

k -way Hypergraph Partitioning via n -Level Recursive Bisection

Sebastian Schlag* Vitali Henne*
Peter Sanders*

Tobias Heuer* Henning Meyerhenke*
Christian Schulz*

Abstract

We develop a multilevel algorithm for hypergraph partitioning that contracts the vertices one at a time. Using several caching and lazy-evaluation techniques during coarsening and refinement, we reduce the running time by up to two-orders of magnitude compared to a naive n -level algorithm that would be adequate for ordinary graph partitioning. The overall performance is even better than the widely used hMetis hypergraph partitioner that uses a classical multilevel algorithm with few levels. Aided by a portfolio-based approach to initial partitioning and adaptive budgeting of imbalance within recursive bipartitioning, we achieve very high quality. We assembled a large benchmark set with 310 hypergraphs stemming from application areas such VLSI, SAT solving, social networks, and scientific computing. We achieve significantly smaller cuts than hMetis and PaToH, while being faster than hMetis. Considerably larger improvements are observed for some instance classes like social networks, for bipartitioning, and for partitions with an allowed imbalance of 10%. The algorithm presented in this work forms the basis of our hypergraph partitioning framework *KaHyPar* (**K**arlsruhe **H**ypergraph **P**artitioning).

1 Introduction

Hypergraphs are a generalization of graphs, where each (hyper)edge can connect more than two vertices. The k -way hypergraph partitioning problem is the generalization of the well-known graph partitioning problem: Partition the vertex set into k disjoint blocks of bounded size (at most $1 + \varepsilon$ times the average block size), while minimizing the total cut size, i.e., the sum of the weights of those hyperedges that connect multiple blocks. However, allowing hyperedges of arbitrary size makes the partitioning problem more difficult in practice [1, 2].

Hypergraph partitioning (HGP) has a wide range of applications. Two prominent areas are VLSI design and scientific computing (e.g. accelerating sparse matrix-vector multiplications) [3]. While the former is an example of a field where small optimizations can lead

to significant savings, the latter is an example where hypergraph-based modeling is more flexible than graph-based approaches [2, 4, 5, 6]. HGP also finds application as a preprocessing step in SAT solving, where it is used to identify groups of connected variables [7].

Since hypergraph partitioning is NP-hard [8] and since it is even NP-hard to find good approximate solutions for graphs [9], heuristic algorithms are used in practice. The most commonly used heuristic is the *multilevel paradigm* [10, 11, 12, 13]. It consists of three phases: In the *coarsening phase*, the hypergraph is recursively coarsened to obtain a hierarchy of smaller hypergraphs that reflect the basic structure of the input. After applying an *initial partitioning* algorithm to the smallest hypergraph in the second phase, coarsening is undone and, at each level, a *local search* method is used to improve the partition induced by the coarser level.

State-of-the-art hypergraph partitioners use matching- or clustering-based algorithms to find groups of highly-connected vertices that can be contracted together to create the next level of the coarsening hierarchy [4, 14, 15]. The rate at which successively coarser hypergraphs are reduced determines the number of levels in the multilevel hierarchy. As already noted in [16], a larger number of levels potentially improves the solution quality, because local search algorithms are used more often. However, it also leads to larger running times and increased memory usage.

In this paper we explore the idea to evade this trade-off by going to the extreme case of (nearly) n levels and removing only a single vertex between two levels. Implementing this in a sophisticated way, we are able to combine high quality with good performance. Our system turns out to be faster than the widely used hMetis system (see Section 5).

Outline and Contribution. After giving a brief overview of related work in Section 2 and introducing basic notation in Section 3, we explain how to compute a k -way partition via recursive bisection in Section 4.1. We show how to derive good values for the balance constraint of each bipartitioning subproblem so that in the end, the balance constraint for the k -way partition is fulfilled. Section 4.2 describes our coarsening strategy. A carefully chosen rating function evaluates how attrac-

*Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Algorithmics II. Email: {sebastian.schlag, meyerhenke, sanders, christian.schulz}@kit.edu, tobias.heuer@gmx.net, vitali.henne@gmail.com

tive it is to contract two vertices. We present an effective strategy to limit the cost for reevaluating the rating function.

The initial partitioner described in Section 4.3 is based on a large portfolio of simple algorithms, each with some randomization aspect (fully random, BFS, label propagation, and nine variants of greedy hypergraph growing). Since these partitioners are very fast and only applied to a small core problem, we can afford to make a large number of attempts taking the best partition as the basis for further processing.

Local improvement steps are expensive since many steps are needed and a naive implementation of the established techniques needs work proportional to the *squares* of the net sizes. We integrate several techniques for reducing this bad behavior for large nets and additionally develop a way to *cache* gain values to further reduce search overhead in Section 4.4. In Section 4.5 we introduce a hypergraph data structure that supports fast contraction and uncontraction of vertex pairs. We evaluate our algorithm and the main competitors on a broad range of hypergraphs derived from well established benchmark sets. The experiments reported in Section 5 indicate that our algorithm computes partitions that are significantly smaller than hMetis and PaToH with considerably larger improvements for special instance classes like social networks, for bipartitioning, and partitions with an allowed imbalance of 10%. At the same time, our algorithm is faster than hMetis. Section 6 concludes the paper.

2 Related Work

Since the 1990s HGP has evolved into a broad research area. We therefore refer to [3, 15, 17, 18] for an extensive overview. Here, we focus on issues closely related to the contributions of our paper. The two most widely used general-purpose tools are PaToH [4] (originating from scientific computing) and hMetis [14, 19] (originating from VLSI design). Other software packages with certain distinguishing characteristics are known, in particular Mondriaan [20] (sparse matrix partitioning), MLPart [16] (circuit partitioning), Zoltan [21] and Parkway [22] (parallel), and UMPa [23] (directed hypergraph model, multi-objective). All of these tools are based on the multilevel paradigm and compute a k -way partition either directly [19, 22, 23, 24] or via recursive bisection [4, 14, 16, 20, 21]. The two most popular local search approaches are greedy algorithms [19, 23] or variations of the Fiduccia-Mattheyses (FM) heuristic [25]. FM-type algorithms move vertices to other blocks in the order of improvements in the objective. Unlike simple greedy methods, FM can escape local optima to some extent since it allows to worsen the objective temporar-

ily. Partitioners based on recursive bisection use FM-based local search algorithms [4, 14, 16, 20, 21], while direct k -way hypergraph partitioners employ greedy methods [19, 22, 23, 24].

n -level algorithms have been used in geometric data structures based on randomized incremental construction [26, 27] and as a preprocessing technique for route planning [28]. More specifically, KaHyPar owes many basic ideas to its graph partitioning (GP) ancestor KaSPar [29]. However, the implementation, the required design choices, and even the overall outcome are very different. KaSPar is a direct k -way partitioner that suffers from data structure overheads and the difficulty to integrate advanced global improvement methods based on flows [30] and shortest paths [31]. In contrast, KaHyPar is based on recursive bipartitioning and currently seems to be the method of choice for a wide range of hypergraph partitioning tasks. In particular, it is actually among the fastest codes available – it seems that the overheads of a dynamic graph data structure are outweighed by many other complex issues in hypergraph partitioning that are unaffected by an n -level approach. There is a previous attempt on hypergraph bipartitioning in a Bachelor thesis [32] with the approach to contract one hyperedge in each level. However, that system was very slow so that we decided to start from scratch. Our second attempt was a direct k -way n -level partitioner [33]. Despite several interesting ideas and best quality in the majority of experiments, the k -way algorithm has not been able to improve on the state of the art consistently in terms of the time-quality trade-off. However, we learned from that paper that recursive bipartitioning seems to be advantageous and thus decided to first focus on the highly optimized n -level recursive bipartitioner presented here.

3 Preliminaries

An *undirected hypergraph* $H = (V, E, c, \omega)$ is defined as a set of n vertices V and a set of m hyperedges E with vertex weights $c : V \rightarrow \mathbb{R}_{\geq 0}$ and hyperedge weights $\omega : E \rightarrow \mathbb{R}_{> 0}$, where each hyperedge is a subset of the vertex set V (i.e., $e \subseteq V$). In the HGP literature, hyperedges are also called *nets* and the vertices of a net are called *pins* [4]. We extend c and ω to sets, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e if $\{v\} \subseteq e$. $I(v)$ denotes the set of all incident nets of v . The *degree* of a vertex v is $d(v) := |I(v)|$. Two vertices are *adjacent* if there exists a net e that contains both vertices. The set $\Gamma(v) := \{u \mid \exists e \in E : \{v, u\} \subseteq e\}$ denotes the neighbors of v . The *size* $|e|$ of a net e is the number of its pins. Nets of size one are called *single-node* nets. If $e_i = e_j$ we call nets e_i and e_j *parallel*.

A *k-way partition* of a hypergraph H is a partition of its vertex set into k *blocks* $\Pi = \{V_1, \dots, V_k\}$ such that $\bigcup_{i=1}^k V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. We use $b[v]$ to refer to the block id of vertex v . A *k-way partition* decomposes the hypergraph into k *section hypergraphs* [34] $H \times V_i := (V_i, \{e \in E \mid e \subseteq V_i\})$. We call a *k-way partition* Π ε -*balanced* if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq L_{max} := (1 + \varepsilon) \lceil \frac{c(V)}{k} \rceil$ for some parameter ε . We call a block V_i *overloaded* if $c(V_i) > L_{max}$ and *underloaded* if $c(V_i) < L_{max}$. Given a *k-way partition* Π , the number of pins of a net e in block V_i is defined as $\Phi(e, V_i) := |\{v \in V_i \mid v \in e\}|$. Net e is *connected* to block V_i if $\Phi(e, V_i) > 0$. Similarly, a block V_i is *adjacent* to a vertex $v \notin V_i$ if $\exists e \in I(v) : \Phi(e, V_i) > 0$. $R(v)$ denotes the set of all blocks adjacent to v . We call a net *internal* if $\Phi(e, i) = |e|$ for one block V_i and *cut* net otherwise. Analogously, a vertex contained in at least one cut net is called *border vertex*. The *k-way hypergraph partitioning problem* is to find an ε -balanced *k-way partition* of a hypergraph H that minimizes the *total cut* $\omega(E')$ for some ε , where E' is the set of all cut nets. This problem is known to be NP-hard [8].

Contracting a pair of vertices (u, v) means merging v into u . We refer to u as the *representative* and v as the *contraction partner*. The weight of u becomes $c(u) := c(u) + c(v)$. We connect u to the former neighbors $\Gamma(v)$ of v , by replacing v with u in all nets $e \in I(v) \setminus I(u)$. Furthermore we remove v from all nets $e \in I(u) \cap I(v)$. *Uncontracting* a vertex u reverses the contraction. The uncontracted vertex v is put in the same block as u and the weight of u is set back to $c(u) := c(u) - c(v)$.

