Engineering a direct k-way Hypergraph Partitioning Algorithm

Yaroslav Akhremtsev* Tobias Heuer Peter Sanders* Sebastian Schlag*

Abstract

We develop a fast and high quality multilevel algorithm that directly partitions hypergraphs into k balanced blocks – without the detour over recursive bipartitioning. In particular, our algorithm efficiently implements the powerful FM local search heuristics for the complicated k-way case. This is important for objective functions which depend on the number of blocks connected by a hyperedge. We also remove several further bottlenecks in processing large hyperedges, develop a faster contraction algorithm, and a new adaptive stopping rule for local search. To further reduce the size of hyperedges, we develop a *pin-sparsifier* based on the min-hashing technique that clusters vertices with similar neighborhood. Extensive experiments indicate that our KaHyPar-partitioner compares favorably with the best previous systems. KaHyPar is faster than hMetis *and* computes better solutions. KaHyPar's results are considerably better than the (faster) PaToH partitioner.

1 Introduction

Hypergraphs are a generalization of graphs, where each (hyper)edge (or *net*) 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 an objective function. However, allowing nets of arbitrary size makes the partitioning problem more difficult in practice [1, 2]. The two most prominent objective functions are the *cut-net* and the *connectivity* (or $\lambda - 1$) metrics. Cut-net is a straightforward generalization of the edge-cut objective in graph partitioning (i.e., minimizing the sum of the weights of those nets that connect more than one block). The connectivity metric additionally takes into account the actual *number* λ of blocks connected by a net. By summing the $\lambda - 1$-values of all nets, one accurately models the total communication volume of parallel sparse matrix-vector multiplication [3] and once more gets a metric that reverts to edge-cut for plain graphs.

*Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Algorithmics II. Email: {yaroslav.akhremtsev, sanders, sebastian.schlag}@kit.edu, tobias.heuer@gmx.net

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) [4]. While the former is an example of a field where small optimizations can lead to significant savings [5], the latter exemplifies problems where hypergraph-based modeling is more flexible than graph-based approaches [2, 3, 6–8].

Since hypergraph partitioning is NP-hard [9] and since it is even NP-hard to find good approximate solutions for graphs [10], heuristic *multilevel* algorithms [11–14] are used in practice. These algorithms either compute a k-way partition directly or via recursive bisection. In *direct* k-way partitioning, the hypergraph is first coarsened to obtain a hierarchy of smaller hypergraphs that reflect the basic structure of the input. Afterwards, an initial partitioning algorithm computes a k-way partition, which is then improved during uncoarsening using k-way local search algorithms. If k is a power of two, *recursive bisection* (RB) algorithms obtain the final k-way partition by first computing a bisection of the initial hypergraph and then recursing on each of the two blocks.

Using a direct k-way partitioning approach instead of RB is known to have a number of advantages in certain situations [15, 16]: First, it allows local search algorithms a global view of all k blocks. This can be especially relevant for more complex metrics such as $(\lambda-1)$ [16], which RB-based partitioners can only optimize implicitly by splitting cut nets after each bisection [3]. Second, RB cannot always compute good partitions if hypergraphs have many large nets [17]. In this case it is difficult for local search algorithms to find meaningful moves that improve solution quality, because large nets are likely to have many vertices in both blocks. Third, direct k-way algorithms can enforce tighter balancing constraints since recursive bisection algorithms need to adaptively adjust their imbalance ratios at each bisection step in order to guarantee the desired final imbalance. Fourth, having to bisect hypergraphs into two roughly equal-sized blocks is known to restrict the feasible solution space [18].

Outline and Contribution. In [19] our group developed a hypergraph partitioner based on recursive bisection that achieves high quality partitions for the cut-

net metric using a very fine grained "n-level" approach, removing only *one* vertex in every layer of the hierarchy. In this paper, we adopt the n-level approach but otherwise use significantly more sophisticated techniques. In particular we now also address the more complex $(\lambda-1)$ metric, support direct k-way partitioning, introduce a fast sparsifier for preprocessing, and greatly accelerate both contraction and local search.

After giving a brief overview of related work in Section 2 and introducing notation in Section 3, we explain a hypergraph sparsification technique based on Min-Hash fingerprints in Section 4 which improves the speed of our partitioner for hypergraphs with large median net sizes. In Section 5.1 we then present a simple and fast coarsening algorithm that eliminates the bottlenecks of [19] without affecting the overall solution quality. The initial partitioning algorithm is described briefly in Section 5.2.

In Section 5.3 we develop a fast local search algorithm for direct k-way hypergraph partitioning based on the Fiduccia-Mattheyses (FM) heuristic [20]. FM-type algorithms move vertices to other blocks in the order of improvements in the objective and allow to worsen the objective temporarily. We view our algorithm as a major contribution since current direct k-way multilevel HGP libraries [15, 16, 21, 22] only use weaker greedy local search which cannot escape from local optima. Sanchis [23,24] describes FM k-way local search in a single-level context but this algorithm is slow because it has to maintain $k(k-1)$ priority queues. It also leads to worse solutions as single-level algorithms are more easily trapped in local minima. Previous work [15, 25–27] explicitly mentions these problems as reasons for sticking to simple greedy algorithms. We overcome these problems by generalizing the gain caching and updating strategies from [19] to the (considerably more complex) k-way case. We also develop an adaptive stopping rule that generalizes a previous solution for n-level graph partitioning [28].

2 Related Work

Since the 1990s HGP has evolved into a broad research area. We refer to [4, 29–31] for an extensive overview, and focus instead on issues closely related to the contributions of our paper. The two most widely used general-purpose tools are PaToH [3] (originating from scientific computing) and hMetis [15, 32] (originating from VLSI design). Other software packages with certain distinguishing characteristics exist, in particular Mondriaan [33] (sparse matrix partitioning), MLPart [34] (circuit partitioning), Zoltan [35] and Parkway [21] (parallel), UMPa [22] (directed hypergraph model, multi-objective), and kPaToH (multiple constraints, fixed vertices) [16]. All of these tools are based on the multilevel paradigm and compute a k-way partition either directly [15, 16, 21, 22] or via recursive bisection [3, 32–35]. n-level algorithms have been used in geometric data structures based on randomized incremental construction [36, 37] and as a preprocessing technique for route planning [38]. Furthermore, the n-level paradigm has been successfully employed in both graph and hypergraph partitioning: KaSPar [28] is a direct k-way graph partitioner, KaHyPar [19] is based on recursive bipartitioning and currently seems to be the method of choice for optimizing the cut-metric unless speed is more important than quality. There is an unpublished attempt to design a direct k-way n-level hypergraph partitioner that optimizes the cut-net metric [39]. Despite several interesting ideas and good quality in the majority of experiments, this algorithm has not been able to improve on the state of the art consistently in terms of the time-quality trade-off. The highly tuned n-level direct k-way partitioner presented here optimizes the $(\lambda-1)$-metric, but builds on the ideas and insights gathered during the development of both hypergraph partitioning algorithms [19, 39]. To the best of our knowledge, a multilevel version of Sanchis' k-way FM algorithm [24] has only been implemented in [17] to optimize the total message latency of parallel matrix vector multiplies by partitioning hypergraphs with very few (≤ 128) nets.

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 (or nets) E with vertex weights $c : V \to \mathbb{R}_{>0}$ and net weights $\omega : E \to \mathbb{R}_{>0}$, where each net is a subset of the vertex set V (i.e., $e \subseteq V$). The vertices of a net are called *pins* [3]. We use P to denote the multiset of all pins in H. 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 $v \in e$. $\mathrm{I}(v)$ denotes the set of all incident nets of v. The *degree* of a vertex v is $d(v) := |\mathrm{I}(v)|$. 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. A *k-way partition* of a hypergraph H is a partition of its vertex set into k *blocks* $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. We use $b[v]$ to refer to the block of vertex v. We call a k-way partition Π *ε-balanced* if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some parameter ε. We call a block V_i *overloaded* if $c(V_i) > L_{\max}$ and *underloaded* if $c(V_i) < L_{\max}$. 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 $\Phi(e, V_i) > 0$. For each net e, $\Lambda(e) := \{V_i \mid \Phi(e, V_i) > 0\}$ denotes the *connectivity set* of e. The *connectivity* of a net e is the cardinality of its connectivity set: $\lambda(e) := |\Lambda(e)|$. A block V_i is *adjacent* to a vertex $v \notin V_i$ if $\exists\ e \in \mathrm{I}(v) : V_i \in \Lambda(e)$. $\mathrm{R}(v)$ denotes the set of all blocks adjacent to v. We call a net *internal* if $|\lambda(e)| = 1$ 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 Π of a hypergraph H that minimizes an objective function over the cut nets for some ε. Several objective functions exist in the literature [9, 30]. The most commonly used cost functions are the *cut-net* metric: $\mathrm{cut}(\Pi) := \sum_{e \in E'} \omega(e)$ and the *connectivity* metric $(\lambda - 1)(\Pi) := \sum_{e \in E'} (\lambda(e) - 1)\ \omega(e)$, where E' is the set of all cut nets [40]. In this paper, we use the connectivity-metric, which accurately models the total communication volume of parallel sparse matrix-vector multiplication [3]. Optimizing both objective functions is known to be NP-hard [9].

