Parallel Flow-Based Hypergraph Partitioning

Lars Gottesbüren
Karlsruhe Institute of Technology, Karlsruhe, Germany

Tobias Heuer
Karlsruhe Institute of Technology, Karlsruhe, Germany

Peter Sanders
Karlsruhe Institute of Technology, Karlsruhe, Germany

Abstract

We present a shared-memory parallelization of *flow-based refinement*, which is considered the most powerful iterative improvement technique for hypergraph partitioning at the moment. Flow-based refinement works on bipartitions, so current sequential partitioners schedule it on different block pairs to improve k-way partitions. We investigate two different sources of parallelism: a parallel scheduling scheme and a parallel maximum flow algorithm based on the well-known push-relabel algorithm. In addition to thoroughly engineered implementations, we propose several optimizations that substantially accelerate the algorithm in practice, enabling the use on extremely large hypergraphs (up to 1 billion pins). We integrate our approach in the state-of-the-art parallel multilevel framework Mt-KaHyPar and conduct extensive experiments on a benchmark set of more than 500 real-world hypergraphs, to show that the partition quality of our code is on par with the highest quality sequential code (KaHyPar), while being an order of magnitude faster with 10 threads.

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

Keywords and phrases multilevel hypergraph partitioning, shared-memory algorithms, maximum flow

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.5

Related Version *Full Version*: `https://arxiv.org/abs/2201.01556`

Supplementary Material
Software (Multilevel Framework): `https://github.com/kahypar/mt-kahypar`
Software (Flow-Based Refinement): `https://github.com/larsgottesbueren/WHFC/tree/parallel`
Dataset (Benchmark Set & Experimental Results): `https://algo2.iti.kit.edu/heuer/sea22/`

Funding This work was partially supported by DFG grants WA654/19-2 and SA933/11-1. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

1 Introduction

Balanced hypergraph partitioning is a classical NP-hard optimization problem with numerous applications. Hypergraphs are a generalization of graphs, where each hyperedge can connect an arbitrary number of vertices. The problem is to partition the vertices of a hypergraph $H = (V, E, \omega)$ into k disjoint blocks $V_1, \dots V_k$ of roughly equal size ($\forall V_i : |V_i| \leq (1+\varepsilon)\frac{|V|}{k}$), such that an objective function defined on the hyperedges is minimized. In this work, we consider the connectivity metric $\sum_{e \in E}(\lambda(e)-1)\cdot\omega(e)$ where $\lambda(e) := |\{V_i \mid e \cap V_i \neq \emptyset\}|$ denotes the number of different blocks connected by hyperedge $e \in E$ and $\omega(e)$ denotes its weight. Often balanced partitioning is used as an acceleration technique for other applications, such as quantum circuit simulation [28], sharding distributed databases [14, 32], load balancing [12], route planning [16, 29], or boosting cache utilization in a search engine backend [7].

There is a substantial amount of literature, which is why we refer to survey articles [4, 9, 45, 50] for a summary. Most of the work focuses on heuristics, with the multilevel paradigm emerging as the most successful approach [2, 12, 18, 26, 34, 40, 49, 50]. Most partitioners use move-based heuristics such as label propagation [48] or variations of the Kernighan-Lin [37] or Fiduccia-Mattheyses [20] algorithms for local search. These heuristics move nodes greedily to different blocks according to their reduction in the objective function and are known to get stuck in local minima [38].

In this situation, maximum flows are an excellent tool as they correspond to (unbalanced) minimum cuts, thus offering a more global view than local move-based routines. Due to their complexity [55], they were long overlooked for partitioning, but have since enjoyed wide-spread adoption [5, 16, 29, 39, 49, 55] in many different algorithmic contexts.

Contribution. In this paper, we parallelize *flow-based refinement*, a powerful technique that is the last missing component in a series of works [26, 27] on parallelizing the state-of-the-art multilevel hypergraph partitioner KaHyPar [50]. Flow-based refinement operates on bipartitions, or on two blocks at a time if used for $k > 2$. Scheduling independent block pairs gives some trivial parallelism. One contribution we make is to improve the parallelism in the scheduler by relaxing certain constraints and showing how to deal with the resulting race conditions. For small k this is still insufficiently parallel, which is why we also parallelize the refinement on two blocks. We adapt an existing parallel flow algorithm to handle the incremental flow problems of the FlowCutter refinement algorithm [23, 24, 29]. Additionally, we engineer an efficient implementation, proposing several optimizations that reduce running time in practice, and fix a so far undocumented bug in the parallel flow algorithm. The result is a parallel partitioner that achieves the same solution quality as the highest quality sequential framework (KaHyPar), but in a fast parallel code. Using 10 threads, our code is an order of magnitude faster than sequential KaHyPar with flow-based refinement.

Outline. The paper is organized as follows. In Section 2 we introduce notation, terminology, and some algorithmic preliminaries. Section 3 briefly deals with related work. More details on additional related work are given in the main sections 5–8, closer to where particular parts are needed. In Section 4, we give an overview of the different components in the framework and how they interact. We complement the algorithmic discussion with extensive experiments in Section 9, before concluding in Section 10.

2 Preliminaries

Hypergraphs. A *weighted hypergraph* $H = (V, E, c, \omega)$ is defined as a set of vertices V and a set of hyperedges/nets E with vertex weights $c : V \to \mathbb{R}_{>0}$ and net weights $\omega : E \to \mathbb{R}_{>0}$, where each net e is a subset of the vertex set V. The vertices of a net are called its *pins*. We extend c and ω to sets in the natural way, 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 $\deg(v) := |\mathrm{I}(v)|$. The *size* $|e|$ of a net e is the number of its pins. We call two nets e_i and e_j *identical* if $e_i = e_j$. Given a subset $V' \subset V$, the *subhypergraph* $H_{V'}$ is defined as $H_{V'} := (V', \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\})$.

Balanced Hypergraph Partitioning. A *k-way partition* of a hypergraph H is a function $\Pi : V \to \{1, \ldots, k\}$. The blocks $V_i := \Pi^{-1}(i)$ of Π are the inverse images. We call Π *ε-balanced* if each block V_i satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1 + \varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some parameter $\varepsilon \in (0,1)$. For each net e, $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e)$ of a net e is $\lambda(e) := |\Lambda(e)|$. A net is called a *cut net* if $\lambda(e) > 1$. A node u that is incident to at least one cut net is called *boundary node*. The number of pins of a net e in block V_i is denoted by $\Phi(e, V_i) := |e \cap V_i|$. The *quotient graph* $\mathcal{Q} := (\Pi, E_\Pi := \{(V_i, V_j) \mid \exists e \in E : \{V_i, V_j\} \subseteq \Lambda(e)\})$ contains an edge between each pair of adjacent blocks of a k-way partition Π. Given parameters ε and k, and a hypergraph H, the *balanced hypergraph partitioning problem* is to find an ε-balanced k-way partition Π that minimizes an objective function defined on the hyperedges. In this work, we minimize the *connectivity metric* $(\lambda - 1)(\Pi) := \sum_{e \in E}(\lambda(e) - 1)\, \omega(e)$.

Flows. A flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ is a directed graph with a dedicated source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$ in which each edge $e \in \mathcal{E}$ has capacity $c(e) \geq 0$. An (s,t)-flow is a function $f : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ that satisfies the *capacity constraint* $\forall u, v \in \mathcal{V} : f(u,v) \leq c(u,v)$, the *skew symmetry constraint* $\forall u, v \in \mathcal{V} : f(u,v) = -f(v,u)$ and the *flow conservation constraint* $\forall u \in \mathcal{V} \setminus \{s,t\} : \sum_{v \in \mathcal{V}} f(u,v) = 0$. The value of a flow $|f| := \sum_{v \in \mathcal{V}} f(s,v) = \sum_{v \in \mathcal{V}} f(v,t)$ is defined as the total amout of flow transferred from s to t. An (s,t)-flow f is a maximum (s,t)-flow if there exists no other (s,t)-flow f' with $|f| < |f'|$. The *residual capacity* is defined as $r_f(e) = c(e) - f(e)$. An edge e is *saturated* if $r_f(e) = 0$. $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f, r_f)$ with $\mathcal{E}_f := \{(u,v) \in \mathcal{V} \times \mathcal{V} \mid r_f(u,v) > 0\}$ is the *residual network*. The max-flow min-cut theorem states that the value $|f|$ of a maximum (s,t)-flow equals the weight of a minimum cut that separates s and t [21]. This is also called a *minimum* (s,t)*-cut*. The source-side cut can be computed by exploring the nodes reachable from the source via residual edges (for example via BFS), and analogously the sink-side cut from the sink.