4 n-Level Hypergraph Partitioning

We now present our main contributions. A high-level overview of our *n*-level hypergraph partitioning framework is provided in Algorithm 1. In Section 4.1, we start by explaining how to compute a *k-way partition* via recursive bisection. As other multilevel algorithms our algorithm has a coarsening, initial partitioning and an uncoarsening phase. During the coarsening phase, we successively shrink the hypergraph by contracting only a *single pair* of vertices *at each level*, until it is small enough to be initially partitioned. We describe the details of our coarsening algorithm in Section 4.2 and briefly discuss our portfolio-based initial partitioning approach in Section 4.3. The initial solution is transferred to the next finer level by performing a *single* uncontraction step. Afterwards, our localized local search algorithm described in Section 4.4 is used to further improve the solution quality. All algorithms use the hypergraph data structure described in Section 4.5.

Algorithm 1: Algorithm Overview

Input: Hypergraph H , lowest block id k_l , highest block id k_h , imbalance parameter ε .

Algorithm **partition**($H := (V, E), \varepsilon, k_l, k_h$)

```

 $k = k_h - k_l + 1$ 
// partition  $H$  into  $k$  blocks with block ids  $k_l, \dots, k_h$ 
 $\Pi_k := \emptyset$ 
if  $k_l = k_h$  then  $\Pi_k := V$ ; return  $\Pi_k$ 
 $\varepsilon' :=$  calculate according to Theorem 4.1
// coarsening phase
while  $H$  is not small enough do
    // choose vertex pair with highest rating
     $(u, v) := \text{argmax}_{u \in V} \text{score}(u)$ 
     $H := \text{contract}(H, u, v)$            //  $H := H \setminus \{v\}$ 
// initial partitioning phase
 $\Pi_2 = (V_0, V_1) := \text{computeBisection}(H, \varepsilon')$ 
// uncoarsening and local search phase
while  $H$  is not completely uncoarsened do
     $(H, \Pi_2, u, v) := \text{uncontract}(H, \Pi_2)$ 
     $(H, \Pi_2) := \text{localSearch}(H, \Pi_2, u, v, \varepsilon')$ 
// recurse on section hypergraphs
 $\Pi_k := \Pi_k \cup \text{partition}(H \times V_0, k_l, k_l + \lfloor k/2 \rfloor - 1, \varepsilon)$ 
 $\Pi_k := \Pi_k \cup \text{partition}(H \times V_1, k_l + \lfloor k/2 \rfloor, k_h, \varepsilon)$ 
return  $\Pi_k$ 
```

Output: ε -balanced *k-way partition* $\Pi = \{V_1, \dots, V_k\}$

4.1 k-way Partitioning via Recursive Bisection.

There are two approaches for computing a *k-way partition* within the multilevel framework. In *direct k-way partitioning*, the initial partitioning algorithm computes a *k-way partition*, which is then improved during uncoarsening using *k-way local search algorithms*. The most commonly used approach in HGP, however, is to use recursive bisection [35]. If k is a power of two, the final *k-way partition* is obtained by first computing a bisection of the initial hypergraph and then recursing on each of the two blocks. Thus it takes $\log_2(k)$ such phases until the hypergraph is partitioned into k blocks. If k is not a power of two, the approach has to be adapted to produce appropriately sized partitions. Our algorithm uses the following technique to compute a *k-way partition* via recursive bisection for arbitrary values of k : We bisect the hypergraph such that one block has a maximum weight of $(1 + \varepsilon') \lceil \lfloor k/2 \rfloor / k c(V) \rceil$ and the other block has a maximum weight of $(1 + \varepsilon') \lceil \lceil k/2 \rceil / k c(V) \rceil$, where ε' is an adapted imbalance parameter that ensures that the final *k-way partition* is ε -balanced. The former block is then partitioned recursively into $k' := \lfloor k/2 \rfloor$ blocks, while the latter is partitioned into $k' := \lceil k/2 \rceil$ blocks. After each bisection step, we therefore have to solve two k' -

way hypergraph partitioning problems. The new imbalance parameter ε' is chosen according to the following lemma:

LEMMA 4.1. *Let $H_0 = H \times V_0$ and $H_1 = H \times V_1$ be the section hypergraphs induced by a bipartition $\Pi = \{V_0, V_1\}$ of a hypergraph $H = (V, E, c, \omega)$ for which we wish to compute an ε -balanced k -way partition. Using an adaptive imbalance parameter*

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

to compute a k' -way partition (with $k' \geq 2$) of a hypergraph H_i via recursive bisection ensures that the final k -way partition of H is ε -balanced. When computing the very first bisection for a k -way partition, we set $H_0 = H$, $k' = k$ and therefore $\varepsilon' := (1 + \varepsilon)^{(1/\lceil \log_2(k) \rceil)} - 1$.

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

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

In order to ensure that the original k -way partition of H is ε -balanced, we therefore have to choose ε in Equation 4.1 such that $c(V_{max}) \leq L'_{max}$.

Thus when recursively partitioning a section hypergraph H_i with weight $c(V_i)$ into k' blocks we choose a new imbalance parameter ε' as follows:

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

Note that by definition, the section hypergraphs do not contain any cut nets. The cut nets of each bipartition can be discarded, since they will always be cut nets

in the final k -way partition and already contribute $\omega(e)$ to the total cut size [36]. This simultaneously reduces the number of nets as well as their average size in each section hypergraph, without affecting the partitioning objective.

4.2 Coarsening. The goal of the coarsening phase is to contract highly connected vertices such that the number of nets remaining in the hypergraph and their size is successively reduced [14]. Removing nets leads to simpler instances for initial partitioning, while small net sizes allow FM-based local search algorithms to identify moves that improve solution quality. Thus, we choose the vertex pairs (u, v) to be contracted according to a rating function. More precisely, we adopt the *heavy-edge* rating function also used by hMetis [14], Parkway [22] and PaToH [36], which prefers vertex pairs that have a large number of heavy nets with small size in common. However, in contrast to these tools, we additionally scale this score inversely with the product of the vertex weights $c(v)$ and $c(u)$ to keep the vertex weights of the coarse hypergraph reasonably uniform:

$$(4.3) \quad r(u, v) := \frac{1}{c(v) \cdot c(u)} \sum_{e \in \{I(v) \cap I(u)\}} \frac{\omega(e)}{|e| - 1}.$$

This is also done in the matching-based coarsening algorithm of MLPart [16] for similar reasons.

Algorithm Outline. At the beginning of the coarsening algorithm, all vertices are rated, i.e., for each vertex u we compute the ratings of all neighbors $\Gamma(u)$ and choose the vertex v with the highest rating as contraction partner for u . Ties are broken randomly to increase diversification. For each vertex, we then insert the vertex pair with the highest score into an addressable priority queue (PQ) using the rating score as key. This allows us to efficiently choose the best-rated vertex pair that should be contracted next. In each iteration, we remove the pair (u, v) with the highest score and contract it. After contraction, the entry of v is removed from the PQ, since v is no longer contained in the hypergraph.

A contraction operation can lead to parallel nets (i.e., nets that contain exactly the same vertices) and single-node nets in $I(u)$. In order to reduce the running time of the coarsening algorithm, we remove these nets from the hypergraph. Single-node nets are easily identified, because $|e| = 1$. In case of parallel nets, we remove all but one from H . The weight of the remaining net e is set to the sum of the weights of the nets that were parallel to e . Parallel nets are detected using an algorithm similar to the one in [37], which identifies vertices with identical structure in a graph:

We create a fingerprint for each net $e \in I(u)$: $f_i := \bigoplus_{v \in e} v \oplus x$, for some seed x .¹ These fingerprints are then sorted and a final scan then identifies parallel nets: If two consecutive fingerprints f_i, f_j are identical, we check whether the nets are truly parallel by comparing their pins². Each contraction potentially influences the ratings of some neighbors $\Gamma(u)$. The *full* re-rating strategy therefore recalculates these ratings and updates the PQ accordingly. To avoid imbalanced inputs for the initial partitioning phase, vertices v with $c(v) > c_{max} := s \cdot \lceil \frac{c(V)}{t} \rceil$ are never allowed to participate in a contraction step and are thus removed from the PQ. The coarsening process is stopped as soon as the number of vertices drops below t or no eligible vertex is left. Parameter s is used to favor the contraction of highly connected vertices. Both parameters will be chosen in Section 5.

Advanced Update Strategy. Continuously re-rating the neighbors $\Gamma(u)$ of a representative u is the most expensive part of the algorithm: After each contraction, we have to look at all pins of all incident nets $I(u)$. The re-rating can therefore easily become the bottleneck – especially if H contains large nets or high degree vertices. To improve the running time in these cases, we developed a variation of the *full* strategy: Our *lazy* strategy does not re-rate any vertices immediately after contraction. Instead, all adjacent vertices $\Gamma(u)$ are marked as *invalid*. If the PQ returns an invalid vertex, we recalculate its rating and update the priority queue accordingly.

4.3 Initial Partitioning. Coarsening is performed until the coarsest hypergraph is small enough to be partitioned by an initial partitioning algorithm. We use a portfolio of several algorithms to compute an initial bipartition. Each algorithm is run twenty times. We select the partition with the best cut and lowest imbalance to be projected back to the original hypergraph. In case all partitions are imbalanced, we choose the partition with smallest imbalance. The portfolio-based approach increases diversification and produced the best results in [38]. In the following, we give a brief overview of the algorithms used and refer to [38] for a more detailed description and evaluation. *Random partitioning* randomly assigns vertices to a block. *Breadth-First-Search* (BFS) starts with a randomly chosen vertex and performs a BFS traversal of the hypergraph until it has discovered half of the hypergraph. The vertices visited during the traversal constitute block V_0 , all remaining vertices constitute V_1 . Furthermore, we use different vari-

ations of *Greedy Hypergraph Growing* (GHG) [4]. Each version first computes two pseudo-peripheral vertices: Starting from a random vertex, we perform a BFS. The last vertex visited serves as the start vertex for the next BFS. This vertex and the last vertex visited by the second BFS are supposed to be “far” away from each other. Therefore one is used as the seed vertex for block V_0 , the other for block V_1 . For each block, we maintain a PQ that stores the neighboring vertices of the growing cluster according to a score function. We use the FM gain, which will be described in the next section, as well as the *Max-Net* and *Max-Pin* gain definitions (which are also used in PaToH [4]) as score functions. Our GHG variants also differ in the way the clusters are grown. *Greedy-Global* always moves the vertex with the highest score in both PQs to the corresponding block. *Greedy-Sequential* first grows block V_0 and then block V_1 , while *Greedy-Round-Robin* grows both blocks simultaneously. This again increases diversification. In total, we thus have nine different initial partitioning algorithms based on GHG. The last algorithm is based on *size-constrained label propagation* (*SCLaP*) [39]. Each vertex has a label representing its block. Initially all labels are empty. The algorithm starts by searching two pseudo peripheral vertices via BFS. One vertex along with τ of its neighbors gets label V_0 , while the other vertex and τ of its neighbors get label V_1 . We then perform label propagation until the algorithm has converged, i.e. no empty labels remain. Vertices with the same label then become the blocks of the partition. The tuning parameter τ is used to prevent labels from completely disappearing in the course of the algorithm. Based on the results in [38], we set τ to five in our experiments.

