Scalable High-Quality Hypergraph Partitioning

LARS GOTTESBÜREN, TOBIAS HEUER, NIKOLAI MAAS, and PETER SANDERS, Karlsruhe Institute of Technology, Germany
SEBASTIAN SCHLAG, Independent Researcher, USA

Balanced hypergraph partitioning is an NP-hard problem with many applications, e.g., optimizing communication in distributed data placement problems. The goal is to place all nodes across k different blocks of bounded size, such that hyperedges span as few parts as possible. This problem is well-studied in sequential and distributed settings, but not in shared-memory. We close this gap by devising efficient and scalable shared-memory algorithms for all components employed in the best sequential solvers without compromises with regards to solution quality.

This work presents the scalable and high-quality hypergraph partitioning framework Mt-KaHyPar. Its most important components are parallel improvement algorithms based on the FM algorithm and maximum flows, as well as a parallel clustering algorithm for coarsening – which are used in a multilevel scheme with $\log(n)$ levels. As additional components, we parallelize the n-level partitioning scheme, devise a deterministic version of our algorithm, and present optimizations for plain graphs.

We evaluate our solver on more than 800 graphs and hypergraphs, and compare it with 25 different algorithms from the literature. Our fastest configuration outperforms almost all existing hypergraph partitioners with regards to both solution quality and running time. Our highest-quality configuration achieves the same solution quality as the best sequential partitioner KaHyPar, while being an order of magnitude faster with ten threads. Thus, two of our configurations occupy all fronts of the Pareto curve for hypergraph partitioning. Furthermore, our solvers exhibit good speedups, e.g., 29.6x in the geometric mean on 64 cores (deterministic), 22.3x ($\log(n)$-level), and 25.9x (n-level).

CCS Concepts: • **Theory of computation → Shared memory algorithms**; Graph algorithms analysis; • **Mathematics of computing** → *Hypergraphs;*

Additional Key Words and Phrases: Graph and hypergraph partitioning, shared-memory, high-quality, multilevel algorithm, determinism, concurrent gain computations, clustering, community detection, work-stealing, FM algorithm, maximum flows

ACM Reference format:
Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. 2024. Scalable High-Quality Hypergraph Partitioning. *ACM Trans. Algor.* 20, 1, Article 9 (January 2024), 54 pages.
https://doi.org/10.1145/3626527

The authors thank Michael Hamann, Daniel Seemaier, Christian Schulz and Dorothea Wagner for helpful discussions over the course of this research. This work was supported in part by DFG grants WA654/19-2 and SA933/11-1. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.
Authors' addresses: L. Gottesbüren, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76135 Karlsruhe, Germany; e-mail: lars.gottesbueren@kit.edu; T. Heuer, N. Maas, and P. Sanders, Karlsruhe Institute of Technology, Germany; e-mails: {tobias.heuer, nikolai.maas, sanders}@kit.edu; S. Schlag, Independent Researcher, Apple Inc., Cupertino, CA, USA; e-mail: sebastian_schlag@apple.com.

1549-6325/2024/01-ART9 $15.00
https://doi.org/10.1145/3626527

1 INTRODUCTION

The *balanced hypergraph partitioning problem* asks for a partition of the node set of a hypergraph into a fixed number of disjoint blocks with bounded size such that an objective function defined on the hyperedges is minimized. The two most prominent objective functions are the *edge cut* and *connectivity* metric. The former counts the number of hyperedges connecting more than one block, while the latter additionally considers the number of blocks spanned by each hyperedge. The problem has gained attraction in the field of very-large-scale-integration (VLSI) design already in the 1960s [37, 55, 98, 108]. Since then, it has been widely adopted in many other areas, such as minimizing the communication volume in parallel scientific simulations [23, 24, 28], storage sharding in distributed databases [31, 66, 78, 109, 125, 126], simulations of distributed quantum circuits [11, 51], and as a branching strategy in satisfiability solvers [5].

Unfortunately, balanced partitioning is NP-hard [41, 86] and hard to approximate [36]. Thus, heuristic solutions are used in practice – with the multilevel scheme emerging as the most successful method to achieve high solution quality in a reasonable amount of time [14, 56]. Figure 1 illustrates this technique, which consists of three phases. First, the hypergraph is *coarsened* to obtain a hierarchy of successively smaller and structurally similar approximations of the input hypergraph by *contracting* pairs or clusters of highly-connected nodes. Once the 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.

There is a diverse landscape of algorithms that implement the multilevel framework with different time-quality trade-offs, as illustrated in Figure 2. The plot shows two major shortcomings of existing solvers: (i) higher solution quality comes at the cost of higher running times often by several orders of magnitude, and (ii) parallel algorithms do not achieve the same solution quality as the best sequential systems because they use comparatively weaker components that are easier to parallelize (with the exception of our new solver Mt-KaHyPar). Historically, the parallel partitioning community has focused on algorithms for the distributed-memory model, which turned out to be not well-suited for the fine-grained parallelism required to effectively parallelize high-quality techniques. However, as the number of cores and main-memory capacity in modern machines increases, we believe that the shared-memory model has become a viable alternative for processing large (hyper)graphs and can be used for closing the quality gap between sequential and parallel partitioning algorithms.

Main Contributions. Figure 2 highlights the main contribution of this work: A shared-memory multilevel algorithm (Mt-KaHyPar) that achieves the same solution quality as the best sequential codes, while being faster than most of the relevant parallel algorithms in its fastest configuration. In particular, our Mt-KaHyPar solvers occupy all points on the Pareto frontier for hypergraphs (left) as well as the middle segment for graphs (right).

This is achieved by implementing parallel formulations for the core techniques used in the best sequential algorithms without compromises in solution quality. Our coarsening algorithm contracts a clustering of highly-connected nodes on each level and is guided by the community structure of the hypergraph. The clustering algorithm uses a less restrictive locking protocol than a previous approach [26] and resolves conflicting clustering decisions *on-the-fly*. Initial partitioning is done via parallel recursive bipartitioning and a portfolio solver, leveraging work-stealing to account for load imbalances. The key feature distinguishing Mt-KaHyPar from previous parallel systems are the substantially stronger local search algorithms. We present the first fully-parallel implementation of the FM algorithm and a parallel version of flow-based refinement. For these algorithms, we propose several novel and easy-to-implement solutions

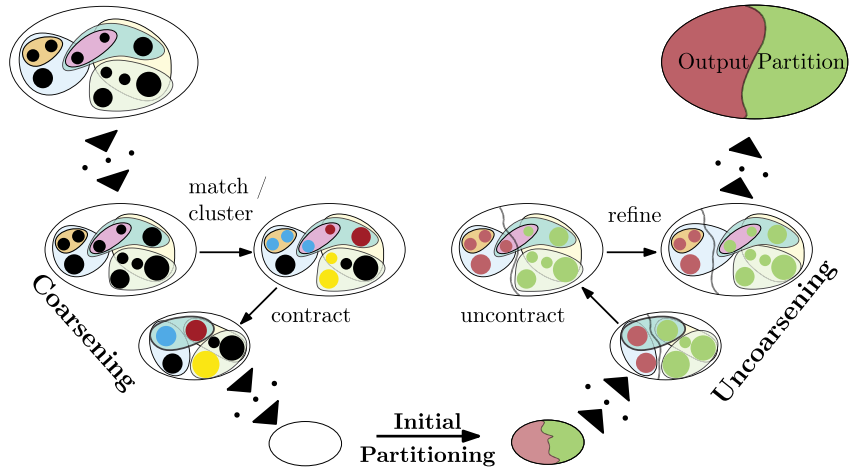

Fig. 1. The multilevel paradigm.

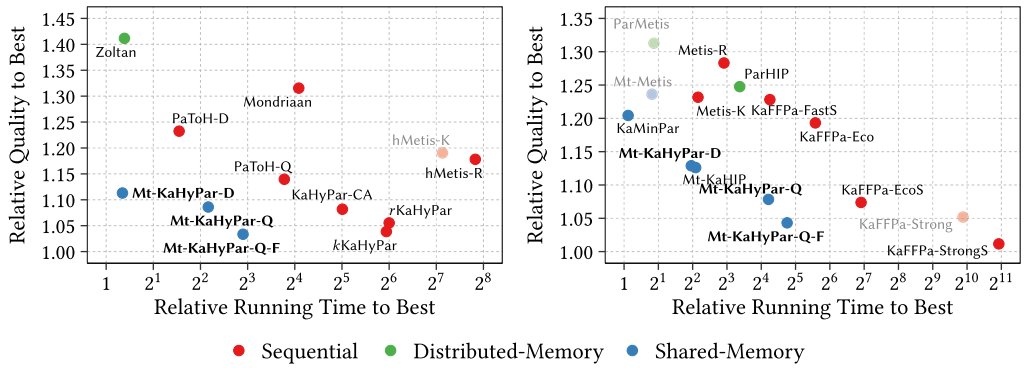

Fig. 2. Solution quality and running times of existing algorithms for hypergraph partitioning (left, connectivity metric) and graph partitioning (right, edge cut metric). For the y-values in the plot (solution quality), we compute the ratios of the objective values of an algorithm relative to the best value produced by any algorithm for each instance and aggregate them using the harmonic mean (similarly for running times on the x-axis). Markers on the lower left side are considered better. We run each parallel algorithm using 10 threads. Instances are restricted to more than two million edges/pins to make parallelism worthwhile (see set M_G and M_{HG} in Section 12). Partially transparent markers indicate solvers producing more than 15% infeasible partitions (either imbalanced or timeout).

to overcome some fundamental parallelization challenges such as, for example, techniques to (re)compute correct gain values for concurrent node moves.

We also present several extensions of the core multilevel algorithm. We devise the first parallel formulation of the n-level partitioning scheme – the most extreme instantiation of the multilevel technique – contracting only a single node on each level. Correspondingly, in each refinement step, only a single node is uncontracted followed by a highly-localized search for improvements around the uncontracted node, leading to more fine-grained refinement and ultimately better solution quality in a single run. Furthermore, we present a deterministic version of our multilevel algorithm. This offers reproducible results and thus also stable results, whereas previous algorithms may have large variance from repeated runs. Furthermore, some

applications even require deterministic results or value them highly (e.g., VLSI design due to manual post-processing). Moreover, we present data structure optimizations that speed up our algorithm by a factor of two when running on plain graphs instead of hypergraphs.

In our extensive experimental evaluation, we compare Mt-KaHyPar to 25 different sequential and parallel graph and hypergraph partitioners on over 800 graphs and hypergraphs with up to 2 billion edges/pins. As of today and to the best of our knowledge, this is the most comprehensive comparison of partitioning algorithms in the literature. As a main result, the highest-quality configuration of Mt-KaHyPar produces partitions that are on par with KaHyPar [104] – the best sequential hypergraph partitioner – while being almost an order of magnitude faster with only ten threads. The fastest configuration of Mt-KaHyPar achieves a self-relative speedup of 22.3 with 64 threads and computes partitions that are 23% better than those of Zoltan [34] (distributed-memory), while being a factor of 2.72 faster on average. Out of all evaluated algorithms, KaFFPa [101] (sequential) computes slightly better solutions, while KaMinPar [48] (shared-memory) is faster than Mt-KaHyPar.

The work presents the main results of several conference publications [43, 46, 47, 50] and summarizes the dissertations of Gottesbüren [49] and Heuer [60]. The added value of the paper is the detailed overview of the overall framework that contains the highest-quality and one of the fastest algorithms for partitioning (hyper)graphs. This paper puts particular focus on our multilevel partitioning algorithm which provides the best time-quality trade-off. We describe the algorithm with a greater level of detail compared to the corresponding conference version [50]. Furthermore, the previously mentioned optimizations for graph partitioning are unpublished. Another key contribution is the large experimental evaluation, going beyond the scope of the individual publications by including graph partitioning, breaking down the running times of individual components, and including even more competing baseline algorithms. We included almost all publicly available multilevel graph and hypergraph partitioning algorithms to provide a comprehensive overview on the landscape of partitioning tools.

Outline. Section 2 introduces basic notation and definitions used throughout this work. We then start the algorithm description with a high-level overview of the multilevel partitioning algorithm in Section 3. The following sections are structured according to the different phases of the multilevel scheme: Section 4 and 5 describe the coarsening and initial partitioning algorithm, while we discuss different concurrent gain (re)computation techniques and the implementation of the parallel FM and flow-based refinement algorithms in Section 6–8. In Section 9, we present the parallelization of the n-level partitioning scheme, and conclude the algorithmic part with our data structure optimizations for graph partitioning and the deterministic version of the multilevel algorithm in Section 10 and 11. We then turn to the experimental evaluation in Section 12. Here, we evaluate the solution quality and scalability of the different configurations of Mt-KaHyPar, and compare them to existing partitioning algorithms. Section 13 concludes the work and presents directions for future research.

As this paper covers a wide range of partitioning techniques, we review relevant literature in the corresponding sections. For a comprehensive overview on (hyper)graph partitioning, we refer the reader to existing surveys [8, 13, 21, 27, 94] and the literature overviews in the theses of Lars Gottesbüren [49], Tobias Heuer [60], and Sebastian Schlag [104].

2 PRELIMINARIES

Hypergraphs. A *weighted hypergraph* $H = (V, E, c, \omega)$ is defined as a set of n nodes V and a set of m hyperedges E (also called *nets*) with node 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 node set V. The nodes of a net are called its *pins*. We extend

c and ω to sets in a natural way, i.e., $c(U) := \sum_{u \in U} c(u)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A node u is *incident* to a net e if $u \in e$. $I(u) := \{e \mid u \in e\}$ is the set of all incident nets of u. The set $\Gamma(u) := \{v \mid \exists e \in E : \{u, v\} \subseteq e\}$ denotes the neighbors of u. Two nodes u and v are *adjacent* if $v \in \Gamma(u)$. The *degree* of a node u is $d(u) := |\mathrm{I}(u)|$. The *size* $|e|$ of a net e is the number of its pins. Nets of size one are called *single-pin* nets. We denote the number of pins of a hypergraph with $p := \sum_{e \in E} |e| = \sum_{v \in V} d(v)$. 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 $H[V'] := (V', \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\}, c, \omega')$ where $\omega'(e \cap V')$ is the weight of hyperedge e in H. The *bipartite graph representation* $G_x := (V \cup E, E_x)$ [65, 108] of an unweighted hypergraph $H = (V, E)$ contains the nodes and nets of H as node set and for each pin $u \in e$, we add an undirected edge $\{u, e\}$ to E_x. More formally, $E_x := \{\{u, e\} \mid \exists e \in E : u \in e\}$.

Clusterings and Partitions. A *clustering* $C = \{C_1, \ldots, C_l\}$ of a hypergraph $H = (V, E, c, \omega)$ is a partition of the node set V into disjoint subsets. A cluster C_i is called a *singleton* cluster if $|C_i| = 1$. A node contained in a singleton cluster is called *unclustered*. A *k-way partition* of a hypergraph H is a clustering into a predefined number of disjoint blocks $\Pi = \{V_1, \ldots, V_k\}$. A 2-way partition is also called a *bipartition*. We denote the block to which a node u is assigned by $\Pi[u]$. 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 $\Phi(e, V_i) := |e \cap V_i|$. We refer to $\Phi(e, V_i)$ as the *pin count value* for a net e and block V_i. The set $E(V_i, V_j) := \{e \in E \mid \{V_i, V_j\} \subseteq \Lambda(e)\}$ represents the cut nets connecting block V_i and V_j. Two blocks V_i and V_j are *adjacent* if $E(V_i, V_j) \neq \emptyset$. The *quotient graph* $Q := (\Pi, E_\Pi := \{(V_i, V_j) \mid E(V_i, V_j) \neq \emptyset\})$ contains an edge between all adjacent blocks.

The Balanced Hypergraph Partitioning Problem. The *balanced hypergraph partitioning problem* is to find a k-way partition Π of a hypergraph H that minimizes an objective function defined on the hyperedges where each block $V' \in \Pi$ satisfies the *balance constraint*: $c(V') \leq L_{\max} := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$[1] for some *imbalance ratio* $\varepsilon \in (0, 1)$. If Π satisfies the balance constraint, we call Π *ε-balanced* or just say *balanced* or *feasible* when ε is clear from the context. For $k = 2$, we refer to the problem as the *bipartitioning* problem. The two most prominent objective functions are the *cut-net* metric $\mathfrak{f}_c := \sum_{e \in E_{\mathrm{Cut}}(\Pi)} \omega(e)$ (also called *edge cut* metric for graph partitioning) and *connectivity* metric $\mathfrak{f}_{\lambda-1}(\Pi) := \sum_{e \in E_{\mathrm{Cut}}(\Pi)} (\lambda(e) - 1) \cdot \omega(e)$ (also called $(\lambda - 1)$-metric) where $E_{\mathrm{Cut}}(\Pi)$ denotes the set of all cut nets. The cut-net metric directly generalizes the edge cut metric from graphs to hypergraphs and minimizes the weight of all cut hyperedges. The connectivity metric additionally considers the number of blocks connected by a net and thus more accurately models the communication volume for parallel computations [28] (e.g., for the parallel sparse matrix-vector multiplication). The hypergraph partitioning problem is NP-hard for both objective functions [86].

Recursive Bipartitioning vs Direct k-way Partitioning. A k-way partition of a hypergraph can be obtained either by *recursive bipartitioning* or *direct k-way partitioning*. The former first computes a bipartition and then calls the bipartitioning routine on both blocks recursively until the input hypergraph is divided into the desired number of blocks. The latter partitions the hypergraph directly into k blocks and applies k-way local search algorithms to improve the solution.

[1]The $\lceil \cdot \rceil$ in this definition ensures that there is always a feasible solution for inputs with unit node weights. However, this does not hold for general weighted inputs as even finding any balanced solution (ignoring the objective function) is an NP-hard problem [40]. There exist several alternative definitions [48, 61], but no commonly accepted way how to deal with feasibility. In this work, we use the original definition since our benchmark instances are unweighted.

ALGORITHM 3.1: The Multilevel Partitioning Algorithm

Input: Hypergraph $H = (V, E)$, number of blocks k
Output: k-way partition Π of H

```
1  H_1 ← H;   H ← ⟨H_1⟩;   n ← 1
2  while V_i has too many nodes do
3      C ← ComputeClustering(H_n)
4      H_{n+1} ← H_n.Contract(C);   H ← H ∪ ⟨H_{n+1}⟩;   ++n
5  Π ← InitialPartition(H_n, k)
6  for i = n − 1 down to 1 do
7      Π ← project Π onto H_i
8      LabelPropagationRefinement(H_i, Π)        // finds easy improvements by moving single nodes
9      FMRefinement(H_i, Π)                      // finds short and non-trivial move sets
10     FlowBasedRefinement(H_i, Π)               // global optimization finding long and complex move sets
11 return Π
```

3 A BRIEF OVERVIEW OF THE PARTITIONING ALGORITHM

Algorithm 3.1 shows the high-level structure of our multilevel partitioning algorithm. While the pseudocode presented does not explicitly exhibit parallelism, it shows the algorithmic components for which we provide parallel implementations.