Push-Relabel Algorithm. The *push-relabel* [22] maximum flow algorithm stores a distance label $d(u)$ and an excess value $\text{exc}(u) := \sum_{v \in \mathcal{V}} f(v,u)$ for each node. It maintains a *preflow* [36] which is a flow where the conservation constraint is replaced by $\text{exc}(u) \geq 0$. A valid distance labeling is defined by the conditions $\forall (u,v) \in \mathcal{E}_f d(u) \leq d(v) + 1$, $d(s) = |\mathcal{V}|$, $d(t) = 0$. A node $u \in \mathcal{V}$ is *active* if $\text{exc}(u) > 0$. An edge $(u,v) \in \mathcal{E}$ is *admissible* if $r_f(u,v) > 0$ and $d(u) = d(v) + 1$. A *push*(u,v) operation sends $\delta = \min(\text{exc}(u), r_f(u,v))$ flow units over (u,v). It is applicable if u is active and (u,v) is admissible. A *relabel*(u) operation updates the distance label of u to $\min(\{d(v) + 1 \mid r_f(u,v) > 0\})$, which is applicable if u is active, and has no admissible edges. The distance labels are initialized to $\forall u \in \mathcal{V} \setminus \{s\} : d(u) = 0$ and $d(s) = |V|$ and all source edges are saturated. Efficient variants use the *discharge* routine, which repeatedly scans the edges of an active node until its excess is zero. All admissible edges are pushed and at the end of a scan, the node is relabeled. Discharging active nodes in FIFO order results in an $\mathcal{O}(|\mathcal{V}|^3)$ time algorithm. The *global relabeling* heuristic [13] frequently assigns exact distance labels by performing a reverse BFS from the sink, to reduce relabel work in practice. Note that preflows already induce minimum sink-side cuts, so if only a minimum cut is required, the algorithm can already stop once no active nodes with distance label $< n$ exist.

Flows on Hypergraphs. The *Lawler expansion* [41] of a hypergraph $H = (V, E, c, \omega)$ is a graph consisting of V and two nodes $e_{\text{in}}, e_{\text{out}}$ for each $e \in E$, with directed edges $\forall u \in V, e \in \mathrm{I}(u) : (u, e_{\text{in}}), (e_{\text{out}}, u)$ with infinite capacity and *bridge edges* $\forall e \in E : (e_{\text{in}}, e_{\text{out}})$ with capacity $\omega(e)$. A minimum (s,t)-cut in the Lawler expansion directly corresponds to one in the hypergraph (since only bridging edges have finite capacity).

Algorithm 1 Parallel Flow-Based Refinement.

```
Input: Hypergraph $H = (V, E, c, \omega)$, $k$-way partition $\Pi$ of $H$
1 $\mathcal{Q} \leftarrow$ buildQuotientGraph($H, \Pi$)                          // Section 5
2 while $\exists$ active $(V_i, V_j) \in \mathcal{Q}$ do in parallel                // Section 5
3     $B := B_i \cup B_j \leftarrow$ constructRegion($H, V_i, V_j$)    // $B_i \subseteq V_i, B_j \subseteq V_j$, Section 6
4     $(\mathcal{N}, s, t) \leftarrow$ constructFlowNetwork($H, B$)                // Section 6
5     $(M, \Delta_{\text{exp}}) \leftarrow$ FlowCutterRefinement($\mathcal{N}, s, t$)        // Section 7 and 8
6     if $\Delta_{exp} \geq 0$                                       // potential improvement
7         $\Delta_{\lambda-1} \leftarrow$ applyMoves($H, \Pi, M$)                    // Section 5
8         if $\Delta_{\lambda-1} > 0$ mark $V_i$ and $V_j$ as active          // found improvement
9         else if $\Delta_{\lambda-1} < 0$ revertMoves($H, \Pi, M$)            // no improvement
```

3 Related Work

The most well-known sequential algorithms are PaToH [12], hMetis [34, 35], and KaHyPar [1, 23, 51]. Notable parallel algorithms are Parkway [53] and Zoltan [18] for distributed memory, as well as BiPart [42] and Mt-KaHyPar [26, 27] for shared memory.

All of these follow the multilevel paradigm that proceeds in three phases: First, the hypergraph is *coarsened* to obtain a hierarchy of successively smaller and structurally similar hypergraphs by *contracting* pairs or clusters of vertices. Once the coarsest hypergraph is small enough, an *initial partition* into k blocks is computed. Subsequently, the contractions are reverted level-by-level, and, on each level, *local search* heuristics are used to improve the partition from the previous level (*refinement phase*).

Sanders and Schulz [49] propose an algorithm to improve the edge cut of bipartitions with flow-based refinement. Their general idea is to grow a size-constrained region around the cut edges of a bipartition. Afterwards, they compute a minimum (s, t)-cut in the subgraph induced by the region and apply it to the original graph, if it satisfies the balance constraint. They extended their algorithm to k-way partitions by scheduling it on pairs of adjacent blocks. Heuer et al. [30] integrated this approach into their hypergraph partitioner KaHyPar. This was improved by Gottesbüren et al. [23] by replacing the bipartitioning routine with FlowCutter [24, 29]. Flow-based refinement substantially improved the solution quality (cut and connectivity metric) of the partitions produced by KaHyPar, making it the method of choice for high-quality hypergraph partitioning [50]. We explain the flow-based refinement routine of KaHyPar in more detail in the main part.

4 Framework Overview

We first give an overview of how the different framework components interact, before providing descriptions in their respective sections. For this we follow the high level structure shown in Algorithm 1. We start with a parallel scheduling scheme of block pairs based on the quotient graph in Section 5, see line 1 and 2. For each block pair, we extract a subhypergraph constructed around the boundary nodes of the blocks, which yields a flow network, see line 3 and 4 and Section 6. On each network we run FlowCutter (line 5), whose partition we convert into a set of moves M and an expected connectivity reduction Δ_{exp}. FlowCutter and its parallelization are discussed in Section 7 and 8 respectively. If FlowCutter claims an improvement, i.e., $\Delta_{\text{exp}} \geq 0$, we apply the moves to the global partition and compute the exact reduction $\Delta_{\lambda-1}$, based on which we either mark the blocks for further refinement, or revert the moves, see line 8 and 9. We distinguish between expected Δ_{exp} and actual improvement $\Delta_{\lambda-1}$, due to concurrency conflicts that arise in the scheduler, which is described again in Section 5.

5 Parallel Active Block Scheduling

Sanders and Schulz [49] propose the active block scheduling algorithm to apply their flow-based refinement algorithm for bipartitions on k-way partitions. Their algorithm proceeds in rounds. In each round, it schedules all pairs of adjacent blocks where at least one is marked as *active*. Initially, all blocks are marked as active. If a search on two blocks improves the edge cut, both are marked as active for the next round.

Parallelization. A simple scheme would be to schedule block pairs that form a maximum matching in the quotient graph $\mathcal{Q}$ in parallel. This allows searches to operate in independent regions of the hypergraph and thus avoids conflicts between different block pairs. However, this scheme restricts the available parallelism to at most $\frac{k}{2}$ threads. Thus, we do not enforce any constraints on the block pairs processed concurrently, e.g., there can be multiple threads running on the same block and they can also share some of their nodes as illustrated in Figure 1 (right). We use $\min(t, \tau \cdot k)$ threads to process the active block pairs in parallel, where t is the number of available threads in the system. The parameter τ controls the available parallelism in the scheduler. With higher values of τ, more block pairs are scheduled in parallel. This can lead to interference between searches that operate on overlapping regions. Lower values for τ can reduce these conflicts but put more emphasis on good parallelization of 2-way refinement to achieve good speedups. In practice, we choose $\tau = 1$.

Our parallel active block scheduling algorithm uses one concurrent FIFO queue A to schedule active block pairs. Each block pair is associated with a round and each round uses an array of size k to mark blocks that become active in the next round. If a search finds an improvement on two blocks (V_i, V_j) and V_i or V_j becomes active, we push all adjacent blocks into A if they are not contained yet (marked using an atomic test-and-set instruction). If either V_i or V_j is already active, we insert (V_i, V_j) into A, if it is not already contained in A. Thus, active block pairs of different rounds are stored interleaved in A and the end of a round does not induce a synchronization point as in the original algorithm [49]. For processing in the first round, we sort active block pairs in descending order of improvement they contributed on previous levels, with ties broken by larger cut size. A round ends when all of its block pairs have been processed and all prior rounds have ended. If the relative improvement at the end of a round is less than 0.1%, we immediately terminate the algorithm. In Appendix A, we describe how to construct and maintain the cut hyperedges between block pairs that induce the quotient graph and are used for the network construction.

