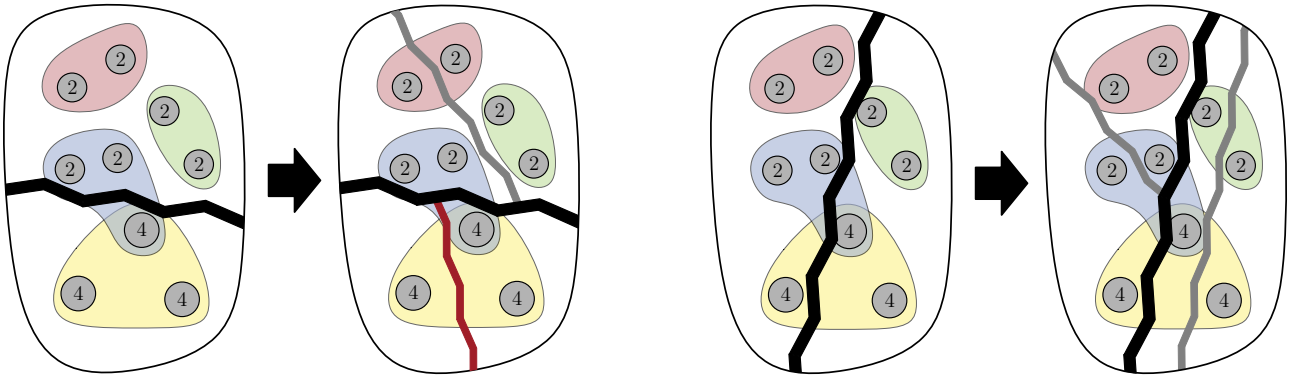

Figure 3: Illustration of the deep balance property. The left bisection is not deeply balanced as no 4-way partition with maximum block weight 7 can be found recursively. The right bisection is deeply balanced.

initial partitioning algorithm in direct k-way mode. While it is straightforward to find a single balanced bisection, problems arise if a k-way partition is calculated via recursive bisection and both blocks of the bisection must be further divided by recursive applications of the algorithm. Unfortunately, the balance of the bisection itself is not sufficient to enable a balanced subdivision of the blocks. We describe this idea with the notion of *deep balance*. In the following, we assume that k is even.

Definition 5.1 (Deep balance)**.** *Let* $\Pi = \{V_1, V_2\}$ *be a bisection of* H*.* Π *is deeply balanced with respect to the maximum block weight* L_{max} *if for every* V_i *with* $i \in \{1, 2\}$ *there is a subpartition* $\Pi_i = \{V_1^{(i)}, \ldots, V_{k/2}^{(i)}\}$ *of* V_i *such that* $c(V_j^{(i)}) \leq L_{max}$ *for all* $j \in [\frac{k}{2}]$*.*

To demonstrate why it is in general non-trivial to find bisections that are deeply balanced, we give an example of a perfectly balanced bisection which does not satisfy deep balance. An illustration of the example is given in Figure 3. The hypergraph consists of nine vertices $V = \{v_1, \ldots, v_9\}$ where the first three vertices have weight 4 and the remaining vertices have weight 2. We search for a 4-way partition via recursive bisection, i.e. $k = 4$. The upper allowed weight per block is $L_{max} = 7$. Consider the bisection $\Pi = \{V_1, V_2\}$ with $V_1 = \{v_1, v_2, v_3\}$ and $V_2 = \{v_4, \ldots, v_9\}$. It is impossible to subdivide V_1 recursively within the allowed block weight: For every bisection Π_1 of V_1, one of the blocks of Π_1 must contain two vertices of V_1 and therefore have a weight of at least 8. That means Π is not deeply balanced although it is perfectly balanced, i.e. $c(V_1) = c(V_2)$, and the upper block weight L_{max} even allows for a higher weight than necessary, as a 4-way partition with block weight 6 is possible.

It should be observed that the definition of deep balance only depends on the vertex weights and is independent of the edge structure of the hypergraph. Thus we will apply the definition analogously to the corresponding load balance instance of a hypergraph. The hypernodes correspond to elements, a partition corresponds to a packing and L_{max} corresponds to the bin capacity. Furthermore, for improved clarity we will call the sets of the top-level packing *blocks* and the subsets of a block which are calculated recursively *bins*.

Finding a deeply balanced bisection. To calculate a balanced k-way partition via recursive bisection, we need to be able to compute a deeply balanced bisection even for non-trivial weight distributions. There are different possible approaches to achieve this:

We could use an initial partitioning algorithm designed to optimize the balance. For example, we describe in Section 5.6 an initial partitioning algorithm based on bin packing techniques. However, there are at least two problems to this approach. An initial partitioner that optimizes

the balance does not optimize the objective function and thus the resulting bisection will have a poor cut for most cases. While the refinement applied after the initial partitioning can improve the cut, there is also a problem to the refinement itself (at least in its usual version): Being a 2-way refinement, it is unaware of the deep balance of the bisection and only asserts that the weight of the two blocks is within the given bound. Therefore, even if a deeply balanced bisection is calculated by the initial partitioner, the deep balance might be destroyed by the refinement that is applied afterwards.

A possible extended approach is to integrate the concept of deep balance into the initial partitioning algorithm and the refinement. More specifically, we could calculate the effect on the deep balance of the bisection for every assignment of a hypernode or move of a hypernode to a block. While this idea has a certain appeal, there are also major disadvantages. It is not clear how it could be efficiently implemented, i.e. without adding significant overhead to the running time of the initial partitioning and refinement. Another problem is that the approach is not generalizable. The integration must be done for every initial partitioning algorithm and refinement algorithm separately and it is doubtful, whether all such algorithms can be easily adjusted to handle this concept. There is also a more fundamental objection: For an input hypergraph H the corresponding coarsened hypergraph H' is usually of much smaller size (e.g. 300 hypernodes for KaHyPar [38]). As the bisection is calculated on H', there are very few hypernodes per block of the final k-way partition if a high value for k is given. But to ensure deep balance, all of those blocks must be considered. The bin packing problem is for most instances harder the fewer elements fit in a bin on average, thus it is unclear how to reason about the deep balance of the bisection if the hypergraph is that coarse.

We propose to use a third alternative which operates on H. It is based on the concept of *fixed vertices* [8, 11]. A fixed vertex is a hypernode which is assigned to a block before the bisection is calculated. This assignment can not be changed by the bisection. The basic idea is the following: If we fix the hypernodes with the highest weights using a bin packing algorithm, we can improve the possibility or even guarantee the existence of a deeply balanced bisection while applying the existing algorithm that optimizes the cut of the bisection. Intuitively, this works because the balance is mostly affected by the heaviest hypernodes. We call such a set of fixed vertices a prepacking and in Section 5.2 we show how to integrate this approach into a multilevel partitioner. How a prepacking can indeed guarantee the deep balance of a bisection is analyzed in Section 5.3.

Note that this approach enables a small cut as long as the prepacking is not too large. Furthermore, it is rather easy to implement. One option is to remove the fixed vertices before applying the bisection and add them to the result afterwards, which has the disadvantage that the bisection algorithm does not know about their effect on the cut. Alternatively, the coarsening, initial partitioning and refinement algorithms can skip the fixed vertices for contractions and packings.

5.2. Balancing Strategy for Recursive Bisection

In this section we present our strategy for integrating the prepacking approach into a partitioner that calculates a k-way partition via recursive bisection. We will explain the strategy based on a single bisection, but naturally it is applied in the same way to all further bisections that are calculated recursively.

Consider an input hypergraph $H = (V, E, c, \omega)$ where a k-way partition for an even k with a parameter ε is searched. A prepacking assigns disjoint subsets P_1 and P_2 of V as *fixed vertices* to their respective blocks. That means that the bisection must assign all vertices in P_1 to the

first block and all vertices in P_2 to the second block. Within this restriction, any (multilevel) partitioning algorithm can be used to calculate the bisection.

As a prepacking reduces the possible solution space for finding a good cut, we try to avoid it as much as possible. Therefore we use a fallback concept: First, we try to calculate a bisection without a prepacking. Only if it fails to be deeply balanced, we calculate a new bisection where a prepacking of the heaviest vertices is applied previously. If a prepacking is necessary, it is still favorable to prepack as few vertices as possible. To achieve this we use multiple restriction levels which are invoked as fallback if the previous levels fail to produce a deeply balanced bisection. The levels differ in the way how the number of fixed vertices is calculated. To implement these fallbacks, a method for testing whether a bisection is deeply balanced is required. We use a similar method to the generalized balance constraint defined in Section 4.2, with a load balancing algorithm that outputs the maximum bin weight.

Definition 5.2 (Current maximum bin)**.** *Let H be a hypergraph, k be the number of blocks and I be the corresponding load balancing instance. For a load balancing algorithm $\mathcal{A}$, the current maximum bin is denoted by* $\max_B(H,k) \coloneqq \mathcal{A}(I)$.

Consider a bisection $\Pi = \{V_1, V_2\}$ of H and the allowed block weight $L_{max} = (1+\varepsilon)\left\lceil \frac{c(V)}{k} \right\rceil$ for the k-way partition. If $\max_B(H_{V_i}, \frac{k}{2}) \leq L_{max}$ holds for the subhypergraph H_{V_i} of H that correspond to the block V_i for all $i \in \{1,2\}$ then Π is deeply balanced. This can be implemented with the LPT algorithm, which provides a good approximation in $\mathcal{O}(|V|\log(|V|))$ time.

We think it is reasonable to restrict the prepacking to not use the maximum possible block weight, allowing more freedom for the lower levels of the bisection. Therefore we use a parameter similar to the adaptive imbalance parameter already used by KaHyPar [38]. We will adjust Equation 1 to use the current maximum bin instead of the average bin weight.

Definition 5.3 (Adaptive bin imbalance)**.** *Let H' the current subhypergraph of H for which a k'-way partition should be calculated. Then the adaptive bin imbalance parameter is given by*

$$\varepsilon'_B \coloneqq \left((1+\varepsilon)\frac{c(V)}{k} \cdot \frac{1}{\max_B(H',k')}\right)^{\frac{1}{\lceil \log_2(k') \rceil}} - 1$$

The procedure for calculating the bisection works as follows: According to the current level, a number of the heaviest vertices is prepacked using bin packing techniques (compare Section 5.6). For the prepacking calculation $L'_{max} \coloneqq (1+\varepsilon'_B)\max_B(H,k)$ is used as upper allowed block weight. Then, a bisection $\Pi = \{V_1, V_2\}$ of H is calculated with a partitioning algorithm which is aware of the fixed vertices. If Π is not deeply balanced according to our test, i.e. $\max_B(H_{V_i}, \frac{k}{2}) > L_{max}$ for either $i = 1$ or $i = 2$, then the procedure is restarted with increased restriction level. Otherwise, we apply recursive bisection to the blocks of Π and consider the resulting k-way partition Π' of H. If Π' is ε-balanced, it is accepted and returned as result. If not, the procedure is restarted with increased level.

Of central importance for the effectiveness of the strategy is the amount of prepacked vertices: If not enough vertices are prepacked, no deeply balanced bisection is found. If on the other hand too many vertices are prepacked, the optimization possibilities are very restricted and the bisection has a poor cut. In Section 5.4 we present a prepacking algorithm which can guarantee the deep balance of a bisection for the calculated prepacking within some constraints (for a proof of this property see Theorem 5.14). However, there are cases where this algorithm prepacks more vertices than necessary for finding a deeply balanced bisection. Therefore we additionally use a heuristic approach which is presented in Section 5.5. The idea is to prepack a smaller number of vertices which is not sufficient to ensure it, but at least has (according

to our experiments) a high likelihood for the deep balance of the bisection. Then, hopefully a bisection is calculated that is still deeply balanced, but has a better cut than one found with the exact prepacking algorithm as the smaller number of fixed vertices allows for more solution possibilities. Thus we use three levels for our strategy: At the first level, no prepacking is applied. At the second level, a prepacking is calculated with the heuristic approach. Finally, the exact prepacking algorithm is applied at the third level. A pseudocode description is given in Algorithm 1.

Algorithm 1: Balanced recursive bisection

```
1  Function recursiveBisection(H, ε, k)
       Input: Hypergraph H = (V, E, c, ω), imbalance parameter ε, number of blocks k
       STATES ← ⟨NoPrepacking, HeuristicPrepacking,
2                ExactPrepacking, Finished⟩
3      L'_max ← (1 + ε'_B) max_B(H, k)
4      state ← NoPrepacking
5      while state ≠ Finished do
6          if state = NoPrepacking then
7              ⟨P_1, P_2⟩ ← ⟨∅, ∅⟩
8          else if state = HeuristicPrepacking then
9              ⟨P_1, P_2⟩ ← calculateHeuristicPrepacking(H, k, L'_max)
10         else if state = ExactPrepacking then
11             ⟨P_1, P_2⟩ ← calculateExactPrepacking(H, k, L'_max)
12         Π = {V_1, V_2} ← bisect(H, ε', P_1, P_2)        // calculate bisection with fixed vertices
13         if Π is deeply balanced then    // early restart: if bisection is not deeply balanced
14             Π_1 ← recursiveBisection(H_{V_1}, ε, k/2)
15             Π_2 ← recursiveBisection(H_{V_2}, ε, k/2)
16             Π' ← Π_1 ∪ Π_2
17             if Π' is balanced then          // late restart: if final partition is not balanced
18                 break
19         state ← nextState(state, STATES)                // switch to next state
       Output: k-way partition Π'
```

It is noteworthy that there are two points where the procedure can be restarted: First when testing the bisection for deep balance, which we call an *early* restart, and second when the resulting k-way partition is imbalanced, which we will call a *late* restart. One may ask why late restarts are necessary if our prepacking algorithm is capable of guaranteeing the deep balance of a bisection and therefore should be sufficient to find a balanced partition for the lower levels. And indeed our experimental results show that late restarts are very rare. However, there are some edge cases that can happen: The prepacking operates on the graph H and the initial partitioning on a coarsed graph H'. Due to the coarsening, the initial partitioning might not find a balanced bisection for H' even though one exists for H. Furthermore, the bin packing algorithm used by the prepacking is only an approximation algorithm and might not find a balanced packing even if one exists. We give an analysis of the impact of restarts on the running time in Appendix A.

5.3. The Balance Property

In order to develop a theoretical foundation for an exact prepacking algorithm, we first present a technique to give a fast estimate for the maximum block weight required for a partition. As it operates only on the vertex weight distribution, we will define it on a load balancing instance analogously to Section 4.1. Nonetheless, it is directly transferable to the corresponding hypergraph.

Given the sequence $L = \langle a_1, \ldots, a_n \rangle$ and k bins for an instance I, we will in addition to the definitions given in Section 2.4 assume that L is sorted in non-increasing order, i.e. $a_i \geq a_{i+1}$ for all $i \in [n-1]$. Furthermore, we assign a *fixed initial weight* w_j to every bin $j \in [k]$, which corresponds to fixed vertices that are already preassigned. The definition of the weight of a bin is adjusted to $w(B_j) \coloneqq w_j + \sum_{i \in B_j} a_i$ and the definitions of $w(I)$, $\overline{w}$ and $\mathrm{imb}(S)$ for a solution S likewise. For a bin B, we denote the set of those of the first i elements of L that are assigned to B by $B^{(i)} \coloneqq B \cap [i]$.

Definition 5.4 (AFD algorithm)**.** *Let $\mathcal{A}$ be an algorithm that calculates a solution for I. We call $\mathcal{A}$ an any fit decreasing (AFD) algorithm if it assigns the elements in L iteratively and an element i is assigned to a bin B only if either the resulting weight of B is at most the average bin weight, i.e. $w(B^{(i-1)}) + a_i \leq \overline{w}$, or B is a bin with minimum weight.*

Note that this definition corresponds closely to the definition of any fit algorithms for bin packing as described in Section 3.2.1. In fact, any of these rules for bin packing can be translated to a load balancing algorithm according to this definition and specifically the LPT algorithm corresponds to a worst fit decreasing (WFD) rule.

AFD algorithms have a specific property that we will use for an upper bound of the resulting imbalance. The total weight of the bins after the first i elements are assigned is $\sum_{j=1}^{k} w_j + \sum_{l=1}^{i} a_l$. Therefore, we can give a bound on the smallest bin after the assignment of i elements and conclude that if an element i is assigned to a bin B where $w(B^{(i-1)}) + a_i > \overline{w}$ we have

$$w(B^{(i-1)}) \leq \frac{1}{k} \left(\sum_{j=1}^{k} w_j + \sum_{l=1}^{i-1} a_l \right) \tag{5}$$

The specific value we use for the bound is the following:

Definition 5.5 (Bound)**.** *For a non-empty sequence $L = \langle a_1, \ldots, a_n \rangle$ in non-increasing order and a number of bins k we define*

$$h_k(L) \coloneqq \max_{i \in [n]} \left\{ a_i - \frac{1}{k} \sum_{l=i}^{n} a_l \right\}$$

An example of the calculation is illustrated in Figure 4. We will use this to give a bound on the imbalance of an AFD packing with k bins. There are already some basic properties we can observe, such as $h_1(L) = 0$ and $h_k(L) \geq a_1 - \overline{w}$, that are suitable for this purpose.

