Advanced Flow-Based Multilevel Hypergraph Partitioning

Lars Gottesbüren
Karlsruhe Institute of Technology, Germany
lars.gottesbueren@kit.edu

Michael Hamann
Karlsruhe Institute of Technology, Germany
michael.hamann@kit.edu

Sebastian Schlag
Karlsruhe Institute of Technology, Germany
sebastian.schlag@kit.edu

Dorothea Wagner
Karlsruhe Institute of Technology, Germany
dorothea.wagner@kit.edu

Abstract

The balanced hypergraph partitioning problem is to partition a hypergraph into k disjoint blocks of bounded size such that the sum of the number of blocks connected by each hyperedge is minimized. We present an improvement to the flow-based refinement framework of KaHyPar-MF, the current state-of-the-art multilevel k-way hypergraph partitioning algorithm for high-quality solutions. Our improvement is based on the recently proposed HyperFlowCutter algorithm for computing bipartitions of unweighted hypergraphs by solving a sequence of incremental maximum flow problems. Since vertices and hyperedges are aggregated during the coarsening phase, refinement algorithms employed in the multilevel setting must be able to handle both weighted hyperedges and weighted vertices – even if the initial input hypergraph is unweighted. We therefore enhance HyperFlowCutter to handle weighted instances and propose a technique for computing maximum flows directly on weighted hypergraphs.

We compare the performance of two configurations of our new algorithm with KaHyPar-MF and seven other partitioning algorithms on a comprehensive benchmark set with instances from application areas such as VLSI design, scientific computing, and SAT solving. Our first configuration, KaHyPar-HFC, computes slightly better solutions than KaHyPar-MF using significantly less running time. The second configuration, KaHyPar-HFC*, computes solutions of significantly better quality *and* is still slightly faster than KaHyPar-MF. Furthermore, in terms of solution quality, both configurations also outperform all other competing partitioners.

2012 ACM Subject Classification Mathematics of computing → Hypergraphs; Mathematics of computing → Network flows; Mathematics of computing → Graph algorithms

Keywords and phrases Hypergraph Partitioning, Maximum Flows, Refinement

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.11

Related Version `https://arxiv.org/abs/2003.12110`

Supplementary Material Source code `https://github.com/larsgottesbueren/WHFC`

Funding This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants WA654/19-2, WA654/22-2, and SA933/11-1. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

1 Introduction

Graphs are a common way to model pairwise relationships (edges) between objects (vertices). However, many real-world problems involve more complex interactions [7, 29]. Hypergraphs are a generalization of graphs, where each hyperedge can connect an arbitrary number of vertices. Thus, hypergraphs are well-suited to model such higher-order relationships.

The balanced k-way hypergraph partitioning problem (HGP) asks to compute a partition of the vertices into k disjoint blocks of bounded weight (at most $(1+\varepsilon)$ times the average block weight) such that few hyperedges are cut, i.e., connect vertices in different blocks [7, 5, 35]. Since hyperedges can connect more than two vertices, several notions of cuts exit in the literature [5]. In this work, we consider the *connectivity* objective, which aims to minimize $\sum_{e \in E} \omega(e)(\lambda(e)-1)$, where E is the set of hyperedges, $\omega(e)$ denotes the weight of hyperedge e, and $\lambda(e)$ denotes the number of blocks connected by hyperedge e. Well-known applications of hypergraph partitioning include VLSI design [5], the parallelization of sparse matrix-vector multiplications [9], and storage sharding in distributed databases [25]. We refer to a survey chapter [39, Ch. 3] and two survey articles [5, 35] for an extensive overview.

Since HGP is NP-hard [31], heuristic algorithms are used in practice. The most successful heuristic for computing high-quality solutions is the three-phase *multilevel* paradigm. In the *coarsening phase*, multilevel algorithms first successively contract the input hypergraph to obtain a hierarchy of smaller, structurally similar instances. After applying an *initial partitioning* algorithm to the smallest hypergraph, the contractions are undone and, at each level, refinement algorithms are used to improve the partition induced by the coarser level.

Related Work. The most well-known and practically relevant multilevel hypergraph partitioners from different application areas are PaToH [9] (scientific computing), hMetis [26, 27] (VLSI design), KaHyPar [24, 23, 2, 40] (general purpose, n-level), Zoltan [13, 41] (scientific computing, parallel), Mondriaan [45] (sparse matrices), and Par$k$way [42] (parallel). Additionally there are MLPart [4] (restricted to bipartitioning), and HYPE [33], a single-level algorithm that grows k blocks using a neighborhood expansion [46] heuristic.

With the exception of KaHyPar-MF [24], all multilevel HGP algorithms solely employ variations of Kernighan-Lin [28] (KL) or Fiduccia-Mattheyses [17] (FM) heuristics in the refinement phase. These algorithms repeatedly move vertices between blocks prioritized by the improvement in the objective function. While they perform well for hypergraphs with small hyperedges, their performance deteriorates in the presence of many large hyperedges [44]. In this case, many single-vertex moves have no immediate effect on the objective function because the vertices of large hyperedges are likely to be distributed over multiple blocks. KaHyPar-MF [24] therefore additionally employs a refinement algorithm based on maximum-flow computations between pairs of blocks. Since flow algorithms find minimum s-t cuts, this refinement technique does not suffer the drawbacks of move-based approaches.

The flow-based refinement framework is a generalization of the approach used in the graph partitioner KaFFPa [38]. To refine a pair of blocks of a k-way partition, KaHyPar first extracts a subhypergraph induced by a set of vertices around the cut of these blocks. This subhypergraph is then transformed into a graph-based flow network, using techniques due to Lawler [30], and Liu and Wong [32], on which a maximum flow is computed. The vertices of the subhypergraph are reassigned according to the corresponding minimum cut. The size of the subhypergraph (and thus the size of the flow network) is chosen adaptively, depending on the outcome of the previous refinement. While larger flow networks may produce better but potentially imbalanced solutions, the smallest flow network guarantees a balanced partition. We further discuss KaHyPar and its flow-based refinement framework in Section 2.

HyperFlowCutter (HFC) [22] computes bipartitions of unweighted hypergraphs by solving a sequence of incremental maximum flow problems. Its advantage over computing a single maximum flow is that it does not reject almost balanced solutions, but systematically trades cut-size for increased balance. ReBaHFC [22] uses HFC as postprocessing to improve an initial bipartition computed with PaToH. HFC computes unit-capacity flows directly on the hypergraph (i.e., without using a graph-based flow network) using a technique of Pistorius and Minoux [37] extended to Dinic's flow algorithm [14].