Apply Moves. We integrate flow-based refinement into Mt-KaHyPar [26, 27], which provides data structures to concurrently access and modify the partition Π, block weights $c(V_i)$, connectivity sets $\Lambda(e)$ and pin counts $\Phi(e, V_i)$ of each $e \in E$ and V_i. When applying a move sequence M to the global partition Π (each move $m := (u, V_i, V_j) \in M$ moves a node u from its current block V_i to V_j), there are three conflict types that can occur: balance constraint violations, $\Delta_{\lambda-1} \neq \Delta_{\text{exp}}$, i.e., the expected does not match the actual connectivity reduction, and nodes may no longer be in the block expected by M. These conflicts arise, because concurrently scheduled block pairs are not independent, which causes data races on the partition assignment Π, pin counts $\Phi(e, V_i)$ and connectivity sets $\Lambda(e)$. These are concurrently

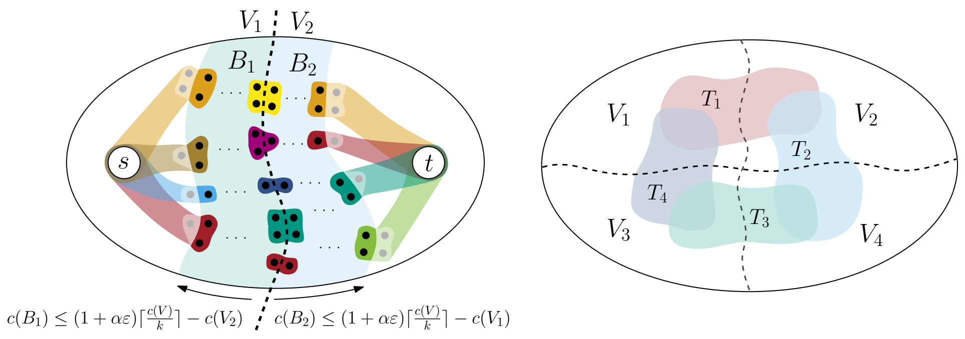
V_1
V_2
B_1
B_2
s
t
$c(B_1) \leq (1+\alpha\varepsilon)\lceil\frac{c(V)}{k}\rceil - c(V_2)$
$c(B_2) \leq (1+\alpha\varepsilon)\lceil\frac{c(V)}{k}\rceil - c(V_1)$
V_1
V_2
V_3
V_4
T_1
T_2
T_3
T_4

Figure 1 Illustrates the flow network construction algorithm (left) and an how we schedule block pairs of the quotient graph $\mathcal{Q}$ in parallel (right, T_i denotes the search region of thread i).

read by the network construction and modified when moves are applied. Updates after the construction are not observed, and thus the state at the time a refinement finishes may differ from the expected state. Since the running time to apply moves is negligible compared to solving flow problems (see Figure 9 in Section 9), we can afford to use a lock so that only one thread applies moves at a time to address these conflicts. We remove all nodes from M that are not in their expected block. Afterwards, we compute the block weights if all remaining moves were applied. If balanced, we perform the moves, during which we compute $\Delta_{\lambda-1}$. For $m = (u, V_i, V_j)$ and $e \in I(u)$, we add $\omega(e)$ to $\Delta_{\lambda-1}$ if $\Phi(e, V_i)$ decreases to zero and $-\omega(e)$ if $\Phi(e, V_j)$ increases to one (connectivity metric is $\sum_{e\in E}(\lambda(e) - 1)\,\omega(e)$). If $\Delta_{\lambda-1} < 0$, we revert all moves.

Implementation Details. KaHyPar [30] established pruning rules to skip unpromising flow computations that we use as well: skip if cut is small or no improvement found on previous levels. We additionally introduce a time limit to abort long-running flow computations.

6 Network Construction

To improve the cut of a bipartition $\Pi = \{V_1, V_2\}$, we grow a size-constrained region B around the cut hyperedges of Π. We then contract all nodes in $V_1 \setminus B$ to the source s and $V_2 \setminus B$ to the sink t [24, 49] as illustrated in Figure 1 (left) and obtain a coarser hypergraph $\mathcal{H}$. We implemented two parallel algorithms to construct $\mathcal{H}$, which are preferable in different situations. These are described in more detail in Appendix B. The flow network $\mathcal{N}$ is then given by the Lawler expansion of $\mathcal{H}$ (see Section 2). Note that reducing the hyperedge cut of a bipartition induced by two adjacent blocks of a k-way partition Π_k optimizes the connectivity metric of Π_k [30].

Sanders and Schulz [49] grow a region $B := B_1 \cup B_2$ with $B_1 \subseteq V_1$ and $B_2 \subseteq V_2$ around the cut hyperedges of Π via two breadth-first-searches (BFS) as illustrated in Figure 1 (left). The first BFS is initialized with all boundary nodes of block V_1 and continues to add nodes to B_1 as long as $c(B_1) \leq (1+\alpha\varepsilon)\lceil\frac{c(V_1)+c(V_2)}{2}\rceil - c(V_2)$, where α is an input parameter. The second BFS that constructs B_2 proceeds analogously. For $\alpha = 1$, each flow computation yields a balanced bipartition with a possibly smaller cut in the original hypergraph, since only nodes of B can move to the opposite block ($c(B_1) + c(V_2) \leq (1+\varepsilon)\lceil\frac{c(V_1)+c(V_2)}{2}\rceil$ and vice versa). Larger values for α lead to larger flow problems with potentially smaller minimum cuts, but also increase the likelihood of violating the balance constraint. However, this is not a problem since the flow-based refinement routine guarantees balance through incremental minimum

Algorithm 2 FlowCutter Core.

```
1  $S \leftarrow \{s\}, T \leftarrow \{t\}$
2  while true do
3      augment flow to maximality regarding $S, T$
4      derive source- and sink-side cut $S_r, T_r \subset \mathcal{V}$
5      if $(S_r, \mathcal{V} \setminus S_r)$ or $(\mathcal{V} \setminus T_r, T_r)$ balanced
6          return balanced partition
7      if $c(S_r) \leq c(T_r)$
8          $S \leftarrow S_r \cup$ selectPiercingNode()
9      else
10         $T \leftarrow T_r \cup$ selectPiercingNode()
```

cut computations (see Section 7). In practice, we use $\alpha = 16$ (also used in KaHyPar [23, 30]). We additionally restrict the distance of each node $v \in B$ to the cut hyperedges to be smaller than or equal to a parameter δ $(= 2)$. We observed that it is unlikely that a node *far* way from the cut is moved to the opposite block by the flow-based refinement.

7 Flow-Based Refinement

In this section we discuss the flow-based refinement on a bipartition. We introduce the aforementioned FlowCutter algorithm [29, 55]. It is parallelized by plugging in a parallel maximum flow algorithm, which we discuss in the next section. To speed up convergence and make parallelism worthwhile, we propose an optimization named *bulk piercing*.

Core Algorithm. FlowCutter solves a sequence of incremental maximum flow problems until a balanced bipartition is found. Algorithm 2 shows pseudocode for the approach. In each iteration, first the previous flow (initially zero) is augmented to a maximum flow regarding the current source set S and sink set T. Subsequently, the node sets $S_r, T_r \subset \mathcal{V}$ of the source- and sink-side cuts are derived. This is done via residual (parallel) BFS (forward from S for S_r, backward from T for T_r). The node sets induce two bipartitions $(S_r, \mathcal{V} \setminus S_r)$ and $(\mathcal{V} \setminus T_r, T_r)$. If neither is balanced, all nodes on the side with smaller weight are transformed to a source (if $c(S_r) \leq c(T_r)$) or a sink otherwise. To find a different cut in the next iteration, one additional node is added, called *piercing node*. Thus, the bipartitions contributed by the currently smaller side will be more balanced in future iterations. Since the smaller side is grown, this process will converge to a balanced partition.

Piercing. For our purpose, there are two important piercing node selection heuristics: *avoid augmenting paths* [29, 55] and *distance from cut* [23]. Whenever possible, a node that is not reachable from the source or sink should be picked, i.e., $v \in \mathcal{V} \setminus (S_r \cup T_r)$. Such nodes do not increase the weight of the cut, while improving balance. As a secondary criterion, larger distances from the original cut are preferred, to reconstruct parts of it.

Most Balanced Cut. Once the partition is balanced, we continue to pierce as long as the cut does not increase. This is repeated with different random choices since it is fast (no flow augmentation). More balance gives other refinement algorithms more leeway for improvement. An equivalent heuristic was already employed in previous flow-based refinement [47, 49].

Bulk Piercing. The complexity of FlowCutter is $O(\zeta m)$, where ζ is the final cut weight, and $m = |\mathcal{E}|$. This bound stems from a pessimistic implementation that augments one flow unit in $O(m)$ work [29, 55]. For refinement, the performance is much better in practice, as the first cut is often close to the final cut. Only few augmenting iterations are needed and much less than $O(m)$ work is spent per flow unit [24], with most work spent on the initial flow.