The coarsening algorithm proceeds in rounds until the hypergraph is considered as small enough for initial partitioning. In each round, we find a clustering of highly-connected nodes and subsequently contract the clustering in parallel. The clustering algorithm iterates over the nodes in parallel and finds the best target cluster for a node according to a rating function. Afterwards, the node joins its desired cluster for which we implement a novel locking protocol that detects and resolves conflicting clustering decisions on-the-fly.

Initial partitioning is done via parallel recursive bipartitioning using a novel work-stealing approach to account for load imbalances within the parallel bipartitioning calls. To compute an initial bipartition, we use a portfolio of *nine* different bipartitioning techniques, which is run several times in parallel. The best bipartition out of all runs is then used as initial solution.

In the uncoarsening phase, we project the partition onto the next hypergraph in the hierarchy by assigning the nodes to the block of their corresponding constituent in the coarser representation. Subsequently, we improve the partition using three different parallel refinement algorithms: label propagation refinement (used in most of the existing parallel partitioning algorithms), a highly-localized version of the FM algorithm (improves an existing implementation used in Mt-KaHIP [4]), and a novel parallelization of flow-based refinement. The rationale behind the use of three different local search algorithms executed in this order is that it allows for increasingly better solution quality at the cost of higher running times.

The following sections are structured according to the different phases of the multilevel scheme, and provide a more detailed explanation of the different algorithmic components of Algorithm 3.1. In Section 4 and 5, we present our coarsening and initial partitioning algorithm. The description of the uncoarsening phase is split into three separate sections: Section 6 describes the partition data structure and several concurrent gain (re)computation techniques, while Section 7 and 8 presents our parallel FM and flow-based refinement algorithm.

4 THE COARSENING PHASE

The goal of the coarsening phase is to find successively smaller and structurally similiar approximations of the input hypergraph [117] such that initial partitioning can find a partition of high quality

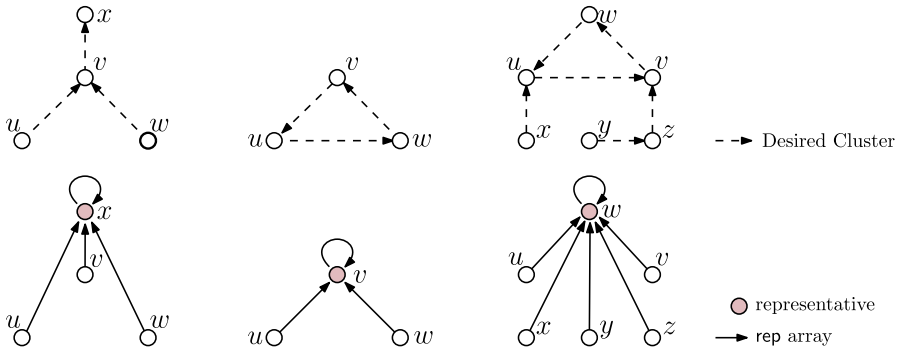

Fig. 3. A path (top-left) and cyclic conflict (top-middle), and a combination of both conflicts (top-right) with their resolutions (bottom).

not significantly worse than the partition that can be found on the input hypergraph [68]. This can be achieved by grouping highly-connected nodes together and merging each group into a single node, which can be done by computing either a matching [30, 34, 56, 64, 70–72, 81, 90, 115, 119, 120] or clustering of the nodes [4, 28, 48, 69, 88, 91, 101, 113]. The latter was shown to be more effective in reducing the size of (hyper)graphs with highly-skewed node degree distributions [1, 91] (e.g., social networks). In the following, we present our parallel clustering-based coarsening algorithm that works similar to the shared-memory version of PaToH's coarsening scheme [26]. However, the algorithm of Çatalyürek et al. [26] excessively locks nodes when evaluating the rating function and adding nodes to clusters. We therefore propose a less restrictive locking protocol that completely omits locking nodes when computing the best target cluster for a node. Moreover, it detects and resolves conflicting clustering decisions *on-the-fly*, while previous approaches relied on a postprocessing step [2, 26, 81].

4.1 The Clustering Algorithm

Our coarsening algorithm repeatedly finds a clustering C of the nodes and subsequently contracts it until the hypergraph is small enough. We represent the clustering C using an array rep of size n. We then choose one representative $v \in C$ for each cluster $C \in \mathcal{C}$ and store $\text{rep}[u] = v$ for each node $u \in C$. Initially, each node is unclustered (i.e., $\text{rep}[u] = u$ for each node $u \in V$). The clustering algorithm then iterates over the nodes in parallel and assigns each unclustered node to the best target cluster according to a rating function, which we introduce in the subsequent paragraph.

Cluster Join Operation. Once a node u chooses its desired target cluster C represented by a node v, we have to set $\text{rep}[u] = v$. Since several nodes can join clusters simultaneously, there may occur conflicts that must be resolved. As illustrated in Figure 3, there are two types of conflicts: *path* and *cyclic* conflicts. A path conflict involves several nodes $u_1, \ldots, u_l$ and occurs when each node u_i tries to join u_{i+1}. In a cyclic conflict, the last node u_l additionally tries to join u_1. It is also possible that a combination of both conflicts occurs, as illustrated in Figure 3 (right). We can resolve a path conflict when each node u_i waits until u_{i+1} has joined its desired cluster. Afterwards, we can set $\text{rep}[u_i] = \text{rep}[u_{i+1}]$ to resolve the conflict. However, applying this resolution scheme to cyclic conflicts would result in a deadlock. Therefore, the threads must agree on a cluster join operation that breaks the cycle and reduces it to a path conflict.

Algorithm 4.1 shows the pseudocode of our cluster join operation, which takes a node u as input, and adds it to a cluster represented by a node v. The algorithm associates each node with one of the following three states: *unclustered*, currently *joining* a cluster, or *clustered*. Unclustered nodes

ALGORITHM 4.1: Cluster Join Operation

```
   Input: A node u that wants to join v's cluster
1  if compare-and-swap(state[u], Unclustered, Joining) then
2      if state[v] = Clustered or compare-and-swap(state[v], Unclustered, Joining) then
3          rep[u] ← rep[v]
4      else                                   // Another thread tries to add v to a cluster
5          while state[v] = Joining do                                  // busy-waiting loop
6              if cyclic conflict detected and u is node with smallest ID in cycle then
7                  rep[u] ← rep[v];   state[u], state[v] ← Clustered;   break
8          if state[u] = Joining then rep[u] ← rep[v]                   // resolves path conflicts
9      state[u], state[v] ← Clustered
```

(rep[u] = u) can join clusters, while an already clustered node is not considered by the clustering algorithm anymore and therefore its representative does not change. If a thread sets the state of a node u from unclustered to joining via an atomic `compare-and-swap` operation, it acquires exclusive ownership for modifying rep[u] and setting its state to clustered. Thus, if we succeed in setting the state of u and v to joining or v is already clustered, we can safely set rep[u] = rep[v] (see Line 1–3) since this guarantees that no other thread modifies rep[u] and rep[v]. Note that the representative of v may have changed due to concurrent cluster join operations. In that case, its representative is stored in rep[v]. We therefore always set rep[u] = rep[v] (instead of rep[u] = v).

If another thread sets the state of v to joining, we know that v also tries to join a cluster. To resolve the conflict, we spin in a busy-waiting loop until the state of v is updated to clustered (see Line 5), and then join its new cluster (path conflict). In the busy-waiting loop, we additionally check if u is part of a cycle of nodes trying to join each other. To detect a cyclic conflict, each node writes its desired target cluster into a globally shared vector and checks if this induces a cycle. If so, the node with the smallest ID in the cycle gets to join its desired cluster, thus breaking the cycle.

Rating Function. A node u joins the cluster C maximizing the heavy-edge rating function

$$r(u, C) = \sum_{e \in I(u) \cap I(C)} \frac{\omega(e)}{|e| - 1}.$$

The rating function is commonly used in the partitioning literature [3, 28, 69] and prefers clusters connected to u via a large number of heavy nets with small size. We evaluate the rating function by iterating over the incident nets $e \in I(u)$ and aggregating the ratings to the representatives rep[v] of each pin $v \in e$ in a thread-local hash table. Afterwards, we iterate over the aggregated ratings and determine the representative rep[v] that maximizes $r(u, \text{rep}[v])$. Ties are broken uniformly at random. Subsequently, we perform the cluster join operation that sets rep[u] = rep[v].

To aggregate ratings, we use fixed-capacity linear probing hash tables with 2^{15} entries and resort to a larger hash table if the fill ratio exceeds $1/3$ of the capacity. This technique can considerably reduce the number of cache misses since most neighborhoods are small in real-world hypergraphs. We further note that the representative of a node can change during the evaluation of the rating function since we do not lock the nodes. However, it has already been shown that such conflicts rarely happen in practice [26] and therefore have a negligible impact on the partitioning result.

Contraction Limit. We stop coarsening when the number of nodes in the smallest hypergraph reaches $160k$. This contraction limit was chosen based on our prior research on sequential hypergraph partitioning [58]. In addition, we terminate the clustering algorithm when the number of

nodes would drop below $\frac{c(V)}{2.5}$ after the contraction step. This prevents the coarsening process from reducing the size of the hypergraph too aggressively [1, 69]. Conversely, we also stop coarsening if the contraction step does not reduce the number of nodes by more than 1%, even if the $160k$ node limit is not reached. This can happen since we enforce an upper weight limit $c_{\max}$ on the weight of the heaviest cluster (set to $\frac{c(V)}{160k}$ as in KaHyPar [58]), which prevents highly-skewed node-weight distributions that would make it difficult for initial partitioning to find a balanced solution [3, 91]. When adding a node to a cluster $C \in \mathcal{C}$, we ensure that $c(C) \leq c_{\max}$ by updating cluster weights via atomic `fetch-and-add` instructions. If $c(C) > c_{\max}$ after the update, we reject the corresponding cluster join operation and revert the cluster weight update. The cluster weight limit can lead to coarsening passes that do not sufficiently reduce the size of the hypergraph.

4.2 The Contraction Algorithm

The hypergraph data structure stores the incident nets $I(u)$ of each node $u \in V$ and the pin-lists of each net $e \in E$ using two adjacency arrays. Each node u and net e additionally stores its weight $c(u)$ and $\omega(e)$. Contracting a clustering $\mathcal{C} = \{C_1, \ldots, C_l\}$ replaces each cluster C_i with one supernode u_i with weight $c(u_i) = \sum_{v \in C_i} c(v)$. For each net $e \in E$, we replace each pin $v \in e$ with the node u_i representing the cluster C_i in which v is contained ($\text{rep}[v] = u_i$). After the replacement, multiple occurrences of the same supernode in a net are discarded.

Our contraction algorithm consists of several simple, easily parallelizable operations including remapping node IDs to a consecutive range, aggregating cluster weights and degrees using atomic `fetch-and-add` instructions, eliminating duplicated entries in pin-lists, and using parallel prefix sum operations to construct the adjacency arrays of the contracted hypergraph. As these steps are rather low level, we refer the reader to Reference [60, p. 87] for more details.

A challenging aspect is removing duplicates from the set of nets. We identify groups of identical nets and remove all but one representative per group to which we assign their aggregate weight. This can reduce the number of pins significantly and therefore accelerates the other algorithmic components. A simple algorithm is to perform pair-wise comparisons between all nets, which is however too expensive in practice. To this end, we parallelize the InrSrt algorithm of Aykanat et al. [12, 33] for identical net detection. It uses *fingerprints* $f(e) := \sum_{v \in e} v^2$ to eliminate unnecessary pairwise comparisons between nets, by grouping nets with equal fingerprints via sorting. Nets with different fingerprints or different sizes cannot be identical. We distribute the fingerprints and their associated nets to the threads using a hash function. Each thread sorts the nets by their fingerprint and size, and then performs pairwise comparisons on the subranges of potentially identical nets. We aggregate the weights of identical nets at a representative and mark the others as invalid in a bitset. A parallel prefix sum over the bitset maps the hyperedge IDs to a consecutive range in the contracted hypergraph. Note that we also remove nets that contain only a single pin since they do not contribute to the cut.

4.3 Community-Aware Coarsening

A popular approach to improve an existing k-way partition Π is the *iterated multilevel cycle* technique [118] (also called *V-cycle*). In the coarsening phase, the algorithm forbids contractions between nodes that are not in the same block in Π, thus preserving the already identified cut structure. While the technique can be effective, using it as a postprocessing step in a multilevel algorithm almost doubles the running time. As a more lightweight alternative, Heuer and Schlag [63] proposed using a clustering of the nodes computed via a community detection algorithm instead of an existing k-way partition. Community detection still captures the sparse cut patterns that are often found in good k-way partitions. The authors showed that this substantially improves the

quality of both the initial and the final partition, and only slightly increases the running time of the overall algorithm.

We also integrate the approach into our partitioning algorithm. We run the algorithm as a preprocessing step before the coarsening phase and then use the clustering to restrict contractions to nodes that belong to the same cluster. The algorithm consists of two steps: transforming the hypergraph into its bipartite graph representation and then running the parallel Louvain method of Staudt and Meyerhenke [18, 111] for modularity maximization, a widely used objective function for community detection [20, 92].

5 THE INITIAL PARTITIONING PHASE

Partitioning algorithms based on the direct k-way partitioning scheme often use multilevel recursive bipartitioning to obtain an initial k-way partition [3, 12, 72, 107], as this leads to partitions with significantly better solution quality than using flat (non-multilevel) k-way partitioning methods. Many parallel partitioners run sequential initial partitioning algorithms in parallel [4, 34, 64, 70, 113, 114, 120]. However, the sequential calls can become a bottleneck when the smallest hypergraph is still large. A more scalable approach parallelizes the recursive calls after each bipartitioning operation [30, 83]. The common approach is to statically split the thread pool along with the subproblems. Since this can lead to load imbalance when processing hypergraphs with unequal densities in the recursive partitioning calls, we instead generate tasks that can be dynamically load balanced using work stealing.

Parallel Recursive Bipartitioning. We compute initial k-way partitions via parallel recursive bipartitioning using Algorithm 3.1 initialized with $k = 2$ (without flow-based refinement). For the bipartitioning case, we replace the initial partitioning call with a portfolio of bipartitioning techniques.

Once we obtain a bipartition $\Pi = \{V_1, V_2\}$ of the input hypergraph H, we extract the subhypergraphs $H[V_1]$ and $H[V_2]$ and recurse on both in parallel by partitioning $H[V_1]$ into $\lceil \frac{k}{2} \rceil$ and $H[V_2]$ into $\lfloor \frac{k}{2} \rfloor$ blocks. We ensure that the final k-way partition obtained via recursive bipartitioning is ε-balanced by adapting the imbalance ratio for each bipartition individually [105]. Let $H[V']$ be a subhypergraph that should be recursively partitioned into $k' \leq k$ blocks. Then,

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

is the imbalance ratio used for the bipartition of $H_{V'}$. If each bipartition is ε'-balanced, then it is guaranteed that the final k-way partition is ε-balanced [104, Lemma 4.1 on p. 104].

Portfolio-Based Bipartitioning. We implemented the same portfolio of initial bipartitioning techniques as in KaHyPar [58, 105], including seven different variants of (greedy) hypergraph growing [25, 28, 69, 71, 104, 105, 107], random assignment [25, 69, 104, 105, 115], and label propagation initial partitioning [104, 105]. We refer the reader to Reference [60, p. 95–96] for more details on their implementation. We run each algorithm independently in parallel for at least 5 and at most 20 times. After 5 runs, we only run an algorithm again if it is likely to improve the best solution Π^* found so far. We estimate this based on the arithmetic mean μ and standard deviation σ of the connectivity values achieved by that algorithm so far, using the 95% rule. Assuming the connectivity values follow a normal distribution, roughly 95% of the runs will fall between $\mu - 2\sigma$ and $\mu + 2\sigma$. If $\mu - 2\sigma > f_{\lambda-1}(\Pi^*)$, we do not run the algorithm again. Additionally, we refine each bipartition using sequential 2-way FM refinement [37]. We continue uncoarsening using the bipartition with the best connectivity value. In case of ties, we prefer the bipartition with the best balance.

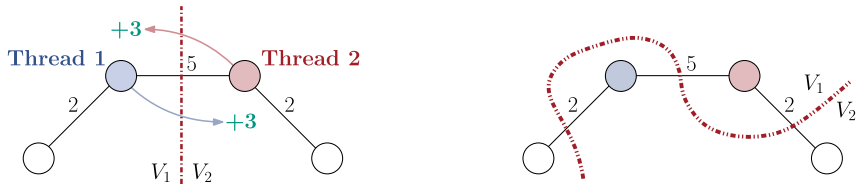

Fig. 4. Example of a move conflict when two nodes are moved simultaneously. Both threads assume that the individual node moves eliminate the edge with weight 5 from the cut. However, the edge is still cut after moving both nodes and the edges with weight 2 become cut edges.

6 GAIN COMPUTATION TECHNIQUES

Local search algorithms greedily move nodes to different blocks according to a *gain value*. The gain value reflects the change in the objective function for a particular node move. For the connectivity metric, the gain $g_u(V_t)$ of moving a node u to a target block V_t can be expressed as follows:

$$g_u(V_t) := \omega(\{e \in I(u) \mid \Phi(e, \Pi[u]) = 1\}) - \omega(\{e \in I(u) \mid \Phi(e, V_t) = 0\}).$$

Moving node u to block V_t decreases the connectivity of all nets by one for which u is the last remaining pin in its current block $\Pi[u]$. Conversely, the move increases the connectivity of all nets $e \in I(u)$ by one for which no pin $v \in e$ is assigned to the target block V_t.

To achieve meaningful speedups, parallel refinement algorithms need to move nodes concurrently. The actual gain of a node move can change between the time it is initially calculated and the time it is applied to the partition, due to concurrent node moves in its neighborhood [70]. As a consequence, two concurrent node moves can worsen the connectivity metric, even if their individual gains suggested an improvement, as illustrated in Figure 4. Thus, correctly calculating gains is a fundamental challenge for parallel refinement algorithms.

These conflicts occur when two adjacent nodes change their blocks simultaneously. Common remedies include computing a node coloring and only moving nodes of the same color at a time [70], scheduling 2-way refinement algorithms on block pairs that form a matching in the quotient graph in parallel [64, 120], allowing only node moves from a block V_s to V_t if $s < t$ [34, 81, 113] (and vice versa in a second phase), or following an optimistic strategy assuming that conflicts happen rarely in practice [4, 48, 88, 91].

The presented approaches still allow all individual node moves, but combining arbitrary moves into a single move sequence might be not always possible. This is problematic for parallelizing local search techniques as their sequential counterparts often identify a set of moves that only yield an improvement if moved together. While ignoring search conflicts appears to be the preferred approach, their impact on solution quality is unpredictable and deserves further consideration.