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 $e \in \mathrm{I}(v) \setminus \mathrm{I}(u)$. Furthermore we remove v from all nets $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$. *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 reset to $c(u) := c(u) - c(v)$.

4 Min-Hash Based Pin Sparsifier

The multilevel framework consists of three phases: coarsening, initial partitioning, and local search. Each of these phases employs algorithms that perform some calculation on vertices and their set of neighbors. This requires an iteration over the set of all pins for all incident nets. For hypergraphs with many large nets, these calculations can therefore have a significant impact on the overall running time. To alleviate this impact, this section presents a pin sparsifier.

The central idea is to identify and contract vertices that share many nets ("close" vertices), while leaving "distant" vertices untouched. The *distance* between two vertices v and w is hereby defined as $D(v, w) = 1 - J(\mathrm{I}(v), \mathrm{I}(w))$, where $J(X, Y) = |X \cap Y|/|X \cup Y|$ is the Jaccard index of sets X, Y. Calculating these distances for each vertex v and all of its neighbors $\Gamma(v)$ would lead to a quadratic algorithm. We therefore use *min-hash*-based fingerprints [41]. A *min-hash* family of hash functions is known to be locality sensitive [59]. First, we give the definition of (R, cR, P_1, P_2)-sensitive families of hash functions.

DEFINITION 4.1. *[59, 60] A family of hash functions $\mathcal{H}$ from S to U is called (R, cR, P_1, P_2)-sensitive if $\forall\ p, q \in S$ the following conditions are true:*

1. $D(p, q) \leq R \Rightarrow \forall\ h \in \mathcal{H} :\ Pr[h(p) = h(q)] \geq P_1$
2. $D(p, q) \geq cR \Rightarrow \forall\ h \in \mathcal{H} :\ Pr[h(p) = h(q)] \leq P_2$

Here $D(p, q)$ is a distance function between p and q.

Now we can proof that the *min-hash* family is (R, cR, P_1, P_2)-sensitive.

THEOREM 4.1. *A min-hash family $\mathcal{H} = \{h_\sigma(X) = \min\{\sigma(e) | e \in X\}\}$ is (R, cR, P_1, P_2)-sensitive, where X is finite set with elements from finite universe S and σ is a random permutation of all elements of S.*

Proof. First we prove that $Pr[h_\sigma(X) = h_\sigma(Y)] = J(X, Y)$.

$$Pr[h_\sigma(X) = h_\sigma(Y)] = \sum_{e \in X \cap Y} Pr[e = h_\sigma(X) \wedge e = h_\sigma(Y)] = \frac{|X \cap Y|}{|X \cup Y|},$$

since

$$Pr[e = h_\sigma(X) \wedge e = h_\sigma(Y)] = \frac{\binom{n}{z} \cdot (z-1)! \cdot (n-z)!}{n!} = \frac{1}{z},$$

where $n = |U|$ and $z = |X \cup Y|$.

Hence, if $D(X, Y) \leq R$ then $Pr[h_\sigma(X) = h_\sigma(Y)] = J(X, Y) \geq 1 - R = P_1$. Analogously, if $D(X, Y) \geq cR$ then $Pr[h_\sigma(X) = h_\sigma(Y)] \leq 1 - cR = P_2$.

Deveci et al. [42] adapt this idea to identify similar hyperedges (i.e., to build a net sparsifier), while Haveliwala et al. [43] use it to cluster URLs. Both papers do not give a generic algorithm without having to choose several parameters manually. We present an adaptive approach to determine these parameters dynamically. We also extend Haveliwala et al.'s approach such that the resulting clusters are more balanced.

Maintaining all permutations of elements of S is not memory efficient, therefore we use a set Σ of hash functions of the form $\sigma(x) = ax + b\ mod\ \mathcal{P}$ [44]. Then we can define a *min-hash* family $\mathcal{H} = \{h_\sigma(\mathrm{I}(v)) = \min\{\sigma(e) | e \in \mathrm{I}(v)\} | \sigma \in \Sigma\}$.

We define a set of fingerprints $\{g_i(x, z) = (h_{i,1}(x), h_{i,2}(x), \cdots, h_{i,z}(x))\}_{i=1\ldots l}$ where each hash function $h_{i,j}$ is chosen uniformly at random from a min-hash family $\mathcal{H}$. Vertices with the same fingerprint will

be put in the same cluster. Two fingerprints are equal, if and only if all z hash values are equal. Since these fingerprints approximate the distance between two vertices, the size of the fingerprint (i.e., the number z of hash values) affects the probability that two vertices are put into the same cluster. By increasing the number of hashes, we decrease the probability that two "distant" vertices have the same fingerprint. However, at the same time, this also decreases the probability of "close" vertices to be in the same cluster. To avoid this problem, we calculate $l > 1$ fingerprints for each vertex. Theorem 4.2 gives the bound on the probability of two vertices having equal fingerprints, which implies that this probability decreases exponentially as the size of a fingerprint increases.

THEOREM 4.2. *Consider a fingerprint $g(x,z) = (h_1(x), h_2(x), \cdots, h_z(x))$, where all z hash functions are chosen uniformly at random from a min-hash family $\mathcal{H}$. Then the following probabilities hold:*

1. $D(x,y) \leq R \Rightarrow Pr[g(x) = g(y)] \geq P_1^z$
2. $D(x,y) \geq cR \Rightarrow Pr[g(x) = g(y)] \leq P_2^z$

Proof. We prove the first case. The second case is proven analogously. Since all z hash functions $h_1(x), \cdots, h_z(x)$ are independent it follows that

$$Pr[g(x) = g(y)] = Pr[\bigwedge_{j=1\ldots z} h_j(x) = h_j(y)] = \prod_{j=1\ldots z} Pr[h_j(x) = h_j(y)] \geq P_1^z.$$

The quality and speed of our algorithm depends on the choice of parameters z and l. Datar et al. [45] choose z and l using a fixed global distance. Since the distance between vertices varies in different parts of a hypergraph, we adaptively choose both z and l for each vertex individually.

The adaptive clustering works as follows: In the i-th iteration we build a hash table T_i by inserting each vertex v using a fingerprint $g_i(\mathrm{I}(v), z(v))$ as a key, where $z(v)$ is the size of the fingerprint of v. Next we use T_i to extend clusters that were previously built using $\{T_1, \ldots, T_{i-1}\}$. Let $T_i[v]$ denote a bucket such that $\forall u \in T_i[v] : g_i(\mathrm{I}(v), z(v)) = g_i(\mathrm{I}(u), z(u))$. By representing $T_i[v]$ as a hash table, we can guarantee insertions an deletions in $\mathcal{O}(1)$ expected time. We now present our algorithm in more detail.

Adaptive construction of hash table T. Bawa et al. [46] use a similar idea to answer nearest-neighbors queries. We build T incrementally using a fingerprint $g(x,z)$. During each iteration for each active vertex v we increment $z(v)$ and reinsert it with a key $g(\mathrm{I}(v), z(v))$; if $|T[v]| \leq c_{\max}$ and $z(v) \geq h_{\min}$ then v becomes inactive. We repeat this process until all vertices are inactive. A pseudocode description is given in Algorithm 1.