Theorem 5.6 (Bound for AFD algorithms)**.** *Let $\mathcal{A}$ be an AFD algorithm and S the solution calculated by $\mathcal{A}$ for an instance $I = (L, k)$. Then* $\mathrm{imb}(S) \leq \max(\{h_k(L)\} \cup \{w_j - \overline{w} \mid j \in [k]\})$. *That means h_k provides an upper bound to the imbalance of any AFD packing.*

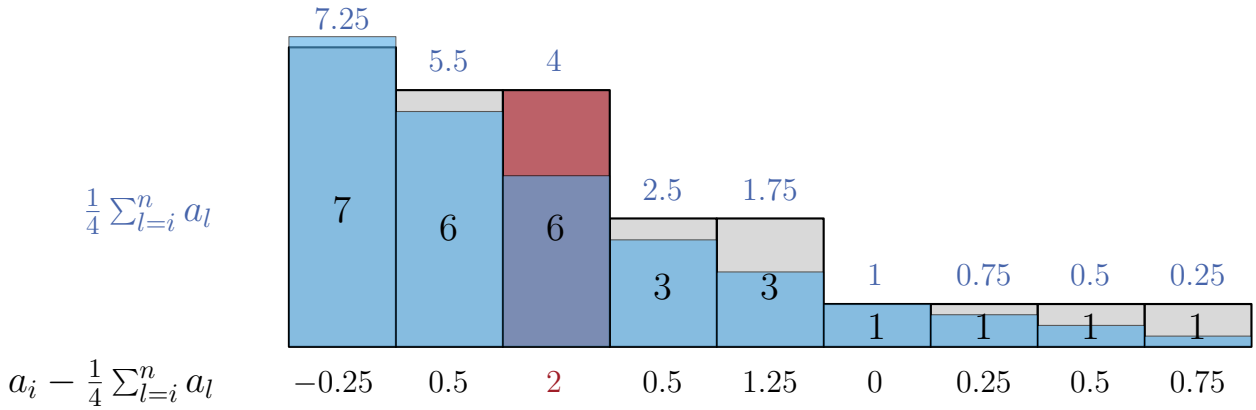

Figure 4: Illustration of the calculation of $h_k(L)$ for $k = 4$. The elements of L are displayed gray and the suffix sum is displayed blue. The maximum value is highlighted in red. The result is $h_4(L) = 2$.

Proof. Let B_j a maximum bin, i.e. $w(B_j) = \max(S)$. If $B_j = \emptyset$ then $\text{imb}(S) = w(B_j) - \overline{w} = w_j - \overline{w}$ by definition. Otherwise, consider any element i in B_j. As $\mathcal{A}$ is an AFD algorithm, either $w(B_j^{(i)}) = a_i + w(B_j^{(i-1)}) \leq \overline{w}$ or we apply Property 5 and get

$$w(B_j^{(i)}) = a_i + w(B_j^{(i-1)}) \leq a_i + \frac{1}{k}\left(\sum_{j=1}^{k} w_j + \sum_{l=1}^{i-1} a_l\right) = a_i + \overline{w} - \frac{1}{k}\sum_{l=i}^{n} a_l \leq \overline{w} + h_k(L)$$

Thus $w(B_j) \leq \overline{w} + h_k(L)$ and the result follows from $\text{imb}(S) = w(B_j) - \overline{w} \leq h_k(L)$. □

It should be noted that the bound is too pessimistic for some instances, e.g. for an instance with k bins and k elements of uniform weight a we have $h_k(L) = (1-\frac{1}{k})a$ but an AFD algorithm finds a packing with zero imbalance. For other instances, e.g. $k+1$ elements of uniform weight, the bound is precise. It can be observed that the bound tends to be smaller for smaller weights. We formalize the notion of a smaller sequence:

Definition 5.7 (Smaller sequence)**.** *Let $L_1 = \langle a_1, \ldots, a_n\rangle$ and $L_2 = \langle b_1, \ldots, b_m\rangle$ be sequences in $\mathbb{R}_{>0}$. We call L_2 smaller than L_1 and write $L_2 \leq L_1$ if each element is smaller than its corresponding element in L_1. More formally, if $b_i \leq a_i$ for all $i \in [\min\{m, n\}]$.*

As a side node, this relation does not define a partial order if sequences of different sizes are allowed, as it is neither antisymmetric nor transitive.

In the following, let $L_1 = \langle a_1, \ldots, a_n\rangle$ and $L_2 = \langle b_1, \ldots, b_m\rangle$ be sequences in non-increasing order. If one sequence is smaller than the other, we can give an estimate on the bounds with the following lemmas:

Lemma 5.8. *If $L_2 \leq L_1$ and $w(L_2) \leq w(L_1)$ then $h_k(L_2) + \overline{w}(L_2) \leq h_k(L_1) + \overline{w}(L_1)$.*

Proof. We show $b_i - \frac{1}{k}\sum_{l=i}^{m} b_l + \overline{w}(L_2) \leq h_k(L_1) + \overline{w}(L_1)$ for every $i = 1, \ldots m$. This is equivalent to the above statement because of $h_k(L_2) = \max\left\{b_i - \frac{1}{k}\sum_{l=i}^{m} b_l \mid i \in [m]\right\}$. Let $i \leq n$. Then

$$\left(b_i - \frac{1}{k}\sum_{l=i}^{m} b_l\right) + \overline{w}(L_2) = b_i + \frac{1}{k}\sum_{l=1}^{i-1} b_l \leq a_i + \frac{1}{k}\sum_{l=1}^{i-1} a_l = \left(a_i - \frac{1}{k}\sum_{l=i}^{n} a_l\right) + \overline{w}(L_1) \leq h_k(L_1) + \overline{w}(L_1)$$

If $n < i \leq m$, we have

$$\left(b_i - \frac{1}{k}\sum_{l=i}^{m} b_l\right) + \overline{w}(L_2) \leq \left(1 - \frac{1}{k}\right) b_i + \overline{w}(L_2) \leq \left(1 - \frac{1}{k}\right) a_n + \overline{w}(L_1) \leq h_k(L_1) + \overline{w}(L_1)$$

□

Lemma 5.9. *If $L_2 \leq L_1$ and $w(L_2) \geq w(L_1)$ then $h_k(L_2) \leq h_k(L_1)$.*

Proof. The proof works analogously to Lemma 5.8. Let $i \leq n$. Then

$$b_i - \frac{1}{k}\sum_{l=i}^{m} b_l = b_i + \frac{1}{k}\left(\sum_{l=1}^{i-1} b_l - w(L_2)\right) \leq a_i + \frac{1}{k}\left(\sum_{l=1}^{i-1} a_l - w(L_1)\right) = a_i - \frac{1}{k}\sum_{l=i}^{n} a_l \leq h_k(L_1)$$

If $n < i \leq m$, we have

$$b_i - \frac{1}{k}\sum_{l=i}^{m} b_l \leq \left(1 - \frac{1}{k}\right) b_i \leq \left(1 - \frac{1}{k}\right) a_n \leq h_k(L_1)$$

□

Together with Theorem 5.6, we will use these results to construct a property that can ensure the deep balance of a partition. Now we consider the recursive case, where a packing with 2 blocks is computed and each block is again recursively divided into $\frac{k}{2}$ subblocks (respective bins). To ensure the deep balance, we use a prepacking as explained in Section 5.1.

Definition 5.10 (Prepacking)**.** *Let L be a sequence with n elements and $m \leq n$. A prepacking of m elements of L is a partition of $[m]$ into disjoint subsets $\Psi = \{P_1, P_2\}$ such that $P_1 \cup P_2 = [m]$.*

Definition 5.11 (Completion)**.** *Let L be a sequence and $\Psi = \{P_1, P_2\}$ be a prepacking of L. A packing $\Omega = \{B_1, B_2\}$ of L with $B_1 \cup B_2 = [n]$ is a completion of Ψ if $P_1 \subseteq B_1$ and $P_2 \subseteq B_2$.*

A completion corresponds in the hypergraph context to the current bisection. In the following, we introduce the parameter u, which corresponds to the upper allowed block weight of the current bisection, and the bin capacity b, which corresponds to the upper allowed block weight L_{max} of the k-way partition. Basically, we want to construct a prepacking that ensures that any possible completion within the allowed block weight u is deeply balanced. Then, every such completion yields a deeply balanced bisection and therefore we can apply algorithms that optimize the objective function for calculating the bisection.

Definition 5.12 (Sufficient balance)**.** *Let $L = \langle a_1, \ldots, a_n \rangle$ be a sequence, k be the number of bins and b be the bin capacity. We call a prepacking Ψ of L sufficiently balanced for an upper allowed block weight u if every completion Ω of Ψ that satisfies $\max(\Omega) \leq u$ is deeply balanced with respect to b.*

Recall first, $\max(\Omega) \leq u$ is equivalent to $w(B_i) \leq u$ for every block $B_i \in \Omega$, and second, Ω is deeply balanced means that for every block $B_i \in \Omega$ there is a subpacking Ω_i of B_i to $\frac{k}{2}$ bins with $\max(\Omega_i) \leq b$. If a prepacking is sufficiently balanced, a completion of the prepacking must only assert that the blocks are balanced with upper block weight u.

Unfortunately, it is clear that sufficient balance can not be verified directly in an efficient way. Instead, we present a property for prepackings that is based on the upper bound defined in 5.6 and implies sufficient balance.

Definition 5.13 (Balance property)**.** *Let $L = \langle a_1, \ldots, a_n \rangle$ be a sequence in non-increasing order, k be the number of bins, u be the upper allowed block weight and b be the bin capacity. Let Ψ be a prepacking of m elements. For an element $i > m$ we denote by $L_i := \langle a_{m+1}, \ldots, a_i \rangle$ the elements up to i that are not prepacked.*

Let $T := \{i \in \{m+1, \ldots, n\} \mid \max(\Psi) + w(L_i) \leq u\}$. We say that Ψ satisfies the balance property with respect to u if $T \neq \emptyset$ and for $t := \max(T)$ the following conditions hold:

(i) Ψ *is deeply balanced*
(ii) $\frac{1}{k/2}u + h_{k/2}(L_t) \leq b$

Condition (i) is obviously necessary: If Ψ already isn't deeply balanced, no completion of it will be deeply balanced. For Condition (ii), we use L_t to construct a worst case bound for the elements that are packed to the block of Ψ with maximum weight, where the definition of T ensures that at most the maximum block weight is used. Then, h_k gives a bound that is within the bin capacity b for the weight of the maximum bin of an AFD packing. Now we will show that a prepacking which satisfies this property is sufficiently balanced.

Theorem 5.14 (Balance property implies sufficient balance)**.** *Let Ψ be a prepacking of m elements of a sequence L, k be the number of bins, u be the upper allowed block weight and b be the bin capacity. If Ψ satisfies the balance property with respect to u then Ψ is sufficiently balanced for u.*

Proof. For convenience, we use $k' := \frac{k}{2}$. To show that Ψ is sufficiently balanced, we need to consider any completion Ω of Ψ that satisfies $\max(\Omega) \leq u$ and show that Ω is deeply balanced with respect to b. Let B_j be a block in Ω and $P_j \subseteq B_j$ the corresponding block in Ψ. Then

$$w(B_j) \leq \max(\Omega) \leq u \tag{6}$$

Consider a balanced subpacking $\Psi_j = \{P'_1, \ldots, P'_{k'}\}$ of P_j as given by Condition (i), i.e. $\max(\Psi_j) \leq b$. We define $w_l := w(P'_l)$ as an initial weight for bin l. Then

$$w_l \leq \max(\Psi_j) \leq b \text{ for } l \in [k'] \tag{7}$$

We will construct a subpacking $\Omega_j = \{B'_1, \ldots, B'_{k'}\}$ of B_j that is based on Ψ_j, i.e. $P'_l \subseteq B'_l$ for all $l \in [k']$. Let $I = B_j \setminus P_j$ be the set of remaining elements in the block and L_I the non-increasing sequence containing the elements of I. We have $I \subseteq \{m+1, \ldots, n\}$ and thus $L_I \leq L_t$ as L is non-increasing.

Claim. $\frac{1}{k'}w(B_j) + h_{k'}(L_I) \leq \frac{1}{k'}u + h_{k'}(L_t)$

If $w(L_I) \leq w(L_t)$, we apply Lemma 5.8 and get $h_{k'}(L_I) + \frac{1}{k'}w(L_I) \leq h_{k'}(L_t) + \frac{1}{k'}w(L_t)$. Using the definition of T we have

$$\frac{1}{k'}w(B_j)+h_{k'}(L_I) = \frac{1}{k'}(w(P_j)+w(L_I))+h_{k'}(L_I) \leq \frac{1}{k'}(\max(\Psi)+w(L_t))+h_{k'}(L_t) \leq \frac{1}{k'}u+h_{k'}(L_t)$$

If $w(L_I) > w(L_t)$, we apply Lemma 5.9 and get $h_{k'}(L_I) \leq h_{k'}(L_t)$. Thus

$$\frac{1}{k'}w(B_j) + h_{k'}(L_I) \leq^{(6)} \frac{1}{k'}u + h_{k'}(L_t)$$

This proves the Claim.

B_j corresponds to a load balancing instance with preassigned weights $w_l = w(P'_l)$, elements I and average weight $\frac{1}{k'}w(B_j)$. By Theorem 5.6, there is a packing Ω_j of B_j with $\text{imb}(\Omega_j) \leq \max\left(\{h_{k'}(L_I)\} \cup \{w_l - \frac{1}{k'}w(B_j) \mid l \in [k']\}\right)$. With the claim, (7) and Condition (ii) we conclude

$$\max(\Omega_j) \leq \max\left\{\frac{1}{k'}w(B_j) + h_{k'}(L_I), \max_{l\in[k']} w_l\right\} \leq \max\left\{\frac{1}{k'}u + h_{k'}(L_t), b\right\} \leq b$$

□

We want to add two remarks concerning the balance property: First, the sufficient balance of a prepacking is useless if u is too small, as in this case it is unlikely that a completion Ω with $\max(\Omega) \leq u$ even exists. Therefore it is necessary to ensure sufficient balance for at least $u \geq \frac{1}{k}w(L)$ so that such a completion is possible. Second, the proof does not use that t is defined as the maximum of T. Indeed any element of T would be sufficient, but the maximum is the best choice to satisfy Condition (ii) (see Lemma 5.9).

5.4. An Exact Prepacking Algorithm

The goal for this section is to construct an algorithm that uses the balance property for efficiently calculating a sufficiently balanced prepacking. Basically, we can use a bin packing algorithm $\mathcal{A}$ to calculate a prepacking $\Psi = \{P_1, P_2\}$ of a given subsequence of the elements. If Ψ is deeply balanced and we can satisfy the second condition of the balance property, then Ψ is already sufficiently balanced. Thus our resulting algorithm is as effective as $\mathcal{A}$ for ensuring the deep balance of the bisection. We show how such an algorithm $\mathcal{A}$ can be constructed in Section 5.6.

As explained previously, we do not only want to ensure the deep balance of the bisection but also retain the possibility for optimizing the cut. Therefore, the algorithms optimization objective is the size of the prepacking.

Definition 5.15 (Prepacking)**.** *Given a sequence L, a number of bins k, an upper block weight u and a bin capacity b, the prepacking problem is to find a prepacking $\{P_1, P_2\}$ of L which is sufficiently balanced for u such that $|P_1| + |P_2|$ is minimized.*

Algorithm 2 shows a basic implementation of this idea, using the balance property to ensure that the calculated prepacking is sufficiently balanced. Given a sequence L in non-increasing order, we iteratively increase the number m of prepacked elements. In each step, $\mathcal{A}$ is applied to get a prepacking $\{P_1, P_2\}$ of the first m elements. We calculate a maximal $t > m$ with $\max\{w(P_1), w(P_2)\} + w(L_t) \leq u$. The prepacking is accepted if the balance property is satisfied, i.e. such a t exists and $\frac{1}{k/2}u + h_{k/2}(L_t) \leq b$ holds. An example calculation is illustrated in Figure 5.

Algorithm 2: Prepacking (naive)

Input: Ordered sequence $L = \langle a_1, \ldots, a_n \rangle$, number of bins k, block weight u, bin weight b
Require: $w(L) \leq 2u \leq k \cdot b$

```
1 ⟨P_1, P_2⟩ ← ⟨∅, ∅⟩
2 for m = 0, ..., n do
3     ⟨P_1, P_2⟩ ← A(⟨a_1, ..., a_m⟩, k)          // bin packing algorithm
4     T ← {i ∈ {m+1, ..., n} | max(Ψ) + w(L_i) ≤ u}
5     if T ≠ ∅ then
6         t ← max(T)                               // determine t
7         L_t ← ⟨a_{m+1}, ..., a_t⟩
8         if (1/(k/2))u + h_{k/2}(L_t) ≤ b then    // calculate upper bound
9             break
```

Output: P_1, P_2

The outer for loop has at most $n+1$ passes, with each iteration applying $\mathcal{A}$ and calculating t and $h_k(L_t)$. Using a direct approach, both can be calculated in $\mathcal{O}(n)$ by iterating over the elements while retaining the current weight sum (in reverse order for $h_k(L_t)$). Thus the algorithm has a running time of $\mathcal{O}(nT(\mathcal{A}) + n^2)$. Independent on $\mathcal{A}$ the resulting runtime is at least quadratic, which is not satisfactory for an algorithm that is intended to be a fast approximation algorithm. For each operation of the algorithm we will present an improved method of calculation, enabling us to achieve a near-linear runtime.