4.4 Localized FM Local Search. Our local search algorithm is similar to the FM-algorithm [25] and is further inspired by the algorithm used in KaSPar [29] for graph partitioning. A key difference to traditional FM is the way a local search pass is started: Instead of initializing the algorithm with all vertices or all border vertices, we perform a highly localized search starting only with the representative and the just uncontracted vertex. The search then gradually expands around this vertex pair by successively considering neighboring vertices. Traditional multilevel FM implementations as well as KaSPar *always* compute the gain of *each* vertex *from scratch* at *each* level of the hierarchy. During an FM pass, these values are then kept up-to-date by delta-gain-updates [4]. Since our algorithm starts only around two vertices, many gain values would never be used during a local search pass. We therefore maintain a *gain cache* that ensures that the gain of a vertex move is calculated at most *once* during *all* local searches along

¹ \oplus is the bitwise XOR

²In principle, this approach could be accelerated using hashing. Parallel net detection, however, did not significantly contribute to the overall running time of our algorithm.

the n -level hierarchy.

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

$$(4.4) \quad g_i(v) := \sum_{e \in I(v)} \{\omega(e) : \Phi(e, i) = |e| - 1\} - \sum_{e \in I(v)} \{\omega(e) : \Phi(e, b[v]) = |e|\}.$$

After insertion, PQs corresponding to *underloaded* blocks become enabled. Since a move to an overloaded block will never be feasible, a queue corresponding to an overloaded block is left disabled. The algorithm then repeatedly queries only the *non-empty, enabled* queues to find the move with the highest gain $g_i(v)$, breaking ties arbitrarily. Vertex v is then moved to block V_i and labeled inactive and marked. We then update all neighbors $\Gamma(v)$ of v as follows: All previously inactive neighbors are activated as described above. Neighbors that have become internal are labeled inactive and the corresponding moves are deleted from the PQs. Finally, we perform *delta-gain-updates* for all moves of the remaining active border vertices in $\Gamma(v)$: If the move changed the gain contribution of a net $e \in I(v)$, we account for that change by incrementing/decrementing the gains of the corresponding moves by $\omega(e)$ using the delta-gain-update algorithm of Papa and Markov [3].

To further decrease the running time, we exclude nets from gain update that can not be removed from the cut in the current local search pass. A net is *locked* in the bipartition, once it has at least one disabled pin in each of the two blocks [40]. In this case, it is not possible to remove such a net from the cut by moving any of the remaining movable pins to another block. Thus it is not necessary to perform any further delta-gain-updates for locked nets, since their contribution to the gain values of their pins does not change any more. This observation was first described by Krishnamurthy [40]. We integrate locking of nets into our algorithm by labeling each net during a local search pass. Initially, all nets are labeled

free. Once the first pin of a net is moved, the net becomes *loose*. It now has a pin in one block that cannot be moved again. Further moves to this block do not change the label of the net. As soon as another pin is moved to the other block, the net is labeled *locked* and is excluded from future delta-gain-updates.

Once all neighbors are updated, local search continues until either no non-empty, enabled PQ remains or a constant number of c moves neither decreased the cut nor improved the current imbalance. The latter criterion is necessary, because otherwise the n -level approach could lead to $|V|^2$ local search steps in total. After local search is stopped, we reverse all moves until we arrive at the lowest cut state reached during the search that fulfills the balance constraint. All vertices become unmarked and inactive and the algorithm is then repeated until no further improvement is achieved.

Caching of Gain Values. We briefly outline the details of the gain cache. Let $c[v]$ denote the cache entry for vertex v . After initial partitioning, the gain cache is empty. If a vertex becomes activated during a local search pass, we check whether or not its gain is already cached. If it is cached, the cached value is used for activation. Otherwise, we calculate the gain according to Eq. (4.4), insert it into the cache and activate the vertex. After moving a vertex v with gain $g_i(v)$ to block V_i , its cache value is set to $c[v] := -g_i(v)$. The delta-gain updates of its neighbors $\Gamma(v)$ are then also applied to the corresponding cache entries. Thus the gain cache always resembles the current state of the hypergraph. Since our algorithm performs a rollback operation at the end of a local search pass that undoes vertex moves, we also have to undo delta-gain updates applied on the cache. This can be done by additionally maintaining a *rollback delta cache* that stores the negated delta-gain updates for each vertex. During rollback, this delta cache is then used to restore the gain cache to a valid state. Each time a local search is started with an uncontracted vertex pair (u, v) , we have to account for the fact that the uncontraction potentially affected $c[u]$. A simple variant of the caching algorithm just invalidates the corresponding cache entry and recalculates the gain. Since v did not exist on previous levels of the hierarchy, its gain must also be computed from scratch.

We now describe a more sophisticated variant that is able to update $c[u]$ based on information gathered during the uncontraction and that further infers $c[v]$ from $c[u]$.

After uncontraction, we initially set $c[v] := c[u]$. Both cache entries are then updated by examining each net $e \in I(u)$. We have to distinguish three cases (see Figure 1 for an example):

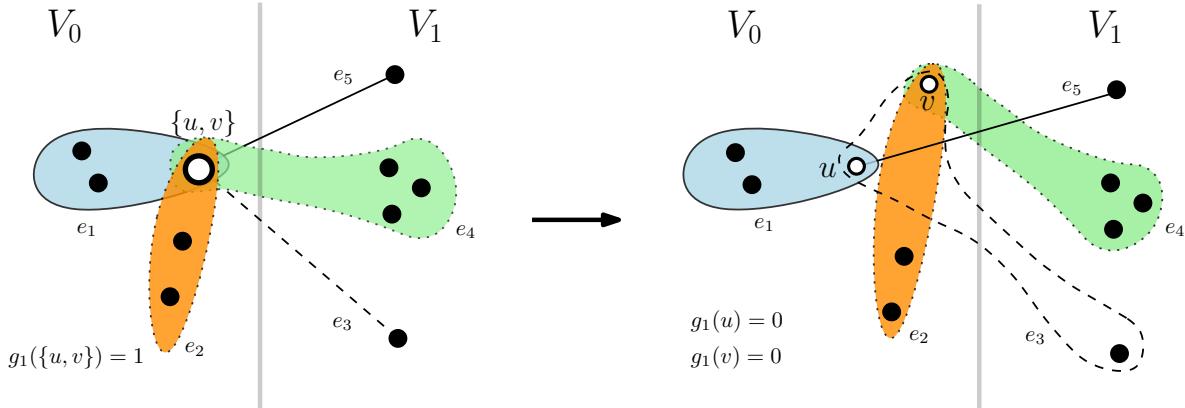


Figure 1: Example of an uncontraction operation that affects the cached gain value $c[u]$ of the representative u . Since nets $\{e_2, e_4\} \notin I(u)$ after the uncontraction, they no longer contribute $-\omega(e_2)$ resp. $\omega(e_4)$ to the gain of u . Similarly, moving u to V_1 now does not remove e_3 from the cut anymore because of v . Its contribution to $c[u]$ therefore becomes zero.

1. After uncontraction, u is not incident to net e any more. If e was a cut net that could have been removed from the cut by moving u to the other block, $c[u]$ has to be decremented by $\omega(e)$ (e_4 in Figure 1). Similarly, if e was an internal net, moving u would have made it a cut net. In this case, $c[u]$ is incremented by $\omega(e)$ (e_2 in Figure 1).
2. After uncontraction, e contains both u and v . If $\Phi(e, b[v]) = 2$, the net cannot be removed from the cut anymore by moving either u or v . We therefore have to decrement both $c[u]$ and $c[v]$ by $\omega(e)$ (e_3 in Figure 1).
3. Finally, we have to account for nets to which v is not incident (e_1 and e_5 in Figure 1). If such a net e can be removed from the cut by moving u , it contributes $\omega(e)$ to $c[u]$. We therefore have to decrement $c[v]$ by $\omega(e)$ to account for the fact that $e \notin I(v)$. Similarly, if moving u makes e a cut net, we have to increment $c[v]$ accordingly.

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

to relink the edge (v, e) to u . A modified adjacency array is used to represent G . It is divided into two offset arrays V , E and an incidence array A that stores the edges leaving v for each node $v \in V \dot{\cup} E$. V stores the starting positions of the entries in A ($V[\cdot].f$) and $d(v)$ ($V[\cdot].s$). E stores the same for nodes representing the nets of H . Nets incident to a vertex v are accessible as $A[V[v].f], \dots, A[V[v].f + V[v].s - 1]$. The pins of a net e are accessed similarly using offset array E . An example is shown in Figure 2. Although the concept of relinking and removing edges remains the same for GP [29] and HGP, the actual implementation is different: While an adjacency array used as a graph data structure allows to access all neighbors $\Gamma(v)$ of node v , we can only access the nets $I(v)$ of a vertex v in HGP. Since [29] does not provide any implementation details, we give a brief description on how contraction and uncontraction operations are implemented in our hypergraph data structure.

Contraction. Contracting a vertex pair $(u, v) \in H$ works as follows: For each net $e \in I(v)$ we have to determine if the corresponding edge $(v, e) \in G$ can simply be deleted or if a relink operation is necessary. This can be done with one iteration over the pins of e . During this iteration, we swap v with the last pin of e located at position $A[E[e].f + E[e].s - 1]$ and additionally search for vertex u . If we found u , then there is no need to perform a relink operation and we can remove v from e by simply decrementing $E[e].s$. If u was not found, we have to relink e to u . A relink operation adds the undirected edge (u, e) to G . To achieve this in our data structure, we have to add e to the subarray of u and vice versa. The latter can be accomplished by reusing the pin slot of v : After the iteration over the pins of e , v is the last entry in the subarray of e .

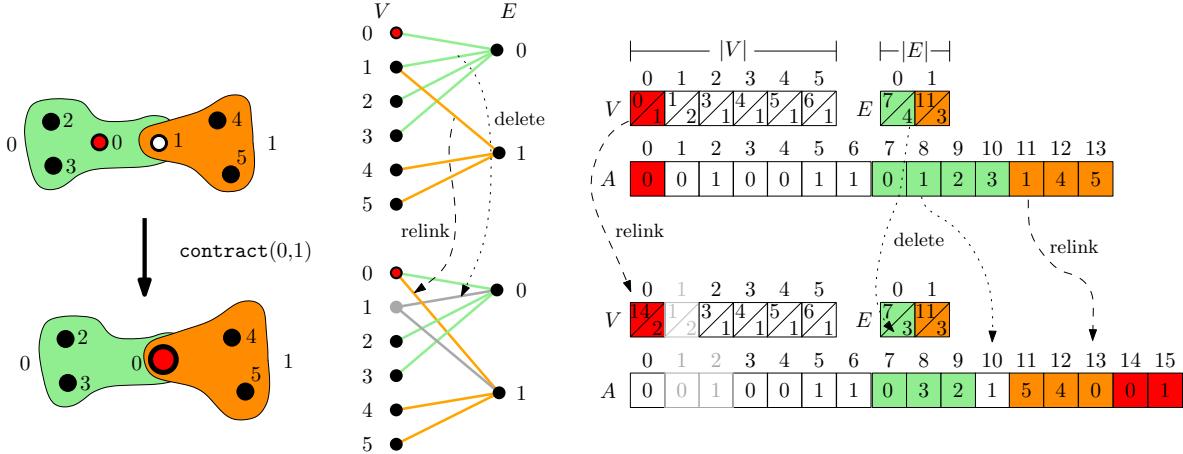


Figure 2: Example of a contraction operation. The hypergraph H is depicted on the left, the corresponding bipartite graph representation is shown in the middle and the adjacency data structure is shown on the right. The contraction leads to an edge deletion operation for net 0 and a relink operation for net 1.