Algorithm 1: Adaptive Hash Table Construction

```
1  Function AdaptiveHashTableConstruction(G)
       Input: Hypergraph G = (V, E)
2      Active ← {V}        // The set of active vertices
3      z ← 1
4      HashTable T
5      while active is not empty ∧ z ≤ h_max do
6          HashTable newT
7          foreach v ∈ active do
8              newT.insert(g(I(v), z), v)
9          foreach v ∈ active do
10             if |T[v]| = |newT[v]| ∧ z ≥ h_min then
               // If T[v] and newT[v] are equal.
11                 for u ∈ newT[v] do
12                     Active ← Active \{u}
13             if v ∈ active ∧ |T[v]| ≤ c_max ∧ z ≥ h_min
               then
14                 for u ∈ T[v] do
15                     Active ← Active \{u}
16         z ← z + 1
17         swap(newT, T)
       Output: Hash table T
```

Our algorithm builds T in $\mathcal{O}(h_{\max} \cdot \sum_{v \in V} d(v)) = \mathcal{O}(h_{\max} \cdot |P|)$ time, where $h_{\max}$ is the maximum number of hash functions.

Adaptive clustering algorithm (ACA). Suppose that we have already performed ACA for $i-1$ hash tables and we want to build and process hash table T_i. For each active vertex $v \in V$ (a vertex is active if its cluster size is less than $c_{\min}$), we iterate over the active vertices in $T_i[v]$, assign them to cluster c_v of v while the size of c_v is less than $c_{\max}$ and remove them from $T_i[v]$. If the size of c_v exceeds $c_{\min}$ then all vertices in c_v become inactive. We introduce parameters $c_{\min}$ and $c_{\max}$ to build a more balanced clustering. The algorithm stops if the number of clusters is less than $|V|/2$ or if it exceeds the maximum number of hash tables l. We process a hash table in $\mathcal{O}(h_{\max} \cdot |V|)$ time, since we look up a bucket of a vertex in $\mathcal{O}(h_{\max})$ time and visit each vertex at most twice. In summary, ACA works in $\mathcal{O}(l \cdot h_{\max} \cdot |P|)$ time, because we have at most l hash tables. In our experiments we set c_{min} to 2, c_{max} to 10, h_{min} to 10, h_{max} to 100 and l to 5.

5 n-Level direct k-way Partitioning

We now describe the three phases of our main partitioning algorithm. We begin by engineering the coarsening

algorithm in Section 5.1. The goal of the coarsening phase it to contract highly connected vertices such that the number of nets remaining in the hypergraph and their size is successively reduced [32]. Note that this differs from the goal of the sparsifier presented in the previous section, because the sparsifier is not intended to explicitly reduce the number of nets. Removing nets leads to simpler instances for initial partitioning, while small net sizes allow FM-based local search algorithms to identify moves that improve the solution quality. Afterwards, we briefly discuss initial partitioning in Section 5.2 and introduce our FM-based direct k-way local search algorithm in Section 5.3.

5.1 Fast n-Level Coarsening Multilevel algorithms either compute vertex matchings [3, 16, 33–35] or clusterings [3, 15, 21, 32] on each level of the coarsening hierarchy. Different rating functions are used to determine the vertices to be matched or clustered [3, 15, 30] together. The contracted vertices then form the vertex set of the coarser hypergraph on the next level. In contrast, n-level partitioning algorithms like the graph partitioner KaSPar [28] and the hypergraph partitioner KaHyPar [19] create a hierarchy of (nearly) n levels by removing only a single vertex between two levels. The two main bottlenecks in the coarsening phase of these algorithms are (i) the use of a priority queue (PQ) to determine which vertex pair to contract next and (ii) the recalculation [28] or updating [19] of already calculated ratings after each contraction. This is necessary since the contraction of a vertex pair (u, v) can change the ratings of some neighboring vertices in $\Gamma(u) \cup \Gamma(v)$. In the following, we present a much simpler n-level coarsening algorithm that eliminates both of these bottlenecks while retaining the solution quality of the approach proposed in [19].

Algorithm Outline. Our algorithm works in passes. At the beginning of each pass, we create a random permutation of the current vertex set. For each vertex u, we then determine its contraction partner v using the *heavy-edge* rating function $r(u, v) := \sum_{e \in E'} \omega(e)/(|e| - 1)$, where $E' := \{\mathrm{I}(v) \cap \mathrm{I}(u)\}$. This rating function is also the default in hMetis [32], Parkway [21] and PaToH [47] and prefers vertex pairs that share a large number of heavy nets with small size. Ties are broken in favor of unmatched vertices to avoid imbalanced inputs for the initial partitioning phase. Furthermore, vertices with $c(v) > c_{\max} := \lceil \frac{c(V)}{t} \rceil$ are never contracted further. While traditional coarsening algorithms first compute the matching/clustering in each level and then use it to build a *new* hypergraph data structure for the next level, we perform the contraction of vertex pairs immediately *on the fly* during the clustering process, i.e., after finding the contraction partner v for a vertex u, we immediately contract (u, v). Thus v will be removed from the hypergraph and will not be visited in this or any future passes over the vertex set. A pass ends as soon as every vertex in the random permutation was considered either as representative or as contraction partner. Then, a new pass is started by creating a new random permutation of the remaining vertices. Coarsening is stopped as soon as the number of vertices drops below t or no eligible vertex is left. Since the initial partitioning algorithm has to compute a k-way partition, the number t of vertices remaining in the coarsest hypergraph should be a function of k [15]. Based on the results of [39], we stop the coarsening process when the number of vertices drops below $t = 160k$ or no eligible vertex is left. To speed up the coarsening process in the presence of large nets, we do not evaluate the rating function for nets larger than 1 000 vertices.

Contractions can lead to parallel nets (i.e., nets containing exactly the same vertices) and single-node nets in $\mathrm{I}(u)$. Single-node nets are easily identified and removed, 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 parallel to e. Parallel nets are detected using the same algorithm as in [19]: For each net $e \in \mathrm{I}(u)$ we create a fingerprint f_e, such that parallel nets must have equal fingerprints. These fingerprints are then sorted and a final scan identifies parallel nets: Since nets with unequal fingerprints cannot be parallel by definition, we have to perform pairwise comparisons of the pin sets only for those nets with same fingerprint and size.[1]

Engineering Parallel Net Detection. Removing parallel nets does not affect the partitioning quality. Instead it is intended to speed up all phases of the multilevel framework. Therefore it is necessary to make this operation as efficient as possible. Having a fast coarsening algorithm at hand, parallel net detection can easily become the new bottleneck during coarsening. We therefore improve the algorithm described above in two ways: (I) In order to keep the number of pairwise pin set comparisons as low as possible, the fingerprint function should produce few false positives, i.e. equal fingerprints for nets that are not actually parallel. In [19], the fingerprint of a net e was defined as $f_e^{\oplus} := \left(\bigoplus_{v \in e} v\right) \oplus x$,

[1] We also tried the hashing-based approach proposed in [48]. However, on average, it did not perform better than the sorting approach. We attribute this to the fact that the set $\mathrm{I}(u)$ of nets on which we perform parallel net detection is not too large. Therefore, the sorting-based approach has a higher data locality (i.e., fewer cache misses) than the hashing-based approach, which has to perform many random accesses due to pointer chasing.

for some seed x.[2] In [48] the authors evaluate different fingerprint functions and conclude that $f_e^2 := \sum_{v \in e} i^2$ performed best. In Section 6.1, we compare these two fingerprint functions and show that the fingerprint used in [19] creates a significant number of false-positives. We therefore use the function proposed in [48] in our algorithm. (II) Instead of calculating the fingerprints for each net $e \in \mathrm{I}(u)$ from scratch after each contraction operation, we exploit the fact that our fingerprint function f_e^2 is both associative and commutative: When constructing our hypergraph data structure we compute the initial fingerprints for each net *once*. After each contraction, we then update the fingerprints of nets $e \in I(u)$ as follows, distinguishing three cases: In case net e was only incident to representative u before the contraction, the fingerprint remains valid since these nets remain unchanged. If net e contained both u and v before the contraction, we have to remove v from the fingerprint: $f_e^2 := f_e^2 - v^2$. If net e was only incident to v but not to u before the contraction, we have to add u to the fingerprint $f_e^2 := f_e^2 + u^2$ in addition to removing v, since it is incident to u after the contraction.