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

If a goal is not met, its remainder is added to next iteration's goal. We track the average weight added per node and from this estimate the number of piercing nodes needed to match the current goal. To boost measurement accuracy, we pierce one node for the first few rounds. The sides have distinct measurements and goals, so that we do not pierce too aggressively when the smaller side flips. This scheme (with $\beta = 0.55$) reduces running time on our largest instances from beyond two hours (time limit) to less than 10 minutes, while not incurring any quality penalties on either small or large instances, as shown in our technical report [25].

8 Parallel Maximum Flow Algorithm

Maximum flow algorithms are notoriously difficult to parallelize efficiently [6, 10, 33, 52]. The synchronous push-relabel approach of Baumstark et al. [10] is a recent algorithm that sticks closely to sequential FIFO and thus shows good results. We first outline their algorithm, then describe a so far undocumented bug followed by our fix, and conclude with implementation details and intricacies of using FlowCutter with preflows. Note that a maximum preflow already yields a minimum cut, which suffices for our purpose.

Synchronous Parallel Push-Relabel. The algorithm proceeds in rounds in which all active nodes are discharged in parallel. The flow is updated globally, the nodes are relabeled locally and the excess differences are aggregated in a second array using atomic instructions. After all nodes have been discharged, the distance labels d are updated to the local labels d' and the excess deltas are applied. The discharging operations thus use the labels and excesses from the previous round. This is repeated until there are no nodes with $\mathrm{exc}(v) > 0$ and $d(v) < n$ left. To avoid concurrently pushing flow on residual arcs in both directions (race condition on flow values), a deterministic winning criterion on the old distance labels is used to determine which direction to push, if both nodes are active. If an arc cannot be pushed due to this, the discharge terminates after the current scan, as the node may not be relabeled in this round. The rounds are interleaved with global relabeling [13], after linear push and relabel work, using parallel reverse BFS.

A Bug in the Synchronous Algorithm. The parallel discharge routine does not protect against push-relabel conflicts [33] as illustrated in Figure 2. In particular the winning criterion does not help. A node u may be relabeled too high if it is concurrently pushed to through a residual arc (v, u) with $d'(v) = d(u) + 1$. The arc (u, v) may not be observed as residual yet, and thus u may set its new label $d'(u) > d'(v) + 1$, violating label correctness. The bug

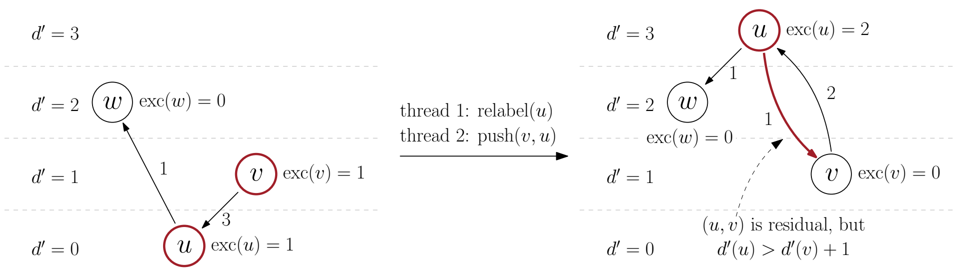

Figure 2 Illustrates a push-relabel conflict in the parallel discharge routine (adapted from Ref. [33]). The numbers on the arcs denote their residual capacities.

becomes noticeable when the algorithm terminates prematurely with incorrect distances. Our fix is to collect mislabeled excess nodes during global relabeling. When the algorithm would terminate, we run global relabeling, and restart the main loop if new active nodes are found. The additional work is already accounted for, because we need to extract the sink-side cut anyways.

Restricting Capacities. Recall that only bridge edges $(e_{\text{in}}, e_{\text{out}})$ have finite capacity $(\omega(e))$ in the Lawler network. Since $(e_{\text{in}}, e_{\text{out}})$ is the only outgoing edge of e_{in} with non-zero capacity, the flow (but not preflow) on edges (u, e_{in}) is also bounded by $\omega(e)$. Adding these capacities during the preflow stage is a trivial optimization, but it reduces running time for one flow computation on our largest instance from over two hours to 14 seconds, when using 16 cores. It also boosts the available parallel work, since hypernodes are not immediately relieved of all their excess. Without this optimization the minimum cut contains only bridge edges, but now may contain edges (u, e_{in}). This matters when tracking cut hyperedges (for collecting piercing candidates), which are detected by checking if e_{in} and e_{out} are on different sides. Therefore, we do not check the capacity and visit e_{in} nodes during forward residual BFSs.

Avoid Pushing Flow Back. Once the correct flow value is found, the algorithm could terminate in theory. This is often achieved in very few discharging rounds ($< 1\%$). Furthermore, we observed that the number of active nodes follows a power law distribution. At this point flow is only pushed back to the source. We terminate once all nodes with $\text{exc}(u) > 0$ have $d(u) \geq n$, which is most often detected by global relabeling. Due to little work per round, it takes many rounds to trigger. We perform additional relabeling, if the flow value has not changed for some rounds (500), and only few active nodes (< 1500) were available in each.

Source-Side Cut. A maximum preflow only yields a sink-side cut via the reverse residual BFS, but we also need the source-side cut. We can run flow decomposition [13] to push excess back to the source, to obtain an actual flow. However, flow decomposition is difficult to parallelize [10]. Instead, we initialize the forward residual BFS with all non-sink excess nodes. This finds the reverse paths that carry flow from the source to the excess nodes, which is what we need.

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

Algorithm 3 Parallel Multilevel Hypergraph Partitioning (Mt-KaHyPar-D-F).

```
   Input: Hypergraph $H = (V, E)$, number of blocks $k$
   Output: $k$-way partition $\Pi$ of $H$
 1 $H_1 \leftarrow H, \mathcal{H} \leftarrow \langle H_1 \rangle, i \leftarrow 1$
 2 while $|V_i|$ is not small enough do                         // Coarsening Phase
 3     $\mathcal{C} \leftarrow$ compute node clustering  // Uses the heavy-edge rating function [1, 12, 34]
 4     $H_{i+1} \leftarrow H_i$.contract($\mathcal{C}$), $\mathcal{H} \leftarrow \mathcal{H} \cup \langle H_{i+1} \rangle, i \leftarrow i + 1$
 5 $\Pi \leftarrow$ initialPartition($H_i, k$)
 6 for $l = i$ to 1 do                                          // Uncoarsening Phase
 7     $\Pi \leftarrow$ project $\Pi$ onto $H_l$
 8     while improvement relevant do
 9         $\Pi \leftarrow$ labelPropagationRefinement($H_l, \Pi$)
10         $\Pi \leftarrow$ fmLocalSearch($H_l, \Pi$)
           // Extends Mt-KaHyPar-D with flow-based refinement
11         $\Pi \leftarrow$ flowBasedRefinement($H_l, \Pi$)
12 return $\Pi$
```

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

9 Experiments

We implemented the flow-based refinement routine in the shared-memory hypergraph partitioner Mt-KaHyPar[1], which is implemented in `C++17`, parallelized using the TBB library [46], and compiled using g++9.2 with the flags `-O3 -mtune=native -march=native`. Mt-KaHyPar provides two partitioners: Mt-KaHyPar-D [26] (**D**efault setting) opts for the traditional $\mathcal{O}(\log n)$ level approach by contracting a vertex clustering on each level and Mt-KaHyPar-Q [27] (**Q**uality setting) implements a parallel version of the n-level scheme [1, 44, 51] that (un)contracts only a single vertex on each level. We refer to the corresponding versions that use flow-based refinement as Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F (-**F**lows, integration is described in the next paragraph). For parallel partitioners we add a suffix to their name to indicate the number of threads used, e.g. Mt-KaHyPar-Q-F 64 for 64 threads. We omit the suffix for sequential partitioners. Note that we performed extensive parameter tuning experiments (e.g., τ, β, δ and the effects of bulk piercing) which we present only in the technical report [25] due to space constraints.

The Multilevel Partitioning Algorithm. The high-level pseudocode of Mt-KaHyPar-D is shown in Algorithm 3. Mt-KaHyPar-D uses a clustering-based coarsening algorithm and parallel multilevel recursive bipartitioning with work-stealing to compute an initial

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

k-way partition of the coarsest hypergraph [26, 27]. In each refinement step, we first run label propagation refinement [48] followed by a highly-localized version of the FM algorithm [1, 2, 20, 49] (see Line 9 and 10). We run our flow-based refinement as the third component. We run all refinement algorithms on each level multiple times in combination and stop if the relative improvement is less than 0.25%.

Instead of contracting a node clustering, Mt-KaHyPar-Q contracts only a single vertex on each level. In the uncoarsening phase, it assembles independent contractions in a batch and uncontracts them in parallel. This induces a hierarchy with $\mathcal{O}(|V|)$ levels and refinement steps, which would incur too much overhead when we use flow-based refinement. Thus, we use approximately $\mathcal{O}(\log n)$ synchronization points similar to Mt-KaHyPar-D and perform FM local search followed by our flow-based refinement. We do not run label propagation here since there were no quality benefits.