Contribution and Outline. Multilevel refinement algorithms must be able to handle both weighted hyperedges and weighted vertices because vertices and hyperedges are aggregated during the coarsening phase. In this work, we improve KaHyPar's flow-based refinement framework – with regard to both running time and solution quality – by using a *weighted* version of HFC instead of the maximum flow computations on differently-sized graph-based flow networks. After introducing notation and briefly presenting additional details about KaHyPar and HFC refinement in Section 2, we discuss how HFC can simulate an approach of KaHyPar and KaFFPa for balancing partitions, extend HFC's existing balancing approach to weighted instances, and propose a heuristic for guiding its incremental maximum flow problems in Section 3. In Section 4, we present our approach for computing maximum flows on weighted hypergraphs, generalizing the technique of Pistorius and Minoux [37] to weighted hypergraphs and arbitrary flow algorithms.

In our experiments (Section 5), we compare two configurations of our new approach with KaHyPar-MF, and seven other partitioning algorithms on a large benchmark set containing hypergraphs from the VLSI, SAT solving, and scientific computing community [23]. While our first configuration, KaHyPar-HFC, computes slightly better solutions than KaHyPar-MF using significantly less running time, the second configuration, KaHyPar-HFC*, computes solutions of significantly better quality and is still slightly faster than KaHyPar-MF. Furthermore, in terms of solution quality, both configurations outperform all other competing partitioners. We conclude the paper in Section 6 and suggest future work.

2 Preliminaries

Hypergraphs. A *hypergraph* $H = (V, E)$ consists of a set of vertices V and a set of hyperedges E, where a hyperedge e is a subset of the vertices V. Additionally, we associate weights $\omega \colon E \to \mathbb{N}^+$, $\varphi \colon V \to \mathbb{N}^+$ with the hyperedges and vertices. A vertex $v \in V$ is *incident* to hyperedge $e \in E$ if $v \in e$. The vertices incident to a hyperedge e are called the *pins* of e. We denote the pin u in hyperedge e as (u, e). By $H[V'] = (V', \{e \cap V' \mid e \in E\})$, we denote the hypergraph induced by the vertex set V'. The *star expansion* of H represents the hypergraph as a bipartite graph $G = (V \dot\cup E, \{(v, e) \in V \times E \mid v \in e\})$ with bipartite node set $V \dot\cup E$ and an edge for every pin. To avoid confusion, we use the terms vertices, hyperedges, and pins for hypergraphs, and the terms nodes and edges for graphs. We extend functions to sets using $f(X) = \sum_{x \in X} f(x)$ for some function f.

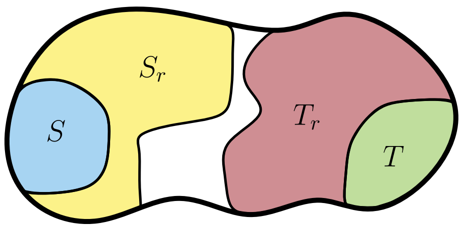
Hypergraph Partitioning. A k-way partition $\pi(H)$ of a hypergraph $H = (V, E)$ is a partition of its vertices into non-empty disjoint *blocks* $V_1, \dots, V_k \subset V$, i.e., $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $i = 1, \dots, k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. For some parameter $\varepsilon \in [0, 1)$ we call $\pi(H)$ ε-balanced, if each block V_i satisfies the *balance constraint* $\varphi(V_i) \leq (1 + \varepsilon)\frac{\varphi(V)}{k}$. Let $\Lambda(e) = \{V_i \in \pi(H) \mid V_i \cap e \neq \emptyset\}$ denote the blocks that are connected by hyperedge $e \in E$. The *connectivity* of a hyperedge e is defined as $\lambda(e) := |\Lambda(e)|$. Given parameters ε and k, and an input hypergraph H, the balanced k-way hypergraph partitioning problem asks for an ε-balanced k-way partition of H that minimizes the *connectivity-metric* $\sum_{e \in E} \omega(e)(\lambda(e) - 1)$.

Maximum Flows. A flow network is a symmetric, directed graph $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ with two disjoint non-empty *terminal* node sets $S, T \subsetneq \mathcal{V}$, the source and target node set, as well as a capacity function $c\colon \mathcal{E} \to \mathbb{N}_0$. A flow is a function $f\colon \mathcal{E} \to \mathbb{Z}$ subject to the *capacity constraint* $f(e) \leq c(e)$ for all edges e, *flow conservation* $\sum_{(u,v)\in\mathcal{E}} f(u,v) = 0$ for all non-terminal nodes v, and *skew symmetry* $f(u,v) = -f(v,u)$ for all edges (u,v). The *value* of a flow $|f| := \sum_{s\in S,(s,u)\in\mathcal{E}} f(s,u)$ is the amount of flow leaving S. The *residual capacity* $r_f(e) := c(e) - f(e)$ is the additional amount of flow that can pass through e without violating the capacity constraint. The residual network with respect to f is the directed graph $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f)$, where $\mathcal{E}_f := \{e \in \mathcal{E} | r_f(e) > 0\}$. A node v is *source-reachable* if there is a path from S to v in $\mathcal{N}_f$, it is *target-reachable* if there is a path from v to T in $\mathcal{N}_f$. We denote the source-reachable and target-reachable nodes by S_r and T_r, respectively. An *augmenting path* is an S-T path in $\mathcal{N}_f$. The flow f is a *maximum flow* if $|f|$ is maximal of all possible flows in $\mathcal{N}$. This is the case if and only if there is no augmenting path in $\mathcal{N}_f$. An S-T edge cut is a set of edges whose removal disconnects S and T. The value of a maximum flow equals the weight of a minimum-weight S-T edge cut [18]. The *source-side cut* consists of the edges from S_r to $\mathcal{V} \setminus S_r$ and the *target-side cut* consists of the edges from T_r to $\mathcal{V} \setminus T_r$. The bipartition $(S_r, \mathcal{V} \setminus S_r)$ is induced by the source-side cut and $(\mathcal{V} \setminus T_r, T_r)$ is induced by the target-side cut. Note that $\mathcal{V} \setminus S_r \setminus T_r$ is not necessarily empty. We also call S_r and T_r the *cutsides* of a maximum flow.