Disassembling the bin packing algorithm. A common class of approximation algorithms for bin packing are *on-line* algorithms such as first fit (see Section 3.2.1). These assign the

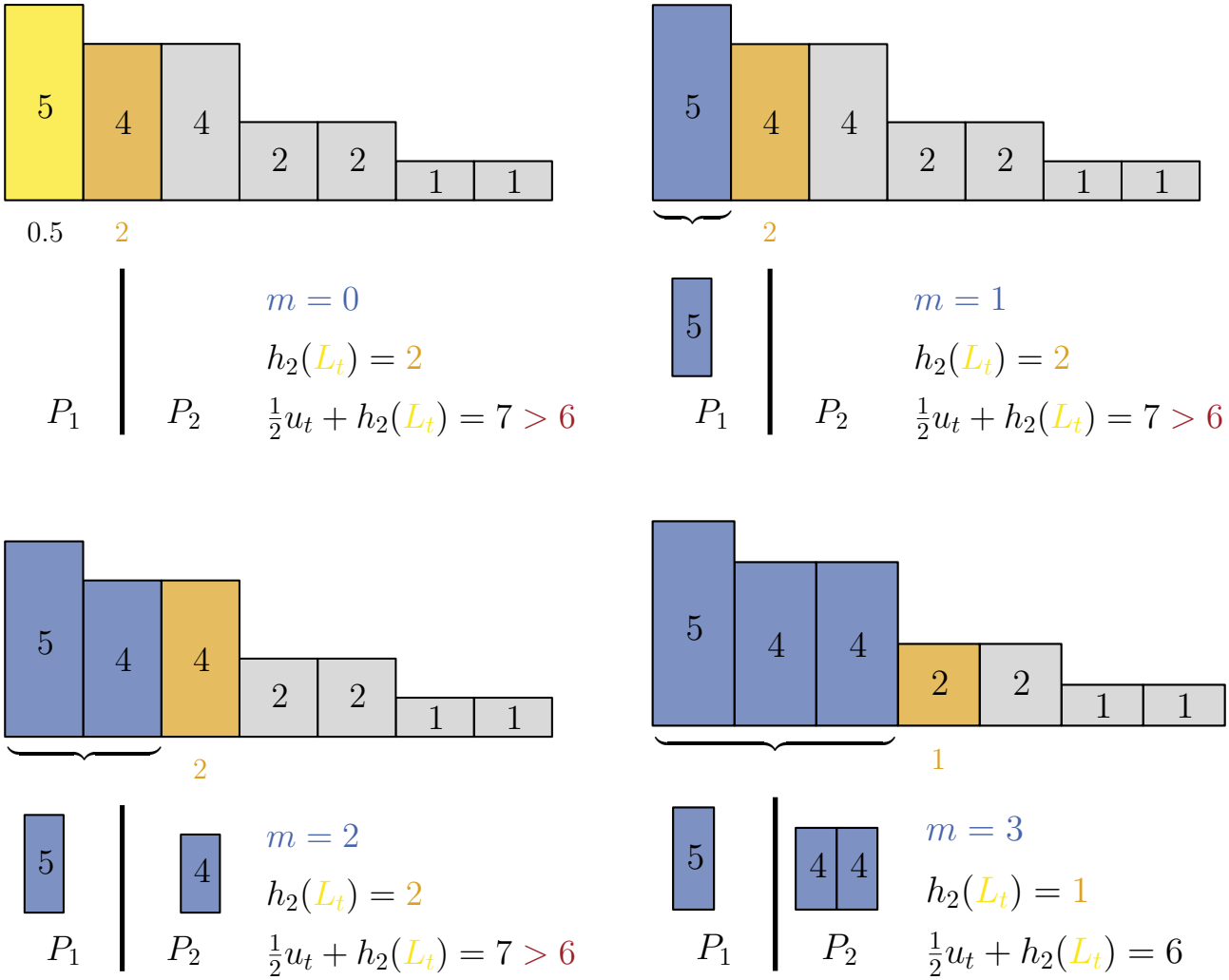

Figure 5: Illustration of Algorithm 2 for the displayed sequence and input parameters $k = 4$, $u = 10$ and $b = 6$. Prepacked elements are highlighted blue. The subsequence L_t is highlighted yellow with the maximum element for the bound in orange. A worst fit (WF) rule is used for the bin packing algorithm that calculates $\{P_1, P_2\}$. The prepacking with 3 elements is accepted.

elements to bins separately and don't move any element after assigning it to a bin. We can use this property to calculate only the bin for the recently added element in every iteration, as the previous elements remain in their respective bins.

We require the base algorithm to consist of two procedures, with the runtime of both in $\mathcal{O}(k \log(k))$ (the logarithmic factor allows for sorting of the bins):

- *insertElement*, which takes the current bins and the element to be added as input and adds the element to one of the bins.
- *assignPartitions*, which takes the current bins, assigns them to two blocks and outputs those.

(Strictly speaking, the procedures should also take the allowed bin weight b or the allowed partition weight u as input, respectively. We omit this for improved conciseness.)

To give an example, we shortly describe the procedures for the first fit decreasing (FFD) algorithm. *insertElement* adds the element to the first bin with enough space available or to the smallest bin, if no bin has enough space. *assignPartitions* sorts the bins and traverses them in decreasing order. The elements of the bin are assigned to the first block if enough space is available and less than $\frac{k}{2}$ bins have been added to the block yet, otherwise to the second block.

Determining t. The second costly operation is determining the value for t, which can be found by iterating over the remaining elements. However, this can be improved by precalculating the prefix sum for all elements and use a binary search based on the sums. The *binarySearch* procedure takes a sequence $S = \langle s_1, \ldots, s_n \rangle$ in increasing order and a searched value v as input. The output is the maximum value for i s.t. $s_i \leq v$ and is calculated in only $\mathcal{O}(\log(n))$ time.

Additionally, we introduce a case distinction on $t = n$: In this case L_t already contains all elements that are not prepacked and thus any completion of the calculated prepacking has an upper block weight of at most $\hat{u} = \max\{w(P_1), w(P_2)\} + w(L_t)$. Therefore it is sufficient to verify the balance property for $\hat{u}$ which makes the condition for accepting the prepacking more likely to hold, leading to fewer prepacked elements in some cases.

Upper bound calculation. The remaining operation to improve is the calculation of $h_{k/2}(L_t)$. The bound is defined as the maximum over a range of values: $h_k(L) = \max\{a_i - \frac{1}{k}\sum_{l=i}^{n} a_l \mid i \in [n]\}$. Thus we can transform the formula for the bound of a subsequence $L' = \langle a_m, \ldots, a_j \rangle$:

$$h_k(L') = \max_{i=m,\ldots,j} \left\{ a_i - \frac{1}{k} \sum_{l=i}^{j} a_l \right\} = \max_{i=m,\ldots,j} \left\{ a_i - \frac{1}{k} \sum_{l=i}^{n} a_l \right\} + \frac{1}{k} \sum_{l=j+1}^{n} a_l$$

.

Using this, we can calculate the bound of a subsequence by precalculating $a_i - \frac{1}{k/2}\sum_{l=i}^{n} a_l$ for every i and determining the maximum over the subsequence. For the maximum we use a segment tree as described in Section 2.5. The segment tree can be built with the elements $a_i - \frac{1}{k/2}\sum_{l=i}^{n} a_l$ in $\mathcal{O}(n)$ time. Then, queries for the maximum of a subsequence can be run efficiently in $\mathcal{O}(\log(n))$ time.

Having improved the runtime for each operation, we explain in the following a last optimization that can be applied to improve the result quality.

Optimizing the allowed block weight. We calculate a prepacking that is sufficiently balanced for the given upper allowed block weight u. However, the prepacking could also be sufficiently balanced for a higher block weight. Thus, after having determined a prepacking in the main

loop of the algorithm we will calculate a maximum value u' such that the prepacking fulfills the balance property with respect to u' and output u' in addition to the resulting partition. This is done by calculating a maximum value for t such that $\frac{1}{k/2}(\max\{w(P_1), w(P_2)\} + w(L_t)) + h_{k/2}(L_t) \leq b$. Then, we can even use $u' := \frac{k}{2}(b - h_{k/2}(L_t))$ as upper weight, satisfying the conditions of the balance property exactly.

The calculation can be implemented in $\mathcal{O}(n)$ as we iterate over at most n elements, updating the required variables and sums per iteration step. Specifically, the bound for the current sequence can be updated in $\mathcal{O}(1)$ by using $h_k(\langle L, a_{n+1}\rangle) = \max\{h_k(L) - \frac{1}{k}a_{n+1}, (1 - \frac{1}{k})a_{n+1}\}$.

Improved algorithm. Combining the explained optimizations, we present an algorithm with improved runtime and result quality: During initialization a sequence $S = \langle a_1, a_1+a_2, \ldots, \sum_{i=1}^{n} a_i\rangle$ of prefix sums is calculated and a segment tree T is built based on the values $a_i - \frac{1}{k/2}\sum_{l=i}^{n} a_l$, which are required for the calculation of $h_{k/2}$. Both operations have $\mathcal{O}(n)$ running time. As before, we iteratively increase the number m of prepacked elements. In each step, *insertElement* is called to insert the current element into a bin and *assignPartitions* to calculate $\{P_1, P_2\}$. A maximum t with $\max\{w(P_1), w(P_2)\} + w(L_t) \leq u$ is found in $\mathcal{O}(\log(n))$ by applying *binarySearch* to S and the upper bound for the prefix sum of t. If $t > m$ we calculate $h_{k/2}(L_t)$ in $\mathcal{O}(\log(n))$ by using a maximum query over the range $[m+1, t]$ on T and adding $\frac{1}{k/2}\sum_{i=t+1}^{n} a_i$ to the result. The prepacking is accepted if either $\frac{1}{k/2}u + h_{k/2}(L_t) \leq b$ or $t = n$ and $\frac{1}{k/2}(\max\{w(P_1), w(P_2)\} + w(L_t)) + h_{k/2}(L_t) \leq b$. Afterwards, a possibly increased upper block weight u' is calculated with an iteration over the remaining elements, starting with $m+1$. This works in $\mathcal{O}(n)$ by updating the current weight and $h_{k/2}(L_t)$ as described in the previous paragraph. A pseudocode description is given in Algorithm 3.

All precalculations and postcalculations can be implemented in $\mathcal{O}(n)$ as explained. The main loop has at most $n + 1$ passes, with only the binary search, the query on the segment tree and the *insertElement* and *assignPartitions* procedures, which are in $\mathcal{O}(k \log(k))$, not running in constant time. This results in an overall running time of $\mathcal{O}(nk \log(k) + n \log(n))$ for the complete algorithm. Note that k is usually much smaller than n.

The upper allowed block weight u is given as a parameter to the algorithm with possible values between $\frac{1}{2}w(L)$ and $\frac{k \cdot b}{2}$. Therefore, to integrate the algorithm into our balancing strategy (see Section 5.2) we need a method for choosing u appropriately. The fundamental trade-off here is that a high maximum block weight (which increases the solution space for finding a bisection with a good cut) requires a prepacking with a higher number of vertices (which reduces the solution space), and vice versa. But there is also another trade-off involved in k-way partitioning via recursive bisection: A block with high weight reduces the solution space for the lower recursion levels and for the refinement, which must preserve the balance constraint. To handle this issue, KaHyPar already uses the adaptive imbalance parameter ε' (compare Equation 1). For our implementation we use the same approach, i.e. we choose $u = (1 + \varepsilon')\left\lceil \frac{c(V)}{2} \right\rceil$. However, in general it seems preferable to choose a value for u that is not too close to $\frac{k \cdot b}{2}$ in order to keep the number of prepacked vertices low. This is analyzed in Appendix B.

Directions for generalization. There are two dimensions where a generalization of the prepacking problem is possible. First, we could apply it to k-way partitioning instead of only 2-way partitioning. This is not required for a partitioner based on recursive bisection: As the name indicates, only bisections (i.e. 2-way partitions) are calculated. Second, instead of assuming an equal number of k bins for every block, we could assign different numbers of bins for every block. This case actually appears when recursive bisection is applied to calculate a k-way

Algorithm 3: Prepacking (improved)

Input: Ordered sequence $L = \langle a_1, \ldots, a_n \rangle$, number of bins k, block weight u, bin weight b
Require: $w(L) \leq 2u \leq k \cdot b$

```
/* initialization */
⟨B_1, ..., B_k⟩ ← ⟨∅, ..., ∅⟩
⟨P_1, P_2⟩ ← ⟨∅, ∅⟩
m ← 0
⟨s_1, ..., s_n⟩ ← ⟨a_1, ..., Σ_{i=1}^{n} a_i⟩
L_H ← ⟨a_1 − (1/(k/2)) Σ_{i=1}^{n} a_i, ..., (1 − 1/(k/2)) a_n⟩
T ← SegmentTree()
T.build(L_H)
/* calculate P_1, P_2 and m */
while m ≤ n do
    s ← u + min{w(P_1), w(P_2)}
    t ← binarySearch(⟨s_1, ..., s_n⟩, s)                    // determine t
    if t > m then
        û ← { u,                                  t < n
            { max{w(P_1), w(P_2)} + s_t − s_m,    t = n
        h ← T.query(m + 1, t) + (1/(k/2))(s_n − s_t)        // calculate upper bound
        if (1/(k/2)) û + h ≤ b then
            break
    m ← m + 1
    insertElement(⟨B_1, ..., B_k⟩, a_m)                     // bin packing algorithm
    ⟨P_1, P_2⟩ ← assignPartitions(⟨B_1, ..., B_k⟩)
/* calculate maximum block weight u' */
u_t ← max{w(P_1), w(P_2)}
h_t ← 0
t ← m + 1
while t ≤ n ∧ (1/(k/2))(u_t + a_t) + h_t ≤ b do
    u_t ← u_t + a_t
    h_t ← max{h_t − (1/(k/2)) a_t, (1 − 1/(k/2)) a_t}
    t ← t + 1
u' ← (k/2)(b − h_t)
```

Output: P_1, P_2, u'

partition where k is not a power of 2. In this case it is necessary to calculate a bisection with blocks of unequal size [38] and our prepacking algorithm should also be applicable.

The k-way case can be accommodated by small adjustments to the algorithm. For improved clarity, we only considered the bisection case, but the definition of deep balance and the balance property are directly generalizable to the case with k blocks. We use k' instead of $\frac{k}{2}$ for the number of bins per block. Basically, it is only necessary to exchange the parameters for the number of blocks in the algorithm, use multiple blocks for the prepacking and adjust the according calculations. However, an interesting aspect is that there is an additional optimization possibility for the result quality (i.e. size of the prepacking) which is meaningless for the 2-way case: In the case of $T = \emptyset$, the prepacking is considered not sufficient as the balance property is not applicable. But in some cases it is still possible to verify the sufficient balance. Note that $T = \emptyset$ means that the next element does not fit in the current maximum block and thus in the 2-way case must be packed to the other block anyway, therefore such an optimization is rather meaningless. But the prepacking is still sufficiently balanced if the following holds for the currently prepacked blocks $P_1, \ldots, P_k$: Every block P_j where the next element does not fit in P_j satisfies $\frac{1}{k'}u + (1 - \frac{1}{k'})(u - w(P_j)) \leq b$. The maximum block P_i where the next element fits satisfies the usual constraint, i.e. $\frac{1}{k'}u + h_{k'}(L_t) \leq b$.

If different numbers of bins are given for each block, the balance property is not applicable and we do not even have a fixed upper block weight u. Instead, we will assume that there are two blocks with k_1 respective k_2 bins and upper allowed block weights u_1 and u_2. There are two possible approaches similar to the balance property: We can verify the corresponding constraint for both blocks, i.e. $\frac{1}{k_1}u_1 + h_{k_1}(L_{t_1}) \leq b$ and $\frac{1}{k_2}u_2 + h_{k_2}(L_{t_2}) \leq b$, where t_1 and t_2 are calculated for both blocks separately. Or we can use a coarser estimate: Let $k_{min} := \min\{k_1, k_2\}$ and $k_{max} := \max\{k_1, k_2\}$. It is sufficient to verify $\frac{1}{k_{min}}u_i + h_{k_{max}}(L_t) \leq b$ for the block with less free space, i.e. where $u_i - w(P_i)$ is minimal. The second approach can have worse results than necessary in some cases. Nonetheless, we use it in our implementation for KaHyPar, as it is directly applicable also in the case of equal blocks, without an unnecessary calculation for the second block. The difference in result quality is probably negligible for most practical cases.

5.5. Heuristic Approaches for Prepacking

Apart from calculating a prepacking that can guarantee the deep balance of a bisection, there is also another approach to the basic idea of prepacking: We can prepack a smaller number of vertices which does not guarantee a sufficiently balanced prepacking. Instead, there should be at least a high likelihood that a completion of the prepacking is deeply balanced. Then, hopefully the resulting bisection is still deeply balanced, but has a better cut compared to the exact prepacking algorithm from Section 5.4, as the solution space is less restricted. Our experimental results indicate that such a heuristic approach works well in many cases. As we will always use a sequence $L = \langle a_1, \ldots, a_n \rangle$ in non-increasing order and prepack the first elements, we can consider both approaches as different rules to determine a number m of vertices to be prepacked for given parameters k, u and b (defined as previously). We will present two heuristics which also use the bound h_k to estimate a value for m.

Both try to determine a bound such that the bisection worsens the required maximum bin weight not more than allowed. Formally, we denote by b' the maximum bin weight required by a load balancing algorithm that is applied to L with k bins. (Note that b' corresponds to $\max_B(H, k)$ while b corresponds to L'_{max} for our balancing strategy presented in Section 5.2).