Algorithm 2: Contract

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

- 1 $\mathcal{M} := \{u, v, V[u].f, V[u].s\}$
- 2 $c(u) := c(u) + c(v)$
- 3 **copy** := True
- 4 **foreach** $e \in I(v)$ **do**
- 5 $\tau, l := E[e].f + E[e].s - 1$
- 6 **for** $i := E[e].f; i \leq l; ++i$ **do**
- 7 **if** $A[i] = v$ **then**
- 8 **swap**($A[i], A[l]$), $--i$
- 9 **else if** $A[i] = u$ **then** $\tau := i$
- 10 **if** $\tau = l$ **then** // relink operation
- 11 $A[l] := u$
- 12 **if** **copy** **then**
- 13 $A.append(I(u))$
- 14 $V[u].f := |A| - V[u].s$
- 15 **copy** := False
- 16 $A.append(e)$
- 17 $++V[u].s$
- 18 **else** // delete operation
- 19 $--E[e].s$
- 20 $V[v].disable()$

Output: Contraction memento \mathcal{M}

Algorithm 3: Uncontract

Input: Contraction Memento \mathcal{M}

- 1 $V[\mathcal{M}.v].enable()$
- 2 // bitset
- 3 $b = [b_0, \dots, b_{m-1}] := [false, \dots, false]$
- 4 // assume no net contained u
- 5 **foreach** $e \in I(\mathcal{M}.v)$ **do**
- 6 $b[e] := \text{True}$
- 7 // these nets actually contained u
- 8 **for** $i := \mathcal{M}.f; i < \mathcal{M}.f + \mathcal{M}.s; ++i$ **do**
- 9 $b[A[i]] := \text{False}$
- 10 **if** $V[\mathcal{M}.u].s - \mathcal{M}.s > 0$ **then**
- 11 // reverse relink operations
- 12 **foreach** $e \in I(\mathcal{M}.u)$ **do**
- 13 **if** $b[e]$ **then**
- 14 **foreach** $p \in e$ **do**
- 15 **if** $p = \mathcal{M}.u$ **then** $p := \mathcal{M}.v$; break
- 16 $V[\mathcal{M}.u].f := \mathcal{M}.f$
- 17 $V[\mathcal{M}.u].s := \mathcal{M}.s$
- 18 $c(\mathcal{M}.u) := c(\mathcal{M}.u) - c(\mathcal{M}.v)$
- 19 **foreach** $e \in I(\mathcal{M}.v)$ **do** // reverse deletes
- 20 **if** $b[e] = \text{False}$ **then** $++E[e].size$

Setting $A[E[e].f + E[e].s - 1] := u$ therefore adds u to the pins of e and simultaneously removes v . The former is more difficult, because we actually have to extend the subarray of u . This is done by copying the subarray of u to the end of A , appending e and

updating $V[u].f$ and $V[u].s$ accordingly.³ To complete the contraction operation, we disable v to remove it from the hypergraph. Algorithm 2 gives the corresponding pseudocode. Note that the subarray of u is copied to

³This could be improved by leaving some free space in A for each node. The copying operations, however, had no noticeable effect on the running time of our algorithm.

the end of A at most once during a contraction as part of the first relink operation.

Uncontraction. To reverse a contraction, we store a memento \mathcal{M} for each contraction consisting of the contracted vertex pair (u, v) and the values $V[u].f$ and $V[u].s$ before the contraction. After re-enabling v , deletions can be reversed by simply increasing $E[e].size$ for the corresponding nets. To reverse a relink operation of an edge (u, e) , we first reset the pin slot containing u to v . Setting $V[u].f := \mathcal{M}.f$ and $V[u].s := \mathcal{M}.s$ then restores $I(u)$ to the state before the contraction. Deletion operations have to be reversed for all nets $\bar{D} := I(v) \cap \{A[\mathcal{M}.f], \dots, A[\mathcal{M}.f + M.s - 1]\}$, because these nets contained both u and v . All remaining nets $\bar{R} := I(v) \setminus \bar{D}$ require the reversal of a relink operation. A pseudocode description of the uncontraction operation can be found in Algorithm 3. After initial partitioning, we initialize $\Phi(e, V_i)$ for each cut net e . For constant-time border vertex checks, we additionally store the number of incident cut nets for each vertex. Both data structures are then maintained and updated during local search.

5 Experiments

System. All experiments are performed on a single core of a machine consisting of two Intel Xeon E5-2670 Octa-Core processors (Sandy Bridge) clocked at 2.6 GHz. The machine has 64 GB main memory, 20 MB L3-Cache and 8x256 KB L2-Cache and is running Red Hat Enterprise Linux (RHEL) 6.4.

Algorithm Configuration and Methodology. The algorithm is implemented in the n -level hypergraph partitioning framework *KaHyPar* (**Karlsruhe Hypergraph Partitioning**). The code is written in C++ and compiled using gcc-4.9.1 with flags `-O3 -mtune=native -march=native`. We performed a large number of experiments to tune the parameters of our algorithms on medium-sized VLSI and sparse matrix instances. The properties of these hypergraphs are summarized in Appendix A. A full description of these experiments is omitted due to space constraints. The following decisions are made based on the results summarized in Figure 8 and Table 2 in Appendix B: The coarsening process is stopped as soon as the number of vertices drops below $t = 320$ or no eligible vertex is left. The scaling factor s for the highest allowed vertex weight during coarsening is set to 3.25. Each local search pass is stopped as soon as $c = 350$ moves did not yield any improvement.

We perform ten repetitions with different seeds for each test instance and report the *arithmetic mean* of the computed cut and running time as well as the best cut found. When averaging over different instances, we

use the *geometric mean* in order to give every instance a comparable influence on the final result. In order to include instances with a cut of zero into the results, we set the corresponding cut values to *one* for geometric mean and ratio computations.

Instances. We evaluate our algorithm on hypergraphs derived from three benchmark sets: The ISPD98 VLSI Circuit Benchmark Suite [41], the University of Florida Sparse Matrix Collection [42] and the international SAT Competition 2014 [43]. From the latter, we randomly selected 100 instances from the application track and converted them into hypergraphs as follows: Each boolean variable (and its complement) is mapped to one vertex and each clause constitutes a net [3]. The Sparse Matrix Collection is organized into 172 groups and each group contains matrices of different application areas. From each group, we choose one matrix for each application area that has between 10 000 and 10 000 000 columns. In case multiple matrices fulfill our criteria, we randomly select one. In total, we include 192 matrices, which are translated into hypergraphs using the row-net model [4], i.e. each row is treated as a net and each column as a vertex. Empty rows are discarded. Both vertices and nets have unit weight. Together with the 18 VLSI instances derived from the ISPD98 Circuit Benchmark Suite, a total of 310 hypergraphs constitute our benchmark set. Each of these hypergraphs is partitioned into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks with $\varepsilon = 0.03$. For each value of k , a k -way partition is considered to be *one* test instance, resulting in a total of 2170 instances. We exclude 173 test instances, because either PaToH-Q could not allocate enough memory or other partitioners did not finish in time. The excluded instances are shown in Appendix C. Note that only PaToH-D was able to partition all instances within the given time limit. The comparison in Section 5.1 is therefore based on the remaining 1997 test instances.

In order to evaluate the performance of our algorithm for different values of ε , we perform additional experiments for a subset of the 293 hypergraphs used in Section 5.1, the same values of k , and $\varepsilon \in \{0.01, 0.1\}$. To estimate the number of hypergraphs necessary to produce the same qualitative results as presented in Section 5.1, we performed the following experiment: For each subset size possible, we took a random sample of that size and verified whether or not the results for the sample match the results for the full benchmark set by comparing the sorted min-cut ratios of KaHyPar to the ratios of all other partitioners. Only in case all ratios of KaHyPar were better than or equal to the ratios of all other partitioners, we counted the sample as a successful trial. Note that this is a rather strong requirement for the selected subset. Figure 3 summarizes these exper-

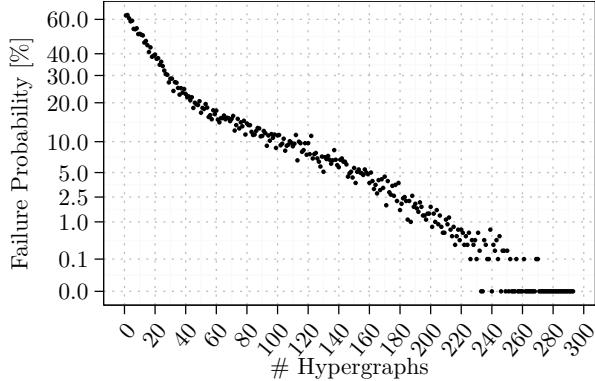


Figure 3: Probability that a random sample of a certain number of hypergraphs does not provide the same results as the full benchmark set. Each data point is based on 1 000 random samples.

iments. Each data point corresponds to 1000 random samples. Based on these results, we use a subset of 100 hypergraphs to be able to reproduce the results with a probability of 90%. The hypergraphs are chosen as follows: We use the 10 largest VLSI hypergraphs (ibm09 - ibm18), 30 randomly chosen SAT hypergraphs and 60 randomly chosen sparse matrix hypergraphs. This subset is used in the experiments presented in Section 5.2.

We compare our algorithms to both the k -way (hMetis-K) and the recursive-bisection variant (hMetis-R) of hMetis 2.0 (p1) [14, 19] and to PaToH 3.2 [4]. We choose these two tools because of the following reasons: PaToH produces better quality than Zoltan’s native parallel hypergraph partitioner (PHG) in serial mode [21, 44]. Parkway does not run in serial mode and was found to be comparable to Zoltan PHG in serial mode [21]. Furthermore, PaToH has been shown to compute better solutions than Mondriaan [45, 46] and MLPart [47]. Additionally, MLPart is restricted to bisections [48, 49]. UMPa does not improve on PaToH when optimizing single objective functions that do not benefit from the directed hypergraph model [50].

hMetis-R defines the maximum allowed imbalance of a partition differently [14]: An imbalance value of 5, for example, allows each block to weigh between $0.45 \cdot c(V)$ and $0.55 \cdot c(V)$ at each bisection step. We therefore translate our imbalance parameter $\varepsilon = 0.03$ to ε' as described in Eq. (5.5) such that it matches our balance constraint after $\log_2(k)$ bisections:

$$(5.5) \quad \varepsilon' := 100 \cdot \left(\left((1 + \varepsilon) \frac{\lceil \frac{c(V)}{k} \rceil}{c(V)} \right)^{\frac{1}{\log_2(k)}} - 0.5 \right)$$

PaToH is configured to use a final imbalance ratio of $\varepsilon \in \{0.01, 0.03, 0.1\}$ to match our balance constraint

for the corresponding experiment. Since it ignores the random seed if configured to use the quality preset, we report both the result of the quality preset (PaToH-Q) and the average over ten repetitions using the default configuration (PaToH-D). All partitioners have a time limit of 15 000s per test instance. The complete benchmark set, detailed statistics for each hypergraph and per-instance partitioning results for all experiments reported in this paper are publicly available [51].