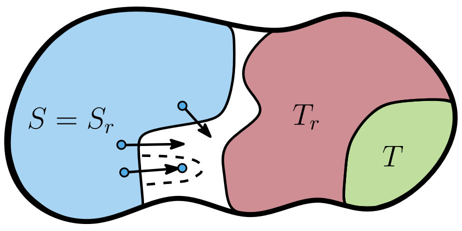
Maximum Flows on Hypergraphs. Lawler [30] uses maximum flows to compute minimum S-T hyperedge cuts. On the star expansion, the construction to model node capacities as edge capacities [1] is applied to the hyperedge-nodes. A hyperedge e is expanded into an *in-node* e_i and an *out-node* e_o joined by a directed *bridge edge* (e_i, e_o) with capacity $c(e_i, e_o) = \omega(e)$. For every pin $u \in e$ there are two directed *external edges* $(u, e_i), (e_o, u)$ with infinite capacity. The transformed graph is called the *Lawler network*. A minimum S-T edge cut in the Lawler network consists only of bridge edges, which directly correspond to S-T cut hyperedges in H. Via the Lawler network, the above notions translate naturally from graphs to hypergraphs, and we use the same terminology and notation for hypergraphs.

The KaHyPar Framework. Since our algorithm is integrated into the KaHyPar framework, we briefly review its core components and outline its k-way flow-based refinement. In contrast to traditional multilevel HGP algorithms that contract matchings or clusterings and therefore work with a coarsening hierarchy of $O(\log n)$ levels, KaHyPar removes only a single vertex between two levels, resulting in almost n levels. Coarsening is restricted to clusters found with a community detection algorithm [23]. Initial partitions of the coarsest hypergraph are computed using a portfolio of simple algorithms [40]. During uncoarsening, it employs a combination of localized FM local search [2, 40] and flow-based refinement [24].

Given a k-way partition $\pi(k) = \{V_1, ..., V_k\}$ of a hypergraph $H = (V, E, \varphi, \omega)$, the flow-based refinement works on pairs (V_i, V_j) of blocks that share cut hyperedges. The blocks are scheduled for refinement as long as this improves the solution (better connectivity, or equal connectivity and less imbalance). To construct a flow problem for a pair of blocks, the algorithm performs two randomized weight-constrained breadth-first searches (BFS) restricted to $H[V_i]$ and $H[V_j]$, respectively. The BFSs are initialized with the vertices of V_i (resp. V_j) that are incident to hyperedges in the cut between V_i and V_j. The first BFS stops if the weight of the visited vertices would exceed $(1 + \alpha \cdot \varepsilon)\lceil \frac{\varphi(V)}{k} \rceil - \varphi(V_j)$, where the scaling parameter α is used to control size of the flow problem. The second BFS visits the vertices of V_j analogously. The hypergraph induced by the visited vertices is then used to



(a) Compute minimum S-T cuts.



(b) Add S_r to S and choose a piercing vertex.

Figure 1 Flow augmentation and computing S_r, T_r in Fig.1a; adding S_r to S and piercing the source-side cut in Fig.1b. S in blue, $S_r \setminus S$ in yellow, T in green, $T_r \setminus T$ in red, $V \setminus S_r, \setminus T_r$ in white. Taken from Gottesbüren et al. [22] with minor adaptations.

build the Lawler network, with size optimizations due to Liu and Wong [32] and Heuer [24]. A minimum S-T cut in the Lawler network induces a bipartition of $H[V_i \cup V_j]$. If the flow computation resulted in an ε-balanced partition, the improved solution is accepted and α is increased to $\min(2\alpha, \alpha')$ for a predefined upper bound of $\alpha' = 16$. Otherwise, α is decreased to $\max(\alpha/2, 1)$. This scaling scheme continues until a maximal number of rounds is reached or a feasible partition that did not improve the cut is found. KaHyPar runs flow-based refinement on exponentially spaced levels, i. e., after 2^i uncontractions for increasing i, since flow-based refinement is too expensive to be run after every uncontraction.

ReBaHFC. ReBaHFC avoids the need for Lawler networks and the corresponding size optimizations [32, 24] by directly constructing a *flow hypergraph* by contracting vertices not visited by a BFS to S or T. Furthermore, it does not require adaptive rescaling because HFC guarantees balanced partitions.

3 Weighted HyperFlowCutter

To keep this paper self-contained, we briefly explain the core HFC algorithm in Section 3.1. Subsequently, we discuss further details of multilevel refinement with HFC in Section 3.2 and our approach for adapting two balancing strategies in Section 3.3.

3.1 The Core Algorithm

The core HFC algorithm computes bipartitions with monotonically increasing cut size and balance by solving a sequence of incremental maximum flow problems, until both blocks satisfy the balance constraint. Given some initial terminal vertex sets S and T, HFC first computes a maximum flow and derives the cutsides S_r and T_r. If either the source-side cut or the target-side cut induces blocks that fulfill the balance constraint, the algorithm terminates. Otherwise, it adds all vertices on the smaller cutside to its corresponding terminal side, i. e., $S := S_r$ if $|S_r| \leq |T_r|$ and $T := T_r$ otherwise. Then, one additional vertex – the *piercing vertex* – is added to the terminal side and the previous flow is augmented to respect the new terminals. This ensures that HFC finds a different cut with each new maximum flow. By growing the smaller side, the algorithm ensures that it finds a balanced partition after at most $|V|$ iterations. Figure 1 illustrates the HFC phases. HFC selects piercing vertices for S from the boundary vertices of the source-side cut, i. e., the vertices incident to hyperedges in the source-side cut that are not already contained in S. Analogously, the candidates for T are the boundary vertices of the target-side cut. Note that the previous flow still satisfies the flow constraints, and only the piercing vertex can break the maximality of the flow. HFC prefers piercing vertices that maintain the maximality of the current flow, i. e., do not create an augmenting path. This strategy is called the *avoid-augmenting-paths* piercing heuristic. Adding a vertex u to S creates an augmenting path if and only if $u \in T_r$.

3.2 Multilevel Refinement Using HyperFlowCutter

For multilevel k-way refinement with HyperFlowCutter we use the block-pair scheduling of KaHyPar-MF and the flow model construction of KaHyPar-MF and ReBaHFC. Unlike ReBaHFC, we do not explicitly contract unvisited vertices to S (resp. T). Instead, we build the flow hypergraph during the BFS and mark pins that will not be in the flow hypergraph as terminals. Hyperedges containing both terminals are removed as they cannot be eliminated from the cut. Subsequently, our weighted HFC algorithm is run on the weighted flow hypergraph. We stop once the flow exceeds the weight of the remaining hyperedges from the original cut. The difference between the weight of the original cut hyperedges and the flow value equals the decrease in connectivity.