We therefore contribute several parallel gain computation techniques to compute accurate gain values and detect conflicts between moves without restricting possible moves. We present a technique named *attributed gains* to double-check the gain of a node move in Section 6.1, a concurrent *gain table* to accelerate gain calculations and communicate updates between threads in Section 6.2, and a novel parallel algorithm for *recomputing exact gains* of a sequence of node moves in Section 6.3. These techniques build on our concurrent partition data structure which we describe in the next section in more detail.

6.1 The Partition Data Structure

Our partition data structure stores and maintains the block assignments Π, the block weights $c(V_i)$, the pin count values $\Phi(e, V_i)$, and connectivity sets $\Lambda(e)$ for each net $e \in E$ and block $V_i \in \Pi$.

ALGORITHM 6.1: The Move Node Operation

Input: A node u that should be moved from its source block V_s to a target block V_t
Output: Attributed gain value Δ

```
1  c_t ← fetch-and-add(c(V_t), c(u))
2  if c_t + c(u) > L_max then                    // Revert block weight update if balance constraint violated
3      c(V_t) −=(atomic) c(u); return 0
4  Π[u] ← V_t;   c(V_s) −=(atomic) c(u);   Δ ← 0
5  for e ∈ I(u) do
6      lock(e);   Φ_s ← --Φ(e, V_s);   Φ_t ← ++Φ(e, V_t);   unlock(e)
       // Update connectivity set Λ(e) and attributed gain value Δ
7      if Φ_s = 0 then Λ(e) ← Λ(e) \ {V_s};   Δ += ω(e)
8      if Φ_t = 1 then Λ(e) ← Λ(e) ∪ {V_t};   Δ −= ω(e)
9      UpdateGainTable(e, Φ_s, Φ_t)                // see Section 6.2
10 return Δ
```

The Move Node Operation. Algorithm 6.1 shows the updates to the partition data structure when moving a node u from its source block V_s to a target block V_t. We only perform a node move if it does not violate the balance constraint, which we ensure by adding the weight of node u to the weight of block V_t via an atomic `fetch-and-add` instruction. If the node move is feasible, we update the block assignment of node u to block V_t and subtract the node weight u from its previous block V_s. If the move is infeasible, we subtract the weight again and reject the move.

Data Layout. The size of a pin count value is bounded by the size of the largest hyperedge. To save memory, we use a packed representation with $\lceil \log(\max_{e \in E} |e|) \rceil$ bits per entry for the $\Phi(e, V_i)$ values. Furthermore, we use a bitset of size k to store the connectivity set $\Lambda(e)$ of each hyperedge $e \in E$. We iterate over the connectivity set $\Lambda(e)$ by taking a snapshot of its bitset and then use *count-leading-zeroes* instructions. We compute the connectivity $\lambda(e) = |\Lambda(e)|$ of a hyperedge e using *pop-count* instructions (counts the number of 1-bits in a machine word). To add or remove a block from the connectivity set, we flip the corresponding bit using an atomic `xor` operation. The move node operation can be made lock-free by updating $\Phi(e, V_i)$ with atomic `fetch-and-add` instructions, but this requires one machine word per value. We therefore use a spin-lock for each net e due to the packed representation.

Attributed Gains. As the gain value of a node move can change between its initial calculation and actual execution due to concurrent node moves in its neighborhood, we additionally compute an *attributed gain* value for each move based on the atomic updates of the pin count values $\Phi(e, V_s)$ and $\Phi(e, V_t)$ in Line 6 of Algorithm 6.1. We attribute a connectivity decrease by $\omega(e)$ to the move that reduces $\Phi(e, V_s)$ to zero (see Line 7) and an increase by $\omega(e)$ for increasing $\Phi(e, V_t)$ to one (see Line 8).

Since we do not lock all incident nets $e \in I(u)$ before moving a node u, there is no guarantee on the order in which concurrent moves perform the pin count updates. Hence, this scheme may distribute the connectivity reductions to different threads, but the sum of the attributed gains of all node moves equals the overall connectivity reduction [60].

Attributed Gains for Label Propagation Refinement. The most widely used refinement technique in parallel partitioning algorithms is label propagation [4, 48, 70, 88, 91, 113, 121]. The algorithm works in rounds. In each round, it iterates over all nodes in parallel, and whenever it visits a node u, it moves it to the block V_t maximizing its move gain $g_u(V_t)$ (respecting the balance constraint). The algorithm only performs moves with positive gain and therefore cannot escape from local

optima. However, we use it in our partitioning algorithm to find all *simple* node moves such that our more advanced refinement techniques can focus on finding non-trivial improvements (for more technical details on its implementation, see Reference [60, p. 68–69]).

Since the label propagation algorithm performs only positive gain moves, we immediately revert a node move if it has negative attributed gain. Note that reverting such a node move does not guarantee to improve the connectivity metric again as other concurrent node moves may have changed the pin count values of the corresponding nets in the meantime. However, reverting them directly after detection decreases the likelihood of such conflicts. Furthermore, we use attributed gains to track the value of the connectivity metric instead of recomputing it after each round.

6.2 The Gain Table

For our FM algorithm, we use a gain table which stores and maintains the gain values for all possible moves. This enables repeatedly looking up gains in $O(1)$ time and is a globalized way of updating the gains of nodes owned by other threads. Gain tables are not a new idea [3, 80] but have gone "out of fashion" due to their memory requirements [4, 101]. To the best of our knowledge, our introduction of *parallel* gain tables is novel.

We use atomic `fetch-and-add` instructions to update the gains as soon as nodes are moved. Updates on some nodes become visible while the overall update procedure is still in flight. Therefore, updates *trickle in* over time, and some outdated or inconsistent values may be read by other threads. Still, with concurrent node moves this is the most accurate we can be.

Recall that the gain $g_u(V_t)$ of moving a node u to a target block V_t can be expressed as follows:

$$g_u(V_t) := \omega(\{e \in I(u) \mid \Phi(e, \Pi[u]) = 1\}) - \omega(\{e \in I(u) \mid \Phi(e, V_t) = 0\}).$$

The first term $b(u) := \omega(\{e \in I(u) \mid \Phi(e, \Pi[u]) = 1\})$ is the *benefit* of moving u out of its block. Conversely, the term $p(u, V_t) := \omega(\{e \in I(u) \mid \Phi(e, V_t) = 0\})$ is the *penalty* for moving u into V_t.

Update Rules. Instead of storing $g_u(V_i)$, we store $b(u)$ and $p(u, V_i)$ separately for each node u, so that changes to $b(u)$ only require one update, instead of updates to k gain values. This approach uses $(k+1)n$ memory words in total. For each net $e \in I(u)$, we update $b(u)$ and $p(u, V_i)$ using atomic `fetch-and-add` instructions as follows.

$$\textbf{If } \Phi(e, V_s) = 0 \textbf{ then } \forall v \in e \qquad \textbf{do } p(v, V_s) \overset{\text{atomic}}{+=} \omega(e) \tag{1}$$

$$\textbf{If } \Phi(e, V_s) = 1 \textbf{ then } \forall v \in e \cap V_s \textbf{ do } b(v) \qquad \overset{\text{atomic}}{+=} \omega(e) \tag{2}$$

$$\textbf{If } \Phi(e, V_t) = 1 \textbf{ then } \forall v \in e \qquad \textbf{do } p(v, V_t) \overset{\text{atomic}}{-=} \omega(e) \tag{3}$$

$$\textbf{If } \Phi(e, V_t) = 2 \textbf{ then } \forall v \in e \cap V_t \textbf{ do } b(v) \qquad \overset{\text{atomic}}{-=} \omega(e) \tag{4}$$

The update conditions implement the `UpdateGainTable` procedure from Line 9 in Algorithm 6.1.

Benefit Pecularities. There is a race condition on $\Pi[v]$ in the check $\Pi[v] = V_s$ (case 2) or $\Pi[v] = V_t$ (case 4). When $\Pi[v]$ changes, we may perform a benefit update on v that was also intended for a different pin of e in the new $\Pi[v]$. The penalty values are not affected since they are independent of the pin's current block. Our FM algorithm is organized in rounds in which each node can be moved at most once. Therefore, once u gets moved, we do not read $b(u)$ for the rest of the round. Due to the race condition it may still be updated, which is why we recalculate $b(u)$ *after* the round is finished instead of recalculating $b(u)$ for the new block *immediately* after the move. We note that it is possible to correctly update benefits by using k benefit values per node [49].

Correctness and Complexity of Gain Updates. In the following, we prove that once all updates for a given set of moves are completed and no further moves are performed, the gain values are correct.

Lemma 6.1. *After performing all gain updates associated with a set of moves M in parallel, each unmoved node $v \in V \setminus M$ has correct $b(v)$, and each $v \in V$ has correct $p(v, V_i)$ terms.*

Proof. First, we note that the updates are correct in the sequential setting [100]. Due to the atomic consistency of pin-count and gain updates, it suffices to prove correctness for arbitrary linearized (sequential) orders of updates. The remaining difficulty is that different orders may yield different intermediate values. However, due to commutativity we arrive at the same final $\Phi(e, V_i)$ values. Thus, it suffices to argue that gain updates triggered by $\Phi(e, V_i)$ += 1 cancel out those triggered by $\Phi(e, V_i)$ −= 1. This statement holds, as case 1 and 3 are complimentary, as well as case 2 and 4. Therefore, the final $p(v, V_i)$ and $b(v)$ values only depend on the final $\Phi(e, V_i)$ values. □

Lemma 6.2 (Sanchis [100]). *The work of gain updates for moving all nodes once is $O(\sum_{e \in E} |e| \cdot \min(k, |e|)) = O(kp)$.*

The core to the argument is that each of the update cases is only triggered a constant number of times per hyperedge and block [37, 100], and costs $O(|e|)$ work per update. This hinges on moving each node at most once. Note that due to the $\min(k, |e|)$ term, this bound matches the $O(m)$ bound on plain graphs. On real-word hypergraphs, we observed work much closer to $O(p)$ since most nets have small size or few pins per block.

6.3 The Parallel Gain Recalculation Algorithm

We now propose a parallel algorithm to recompute exact gain values of a sequence of node moves $M = \langle m_1, \ldots, m_l \rangle$ if they are supposed to be performed in this order. Each move $m_i \in M$ is of the form $m_i = (u, V_s, V_t)$, which means that node u is moved from block V_s to V_t. Again, we assume that each node is moved at most once. Recall that a move of a node u from block V_s to V_t decreases the connectivity of a hyperedge e, if $\Phi(e, V_s)$ decreases to zero. Conversely, it increases the connectivity if $\Phi(e, V_t)$ increases to one. The idea of the following algorithm is to iterate over the hyperedges in parallel, and identify the node moves in M that increase or decrease the connectivity of a hyperedge using Algorithm 6.2.

Consider a hyperedge e and a block $V_i \in \Pi$. The first observation is that if we move a pin $v \in e$ to V_i, then $\Phi(e, V_i)$ cannot decrease to zero anymore since each node is moved at most once. In order to decrease $\Phi(e, V_i)$ to zero, we have to move all pins $u \in e \cap V_i$ out of block V_i before we move the first pin $v \in e \setminus V_i$ to block V_i. In this case, the last pin $u \in e$ moved out of block V_i decreases the connectivity of e and the first pin $v \in e$ moved to block V_i increases its connectivity again. Thus, we can decide whether or not a move increases or decreases the connectivity of a hyperedge by simply comparing the indices of the node moves in M, which were last moved out and first moved to a particular block. Additionally, we need to know if the move sequence M moves all pins out of block V_i. To do so, we count the number of non-moved pins $v \in e$ in each block. If the number of non-moved pins is zero for a block V_i, then either $\Phi(e, V_i)$ was zero before, or the move sequence M moved all nodes out of block V_i.

Algorithm 6.2 shows the pseudocode that identifies the node moves in M that increase or decrease the connectivity of a hyperedge e. The algorithm uses two loops, both iterating over the pins of hyperedge e. The first loop computes the indices of the node moves that first moved to and last moved out of each block $V_i \in \Pi$ (see Line 6), in addition to the number of pins in e that were not moved (see Line 7).

The second loop then decides for each moved pin $u \in e$ whether or not it increases or decreases the connectivity of hyperedge e by evaluating the conditions shown in Lines 11 and 12.

ALGORITHM 6.2: Parallel Gain Recalculation

Input: Hyperedge e, a sequence of node moves $M = \langle m_1, \ldots, m_l \rangle$ and a shared gain vector $\mathcal{G} = \langle g_1, \ldots, g_l \rangle$ representing the recalculated gain values
Output: Updated gain values $\mathcal{G} = \langle g_1, \ldots, g_l \rangle$

```
1  first_in ← [∞, ..., ∞];   last_out ← [−∞, ..., −∞]                    // Arrays of size k
2  non_moved ← [0, ..., 0]                                                // Array of size k
3  for u ∈ e do
4      if u was moved then                                                // moved nodes are marked in a bitset
5          m_i := (u, V_s, V_t) ← find corresponding move in M
6          last_out[V_s] ← max(i, last_out[V_s]);   first_in[V_t] ← min(i, first_in[V_t])
7      else ++non_moved[Π[u]]
8  for u ∈ e do
9      if u was moved then
10         m_i := (u, V_s, V_t) ← find corresponding move in M
11         if last_out[V_s] = i ∧ i < first_in[V_s] ∧ non_moved[V_s] = 0 then g_i atomic+= ω(e)
12         if first_in[V_t] = i ∧ i > last_out[V_t] ∧ non_moved[V_t] = 0 then g_i atomic−= ω(e)
```

Let $m_i := (u, V_s, V_t)$ be the corresponding node move of pin $u \in e$ in M. If M moves all nodes out of block V_s ($\texttt{non\_moved}[V_s] = 0$) and u is the last pin moved out of block V_s ($\texttt{last\_out}[V_s] = i$), while the first move that moves a pin into block V_s happens strictly after m_i ($i < \texttt{first\_in}[V_s]$), then m_i reduces the connectivity metric by $\omega(e)$. Conversely, if M moves all nodes out of block V_t ($\texttt{non\_moved}[V_t] = 0$) and u is the first pin moved into block V_t ($\texttt{first\_in}[V_t] = i$), while the last move that moves a pin out of block V_t happens strictly before m_i ($i > \texttt{last\_out}[V_t]$), then m_i increases the connectivity metric by $\omega(e)$. Since we run the algorithm for each hyperedge in parallel, several threads can modify the gain value g_i of a node move m_i simultaneously. We therefore use atomic `fetch-and-add` instructions (see Line 11 and 12).

To further reduce the complexity of the algorithm, we only process hyperedges containing moved nodes. To do so, we iterate over the node moves in M in parallel and run Algorithm 6.2 only for incident edges of moved nodes. We mark already processed hyperedges in a shared bitset using atomic `test-and-set` instructions.

7 THE FIDUCCIA-MATTHEYSES ALGORITHM

The *Fiduccia-Mattheyses (FM)* algorithm [37] is the most widely used local search algorithm in sequential partitioning algorithms. Most of the existing variants insert all possible moves or only the highest gain move for each boundary node into a *priority queue (PQ)* and then perform the following two steps: (i) repeatedly perform the highest gain move subject to the balance constraint, followed by (ii) reverting moves back to the prefix with the highest cumulative gain in the sequence of performed moves. The revert is necessary, since moves with negative gains are allowed, so the algorithm is able to escape from local minima. Unfortunately, calculating the same move sequence as FM is P-hard [103], i.e., it is unlikely that a parallel algorithm with poly-log depth exists.

Sanders and Schulz [101] proposed a relaxed version that inserts only the highest gain move for a single seed node into a PQ and then gradually expands around the node by claiming neighbors of the moved node (localized FM). The algorithm not only produces better solutions than boundary FM [52], it is also highly amenable to parallelization as multiple FM searches can run in parallel, each starting from a different seed node. In the following, we present our parallel implementation of the localized FM algorithm, and discuss its main differences to an existing parallelization [4].

The Parallel k-Way FM Algorithm. Algorithm 7.1 shows the pseudocode of our parallel FM algorithm. The algorithm proceeds in rounds, and each round starts with inserting all boundary nodes into a globally shared task queue Q. The threads then poll a fixed number of nodes ($= 25$) from Q that they use as seed nodes for the localized FM searches, which expand to neighbors of moved nodes.

The searches are non-overlapping, i.e., threads acquire exclusive ownership of nodes, while hyperedges can touch multiple searches. Node moves performed by the different searches are not visible to other threads, as they are performed locally using thread-local hash tables. However, once a thread finds an improvement, it immediately applies it to the global partition. The local moves are atomically appended to a global move sequence (using one atomic `fetch-and-add` for all local moves). We repeatedly start localized FM searches until the task queue is empty. Note that we initialize the searches with multiple seed nodes instead of a single node as this substantially accelerates the algorithm in practice without sacrifices in solution quality.

Once the task queue is empty, we proceed to the second phase, where we recalculate the gains of the global move sequence (see Section 6.3) and then use a parallel prefix sum and reduce operation on the recomputed gain values to identify and revert to the best seen solution. We perform multiple rounds until a maximum number is reached or the connectivity metric is not improved.

Localized k-Way FM Search. The localized FM search uses a single PQ storing the move with the highest gain for each inserted node. We initialize the PQ with several seed nodes and use the gain table to compute the initial best move for each node (see Line 10). Then, we repeatedly select the move with the highest gain and apply it to a thread-local partition $\Delta\Pi$. Changes on $\Delta\Pi$ are not visible to other threads for now. However, we apply the move sequence to the global partition Π as soon as we find an improvement (see Line 18), then triggering gain updates in the global gain table.

When we move a node u locally, we collect the nets $e \in I(u)$ affected by gain updates. We use them to update the gain values of nodes in the PQ – combining global gain table and $\Delta\Pi$ data, thus gradually infusing updates from other threads into the search – and expand the search to neighbors of moved nodes. A localized search terminates when the PQ becomes empty or the adaptive stopping rule of Osipov and Sanders [3, 93] is triggered. The stopping rule assumes that the observed gain values follow a normal distribution and terminates a search when it becomes unlikely to find further improvements. We release the ownership of non-moved nodes at the end such that other searches can acquire them again. We do not release the ownership of moved nodes to ensure that each node is moved at most once during an FM pass.

We explicitly allow moves with negative gains, which will worsen the solution quality intermediately. At the end of each localized search, we thus revert back to the best seen solution. If we directly applied moves to the global partition, other searches could base their decisions on states that will later be reverted. Therefore, we apply node moves to a thread-local partition $\Delta\Pi$ first and only perform them on the global partition if they lead to an improvement, i.e., will not be reverted for now.

The thread-local partition $\Delta\Pi$ stores changes relative to the global partition in a set of hash tables. For example, we compute the weight of a block V_i by calculating $c(V_i) + \Delta c(V_i)$ where $c(V_i)$ is the weight of block V_i stored in the global partition data structure and $\Delta c(V_i)$ is the weight of all nodes that locally moved to block V_i minus the weight of nodes that moved out of block V_i. We maintain the block ID, pin count values, as well as benefit and penalty terms of the gain table analogously.