Setup. For comparison with sequential partitioners, we use the established benchmark set of Heuer and Schlag [31] (referred to as set A, 488 hypergraphs). For these experiments, we use $k \in \{2, 4, 8, 16, 32, 64, 128\}$, $\varepsilon = 0.03$, ten different seeds and a time limit of eight hours. The experiments are done on a cluster of Intel Xeon Gold 6230 processors (2 sockets with 20 cores each) running at 2.1 GHz with 96GB RAM (machine A). To measure speedups and to compare our implementation with other parallel partitioners, we use a benchmark set composed of 94 large hypergraphs (referred to as set B) that was initially assembled to evaluate Mt-KaHyPar-D [26]. On set B, we evaluate $k \in \{2, 8, 16, 64\}, \varepsilon = 0.03$ and use three seeds each with a time limit of two hours. These experiments are run on an AMD EPYC Rome 7702P (one socket with 64 cores) running at 2.0–3.35 GHz with 1024GB RAM (machine B). The parameter space on set B is restricted, since we only have access to one machine of type B. We describe the sources and properties of the instances in Appendix C[2].

Methodology. Each partitioner optimizes the connectivity metric, which we also refer to as the quality of a partition. For each instance (hypergraph and k), we aggregate running times using the arithmetic mean over all seeds. To further aggregate over multiple instances, we use the geometric mean for absolute running times and self-relative speedups. For runs that exceeded the time limit, we use the time limit itself in the aggregates. In plots, we mark these instances with ⏲ if *all* runs of that algorithm timed out.

To compare the solution quality of different algorithms, we use *performance profiles* [19]. Let $\mathcal{A}$ be the set of algorithms we want to compare, $\mathcal{I}$ the set of instances, and $q_A(I)$ the quality of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm A, we plot the fraction of instances (y-axis) for which $q_A(I) \le \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)$, where τ is on the x-axis. Achieving higher fractions at lower τ-values is considered better. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm performs best. The ⏲ tick indicates the fraction of instances for which *all* runs of that algorithm timed out.

Medium-Sized Instances. On set A, we compare Mt-KaHyPar with KaHyPar-HFC [24, 30] (similar components as Mt-KaHyPar-Q-F) which is currently the best sequential partitioner in terms of solution quality [50], the recursive bipartitioning version (hMetis-R) of hMetis 2.0 [34], as well as the default (PaToH-D) and quality preset (PaToH-Q) of PaToH 3.3 [12]. All configurations of Mt-KaHyPar use 10 threads.

[2] Benchmark sets and results are available from `https://algo2.iti.kit.edu/heuer/sea22`

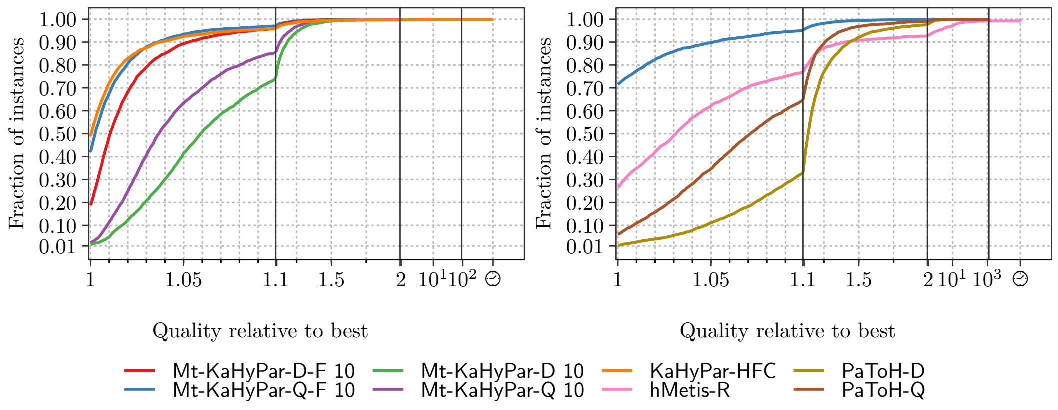

Figure 3 Solution quality of Mt-KaHyPar-Q-F compared with different partitioners on set A.

Figure 3 compares the solution quality of Mt-KaHyPar with different partitioners on set A (see Figure 5 (left) for running times). In an individual comparison, Mt-KaHyPar-Q-F finds better partitions than PaToH-D, PaToH-Q, Mt-KaHyPar-D, Mt-KaHyPar-Q, Mt-KaHyPar-D-F, hMetis-R and KaHyPar-HFC on 94.7%, 87.7%, 97.3%, 95.7%, 73.5%, 74.5% and 51.3% of the instances, respectively.

The median improvement of Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F compared to the configurations that use no flow-based refinement is 4.2% and 2.7% while only incuring a slowdown by a factor of 3.1 (gmean time 2.73s vs 0.89s) and 1.7 (5.08s vs 2.99s). To put this into perspective, the quality preset of PaToH (PaToH-Q) improves the default preset (PaToH-D) by 5.3% in the median and is a factor of 5 slower (5.86s vs 1.17s). The median improvement of hMetis-R compared to PaToH-Q is 2.6% while it is a factor of 15.9 slower (93.21s vs 5.86s). The solutions produced by Mt-KaHyPar-Q-F are 3% better than those of hMetis-R in the median and it has a similar running time as PaToH-Q (5.08s vs 5.86s). If we compare our two partitioners that use flow-based refinement, we can see that Mt-KaHyPar-Q-F gives only minor quality improvements over Mt-KaHyPar-D-F (median improvement is 0.6% whereas without flow-based refinement it is 1.9%). This demonstrates the effectiveness of flow-based refinement. The solution quality of Mt-KaHyPar-Q-F and KaHyPar-HFC are on par, while Mt-KaHyPar-Q-F is an order of magnitude faster with 10 threads (5.08s vs 48.98s). In conclusion, we achieved the solution quality of the currently hiqhest quality sequential partitioner in a fast parallel code.

Large Instances. On set B, we compare Mt-KaHyPar with the parallel algorithms Zoltan 3.83 [18] and BiPart [42], as well as PaToH-D (which is fast enough for set B as opposed to other sequential algorithms). All parallel algorithms use 64 threads.

Figure 4 compares the solution quality of Mt-KaHyPar with different partitioners on set B (see Figure 5 (right) for running times). The quality of the partitions produced by Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F are comparable while Mt-KaHyPar-D-F is a factor of 1.9 faster (gmean time 30.38s vs 58.24s). Therefore, we focus on Mt-KaHyPar-D-F in this evaluation. In an individual comparison, Mt-KaHyPar-D-F finds better partitions than BiPart, Zoltan, PaToH-D, Mt-KaHyPar-D, Mt-KaHyPar-Q and Mt-KaHyPar-Q-F on 97.3%, 96%, 92%, 91%, 85.1% and 47.3% of the instances, respectively.

The median improvement of Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F compared to the configurations that use no flow-based refinement is 5.2% and 3.4% while they are slower by a factor of 6.6 (30.38s vs 4.63s) and 1.9 (58.24s vs 29.99s). Both the improvements and

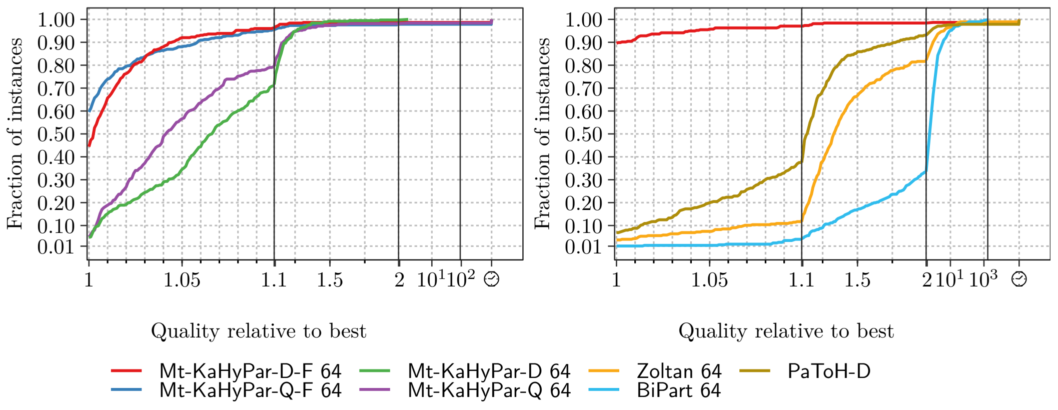

Figure 4 Solution quality of Mt-KaHyPar-D-F compared with different partitioners on set B.