Distance-Based Piercing. We can use the original cut to guide HFC. To avoid that bad piercing decisions make it impossible for HFC to recover parts of the original cut, we use BFS-distances from the original cut as an additional piercing heuristic. We prefer larger distances, secondary to avoiding augmenting paths. Vertices from the other side of the original cut are rated with distance -1, i. e., chosen only after one side has been entirely added to the corresponding terminal vertices. This is similar to the flow network rescaling of KaHyPar, as we first use vertices as terminals that could only be contained in larger flow networks. We maintain the boundary vertices in a bucket priority queue and select candidates uniformly at random from the highest-rated non-empty bucket. New terminal vertices are removed lazily.

3.3 Improved Balance

KaHyPar uses both flows and FM local search to refine a partition. Because FM only considers moves that maintain the balance constraint, partitions with small imbalance tend to give FM more leeway for improving the current solution. In this section, we discuss two approaches to improve the balance during HFC refinement. These can also improve the solution quality, since HFC would otherwise trade better balance for a larger cut.

Keep Piercing. Given a maximum flow and minimum cut, finding a most balanced cut of the same weight is NP-hard [8]. All vertices of a strongly connected component (SCC) of the residual network belong to the same side in a minimum cut. Hence, finding a most balanced minimum cut corresponds to a knapsack problem with partial order constraints induced by the directed acyclic graph (DAG) obtained from contracting all SCCs [36]. Each topological ordering of the DAG corresponds to a series of minimum cuts. KaHyPar computes several random topological orderings to find a cut with the same weight and less imbalance, which considerably improved solution quality [24].

Using the avoid-augmenting-paths piercing heuristic, we can perform such a sweep without the need to explicitly construct the DAG and to compute a topological ordering. Instead of stopping at the first balanced partition, we continue to pierce as long as no augmenting path is created. Since this process is fast and piercing decisions are randomized, we repeat it several times and select the partition with the smallest imbalance.

Reassigning Isolated Vertices. A vertex $v \notin S \cup T$ is called *isolated* if all of its incident hyperedges have pins in both S and T. An isolated vertex v can be moved without affecting the cut, because $v \notin S_r \cup T_r$ and its incident hyperedges remain in both the source- and target-side cut over the course of the algorithm. Using the terminology above, v is not affected by partial order constraints. This reduces the optimal assignment problem for isolated vertices to a subset sum problem. With unweighted vertices, we can easily distribute the total weight $|L|$ of a set L of isolated vertices among the two sides to achieve optimum balance. Introducing vertex weights makes the subset sum problem non-trivial, because arbitrary divisions of the total weight $\varphi(L)$ are not necessarily possible. Since isolated vertices remain isolated, the problem instances are incremental. To solve the problem, we use the pseudo-polynomial dynamic program (DP) for subset sum [10, Section 35.5]. It maintains a lookup table of partition weights that are summable with isolated vertices. After obtaining a new cut, we update the DP table to incorporate potential new isolated vertices. For each new isolated vertex v, we iterate over the subset sums x in the table and insert $\varphi(v) + x$ if it was not yet a subset sum. As an optimization, we maintain a list of ranges that are summable (consecutive entries). The balance check takes constant time per range. To merge ranges efficiently, we store a pointer from each DP table entry to its range in the list. When a new subset sum x is obtained, we check whether $x - 1$ and $x + 1$ are also subset sums, and extend or merge ranges as appropriate. Since entries in the DP table cannot be reverted easily, we do not add isolated vertices to the DP after the first balanced partition is found.

As the DP is only pseudo-polynomial, its running time may become prohibitive on instances with very large vertex weights. In our experiments, non-unit vertex weights are only the result of contractions during coarsening. Hence, the DP is polynomial in the size of the unit-weight input hypergraph. We plan to implement a simple classifier to decide whether the running time can be harmful, and deactivate the DP accordingly.

4 Maximum Flows on Weighted Hypergraphs

In this section, we present our technique for computing maximum flows on weighted hypergraphs. It generalizes the approach of Pistorious and Minoux [37] (which computes maximum flows directly on unweighted hypergraphs using the Edmonds-Karp algorithm [16]) to weighted hypergraphs and to arbitrary flow algorithms.

Let P denote the set of pins, and let $\widetilde{f}(u, e) \colon P \to \mathbb{Z}$ denote the amount of flow that vertex u sends into hyperedge e. Negative values indicate that u receives flow from e. Let $\widetilde{f}(u, e)^+ := \max(\widetilde{f}(u, e), 0)$ denote the net flow u sends into e and $\widetilde{f}(u, e)^- := \max(-\widetilde{f}(u, e), 0)$ the net flow u receives from e. Then, $f(e)$ can also be written as $\sum_{u \in e} \widetilde{f}(u, e)^+ = \sum_{u \in e} \widetilde{f}(u, e)^-$. We can push up to $c(e) - f(e) + \widetilde{f}(u, e)^- + \widetilde{f}(v, e)^+$ flow from u via e to another pin $v \in e$. The advantage of implementing flow algorithms directly on the hypergraph is the ability to identify and skip cases in which we cannot push any flow. If $c(e) - f(e) = 0$ and $\widetilde{f}(u, e) \geq 0$, we only need to iterate over the pins $v \in e$ with $\widetilde{f}(v, e) > 0$. A graph-based flow algorithm on the residual Lawler network would scan the edge (e_i, v) for *every* pin $v \in e$. For unweighted hypergraphs, each $e \in E$ has at most one pin $v \in e$ with $\widetilde{f}(v, e) > 0$, which can be stored in a separate array of size $|E|$. Since HFC needs to compute flows in forward and backward direction, an additional array is needed for pins $v' \in e$ with $\widetilde{f}(v', e) < 0$. A weighted hyperedge can have many such pins, which would require arrays of size $|P|$. Instead, we divide the pins of e into subranges $\widetilde{f}(v, e)\{< 0, = 0, > 0\}$. When the sign of $\widetilde{f}(v, e)$ changes, we insert pin v into the correct subrange by performing swaps with the range boundaries.