5.2 Initial Partitioning The initial k-way partition is computed using our n-level recursive bisection algorithm [19], because it produced the best results in preliminary experiments [49]. Since this algorithm was designed to optimize the *cut-net* metric, it discards cut nets after each bisection. We therefore adapt it such that each cut net n is split into two pinwise-disjoint nets $n_0 := \{v \in e : b[v] = V_0\}$ and $n_1 := \{v \in e : b[v] = V_1\}$ after each bisection $\Pi = \{V_0, V_1\}$. These nets are then added to the respective hypergraphs, allowing the algorithm to optimize their connectivity during further bisections. To improve the running time of the initial partitioning phase, we replace the coarsening algorithm of [19] with the algorithm described in the previous section and ignore nets larger than 1 000 vertices. The remaining configuration is left unchanged.

5.3 Localized adaptive k-way FM Local Search Our local search algorithm follows ideas similar to the k-way FM-algorithm proposed by Sanchis [23]. Sanchis' algorithm works in passes. In each pass, the local search algorithm repeatedly moves the vertex with the maximum improvement in the objective (even if it is negative) and tracks the best solution encountered. If no further move is possible, a rollback operation then reverts all moves until it arrives at the state of the best solution.

We simplify the algorithm and employ several techniques to improve its running time. While Sanchis uses $k(k-1)$ PQs to maintain all possible moves for all vertices, we reduce the number of PQs to k – one queue P_i for each block V_i. Furthermore, we only consider moving a vertex v to *adjacent* blocks $R(v) \setminus \{b[v]\}$ rather than calculating and maintaining gains for moves to *all* blocks. This simultaneously reduces the memory requirements and restricts the search space of the algorithm to moves that are more likely to improve the solution. Thus the k PQs require $\mathcal{O}(k|V_B|)$ space in total, where V_B is the set of border vertices. Like the 2-way FM algorithm of [19], our algorithm is highly localized, starting only with the representative and the just uncontracted vertex. The search then gradually expands around this vertex pair by successively inserting moves for neighboring vertices into the queues. An outline of the algorithm is given in Section 5.3.1. Calculating and maintaining the gain values for each potential vertex move is the main bottleneck of most FM implementations [4]. In Section 5.3.2, we therefore generalize the *gain cache* introduced in [19] from 2-way to k-way partitioning to avoid recomputing the gains of valid vertex moves. Delta-gain-updates are used to keep these values up to date during an FM pass. In order to further reduce the running time of the gain update step we outline a way to exclude nets from gain updates in Section 5.3.3. In Section 5.3.4 we then develop an adaptive stopping rule that generalizes a previous solution for n-level graph partitioning [28].

5.3.1 Algorithm Outline At the start of a local search pass, all k queues are empty and disabled. A disabled PQ will not be considered when searching for the next highest gain move. All vertices are labeled inactive and unmarked. Only unmarked vertices are allowed to become active. To start local search 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 means that we calculate the *gain* $g_i(v)$ for moving v to all adjacent blocks $V_i \in R(v) \setminus \{b[v]\}$ and insert v into the corresponding queues P_i using $g_i(v)$ as key. The gain $g_i(v)$ is defined as $g_i(v) := \sum_{e \in \mathrm{I}(v)} \{\omega(e) : \Phi(e, b[v]) = 1\} - \sum_{e \in \mathrm{I}(v)} \{\omega(e) : \Phi(e, V_i) = 0\}$. In order to speed up local search in the presence of large nets, we do not activate vertices that are only incident to nets with $|e| \geq 1\,000$, since it is unlikely that such a vertex entails a move with positive gain.

After insertion, all PQs corresponding to *underloaded* blocks become enabled. The algorithm then repeatedly queries only the *non-empty, enabled* queues to find the move with the highest gain $g_i(v)$, breaking ties randomly. If this move is feasible, vertex v is then moved

[2] $\oplus$ is the bitwise XOR

Algorithm 2: Delta-Gain-Update

```
1  Function DeltaGainUpdate(v, V_from, V_to)
       Input: Vertex v that was moved from block V_from to V_to
2      foreach e ∈ I(v) do                                          // walk all incident nets
3          foreach u ∈ e \ {v} do                                   // and consider each pin
4              if u is marked then continue                         // skip marked vertices
5              if Φ(e, V_from) = 0 then                             // connectivity of net e decreased
6                  if V_from ∉ R(u) then P_from.remove(u)           // u is not adjacent to V_from anymore
7                  else P_from.update(u, −ω(e))                     // u remains adjacent to V_from
8              if Φ(e, V_to) = 1 then                               // connectivity of net e increased
9                  if V_to ∉ R(u) then                              // u was not adjacent to V_to
10                     g_to ← calculate gain for V_to according to Section 5.3
11                     P_to.insert(u, g_to)
12                     R(u) ← R(u) ∪ {V_to}
13                 else P_to.update(u, ω(e))                        // u was already adjacent to V_to
14             if b[u] = V_from ∧ Φ(e, V_from) = 1 then              // moving u will decrease λ(e)
15                 foreach V_i ∈ R(u) \ b[u] do P_i.update(u, ω(e))
16             else if b[u] = V_to ∧ Φ(e, V_to) = 2 then             // moving u can't decrease λ(e) anymore
17                 foreach V_i ∈ R(u) \ b[u] do P_i.update(u, −ω(e))
       Output: The gains for all moves of all neighbors Γ(v) are updated.
```

to block V_i, labeled inactive, and marked. Otherwise it is skipped, since it would violate the balance constraint. Because each vertex is allowed to move at most once during each pass, we remove all other moves of v from the PQs. After a successful move, we then update all neighbors $\Gamma(v)$ of v as follows: All previously inactive neighbors are activated as described above. Neighbors that have become internal are labeled inactive and the corresponding moves are deleted from the PQs. Finally, we perform *delta-gain-updates* for all moves of the remaining active border vertices in $\Gamma(v)$. If the move changed the gain contribution of a net $e \in \mathrm{I}(v)$, we account for that change by increasing/decreasing the gains of the corresponding moves by $\omega(e)$. Moving a vertex v can furthermore change the connectivity of a net $e \in \mathrm{I}(v)$ which in turn can affect the set of adjacent blocks $R(\cdot)$ for each neighbor in $\Gamma(v)$. The delta-gain-update algorithm takes these changes into account by inserting moves to new adjacent blocks and removing moves to blocks that are not adjacent anymore. A pseudocode description of the delta-gain-update process can be found in Algorithm 2. Once all neighbors are updated, local search continues until either no non-empty, enabled PQ remains or the stopping criterion described in Section 5.3.4 is fulfilled. After local search is stopped, we undo 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.

5.3.2 k-way Gain Caching In order to generalize the 2-way gain cache [19] to a k-way gain cache, a complete redesign of the data structure is necessary. Since in 2-way FM there is only one possible move for each vertex (i.e., moving it to the other block $V_i \neq b[v]$ of the bisection), a simple array is enough to store the gain values. When doing k-way local search, each vertex can potentially be moved to $k-1$ different blocks. We therefore use a modified version of the sparse set data structure proposed by Briggs and Torczon [50]. For each vertex v, this data structure uses $\mathcal{O}(k)$ space to store the set of adjacent blocks $R(v) \setminus \{b[v]\}$ along with corresponding gain values for moving v to these blocks. This allows us to add a new block V_i to $R(v)$ or remove a block V_j from $R(v)$ in $\mathcal{O}(1)$ time, while still being able to iterate over $R(v)$ in time $\Theta(|R(v)|)$. Furthermore, updates to a specific cache entry can be done in $\mathcal{O}(1)$ time. We now briefly outline the details of the gain cache. Let $c_v[i]$ denote the cache entry for vertex v and adjacent block V_i. After initial partitioning, we initialize the gain cache with all possible moves of all vertices of the coarsest hypergraph. Each time a local search is started with an uncontracted vertex pair (u, v), we invalidate and recompute their corresponding cache entries. This is necessary since v did not exist on previous levels of the hierarchy and since the uncontraction potentially affected both $R(u)$ and the corresponding gain values.[3]

[3] We also tried a more sophisticated version that updates the cache entries of u based on the information gathered during