Effects of Engineering Efforts. Engineering the algorithms of the coarsening and local search phase is critical to the overall performance of our n -level hypergraph partitioner. We partitioned the hypergraph derived from the sparse matrix `wb-edu` into two blocks as an example. Using the *lazy* re-rating strategy for coarsening reduces the running time of the coarsening phase by *two orders of magnitude*: While coarsening took 38349.6s using the full re-rating strategy, the lazy re-rating strategy only took 450.3s. In KaSPar [29] it was sufficient to calculate the gain values from scratch on each local search pass. However, this approach leads to poor performance in n -level hypergraph partitioning, although our implementation already employs several known speedup techniques (delta-gain-updates, locked nets). Without gain caching, local search took 2353.5s. Activating the caching mechanism reduced the running time by a factor of four down to 579.9s.

5.1 Comparison on full Benchmark Set. In 3700 out of 19970 experiments hMetis-K produced imbalanced partitions (up to 14% imbalance). KaHyPar produced ten imbalanced partitions (up to 4% imbalance) and one partition of PaToH-Q had 9% imbalance. All instances partitioned by hMetis-R and PaToH-D fulfilled the balanced constraint of 3%. In the following comparisons hMetis-K therefore has slight advantages because we do not disqualify imbalanced partitions.

Figure 4 summarizes the results of our experiments for $\varepsilon = 0.03$. For each algorithm, it relates the smallest average (left column) and minimum (right column) cut of all algorithms to the corresponding cut produced by the algorithm on a per-instance basis. For each algorithm, these ratios are sorted in increasing order. Note that these plots use a cube root scale for both axes to reduce right skewness [52]. Since the plot shows $1 - (\text{best}/\text{algorithm})$, a value of zero indicates that the corresponding algorithm produced the best solution. A point close to one indicates that the partition produced by the corresponding algorithm was considerably worse than the partition produced by the best algorithm. Thus an algorithm is considered to perform better than another algorithm, if its corresponding ratio values are below those of the other algorithm.

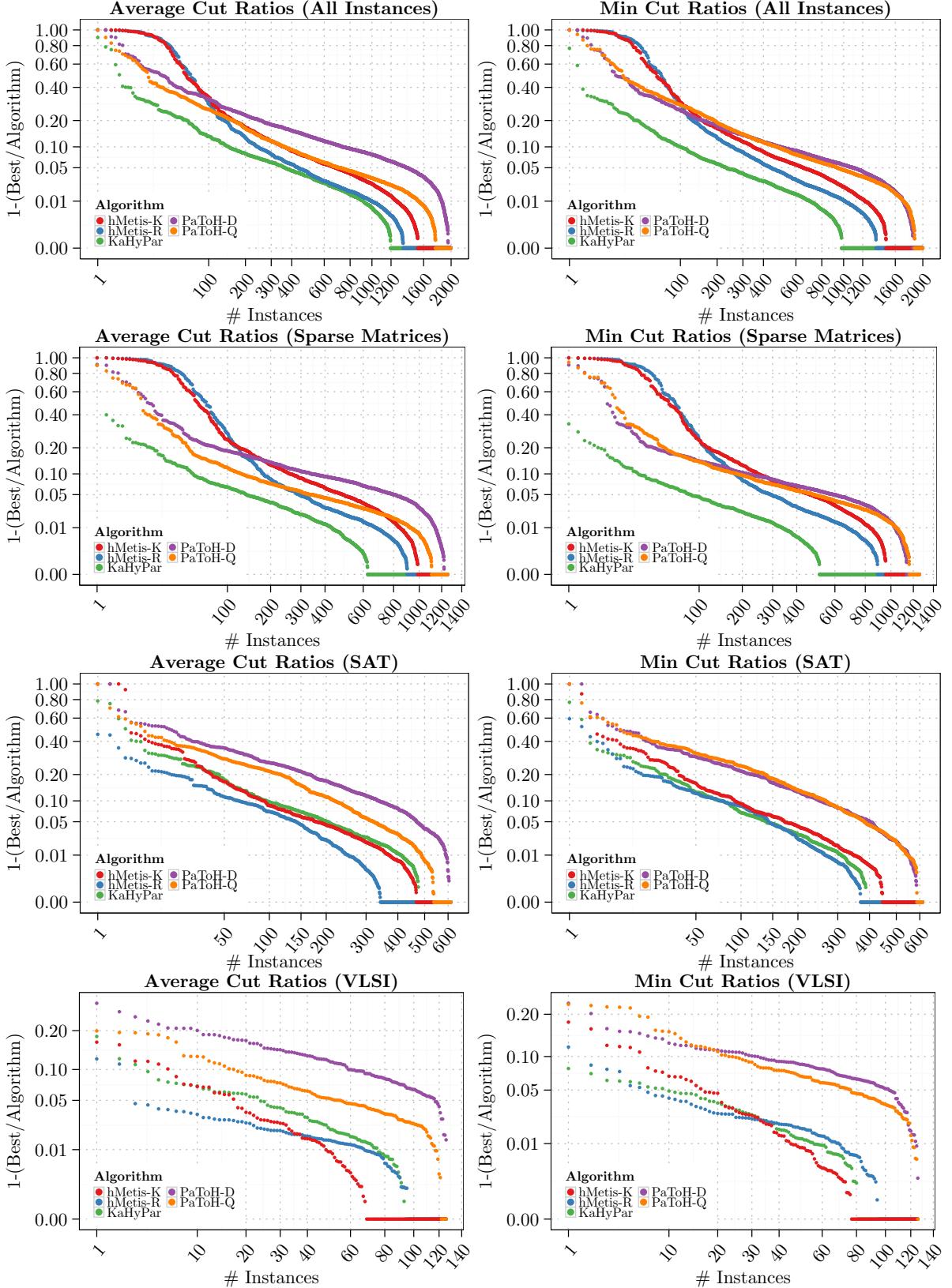


Figure 4: Performance plots for $\varepsilon = 0.03$. The y-axis shows the ratio between the smallest cut of all algorithms and the cut produced by the corresponding algorithm. Since we plot $1 - (\text{best}/\text{algorithm})$, a value of zero indicates that the corresponding algorithm produced the best solution. Note the cube root scale for both axes.

Looking at the solution quality across all instances (top), KaHyPar outperforms all other systems regarding both average and best solution quality. KaHyPar produced the best partitions for 1018 of the 1997 instances. It is followed by hMetis-R (643) and hMetis-K (519). PaToH-D computed the best partitions for 147 and PaToH-Q for 123 instances. Note that for some instances multiple partitioners computed the same solution. Comparing the best solutions of KaHyPar to each partitioner individually, KaHyPar produced better partitions than PaToH-Q, PaToH-D, hMetis-K, hMetis-R in 1742, 1701, 1247 and 1146 cases, respectively. Note that hMetis-R outperforms hMetis-K, although we had to tighten the balance constraint for hMetis-R and allowed imbalanced solutions for hMetis-K. Except for a small number of instances where of both hMetis variants compute partitions that are significantly inferior to the best solution, the hMetis variants perform better than both PaToH variants. The running times of each partitioner are compared in Figure 7. The plots show the ratio between the running time of each algorithm and the running time of the fastest algorithm on a per-instance basis. Ratios are again sorted in increasing order. Note that these plots use a log-scaled y-axis. PaToH is the fastest partitioner. Reasons for this are not only the smaller number of levels but also aggressive optimizations like ignoring large hyperedges during coarsening and a memory management that makes PaToH-Q fail for more than 3 % of the original instances. Taking *both* running time and quality into account KaHyPar dominates hMetis-R and hMetis-K.

For certain well defined subgroups of instances significantly larger quality improvements are observed. Figure 5 show that this is the case for plain bipartitioning and for matrices derived from web crawls and social networks⁴. On the other hand, for SAT instances, KaHyPar is slightly worse than hMetis-R. However, it is almost twice as fast in this case. On sparse matrix instances both hMetis variants and both PaToH variants are inferior to KaHyPar regarding partitioning quality, while the running time of KaHyPar is comparable to hMetis. On VLSI hypergraphs, the performance in terms of running time and solution quality is comparable to hMetis-R. Tables summarizing the average and best cuts found as well as the running times can be found in Appendix D. Note that these tables also support the interpretation of the results presented in this section.

We conducted a Wilcoxon matched pairs signed rank test [53] (using a 1% significance level) to deter-

⁴Based on the following matrices: `webbase-1M`, `ca-CondMat`, `soc-sign-epinions`, `wb-edu`, `PGPgiantcompo`, `NotreDame-www`, `NotreDame_actors`, `IMDB`, `p2p-Gnutella25`, `Stanford`, `cnr-2000`

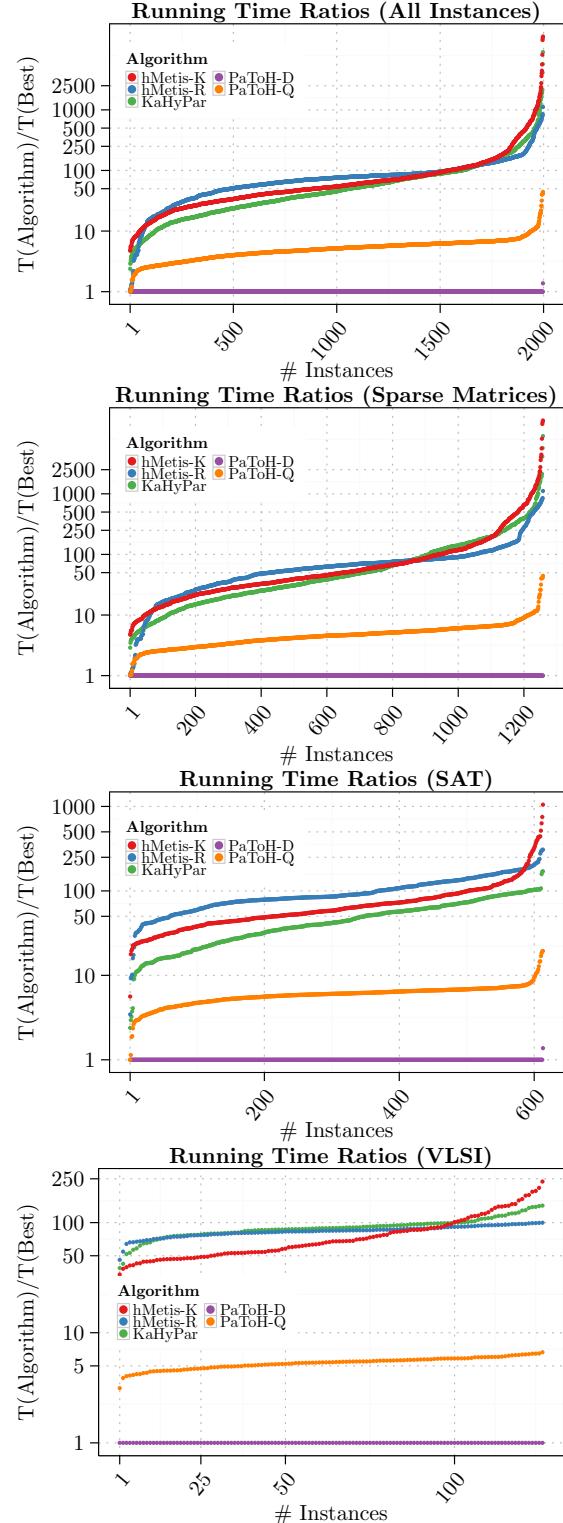


Figure 7: Running time ratio plots for $\varepsilon = 0.03$. The log-scaled y-axis shows the ratio between the running time of the corresponding algorithm and the running time of the fastest algorithm. For each algorithm, the ratios are sorted in increasing order.