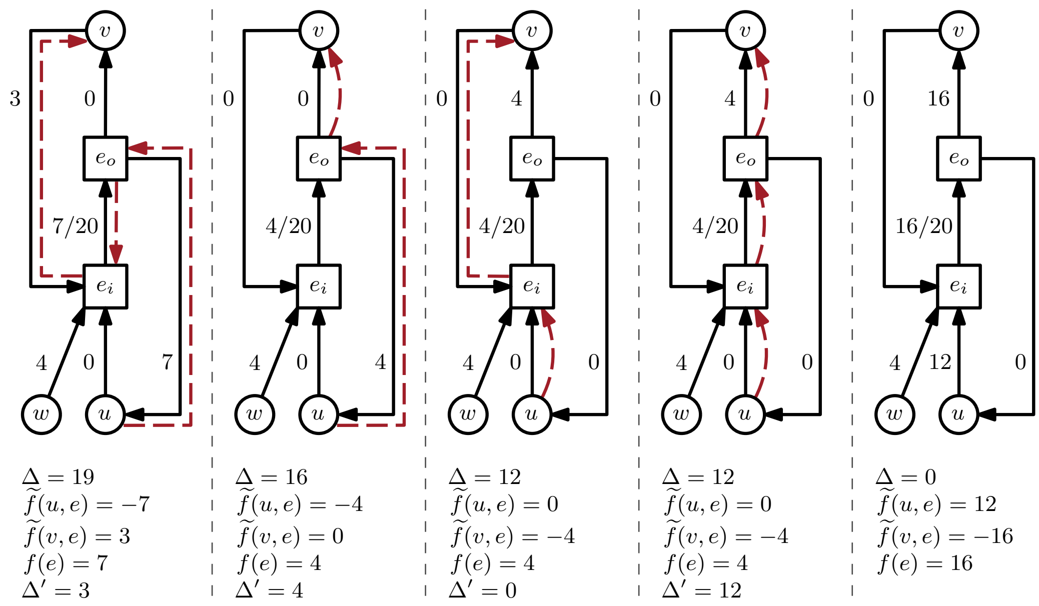


Figure 2 Example illustrating the four steps for pushing $\Delta = 19$ units of flow from u via hyperedge $e = \{u, v, w\}$ to v. Black edges show the direction of the flow, dashed red arrows the direction we want to push flow in the Lawler network. The edge (e_0, w) is omitted for readability. Current values of Δ, $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, $f(e)$, and Δ' are shown at the bottom for each state.

In addition to scanning hyperedges like vertices, pushing flow over a hyperedge is the elementary operation necessary to implement any flow algorithm on hypergraphs. Let Δ be the amount of flow to push. We update the values $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and $f(e)$ in four steps (see Figure 2 for an example). These steps correspond to paths $p_1 = (u, e_o, e_i, v)$, $p_2 = (u, e_o, v)$, $p_3 = (u, e_i, v)$, $p_4 = (u, e_i, e_o, v)$ in the residual Lawler network. The order of these steps is important to correctly update $f(e)$. First, we push $\Delta' := \min(\Delta, \widetilde{f}(u,e)^-, \widetilde{f}(v,e)^+)$ along p_1 by setting $f(e) := f(e) - \Delta'$, $\widetilde{f}(u,e) := \widetilde{f}(u,e) + \Delta'$, $\widetilde{f}(v,e) := \widetilde{f}(v,e) - \Delta'$, and $\Delta := \Delta - \Delta'$. Then, we push $\Delta' := \min(\Delta, \widetilde{f}(u,e)^-)$ along p_2, by updating $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and Δ as before. Note that we do not update $f(e)$ since the bridge edge (e_i, e_o) is not in p_2. Analogously to p_2, we push $\Delta' := \min(\Delta, \widetilde{f}(v,e)^+)$ along p_3. Finally, we push the remaining Δ along p_4 and update $f(e)$, $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and Δ as for p_1.

Note that the Lawler network is just used as a means of illustration. In our implementation, we only update the $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and $f(e)$ values as shown at the bottom of Figure 2.

Flow Algorithm. We chose to implement Dinic's flow algorithm [14] with capacity scaling, because the flow problems for HFC refinement on our benchmark set predominantly have a small diameter (due to the BFS-based construction). Dinic's algorithm consists of two alternating phases: assigning hop-distance labels to vertices by performing a BFS on the residual network, and using DFS to find edge-disjoint augmenting paths with distance labels increasing by one along the path. In addition to vertex distance labels, we maintain hyperedge distance labels. For a vertex u, we only traverse those incident hyperedges e whose distance label matches that of u. The pins $v \in e$ are only traversed if the distance of v is 1 plus the distance of e. To distinguish the case that we can only push flow to pins v with $\widetilde{f}(v,e) > 0$, we actually maintain two different distance labels per hyperedge.

Optimizations. It suffices to initialize the BFS and DFS with the last piercing vertex p of a side, since only p can lead to newly reachable vertices. When Dinic's algorithm terminates, we already know S_r or T_r (depending on which side was pierced) and thus only compute the reachable vertices of the other side. While computing this set, we also compute the corresponding distance labels, so that the next flow computation can directly start with the DFS phase. Additionally, we infer the sets S_r, T_r, S, and T from the distance labels.

5 Experimental Evaluation

The C++17 source code for Weighted HyperFlowCutter[1] and KaHyPar-HFC[2] are available as open-source software. Experiments are performed sequentially on a cluster of Intel Xeon E5-2670 Sandy Bridge nodes with two Octa-Core processors clocked at 2.6 GHz with 64 GB RAM, 20 MB L3- and 8×256 KB L2-Cache, using only one core of a node.

We consider two configurations which differ in the constraint for vertices from V_i in the flow hypergraph. KaHyPar-HFC uses $\frac{1}{5} \cdot \varphi(V_i)$ (as used for RebaHFC [22]), while KaHyPar-HFC* uses $(1 + 16 \cdot \varepsilon)\lceil\frac{\varphi(V)}{k}\rceil - \varphi(V_j)$ (as used for KaHyPar-MF [24]). In the TR [21] we assess the impact of the algorithmic components on the solution quality of KaHyPar-HFC. Both configurations use all components.

Benchmark Set. We use a comprehensive benchmark set of real-world hypergraphs compiled by Schlag [23].[3] It consists of 488 unit-weight hypergraphs from four sources: the ISPD98 VLSI Circuit Benchmark Suite [3] (ISPD98, 18 hypergraphs), the DAC 2012 Routability-Driven Placement Benchmark Suite [34] (DAC, 10), the SuiteSparse Matrix Collection [12] (SPM, 184) and the international SAT Competition 2014 [6] (Literal, Primal, Dual, 92 hypergraphs each). Refer to the TR [21] for hypergraph sizes and refer to [23] for information on how the hypergraphs were derived. We compute partitions for $\varepsilon = 3\%$ and $k \in \{2, 4, 8, 16, 32, 64, 128\}$. Each combination of a hypergraph and a value of k constitutes one *instance*, resulting in a total of 3416 instances.