If a vertex becomes activated during a local search pass, the cached gain values are used for activation. After moving a vertex v with gain $g_i(v)$ from block V_{from} to block V_{to}, its cache value is updated as follows: First, we remove the entry of $c_v[\text{to}]$, since $b[v] = V_{\text{to}}$ after the move and we only cache gain values for moves to adjacent blocks $R(v) \setminus \{b[v]\}$. If v remains connected to V_{from}, we set $c_v[\text{from}] := -c_v[\text{to}]$. The remaining blocks in $B(v) := R(v) \setminus \{V_{\text{from}}, V_{\text{to}}\}$ are updated as follows: For each net $e \in \text{I}(v)$ we have to distinguish two cases: If v was the only pin in V_{from} and $\Phi(e, V_{\text{to}}) \neq 1$ after the move, then moving v to a block $V_i \in B(v)$ would have decreased $\lambda(e)$. We thus have to decrement the corresponding cache entries by $\omega(e)$. Similarly, if v was not the only pin in V_{from} and $\Phi(e, V_{\text{to}}) = 1$ after the move, then moving v to a block $V_i \in B(v)$ in future local search passes will decrease $\lambda(e)$. The corresponding cache entries are therefore incremented by $\omega(e)$. Note that we do not have to increment the cache entry of $c_v[\text{from}]$, since it is already up to date. Afterwards, delta-gain updates of the neighbors $\Gamma(v)$ are then also applied to the corresponding cache entries. Thus the gain cache always reflects the current state of the hypergraph. 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 and changes of adjacent blocks applied to the cache. This can be done by additionally maintaining a *rollback delta cache* that stores the negated delta-gain updates for each vertex as well as the corresponding add/remove operation for $R(\cdot)$. During rollback, this delta cache is then used to restore the gain cache to a valid state.

5.3.3 Excluding Nets from Delta-Gain-Updates

To further reduce the running time of the delta-gain algorithm, we exclude nets from the update procedure if their contribution to the gain values of their pins can not change. The key observation is that after moving a vertex v to a block V_{to}, this block remains connected to all nets $e \in \text{I}(v)$ during this local search pass, because v is not allowed to be moved again. In this case we say that block $V_{\text{to}} \in \Lambda(e)$ has become *unremovable* for net e. Using the following lemma, we exclude nets $e \in \text{I}(v)$ after moving a vertex v from V_{from} to V_{to} if both blocks $\{V_{\text{from}}, V_{\text{to}}\} \in \Lambda(e)$ are marked as unremovable:

LEMMA 1. *Assume that local search moved vertex v from V_{from} to V_{to}. If both blocks are marked as unremovable in the connectivity set $\Lambda(e)$ of a net $e \in \text{I}(v)$, net e does not change its gain contribution for any of its pins.*

Proof. As can be seen in Algorithm 2, there are four different cases that induce a gain change for any of the pins of net $e \in \text{I}(v)$. We now show that none of these conditions can be satisfied, if both V_{from} and V_{to} are marked as unremovable for net e: (i) Since V_{from} is unremovable, there exists at least one pin $p \in e$ that was moved to this block before and this pin is not allowed be moved again during the current local search pass. Therefore it holds that $\Phi(e, V_{\text{from}}) \neq 0$ (line 5). (ii) Since V_{to} is unremovable, there exists at least one pin $p \in e$ that was moved to this block before and this pin is not allowed be moved again during the current local search pass. Furthermore, local search just moved vertex v to block V_{to}. Therefore it holds that $\Phi(e, V_{\text{to}}) \geq 2$ (line 8). (iii) The if-condition in line 14 will only be evaluated if there exists a vertex $u \in e$ with $b[u] = V_{\text{from}}$ that is not marked as moved, since delta-gain updates are only performed for unmarked (i.e., yet unmoved) vertices. However in this case $\Phi(e, V_{\text{from}}) \geq 2$, since V_{from} is already marked as unremovable. (iv) Analogously, the last if-condition in line 16 will only be evaluated if there exists a vertex $u \in e$ with $b[u] = V_{\text{to}}$ that is not marked as moved. However, since V_{to} is marked as unremovable, there exists at least one marked vertex $p \in e$ with $b[p] = V_{\text{to}}$. Furthermore local search just moved vertex v to block V_{to}. Thus, in case there exists an unmarked vertex $u \in e$ with $b[u] = V_{\text{to}}$, then $\Phi(e, V_{\text{to}}) \geq 3$.

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

5.3.4 Adaptive Stopping Rule

Sanders and Osipov [28] propose to make the decision when to stop the current local search pass dependent on the past history of the search. For this purpose, they model the gain values in each step as identically distributed, independent random variables whose expectation μ and variance σ^2 is obtained from the previously observed p steps. They show that it is unlikely that local search can still give an improvement if $p > \sigma^2/4\mu^2$, where μ is the average gain since the last improvement and σ^2 is the

uncontraction and then infers the cache entries of v from those of u. However, recomputation turned out to be faster, since updating and inferring the cache values is significantly more complicated.

variance observed throughout the current local search. We integrate a refined version of this adaptive stopping criterion into our algorithm: On each level, local search performs at least $\beta := \log n$ steps after an improvement is found and continues as long as $\mu > 0$. If μ is still 0 after β steps, local search is stopped. This prevents the algorithm from getting stuck with 0-gain moves, which is likely if nets are large. Otherwise (i.e., if $\mu \neq 0$) we evaluate the equation and act accordingly.

6 Experiments

The algorithm is implemented in the n-level hypergraph partitioning framework *KaHyPar* (**Ka**rlsruhe **Hy**pergraph **Par**titioning). The code is written in C++ and compiled using g++-5.2 with flags `-O3 -mtune=native -march=native`. We refer to the algorithm presented in this paper as KaHyPar-K.

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 Ret Hat Enterprise Linux (RHEL) 7.2.

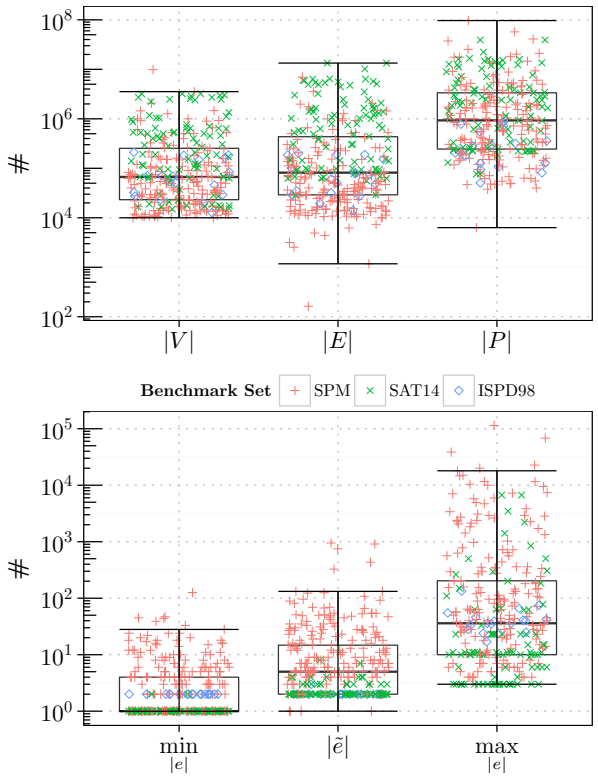

Figure 1: Properties of the benchmark set.

Instances. We evaluate our algorithm on a large collection of hypergraphs[4], which contains instances from three benchmark sets: the ISPD98 VLSI Circuit Benchmark Suite [52], the University of Florida Sparse Matrix Collection [53], and the international SAT Competition 2014 [54]. All hypergraphs have unit vertex and net weights. Properties of the benchmark set are shown in Figure 1. Sparse Matrices are translated into hypergraphs using the row-net model [3], i.e. each row is treated as a net and each column as a vertex. For SAT instances, each boolean variable (and its complement) is mapped to one vertex and each clause constitutes a net [4]. To evaluate the pin sparsifier and to compare our algorithm to other systems, we use the 294 hypergraphs that were also used in [19]. To show the effects of our engineering efforts, we use a representative subset of 100 hypergraphs. In each case, the hypergraphs are partitioned into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks with $\varepsilon = 0.03$. To further evaluate the partitioning performance for non-powers of two, we perform additional experiments on the benchmark subset with $k \in \{5, 23, 47, 107\}$ and $\varepsilon = 0.03$. For each value of k, a k-way partition is considered to be *one* test instance, resulting in a total of 700 (resp. 400) instances for experiments on the subset and 2058 instances for the full benchmark set.