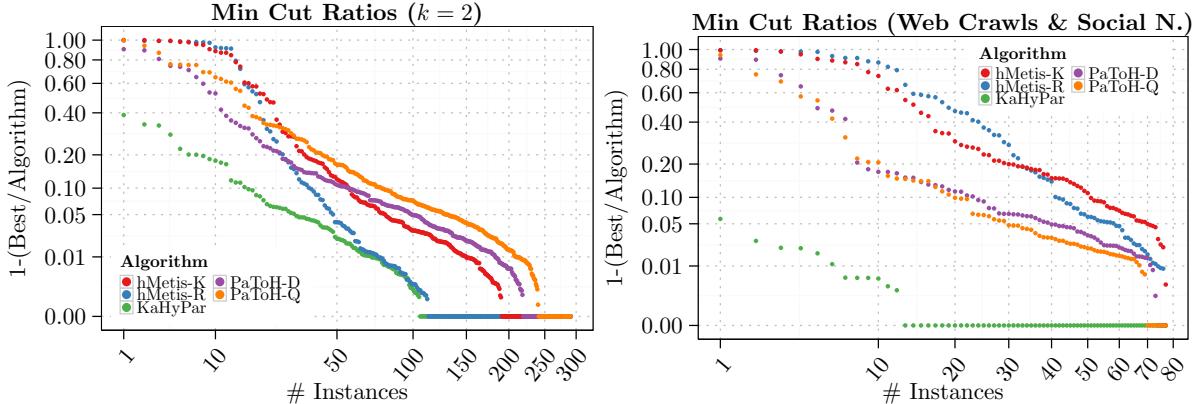


Figure 5: Performance plots for subgroups of instances with large quality improvements ($\varepsilon = 0.03$): bipartitioning (left), web crawls and social networks (right). Note the cube root scale for both axes.

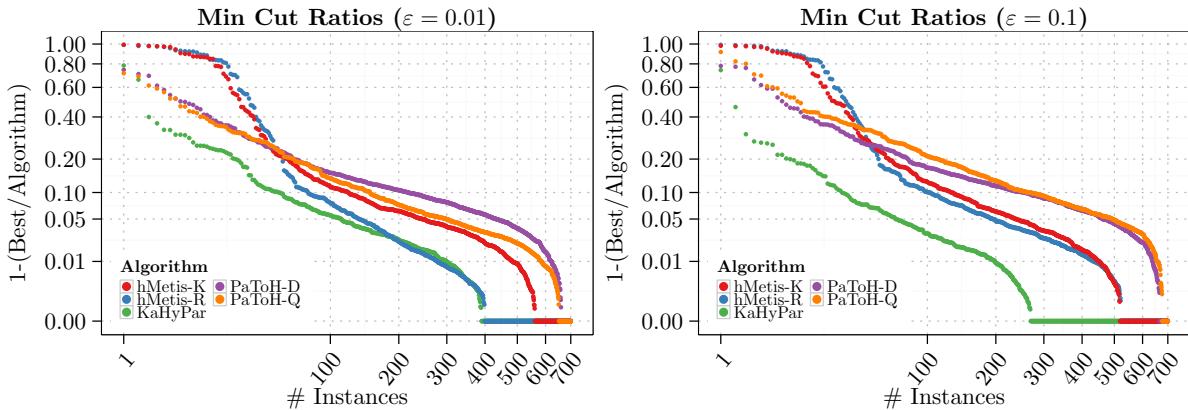


Figure 6: Performance plots for $\varepsilon \in \{0.01, 0.1\}$. Note the cube root scale for both axes.

mine whether or not the difference of KaHyPar and the other algorithms is statistically significant. At a 1% significance level, a z-score with $|Z| > 2.58$ is considered significant. We report the z-scores and the corresponding p-values. Unless stated otherwise, the p-value reported was $p < 2.2 \cdot 10^{-16}$. For sparse matrices, the best cuts of KaHyPar are significantly better than those of hMetis-R ($Z = -13.861$), hMetis-K ($Z = -17.953$), PaToH-Q ($Z = -26.894$) and PatoH-D ($Z = -26.273$). The difference of KaHyPar and hMetis-R was found to be not significant on SAT ($Z = 1.9943, p = 0.04612$) and VLSI instances ($Z = -0.4821, p = 0.6297$). Comparing the best solutions of KaHyPar and hMetis-K, the difference was significant for SAT instances ($Z = -3.0483, p = 0.002302$) but not for VLSI instances ($Z = 0.18261, p = 0.8551$). However, the solutions of KaHyPar were significantly better than PatoH-Q ($Z = -15.636$) and PaToH-D ($Z = -16.34$) on SAT instances and VLSI instances (PatoH-Q: $Z = -9.5529$ and PaToH-D: $Z = -9.6125$).

5.2 Effects of Imbalance Parameter. For $\varepsilon = 0.01$, 3087 out of 7000 partitions produced by hMetis-K were imbalanced (up to 14% imbalance). hMetis-R produced 163 imbalanced partitions (up to 1.3% imbalance), KaHyPar produced 58 imbalanced partitions (up to 2.5% imbalance) and 205 partitions of PaToH-Q had up to 2% imbalance. Only PaToH-D fulfilled the balance constraint in all experiments. For $\varepsilon = 0.1$ hMetis-K produced 24 imbalanced partitions with up to 14% imbalance. All other partitioners produced balanced partitions. The overall performance is summarized in Figure 6. Performance plots for each instance class can be found in Appendix E. For an allowed imbalance of 1% (left) the minimum cut ratios of KaHyPar are slightly better than hMetis-R. Due to the localized view of our local search algorithm it becomes difficult to find feasible moves around the uncontracted vertex pair that improve the solution quality. However, our algorithm still outperforms hMetis-K, PaToH-D and PaToH-Q. If we allow up to 10% imbalance (right), the performance of KaHyPar improves significantly. It now produces the

best partitions for 431 out of 700 instances.

6 Conclusions and Future Work

We presented a novel k -way hypergraph partitioning algorithm that instantiates the multilevel paradigm in its most extreme version, by removing only a single vertex between two levels. Our algorithm computes high quality partitions for a large set of hypergraphs derived from various application domains. Four key aspects yield a tool/algorith that dominates the popular hMetis system in *both* solution quality and running time: (i) active exploitation of the fact that we represent the hypergraph as a bipartite graph to derive an efficient hypergraph data structure, (ii) an engineered coarsening algorithm, (iii) a portfolio of initial partitioning algorithms and (iv) a highly tuned local search algorithm.

Several ideas exist to narrow the gap between the running time of KaHyPar and PaToH: Ignoring nets that are larger than a certain threshold size during coarsening, as suggested by [4]. The running time of local search could be improved by developing an adaptive stopping rule as in [29]. With respect to quality we could introduce V-cycles [14, 30, 54] and an evolutionary algorithm along the lines of KaHIP [55]. Generalizing the gain cache to direct k -way partitioning as in [33] might give good quality for large k without incurring excessive performance penalties. Having shown that our algorithm computes high quality partitions when optimizing the total cut size, future work could also look at different partitioning objectives that rely on a global view of the problem, like the $(\lambda - 1)$ or sum-of-external-degrees metric [19].

References

- [1] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proceedings VLDB Endow.*, 3(1-2):48–57, September 2010.
- [2] B. Heintz and A. Chandra. Beyond graphs: Toward scalable hypergraph analysis systems. *SIGMETRICS Perform. Eval. Rev.*, 41(4):94–97, April 2014.
- [3] D. A. Papa and I. L. Markov. Hypergraph Partitioning and Clustering. In T. F. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007.
- [4] Ü. V. Catalyürek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.
- [5] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.
- [6] S. Klamt, U. Haus, and F. Theis. Hypergraphs and Cellular Networks. *PLoS Comput Biol*, 5(5):e1000385, 05 2009.
- [7] F. A. Aloul, I. L. Markov, and K. A. Sakallah. MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation. *J. UCS*, 10(12):1562–1596, 2004.
- [8] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.
- [9] Thang Nguyen Bui and Curt Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992.
- [10] T.N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In *SIAM Conference on Parallel Processing for Scientific Computing*, pages 445–452, 1993.
- [11] J. Cong and M. Smith. A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design. In *30th Conference on Design Automation*, pages 755–760, June 1993.
- [12] S. Hauck and G. Borriello. An Evaluation of Bipartitioning Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, Aug 1997.
- [13] B. Hendrickson and R. Leland. A Multi-Level Algorithm For Partitioning Graphs. *SC Conference*, 0:28, 1995.
- [14] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration VLSI Systems*, 7(1):69–79, 1999.
- [15] A. Trifunovic. *Parallel Algorithms for Hypergraph Partitioning*. PhD thesis, University of London, 2006.
- [16] C. J. Alpert, J.-H. Huang, and A. B. Kahng. Multilevel Circuit Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, 1998.
- [17] C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: a Survey. *Integration, the VLSI Journal*, 19(1–2):1 – 81, 1995.
- [18] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Proc. Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop*, volume 588 of *Contemporary Mathematics*. AMS, 2013.
- [19] G. Karypis and V. Kumar. Multilevel K -way Hypergraph Partitioning. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 343–348. ACM, 1999.
- [20] B. Vastenhoud and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Rev.*, 47(1):67–95, 2005.
- [21] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Catalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Conference on Parallel and Distributed Processing*, IPDPS, pages 124–124. IEEE, 2006.