Methodology. In addition to the KaHyPar configurations, we consider hMetis [26, 27] in both recursive bisection (-R) and direct k-way (-K) mode, PaToH [9] with default (-D) and quality preset (-Q), Zoltan-AlgD [41], Mondriaan [45], as well as HYPE [33].

For each instance and partitioner, we perform ten runs with different random seeds. The only exception is HYPE [33] which produced worse solutions when randomized [24]. Hence, we report only one non-randomized run of HYPE. Running times and connectivity values per instance are aggregated using the arithmetic mean, while running times across instances are aggregated using the geometric mean to give instances of different sizes a comparable influence. To compare running times we use a combination of a scatter plot, which shows the arithmetic mean time per instance, and a box-and-whiskers plot [43]. Because small running times are more frequent, we use a fifth-root scale [11] on the y-axis. Runs that exceeded the 8 hour time limit count as 8 hours in the reported aggregates and plots.

[1] `https://github.com/larsgottesbueren/WHFC`
[2] `https://github.com/SebastianSchlag/kahypar`
[3] `https://algo2.iti.kit.edu/schlag/sea2017/`

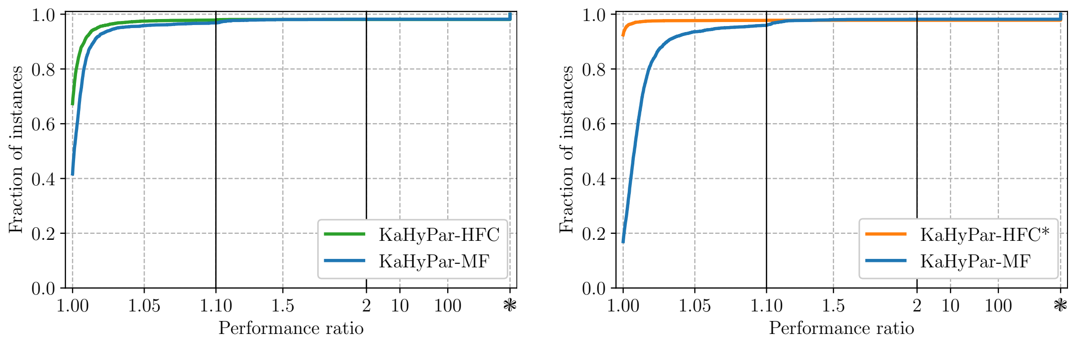


Figure 3 Performance profiles comparing our new variants with KaHyPar-MF.

For comparing solution quality we use performance profiles [15]. Let $\mathcal{A}$ denote a set of algorithms, $\mathcal{I}$ a set of instances and $\mathrm{obj}(a, i)$ denote the objective value computed by $a \in \mathcal{A}$ on $i \in \mathcal{I}$ – in our case the arithmetic mean connectivity of 10 runs. The performance ratio

$$r(a, i) = \frac{\mathrm{obj}(a, i)}{\min\{\mathrm{obj}(a', i) \mid a' \in \mathcal{A}\}}$$

indicates by what factor a deviates from the best solution on instance i. In particular, algorithm a found the best solution on instance i if $r(a, i) = 1$. The performance profile

$$\rho_a \colon [1, \infty) \to [0, 1], \tau \mapsto \frac{|\{i \in \mathcal{I} \mid r(a, i) \leq \tau\}|}{|\mathcal{I}|}$$

of a is the fraction of instances for which it is within a factor of τ from the best solution. Runs that did not finish within the time limit or resulted in an error (balance violation or crash) are excluded. If this concerns all runs of an algorithm on an instance, we report the corresponding fractions as the steps at special symbols (❋, ✘ respectively).

5.1 Comparison with KaHyPar-MF

KaHyPar-HFC computes solutions with better, equal, or worse quality than KaHyPar-MF on 1933, 367, 1056 instances, respectively. On the remaining 60 instances neither finished within the time limit. As Figure 3 (left) shows, the performance ratios are consistently better, though not by a large margin. The median fraction of flow-based refinement time of KaHyPar-HFC vs KaHyPar-MF is 0.18, the 75th percentile is 0.26, and the 90th percentile is 0.51. Hence, the flow-based refinement of KaHyPar-HFC is significantly faster than that of KaHyPar-MF. Flow-based refinement constitutes about 40% of KaHyPar-MF's overall running time [24]. With a mean overall running time of 44.84s, KaHyPar-HFC is about 33% faster than KaHyPar-MF at 67.07s. The improved running time is partially due to faster flow computation and partially due to smaller flow hypergraphs. Since KaHyPar-HFC uses smaller flow hypergraphs, the improved solution quality can be attributed to the HyperFlowCutter approach. With 62.49s, KaHyPar-HFC* is moderately faster than KaHyPar-MF and computes solutions of better, equal, or worse quality on 2776, 381, 198 instances, respectively (with 61 instances on which neither the -HFC* nor the -MF variant finished within the time limit). Hence, the faster flow computation more than compensates the additional work incurred by HFC. Figure 4 shows box plots for the different phases of KaHyPar. The running times of preprocessing, coarsening, and initial partitioning remain unchanged, as they are not influenced by the refinement phase. During the refinement phase,

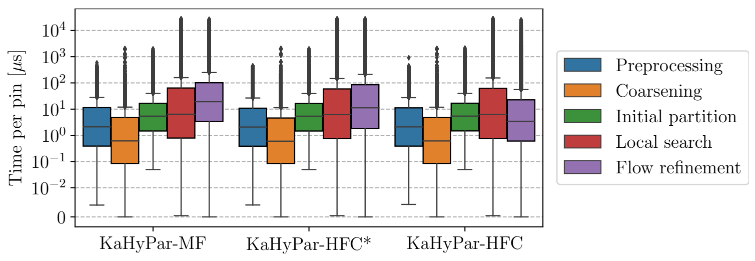


Figure 4 Box plots comparing running times (in μs per pin) of the different phases of KaHyPar.