For the first heuristic, we define m_1 as the minimum value such that $b' + h_k(L') \leq b$ for $L' := \langle a_{m_1+1}, \ldots, a_n \rangle$. The idea is to prepack enough elements such that at least the remaining subsequence can be packed easily into the k bins. The calculation can be efficiently implemented

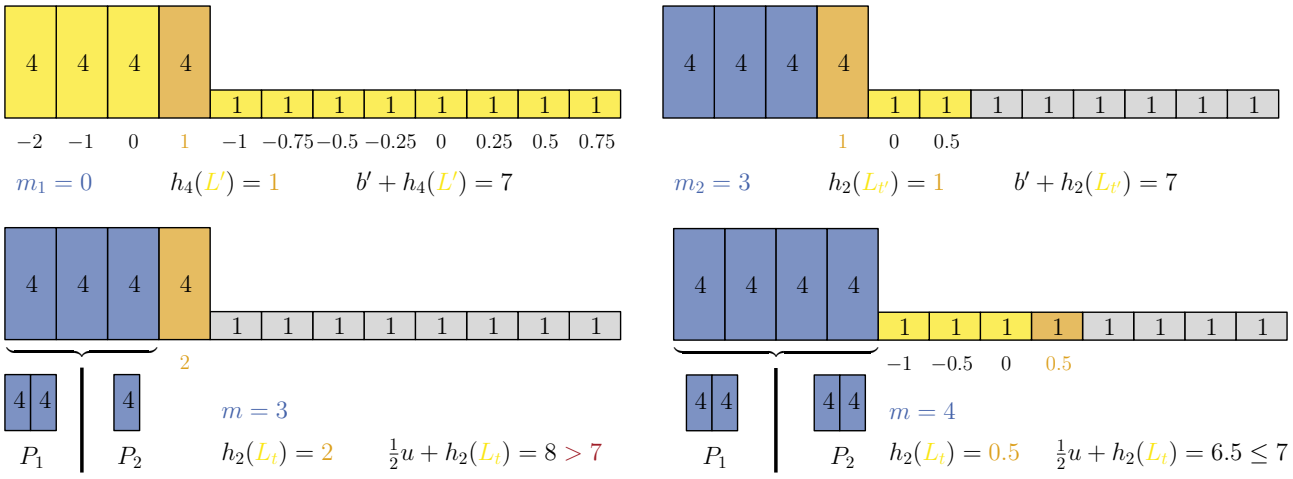

Figure 6: Illustration of two heuristic approaches (above) and the exact prepacking algorithm (below) for the same problem instance with $k = 4$, $u = 12$ and $b = 7$. Prepacked elements are highlighted blue. The range where h_k is applied is highlighted yellow with the maximum element in orange. The left prepacking is not accepted.

in $\mathcal{O}(n)$ by iterating over L in reverse order, calculating $a_i - \frac{1}{k}\sum_{l=i}^{n} a_l$ for each element i and stopping when this value exceeds $b - b'$.

For the second heuristic, we define t' for a given m as the maximum element such that the subsequence $L_{t'} = \langle m+1, \ldots, a_{t'} \rangle$ has at most half the weight of the remaining elements, i.e. $w(L_{t'}) \leq \frac{1}{2} w(L')$ with $L' := \langle a_{m+1}, \ldots, a_n \rangle$. m_2 is defined as the minimum value such that $b' + h_{k/2}(L_{t'}) \leq b$. The idea is to use the simplified assumption that the elements of L' are distributed to both blocks of the bisection with equal weight. Then, we consider the worst case for this scenario, where all heavy elements are assigned to the same block. The calculation can be implemented in $\mathcal{O}(n \log(n))$ in a similar way to the prepacking algorithm, using a segment tree to calculate $h_{k/2}(L_{t'})$. Additionally, we can exploit that t' increases monotonically with m. This is possible by increasing t' as necessary per iteration step instead of using a binary search.

To demonstrate that this approach does not guarantee a deeply balanced bisection, we give a counterexample where for both m_1 and m_2 the calculated prepacking is not sufficiently balanced. Consider a sequence L containing four elements of weight 4 and eight elements of weight 1 with $k = 4$, $u = 12$ and $b = 7$. Note that $b' = 6$ as the weights $\langle 4, 1, 1 \rangle$ can be packed to each one of the 4 bins and that $\frac{1}{2} w(L) \leq u \leq k \cdot b$ holds. As illustrated in Figure 6 we have $m_1 = 0$ and $m_2 = 3$. For any $m \leq 3$, a completion of the prepacking can pack three elements of weight 4 into the same block within the allowed block weight 12. Because the block has only two bins of capacity 7, no balanced subpartition exists. Thus the prepacking is not sufficiently balanced.

Concerning the strictness of the definitions we have approximately $m_1 \lesssim m_2 \lesssim m$, where m is the value calculated by the exact prepacking algorithm, but none of the estimates is strict due to some edge cases. Overall, using a heuristic approach offers possibilities for a better cut. In Section 6.3 we compare both heuristic approaches experimentally.

5.6. Bin Packing Initial Partitioner

Finding a balanced k-way partition is similar to the bin packing problem, thus it is natural to use a bin packing algorithm as a foundation for an initial partition algorithm that optimizes the deep balance of a bisection. However, just applying a bin packing algorithm that uses the blocks of the bisection as bins is not sufficient. Such an algorithm would not be aware of the

deep balance of the partition as it depends on the recursive subdivision of the blocks, but not on the balance of the blocks themselves. Thus we need a more refined approach.

We will introduce a concept that we call the *two-level* approach. It is also based on a bin packing algorithm, but instead of applying this algorithm directly, we use two passes to ensure the deep balance of the bisection. As before, we only consider bisections, but the concept can be generalized to partitions with more blocks. The basic idea is that in the first pass, a balanced k-way partition is calculated. In the second pass, the resulting subblocks are bin packed into the blocks of the bisection based on their total weight. This allows us to ensure the balance of the final k-way partition just as well as if it is calculated directly with the bin packing algorithm.

Lemma 5.16 (Two-Level Approach)**.** *Let $H = (V, E, c, \omega)$ be a hypergraph, k be the number of blocks and L_{max} be the upper allowed block weight. Further let $\Pi = \{V_1, V_2\}$ be a bisection of H and $\Pi' = \{B_1, \ldots, B_k\}$ be a k-way partition of H with $c(B_i) \leq L_{max}$ for all $i \in [k]$. If V_1 is the union of exactly $\frac{k}{2}$ blocks of Π' then Π is deeply balanced with respect to L_{max}.*

Proof. Note that because V_1 is the union of exactly $\frac{k}{2}$ blocks of Π', this also holds for V_2 as $V_2 = V \setminus V_1$. The deep balance is derived directly: For V_i with $i \in \{1, 2\}$ let $\mathcal{B}$ be the set of $\frac{k}{2}$ blocks with $\bigcup_{B \in \mathcal{B}} B = V_i$. Then, $\mathcal{B}$ is a subpartition of V_i with $c(B) \leq L_{max}$ for all $B \in \mathcal{B}$. □

In order to formulate an algorithm for this approach, we will use a bin packing algorithm $\mathcal{A}$ that assigns a sequence of elements to a fixed number of bins. An important aspect is that it must be asserted that an equal number of subblocks is assigned to both blocks of the bisection. As we use $\mathcal{A}$ for both levels, this makes an additional requirement necessary: $\mathcal{A}$ must support specifying a maximum number of elements allowed per bin. Note that for on-line bin packing algorithms (see Section 3.2.1) this requirement is easily implemented by only considering the bins which have not reached the maximum number of elements when inserting an element.

Our algorithm works as follows: At the first level, we use $\mathcal{A}$ to pack the nodes of the hypergraph into k bins based on their weight. At the second level, the resulting bins are packed into the two blocks by applying $\mathcal{A}$ based on the total weight of the bins. Here, we additionally require that at most $\frac{k}{2}$ bins can be packed into the same block. The result is the bisection consisting of all elements of the assigned bins for each block.

For an efficient implementation it should be avoided to update all nodes contained in a bin when the bin is inserted to a block. Instead, the assignments from node to bin and from bin to block should be saved separately and all nodes should get updated in a single pass at the end, which enables inserting a bin into a block in constant time. By using appropriate data structures, common bin packing algorithms such as first fit decreasing (FFD) can be implemented in a way that the required time is dominated by the sorting of the elements [27]. Then, the first level requires $\mathcal{O}(|V| \log(|V|))$ and the second level only $\mathcal{O}(k \log(k))$ computational time. Because $k \leq |V|$ for the hypergraph partitioning problem, the resulting running time is in $\mathcal{O}(|V| \log(|V|))$.

As already mentioned (see Section 5.1), the bin packing partitioner is by itself no good solution for the problem of finding a deeply balanced bisection. The reason is that the deep balance is only guaranteed when no refinement is applied afterwards, as this could destroy the deep balance. But unfortunately, the algorithm is completely unaware of the hyperedges and thus unaware of the cut of the bisection. This means that the cut is poor in most cases and can not even be improved by a refinement. Nonetheless, the two-level approach is an important concept for ensuring deep balance also for the prepacking approach (see Sections 5.2 and 5.4) and the bin packing initial partitioner is a useful addition for a portfolio of initial partitioning algorithms as used by most state-of-the-art partitioners [38, 28].

6. Experimental Results

In this section, we evaluate configuration possibilities and the performance of the final configuration of our algorithm. First, we make a quantitative comparison of the resulting imbalance parameter for multiple possible definitions of the balance constraint. Then, we analyze different approaches for determining a prepacking threshold and the performance of different bin packing algorithms. To provide some insights into the effectiveness of our balancing strategy, we evaluate the number of restarted bisections. Our final configuration is compared with other state-of-the-art partitioners.

6.1. Setup and Methodology

Our algorithm is implemented in the direct k-way mode of the hypergraph partitioning framework *KaHyPar*. We prefer the direct k-way mode (KaHyPar-K) over the recursive bisection mode (KaHyPar-R) for two reasons: First, in direct k-way mode our algorithms can be integrated as a part of the initial partitioning phase. Because of this, they are applied to a hypergraph that is already coarsened, but still large enough for efficient bin packing (compare Section 3.1.2), resulting in a very low running time overhead. Second, KaHyPar-K produced better results than KaHyPar-R in multiple experiments and currently seems the best choice in terms of result quality [1, 37]. We refer to our implementation which uses a **w**orst **f**it bin packing algorithm as KaHyPar-WF.

Instances. For the evaluation of the performance of our algorithm we use a benchmark set consisting of 50 instances from different application areas. All hypergraphs have varying vertex weights but unit net weights. For VLSI design, we use instances from the ISPD98 VLSI Circuit Benchmark Suite (ISPD98) [2], where the weight of a hypernode corresponds to the area required by the module. Then, we use sparse matrix instances from the SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection) [14] which are interpreted as hypergraphs using the row-net model [10]. Each column corresponds to a vertex and each row to a net which connects all vertices where the row has a non-zero entry in the respective column. The weight of a vertex is the number of non-zero entries in the respective column. This represents accurately the computational weight of a task in parallel sparse-matrix vector multiplication [10]. Note that in this model, the weight of a vertex is equal to its degree. Specifically, we use a set of instances belonging to the **S**tanford **N**etwork **A**nalysis **P**latform (SNAP) [32] and a set of highly **as**ymmetric **m**atrices (ASM) [13].

Additionally, we include 5 instances with an artificially generated vertex weight distribution. The purpose of this is to provide some instances where it is hard to satisfy the balance constraint, in order to demonstrate that such instances are problematic for current state-of-the-art partitioners. These instances are constructed in a way that makes satisfying the balance constraint hard for a certain k: To provide a net structure, we use one of the aforementioned instances as a baseline. The hyperedges are kept, but we assign adjusted vertex weights. The vertices are separated by random selection into two groups: The first group consists of the majority of vertices and the second group is of fixed size chosen randomly from $[\frac{6}{5}k, 2k]$. We assign random low weights to the first group. To choose the weights for the second group, let w denote the total weight of the vertices in the first group. We choose a minimal weight a for the second group randomly from $[\frac{20}{k}w, \frac{60}{k}w]$. Then, for every vertex in the second group we generate a weight from an integer unitary distribution on $[a, 2a]$. We used $k = 128$ for all instances included in the benchmark set. In the following, we refer to instances that are not artificially generated as real-world instances.

For our evaluation, we give each algorithm a time limit of two hours and use an imbalance parameter of $\varepsilon = 0.03$ where not stated otherwise. We use a reduced benchmark set which consists of 26 instances to test different configurations of our algorithm. These instances are chosen to be evenly representative for the different sources of the instances. The set contains all artificially generated instances and asymmetric matrices and a subset of the ISPD98 and SNAP instances. We always use $k \in \{2, 32, 128\}$ on the reduced benchmark set and $k \in \{2, 4, 8, 16, 32, 64, 128\}$ on the full benchmark set.

System. The experiments are performed on a single core of a machine consisting of two Intel Xeon Gold 6230 processors. Each processor is clocked at 2.10 GHz, has 20 cores and 27.5 MB L3-Cache. The machine runs Red Hat Enterprise Linux (RHEL) 7.7 and has 96 GB main memory. Our implementation is written in C++ and compiled using g++-9.2 with flags `-O3 -mtune=native` and `-march=native`.

Performance Profiles. For a visual comparison of the result quality of different partitioners, we use plots based on performance profiles [15]. We compare the performance for the $(\lambda - 1)$-metric. We always use 10 seeds and report the partition with the best quality. More precisely, for each instance and parameters k and ε, we calculate the minimum of the objective function for all seeds for which the resulting partition is balanced. If the average imbalance over the seeds exceeds the imbalance parameter, the instance is considered imbalanced (even if there is a seed with a balanced partition). The plot is constructed by showing a curve for each included algorithm in a specific color corresponding to the algorithm. The x-axis shows the quality τ relative to the quality of the best partition found by all algorithms. The y-axis shows the fraction of instances of the algorithm where the quality is within a factor of τ relative to the best partition. For example, if an algorithm has the value 0.8 on the y-axis for $\tau = 1.5$, then 80% of the partitions computed by this algorithm have a quality of at least 1.5 times the quality of the best solution found. For $\tau = 1$ the y-value indicates the fraction of instances where the algorithm computed the best solution. Note that the x-axis is separated into three areas with different scale: The first area ranges from 1 to 1.1, indicating instances which are very close (within 10%) to the quality of the best result. In the second area, instances with medium quality, i.e. at most 2 times the value of the objective function for the best result, are displayed. The third area uses a logarithmic scale for the remaining instances, converging to infinity. The fraction of imbalanced results is displayed at the right side where the x-axis is marked with a ✗-tick. Remaining instances are either timeouts or the partitioner was unable produce a valid partition for other reasons. An algorithm is considered to outperform another algorithm in terms of result quality if its corresponding curve is above that of the other algorithm.

Competitors. We compare our configuration KaHyPar-WF to the latest version of KaHyPar and to the state-of-the-art hypergraph partitioners hMetis [28, 29], PaToH [11] and the sparse matrix partitioner Mondriaan [39]. These competitors are chosen because they provide a good solution quality [38, 37]. We use both the direct k-way (hMetis-K) and the recursive bisection (hMetis-R) variant of hMetis. For PaToH the default configuration (PaToH-D) and the quality preset (PaToH-Q) is used. As hMetis does not directly optimize the $(\lambda-1)$-metric we configure it to optimize the *sum-of-external-degrees* metric (SOED). Because it is closely related to the connectivity metric, we can calculate $(\lambda - 1)(\Pi) = \text{SOED}(\Pi) - \text{cut}(\Pi)$. The same approach is used by the authors of hMetis [29]. While the other partitioners use the same imbalance definition as KaHyPar, the definition used by hMetis-R is different [28]. For example, an imbalance value of 5 means that both blocks are allowed to weigh between $0.45 \cdot c(V)$ and

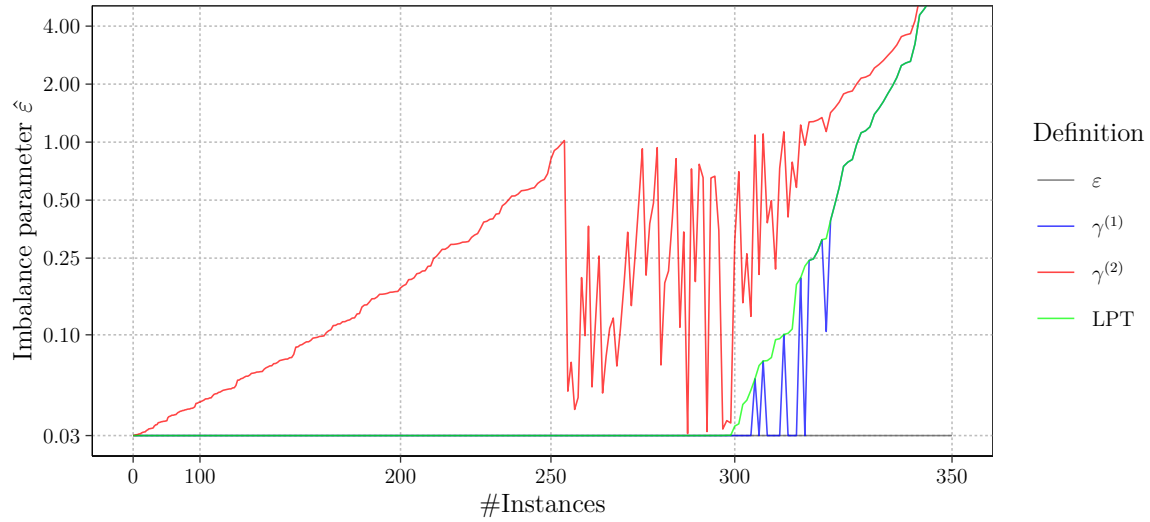

Figure 7: Comparison of different possible definitions for a generalized balance constraint, showing the resulting modified parameter $\hat{\varepsilon}$ based on $\varepsilon = 0.03$.

$0.55 \cdot c(V)$ at each bisection step. Therefore we translate the imbalance parameter ε to a modified parameter ε' such that the correct allowed block weight is matched after $\log_2(k)$ bisections:

$$\varepsilon' := 100 \cdot \left(\left((1+\varepsilon) \frac{\left\lceil \frac{c(V)}{k} \right\rceil}{c(V)} \right)^{\frac{1}{\log_2(k)}} - 0.5 \right)$$

As Mondriaan is a partitioner for matrices and not for hypergraphs, we translate our hypergraph instances to weighted matrices using the row-net model, which accurately corresponds to the connectivity metric for hypergraphs. For all compared partitioners, a modified imbalance parameter $\hat{\varepsilon}$ is used that corresponds to the generalized definition of the balance constraint as proposed in Section 4.2. The value for $\hat{\varepsilon}$ is calculated with the LPT algorithm according to Equation 4 in a preprocessing step.