- [22] A. Trifunovi and W. J. Knottenbelt. Parallel Multi-level Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563 – 581, 2008.
- [23] Ü. V. Çatalyürek and M. Deveci and K. Kaya and B. Uçar. UMPa: A multi-objective, multi-level partitioner for communication minimization. In Bader et al. [18], pages 53–66.
- [24] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level Direct K-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices. *J. Parallel Distrib. Comput.*, 68(5):609–625, 2008.
- [25] C. Fiduccia and R. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *19th ACM/IEEE Design Automation Conf.*, pages 175–181, 1982.
- [26] M. Birn, M. Holtgrewe, P. Sanders, and J. Singler. Simple and fast nearest neighbor search. In *11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2010.
- [27] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1-6):381–413, 1992.
- [28] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *7th Workshop on Experimental Algorithms (WEA)*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.
- [29] V. Osipov and P. Sanders. *n*-Level Graph Partitioning. In *18th European Symposium on Algorithms (ESA)*, volume 6346 of *LNCS*, pages 278–289, 2010.
- [30] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *19th European Symposium on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.
- [31] P. Sanders and C. Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms*, pages 164–175. Springer Berlin Heidelberg, 2013.
- [32] F. Ziegler. *n*-Level Hypergraph Partitioning. Bachelor’s thesis, KIT, 2012.
- [33] V. Henne, H. Meyerhenke, P. Sanders, S. Schlag, and C. Schulz. *n*-Level Hypergraph Partitioning. Technical Report arXiv:1505.00693, KIT, May 2015.
- [34] C. Berge. Isomorphism problems for hypergraphs. In Jr. Hall, M. and J.H. van Lint, editors, *Combinatorics*, volume 16 of *NATO Advanced Study Institutes Series*, pages 205–214. Springer Netherlands, 1975.
- [35] G. Karypis. Multilevel Hypergraph Partitioning. Technical Report TR 02-025, Department of Computer Science and Engineering, University of Minnesota, June 2002.
- [36] Ü. V. Catalyürek and C. Aykanat. PaToH: Partitioning Tool for Hypergraphs. <http://bmi.osu.edu/umit/PaToH/manual.pdf>, 1999.
- [37] B. Hendrickson and E. Rothberg. Improving the Run Time and Quality of Nested Dissection Ordering. *SIAM J. on Scientific Computing*, 20(2):468–489, 1998.
- [38] T. Heuer. Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning. Bachelor’s thesis, KIT, 2015.
- [39] H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-Constrained Clustering. In *Experimental Algorithms*, volume 8504 of *LNCS*, pages 351–363. Springer, 2014.
- [40] B. Krishnamurthy. An Improved Min-Cut Algorithm for Partitioning VLSI Networks. *IEEE Transactions on Computers*, 33(5):438–446, 1984.
- [41] C. J. Alpert. The ISPD98 Circuit Benchmark Suite. In *Proceedings of the 1998 International Symposium on Physical Design*, pages 80–85, New York, 1998. ACM.
- [42] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.
- [43] A. Belov, D. Diepold, M. Heule, and M. Järvisalo. The SAT Competition 2014. <http://www.satcompetition.org/2014/>, 2014.
- [44] E. Boman, K. Devine, V. Leung, S. Rajamanickam, L. A. Riesen, and Ü. V. Catalyürek. Zoltan User’s Guide. http://www.cs.sandia.gov/Zoltan/ug_html/ug_alg_pato.html, 2012.
- [45] R. H. Bisseling, B. O. Fagginger Auer, A. N. Yzelman, T. van Leeuwen, and Ü. V. Catalyürek. Two-dimensional Approaches to Sparse Matrix Partitioning. In *Combinatorial Scientific Computing*, pages 321 – 349. CRC Press, New York, 2012.
- [46] S. Riyavong. Experiments on Sparse Matrix Partitioning. Technical Report CERFACS Working Note WN/PA/03/32, CERFACS, 2003.
- [47] Catalyürek, Ü. V. ISPD98 Benchmark. <http://bmi.osu.edu/umit/PaToH/ispd98.html>.
- [48] A.E. Caldwell, A.B. Kahng, and I.L. Markov. Improved Algorithms for Hypergraph Bipartitioning. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000.*, pages 661–666, June 2000.
- [49] J. Cong, M. Romesis, and M. Xie. Optimality, scalability and stability study of partitioning and placement algorithms. In *International Symposium on Physical Design*, ISPD, pages 88–94. ACM, 2003.
- [50] M. Deveci, K. Kaya, B. Uar, and Ü. V. Catalyürek. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *Journal of Parallel and Distributed Computing*, 77(0):69 – 83, 2015.
- [51] Sebastian Schlag. Benchmark Hypergraphs and Detailed Experimental Results. <http://dx.doi.org/10.5281/zenodo.30176>, September 2015.
- [52] Nicholas J. Cox. Stata tip 96: Cube roots. *Stata Journal*, 11(1):149–154(6), 2011.
- [53] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.
- [54] C. Walshaw. Multilevel Refinement for Combinatorial Optimisation Problems. *Annals of Operations Research*, 131(1-4):325–372, 2004.
- [55] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *ALENEX 2012*, pages 16–29. SIAM, 2012.

A Parameter Tuning Instances

Table 1: Properties of our benchmark set used for parameter tuning. The table is split into two groups: VLSI instances and sparse matrix instances. Within each group, the hypergraphs are sorted by $|V|$.

Hypergraph	$ V $	$ E $	$ pins $	$d(v)$			$ e $		
				min	avg	max	min	avg	max
ibm01	12 752	14 111	50 566	1	3.97	39	2	3.58	42
ibm02	19 601	19 584	81 199	1	4.14	69	2	4.15	134
ibm03	23 136	27 401	93 573	1	4.04	100	2	3.41	55
ibm04	27 507	31 970	105 859	1	3.85	526	2	3.31	46
ibm05	29 347	28 446	126 308	1	4.30	9	2	4.44	17
ibm06	32 498	34 826	128 182	1	3.94	91	2	3.68	35
ibm07	45 926	48 117	175 639	1	3.82	98	2	3.65	25
vibrobox	12 328	12 328	342 828	9	27.81	121	9	27.81	121
bcsstk29	13 992	13 992	619 488	5	44.27	71	5	44.27	71
memplus	17 758	17 758	126 150	2	7.10	574	2	7.10	574
bcsstk30	28 924	28 924	2 043 492	4	70.65	219	4	70.65	219
bcsstk31	35 588	35 588	1 181 416	2	33.20	189	2	33.20	189
bcsstk32	44 609	44 609	2 014 701	2	45.16	216	2	45.16	216

B Parameter Tuning Results

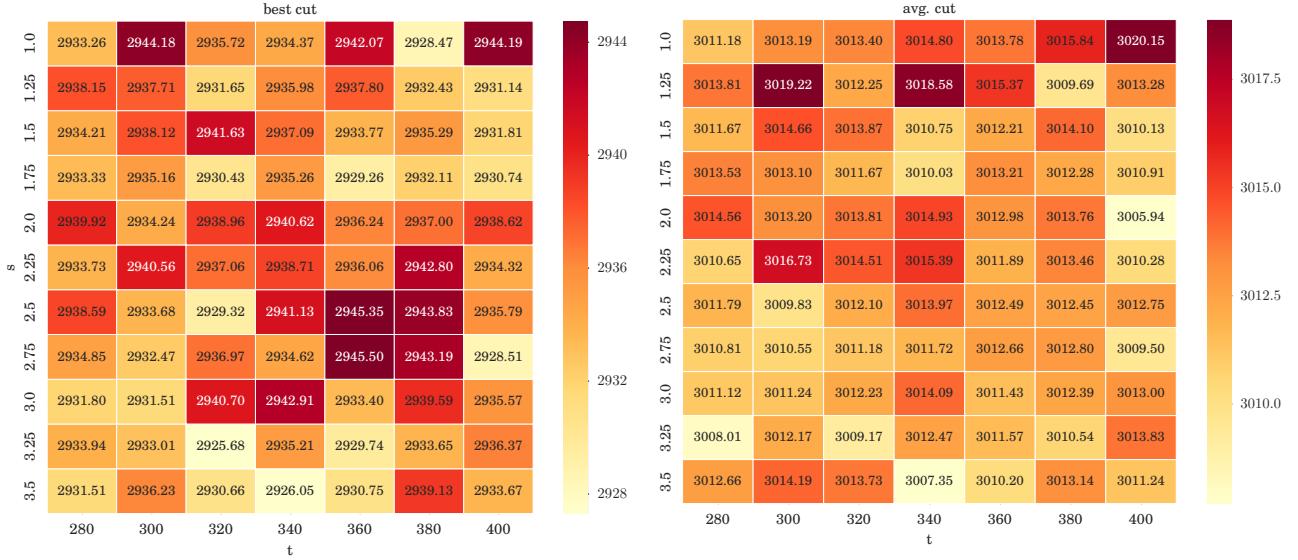


Figure 8: Results of our parameter tuning experiments regarding the minimum size of the coarsest hypergraph t and the scaling factor of maximum allowed vertex weight s . t -values less than 280 have been omitted because all of them resulted in worse solution quality. We set s to 3.25 and t to 320.

Table 2: Results of our parameter tuning experiments regarding the allowed maximum number of moves c without improvement. Based on these experiments we set c to 350.

c	avg. cut	best cut	t [s]
10	3 044.8	2 969.7	0.7
25	3 031.6	2 959.7	1.3
50	3 026.1	2 959.2	2.2
100	3 021.3	2 957.0	3.9
150	3 020.4	2 954.7	5.6
200	3 018.4	2 946.1	7.3
250	3 017.4	2 945.3	9.0
300	3 016.8	2 942.2	10.9
350	3 015.9	2 942.1	12.7
400	3 015.8	2 942.1	14.9
450	3 015.2	2 943.9	16.3
500	3 015.3	2 942.4	17.8

C Excluded Test Instances

Table 3: Out of 2170 test instances, we excluded the following 173 instances either because PaToH-Q could not allocate enough memory or one of the other partitioners could not partition the instances in the given time limit. The table is split into two groups: sparse matrix instances and SAT instances.

Hypergraph	k						
	2	4	8	16	32	64	128
12month1	△	△	△	△	△○	△○	△○
192bit	□						
ASIC_680k	△	△	△	△	△	△	△
ESOC	□	□			□	□○	□○
GL7d22	□△●○	□△●○	□△●○	□△●○	□△●○	□△●○	□△●○
LargeRegFile	△	△	△	△	△	△	△
Rucci1					□		
Trec14						○	○
appu					○	○	
circuit5M	□△	□△	□△●	□△●	□△●○	□△●○	□△●○
gupta3						○	
hollywood-2009	△	△●○	△●○	△●○	△●○	△●○	△●○
human_gene2					○	○	
kron_g500-logn16					○	○	
n4c6-b11	□	□	□	□	□	□	□
nd12k						○	
nlpkkt120				●	●	●	●
rel9	□	□△●○	□△●○	□△●○	□△●○	□△●○	□△●○
sls	□	□	□○	□○	□○	□○	□○
004-80-8	□	□	□	□	□	□	□
005-80-12	□	□	□	□	□	□	□
007-80-8	□	□	□	□	□	□	□
008-80-12	□	□	□	□	□	□	□
008-80-8	□	□	□	□	□	□	□
010-80-12	□	□	□	□	□	□	□
11pipe_k		○	○	○	○	○	○
11pipe_q0_k						○	
9vliw_m_9stages_iq3_C1_bug10	○	●○	●○	●○	△●○	△●○	△●○
9vliw_m_9stages_iq3_C1_bug7	●○	●○	△●○	△●○	△●○	△●○	△●○
9vliw_m_9stages_iq3_C1_bug8	●○	●○	●○	△●○	△●○	△●○	△●○
bjrb07amba_10andenv	□	□	□	□	□	□	□
blocks-blocks-37-1.		□	□	□	□	□	□
130-NOTKNOWN		□	□	□	□	□	□
q_query_3.L150_coli.sat							○
q_query_3.L200_coli.sat			○	○	○	△○	△○
velev-vliw-uns-2.0-uq5							○

- △ : KaHyPar exceeded time limit
- : hMetis-R exceeded time limit
- : hMetis-K exceeded time limit
- : PaToH-Q memory allocation error

D Geometric Mean Comparison

Table 4: Comparison with other systems for all 1997 test instances and $\varepsilon = 0.03$. Cuts of hMetis and PaToH are shown as increase in *percent* relative to KaHyPar.