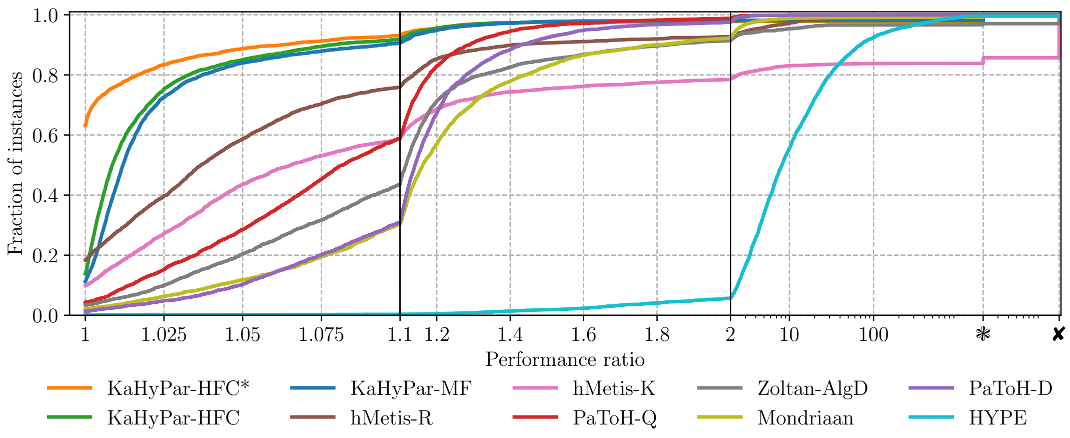


Figure 5 Performance profiles of all considered partitioners.

local search and flow-based improvement both modify the solution and thus influence one another. The plots show that the running time of local search remains largely unchanged, while our variants reduce the running time of flow-based refinement.

5.2 Comparison with other Partitioners

We now compare the three KaHyPar variants with the other state-of-the-art algorithms. Figure 5 shows that KaHyPar-HFC* outperforms all competing algorithms, and that hMetis-R emerges as the best competitor outside the KaHyPar variants. KaHyPar-HFC* computes the best solutions on 63% of all instances, KaHyPar-HFC on 14%, and hMetis-R on 18%, as shown by their $\rho_a(1)$ values. Note that these values alone do not permit a ranking between the algorithms. Both KaHyPar-MF and KaHyPar-HFC compete with KaHyPar-HFC* for the best solutions on similar instances, and thus end up with a lower $\rho_a(1)$ value. Compared individually, KaHyPar-HFC is better than hMetis-R on 69.9% of the instances. Additionally, KaHyPar-HFC and KaHyPar-MF approach the profile of KaHyPar-HFC* much faster. The KaHyPar variants are all within a factor of 1.1 of the best solution on over 90% of the instances, and within 1.4 on over 97%, whereas hMetis-R achieves 76% and 90%. PaToH-Q and PaToH-D solve more instances than hMetis-R within factors of roughly 1.2 and 1.4, and more instances than hMetis-K within 1.1 and 1.2. Mondriaan is similar to PaToH-D and Zoltan-AlgD settles between PaToH-D and PaToH-Q. The only non-multilevel algorithm HYPE is considerably worse, with only 5.7% of solutions within a factor 2 of the best.

Figure 6 shows running times for each instance. We categorize the algorithms into two groups. Algorithms in the first group, consisting of KaHyPar, hMetis and Zoltan-AlgD, invest substantial running time to aim for high-quality solutions. On the other hand, PaToH,

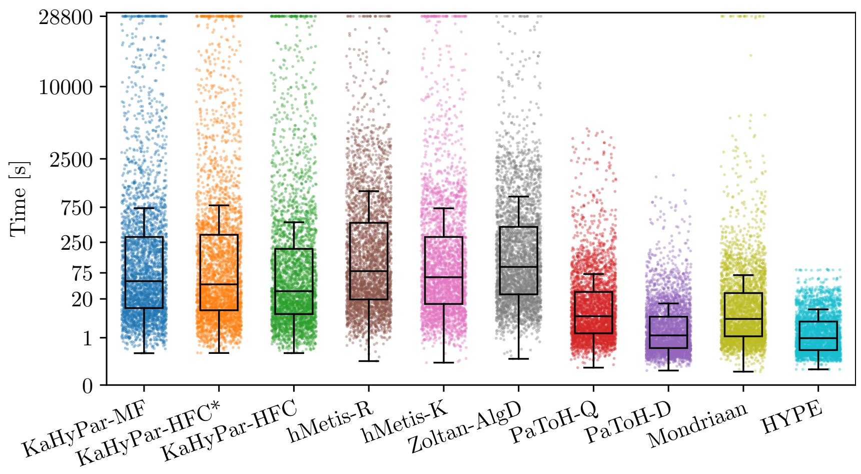


Figure 6 Box and scatter plots of arithmetic mean running times per instance.

Mondriaan and HYPE aim for fast running time and reasonable solution quality. The results show that while PaToH gives the best time-quality trade-off, KaHyPar-HFC* is the best algorithm for high-quality solutions, whereas KaHyPar-HFC offers a better time-quality trade-off than other algorithms from the first group.

In the TR [21] we report aggregate running times, timeouts and imbalanced solutions, present performance profiles for different values of k, the different instance classes of the benchmark set and a plot with only the best algorithm from each family of partitioning algorithms. The performance difference between KaHyPar and the other algorithms increases with k, which can be explained by the fact that all other partitioners except hMetis-K use recursive bisection. Furthermore, the improvement of KaHyPar-HFC* over KaHyPar-MF is especially pronounced on dual SAT instances, which have many large hyperedges.

6 Conclusion and Future Work

We leverage the powerful HyperFlowCutter refinement algorithm in the multilevel setting for k-way partitioning by integrating it into KaHyPar. For this, we extend unweighted HyperFlowCutter to weighted hypergraphs by adapting its balancing heuristics and presenting an approach to compute flows directly on weighted hypergraphs. Furthermore, we propose a distance-based piercing heuristic and use the existing avoid-augmenting-paths piercing heuristic to find partitions with small imbalance.

The most pressing area of research is to reduce the running time when using large flow hypergraphs, e. g., by further pruning of scheduled block pairs or more advanced flow algorithms like (E)IBFS [20, 19]. Furthermore, the impact of HFC refinement with small flow hypergraphs on fast and medium-quality partitioners such as PaToH could be a promising direction, since previous work on bipartitioning [22] already gave promising results.