State-of-the-Art Competitors. In Section 6.2 we compare KaHyPar-K to our recursive bisection partitioner KaHyPar-R, to the k-way (hMetis-K) and the RB variant (hMetis-R) of hMetis 2.0 (p1) [15,32], and to PaToH 3.2 [3]. The comparison to KaHyPar-R is done since it was adapted to optimize $(\lambda - 1)$ in Section 5.2. hMetis and PaToH were chosen because they provide the best solution quality (also for the $\lambda - 1$ metric) [19]. We also evaluated kPaToH [16] in preliminary experiments. However, it did not perform better than PaToH (see Appendix A). hMetis does not directly optimize the $(\lambda - 1)$ metric. Instead it optimizes the *sum-of-external-degrees* (SOED), which is closely related to the connectivity metric: $(\lambda-1)(\Pi) = \text{SOED}(\Pi) - \text{cut}(\Pi)$ for unweighted hypergraphs (i.e., each cut net contributes λ times its weight to the objective). We therefore set both hMetis versions to optimize SOED and calculate the $(\lambda - 1)$-metric accordingly. This approach is also used by the authors of hMetis-K [15]. While hMetis-K uses the same imbalance definition as KaHyPar and PaToH, hMetis-R defines the maximum allowed imbalance differently [32]. 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 trans-

[4]The complete benchmark set along with detailed statistics for each hypergraph is publicly available [51].

late our imbalance parameter ε to ε' as described in Eq. (6.1) such that it matches our balance constraint after $\log_2(k)$ bisections:

$$\varepsilon' := 100 \cdot \left(\left((1+\varepsilon) \frac{\lceil \frac{c(V)}{k} \rceil}{c(V)} \right)^{\frac{1}{\lceil \log_2(k) \rceil}} - 0.5 \right) \quad (6.1)$$

PaToH is configured to use a final imbalance ratio of ε to match our balance constraint. Since PaToH 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 *eight* hours per test instance.

Methodology. 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 compare different algorithms in terms of solution quality, we perform a more detailed analysis using the performance plots introduced in [19]: For each algorithm, these plots relate the smallest minimum 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. The plots use a cube root scale for both axes to reduce right skewness [55] and show $1 - (\text{best}/\text{algorithm})$ on the y-axis to highlight the instances were each partitioner performs badly. 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. A value of zero therefore indicates that the corresponding algorithm produced the best solution. Points above one correspond to infeasible solutions that violated the balance constraint. Thus an algorithm is considered to outperform another algorithm if its corresponding ratio values are below those of the other algorithm. In order to include instances with a cut of zero into the results, we set the corresponding cut values to *one* for ratio computations. Furthermore, we conduct Wilcoxon matched pairs signed rank tests [56] (using a 1% significance level) to determine whether or not the difference of KaHyPar-K and the other algorithms is statistically significant. At a 1% significance level, a Z-score with $|Z| > 2.58$ is considered significant. Z-scores and the corresponding p-values are reported in Table 4.

6.1 Effects of Engineering Efforts To evaluate the fingerprint functions $f_e^{\oplus}$ and f_e^2 for parallel net detection, we coarsened all 294 hypergraphs and, for each hypergraph, recorded the number of false positives, i.e. nets that had the same fingerprint albeit *not* being parallel. On average, $f_e^{\oplus}$ produced 67 693.06 false positives, while f_e^2 produced only 7.17 false positives. Additionally, we analyzed the proportion of times a simple comparison of the net sizes was enough to resolve the fingerprint collision and thereby avoiding the actual comparison of the pin sets. 75.95% of all false positives induced by f_e^2 could be resolved in that manner. For $f_e^{\oplus}$, it was only 43.41%.

As can be seen in Table 1 the simplified coarsening algorithm (+C) is more than three times faster than the algorithm (−C) of [19]. Activating gain caching (+G) reduces the running time by 45%, while adaptively stopping (+AS) local search leads to a factor of 9 speedup at the cost of a slight decrease in solution quality. Excluding nets from delta-gain-updates further

Table 1: Average cut and running times of the coarsening (t_c) and local search phase (t_{ls}) with activated tunings: simplified coarsening (C), gain caching (GC), adaptive stopping (AS), and excluding nets from Delta-Gain-Updates (NE) for the benchmark subset. A $*$ is used to denote no effect.

C	GC	AS	NE	cut	t_c[s]	t_{ls}[s]
−	−	−	−	6506	1.84	56.87
+	−	−	−	6509	0.50	*
+	+	−	−	6505	*	31.20
+	+	+	−	6537	*	3.48
+	+	+	+	6537	*	3.06

Table 2: Effects of applying the pin sparsifier using all instances.

		use sparsifier		
		never	always	if $\lvert\tilde{e}\rvert \geq 28$
$\lvert\tilde{e}\rvert \geq 28$	cut	12 891	12 879	12 879
	t[s]	20.0	13.3	13.3
$\lvert\tilde{e}\rvert < 28$	cut	7480	7482	7480
	t[s]	14.8	15.0	14.8

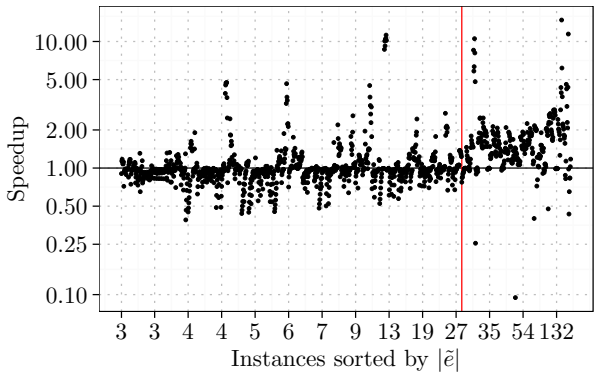

Figure 2: Threshold for pin sparsifier.

Table 3: Comparing the average running times of KaHyPar-K with KaHyPar-R and KaHyPar-R* on the benchmark subset (top) and with other partitioners on the full benchmark set (bottom).

Algorithm	$\lvert\tilde{e}\rvert \geq 3$	$\lvert\tilde{e}\rvert < 3$	$\lvert\tilde{e}\rvert \geq 28$
KaHyPar-K	7.4	24.6	6.8
KaHyPar-R	25.3	73.3	42.0
KaHyPar-R*	11.4	30.7	14.2
KaHyPar-K	10.9	26.7	13.3
hMetis-R	45.1	103.6	90.0
hMetis-K	37.2	75.3	92.6
PaToH-Q	3.8	6.3	10.4
PaToH-D	0.8	1.1	3.1

reduces the running time of local search by 12%. The sparsifier presented in Section 4 is intended to improve the running time for hypergraphs with large nets. As can be seen in Figure 2 and in Table 2, the preprocessing slows down the partitioning process if the median net size $|\tilde{e}|$ is small. For hypergraphs with larger median net sizes however, the sparsifier speeds up the partitioning process by a factor of 1.5. In all following experiments, we therefore employ it only for hypergraphs with $|\tilde{e}| \geq 28$.

6.2 Comparison with other Systems The upper parts of Figure 4 and Table 3 compare the performance of KaHyPar-K with KaHyPar-R and a version of KaHyPar-R that uses the simplified coarsening algorithm and the adaptive stopping rule for local search (KaHyPar-R*) on the benchmark subset. For hypergraphs with $|\tilde{e}| \geq 3$, KaHyPar-K produces significantly better cuts than both -R and -R*. If at least half of the nets are normal graph edges, the cuts of -R are better that those of -K, while those of -R* are worse. Comparing the running times, we see that KaHyPar-K is more than 2.5 times faster than -R and also outperforms -R*.