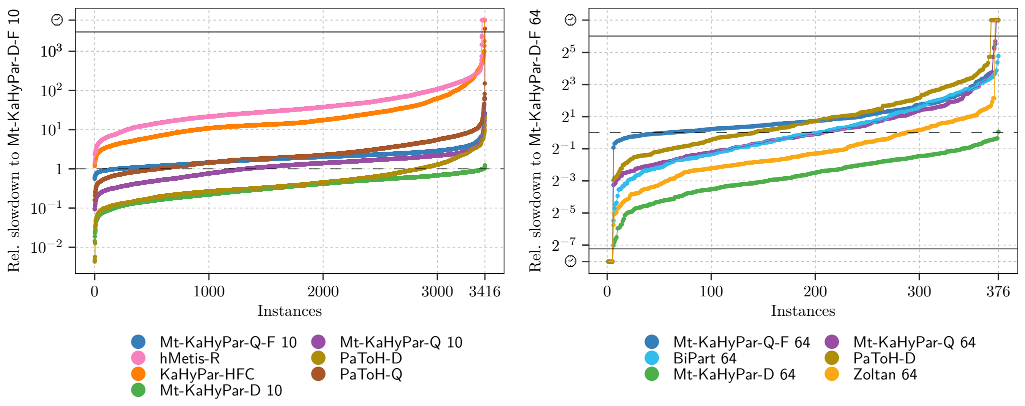

Figure 5 Running times relative to Mt-KaHyPar-D-F on set A (left) and B (right). The ⏲ axis markers represent timeouts for the baseline Mt-KaHyPar-D-F (at the bottom) or the compared algorithm (at the top).

slowdowns are more pronounced here than on set A. The slowdowns are expected since the size of the flow problems scales linearly with instance sizes, while the complexity of the flow-based refinement routine does not. Mt-KaHyPar-D-F (30.38s) is slower than Zoltan (12.6s) and BiPart (29.19s), but faster than PaToH-D (50.3s). However, Mt-KaHyPar-D-F computes partitions that are 33% better than Zoltan's and twice as good as BiPart's in the median.

Scalability. Figure 6 shows self-relative speedups with varying number of threads $t \in \{4, 16, 64\}$. In the plot, we represent the speedup of each instance as a point and the centered rolling geometric mean with a window size of 25 as a line. The x-axis shows the sequential running time of Mt-KaHyPar-D-F for each instance.

We found that any one of the common (hyper)graph metrics (such as number of pins) is not well correlated with speedups (or running times for that matter), since the running time depends on a variety of different factors (metrics and events that trigger repetitions). Fitting suitable parameters for a combination of the metrics seems much more complicated than plotting against sequential running time, which is often nicely correlated with speedups. Furthermore, the longer an algorithm runs sequentially, the more important is an efficient parallelization to achieve reasonable running times.

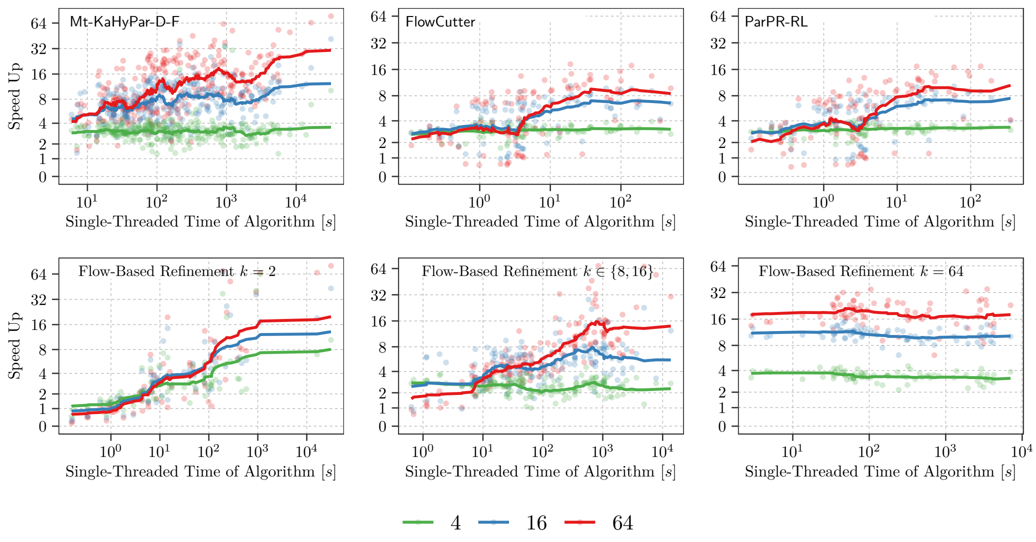

Figure 6 Speedups of Mt-KaHyPar-D-F and the flow-based refinement routine (for different values of k) as well as of the FlowCutter and parallel flow algorithm (ParPR-RL).

To assess FlowCutter and parallel push-relabel (referred to as ParPR-RL), we extract flow networks from bipartitions of the instances in set B[3]. The results are shown in the top-middle and -right plot. With 4 threads, we observe near-perfect speedups throughout, with fairly small variance. For $t = 16, 64$, the parallelization overheads are only outweighed for longer running instances, with more threads becoming worthwhile at about 10 seconds of sequential time. Unfortunately, we even experience some minor slowdowns and the speedups are strongly scattered. The maximum achieved speedups are 10.4, 18.4 for FlowCutter and 10.3, 17.3 for ParPR-RL. These results match what we expected from Ref. [10]. Restricted to instances with sequential running time ≥ 10 seconds, the geometric mean speedups are 6.5 and 8.6 for FlowCutter and 7, 9.9 for ParPR-RL.

To evaluate the speedups of Mt-KaHyPar-D-F, we use a subset of set B (76 out of 94 hypergraphs)[4]. We measure the full partitioning process (top left) and just flow-based refinement including scheduling (bottom row). Note that we use a sequential push-relabel when there are enough flow problems that can be solved independently. The geometric mean speedup of Mt-KaHyPar-D-F is 3.1 for $t = 4$, 7.4 for $t = 16$ and 10.62 for $t = 64$. If we only consider instances with a single-threaded running time $\geq$ 100s, we achieve a geometric mean speedup of 14.5 for $t = 64$. For $k = 2$, the scalability of the flow-based refinement routine largely depends on FlowCutter as the only parallelism source. We can see that the speedups of the two are comparable (compare Figure 6 top-middle with bottom-left). There are a few outliers (e.g. `nlpkkt200` with a speedup of 80.05 for $t = 64$) where the flow network construction dominates the execution time for $t = 1$. For $k = 64$ and $t = 64$, we achieve a

[3] The instances are available from `https://algo2.iti.kit.edu/heuer/sea22`.

[4] Subset contains all hypergraphs on which Mt-KaHyPar-D-F 64 was able to complete in under 600 seconds for $k \in \{2, 8, 16, 64\}$. We omit scalability experiments with Mt-KaHyPar-Q-F due to the long time requirements and because flow-based refinement is used in the same context in Mt-KaHyPar-D-F. This experiment still took 4 weeks on machine B.

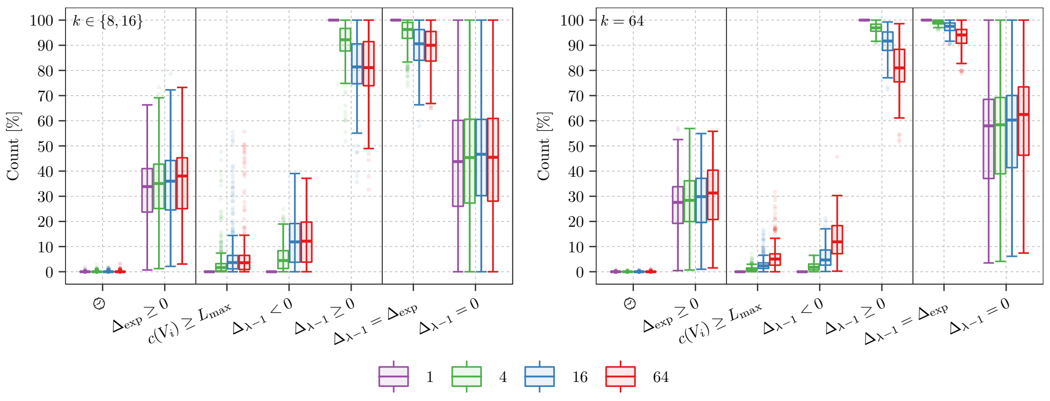

Figure 7 Conflicts for $k \in \{8, 16\}$ (left) and $k = 64$ (right) on set B. For each instance, we count the refinements that exceed the time limit (⏲), the potential improvements ($\Delta_{\exp} \geq 0$) and move sequences that violate the balance constraint ($c(V_i) \geq L_{\max}$) or degrade ($\Delta_{\lambda-1} < 0$) or improve the connectivity metric ($\Delta_{\lambda-1} \geq 0$). For move sequences with $\Delta_{\lambda-1} \geq 0$, we count if the actual improvements equals the expected ($\Delta_{\lambda-1} = \Delta_{\exp}$) and zero-gain improvements ($\Delta_{\lambda-1} = 0$).