Applying a move sequence to the global partition makes it immediately visible to the searches on other threads. Since $\Delta\Pi$ stores local changes relative to the global partition, the block weights and pin count values are still correct. However, some gain values may be incorrect since the gain table

ALGORITHM 7.1: Parallel k-Way FM Algorithm

```
1  Function FMRefinement(Hypergraph H, k-way partition Π)
2      while improvement found and maximum number of rounds not reached do
3          Q ← initialize task queue with all boundary nodes
4          while Q not empty do in parallel
5              V_seed ← poll 25 seed nodes from Q
6              LocalizedFMRefinement(H, Π, V_seed)
7          recompute gains of global move sequence and revert to best prefix        // see Section 6.3
8
9  Function LocalizedFMRefinement(H, Π, V_seed)
10     for u ∈ V_seed do                                                  // Initialize PQ with seed nodes
11         (g_t, V_t) ← ComputeMaxGainMove(u);   PQ.Insert(u, g_t, V_t)
12     Δ ← 0;   M ← ∅
13     while PQ not empty and search should continue do
14         (u, g_t, V_t) ← PQ.PopMaxGainMove()
15         move u to V_t in thread-local partition ΔΠ with gain table update
16         Δ ← Δ + g_t;   M ← M ∪ {(u, V_t)}
17         if Δ > 0 or (Δ = 0 and move improved balance) then
18             apply move sequence M to global partition Π
19             Δ ← 0;   M ← ∅;   ΔΠ ← Π
20         for all nets e ∈ I(u) involved in a gain update do
21             for v ∈ e do
22                 if v is not marked then
23                     if PQ.Contains(v) then update gain of v in PQ
24                     else if try to acquire node v then
25                         (g_t, V_t) ← ComputeMaxGainMove(v);   PQ.Insert(v, g_t, V_t)
26                     mark v
27         unmark all nodes
```

updates on the global partition do not consider moves performed locally. This is only a small issue since thread-local deltas are cleared after applying the moves to the global partition. In practice, the scheme drastically reduces conflicts.[2] Another reason for applying moves as soon as possible is to keep the memory footprint of the hash tables small. The overall peak memory incurred by thread-local partition data is small, because the memory is proportional to the number of moves, and long-running searches must find improvements to keep going.

Differences to Mt-KaHIP. The FM implementation in Mt-KaHIP [4] performs node moves only locally, which are therefore *not visible to other threads*. At the end of an FM pass, the move sequences found by the different searches are concatenated to a global move sequence, for which *gains are recomputed sequentially*. We improved the algorithm by making improvements immediately visible to other threads using the thread-local partition and gain table data structure, leading to more accurate gain values. Moreover, we removed the last sequential part of the algorithm with our parallel gain recomputation technique.

[2]We have found that the recomputed gain values of the global move sequence match the observed gain values during the localized FM searches in most cases.

ALGORITHM 8.1: Parallel Flow-Based Refinement

```
Input: Hypergraph H = (V, E, c, ω) and k-way partition Π of H
1 Q ← BuildQuotientGraph(H, Π)                                  // see Section 8.1
2 while ∃ active (V_i, V_j) ∈ Q do in parallel                  // see Section 8.1
3     B ← ConstructRegion(H, V_i, V_j)                          // see Section 8.2
4     (H, s, t) ← ConstructFlowNetwork(H, B)                    // see Section 8.2
5     (M, Δ_exp) ← FlowCutterRefinement(H, s, t)                // see Section 8.3–8.4
6     if Δ_exp ≥ 0 then                                         // potential improvement
7         Δ ← ApplyMoves(H, Π, M)                               // see Section 8.1
8         if Δ > 0 then mark V_i and V_j as active              // found improvement
9         else if Δ < 0 then RevertMoves(H, Π, M)               // no improvement
```

8 FLOW-BASED REFINEMENT

A major shortcoming of move-based local search algorithms is that they greedily move nodes to other blocks based on a gain value considering only the block assignment of adjacent nodes. Thus, the decision to apply a move depends only on *local* information, which may not be sufficient to find some non-trivial improvements [99]. Maximum flows overcome this limitation by deriving a minimum cut separating two nodes [38] and therefore have a more *global* view on the partitioning problem. Although it seems natural to use them as local search strategy in partitioning algorithms, maximum flows were long perceived as computationally expensive and it was unclear how to derive balanced partitions [76, 124]. This changed over the last two decades as flow-based refinement techniques were successfully implemented in the highest-quality sequential graph and hypergraph partitioning algorithms [44, 59, 62, 101, 106]. Today they are considered to be the most powerful improvement heuristics for (hyper)graph partitioning. However, since they come at the cost of substantially higher running times, they can be impractical for partitioning very large hypergraphs.

Algorithm Overview. In this section, we present the first parallel formulation of the sequential flow-based refinement approach used in KaHyPar [44, 62]. The high-level pseudocode of the algorithm is outlined in Algorithm 8.1. Flow-based refinement works on bipartitions and can be scheduled on different block pairs to improve k-way partitions [44, 62, 101]. We therefore start with a parallel scheduling scheme of adjacent block pairs based on the quotient graph in Section 8.1 (see Line 1 and 2). In Section 8.2, we describe the flow network construction algorithm that extracts a subhypergraph induced by a region $B \subseteq V$ around the boundary nodes of two adjacent blocks, which then yields a flow network (see Line 3 and 4). On each network, we run the FlowCutter algorithm [53, 124] to derive a balanced minimum cut using incremental maximum flow computations. FlowCutter and its parallelization are discussed in Sections 8.3 and 8.4. We then convert the minimum cut into a set of moves M and an expected connectivity reduction Δ_{exp}. If FlowCutter claims an improvement, i.e., if $\Delta_{\text{exp}} \geq 0$, we apply the moves to the global partition and recompute the reduction Δ. Based on Δ we either schedule the blocks for further refinement, or revert the moves (see Line 8 and 9).

Maximum Flows. A flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ is a directed graph with a dedicated source node $s \in \mathcal{V}$ and sink node $t \in \mathcal{V}$ in which each edge $e \in \mathcal{E}$ has capacity $c(e) \geq 0$, and each non-edge $(u, v) \in (\mathcal{V} \times \mathcal{V}) \setminus \mathcal{E}$ has capacity $c(u, v) = 0$. 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 amount 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 $r_f(e) = 0$. The *residual network* $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f, r_f)$ with $\mathcal{E}_f := \{(u,v) \in \mathcal{V} \times \mathcal{V} \mid r_f(u,v) > 0\}$ contains all non-saturated edges. 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 [38]. This is also called a *minimum* (s,t)*-cut*. The minimum (s,t)-cut can be derived by exploring the nodes reachable from the source or sink via residual edges ($r_f(e) > 0$), which is also called the *source-side* or *sink-side cut*.

8.1 Parallel Active Block Scheduling

Sanders and Schulz [101] propose the active block scheduling strategy 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 finds an improvement, both are marked as active for the next round.

Parallelization. Our parallel implementation schedules multiple flow computations on adjacent block pairs in parallel. 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. 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 and the parameter τ controls the available parallelism in the scheduler. With higher values of τ, more block pairs are scheduled in parallel, which can lead to more interferences between searches that operate on overlapping regions. Threads that are not involved in scheduling can join parallel flow computations. In a parameter study [60, p. 108], we found that $\tau = 1$ offers a good trade-off between conflicting searches and scalability.

Initially, we push all pairs of adjacent blocks into a concurrent FIFO queue A. The threads then poll from A and if a search finds an improvement on a block pair (V_i, V_j), we mark both as active using a separate bitset for each round. If either V_i or V_j becomes active, we push all adjacent blocks into A if they are not contained yet. 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 [101]. 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.

Apply Moves. Since concurrently scheduled flow computations can operate on overlapping regions, there are three conflict types that can occur when applying a sequence of node moves M to the global partition Π: balance constraint violations, $\Delta_{\text{exp}} \neq \Delta$ (i.e., the expected does not match the actual connectivity reduction), and nodes in M may already be moved by other searches.

In practice, the running time to apply a sequence of node moves is negligible compared to solving flow problems [60, see Figure 5.21 on p. 119]. Thus, we can afford to use a lock so that only one thread applies moves at a time to address these conflicts. First, we remove all nodes from M that are not in their expected block, i.e., they were moved by a different search in the meantime. Afterwards, we compute the block weights as if all remaining moves were applied. If the resulting partition is balanced, we perform the moves, during which we aggregate the attributed gains Δ of each move. If $\Delta < 0$, we revert all moves.

8.2 Flow Network Construction

To improve the cut of a bipartition $\Pi = \{V_1, V_2\}$, we grow a size-constrained 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)* [101].

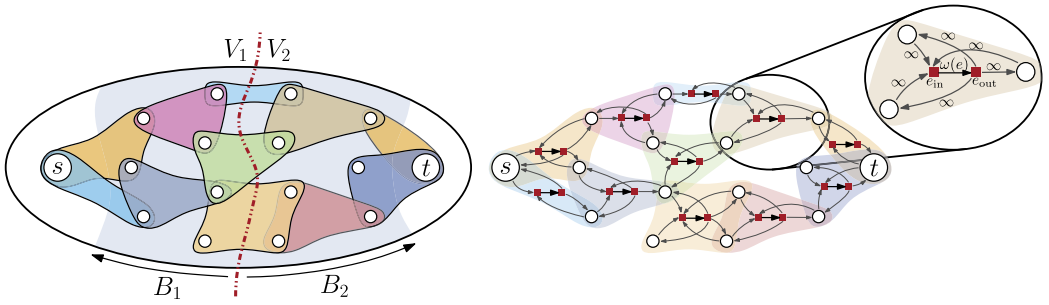

Fig. 5. A hypergraph $\mathcal{H}$ (left) induced by a region $B := B_1 \cup B_2$ and the flow network $\mathcal{N}$ (right) given by the Lawler expansion of $\mathcal{H}$.

The first BFS is initialized with all boundary nodes of block V_1 and continues to add nodes to B_1 as long as $c(B_1) \leq (1 + \alpha\varepsilon)\lceil \frac{c(V)}{2} \rceil - c(V_2)$, where α is an input parameter. The second BFS that constructs B_2 proceeds analogously. We then contract all nodes in $V_1 \setminus B$ to the source s and $V_2 \setminus B$ to the sink t [45, 101] and obtain a coarser hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. The flow network $\mathcal{N}$ is then given by the Lawler expansion of $\mathcal{H}$ [84], which is illustrated in Figure 5. For each hyperedge $e \in \mathcal{E}$, we add two nodes e_{in} and e_{out} and a *bridging edge* $(e_{\text{in}}, e_{\text{out}})$ with capacity $c(e_{\text{in}}, e_{\text{out}}) = \omega(e)$ to $\mathcal{N}$. For each pin $u \in e$, we add two edges (u, e_{in}) and (e_{out}, u) with infinite capacity to $\mathcal{N}$. Note that we do not construct $\mathcal{N}$ explicitly in our actual implementation, since our maximum flow algorithm runs on $\mathcal{H}$ by implicitly exploiting the structure of the Lawler expansion.

The parameter α controls the size of the flow network. 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 ($c(B_1) + c(V_2) \leq (1 + \varepsilon)\lceil \frac{c(V)}{2} \rceil$ and vice versa for block B_2). 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 cut computations (see Section 8.3). In practice, we use $\alpha = 16$ (also used in KaHyPar [44, 62]). 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.

8.3 The FlowCutter Algorithm

In this section, we discuss the flow-based refinement on a bipartition. We introduce the aforementioned FlowCutter algorithm [53, 124], the parallelization of which is described in the next section. To speed up convergence and make parallelism worthwhile, we propose an optimization named *bulk piercing*.

Algorithm Overview. FlowCutter solves a sequence of incremental maximum flow problems until a balanced bipartition is found. Algorithm 8.2 shows the 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. As this would yield the same cut in the next iteration, we add one additional node, called *piercing node*, to the terminal set of the smaller side. Thus, the bipartitions contributed

ALGORITHM 8.2: The FlowCutter Algorithm

Input: Original hypergraph $H = (V, E, c, \omega)$, flow network $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and a source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$
Output: Balanced Bipartition of $\mathcal{H}$

```
1 S ← {s}, T ← {t}                                   // initialize source and sink set
2 initialize flow f : V × V → ℝ≥0 with ∀(u, v) ∈ V × V : f(u, v) = 0
3 while no balanced bipartition found do
4     f ← ParallelMaxPreflow(H, S, T, f)             // augment f to a maximum preflow
5     (S_r, T_r) ← derive source- and sink-side cut S_r, T_r ⊂ V
6     if (S_r, V \ S_r) is balanced then return (S_r, V \ S_r)
7     else if (V \ T_r, T_r) is balanced then return (V \ T_r, T_r)
8     if c(S_r) ≤ c(T_r) then S ← S_r ∪ selectPiercingNode(S ∪ S_r)
9     else T ← T_r ∪ selectPiercingNode(T ∪ T_r)
```

by the currently smaller side will be more balanced with a possibly larger cut in future iterations. Since the smaller side is grown, this process will converge to a balanced bipartition.

For our purpose, there are two important piercing node selection heuristics: *avoid augmenting paths* [53, 124] and *distance from cut* [44]. 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 [97]. As a secondary criterion, larger distances from the original cut are preferred, to reconstruct parts of it.

Bulk Piercing Optimization. On larger instances, piercing only one node per iteration converges slowly. We therefore increase the amount of work in each iteration by piercing multiple nodes, as long as we are far from balance.

To achieve a small number of iterations (e.g., poly-log) we set a goal on the weights of the sides of the bipartition, and pierce more aggressively the further we are from it. Assume we want to pierce the source side next and have already performed $r - 1$ piercing iterations on it. In the r-th iteration we want to add $\frac{1}{2^r} \cdot (\frac{c(V)}{2} - c(S))$ new weight to the source side, where $c(S)$ is the weight of the initial source-side terminals (before any piercing) and $\frac{c(V)}{2}$ is the weight of a perfectly balanced bipartition. Thus, the overall weight goal for the r-th iteration on the source side is set to $(\frac{c(V)}{2} - c(S)) \sum_{i=1}^{r} \frac{1}{2^i}$. This is chosen such that we allow a lot of progress early on and become more careful as we get closer to a balanced bipartition. We track the average weight added per node in previous iterations and from this estimate the number of required piercing nodes to reach the goal for the r-th iteration. To boost measurement accuracy, we pierce only one node for the first few rounds, and then switch to bulk piercing.

8.4 Parallel Maximum Flow Algorithm

Maximum flow algorithms are notoriously difficult to parallelize efficiently [9, 15, 67, 110]. The synchronous push-relabel approach of Baumstark et al. [15] is a recent algorithm that sticks closely to sequential FIFO and thus shows good results. We first describe the sequential push-relabel algorithm proposed by Goldberg and Tarjan [42] and then briefly outline its parallelization. We conclude with implementation details and intricacies of using FlowCutter with preflows.

Push-Relabel Algorithm. The *push-relabel* algorithm [42] 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* [74] which is a flow where the conservation constraint is replaced by $\text{exc}(u) \geq 0$. The distance labels represent a lower bound

for the distance of each node to the sink. 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. The *global relabeling* heuristic [29] frequently assigns exact distance labels by performing a reverse BFS from the sink to reduce relabel work in practice. Note that a maximum preflow already induces a minimum sink-side cut, so if only a minimum cut is required, the algorithm can already stop once no active nodes with distance label $< n$ exist.

The parallel push-relabel algorithm of Baumstark et al. [15] 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 $\text{exc}(v) > 0$ 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 [29], after linear push and relabel work, using parallel reverse BFS in the residual network. We additionally fixed an undocumented bug in the original algorithm (not source code) for which we refer the reader to Reference [46].

Intricacies with Preflows and FlowCutter. 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 [29] to push excesses back to the source. However, flow decomposition is difficult to parallelize [15]. Instead, we initialize the forward residual BFS with all active non-sink excess nodes. This finds the reverse paths that carry flow from the source to the excess nodes, which is what we need.

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.

Finally, we want to reuse the distance labels from the previous round 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. 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.

Implementation Details. Since $(e_{\text{in}}, e_{\text{out}})$ is the only outgoing edge of e_{in} with non-zero capacity in the Lawler expansion (see Figure 5), the flow on edges (u, e_{in}) is also bounded by $\omega(e)$ (instead of ∞). Adding these capacities is a trivial optimization, but significantly accelerates the algorithm and increases the available parallelism. This can be explained by the fact that a hypernode u does not immediately relieve all of its excess to one of its incident nets $e \in I(u)$ during the discharge routine, which is later pushed back due to the $\omega(e)$ bound. We set the capacities $c(u, e_{\text{in}})$ to ∞ again when deriving the source- and sink-side cut, since only bridging edges can be cut in the Lawler expansion.

Moreover, we observed that the number of active nodes follows a power-law distribution. Due to little work in later rounds, it takes many rounds to trigger the global relabeling step that also

ALGORITHM 9.1: The n-Level Partitioning Algorithm

Input: Hypergraph $H = (V, E)$, number of blocks k
Output: k-way partition Π of H

```
while V has too many nodes do
    for u ∈ V in random order do in parallel
        v ← arg max_v Σ_{e∈I(u)∩I(v)} ω(e)/(|e|−1)      // find best contraction partner for u
        if (v, u) is eligible for contraction then contract v onto u
Π ← InitialPartition(H, k)
B = ⟨B_1, ..., B_l⟩ ← ConstructBatches(F)
for B ∈ B do                                            // |B| ≈ b_max
    for v ∈ B do in parallel
        uncontract v from rep[v];   Π[v] ← Π[rep[v]]
    LabelPropagationRefinement(H, Π, B);   FMRefinement(H, Π, B)
```

terminates the algorithm when a maximum preflow is found. Therefore, 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.

9 A PARALLELIZATION OF THE n-LEVEL PARTITIONING SCHEME

Our multilevel algorithm contracts a clustering of highly-connected nodes on each level, which induces a hierarchy with a logarithmic number of levels. In contrast, KaHyPar [3, 104] – the currently best sequential partitioning algorithm with regards to solution quality – contracts only a single node on each level. Correspondingly, in each refinement step, only a single node is uncontracted followed by a highly-localized search for improvements around the uncontracted node. This technique produces almost n levels and is therefore known as the *n-level partitioning scheme*. More levels provide "more opportunities to refine the current solution" [8] at different granularities but also increase the running time of multilevel algorithms. Therefore, KaHyPar is the method of choice for computing high-quality partitions but comes at the cost of substantially higher running times than other systems – prohibitively so for very large hypergraphs. Although n-level partitioning seems inherently sequential, we present the first shared-memory parallelization of the technique which achieves good speedups and comparable solution quality to KaHyPar in a fraction of its running time.