6.2. Balance Constraint Definitions

In Section 4 we discussed different possibilities for a generalization of the balance constraint. To provide a better understanding of the behavior of the different possibilities, we compare them for the hypergraphs in our full benchmark set. In the following, we refer to an instance as a unique pair of a hypergraph and a parameter k. We use $k \in \{2, 4, 8, 16, 32, 64, 128\}$ for each hypergraph. Each of the possible definitions is represented by a curve in the corresponding color. LPT denotes the generalized definition calculated with the LPT algorithm. $\gamma^{(1)}$ and $\gamma^{(2)}$ refer to the alternatives defined in Equation 2 and 3. We assume an imbalance parameter of $\varepsilon = 0.03$ as a baseline. Then we calculate for each instance the modified imbalance parameter $\hat{\varepsilon}$ according to the different variants for a k-way partition of the hypergraph. The instances are sorted in increasing order of the value given by LPT. The results are given in Figure 7. The x-axis shows the number of instances, using an inverse square root scale to highlight the interesting instances, while the y-axis uses a logarithmic scale and shows the value of $\hat{\varepsilon}$ for the current instance.

As expected, the $\gamma^{(2)}$ variant has much higher values than necessary for most instances. $\gamma^{(1)}$ is more interesting: The value is nearly identical to LPT for many instances, but too small for

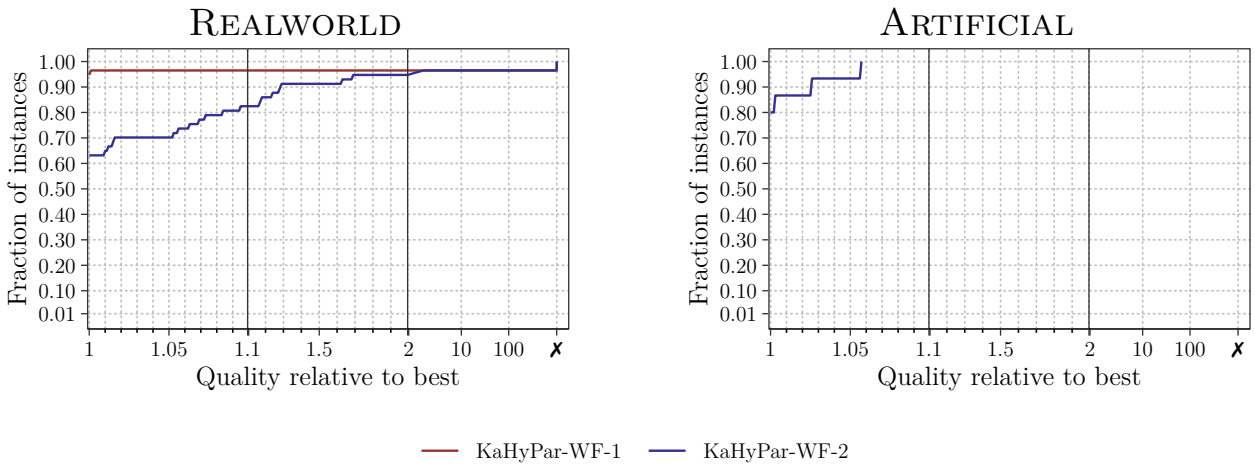

Figure 8: Connectivity performance profiles for different heuristic prepacking approaches with $\varepsilon = 0.03$ on the reduced benchmark set. A worst fit algorithm is used for the prepacking.

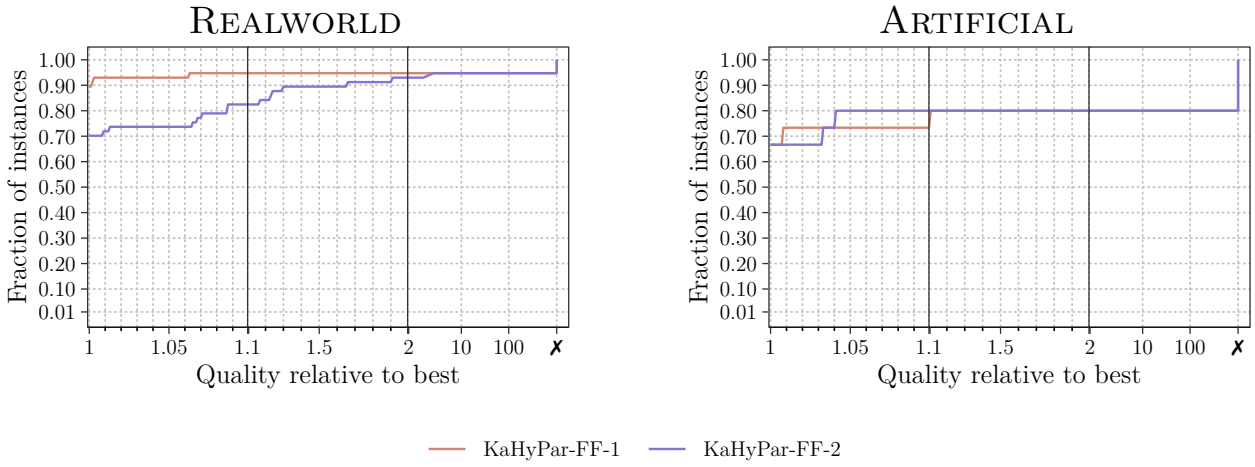

Figure 9: Connectivity performance profiles for different heuristic prepacking approaches with $\varepsilon = 0.03$ on the reduced benchmark set. A first fit algorithm is used for the prepacking.

others. The instances were the value is significantly different to LPT all have an artificially created weight distribution. This suggests that $\gamma^{(1)}$ might work well for most real-world instances. However, it is clear that no feasible solution for the artificial instances exists for the imbalance parameter given by $\gamma^{(1)}$.

6.3. Different Prepacking Approaches

In addition to our prepacking algorithm, which guarantees the deep balance of a bisection, we want to use a heuristic prepacking approach as proposed in Section 5.2. To evaluate the two heuristic approaches described in Section 5.5, we use the direct k-way mode of KaHyPar with a special configuration: Instead of using a fallback, the evaluated approach is applied before every bisection which is not at the lowest level (i.e. at least one recursive step is still applied to the resulting blocks afterwards). This demonstrates not only how good the approaches can ensure a balanced partition, but also emphasizes the effect on the objective function. We compare both heuristic approaches on the reduced benchmark set using performance profiles. From the 78

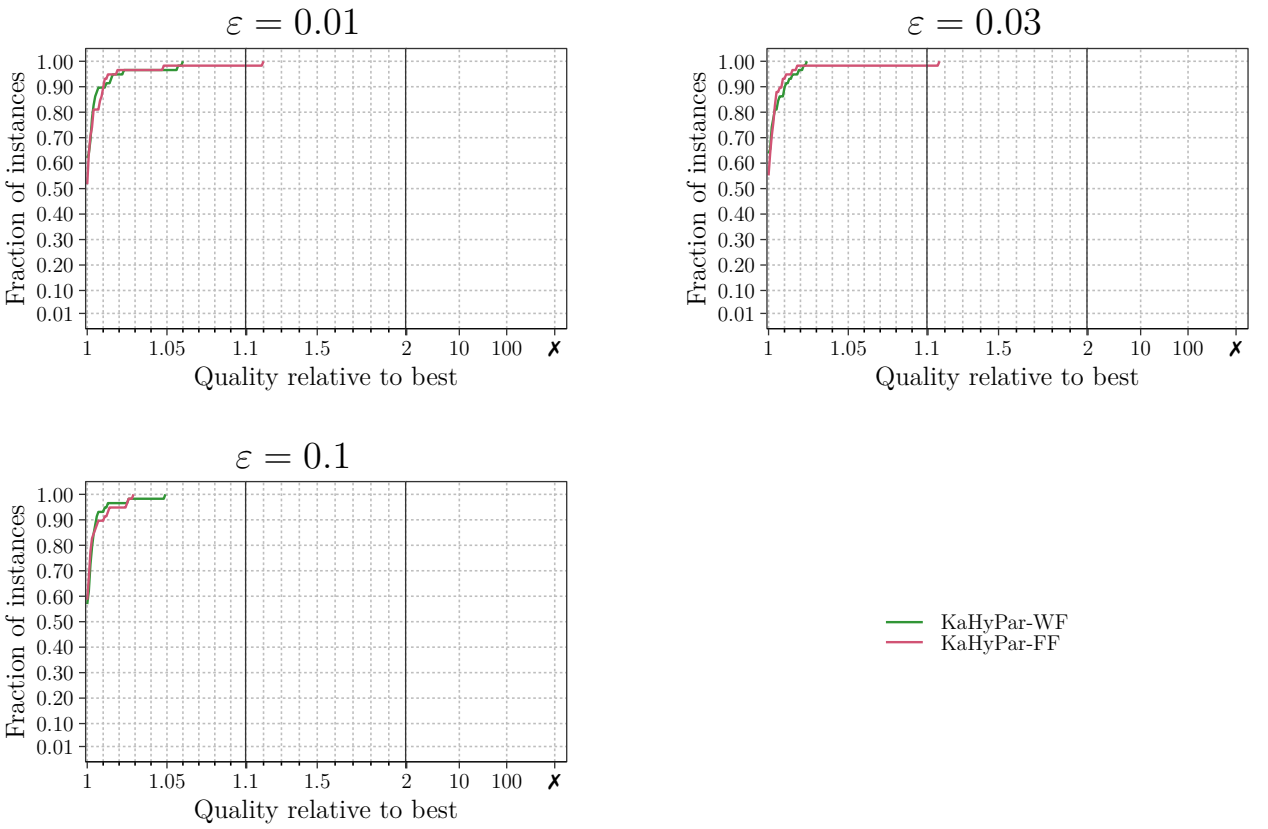

Figure 10: Connectivity performance profiles comparing different bin packing algorithms for multiple values of ε on *real-world* instances of the reduced benchmark set.

pairs consisting of an instance and value for k we excluded 6 where the time limit was exceeded.

Figure 8 shows the results using a worst fit bin packing algorithm. KaHyPar-WF-1 denotes the configuration using the first heuristic and KaHyPar-WF-2 denotes the configuration using the second heuristic as presented is Section 5.5. Figure 9 shows the results for using a first fit bin packing algorithm, the configurations are denoted in the same way as before. Interestingly, the proportion of imbalanced results is equal for both heuristic approaches. Both ensure the balance for the artificial instances if a worst fit algorithm is used, while some imbalanced partitions are produced for the real-world instances. However, the first approach is clearly superior in terms of solution quality (probably because fewer vertices are prepacked) and thus used for our final configuration. Note that the first fit algorithm does not work well for the artificial instances. Similar results for this can be observed in Section 6.4.

6.4. Bin Packing Algorithms

Our prepacking approach works with different bin packing algorithms. To evaluate the impact of the bin packing algorithm, we implemented two variants of our algorithm: A configuration that uses a **w**orst **f**it rule, denoted by KaHyPar-WF, and a configuration that uses a **f**irst **f**it rule, denoted by KaHyPar-FF. These bin packing algorithm were chosen because their behavior is contrary: Worst fit distributes the vertices with high weight equally, while first fit tends to group them together. The configurations are integrated into our balancing strategy (see Section 5.2) and evaluated on the reduced benchmark set with performance profiles. As before, we excluded 6 timeout instances. We compare both configurations for $\varepsilon \in \{0.01, 0.03, 0.1\}$ in order to assert that our final configuration works even for a tight imbalance parameter.

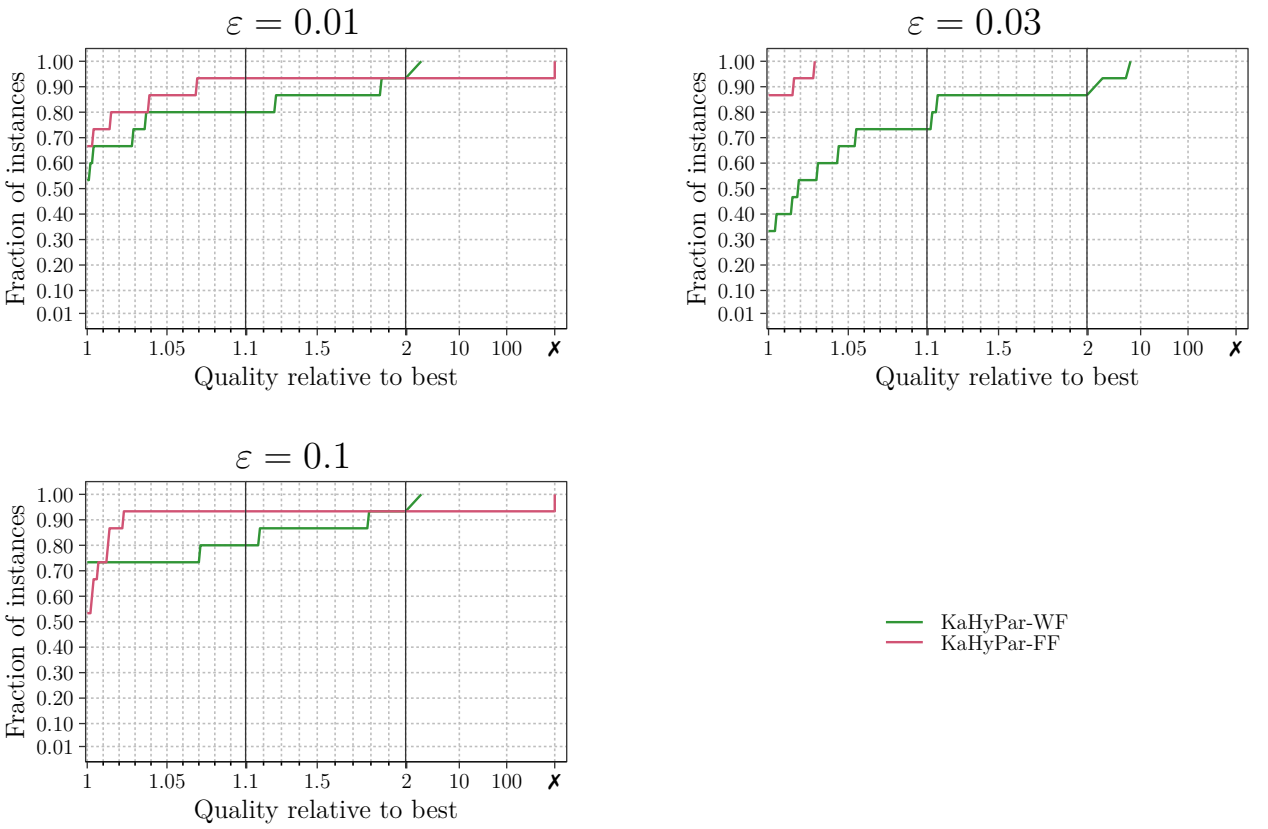

Figure 11: Connectivity performance profiles comparing different bin packing algorithms for multiple values of ε on *artificial* instances of the reduced benchmark set.

The results for the real-world instances are summarized in Figure 10. Here, the performance of both algorithms is quite equal. However, there are significant differences on the artificial instances as shown in Figure 11. While first fit produces a better connectivity for some partitions up to a factor of 2, there are also imbalanced instances. The reason for this could be that the WF algorithm is more similar to the LPT algorithm, which is used to determine the imbalance parameter and to estimate the current maximum bin (see Sections 4.2, 5.2). As the main goal of our efforts is to reliably produce balanced partitions, we choose KaHyPar-WF as our final configuration.

6.5. Restarted Bisections

The balancing strategy presented in Section 5.2 is based on restarts of the current bisection, where early restarts and late restarts are possible. To examine our assumptions and the effects

	Early/Heuristic		Early/Exact		Late/Heuristic		Late/Exact		
k	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Included[%]
8	0.80	1	0.10	1	0.00	0	0.00	0	2.3
16	0.67	1	0.27	1	0.00	0	0.00	0	7.0
32	1.73	4	0.83	3	0.00	0	0.00	0	7.5
64	2.18	14	0.84	8	0.00	0	0.00	0	12.8
128	0.64	4	0.17	2	1.70	32	0.96	26	21.6[1]

Table 1: Number of restarts for different values of k and $\varepsilon = 0.03$ on *real-world* instances.

	Early/Heuristic		Early/Exact		Late/Heuristic		Late/Exact		
k	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Included[%]
16	1.17	5	0.23	1	0.00	0	0.00	0	60.0
32	3.68	10	0.04	1	0.00	0	0.00	0	100.0
64	8.68	17	0.18	1	0.00	0	0.00	0	100.0
128	30.52	45	0.00	0	0.00	0	0.00	0	100.0

Table 2: Number of restarts for different values of k and $\varepsilon = 0.03$ on *artificial* instances.

Algorithm	All	ISPD98	ASM	SNAP social	SNAP web	SNAP other	Artificial
KaHyPar-WF	0.0	0.0	0.0	0.0	0.0	0.0	0.0
KaHyPar-K	3.2	2.3	2.3	0.0	0.0	0.4	21.7
KaHyPar-R	5.5	9.9	2.3	0.0	1.5	1.4	23.1
PaToH-Q	11.5	13.6	4.7	6.5	11.1	6.0	36.3
PaToH-D	12.0	13.4	5.6	8.2	10.4	6.4	37.1
hMetis-K	27.7	19.8	41.9	18.7	21.1	19.3	64.9
hMetis-R	10.8	12.7	4.4	4.2	5.0	5.1	46.0
Mondriaan	7.8	6.8	2.3	0.0	0.9	0.0	54.6

Table 3: Percentage of imbalanced runs for different instance sets with $\varepsilon = 0.03$.

of the strategy, we track the frequency of the different kinds of restarts for our final configuration KaHyPar-WF. Note that in addition to the distinction between early and late restarts either a heuristic prepacking or an exact prepacking is applied, resulting in four kinds of restarts. We use the full benchmark set without timeout instances (see Section 6.6). Only instances where at least one restart appeared for one of the 10 seeds are considered and the percentage of these instances is reported in addition to the frequency of restarts.