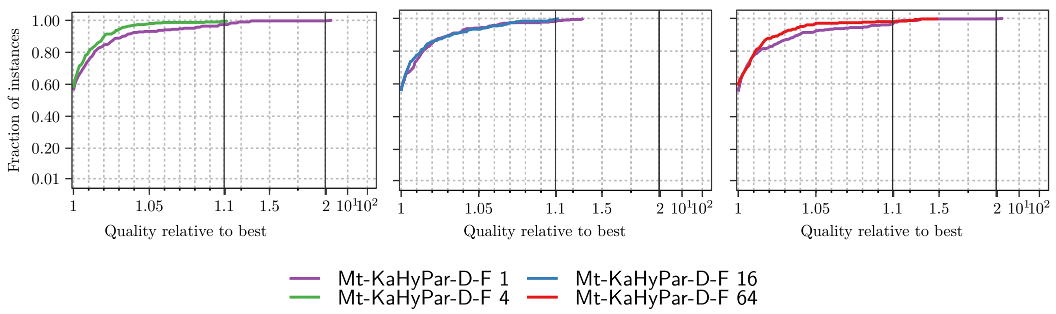

Figure 8 Performance profiles comparing solution quality of Mt-KaHyPar-D-F with increasing number of threads on set B.

geometric mean speedup of 18.48. In this case, all parallelism is leveraged in the scheduler, and none in FlowCutter, which explains why the speedups are more reliable than for other k. For $k \in \{8, 16\}$, both parallelism sources are used and speedups are better than for $k = 2$.

Search Interference. Figure 7 gives an overview on the different types of conflicts in the flow-based refinement routine (as explained in Section 5) and how often they occur. In the median, 33.38% of flow-based refinements find a potential improvement ($\Delta_{\exp} \geq 0$), of which we successfully apply 85.62% to the global partition for $t = 64$ (90.48% for $t = 16$ and 96.73% for $t = 4$). For $t = 64$, 2.55% of the move sequences violate the balance constraint (2% for $t = 16$ and 0.77% for $t = 4$) and 8.06% would actually degrade the solution quality before being reverted (4.74% for $t = 16$ and 1.72% for $t = 4$). However, increasing the number of threads does not adversely affect the solution quality of Mt-KaHyPar-D-F (see Figure 8), as repetitions from multiple rounds and on different levels can compensate effectively.

Detailed Running Times of Flow-Based Refinement. Figure 9 shows the running times of the different phases of the flow-based refinement routine relative to its total running time. For $k \leq 16$, FlowCutter dominates the running time. For $k = 64$, the flow network construction and FlowCutter have the same share on the total running time, while applying move sequences and growing the region B are negligible.

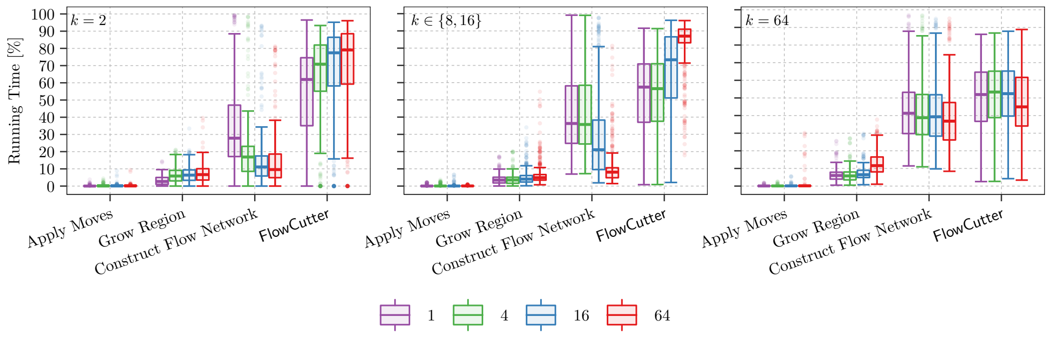

Figure 9 Running times of the different phases of the flow-based refinement routine relative to its total running time for $k = 2$ (left), $k \in \{8, 16\}$ (middle) and $k = 64$ (right) on set B.

10 Conclusion and Future Work

This work marks the end of a series of publications with the aim to transfer techniques used in modern sequential partitioning algorithms into the shared-memory context without comprises in solution quality. The result is a set of parallel algorithms unified in one framework [26, 27] (Mt-KaHyPar) that outperforms all popular hypergraph partitioners. Summarizing our experimental results, we obtain good speedups for larger values of k even on small instances, where scheduling provides lots of parallelism. This is more difficult for small k where the parallelism stems from the flow algorithm, yet we still obtain good speedups that match those in Ref. [10]. In particular on long-running instances, the speedups are on par with those for large k. Using 10 threads, our system is 10 times faster than the sequential state-of-the-art system KaHyPar with flow-based refinement, while achieving the same solution quality.

Future work includes a deterministic version of parallel flow-based refinement, as well as a highly localized version used in an n-level partitioner that only constructs small flow problems around uncontracted nodes.

References

1 Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a Direct k-way Hypergraph Partitioning Algorithm. In *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 28–42. SIAM, January 2017. doi:10.1137/1.9781611974768.3.

2 Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-Quality Shared-Memory Graph Partitioning. In *European Conference on Parallel Processing (Euro-Par)*, pages 659–671. Springer, August 2017. doi:10.1007/978-3-319-96983-1_47.

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

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

5 Reid Andersen. and Kevin J. Lang. An Algorithm for Improving Graph Partitions. In *Proc. of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 651–660. Society for Industrial and Applied Mathematics, 2008. doi:10.5555/1347082.1347154.

6 Richard J. Anderson and João C. Setubal. A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem. *J. Parallel Distributed Comput.*, 29(1):17–26, 1995. doi:10.1006/jpdc.1995.1103.

7 Aaron Archer, Kevin Aydin, Mohammad Hossein Bateni, Vahab S. Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. Cache-Aware Load Balancing of Data Center Applications. *Proceedings of the VLDB Endowment*, 12(6):709–723, 2019. doi:10.14778/3311880.3311887.

8 Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. Multi-level Direct k-way Hypergraph Partitioning With Multiple Constraints and Fixed Vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008. doi:10.1016/j.jpdc.2007.09.006.

9 David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph Partitioning and Graph Clustering*, volume 588. American Mathematical Society Providence, RI, 2013. doi:10.1090/conm/588.

10 Niklas Baumstark, Guy E. Blelloch, and Julian Shun. Efficient implementation of a synchronous parallel push-relabel algorithm. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, volume 9294 of *Lecture Notes in Computer Science*, pages 106–117. Springer, 2015. doi:10.1007/978-3-662-48350-3_10.

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

12 Ümit V. 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.

13 Boris V. Cherkassky and Andrew V. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997. doi:10.1007/PL00009180.

14 Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment*, 3(1):48–57, 2010. doi:10.14778/1920841.1920853.

15 Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011. doi:10.1145/2049662.2049663.

16 Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *Proc. of the 25th International Parallel and Distributed Processing Symposium*, pages 1135–1146, 2011. doi:10.1109/IPDPS.2011.108.

17 Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. Hypergraph Sparsification and Its Application to Partitioning. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 200–209, 2013. doi:10.1109/ICPP.2013.29.

18 Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Catalyurek. Parallel Hypergraph Partitioning for Scientific Computing. In *IEEE Transactions on Parallel and Distributed Systems*, pages 10–pp. IEEE, 2006. doi:10.1109/IPDPS.2006.1639359.

19 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91(2):201–213, 2002. doi:10.1007/s101070100263.

20 Charles M. Fiduccia and Robert M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Conference on Design Automation (DAC)*, pages 175–181, 1982. doi:10.1145/800263.809204.

21 Lester Randolph Ford and Delbert R Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi:10.4153/CJM-1956-045-5.

22 Andrew V. Goldberg and Robert Endre Tarjan. A New Approach to the Maximum-Flow Problem. *Journal of the ACM*, 35(4):921–940, 1988. doi:10.1145/48014.61051.

23 Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced Flow-Based Multilevel Hypergraph Partitioning. *18th International Symposium on Experimental Algorithms (SEA)*, 2020. doi:10.4230/LIPIcs.SEA.2020.11.

24 Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm. In *27th European Symposium on Algorithms (ESA)*, pages 52:1–52:17, 2019. doi:10.4230/LIPIcs.ESA.2019.52.

25 Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel Flow-Based Hypergraph Partitioning. Technical report, Karlsruhe Institute of Technology, 2022.