In the following, we exclude 55 out of 2058 instances, because either PaToH-Q could not allocate enough memory or other partitioners did not finish in time. The following comparison is therefore based on the remaining 2003 test instances.[5] Note that hMetis-K produces infeasible solutions for more than 500 instances (with up to 13.4% imbalance). As can be seen in Figure 4 (middle), KaHyPar-K performs best for hypergraphs with $|\tilde{e}| \geq 3$. It produces the best partitions for 794 of the 1327 instances and performs significantly better than hMetis-R, PaToH-Q, PaToH-D and hMetis-K. For hypergraphs with $|\tilde{e}| < 3$ hMetis-R computes better partitions than KaHyPar-K. The solution quality of KaHyPar-K is comparable to hMetis-K in this case. Comparing KaHyPar-K with PaToH, we see that it performs significantly better than both PaToH-Q and PaToH-D. For median net sizes $|\tilde{e}| \geq 28$ KaHyPar-K performs best on more than two thirds of the instances (see Figure 3 (top)). The running times of each partitioner are compared the in lower part of Table 3. On average, KaHyPar-K is more than a factor of 4 faster than hMetis-R and more than 2.5 times faster than hMetis-K, while PaToH is still the fastest partitioner. However this is not surprising, since n-level partitioning comes with an inherent overhead. Taking *both* running time and quality into account, KaHyPar-K dominates hMetis-R and hMetis-K for instances with $|\tilde{e}| \geq 3$. If the median net size is small, it is several times faster than both versions of hMetis while producing partitions that are only slightly worse than hMetis-R.

Figure 4 (bottom) and Table 4 (bottom) compare the performance of all partitioners for values of k

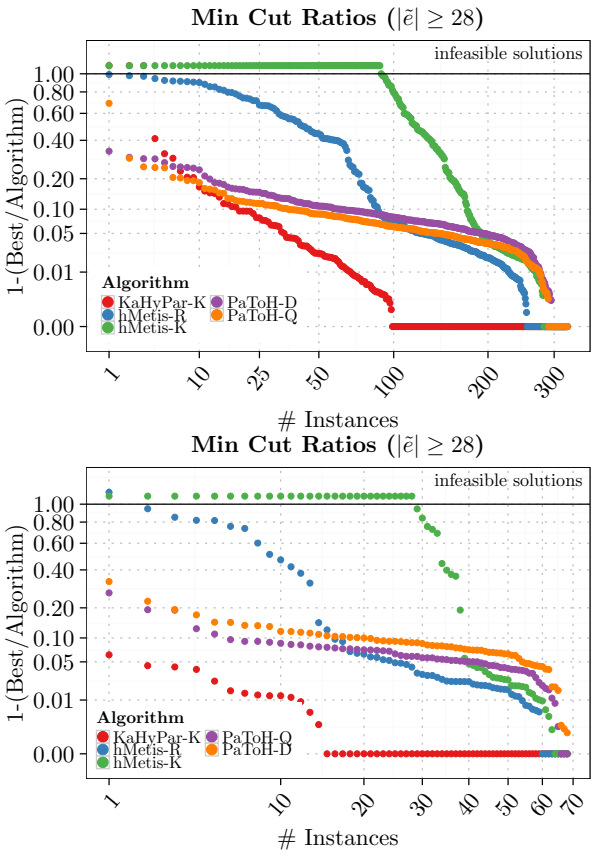

Figure 3: Performance plots comparing KaHyPar-K with other systems on the full benchmark set (top) and with other systems on the benchmark subset with $k \in \{5, 23, 47, 107\}$ (bottom). The plots show instances with $|\tilde{e}| \geq 28$.

[5] **Interactive** visualizations of the performance plots, detailed per-instance results, and the list of excluded instances can be found on the website accompanying this publication: `http://algo2.iti.kit.edu/schlag/alenex2017/`.

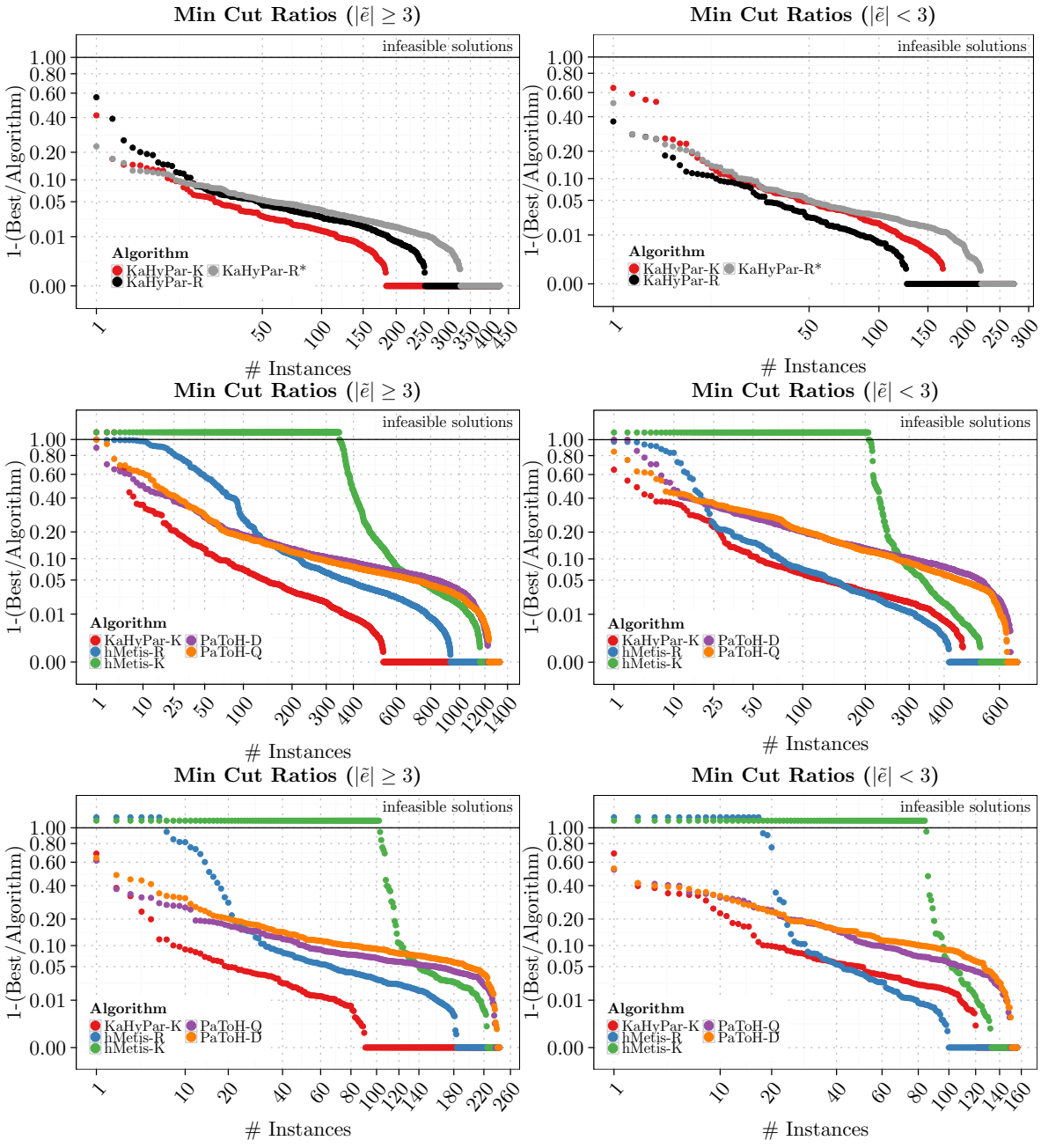

Figure 4: Performance plots comparing KaHyPar-K with KaHyPar-R and KaHyPar-R* on the benchmark subset (top), with other systems on the full benchmark set (middle), and with other systems on the benchmark subset with $k \in \{5, 23, 47, 107\}$ (bottom). Instances are divided into two classes: Hypergraphs with median net size $|\tilde{e}| \geq 3$ and $|\tilde{e}| < 3$.

Table 4: Results of significance tests comparing KaHyPar-K with KaHyPar-R and KaHyPar-R* on the benchmark subset (top), with other systems on the full benchmark set (middle), and with other systems on the benchmark subset with $k \in \{5, 23, 47, 107\}$ (bottom).