Table 1 shows the results on real-world instances. Unsurprisingly, much fewer restarts are necessary than for the artificial instances as shown in Table 2. Very few late restarts appeared for KaHyPar-WF (unlike to KaHyPar-FF), which supports the conjecture that early restarts are sufficient for most cases. On average, exact restarts are more seldom than heuristic restarts by at least a factor of two. That means in many cases a deeply balanced bisection is already found with a heuristic prepacking. Furthermore, the number of restarts on real-world instances is much smaller than the total number of bisection necessary for a k-way partition, which is $k - 1$.

6.6. Comparison with Other Hypergraph Partitioners

We compare KaHyPar-WF with the current version of KaHyPar and other state-of-the-art partitioners on the full benchmark set with $k \in \{2, 4, 8, 16, 32, 64, 128\}$. From the 350 pairs consisting of an instance and value for k we excluded 26 because at least one algorithm exceeded the time limit of two hours. A comparison containing the timeout instances is given in Appendix C. Note that there are also instances where either hMetis-K or hMetis-R could not produce a valid solution (terminating with a segmentation fault), which was nonetheless not caused by a timeout. It seems that hMetis can not handle inputs with hypernodes of much higher weight than average block weight.

[1] Our tested configuration contained a rounding error with the block weight calculation, which caused unnecessary late restarts in some cases. We guess that the correct number of late restarts is much smaller or even

Algorithm \ k	2	4	8	16	32	64	128
KaHyPar-WF	0.0	0.0	0.0	0.0	0.0	0.0	0.0
KaHyPar-K	0.0	0.0	0.2	0.8	2.2	8.4	12.6
KaHyPar-R	0.0	0.0	2.1	5.4	4.0	13.0	15.7
PaToH-Q	0.0	0.0	4.4	7.7	16.2	27.0	29.5
PaToH-D	0.0	0.0	5.2	6.7	17.1	28.0	31.7
hMetis-K	0.0	4.2	11.5	24.8	45.0	63.7	61.7
hMetis-R	0.4	0.0	2.5	11.2	15.3	26.0	27.5
Mondriaan	1.4	2.1	3.3	6.2	10.4	16.4	16.9

Table 4: Percentage of imbalanced runs for different values of k with $\varepsilon = 0.03$ on the full benchmark set.

Table 3 summarizes the proportion of imbalanced partitions for different subsets of the benchmark set and Table 4 lists the proportion of imbalanced partitions for different values of k. Here, we report the proportion over all seeds, i.e. each run is counted separately. KaHyPar-WF is able to compute a feasible solution at every run, while the other algorithms have up to 27.7% imbalanced partitions (as explained, both hMetis configurations do not produce a valid partition at all for some instances in addition to the imbalanced results). KaHyPar-K and KaHyPar-R have the next lowest imbalance values. Considering its low running time, Mondriaan also ensures balance quite well. Unsurprisingly, the imbalance is much worse for high values of k and specifically $k \in \{64, 128\}$. We want to remark that no competing algorithm is able to compute any valid solution for any artificial instance with $k = 128$.

In Figure 12 we compare the performance of the algorithms for different subsets of the benchmark set. KaHyPar-K and KaHyPar-WF outperform the other algorithms in terms of result quality, producing the best partitions on 41% of all benchmark instances. KaHyPar-WF achieves equivalent result quality to KaHyPar-K while always producing a feasible solution. Especially on the VLSI instances of the ISPD98 subset KaHyPar-WF computes the best results.

Additionally, we ran experiments with $\varepsilon \in \{0.01, 0.1\}$. The results for the full benchmark set are compared in Figure 13, more detailed plots are shown in Appendix C. As expected, the highest proportion of imbalanced results is produced for $\varepsilon = 0.01$. This demonstrates that finding a balanced partition is even harder for tighter balance constraints. In particular, PaToH-Q performs much worse for $\varepsilon = 0.01$ than for higher values of ε, producing 48.5% imbalanced partitions.

Figure 14 gives an overview over the running time of the algorithms. We calculate the *arithmetic mean* for the different seeds, but use the *geometric mean* for the average over different instances in order to give every instance a comparable influence. KaHyPar-WF has nearly equal running time to KaHyPar-K and is even faster for some instances. For both algorithms, the running time is comparable to hMetis-K and hMetis-R.

In summary, we observe that KaHyPar-K is already the best performing competitor in terms of balanced results and solution quality. Our configuration builds on top of KaHyPar-K, further improving it by computing a balanced partition for every tested instance. This is achieved without a regression in quality or runtime.

zero. Unfortunately, the computing cluster that we used for our experiments was unavailable for an unknown duration due to a security incident and thus we could not restart the experiments before the submission deadline.

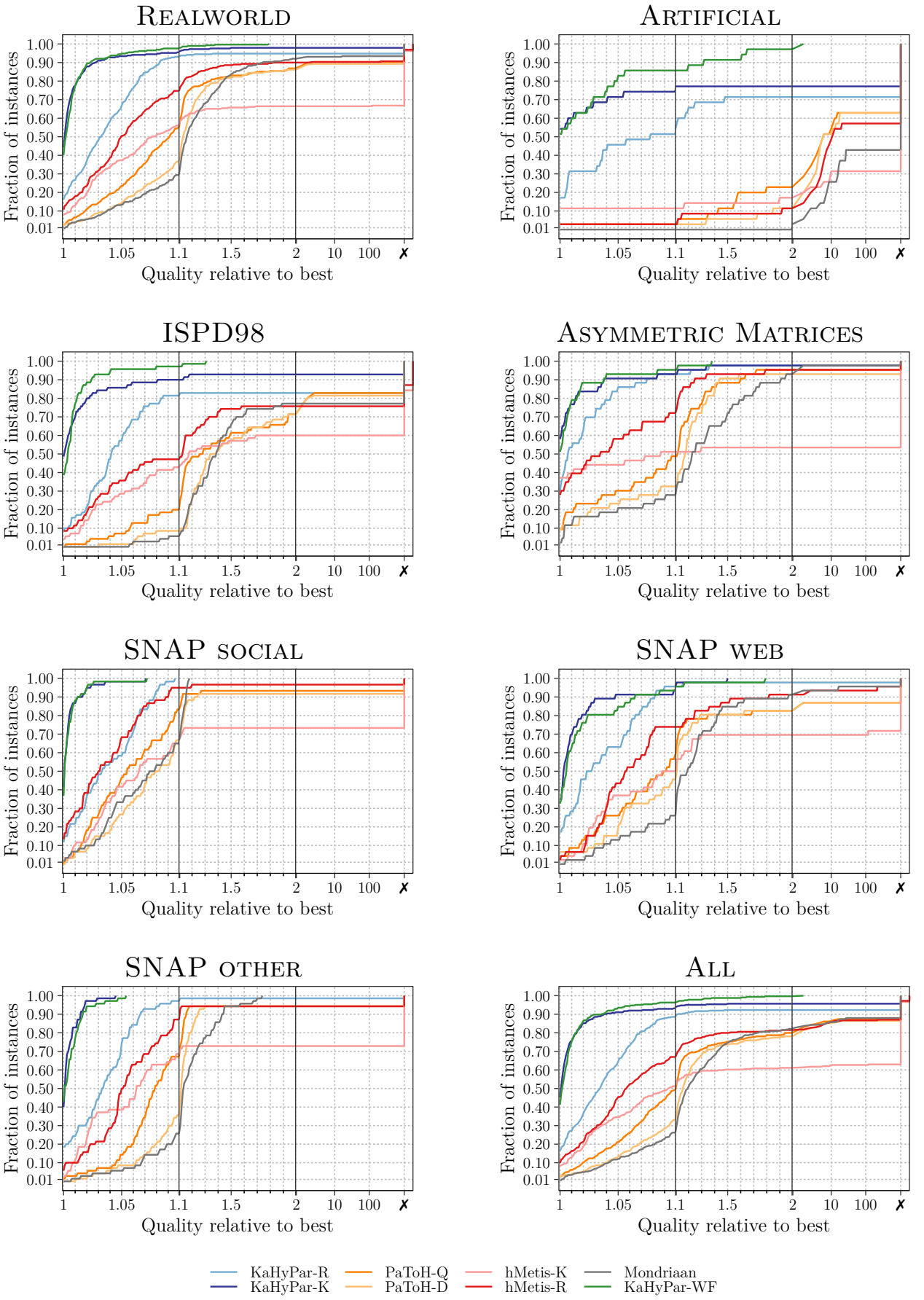

Figure 12: Connectivity performance profiles for our configuration and state-of-the-art partitioners with $\varepsilon = 0.03$ on the full benchmark set.

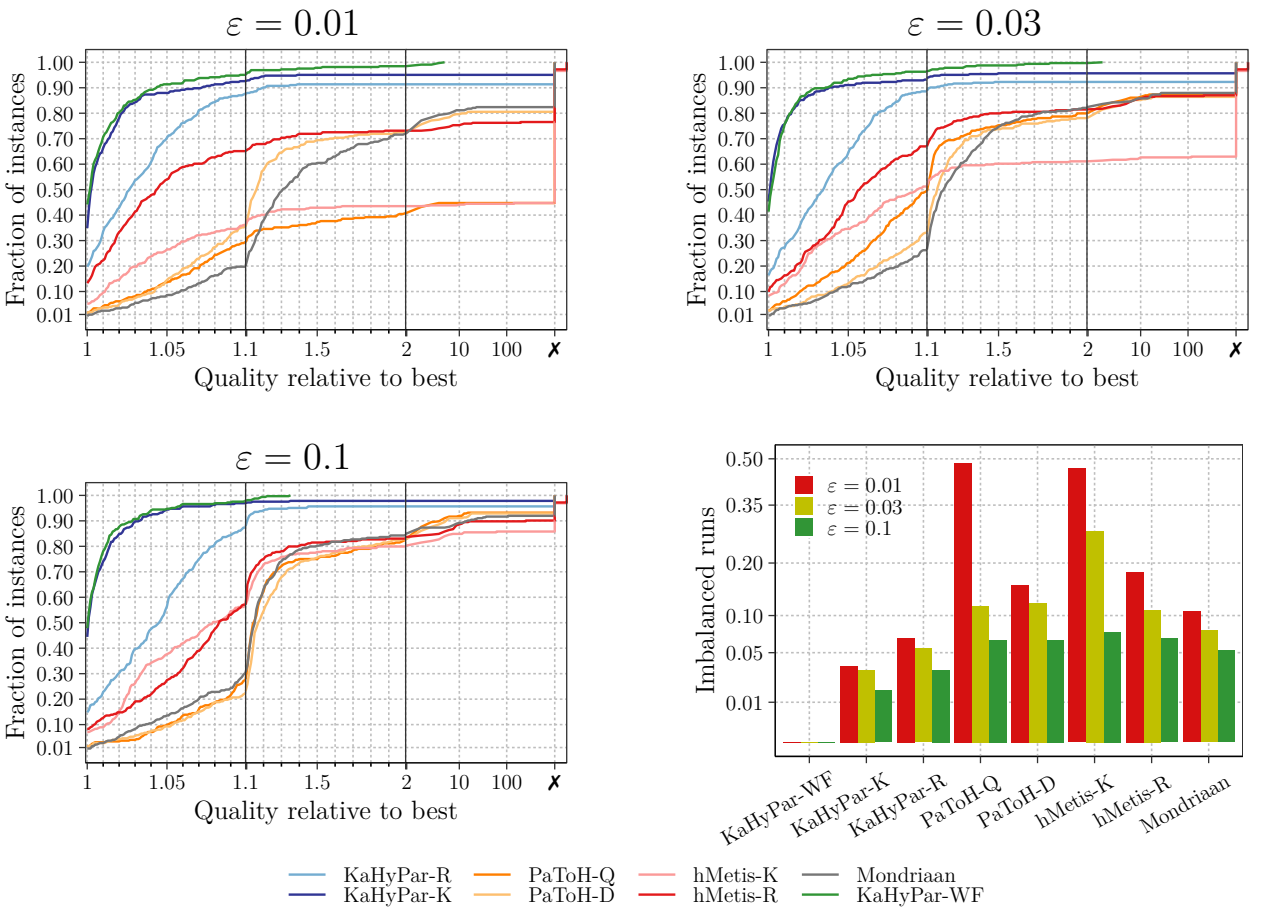

Figure 13: Connectivity performance profiles for different values of ε on the full benchmark set. Additionally, the fraction of imbalanced runs is presented with a barplot using a square root scale.

Algorithm	All	ISPD98	ASM	SNAP	Artificial
KaHyPar-WF	70.9	22.4	19.2	157.9	62.8
KaHyPar-K	73.1	25.1	20.4	155.3	67.6
KaHyPar-R	64.8	41.0	28.1	105.2	39.7
PaToH-Q	10.4	9.0	3.9	14.0	10.7
PaToH-D	1.2	0.8	0.9	1.3	1.7
hMetis-K	61.9	32.6	24.1	105.7	39.1
hMetis-R	65.6	49.3	31.3	87.0	64.3
Mondriaan	4.2	3.4	4.4	4.4	4.7

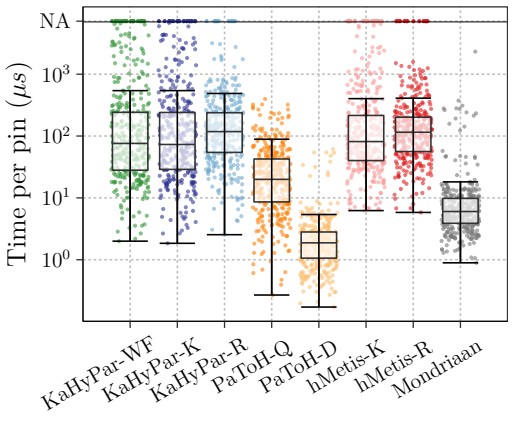

Figure 14: Comparing the running times for $\varepsilon = 0.03$ on the full benchmark set. The table lists different benchmark subsets using the *geometric mean* over the instances. We visualize the running times normalized by the number of pins, using a combination of a scatter plot (consisting of a point for each instance) and a box plot that shows the median and quartiles. The whiskers indicate the smallest/largest data point within 1.5 times the inter-quartile range to the lower/upper quartile. Instances where no valid partition is produced (e.g. timeouts) are displayed above the line labeled with "NA".

7. Conclusion

Our exploration of the issues related to finding a balanced partition for hypergraphs with vertex weights included several aspects of multilevel partitioning algorithms. In the following, we summarize the most important insights.

For weighted hypergraph, an adjusted definition for the balance of partitions is required. We investigated existing approaches for defining the upper allowed block weight that use the maximum vertex weight of the hypergraph. However, these approaches either can not guarantee a feasible solution or yield an unnecessary relaxed constraint. Instead, we developed a more suitable definition based on bin packing techniques which overcomes these problems.

In order to construct a multilevel partitioner that can guarantee a balanced partition, different considerations are necessary for the application of the multilevel paradigm: In the coarsening phase, it is both necessary and sufficient to restrict the allowed weight of vertex contractions to a reasonable threshold. This assures that the input hypergraph for the initial partitioning is not too imbalanced. If the k-way partition is calculated directly, our approach relies on the initial partitioning phase for ensuring a balanced solution. Provided a balanced initial partition, e.g. via bin packing algorithms, it is straightforward to project it to a balanced partition of the input hypergraph. This is because most refinement algorithms already ensure that the balance of the partition is preserved.

If a k-way partition is calculated via recursive bisection, additional measures are required. Balancing the blocks of a bisection is insufficient to guarantee the balance of the resulting partition. Therefore we introduce the notion of the *deep balance* of a bisection, which requires that it is possible to recursively subdivide the blocks into a balanced k-way partition. For this, it must be considered that unlike the direct k-way case a 2-way refinement might destroy the deep balance of an initial bisection. In particular, problems arise if too many heavy vertices are assigned to the same block. For this reason, our approach is based on an assignment of heavy vertices as *fixed vertices* to both blocks before the bisection is calculated, which we call a *prepacking*. Based on theoretical results, we developed an exact prepacking algorithm with near-linear running time that guarantees the deep balance of a bisection, using modified bin packing algorithms. Unfortunately, a prepacking is unaware of the objective function and thus can affect the result quality negatively by restricting the possible solution space. Therefore we additionally developed a heuristic prepacking approach that uses fewer vertices and tries to improve the likelihood that the bisection is deeply balanced. To integrate all this into a multilevel partitioner, we use a strategy based on a fallback concept: A prepacking is applied only if the plain bisection algorithm fails to produce a deeply balanced bisection. Furthermore, we use the heuristic prepacking for the first fallback and apply the exact prepacking only if this is not sufficient.

In our experiments, no state-of-the-art partitioner can reliably produce a balanced solution on all instances of our benchmark set. The proportion of imbalanced results ranges from 1.7% (KaHyPar-K) to 7.5% (hMetis-K) for $\varepsilon = 0.1$, 3.2% (KaHyPar-K) to 27.7% (hMetis-K) for $\varepsilon = 0.03$ and 3.6% (KaHyPar-K) to 48.5% (PaToH-Q) for $\varepsilon = 0.01$. However, our new configuration KaHyPar-WF always produces a balanced partition without noticeable running time overhead (it is slightly faster) compared to KaHyPar-K. The solution quality is at least equivalent to KaHyPar-K, indicating the overall effectiveness of our balancing strategy.

7.1. Future Work

As this is to our knowledge the first work concerning vertex weights in context of the multilevel paradigm, it is not surprising that many paths are left open for future research. One approach

that we did not examine further is *rebalancing*: Integrating techniques into the refinement phase to actively restore the balance of a partition. For our framework it is not necessary to use such approaches as we were successful in ensuring a balanced initial partition. However, it might still be interesting to explore possibilities for rebalancing. First, allowing some imbalance in the initial partition could open more possibilities for optimizing the objective function. Second, commonly used refinement techniques are incapable of moving heavy vertices even if the initial partition is balanced. The reason fo this is that local search algorithms are usually based on single vertex moves and only allow moves that preserve the balance constraint [16]. Therefore, new methods that allow for moves of heavy vertices could be useful to improve the refinement even for partitions that are already balanced.