26 Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable Shared-Memory Hypergraph Partitioning. In *23st Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, January 2021. doi:10.1137/1.9781611976472.2.

27 Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Shared-Memory n-level Hypergraph Partitioning. In *24th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, January 2022. doi:10.1137/1.9781611977042.11.

28 Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021. doi:10.22331/q-2021-03-15-410.

29 Michael Hamann and Ben Strasser. Graph Bisection with Pareto Optimization. *ACM Journal of Experimental Algorithmics*, 23, 2018. doi:10.1145/3173045.

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

31 Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *16th International Symposium on Experimental Algorithms (SEA)*, pages 21:1–21:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2017. doi:10.4230/LIPIcs.SEA.2017.21.

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

33 Gökçehan Kara and Can C. Özturan. Graph Coloring Based Parallel Push-relabel Algorithm for the Maximum Flow Problem. *ACM Transactions on Mathematical Software*, 45(4):46:1–46:28, 2019. doi:10.1145/3330481.

34 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.

35 George Karypis and Vipin Kumar. Multilevel k-way Hypergraph Partitioning. *VLSI Design*, 2000(3):285–300, 2000. doi:10.1155/2000/19436.

36 Alexander V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. In *Soviet Mathematics Doklady*, volume 15, pages 434–437, 1974.

37 Brian W. Kernighan and Shen Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.

38 Balakrishnan Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. Computers*, 33(5):438–446, 1984. doi:10.1109/TC.1984.1676460.

39 Kevin J. Lang and Satish Rao. A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts. In *Proc. of 10th International Integer Programming and Combinatorial Optimization Conference*, volume 3064 of *LNCS*, pages 383–400. Springer, 2004. doi:10.1007/978-3-540-25960-2_25.

40 Dominique LaSalle and George Karypis. Multi-Threaded Graph Partitioning. In *IEEE Transactions on Parallel and Distributed Systems*, pages 225–236. IEEE, 2013. doi:10.1109/IPDPS.2013.50.

41 Eugene L. Lawler. Cutsets and Partitions of Hypergraphs. *Networks*, 3(3):275–285, 1973. doi:10.1002/net.3230030306.

42 Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. BiPart: A Parallel and Deterministic Hypergraph Partitioner. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–174, 2021. `doi:10.1145/3437801.3441611`.

43 Zoltán Á. Mann and Pál A. Papp. Formula Partitioning Revisited. In *5th Pragmatics of SAT Workshop*, pages 41–56, 2014. `doi:10.29007/9skn`.

44 Vitaly Osipov and Peter Sanders. n-Level Graph Partitioning. In *18th European Symposium on Algorithms (ESA)*, pages 278–289. Springer, 2010. `doi:10.1007/978-3-642-15775-2_24`.

45 David A. Papa and Igor L. Markov. Hypergraph Partitioning and Clustering. In *Handbook of Approximation Algorithms and Metaheuristics*. Citeseer, 2007. `doi:10.1201/9781420010749.ch61`.

46 Chuck Pheatt. Intel Threading Building Blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

47 Jean-Claude Picard and Maurice Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Math. Program.*, 22(1):121, 1982. `doi:10.1007/BF01581031`.

48 Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 76(3):036106, 2007. `doi:10.1103/PhysRevE.76.036106`.

49 Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *19th European Symposium on Algorithms (ESA)*, pages 469–480. Springer, 2011. `doi:10.1007/978-3-642-23719-5_40`.

50 Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2020. `doi:10.5445/IR/1000105953`.

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

52 Yossi Shiloach and Uzi Vishkin. An O(n^2 log n) Parallel Max-Flow Algorithm. *Journal of Algorithms*, 3(2):128–146, 1982. `doi:10.1016/0196-6774(82)90013-X`.

53 Aleksandar Trifunovic and William J. Knottenbelt. Parkway 2.0: A Parallel Multilevel Hypergraph Partitioning Tool. In *International Symposium on Computer and Information Sciences*, pages 789–800. Springer, 2004. `doi:10.1007/978-3-540-30182-0_79`.

54 Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite. In *49th Conference on Design Automation (DAC)*, pages 774–782. ACM, June 2012. `doi:10.1145/2228360.2228500`.

55 Hannah Honghua Yang and D.F. Wong. Efficient Network Flow Based Min-Cut Balanced Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1533–1540, 1996. `doi:10.1007/978-1-4615-0292-0_41`.

A Quotient Graph Maintenance

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

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

B Network Construction Algorithm

We implemented two construction algorithms that are preferable in different situations. Both construct the hypergraph $\mathcal{H}$ as explained at the beginning of Section 6. In the following, we will denote with $E_B := \{e \in E \mid e \cap B \neq \emptyset\}$ the set of hyperedges that contain nodes of the region B.

The first algorithm iterates over all nets $e \in E_B$. If a pin $p \in e$ is contained in B, we add p to hyperedge e in $\mathcal{H}$. Otherwise, we add the source s or sink t to e, if $p \in V_1$ or $p \in V_2$. The second algorithm iterates over all nodes $u \in B$ and for each net $e \in I(u)$, we insert a pair (e, u) into a vector. Sorting the vector (lexicographically) yields the pin lists of the subhypergraph H_B. Afterwards, we insert each net e in the pin list vector into $\mathcal{H}$ and add the source s or sink t to e, if $\Phi(e, B_1) < \Phi(e, V_1)$ or $\Phi(e, B_2) < \Phi(e, V_2)$.

The first algorithm has linear running time, but has to scan all hyperedges of H in their entirety in the worst case even if most of their pins are not contained in $\mathcal{H}$. The complexity of the second algorithm only depends on the number of pins in $\mathcal{H}$, but requires to sort the pin lists in a temporary vector. We use the second algorithm for hypergraphs with a low density $p := |E|/|V|$ (≤ 0.5) or a large average hyperedge size $\overline{|e|}$ (≥ 100).

Note that both algorithms discard single-pin nets and nets that contain both the source and sink (such nets cannot be removed from the cut).

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

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

Identical Net Removal. Since some nets of H are only partially contained in $\mathcal{H}$, some of them may become identical. Therefore, we further reduce the size of $\mathcal{H}$ by removing all identical nets except for one representative at which we aggregate their weight. We use the identical net detection algorithm of Aykanat et al. [8, 17]. It uses *fingerprints* $f_e := \sum_{v \in e} v^2$ to eliminate unnecessary pairwise comparisons between nets. Nets with different fingerprints or different sizes cannot be identical. If we insert a net e into $\mathcal{H}$, we store the pair (f_e, e) in a hash table with chaining to resolve collisions (uses concurrent vectors to handle parallel access). We can then use the hash table to perform pin-list comparisons between the nets with the same fingerprint for subsequent net insertions. Note that in the parallel scenario we may not be able to detect all identical nets due to simultaneous insertions into the hash table. However, this does not affect correctness of the refinement, as removing identical nets is only a performance optimization.

C Benchmark Sets

All instances of the benchmark sets used in the experimental evaluation are derived from four sources encompassing three application domains: the ISPD98 VLSI Circuit Benchmark Suite [3], the DAC 2012 Routability-Driven Placement Contest [54], the SuiteSparse Matrix Collection [15], and the 2014 SAT Competition [11]. VLSI instances are transformed into hypergraphs by converting the netlist of each circuit into a set of hyperedges. Sparse matrices

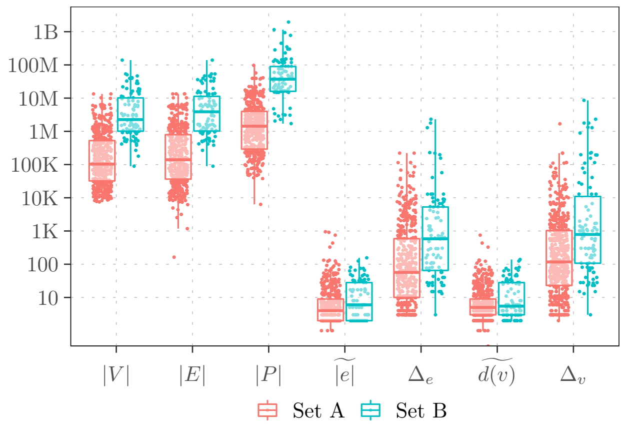

Figure 10 Summary of different properties for our two benchmark sets. It shows for each hypergraph (points), the number of vertices $|V|$, nets $|E|$ and pins $|P|$, as well as the median and maximum net size ($\widetilde{|e|}$ and Δ_e) and vertex degree ($\widetilde{d(v)}$ and Δ_v).

are translated to hypergraphs using the row-net model [12] and SAT instances to three different hypergraph representations: *literal*, *primal*, and *dual* [43, 45] (see Ref. [31] for more details). All hypergraphs have unit vertex and net weights. Figure 10 shows that the hypergraphs of set B are more than an order of magnitude larger than those of set A.