Algorithm	KaHyPar-K					
	$\lvert\tilde{e}\rvert \geq 3$		$\lvert\tilde{e}\rvert < 3$		$\lvert\tilde{e}\rvert \geq 28$	
	Z	p	Z	p	Z	p
KaHyPar-R	-5.3895	$7.067 \cdot 10^{-8}$	2.1913	0.028 43	-4.3614	$1.292 \cdot 10^{-5}$
KaHyPar-R*	-8.4088	$2.2 \cdot 10^{-16}$	-1.5587	0.1191	-3.6184	0.0002965
hMetis-R	-13.633	$2.2 \cdot 10^{-16}$	2.2631	0.023 63	-10.968	$2.2 \cdot 10^{-16}$
hMetis-K	-19.001	$2.2 \cdot 10^{-16}$	-0.7768	0.4373	-12.38	$2.2 \cdot 10^{-16}$
PaToH-D	-26.329	$2.2 \cdot 10^{-16}$	-18.346	$2.2 \cdot 10^{-16}$	-12.245	$2.2 \cdot 10^{-16}$
PaToH-Q	-24.935	$2.2 \cdot 10^{-16}$	-17.147	$2.2 \cdot 10^{-16}$	-11.13	$2.2 \cdot 10^{-16}$
hMetis-R	-6.5383	$6.223 \cdot 10^{-11}$	3.3557	0.000 791 6	-6.068	$1.295 \cdot 10^{-9}$
hMetis-K	-8.3024	$2.2 \cdot 10^{-16}$	4.0767	$4.569 \cdot 10^{-5}$	-6.0558	$1.398 \cdot 10^{-9}$
PaToH-D	-12.71	$2.2 \cdot 10^{-16}$	-8.6834	$2.2 \cdot 10^{-16}$	-7.1369	$9.546 \cdot 10^{-13}$
PaToH-Q	-12.358	$2.2 \cdot 10^{-16}$	-7.903	$2.722 \cdot 10^{-15}$	-7.0331	$2.02 \cdot 10^{-12}$

that are non-powers of two. KaHyPar-K performs significantly better than hMetis-R, PaToH-Q, PaToH-D and hMetis-K for hypergraphs with $|\tilde{e}| \geq 3$. For hypergraphs with $|\tilde{e}| < 3$ both versions of hMetis compute better partitions. However, more than half of the solutions of hMetis-K are imbalanced (with up to 27% imbalance). hMetis-R produced 17 imbalanced partitions (with up to 67% imbalance). Figure 3 (bottom) shows the partitioning results for median net sizes $|\tilde{e}| \geq 28$. KaHyPar-K produces the best partitions for 54 of the 68 instances and performs significantly better than all other partitioners.

7 Conclusions and Future Work

We describe a bundle of improvements to n-level hypergraph partitioning that make it a fast high-quality approach for optimizing the connectivity metric. Since this works quite reliably over a large spectrum of instances, we release KaHyPar as a hypergraph partitioner for general use. Our implementation is available from `https://github.com/SebastianSchlag/kahypar`. We are currently working on further ideas to improve quality and speed. For example, one might want to adapt the flow methods [57] and evolutionary methods [58] successfully used for graph partitioning. Also parallelization is an important issue.

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] 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.

[4] 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.

[5] S. Wichlund. On multilevel circuit partitioning. In *1998 International Conference on Computer-aided Design*, ICCAD, pages 505–511. ACM, 1998.

[6] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[7] S. Klamt, U. Haus, and F. Theis. Hypergraphs and Cellular Networks. *PLoS Comput Biol*, 5(5):e1000385, 05 2009.

[8] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *International Symposium on Solving Irregularly Structured Problems in Parallel*, pages 218–225, 1998.

[9] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.

[10] Thang Nguyen Bui and Curt Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992.

[11] 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.

[12] 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.

[13] S. Hauck and G. Borriello. An Evaluation of Biparti-

tioning Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, Aug 1997.

[14] B. Hendrickson and R. Leland. A Multi-Level Algorithm For Partitioning Graphs. *SC Conference*, 0:28, 1995.

[15] 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.

[16] 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.

[17] B. Uar and C. Aykanat. Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[18] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, September 1997.

[19] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. *k-way Hypergraph Partitioning via n-Level Recursive Bisection*, chapter 4, pages 53–67. 2016.

[20] C. Fiduccia and R. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *19th ACM/IEEE Design Automation Conf.*, pages 175–181, 1982.

[21] A. Trifunovi and W. J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563 – 581, 2008.

[22] 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. [31], pages 53–66.

[23] L. A. Sanchis. Multiple-way Network Partitioning. *IEEE Trans. on Computers*, 38(1):62–81, 1989.

[24] L. A. Sanchis. Multiple-way network partitioning with different cost functions. *IEEE Transactions on Computers*, 42(12):1500–1504, Dec 1993.

[25] J. Gong and S. K. Lim. Multiway Partitioning with Pairwise Movement. In *IEEE/ACM Int. Conf. on Computer-Aided Design*, pages 512–516, 1998.

[26] M. Wang, S. Lim, J. Cong, and M. Sarrafzadeh. Multiway partitioning using bi-partition heuristics. In *2000 Asia and South Pacific Design Automation Conference*, ASP-DAC, pages 667–672, 2000.

[27] A. Trifunovic and W. J. Knottenbelt. A parallel algorithm for multilevel k-way hypergraph partitioning. In *Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 114–121, July 2004.

[28] V. Osipov and P. Sanders. n-Level Graph Partitioning. In *18th European Symposium on Algorithms (ESA)*, volume 6346 of *LNCS*, pages 278–289, 2010.

[29] A. Trifunovic. *Parallel Algorithms for Hypergraph Partitioning*. PhD thesis, University of London, 2006.

[30] C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: a Survey. *Integration, the VLSI Journal*, 19(1–2):1 – 81, 1995.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1-6):381–413, 1992.

[38] 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.

[39] V. Henne, H. Meyerhenke, P. Sanders, S. Schlag, and C. Schulz. n-Level Hypergraph Partitioning. Technical Report arXiv:1505.00693, KIT, May 2015.

[40] W.E. Donath. Logic partitioning. *Physical Design Automation of VLSI Systems*, pages 65–86, 1988.

[41] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[42] Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. Hypergraph sparsification and its application to partitioning. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 200–209, Washington, DC, USA, 2013. IEEE Computer Society.

[43] Taher H. Haveliwala. Scalable techniques for clustering the web. In *In Proc. of the WebDB Workshop*, pages

129–134, 2000.

[44] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-Wise Independent Permutations. volume 60, pages 630–659, Orlando, FL, USA, June 2000. Academic Press, Inc.

[45] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

[46] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH Forest: Self-tuning Indexes for Similarity Search. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 651–660, New York, NY, USA, 2005. ACM.

[47] Ü. V. Catalyürek and C. Aykanat. PaToH: Partitioning Tool for Hypergraphs. `http://bmi.osu.edu/umit/PaToH/manual.pdf`, 1999.

[48] M. Deveci, K. Kaya, and Ü. V Catalyürek. Hypergraph sparsification and its application to partitioning. In *42nd International Conference on Parallel Processing*, ICPP, pages 200–209, 2013.

[49] T. Heuer. Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning. Bachelor's thesis, KIT, 2015.

[50] P. Briggs and L. Torczon. An Efficient Representation for Sparse Sets. *ACM Lett. Program. Lang. Syst.*, 2(1-4):59–69, March 1993.

[51] S. Schlag. Benchmark Hypergraphs and Detailed Experimental Results. `http://dx.doi.org/10.5281/zenodo.30176`, September 2015.

[52] 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.

[53] 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.

[54] A. Belov, D. Diepold, M. Heule, and M. Järvisalo. The SAT Competition 2014. `http://www.satcompetition.org/2014/`, 2014.

[55] Nicholas J. Cox. Stata tip 96: Cube roots. *Stata Journal*, 11(1):149–154(6), 2011.

[56] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[57] 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.

[58] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *ALENEX 2012*, pages 16–29. SIAM, 2012.

[59] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[60] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

A Evaluation of kPaToH

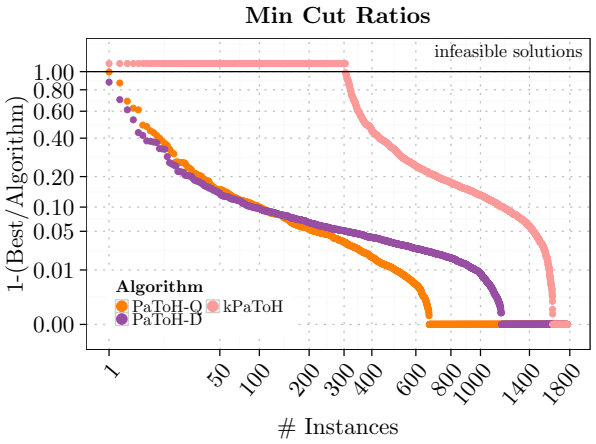

Figure 5: Performance plots comparing kPaToH and PaToH.