We start this section with a formal definition of the (un)contraction operation, and provide a high-level overview of our n-level partitioning algorithm. We then discuss and present solutions for the main challenges in this algorithm: finding a parallel schedule of (un)contraction operations and performing them on a dynamic hypergraph data structure in parallel. We conclude the algorithm description with a discussion on how the refinement algorithms from the previous section are integrated into the n-level algorithm.

The Contraction and Uncontraction Operation. Contracting a node v onto another node u replaces v with u in all nets $e \in I(v) \setminus I(u)$ and removes v from all nets $e \in I(u) \cap I(v)$. The weight of node u is then $c(u) + c(v)$. We call u the *representative* of the contraction and v its *contraction partner*. Uncontracting a node v reverses the corresponding contraction operation.

Algorithm Overview. Algorithm 9.1 shows the high-level pseudocode of our n-level partitioning algorithm. In the coarsening phase, we iterate in parallel over all nodes and find the best contraction partner v for each node u using the heavy-edge rating function [3, 28, 69] (similar as in our

multilevel algorithm). We then check whether v can be contracted right away onto u or if there are any other pending contractions that must be performed before. In the latter case, we transfer the responsibility of contracting v onto u to the thread resolving the last dependency that defers the contraction. Once the hypergraph is small enough, we compute an initial partition into k blocks.

Uncontracting only a single node followed by a localized refinement step is inherently sequential, which is why we have to relax the n-level idea in the uncoarsening phase. We construct a sequence of batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$ of contracted nodes, such that $|B_i| \approx b_{\max}$ where $b_{\max}$ is an input parameter. Batches are processed one after another, enabling the uncontraction of nodes in subsequent batches. Nodes in the same batch are uncontracted in parallel. The main challenge is to identify which nodes can or even must appear in the same batch. After uncontracting each batch, we apply highly-localized refinement algorithms around the batched nodes.

A Forest-Based Scheduling of Contraction Operations. Let us consider a sequence of contractions $C := \langle (v_1, u_1), \ldots, (v_n, u_n) \rangle$ executed exactly in this order (v_i is contracted onto u_i). In this sequence, each node is contracted onto at most one representative, and there are no cyclic contraction dependencies. Thus, the sequence of contractions form a forest $\mathcal{F} := (V, C)$ if interpreted as a graph with directed edges $(v_i, u_i) \in C$. We will refer to $\mathcal{F}$ as the *contraction forest*. Our parallelization uses the observation that there exist several permutations of C leading to the same contraction forest $\mathcal{F}$. We can contract a node as soon as all of its children in $\mathcal{F}$ have been contracted. To obtain parallelism, different subtrees and siblings can be contracted independently, i.e., we traverse $\mathcal{F}$ in a bottom-up fashion in parallel.

In our actual algorithm, we do not know $\mathcal{F}$ in advance. However, we show how to construct $\mathcal{F}$ dynamically and from that we derive a parallel schedule of contraction operations. We call a contraction (v, u) *compatible* with existing contractions if it satisfies the following three conditions: (i) v must be a root of $\mathcal{F}$, (ii) adding (v, u) to $\mathcal{F}$ must not induce a cycle, and (iii) the contraction of u onto its parent in $\mathcal{F}$ must not have started yet. We represent $\mathcal{F}$ using an array rep of size n storing for each node its representative (u is a root if $\text{rep}[u] = u$). Additionally, we use a zero-initialized array pending, where $\text{pending}[u]$ stores the number of children of u whose contraction is not finished. If $\text{pending}[u] = 0$ and $\text{rep}[u] \neq u$, we assume that the contraction of u onto $\text{rep}[u]$ has started and prevent further contractions onto u. The entries $\text{rep}[u]$ and $\text{pending}[u]$ are only modified while holding a node-specific lock for u.

If we add a contraction (v, u) to $\mathcal{F}$, we first lock v and check if v is a root. If $\text{rep}[v] \neq v$, we discard the contraction as another thread has already selected a representative for v. Otherwise, we walk the path towards the root of u's tree in $\mathcal{F}$ to find the lowest ancestor w of u whose contraction has not started yet ($rep[w] = w$ or $\text{pending}[w] > 0$, in most cases $w = u$). If v is found on this path, the contraction is discarded, as it would induce a cycle in $\mathcal{F}$. If no cycle is found, we lock w, and check $\text{rep}[w]$ and $\text{pending}[w]$ again. If they changed, we find a new suitable ancestor and perform the cycle check again. Otherwise, we set $\text{rep}[u] = w$ and increment $\text{pending}[w]$ by one, and unlock v and w. We immediately contract v onto w if $\text{pending}[v] = 0$ and subsequently reduce $\text{pending}[w]$ by one. If this reduces $\text{pending}[w]$ to zero, we recursively apply this process to the contraction $(w, \text{rep}[w])$ if $\text{rep}[w] \neq w$.

Batch Uncontractions. For the uncoarsening phase, we construct a sequence of batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$ where each batch B_i contains roughly $b_{\max}$ contracted nodes that can be uncontracted independently in parallel. The batch size $b_{\max}$ is an input parameter (set to $b_{\max} = 1000$ in our implementation) that interpolates between scalability (high values) and the inherently sequential n-level scheme ($b_{\max} = 1$). Uncontracting a batch B_i resolves the last dependencies required to uncontract the next batch B_{i+1}. After each batch uncontraction, we apply a highly-localized version

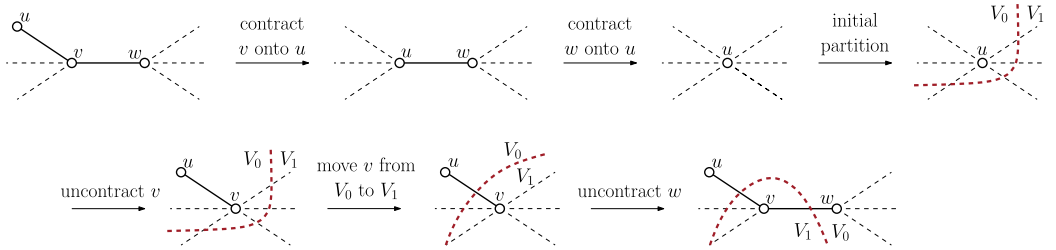

Fig. 6. Uncontracting w increases the cut by one since we do not uncontract v and w in reverse order.

of the label propagation and FM algorithm searching for improvements in a small region around the uncontracted nodes.

We construct the batches via a top-down traversal of the contraction forest $\mathcal{F}$. There are two constraints that we need to consider when constructing the batches: (i) a node must appear in a batch strictly after the batch containing its representative, and (ii) siblings in $\mathcal{F}$ must be uncontracted in reverse order of contraction. The second condition prevents uncontractions increasing the cut size as illustrated in Figure 6, which would violate a fundamental property of the multilevel scheme. Since contractions can be performed at the same time, it is often not possible to define a strict order in which we have to revert the contractions. We therefore associate each contraction (v, u) with a time interval $[s_v, e_v]$ by atomically incrementing a counter before starting (s_v) and after finishing (e_v) a contraction operation. If the time interval of two nodes overlap, we assume they were contracted at the same time, otherwise one is strictly earlier than the other. Among siblings, we compute the transitive closure of nodes with overlapping time intervals and order them decreasingly if one is strictly earlier than the other. We then use this as the reverse order of contractions, while we add siblings with overlapping time intervals to the same batch. As this is only a high-level description of the batch construction algorithm, we refer the reader to Reference [60, p.129–131] for more details.

The Dynamic Hypergraph Data Structure. Figure 7 illustrates the dynamic hypergraph data structure that stores the pin-list of each net e and the incident nets $I(u)$ of each node u using two separate adjacency arrays. When we contract a node v onto another node u, we iterate over the incident nets of v and search for u and v in the pin-list of each net $e \in I(v)$. If we do not find u in e, we replace v with u ($e \in I(v) \setminus I(u)$). Otherwise, we swap v to the end of e's pin-list and decrement the size of e by one ($e \in I(u) \cap I(v)$), dividing its pin-list into an *active* and *inactive* part. We use a separate lock for each net to synchronize edits to the pin-lists. We further mark the nets $e \in I(u) \cap I(v)$ in a bitset X, which we then use to update the incident nets of u.

The key idea for updating the incident nets is to remove $I(u) \cap I(v)$ from $I(v)$ and concatenate u and v in a doubly-linked list L_u. All nodes contracted onto u are then stored in L_u. We can then iterate over the incident nets of u by iterating over all entries $w \in L_u$ and the modified $I(w)$ arrays. We associate each $I(w)$ array with a counter t_w and each entry $e \in I(w)$ with a marker $t_{w,e}$ (initially set to zero). Entries with markers $\geq t_w$ are *active*, i.e., were not removed yet. For removing $I(u) \cap I(v)$ from $I(v)$ (marked in bitset X), we iterate over all nodes $w \in L_v$ and increment t_w by one. We then iterate over the previously active entries of $I(w)$ (now marked with $t_w - 1$) and if an entry is not in X, we set its marker to t_w. Otherwise, we swap the entry to the end of the active part but keeping its marker at $t_w - 1$. A simpler approach would be to represent the incident nets of each node as an adjacency list and add $I(v) \setminus I(u)$ to $I(u)$, as it is done in KaHyPar [104, 105]. However, this could lead to quadratic memory usage, and is therefore not practical for large hypergraphs.

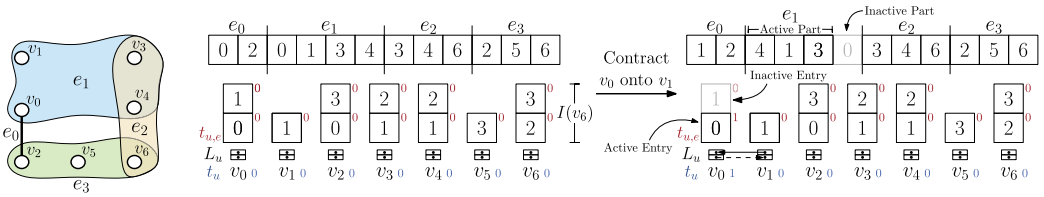

Fig. 7. Contraction operation applied on the dynamic hypergraph data structure.

When uncontracting a node v from its representative u, we first restore L_v from L_u. To do this, we additionally store the last node in L_v at the time v is contracted onto u. We then iterate over all nodes $w \in L_v$ and decrement their counters t_w by one. This reactivates all nets $e \in I(w)$ that became inactive due to contracting v, i.e., were part of $I(u) \cap I(v)$ before the contraction. In these nets, we swap v to the active part of their pin-lists again. All previously active nets $e \in I(w)$ (now marked with $t_{w,e} > t_w$) were part of $I(v) \setminus I(u)$ before the contraction in which we then replace u with v again. Note that we sort all pins in the inactive part of a pin-list by the batches in which they are uncontracted. Then, all pins of a net e part of the current batch can be restored simultaneously by appropriately incrementing the size of e. Only one thread that triggers the restore case on a net performs the restore operation, which we ensure with an atomic `test-and-set` instruction.

Removing Single-Pin and Identical Nets. We remove single-pin nets and aggregate the weight of all identical nets at one representative after a pass over all nodes in the coarsening phase using the same algorithm as already described in Section 4. This adds several synchronization points ($\approx \log n$) at which we have to restore them in the uncoarsening phase. KaHyPar [104, 105] removes these nets directly after each contraction operation. However, doing this in the parallel setting would introduce additional dependencies for batches, which is why we decided against it.

Refinement. After uncontracting a batch, we run a highly-localized version of label propagation and FM refinement initialized with the boundary nodes of the current batch. The searches then expand to a small region around the uncontracted nodes. We complement the localized refinement with a refinement pass on the entire hypergraph after restoring single-pin and identical nets. Here, we run FM (initialized with all boundary nodes) and flow-based refinement. Additionally, we implemented a concurrent gain table update procedure for batch uncontractions for which we refer the reader to Reference [60, p.132–133] for more details.

10 UNIFYING HYPERGRAPH AND GRAPH PARTITIONING

Hypergraph partitioning (HGP) is considered "inherently more complicated" [75] and therefore more complex "in terms of implementation and running time" [21] than *graph partitioning (GP)*. However, the high-level description of partitioning algorithms often does not reveal any difference between the two. For example, label propagation refinement iterates over all nodes, and moves each node to the block with the highest gain value. While the algorithm is widely used in GP and HGP, the main difference in its implementation lies in the representation of the graph data structure and the computation of gain values. GP tools build on data structures using *one* adjacency array to represent the neighbors of nodes, while HGP requires *two* adjacency arrays storing the pin-lists of hyperedges and the incident nets of nodes. This results in a better cache utilization and faster access times for graph algorithms. Moreover, the gain value of a node move for the edge cut metric depends on the block assignments of neighbors for GP, while HGP tools have to maintain or compute the pin count values of hyperedges to decide whether or not it can be removed from the cut. In the following, we present an optimized graph and partition data structure

for GP implementing the interface of our hypergraph data structure such that we can use them as a drop-in replacement in our partitioning algorithm. We focus on the multilevel algorithm and refer the reader to Reference [60, p.150–153] for a description of the n-level graph data structure.

Terminology. An undirected and weighted graph $G = (V, E, c, \omega)$ can be considered as a hypergraph where each net contains only two pins (also called an *edge*). Therefore, the definitions and notations for hypergraphs also apply to undirected graphs. We define the weight of an edge $e = \{u, v\} \in E$ as $\omega(u, v) := \omega(e)$. If $\{u, v\} \notin E$, then $\omega(u, v) = 0$. An edge $\{u, u\} \in E$ is called a *selfloop*. For a subset $V' \subseteq V$, $\omega(u, V') := \sum_{v \in V'} \omega(u, v)$ is the weight of all edges connecting node u to V'.

10.1 Graph Data Structure

We use one adjacency array to represent an undirected graph. As the algorithms are implemented for hypergraphs, we have to support iteration over the pin-list of an edge. For an edge $e = (u, v)$, the slot for e in the range of edges incident to u thus stores both the source node u and the target node v. This data structure requires twice as much memory as traditional adjacency arrays which only need to store the target node.

Contraction. Our multilevel partitioning algorithm contracts a clustering of the nodes on each level. The coarsening algorithm stores the clustering in an array rep where $\text{rep}[u] = v$ stores the representative of u's cluster. For each representative v, we maintain the invariant that $\text{rep}[v] = v$.

The contraction algorithm first remaps cluster IDs to a consecutive range by computing a parallel prefix sum on an array of size n that has a one at position v if v is a representative of a cluster and zero otherwise. Then, we accumulate the weights and degrees of nodes in each cluster using atomic `fetch-and-add` instructions. Afterwards, we copy the incident edges of each cluster to a consecutive range in a temporary adjacency array by computing a parallel prefix sum over the cluster degrees. We then iterate over the adjacency lists of each cluster in parallel, sort them, and remove selfloops and identical edges except for one representative at which we aggregate their weights. Finally, we construct the adjacency array of the coarse graph by computing a parallel prefix sum over the remaining cluster degrees.

10.2 The Partition Data Structure

For hypergraphs, our partition data structure stores the block assignments Π, the block weights $c(V_i)$, the pin count values $\Phi(e, V_i)$, and connectivity sets $\Lambda(e)$ for each net $e \in E$ and block $V_i \in \Pi$. Since a graph edge connects only two nodes, we can remove the pin count values and connectivity sets as we can calculate them on-the-fly. However, we used the synchronized writes to the pin count values to update the gain table and compute the attributed gain values. We therefore present alternative approaches for both techniques that exploit the properties of graphs.

The Gain Table. The connectivity metric reverts to the edge cut metric for plain graphs (since $\lambda(e) \leq |e| = 2$). The gain value of moving a node u to another block V_t is then defined as $g_u(V_t) := \omega(u, V_t) - \omega(u, \Pi[u])$ (external minus internal edges). Thus, the gain table for graph partitioning stores and maintains the $\omega(u, V_i)$ values for each node $u \in V$ and block $V_i \in \Pi$ ($n \cdot k$ entries). If we move a node u from block V_s to V_t, we update the gain table by adding $\omega(u, v)$ to $\omega(v, V_t)$ and $-\omega(u, v)$ to $\omega(v, V_s)$ for each neighbor $v \in \Gamma(u)$ using atomic `fetch-and-add` instructions. Hence, the complexity of the gain table updates when each node is moved at most once is $\sum_{u \in V} 2d(u) = O(m)$.

Attributed Gains. For a node u moved from block V_s to V_t, we attribute a connectivity reduction or increase by $\omega(e)$ to each net $e \in I(u)$ based on the synchronized writes to $\Phi(e, V_s)$ and $\Phi(e, V_t)$. Since we do not longer maintain the pin count values, we need another synchronization

mechanism to decide if a node move removes an edge from the cut or makes it a cut edge, and based on that attribute a reduction or an increase by the weight of the edge to the move.

We therefore use an array B of size m (initialized with $\perp$) to synchronize the node moves for each edge. To compute the attributed gain of a node move u from block V_s to V_t, we iterate over all incident edges $e = \{u, v\} \in I(u)$ and write the target block V_t of u to $B[e]$ using an atomic `compare-and-swap` operation. If the operation succeeds, no other thread has moved its neighbor v yet. In this case, e becomes an internal edge if $V_t = \Pi[v]$ (reduces the edge cut by $\omega(e)$) and a cut edge otherwise (increases the edge cut by $\omega(e)$). If we do not succeed in setting $B[e]$ from $\perp$ to V_t, another thread has already moved or is currently moving v to another block. In both cases, its target block is $B[e]$ and we can compute the attributed gain value for edge e as before by comparing V_t and $B[e]$. After calculating the attributed gain value for each net $e \in I(u)$, we set the block ID of u to V_t. Note that the algorithm only works when each node is moved at most once, as it is done in our refinement algorithms ($B[e]$ values are reset to $\perp$ after each refinement round).

11 DETERMINISTIC PARTITIONING

A program is *externally deterministic* [17] if, given the same input, it produces the same output, each time it is run. Sequential programs are usually deterministic by default, whereas parallel programs are non-deterministic by default due to randomness in scheduling. Yet, researchers have advocated the benefits of deterministic parallel programs for several decades [19, 85, 112]. It is easier to debug the program, to reason about performance and it yields reproducible results: in experiments and applications. Unfortunately, with the exception of BiPart [88], all published parallel partitioning algorithms so far are non-deterministic. This stems from concurrently performed moves affecting other ongoing move decisions.

In this section, we present deterministic versions for a subset of the components in our multilevel framework: label propagation refinement, heavy-edge clustering for coarsening, and the Louvain community detection method [18] (optimizing the popular modularity metric), which we used to guide coarsening decisions. These clustering algorithms all follow the local moving scheme. Nodes are visited asynchronously in parallel and are moved to the best cluster in their neighborhood.

To achieve determinism, we use the synchronous local moving approach which is popular in distributed Louvain implementations for community detection [54]. Moves are calculated but not applied until the end of a local moving round and thus do not influence one another. The difficult part and difference to prior work is that not all calculated moves can be applied, for example due to the balance constraint. We must select a subset that is as profitable as possible. We also break down each round into further sub-rounds, to trade off more frequent synchronization for more accurate gains.