References

1 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

2 Yaroslav Akhremtsev, Tobias Heuer, Sebastian Schlag, and Peter Sanders. Engineering a direct k-way Hypergraph Partitioning Algorithm. In *Proceedings of the 19th Meeting on*

Algorithm Engineering and Experiments (ALENEX'17), pages 28–42. SIAM, 2017. doi:10.1137/1.9781611974768.3.

3 Charles J. Alpert. The ISPD98 Circuit Benchmark Suite. In *Proceedings of the 1998 International Symposium on Physical Design*, pages 80–85, 1998. doi:10.1145/274535.274546.

4 Charles J. Alpert, Jen-Hsin Huang, and Andrew B. Kahng. Multilevel Circuit Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, August 1998. doi:10.1109/43.712098.

5 Charles J. Alpert and Andrew B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration: The VLSI Journal*, 19(1-2):1–81, 1995. doi:10.1016/0167-9260(95)00008-4.

6 Anton Belov, Daniel Diepold, Marijn JH Heule, and Matti Järvisalo. The SAT Competition 2014, 2014. URL: http://www.satcompetition.org/2014/.

7 Charles-Edmond Bichot and Patrick Siarry, editors. *Graph Partitioning*. Wiley, 2011. doi:10.1002/9781118601181.

8 Paul Bonsma. Most balanced minimum cuts. *Discrete Applied Mathematics*, 158(4):261–276, 2010. doi:10.1016/j.dam.2009.09.010.

9 Umit Catalyurek and Cevdet Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. doi:10.1109/71.780863.

10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

11 Nicholas J. Cox. Stata tip 96: Cube roots. *The Stata Journal*, 11(1):149–154, 2011. doi:10.1177/1536867X1101100112.

12 Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1), 2011. doi:10.1145/2049662.2049663.

13 Karen Devine, Erik Boman, Robert Heaphy, Rob Bisseling, and Umit Catalyurek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE Computer Society, 2006. doi:10.1109/IPDPS.2006.1639359.

14 Yefim Dinitz. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Mathematics-Doklady*, 11(5):1277–1280, September 1970.

15 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002. doi:10.1007/s101070100263.

16 Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972. doi:10.1145/321694.321699.

17 Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Conference on Design Automation*, pages 175–181, 1982. doi:10.1145/800263.809204.

18 Lester R. Ford, Jr. and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

19 Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E. Tarjan, and Renato F. Werneck. Faster and More Dynamic Maximum Flow by Incremental Breadth-First Search. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science*, pages 619–630. Springer, 2015. doi:10.1007/978-3-662-48350-3_52.

20 Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Robert E. Tarjan, and Renato F. Werneck. Maximum Flows by Incremental Breadth-First Search. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA'11)*, volume 6942 of *Lecture Notes in Computer Science*, pages 457–468. Springer, 2011. doi:10.1007/978-3-642-23719-5_39.

21 Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced flow-based multilevel hypergraph partitioning. *arXiv preprint arXiv:2003.12110*, 2020.

22 Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*, Leibniz International Proceedings in Informatics, pages 52:1–52:17, 2019. doi:10.4230/LIPIcs.ESA.2019.52.

23 Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*, volume 75 of *Leibniz International Proceedings in Informatics*, 2017. doi:10.4230/LIPIcs.SEA.2017.21.

24 Tobias Heuer, Sebastian Schlag, and Peter Sanders. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. *ACM Journal of Experimental Algorithmics*, 24(2):2.3:1–2.3:36, September 2019. doi:10.1145/3329872.

25 Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. *Proceedings of the VLDB Endowment*, 10(11):1418–1429, 2017. doi:10.14778/3137628.3137650.

26 George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999. doi:10.1109/92.748202.

27 George Karypis and Vipin Kumar. Multilevel K-Way Hypergraph Partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, 1999. doi:10.1145/309847.309954.

28 Brian W. Kernighan and Shen Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970. doi:10.1002/j.1538-7305.1970.tb01770.x.

29 Renaud Lambiotte, Martin Rosvall, and Ingo Scholtes. From networks to optimal higher-order models of complex systems. *Nature Physics*, 15(4):313–320, 2019. doi:10.1038/s41567-019-0459-y.

30 Eugene L. Lawler. Cutsets and Partitions of hypergraphs. *Networks*, 3:275–285, 1973. doi:10.1002/net.3230030306.

31 Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley, 1990.

32 Huiqun Liu and D.F. Wong. Network-Flow-Based Multiway Partitioning with Area and Pin Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):50–59, 1998. doi:10.1109/43.673632.

33 Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. HYPE: Massive Hypergraph Partitioning With Neighborhood Expansion. In *IEEE International Conference on Big Data*, pages 458–467. IEEE Computer Society, 2018. doi:10.1109/BigData.2018.8621968.

34 Viswanath Nagarajan, Charles J. Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 Routability-driven Placement Contest and Benchmark Suite. In *Proceedings of the 49th Annual Design Automation Conference*, 2012. doi:10.1145/2228360.2228500.

35 David A. Papa and Igor L. Markov. Hypergraph Partitioning and Clustering. In Teofilo F. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007. doi:10.1201/9781420010749.

36 Jean-Claude Picard and Maurice Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Mathematical Programming, Series A*, 22(1):121, December 1982. doi:10.1007/BF01581031.

37 Joachim Pistorius and Michel Minoux. An Improved Direct Labeling Method for the Max–Flow Min–Cut Computation in Large Hypergraphs and Applications. *International Transactions in Operational Research*, 10(1):1–11, 2003. doi:10.1111/1475-3995.00389.

38 Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA'11)*, volume

6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2011. doi:10.1007/978-3-642-23719-5_40.

39 Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2020.

40 Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way Hypergraph Partitioning via n-Level Recursive Bisection. In *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX'16)*, pages 53–67. SIAM, 2016. doi:10.1137/1.9781611974317.5.

41 Ruslan Shaydulin, Jie Chen, and Ilya Safro. Relaxation-Based Coarsening for Multilevel Hypergraph Partitioning. *Multiscale Modeling and Simulation*, 17(1):482–506, 2019. doi:10.1137/17M1152735.

42 Aleksandar Trifunovic and William J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563–581, 2008. doi:10.1016/j.jpdc.2007.11.002.

43 John W. Tukey. Box-and-whisker plots. *Exploratory data analysis*, pages 39–43, 1977.

44 Bora Uçar and Cevdet 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. doi:10.1137/S1064827502410463.

45 Brendan Vastenhouw and Rob Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005. doi:10.1137/S0036144502409019.

46 Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. Graph Edge Partitioning via Neighborhood Heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 605–614. ACM Press, 2017. doi:10.1145/3097983.3098033.