$\varepsilon = 0.03$ Algorithm	All Instances			Sparse Matrices			SAT			VLSI		
	avg.	best	t [s]	avg.	best	t [s]	avg.	best	t [s]	avg.	best	t [s]
KaHyPar	6 776	6 581	37	4 801	4 696	30	14 821	14 177	59	4 694	4 575	29
hMetis-R	+7.91	+8.80	51	+14.52	+14.53	35	-2.75	-0.38	131	-1.15	+0.09	27
hMetis-K	+10.19	+10.06	47	+14.55	+14.74	36	+3.94	+2.97	92	-0.68	+0.41	23
PaToH-Q	+4.39	-	4	+3.56	-	3	+6.33	-	8	+3.34	-	2
PaToH-D	+10.05	+7.81	1	+8.20	+6.15	1	+14.02	+11.47	1	+9.69	+6.94	1

Table 5: All Benchmark Instances for $k = 2$ and $\varepsilon = 0.03$

Algorithm	avg.	best	t [s]
KaHyPar	1 173	1 105	11.8
hMetis-R	+19.96	+21.64	20.6
hMetis-K	+22.72	+25.13	19.2
PaToH-Q	+7.97	-	1.5
PaToH-D	+11.48	+8.19	0.3

Table 6: Web-Crawls and Social Networks for $\varepsilon = 0.03$

Algorithm	avg.	best	t [s]
KaHyPar	6 680	6 269	190
hMetis-R	+86.63	+92.03	198
hMetis-K	+66.82	+71.64	263
PaToH-Q	+9.03	-	7
PaToH-D	+17.88	+18.72	1

Table 7: Comparison with other systems for 700 test instances and $\varepsilon = 0.01$ (top) and $\varepsilon = 0.1$ (bottom).

$\varepsilon = 0.01$ Algorithm	All Instances			Sparse Matrices			SAT			VLSI		
	avg.	best	t [s]	avg.	best	t [s]	avg.	best	t [s]	avg.	best	t [s]
KaHyPar	6 272	6 065	37	4 386	4 280	29	12 367	11 725	54	6 992	6 793	51
hMetis-R	+8.77	+9.74	60	+19.44	+19.46	41	-6.43	-4.06	136	-2.56	-1.28	52
hMetis-K	+10.63	+10.92	44	+19.81	+19.26	33	-2.26	-0.73	82	-0.62	+0.11	38
PaToH-Q	+1.77	-	4	+1.73	-	3	+1.69	-	8	+2.30	-	3
PaToH-D	+10.44	+7.35	1	+9.07	+6.41	1	+13.06	+9.06	1	+10.94	+7.88	1

$\varepsilon = 0.1$ Algorithm	All Instances			Sparse Matrices			SAT			VLSI		
	avg.	best	t [s]	avg.	best	t [s]	avg.	best	t [s]	avg.	best	t [s]
KaHyPar	5 847	5 676	36	4 154	4 061	28	11 132	10 626	52	6 597	6 455	49
hMetis-R	+11.74	+12.60	58	+21.79	+21.56	40	-2.50	+0.20	132	+0.34	+0.96	50
hMetis-K	+11.33	+11.74	44	+20.12	+20.21	34	-0.62	+0.37	81	-0.78	-0.54	38
PaToH-Q	+9.16	-	4	+7.41	-	3	+12.98	-	8	+8.43	-	3
PaToH-D	+13.02	+10.15	1	+11.20	+8.58	1	+16.29	+13.27	1	+14.37	+10.34	1

E Comparison for Different Imbalance Values

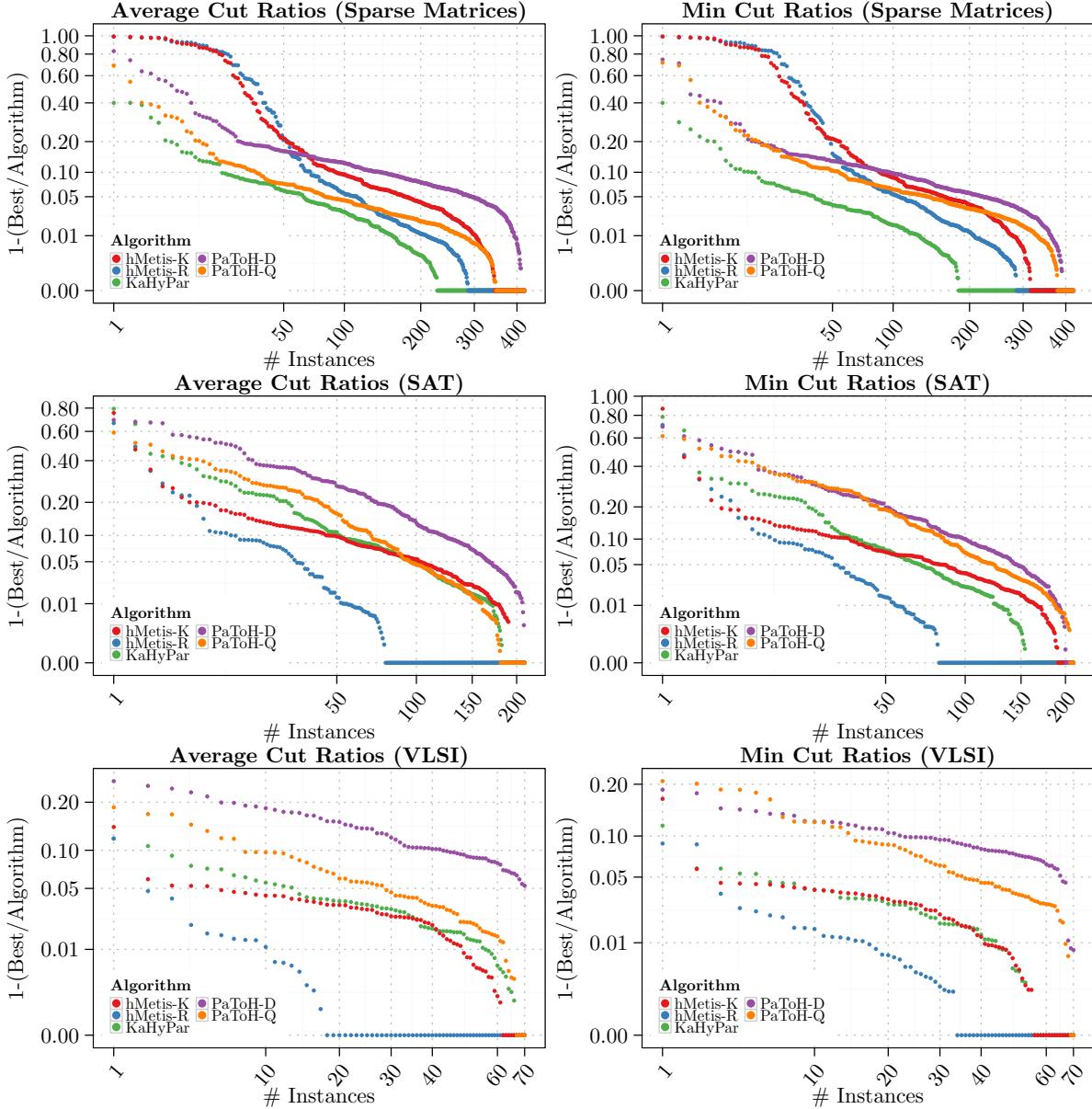


Figure 9: Performance plots per benchmark set for $\varepsilon = 0.01$. Note the cube root scale for both axes.

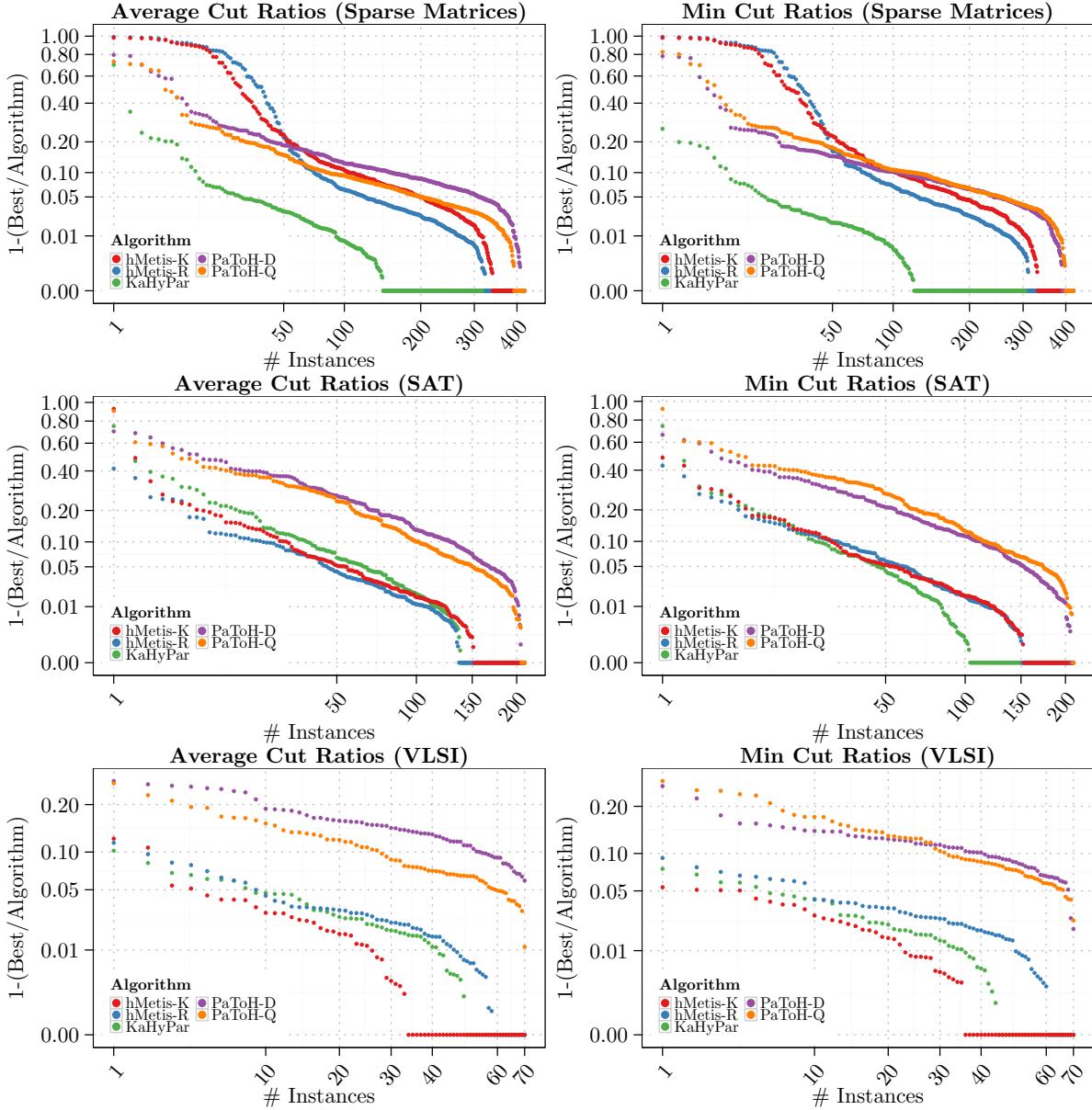


Figure 10: Performance plots per benchmark set for $\epsilon = 0.1$. Note the cube root scale for both axes.