Except for the use of non-internally deterministic sub-routines such as sorting, group-by, and emitting elements to a collection in parallel, our algorithms are internally deterministic, i.e., additionally pass through the same internal states on each run [17].

Deterministic Label Propagation Refinement. In synchronous label propagation, we first calculate the highest gain move for each node in the current sub-round. In a second step, we perform balance-preserving swaps between block pairs, prioritized by the gains of the calculated moves. This generalizes a previous approach in SocialHash [66] to weighted hypergraphs, and thus allows the use in a multilevel framework.

For each block pair (V_s, V_t), we sort the two move sequences M_{st} from V_s to V_t and M_{ts} from V_t to V_s by gain and then select a prefix $M_{st}[0 : i]$ and $M_{ts}[0 : j]$, from each sequence to apply. We use the node ID as tie-breaker for determinism. Let $x(i, j) := \sum_{a=0}^{i-1} c(M_{st}[a]) - \sum_{a=0}^{j-1} c(M_{ts}[a])$ be the weight added to block V_t and removed from block V_s after swapping the nodes in the corresponding

prefixes. We call i, j feasible if $-(L_{\max} - c(V_s)) \leq x(i,j) \leq L_{\max} - c(V_t)$, i.e., after the swaps the partition is still balanced. To maximize gain, we look for the longest feasible prefixes. This can be computed similar to merging two sorted arrays. Keep two pointers i, j to the current prefixes of $M_{st}[0:i], M_{ts}[0:j]$. In each step advance the pointer of the sequence whose source block receives more weight, i.e., advance j if $x(i,j) < 0$ or i if $x(i,j) > 0$. If $x(i,j) = 0$ advance either, if the end of the corresponding sequence is not yet reached.

The parallelization follows a common idea for parallel merging. We first compute the cumulative gains of the sequences via parallel prefix sum operations. Then the following algorithm is applied recursively to perform the selection. We do binary search to find the smallest index q in the shorter sequence whose cumulative weight is not less than the cumulative weight of the middle of the longer sequence. The two sub-sequence pairs to the left and right of the middle and q can be searched independently in parallel. Let l denote the length of the longer sequence, then the algorithm does $O(l)$ work and has $O(\log^2(l))$ depth. There are parallel merge algorithms with $O(\log(l))$ depth, but these are more complicated and unlikely to yield faster running time in practice.

Additionally, we propose two optimizations that are helpful in practice but do not affect the theoretical running time. If the right parts contain feasible prefixes we return them as we prefer longer prefixes, otherwise we return the result from the left parts. If the prefixes at the splitting points are feasible, we can omit the left call. Further, we can omit the right call if the cumulative weight at the middle of the longer sequence exceeds that at the end of the shorter sequence.

Deterministic Louvain Method. There is no weight constraint on clusters in the Louvain algorithm. Therefore, we can apply all of the calculated moves. However, there is an intricacy with floating point weights, which we need due to the edge weight model employed [63]. In modularity optimization with the Louvain method, the cluster volume (weighted degree sum of the cluster) is part of the move decisions. Usually, the volumes are updated in parallel after a node move [111]. Unfortunately, floating point arithmetic is not associative: different schedules will lead to slightly different rounded values, which actually resulted in non-deterministic outcomes. One option is to recompute the volumes after each local moving subround. However, this is substantially slower than only considering updates from moved nodes, particularly at later stages when fewer nodes are moved. Therefore, we have to establish an order in which the volume updates of each cluster are aggregated. First we group the updates by cluster, then sort by node ID. Subsequently, we perform a reduction on each group with static load balancing, which is needed for determinism.

Deterministic Clustering for Coarsening. During coarsening, we bound the weight of the heaviest cluster by an upper weight limit $c_{\max}$, to ensure that initial partitioning can find a feasible solution. The difference to refinement is that we have significantly more clusters, and only unclustered nodes (singletons) can move. Therefore, the approach from refinement is not applicable here, which is why we use a simpler scheme.

Each unclustered node in a sub-round first determines its desired target cluster according to the heavy-edge rating function. Then, we group the moves by the target cluster, and sort them in order of ascending node weight and use the node ID as tie-breaker for determinism. For each group, we compute a prefix sum on the node weights, and apply the longest prefix that does not exceed the weight constraint, rejecting the remaining moves.

As an optimization to reduce the amount of work in the group-by stage (second largest bottleneck), we already sum up cluster weights during the target-cluster calculation step (main bottleneck). If all moves into a cluster combined do not exceed the weight constraint, we simply approve them all and exclude the target cluster from the group-by stage.

For further details such as implementation details and group-by mechanisms used, as well as initial partitioning and contraction algorithms, we refer to [43, 49].

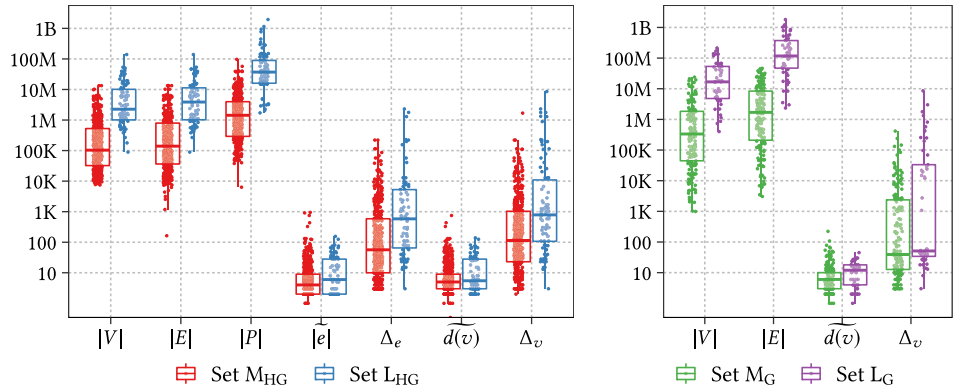

Fig. 8. Summary of different properties for our benchmark sets. It shows for each (hyper)graph (points), the number of nodes $|V|$, nets $|E|$ and pins $|P|$, as well as the median and maximum net size ($\widetilde{|e|}$ and Δ_e) and node degree ($\widetilde{d(v)}$ and Δ_v).

12 EXPERIMENTS

All presented algorithms have been made available in the *Multi-Threaded Karlsruhe Hypergraph Partitioning framework Mt-KaHyPar*.[3] It implements a parallel multilevel and n-level partitioning algorithm, as well as a deterministic version of the multilevel algorithm and optimized data structures for graph partitioning. Mt-KaHyPar optimizes the connectivity metric for hypergraph partitioning and the edge cut metric for graph partitioning.

The following experimental evaluation is structured as follows: We first decribe the four different benchmark sets composed of over 800 graphs and hypergraphs, and discuss the experimental setup and methodology used in our experiments. We then evaluate the time-quality trade-offs and speedups of Mt-KaHyPar's different partitioning configurations, and analyze the running time of its algorithmic components in Section 12.1. We conclude the evaluation by comparing Mt-KaHyPar to 25 different sequential and parallel graph and hypergraph partitioning algorithms in Section 12.2.

Instances. We assembled four different benchmark sets. Two of the sets consist of graphs (G), while the other two consists of *hypergraphs (HG)*. The sets are further subdivided into medium-sized (M) and large instances (L). We abbreviate the name of a benchmark set, e.g., with L_{HG}. The baseline denotes the size of the instances, while the subscript indicates whether it contains graphs or hypergraphs. We summarize the properties of the instances contained in the benchmark sets in Figure 8.[4] Note that all graphs and hypergraphs have unit node and (hyper)edge weights.

The hypergraph instances are derived from four sources encompassing three application domains: the ISPD98 VLSI Circuit Benchmark Suite [6] (Ispd98), the DAC 2012 Routability-Driven Placement Contest [116] (Dac2012), the SuiteSparse Matrix Collection [32] (Spm), and the International SAT Competition 2014 [16] (Sat14). We interpret the rows and columns of a sparse matrix as nets and nodes, and a non-zero entry in a cell (i, j) indicates whether or not node j is a pin of net i [28]. We translate satisfiability formulas into three different hypergraph representations [89]. The Primal resp. Literal representation interpretes the variables resp. literals as nodes, while

[3] Mt-KaHyPar is publicly available from https://github.com/kahypar/mt-kahypar
[4] We made all benchmark sets and detailed statistics of their properties publicly available from https://algo2.iti.kit.edu/heuer/talg/

the clauses form the hyperedges spanning the corresponding nodes. The Dual representation models the clauses as nodes and variables as hyperedges.

Set M_{HG} contains all 488 hypergraphs from the well-established benchmark set of Heuer and Schlag [63] (18 Ispd98, 10 Dac2012, 184 Spm, 276 Primal, Literal, and Dual instances). The benchmark set L_{HG} is composed of 94 large hypergraphs that we selected in order to have more inputs where parallelization is important and useful. It contains the 8 largest Sat14 instances from set M_{HG} that we enhanced with 6 even larger satisfiability formulas from the International SAT Competition 2014 [16] ($3 \cdot 14 = 42$ different hypergraph representations). We also included 42 sparse matrices with at least 15 million non-zeros, randomly sampled from the SuiteSparse Matrix Collection [32]. Additionally, we added all Dac2012 instances from set M_{HG}. The largest hypergraph of set L_{HG} has roughly two billion pins.

Our graph benchmark sets are composed of instances from the 10th DIMACS Implementation Challenge [13] (Dimacs), the Stanford Large Network Dataset Collection [87] and the Laboratory for Web Algorithms [79] (Social Networks), the DAC 2012 Routability-Driven Placement Contest [116] (Dac2012), the SuiteSparse Matrix Collection [32, 123] (Spm), and several randomly generated graphs [39, 77] (Random Graphs).

Set M_G was initially assembled by Gottesbüren et al. [48] (195 graphs) from which we excluded the 39 largest graphs and additionally added 16 social networks from the Stanford Large Network Dataset Collection [87] (114 Dimacs, 30 Social Networks, 15 Random Graphs, 3 Spm, and 10 Dac2012 instances). The benchmark set L_G contains 38 out of 42 instances from a graph collection assembled by Ahkremtsev [2] (four instances were considered as too large as they were used to evaluate external memory algorithms). Additionally, we enhanced set L_G with 15 graphs that we excluded from set M_G and were not contained in set L_G yet (16 Dimacs, 16 Social Networks, 15 Random Graphs, and 6 Spm instances). The largest graph of set L_G has roughly two billion edges.

Experimental Setup. Experiments on medium-sized instances (set M_G and M_{HG}) run on a cluster of Intel Xeon Gold 6230 processors (2 sockets with 20 cores each) running at 2.1 GHz with 96GB RAM. In these experiments, we partition each (hyper)graph ten times using different random seeds into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks with an allowed imbalance of $\varepsilon = 3\%$ and a time limit of eight hours. Experiments on large instances (set L_G and L_{HG}) are done on an AMD EPYC 7702 processor (1 socket with 64 cores) running at 2.0–3.5 GHz with 1024GB RAM. Here, we partition each (hyper)graph three times using different random seeds into $k \in \{2, 8, 16, 64\}$ blocks with an allowed imbalance of $\varepsilon = 3\%$ and a time limit of two hours. Note that we restrict the parameter space for experiments on large instances due to limited computational resources. For graph partitioning, we configure the algorithms to optimize the edge cut metric, while we focus on the connectivity metric for hypergraphs. We will also refer to both metrics as the solution quality of a partition.

Aggregating Performance Numbers. We call a (hyper)graph partitioned into k blocks an *instance*. For each instance, we aggregate running times and the solution quality 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. If all runs of an algorithm produced an imbalanced partition or ran into the time limit on an instance, we consider the solution as *infeasible*. In plots, we mark imbalanced solutions with ✗ and similarly instances that timed out with ⏲. Runs with imbalanced partitions are not excluded from aggregated running times. For runs that exceeded the time limit, we use the time limit itself in the aggregates. When comparing running times, we say that an algorithm $\mathcal{A}$ is faster than $\mathcal{B}$ by a factor of x on average if $x := \overline{t_{\mathcal{B}}}/\overline{t_{\mathcal{A}}} > 1$ where $\overline{t_{\mathcal{A}}}$ and $\overline{t_{\mathcal{B}}}$ are geometric mean running times of $\mathcal{A}$ and $\mathcal{B}$.

Performance Profiles. Performance profiles can be used to compare the solution quality of different algorithms [35]. Let $\mathcal{X}$ be the set of all algorithms, $\mathcal{I}$ the set of instances, and $q_{\mathcal{A}}(I)$ the

quality of algorithm $\mathcal{A} \in \mathcal{X}$ on instance $I \in \mathcal{I}$ ($q_{\mathcal{A}}(I)$ is the arithmetic mean over all seeds). For each algorithm $\mathcal{A}$, performance profiles show the fraction of instances (y-axis) for which $q_{\mathcal{A}}(I) \leq \tau \cdot \text{Best}(I)$, where τ is on the x-axis and $\text{Best}(I) := \min_{\mathcal{A}' \in \mathcal{X}} q_{\mathcal{A}'}(I)$ is the best solution produced by an algorithm $\mathcal{A}' \in \mathcal{X}$ for an instance $I \in \mathcal{I}$. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm $\mathcal{A} \in \mathcal{X}$ performs best. Achieving higher fractions at smaller τ values is considered better. The ✗- and ☹-tick indicates the fraction of instances for which *all* runs of that algorithm produced an imbalanced solution or timed out. Note that these plots relate the quality of an algorithm to the best solution and thus do not permit a full ranking of three or more algorithms.

Effectiveness Tests. Ahkremtsev et al. [4] introduce *effectiveness tests* to compare solution quality when two algorithms are given a similar running time by performing additional repetitions with the faster algorithm. Following this approach, we generate virtual instances that we compare using performance profiles. Consider two algorithms $\mathcal{A}$ and $\mathcal{B}$, and an instance I. We first sample one run of both algorithms for instance I. Let $t^1_{\mathcal{A}}, t^1_{\mathcal{B}}$ be their running times and assume that $t^1_{\mathcal{A}} \leq t^1_{\mathcal{B}}$. We then sample additional runs without replacement for $\mathcal{A}$ until their accumulated time exceeds $t^1_{\mathcal{B}}$ or all runs have been sampled. Let $t^2_{\mathcal{A}}, \ldots, t^l_{\mathcal{A}}$ denote their running times. We accept the last run with probability $(t^1_{\mathcal{B}} - \sum_{i=1}^{l-1} t^i_{\mathcal{A}})/t^l_{\mathcal{A}}$ so that the expected time for the sampled runs of $\mathcal{A}$ equals $t^1_{\mathcal{B}}$. The solution quality is the minimum out of the sampled runs. For each instance, we generate 10 virtual instances.

Statistical Significance Tests. We use the Wilcoxon signed-rank test [122] to determine whether the difference of the solutions produced by two algorithms is statistically significant. At a 1% significance level ($p \leq 0.01$), a Z-score with $|Z| > 2.576$ is deemed significant [22, p. 180].

12.1 Evaluation of Framework Configurations

In this section, we present a detailed evaluation of our shared-memory partitioning algorithm Mt-KaHyPar. We first describe its different configurations and compare them regarding solution quality and running time. We then discuss the running times of the different algorithmic components, present speedups, and evaluate the impact of our optimizations for plain graphs.

Framework Configurations. The Mt-KaHyPar framework provides a multilevel (Mt-KaHyPar-D, **D**efault) and n-level partitioning algorithm (Mt-KaHyPar-Q, **Q**uality), as well as configurations extending them with flow-based refinement (Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F, **F**lows). It also implements a deterministic version of the multilevel algorithm (Mt-KaHyPar-SDet, **S**peed-**Det**erministic), which does not use the FM algorithm. The code is written in `C++17`, parallelized using the `TBB` parallelization library [96], and compiled using `g++9.2` with the flags `-O3 -mtune=native -march=native`. All of these algorithms have a large number of configuration options and were carefully tuned to provide the best trade-off between solution quality and running time. However, a detailed parameter tuning study is beyond the scope of this paper. We already mentioned specific choices for relevant parameters in the text and refer the reader to our conference publications [43, 46, 47, 50] and the dissertation of Heuer [60, see Table 5.1 on p. 98–99] for a detailed overview.[5]

Time-Quality Trade-Off. Figure 9 compares the solution quality of the partitions produced by the different configurations of Mt-KaHyPar and their running times relative to Mt-KaHyPar-D on set

[5]Parameter tuning was done on a subset M_P of set M_{HG} that consists of 100 instances not contained in set L_{HG}. We compared the quality produced by different partitioning algorithms on set M_{HG} and $M_{HG} \setminus M_P$ using performance profiles and found that they do not differ [60, see Figure 8.1 on p. 160]. Thus, we decided to include the parameter tuning instances in the final evaluation to increase the evidence of the following experimental results.

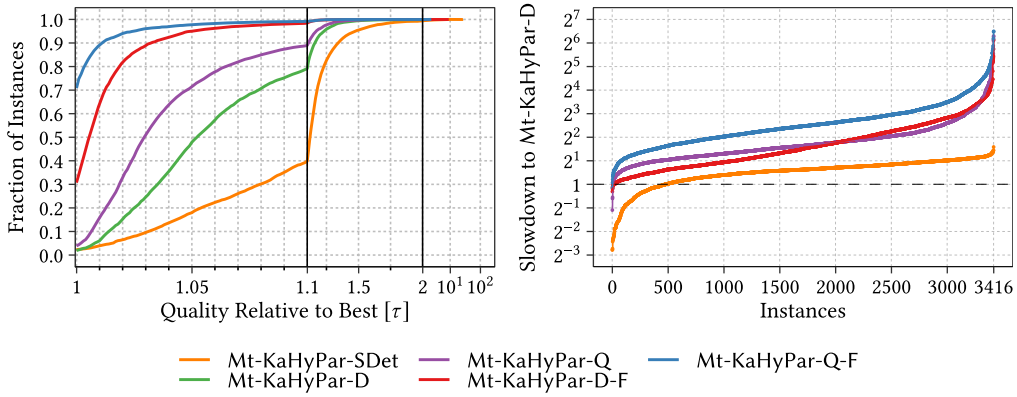

Fig. 9. Performance profiles and running times comparing the different configurations of Mt-KaHyPar executed with 10 threads on set M_{HG}.

M_{HG}. The configurations can be ranked from lowest to highest quality as follows: Mt-KaHyPar-SDet (geometric mean running time 1.25s), Mt-KaHyPar-D (0.88s), Mt-KaHyPar-Q (2.99s), Mt-KaHyPar-D-F (2.73s), and Mt-KaHyPar-Q-F (5.08s). The ranking looks similar for running times except for Mt-KaHyPar-D which is faster than Mt-KaHyPar-SDet. However, this changes when we compare their running times on the larger instances of set L_{HG}. Here, our deterministic configuration is faster than Mt-KaHyPar-D (Mt-KaHyPar-SDet: 3.14s vs Mt-KaHyPar-D: 4.65s with 64 threads). For smaller instances, initial partitioning is the most time-consuming component since we stop coarsening when we reach $160k$ nodes which can be close to the original number of nodes for some instances (e.g., 10240 nodes for $k = 64$). To reduce the running time of initial partitioning, Mt-KaHyPar-D adaptively adjusts the number of repetitions of the different algorithms in the bipartitioning portfolio based on their success so far. For larger instances, the smallest hypergraph is often significantly smaller than the input, and therefore the running time of initial partitioning becomes negligible compared to the other phases.