Concerning the problem formulation itself, there is still a problem with our generalized definition presented in Section 4: If there are vertices with higher weight than average block weight our definition ensures the existence of a feasible solution, but for most blocks (except those containing the vertices with high weight) the upper allowed block weight is much higher than necessary. This issue could be solved with variable block weights, i.e. by assigning different upper allowed weights to each block. One possible approach is to assign all vertices with higher weight than average block weight to a separate block and determine the upper allowed weight for the remaining blocks independently of those vertices.

In Section 5.4, we introduced the prepacking problem and a prepacking algorithm for solving it. Although it can be considered as a separate algorithm independent to partitioning, we did not perform an experimental evaluation. One reason is that we do not know of a similar algorithm that could be used for comparison. Additionally, it is unclear how an experimental framework could look like. What are suitable test instances and what is the best way to measure the result quality? Another question related to the prepacking algorithm is how to optimally choose the allowed block weight u when calling it.

There are also alternatives to the prepacking approach for ensuring a deeply balanced bisection. For example, we did not make a closer examination of the possibilities for integrating the notion of deep balance into currently used algorithms for initial partitioning or refinement.

Another related idea that might be worth exploring is whether it is possible to construct a bin packing algorithm that is aware of the objective function. Maybe a kind of *any fit* algorithm which tries to choose a bin that is optimal according to the objective function for the elements that are already packed could be considered. Indeed, we currently use rather primitive bin packing algorithms (namely the *worst fit* and the *first fit* rules) and the use of more refined methods could also be helpful for minimizing the imbalance.

Furthermore, there are possible improvements to our strategy for integrating the balancing methods into the multilevel framework. It might be useful to test other heuristic approaches for ensuring the deep balance than a heuristic prepacking, e.g. a bisection without prepacking but with reduced imbalance parameter. Additionally, we explain a few alternative ideas that were not included in the final framework (compare Section 5.2) and our reasoning for the decision:

(i) A simpler alternative based on the fallback concept could use a bin packing initial partitioner instead of a prepacking (see Section 5.6). Either by replacing the third level or using only two levels, where the bin packing initial partitioner is the second level. The partitioner should be applied directly to the input hypergraph, i.e. without coarsening and refinement, as the refinement is not aware of the deep balance. This is simpler to implement and has a lower runtime overhead than the prepacking approach. However, a bin packing initial partitioner is inferior concerning the objective function, specifically when no refinement is applied afterwards.

(ii) For instances with hypernodes with high weight, it can be necessary to allow a rather high imbalance for the blocks (compare Equation 4). It seems beneficial to use this to allow for a higher imbalance of the single bisections, increasing possibilities for finding a better cut. Therefore we developed an alternative to the adaptive imbalance parameter ε' (Equation 1) which is based on the adaptive bin imbalance introduced in Definition 5.3: We define $\varepsilon'' := (1 + \varepsilon'_B)\frac{k}{c(V')} \max_B(H', k') - 1$, where H' is the current subhypergraph. ε'' allows for a higher imbalance as $\varepsilon'' \geq \varepsilon'$. However, in our tests the use of ε'' had worse results on average. We guess that the reason for this is the following: A higher allowed imbalance leads more frequently to bisections with high imbalance. This restricts the possibilities for bisections that are applied recursively to the block with high weight and, more importantly, it restricts the possibilities for the refinement as the refinement only considers moves within the allowed block weight, therefore resulting in a worse objective function.

(iii) We could allow the prepacking to use the complete allowed block weight $(1 + \varepsilon)\left\lceil \frac{c(V)}{k} \right\rceil$ instead of $(1 + \varepsilon'_B) \max_B(H, k)$ as bin capacity. An advantage of this is that then fewer vertices must be prepacked for the current bisection. On the other hand, it will be much more likely that prepackings are required for the bisections at lower levels, too.

(iv) Reversely, we could use $(1 + \varepsilon'_B) \max_B(H, k)$ as a more restrictive bound for the current maximum bin when testing the deep balance of a bisection, ensuring a strict balance at every level. But we think it is better to avoid the use of a prepacking as long as possible.

(v) KaHyPar uses a pool of different algorithms for initial partitioning which are run multiple times each, choosing the balanced result with the best objective function [23]. An interesting idea is to test the deep balance of the resulting bisection for each of those algorithms separately, restarting the algorithm with a prepacking if necessary. Then, solutions that are already deeply balanced without the use of a prepacking are not discarded only because a solution with a better cut was chosen which was not deeply balanced. Unfortunately, this idea can not be integrated efficiently into the current partitioning process: The initial partitioning algorithms operate on a coarsened hypergraph of constant size which is too small for efficient bin packing. Additionally, the refinement can affect the deep balance. Therefore it would be necessary to uncoarsen the hypergraph before the test. Doing this for every bisection calculated by the pool would increase the required running time significantly.

Finally, this thesis focuses on the task of ensuring a balanced partition. But independent to the balance it is also an interesting question how vertices with high weight influence the optimization possibilities for the objective function. Possibly there are optimization techniques which can take advantage of the weight distribution of a hypergraph that are not applicable to unweighted hypergraphs. If so, this could be an exciting area to explore.

References

[1] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. Engineering a Direct k-Way Hypergraph Partitioning Algorithm. In *Proceedings of 19th Workshop on Algorithm Engineering and Experiments*, pages 28–42. SIAM, 2017.

[2] C. J. Alpert. The ISPD98 Circuit Benchmark Suite. In *Proceedings of International Symposium on Physical Design*, pages 80–85, 1998.

[3] B. F. Auer and R. H. Bisseling. Efficient Matching for Column Intersection Graphs. *Journal of Experimental Algorithmics*, 19:1–1, 2015.

[4] J. Balogh, J. Békési, G. Dósa, L. Epstein, and A. Levin. A New Lower Bound for Classic Online Bin Packing. In *International Workshop on Approximation and Online Algorithms*, pages 18–28. Springer, 2019.

[5] J. Békési, G. Galambos, and H. Kellerer. A 5/4 Linear Time Bin Packing Algorithm. *Journal of Computer and System Sciences*, 60(1):145–160, 2000.

[6] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[7] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992.

[8] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Hypergraph Partitioning With Fixed Vertices. In *Proceedings of 36th annual ACM/IEEE Design Automation Conference*, pages 355–359, 1999.

[9] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Iterative Partitioning with Varying Node Weights. *VLSI Design*, 11(3):249–258, 2000.

[10] U. V. Catalyurek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[11] Ü. V. Çatalyürek and C. Aykanat. Patoh (Partitioning Tool for Hypergraphs). In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011.

[12] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[13] T. Davis, I. S. Duff, and S. Nakov. Design and Implementation of a Parallel Markowitz Threshold Algorithm. Technical report, RAL-TR-2019-003, Rutherford Appleton Laboratory, 2019.

[14] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011.

[15] E. D. Dolan and J. J. Moré. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[16] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*, pages 175–181. IEEE, 1982.

[17] K. Fleszar and C. Charalambous. Average-Weight-Controlled Bin-Oriented Heuristics for the One-Dimensional Bin-Packing Problem. *European Journal of Operational Research*, 210(2):176–184, 2011.

[18] K. Fleszar and K. S. Hindi. New Heuristics for One-Dimensional Bin-Packing. *Computers & Operations Research*, 29(7):821–839, 2002.

[19] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[20] J. N. Gupta and J. C. Ho. A New Heuristic Algorithm for the One-Dimensional Bin-Packing Problem. *Production Planning & Control*, 10(6):598–603, 1999.

[21] J. N. Gupta and A. J. Ruiz-Torres. A LISTFIT Heuristic for Minimizing Makespan on Identical Parallel Machines. *Production Planning & Control*, 12(1):28–36, 2001.

[22] J. Hartmanis. Computers and Intractability: A Guide to the Theory of NP-Completeness (Michael R. Garey and David S. Johnson). *SIAM Review*, 24(1):90–91, 1982.

[23] T. Heuer. Engineering Initial Partitioning Algorithms for Direct k-Way Hypergraph Partitioning. Master's thesis, Karlsruher Institut für Technologie (KIT), 2015.

[24] T. Heuer. High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations. Master's thesis, Karlsruher Institut für Technologie (KIT), 2018.

[25] T. Heuer and S. Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *Proceedings of 16th International Symposium on Experimental Algorithms*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[26] D. S. Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[27] D. S. Johnson. Fast Algorithms for Bin Packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[28] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration Systems*, 7(1):69–79, 1999.

[29] G. Karypis and V. Kumar. Multilevel k-Way Hypergraph Partitioning. *VLSI Design*, 11(3):285–300, 2000.

[30] A. H. Kashan and B. Karimi. A Discrete Particle Swarm Optimization Algorithm for Scheduling Parallel Machines. *Computers & Industrial Engineering*, 56(1):216–223, 2009.

[31] W.-C. Lee, C.-C. Wu, and P. Chen. A Simulated Annealing Approach to Makespan Minimization on Identical Parallel Machines. *The International Journal of Advanced Manufacturing Technology*, 31(3-4):328–334, 2006.

[32] J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, 2014.

[33] E. C. man Jr, M. Garey, and D. Johnson. Approximation Algorithms for Bin Packing: A Survey. *Approximation Algorithms for NP-Hard Problems*, pages 46–93, 1996.

[34] Z. Á. Mann and P. A. Papp. Formula Partitioning Revisited. In *5th Pragmatics of SAT Workshop*, volume 27 of *EPiC Series in Computing*, pages 41–56. EasyChair, 2014.

[35] E. Pacini, C. Mateos, and C. G. Garino. Distributed Job Scheduling based on Swarm Intelligence: A Survey. *Computers & Electrical Engineering*, 40(1):252–269, 2014.

[36] D. A. Papa and I. L. Markov. Hypergraph Partitioning and Clustering. In *Handbook of Approximation Algorithms and Metaheuristics*, pages 61–1–61–19. CRC Press, 2007.

[37] S. Schlag. High-Quality Hypergraph Partitioning. 2020.

[38] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k-Way Hypergraph Partitioning via n-Level Recursive Bisection. In *Proceedings of 18th Workshop on Algorithm Engineering and Experiments*, pages 53–67. SIAM, 2016.

[39] B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005.

[40] M. Yue. On the Exact Upper Bound for the Multifit Processor Scheduling Algorithm. *Annals of Operations Research*, 24(1):233–259, 1990.

A. Running Time Overhead of the Balancing Strategy

The strategy for computing deeply balanced bisections presented in Section 5.2 builds on a bisection algorithm. However, the techniques that ensure the deep balance add additional overhead and can even cause the restart of the computation of a bisection. Therefore we provide an analysis of the impact the strategy has on the running time of a recursive bisection algorithm.

Assume a k-way partition of a hypergraph $H = (V, E, c, \omega)$ is calculated via recursive bisection. In addition to the bisection algorithm, we use the LPT algorithm which requires $\mathcal{O}(|V| \log(|V|))$ time to test whether the calculated bisection is deeply balanced. If the calculation is restarted, we apply a heuristic prepacking that is also in $\mathcal{O}(|V| \log(|V|))$ and possibly our prepacking algorithm, which has a running time in $\mathcal{O}(|V|k \log(k) + |V| \log(|V|))$ (see Sections 5.4 and 5.5). Additionally the bisection algorithm itself is repeated, adding a constant factor of at most 3. Thus if only *early restarts* appear, i.e. the calculation is restarted before the algorithm is applied recursively to the blocks, the overhead is rather small.

For a practical implementation it should be observed that the LPT algorithm and both prepackings require a sorting of the hypernodes. As all operations are applied to H or a subhypergraph of H, it is sufficient to sort the hypernodes only once and cache the resulting order. The sorting could even be reused in recursive applications of the algorithm. If the hypernodes are already sorted, the LPT algorithm and the heuristic prepacking only require $\mathcal{O}(|V| \log(k))$ running time if appropriate data structures are used. The exact prepacking algorithm is more expensive, but it should be noted that in most applications k is much smaller than n. In most cases its running time is much better than the worst case anyway, sometimes even linear in $|V|$ (compare Appendix B).

The situation is more complicated if *late restarts* appear. In theory, the required running time could increase significantly. The reason is that not only the current bisection, but also all recursive bisections are recalculated. If we assume that two late restarts happen for every bisection, many more bisections are computed: The depth of the bisection tree built by recursive bisection for a k-way partition is $\lceil \log_2(k) \rceil$. Without restarts, the number of bisections doubles at every level and thus the total number of computed bisections is in $\mathcal{O}(2^{\log_2(k)}) = \mathcal{O}(k)$. With two restarts the number of bisections increases by a factor of 6 and the total number is in $\mathcal{O}(6^{\log_2(k)}) = \mathcal{O}(k^{\log_2(6)}) \approx \mathcal{O}(k^{2.585})$. Fortunately, the early restarts are sufficient in most cases which makes the worst case extremely unlikely to happen and it will probably never appear in practice. But it is useful to introduce an additional rule for avoiding useless restarts: If $\max_B(H, k) > (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$ it is very unlikely to find a balanced partition and we do not restart the procedure.

B. Detailed Running Time Analysis of the Prepacking Algorithm

The prepacking algorithm presented in Section 5.4 has a worst case running time of $\mathcal{O}(nk\log(k)+n\log(n))$, where n is the number of elements and k the number of bins per block. A noteworthy detail is that the running time of a single execution heavily depends on the actual number of iterations of the main loop. Furthermore, the number of iterations corresponds to the number m of prepacked vertices and thus to the result quality, i.e. a small number of iterations is desirable in terms of both result quality and runtime. Therefore, it might be useful to analyze what value of m can be expected for most instances. But a formal analysis is unfortunately not easily possible. m depends primarily on the weight distribution of the elements and it is unclear what an expected distribution could be (we could indeed consider different probability distributions, but this is not in the scope of this thesis and probably not very useful). However, we can observe that m also depends on the input parameters and tends to be smaller if the bins have a lot of free space, i.e. $\frac{k\cdot b}{2}-u$ is relatively large. We denote this idea of free space by

$$\delta := \frac{k\cdot b}{2u} - 1$$

Then it is possible to give a simple bound for m in these terms:

Lemma B.1 (Bound for m)**.** *Let $L=\langle a_1,\dots,a_n\rangle$ be a sequence in non-increasing order, k the number of bins, u the upper allow block weight and b the bin capacity. Let m be minimal such that $\frac{1}{k/2}u + h_{2/k}(L_t) \le b$ holds for any subsequence $L_t=\langle a_{m+1},\dots,a_t\rangle$ with $m<t\le n$. If $w(L)\le 2u\le k\cdot b$ then $m\le \frac{1}{\delta}k$.*

Proof. W.l.o.g. let $m>0$. We claim $a_m > b-\frac{1}{k/2}u$. Assume not, let $m'=m-1$ and L'_t any subsequence of the form $L'_t=\langle a_m,\dots,a_t\rangle$. As L is non-increasing the largest element in L'_t is a_m. Thus we have the bound $h_{k/2}(L'_t)\le(1-\frac{1}{k/2})a_m \le b-\frac{1}{k/2}u$ and therefore $\frac{1}{k/2}u+h_{k/2}(L'_t)\le b$, i.e. m' is a possible smaller choice for m, contradiction.

Considering the first m elements, we have

$$m\cdot a_m \le \sum_{i=1}^{m} a_i \le w(L) \le 2u$$

and conclude

$$m \le \frac{2u}{a_m} \le \frac{2u}{b-\frac{1}{k/2}u} = \frac{k}{\frac{k\cdot b}{2u}-1} = \frac{1}{\delta}k$$

□

The definition for m used in the lemma is a worst-case estimate for the actual value calculated by the algorithm. Specifically, we have $m\in\mathcal{O}(\frac{1}{\delta}k)$ for every valid input. Using this, we can state the alternative bound $\mathcal{O}(\frac{1}{\delta}k^2\log(k)+\frac{1}{\delta}k\log(n)+n)$ for the runtime of our prepacking algorithm. This bound might not be very useful, but it is certainly interesting to see how the runtime and the size of the prepacking depend on the bin capacity b and the choice of u.

When calling the algorithm, we can choose the upper allowed block weight u freely between $\frac{1}{2}w(L)$ and $\frac{k\cdot b}{2}$. As explained in Section 5.4, we use the value given by the adaptive imbalance parameter ε' for u in principle. However, the relationship between δ and m indicates that it is favorable to choose u in a way such that δ is not too small, i.e. u should be quite a

bit smaller than $\frac{k \cdot b}{2}$. Thus we included an additional check in our implementation: We set $u = \frac{1}{4}(w(L) + k \cdot b)$ if the value given by ε' is greater than $\frac{1}{2}(w(L) + \frac{9}{10}(k \cdot b - w(L)))$.
Additionally, we want to remark that there are some optimization possibilities for the prepacking algorithm which might be useful for a practical implementation. They don't affect the asymptotic runtime and are also not used by our current implementation, but we will mention them for the sake of completeness:

(i) The values for t and h calculated in the main loop could be reused as initial values for the second loop that computes u', as the index calculated by the main loop is a lower bound for the one calculated by the second loop. This could reduce the number of passes for the second loop significantly.

(ii) Instead of using a binary search, it would be possible to extend the segment tree structure to include the prefix sums and calculate t and $h_{k/2}(L_t)$ in a single pass over the tree, perhaps improving cache locality. However, while the feasibility is clear the implementation details are a bit involved.

(iii) A lazy segment tree implementation could be beneficial. I.e. an implementation without a build procedure, calculating the values of the nodes only if actually queried. As only elements smaller than the current value of t are queried and m is probably small for most instances, a significant part of the elements might never be queried and initializing the nodes for this suffix range is unnecessary.