The median improvement in solution quality of Mt-KaHyPar-D over Mt-KaHyPar-SDet is 6%, while flow-based refinement (Mt-KaHyPar-D-F) improves Mt-KaHyPar-D by 4.2% in the median at the cost of a 3 times slower running time on average. When we compare the multilevel (Mt-KaHyPar-D) and n-level partitioning algorithm (Mt-KaHyPar-Q), we see that n-level partitioning produces partitions that are 1.9% better than those produced by our multilevel algorithm in the median, but its running time is 3.4 times slower on average. The differences in solution quality and running time are less pronounced when both configurations use flow-based refinement (median improvement of Mt-KaHyPar-Q-F over Mt-KaHyPar-D-F is 0.6%). Note that multilevel partitioning with flow-based refinement produces better partitions than our n-level configuration (2%), while it is also slightly faster.

We have seen that using stronger refinement algorithms leads to substantially better solution quality at the cost of higher running times. Moreover, traditional multilevel algorithms can produce better partitions than n-level algorithms when flow-based refinement is used.

Effectiveness Tests. Our n-level algorithm computes better partitions than our multilevel algorithm without flow-based refinement, but is 3 times slower on average. When both configurations use flow-based refinement, the difference in solution quality becomes less pronounced. We therefore use effectiveness tests to compare Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F) when both are given the same amount of time by performing additional repetitions with the faster algorithm until the accumulated running time equals the running time of the slower algorithm.

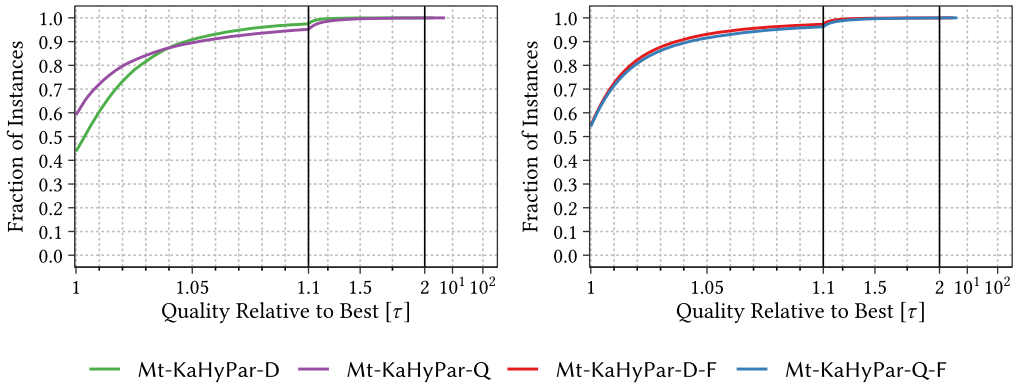

Fig. 10. Effectiveness tests comparing Mt-KaHyPar-D and Mt-KaHyPar-Q (left), and both configurations that extend them with flow-based refinement (right) on set M_{HG}.

Figure 10 shows the results of these experiments. As we can see, the performance lines of Mt-KaHyPar-D(-F) and Mt-KaHyPar-Q(-F) are almost identical in the performance profiles. This means that Mt-KaHyPar-D(-F) computes partitions of comparable quality to its n-level counterpart when we give more time for additional repetitions.

In contrast to the prevalent perception in the literature that more levels lead to better partitioning results [7, 99, 104], we showed that already a logarithmic numbers of levels suffices to compute solutions of high quality. However, we still see a large potential in the n-level scheme as it provides a greater design space for future improvements.

Running Time of Algorithmic Components. We now analyze the running times of the different algorithmic components of Mt-KaHyPar on set L_{HG}.[6] Figure 11 shows the fraction of instances (x-axis) for which the share of a component on the total execution time is $\geq y\%$ for each configuration of Mt-KaHyPar. The intersection of $x = 0.5$ with the line of a component is the median share of the component on the overall partitioning time.

The most time-consuming components of Mt-KaHyPar-D are preprocessing (consisting of the community detection algorithm presented in Section 4.3), coarsening, and the FM algorithm. These components have similar shares on the total partitioning time, which is between 21% and 23% in the median. However, there are some long-running outliers for the FM algorithm on instances with many large hyperedges. Here, the FM searches tend to move more nodes due to many zero-gain moves. The median share of initial partitioning on the total execution time is 8.3%. Longer running times can be observed for instances where we do not reach the contraction limit as, e.g., social networks with highly-skewed node degree distributions. The running time of label propagation is negligible on most of the instances.

In the deterministic version of Mt-KaHyPar, preprocessing (median share on the total execution time is 42.7%) and coarsening (28.1%) takes the most time, while flow-based refinement (77.8%) dominates the running time of Mt-KaHyPar-D-F (the same is holds for Mt-KaHyPar-Q-F, which is why it is omitted in the plot). In our n-level partitioning algorithm, the most time-consuming components are coarsening (16%), batch uncontractions (17.3%), and the localized version of the FM algorithm (22.1%).

[6]Since initial partitioning uses most of the other components within multilevel recursive bipartitioning, we evaluate running times on the larger instances of set L_{HG} such that initial partitioning becomes less time-consuming as explained earlier. We evaluated the solution quality of Mt-KaHyPar's different configurations on set M_{HG} due to the effectiveness tests, which require a large number of repetitions per instance (10 repetitions on set M_{HG} vs 3 repetitions on set L_{HG}).

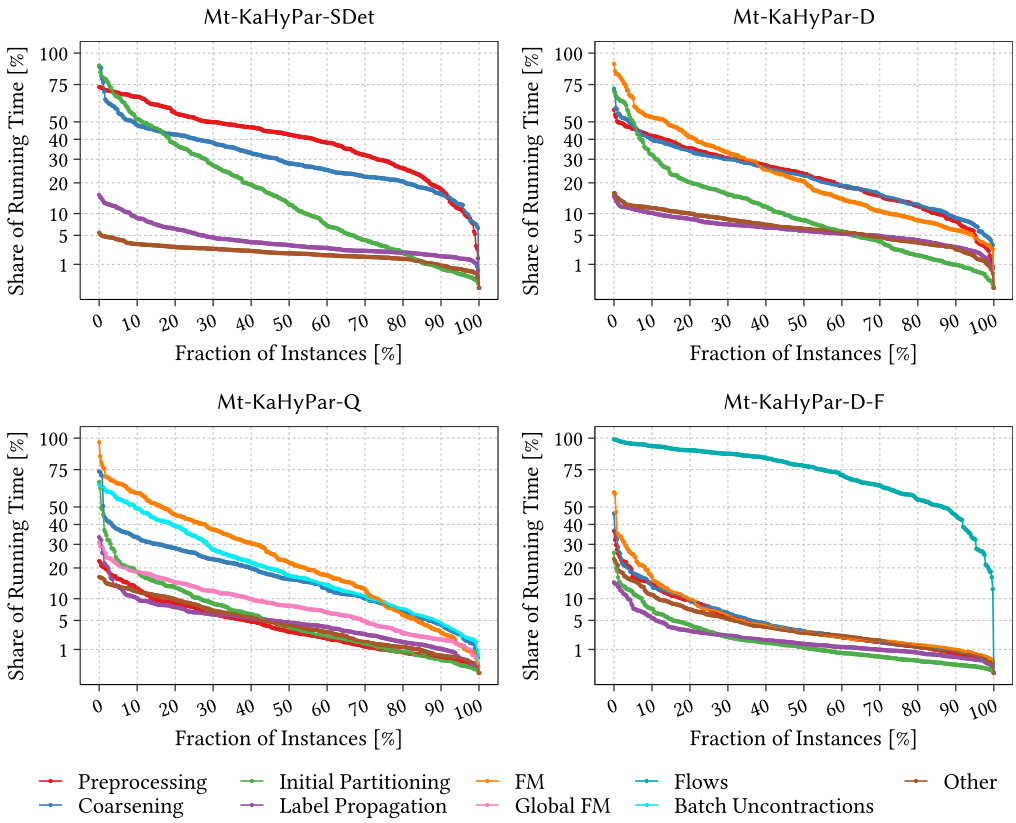

Fig. 11. Running time shares of the algorithmic components on the total execution time of the different configurations of Mt-KaHyPar. For Mt-KaHyPar-Q, label propagation and FM corresponds to their localized versions that run after each batch uncontraction, while *Global FM* refers to the FM version that runs on the entire hypergraph after restoring identical nets.

Scalability. In Figure 12 and 13, and Table 1, we summarize self-relative speedups of Mt-KaHyPar for each configuration and the different phases of the multilevel scheme with an increasing number of threads $t \in \{1, 4, 16, 64\}$. The scalability experiments run on set L_{HG}. However, we used a subset for Mt-KaHyPar-Q (77 out of 94 hypergraphs) and Mt-KaHyPar-D-F (76 out of 94 hypergraphs) to ensure reasonable running times. This set consists of instances where Mt-KaHyPar-Q/-D-F was able to finish in under 600 seconds with 64 threads for all tested values of k. The experiment still took 6 weeks to complete for each configuration. Note that we only rerun the experiments for Mt-KaHyPar-SDet/-D for this work, while the speedups of Mt-KaHyPar-Q/-D-F are based on the data from the corresponding conference publications [46, 47] due to the high time requirements. In the plot, we represent the speedup (y-axis) of each instance as a point and the centered rolling geometric mean over the points with a window size of 25 as a line. The x-axis shows the single-threaded running time of the corresponding configuration resp. component.[7]

[7]In contrast to many other publications in the parallel partitioning community, we do not correlate speedups to any of the common hypergraph metrics (such as the number of pins). We found that the running time often depends on a variety of different factors. 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.

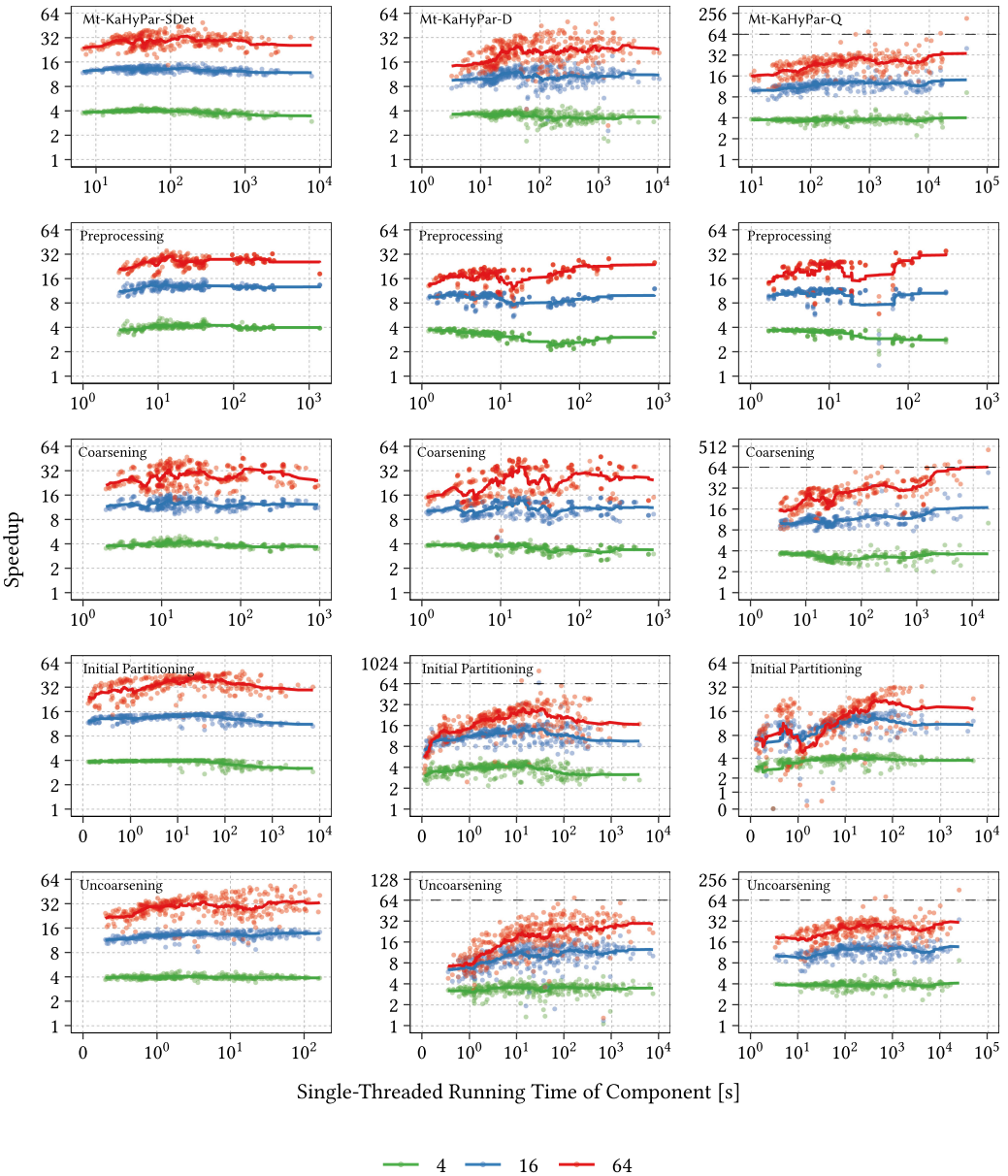

Fig. 12. Speedups of Mt-KaHyPar-SDet (left), Mt-KaHyPar-D (middle), and Mt-KaHyPar-Q (right).

The overall geometric mean speedup of Mt-KaHyPar-D is 3.5 for $t = 4$, 10.6 for $t = 16$, and 20.5 for $t = 64$. If we only consider instances with a single-threaded running time $\geq$ 100s, the geometric mean speedup increases to 22.3 for $t = 64$. For $t = 4$, the speedup is at least 3 on 89.1% of the instances. The community detection algorithm (referred to as *preprocessing*) and coarsening share many similarities in their implementation and both show reliable speedups for an increasing number of threads. For initial partitioning and the uncoarsening phase, we observe that longer single-threaded execution times leads to substantially better speedups. The most time-consuming

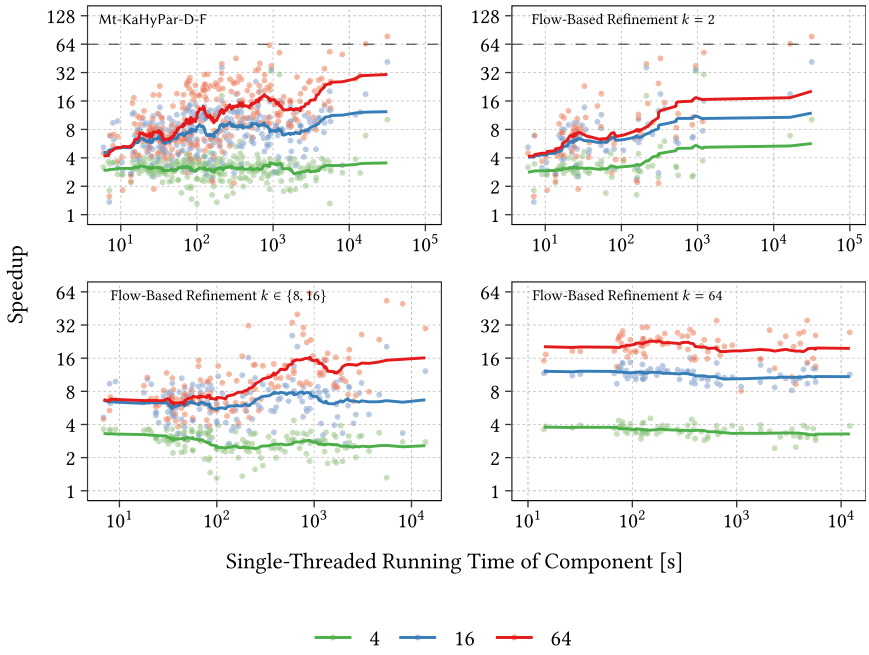

Fig. 13. Speedups of Mt-KaHyPar-D-F and flow-based refinement for different values of k.

Table 1. Geometric mean speedups of the total execution time, preprocessing (P), coarsening (C), initial partitioning (IP), and uncoarsening (UC) of the different configurations of Mt-KaHyPar over all instances and instances with a single-threaded time ≥ 100s

		Mt-KaHyPar-SDet		Mt-KaHyPar-D		Mt-KaHyPar-Q		Mt-KaHyPar-D-F	
Num. Threads		All	≥ 100s	All	≥ 100s	All	≥ 100s	All	≥ 100s
Total	4	3.9	3.8	3.5	3.3	3.7	3.7	3.1	3.1
	16	12.8	12.5	10.6	10.4	11.9	12.5	7.4	8.4
	64	28.8	29.6	20.5	22.3	23.7	25.9	10.6	14.5
P	4	4.1	3.9	3.1	2.8	3.4	2.8	3.3	3
	16	12.6	12.5	9.1	9.4	10	10.3	9.6	10.1
	64	25.7	27	17.4	23.1	19.7	28.6	15.2	22.3
C	4	3.9	3.7	3.6	3.3	3.3	3.4	3.7	3.3
	16	12.4	12.4	10.9	11.1	11.2	13	11.4	12
	64	27.4	29.8	22.9	27	25.4	36.4	22.4	32.1
IP	4	3.8	3.4	3.7	3.2	3.7	3.8	3.8	3.5
	16	13.5	12.4	11.5	10.8	10.1	11.2	11.2	12
	64	34.4	33.8	17.9	18.8	11.6	18.2	13.6	19.7
UC	4	4	3.9	3.4	3.5	3.9	3.9	2.9	3.1
	16	13	13.8	9.8	11.6	12	12.7	5.9	8.1
	64	28.9	32.2	17.3	25.6	24.3	26.5	7.8	13.9

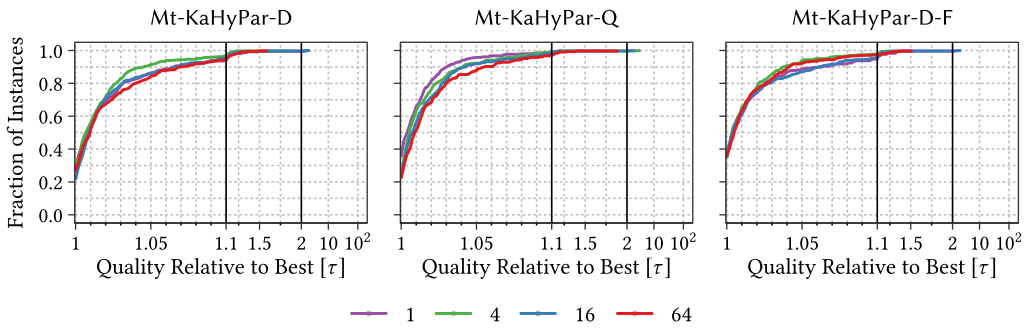

Fig. 14. Performance profiles comparing the solution quality of Mt-KaHyPar with an increasing number of threads on set L_{HG}.

component of the uncoarsening phase is the FM algorithm. The geometric mean speedup of the FM algorithm is 21.1 for $t = 64$, which increases to 27.4 for instances with sequential time $\geq$ 100s.