C. Detailed Comparison with Other Partitioners

In the following, we present some more detailed experimental results for the comparison with state-of-the-art partitioning algorithms, supplementing the data given in Section 6.6. We show the proportion of imbalanced runs for $\varepsilon \in \{0.01, 0.1\}$ and multiple additional performance profiles. These compare the partitioners on the full benchmark set based on the $(\lambda - 1)$-metric. An explanation of the performance profiles is given in Section 6.1.

Algorithm	All	ISPD98	ASM	SNAP social	SNAP web	SNAP other	Artificial
KaHyPar-WF	0.0	0.0	0.0	0.0	0.0	0.0	0.0
KaHyPar-K	3.6	2.7	2.3	0.0	0.0	1.3	22.9
KaHyPar-R	6.7	11.6	2.3	0.0	2.4	3.4	26.3
PaToH-Q	48.5	37.6	56.3	47.2	44.8	46.6	72.0
PaToH-D	15.4	16.7	7.4	11.3	12.8	9.3	44.9
hMetis-K	46.6	34.6	78.4	30.2	35.2	38.9	86.6
hMetis-R	18.1	13.6	9.1	14.0	10.2	15.0	60.3
Mondriaan	10.6	10.6	2.6	0.0	3.9	0.0	68.9

Table 5: Percentage of imbalanced runs for different instance sets with $\varepsilon = 0.01$.

Algorithm	All	ISPD98	ASM	SNAP social	SNAP web	SNAP other	Artificial
KaHyPar-WF	0.0	0.0	0.0	0.0	0.0	0.0	0.0
KaHyPar-K	1.7	0.6	0.0	0.0	0.0	0.0	14.3
KaHyPar-R	3.3	5.0	1.2	0.0	0.2	1.4	15.7
PaToH-Q	6.5	8.4	2.3	3.3	3.3	2.7	25.4
PaToH-D	6.5	6.9	3.0	4.7	4.6	2.0	24.6
hMetis-K	7.5	13.0	3.5	2.4	3.9	1.9	28.0
hMetis-R	6.7	7.9	2.3	2.7	2.4	2.1	31.7
Mondriaan	5.3	3.2	0.9	0.0	2.2	0.0	38.3

Table 6: Percentage of imbalanced runs for different instance sets with $\varepsilon = 0.1$.

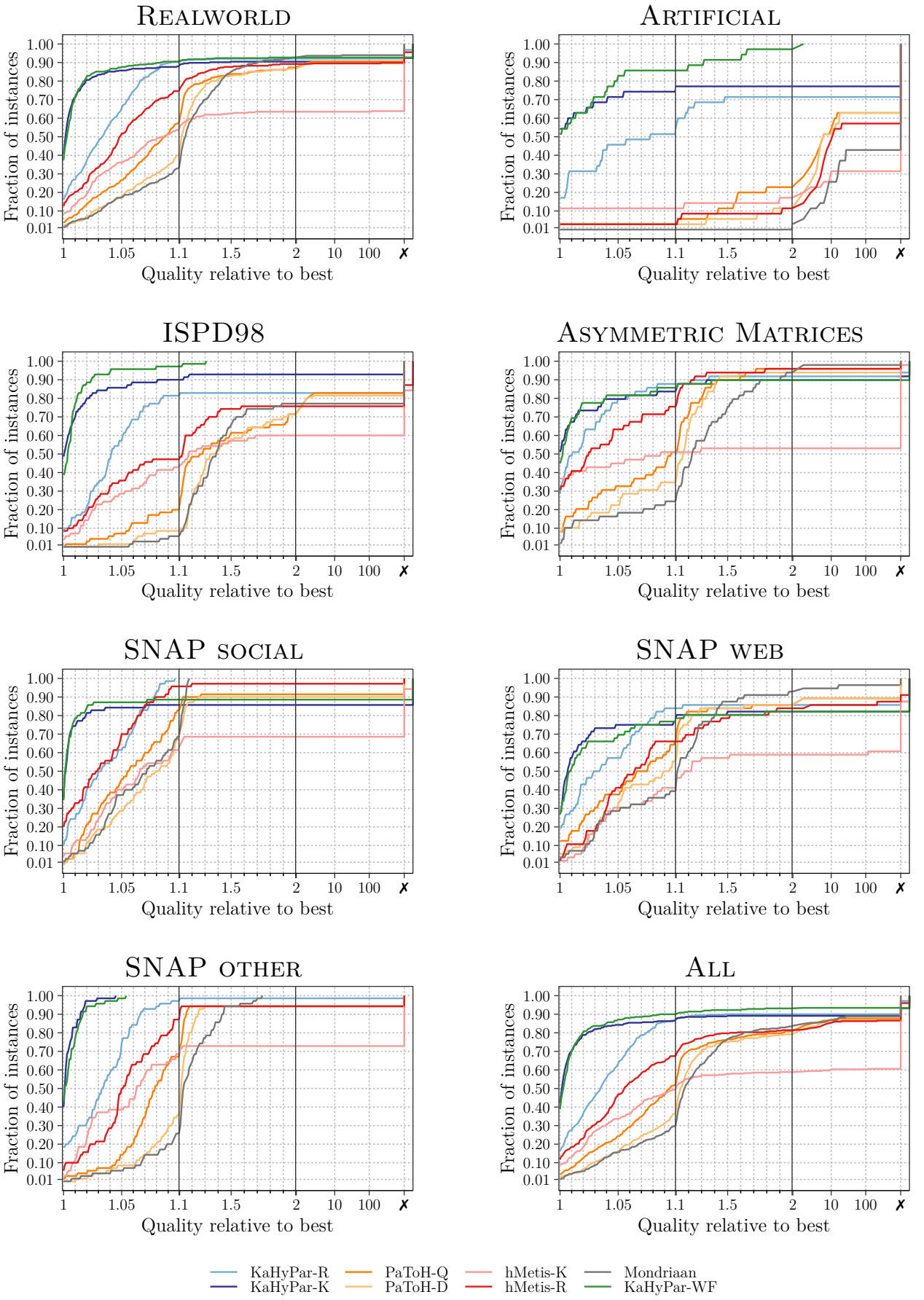

Figure 15: Connectivity performance profiles for our configuration and state-of-the-art partitioners with included timeouts and $\varepsilon = 0.03$ on the full benchmark set.

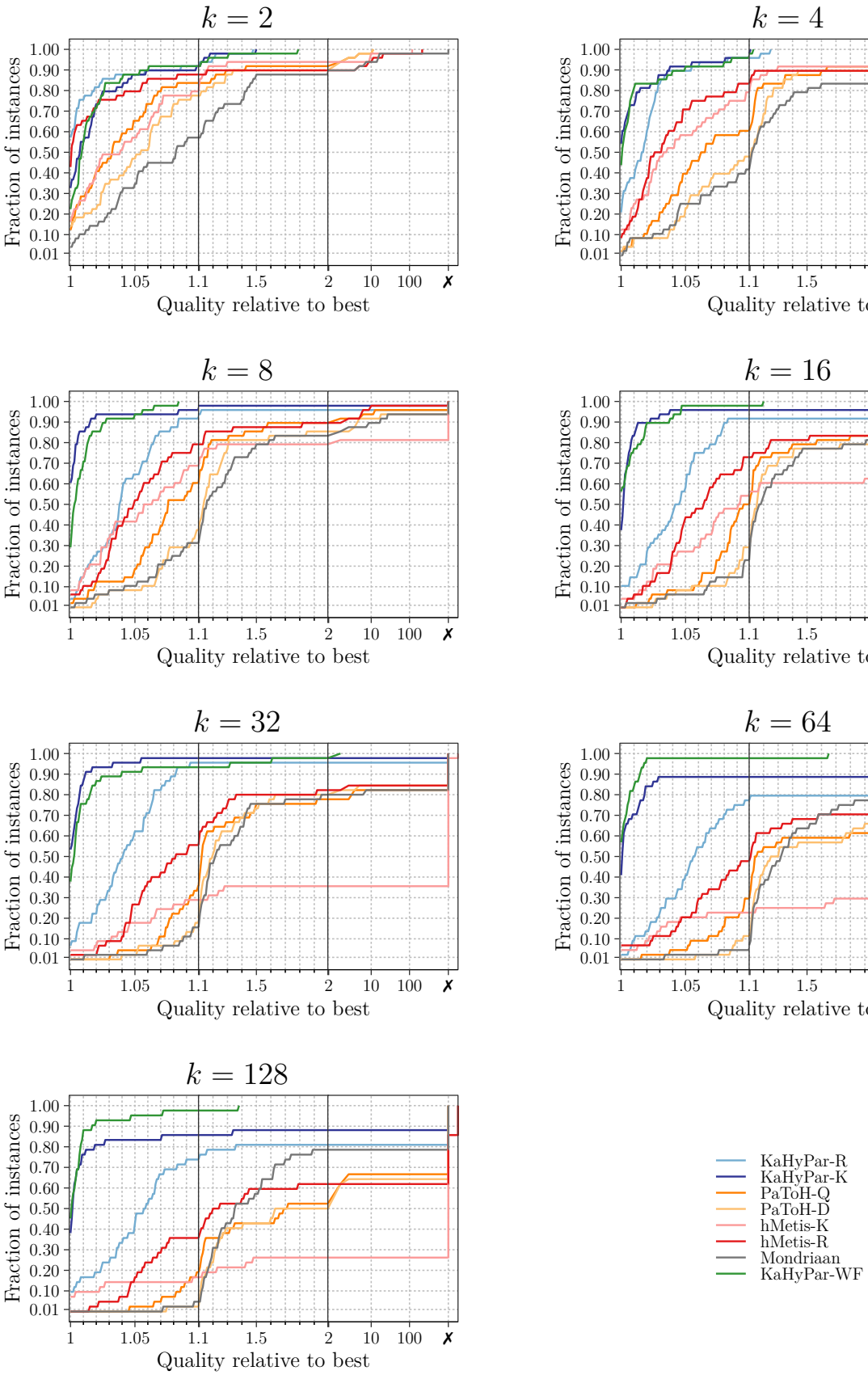

Figure 16: Connectivity performance profiles for our configuration and state-of-the-art partitioners for different values of k and $\varepsilon = 0.03$ on the full benchmark set.

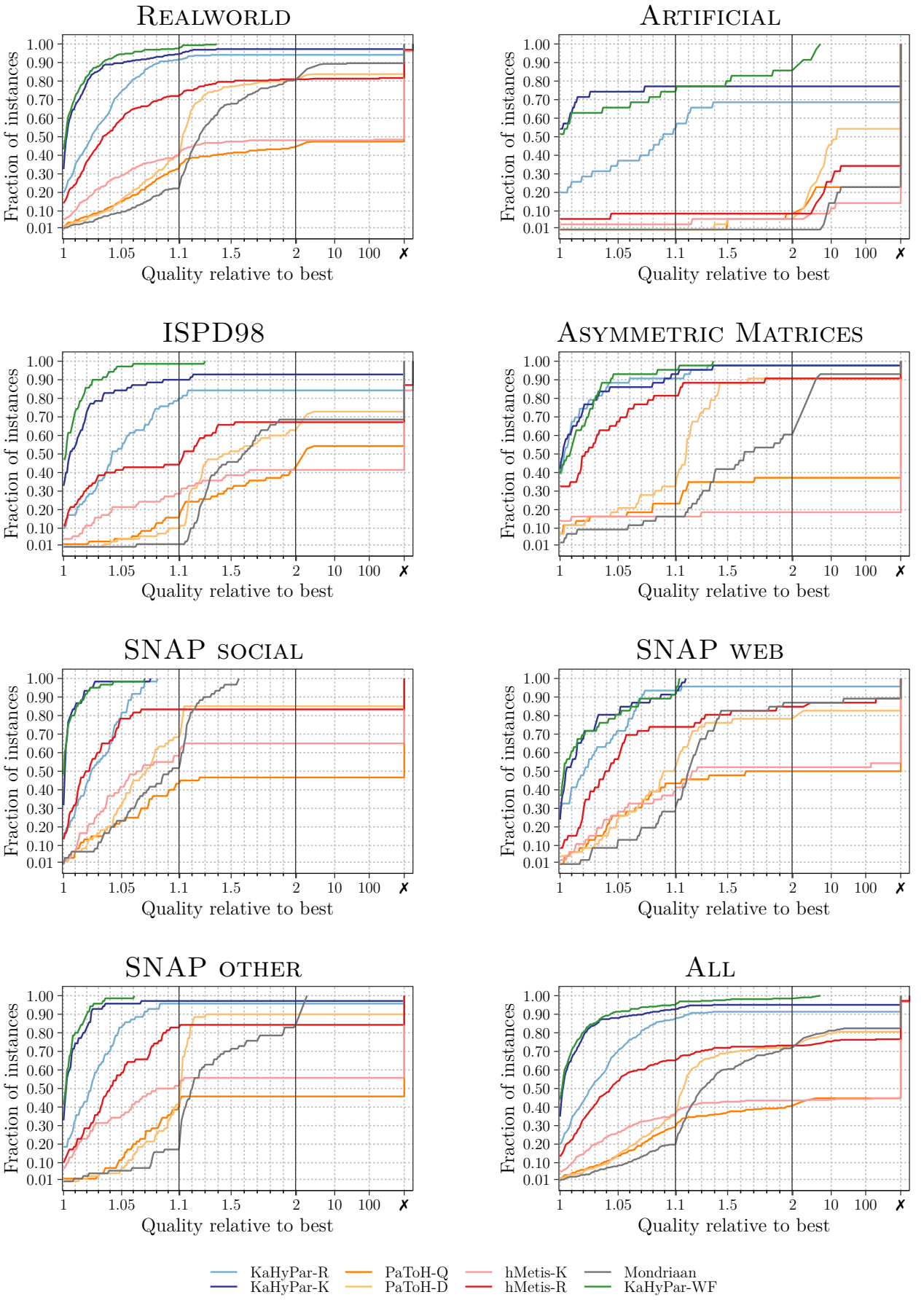

Figure 17: Connectivity performance profiles for our configuration and state-of-the-art partitioners with $\varepsilon = 0.01$ on the full benchmark set.

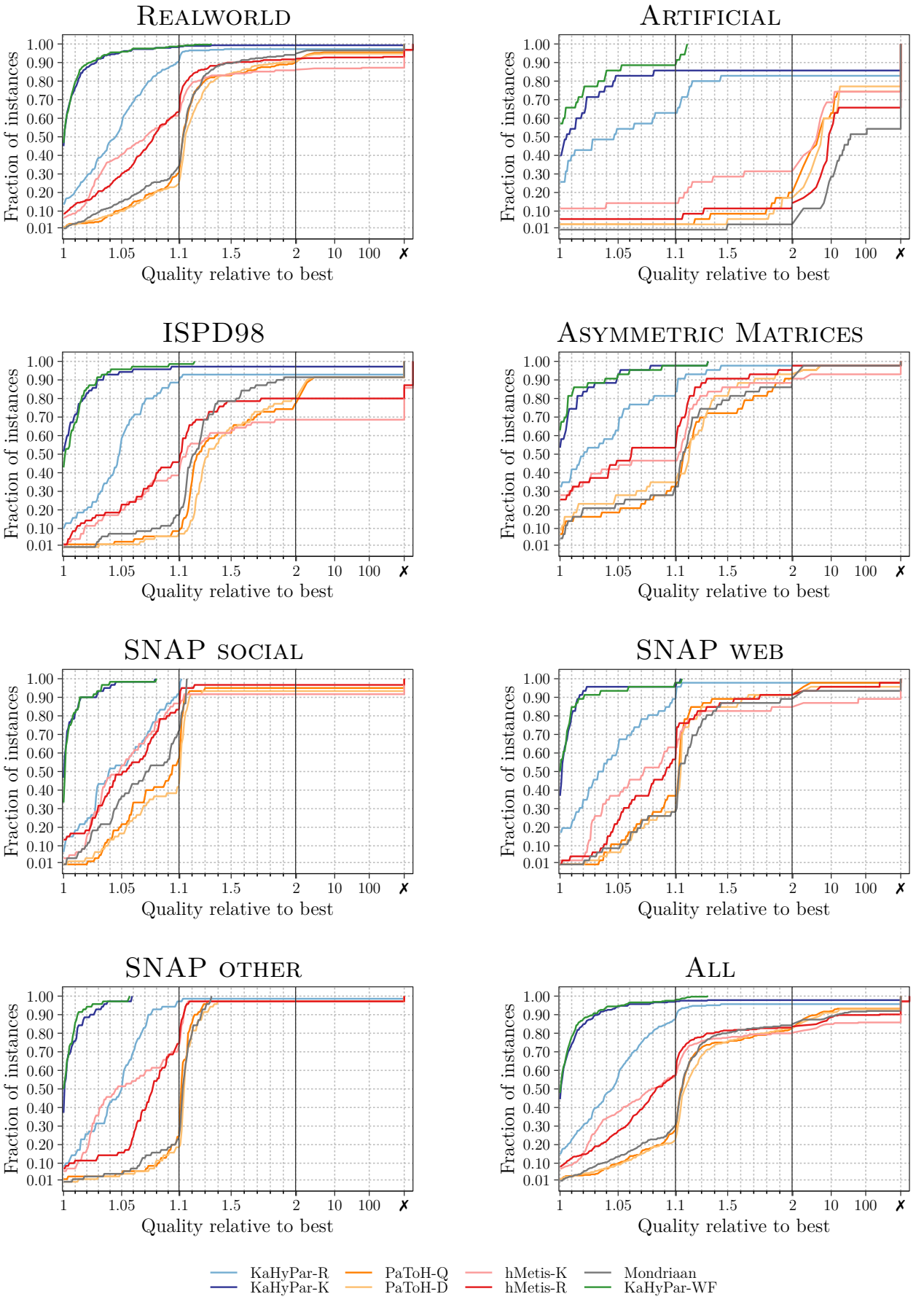

Figure 18: Connectivity performance profiles for our configuration and state-of-the-art partitioners with $\varepsilon = 0.1$ on the full benchmark set.