If we compare the speedups of Mt-KaHyPar-SDet to its non-deterministic counterpart Mt-KaHyPar-D, we see that it achieves much more reliable speedups. Especially, the speedups of initial partitioning increase substantially with a geometric mean speedup of 34.4 for $t = 64$. Since Mt-KaHyPar-SDet does not adaptively adjust the number of repetitions in the bipartitioning portfolio, it performs more work in the initial partitioning phase and is not affected by non-deterministic decisions, which increases its scalability (geometric mean running time of initial partitioning is 9.36s in Mt-KaHyPar-SDet vs 4.23s in Mt-KaHyPar-D for $t = 1$). The overall geometric mean speedup of Mt-KaHyPar-SDet is 3.9 for $t = 4$, 12.8 for $t = 16$, and 28.8 for $t = 64$.

The coarsening and batch uncontraction algorithm are the components that differentiate our n-level partitioning algorithm Mt-KaHyPar-Q from the other multilevel algorithms. Both components exhibit good speedups, while coarsening (geometric mean speedup is 25.4 for $t = 64$) scales slightly better than the batch uncontractions (23.2 for $t = 64$). Moreover, the speedups of the localized version of FM algorithm that runs after each batch uncontraction operation are less pronounced than the speedups of the FM algorithm in Mt-KaHyPar-D (geometric mean speedup 19 vs 21.1 for $t = 64$). Note that we also observe super-linear speedups, which are caused by non-deterministic coarsening decisions. The geometric mean speedup of Mt-KaHyPar-Q is 3.7 for $t = 4$, 11.9 for $t = 16$, and 23.7 for $t = 64$.

Mt-KaHyPar-D-F extends Mt-KaHyPar-D with flow-based refinement. We therefore only show speedups for this component in Figure 13. Unfortunately, the speedups are less promising as for the other configurations. The geometric mean speedup of Mt-KaHyPar-D-F is 3.1 for $t = 4$, 7.4 for $t = 16$, and 10.6 for $t = 64$. However, we achieve better speedups for larger values of k where all parallelism is leveraged in the scheduling algorithm, and none in the FlowCutter algorithm. For $k = 2$, the scalability depends on our parallel maximum flow algorithm for which we observe similar speedups as reported in Reference [15] – the work on which our parallel implementation is based on. Thus, increasing the scalability of maximum flow algorithms is an important avenue for future research.

In Figure 14, we compare the solution quality of the different configurations when increasing the number of threads. We can see that using more threads adversely affects the solution quality of the partitions produced by Mt-KaHyPar-Q, but only by a small margin (solutions are 0.4% better with one compared to 64 threads). Mt-KaHyPar-D and Mt-KaHyPar-D-F produce comparable solutions when increasing the number of threads.

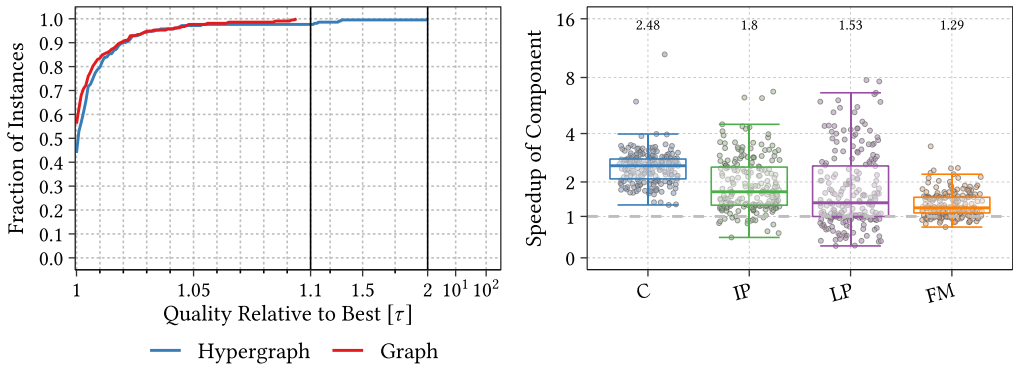

Fig. 15. Performance profile (left) and speedups (right) of coarsening (C), initial partitioning (IP), label propagation (LP), and FM refinement comparing Mt-KaHyPar-D with and without our optimized graph data structure on set L_G.

Effects of Graph Optimizations. In Section 10, we presented optimized data structures for graph partitioning used as a drop-in replacement in our partitioning algorithm. Figure 15 shows their impact on the solution quality and speedups for different algorithmic components of Mt-KaHyPar-D on set L_G. As it can be seen, replacing our hypergraph with the graph data structures does not adversely affect the solution quality of Mt-KaHyPar-D as both performance lines are almost identical and converge quickly towards $y = 1$.

The coarsening algorithm benefits most from our optimized graph data structure (geometric mean speedup 2.48). The hypergraph version computes a clustering of the nodes by iterating over the pin-lists of nets to aggregate ratings, and subsequently contract that clustering by collapsing two adjacency arrays (one for the pin-lists and one for the incident nets). The cache-friendly memory layout for graphs (only one adjacency array for neighbors) leads to faster access times to enumerate neighbors and to a simpler contraction algorithm. The FM algorithm has the least promising speedups (1.29). One of the most time-consuming parts of the algorithm is retrieving and updating entries from the gain table, which has the same asymptotic worst-case complexity in both implementations. The initial partitioning phase (1.8) has better speedups than both refinement algorithms but slightly worse speedups than coarsening. This can be explained by the fact that initial partitioning uses all algorithms within multilevel recursive bipartitioning.

The overall speedup of the graph version of Mt-KaHyPar-D over its hypergraph counterpart is 1.75 on average (geometric mean running time 10.8s vs 18.94s). In the dissertation of Heuer [60, p. 150–153], we also present an optimized graph data structure for n-level partitioning, which accelerates Mt-KaHyPar-Q by a factor of 1.91 on average (97.45s vs 186.32s).

12.2 Comparison to Other Systems

We now compare Mt-KaHyPar to existing partitioning algorithms to see if it can improve the state-of-the-art. We did an extensive research on publicly available partitioning tools and were able to include 25 different sequential and parallel graph and hypergraph partitioners that we compare on over 800 graphs and hypergraphs. Thus, to the best of our knowledge, this study represents the most comprehensive comparison of partitioning algorithms to date. We primarily focus on multilevel algorithms as it has been shown that they provide an excellent trade-off between solution quality and running time [55, 56]. While there are even faster partitioning methods that omit the multilevel scheme, it has been shown that they are inferior to multilevel algorithms in terms of solution quality [55, 104]. Moreover, algorithms that achieve even higher solution quality than

Table 2. Listing of graph and hypergraph partitioning algorithms (GP and HGP) included in the experimental evaluation. For algorithms publicly available on GitHub, we report the first seven characters of the corresponding commit hash indicating the used version

	Sequential		Parallel		
	Algorithm	Version	Algorithm	Version	Machine Model
GP	Metis [71, 72]	5.1.0	KaMinPar [48]	29101f6	Shared-Memory
	Metis-R and Metis-K		Mt-Metis [81–83]	0.6.0	Shared-Memory
	KaFFPa [101, 107]	f239f7a	ParMetis [70]	4.0.3	Distributed-Memory
	KaFFPa-Fast(S)/-Eco(S)/-Strong(S)		Mt-KaHIP [2, 4]	30de737	Shared-Memory
	Scotch [95]	6.1.3	ParHIP [91]	f239f7a	Distributed-Memory
			ParHIP-Fast and ParHIP-Eco		
HGP	PaToH [28]	3.3	Zoltan [34]	3.83	Distributed-Memory
	PaToH-D and PaToH-Q		BiPart [88]	49a59a6	Shared-Memory
	hMetis [69, 73]	2.0pre1			
	hMetis-R and hMetis-K				
	KaHyPar [104]	876b776			
	KaHyPar-CA, *r*KaHyPar, and *k*KaHyPar				
	Mondriaan [115]	4.2.1			

multilevel algorithms such as evolutionary algorithms [10, 102] and approaches based on integer linear programming [57] would not run in a reasonable time frame on our benchmark sets.

We first provide a description of the partitioning algorithms included in our study and explain how we configured them. We then identify a subset of Pareto-optimal algorithms to which we then compare Mt-KaHyPar.[8]

Included Algorithms. Table 2 lists all partitioning algorithms included in the following experimental evaluation. Many of these algorithms provide multiple partitioning configurations offering different trade-offs in running time and solution quality (e.g., KaFFPa-Fast/-Eco/-Strong, or the default (-D) and quality preset (-Q) of PaToH), or are based on either recursive bipartitioning (e.g., hMetis-R) or direct k-way partitioning (e.g., hMetis-K). The graph partitioner KaFFPa also provides different settings for partitioning *social* networks (KaFFPa-FastS/-EcoS/-StrongS). Thus, we include all three social configurations as well as their non-social counterparts (KaFFPa-Fast/-Eco/-Strong). For the n-level algorithm KaHyPar, we include the recursive bipartitioning (*r*KaHyPar) and direct k-way version (*k*KaHyPar, which uses similar algorithmic components as Mt-KaHyPar-Q-F), as well as a configuration without flow-based refinement (KaHyPar-CA, which uses similar algorithmic components as Mt-KaHyPar-Q).

Unfortunately, we were not able to include the publicly available versions of Parkway [113] (distributed-memory), PT-Scotch [30] (distributed-memory), and Chaco [56] (sequential). These algorithms failed with segmentation faults on most instances of our benchmark sets.

Algorithm Configuration. We configure all graph partitioning algorithms to optimize the edge cut metric, while we optimize the connectivity metric for hypergraph partitioning. We run Mt-KaHyPar using *ten* threads for comparisons to sequential algorithms as this is a typical number of available cores in a modern commodity workstation. We add a suffix to the name of parallel algorithms indicating the number of threads used, e.g., Mt-KaHyPar 64 for 64 threads. We omit

[8]We made all experimental results publicly available from https://algo2.iti.kit.edu/heuer/talg/

Table 3. Summary of algorithms (first column) outperforming others (second column)

	Sequential				Parallel (64 threads)			
	Base Algo.	Outperformed	Med. [%]	Rel. Slow.	Base Algo.	Outperformed	Med. [%]	Rel. Slow.
GP	Metis-K	Metis-R	2.9	1.4	KaMinPar	Mt-Metis	0	9.11
	Metis-K	KaFFPa-Fast	5.8	4.3	KaMinPar	ParMetis	4.4	211.2
	Metis-K	KaFFPa-FastS	2.2	4,79	KaMinPar	ParHIP-Fast	2.8	8.18
	Metis-K	Scotch	2.5	4.66	Mt-KaHIP	ParHIP-Eco	2.2	11.62
	KaFFPa-EcoS	KaFFPa-Eco	3.2	1.04				
HGP	PaToH-D	Mondriaan	0.6	5.63	Zoltan	BiPart	69	2.31
	KaHyPar-CA	hMetis-R	0.5	3.31				
	KaHyPar-CA	hMetis-K	2.6	2.62				
	*k*KaHyPar	*r*KaHyPar	2.1	~ 1				

It shows the median improvement in the connectivity resp. edge cut metric in percent for each baseline over the outperformed algorithm and the average slowdown of the outperformed relative to the baseline algorithm.

the suffix for sequential algorithms. For graph partitioning, Mt-KaHyPar uses the partition and graph data structure presented in Section 10.

We use the default settings provided by the authors to configure the different partitioning algorithms. However, for algorithms based on recursive bipartitioning, we adjust the input imbalance parameter ε to $\varepsilon' := (1+\varepsilon)^{\frac{1}{\lceil \log_2 k \rceil}}$ (based on Equation (1) by applying it to the first bipartitioning step) when we observed that most of the computed partitions are imbalanced. This applies to Metis-R, hMetis-R, and BiPart. We further set hMetis to optimize the *sum-of-external-degree* metric $f_s(\Pi) := \sum_{e \in E_{\text{Cut}}(\Pi)} \lambda(e) \cdot \omega(e) = f_{\lambda-1}(\Pi) + f_c(\Pi)$ (connectivity plus cut-net metric) and calculate the connectivity metric accordingly. We additionally configure Mt-Metis to use its hill-scanning refinement algorithm [82]. Moreover, we do not perform multiple repetitions when running Scotch or BiPart as both do not provide a command line parameter for setting a seed value.

Identifying Competitors. Since some of the included algorithms already outperform others with regards to solution quality and running time, we compare Mt-KaHyPar only to a subset of Pareto-optimal partitioning algorithms. Table 3 presents a summary of the results that we used to identify our main competitors. The data is based on a detailed evaluation that can be found in the dissertation of Heuer [60, see Section 8.2 on p. 160–167]. We added the performance profiles and running time plots used for this evaluation in Appendix A. In the table, the algorithms in the second column are outperformed by the algorithms in the first column and are therefore excluded from the following experimental evaluation. The included systems can be classified into fast partitioning methods (PaToH-D, Zoltan, Metis-K, and KaMinPar), configurations providing a good trade-off between solution quality and running time (PaToH-Q, KaFFPa-EcoS, and Mt-KaHIP), and high-quality partitioning algorithms (KaHyPar-CA, *k*KaHyPar, and KaFFPa-Strong/-StrongS). To simplify the following evaluation, we compare the high quality algorithms to Mt-KaHyPar-Q-F (highest quality configuration) and all others to Mt-KaHyPar-D (fastest configuration).

Comparison to Sequential Systems. Figure 16 compares Mt-KaHyPar to the sequential hypergraph partitioners PaToH and KaHyPar on set M_{HG}. In an individual comparison, Mt-KaHyPar-D (geometric mean running time 0.88s) computes better partitions than PaToH-D (1.17s) and PaToH-Q (5.85s) on 82.9% and 58.34% of the instances (median improvement is 6.6% and 1.2%),[9] while

[9]It appears that Mt-KaHyPar-D performs slightly worse than PaToH-Q in the performance profiles. However, if we would compare them in a performance profile individually, we would see that the performance line of Mt-KaHyPar-D lies strictly above the line of PaToH-Q. We therefore point out that performance profiles do not permit a full ranking between three or more algorithms.

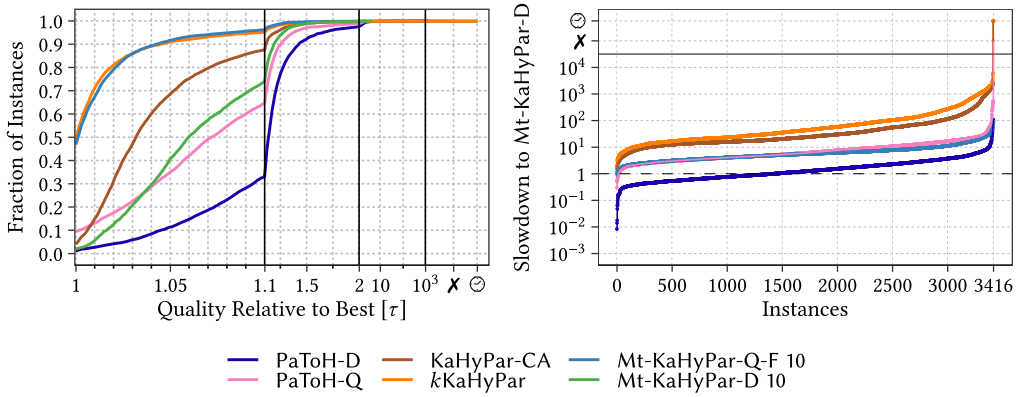

Fig. 16. Performance profiles and running times comparing Mt-KaHyPar to PaToH and KaHyPar on set M_{HG}.

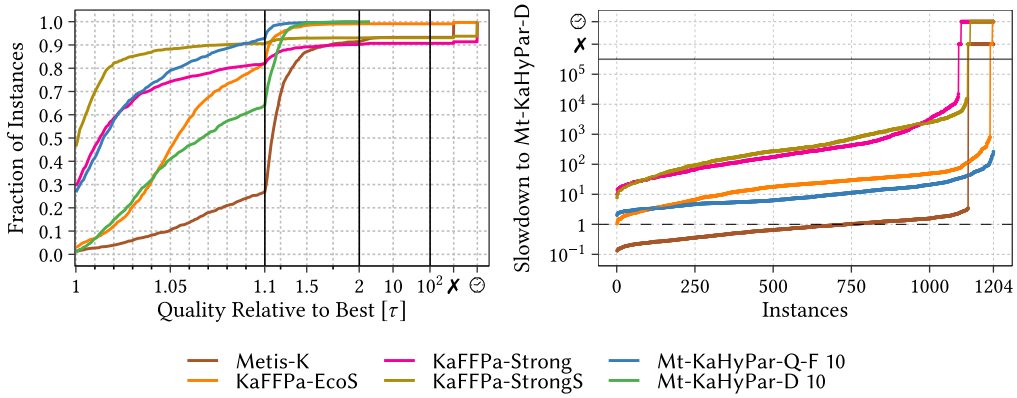

Fig. 17. Performance profiles and running times comparing Mt-KaHyPar to Metis and KaFFPa on set M_G.

it achieves a speedup of 1.32 w.r.t. PaToH-D and 6.6 w.r.t. PaToH-Q with ten threads on average. Thus, Mt-KaHyPar-D outperforms PaToH-D and PaToH-Q.

We can also see that the performance lines of Mt-KaHyPar-Q-F and *k*KaHyPar – the currently best sequential hypergraph partitioning algorithm – are almost identical, which means that both compute partitions of comparable solution quality. Mt-KaHyPar-Q-F (5.08s) is faster than KaHyPar-CA (28.14s) and *k*KaHyPar (48.97s) on almost all instances with ten threads ($\geq$ 99%). This shows that we achieved the same solution quality as the currently highest-quality sequential partitioning algorithm, while being almost an order of magnitude faster with only ten threads. Moreover, Mt-KaHyPar-Q-F is also slightly faster than PaToH-Q, while it computes better partitions than PaToH-Q on 87.7% of the instances (median improvement is 6.4%).

Figure 17 compares Mt-KaHyPar to the sequential graph partitioners Metis-K and KaFFPa on set M_G. Mt-KaHyPar-D (geometric mean running time 0.55s) is slightly slower than Metis-K (0.39s) with ten threads but produces significantly better edge cuts (median improvement is 5.9%). If we disable the FM algorithm in Mt-KaHyPar-D, we obtain a configuration that is slightly faster than Metis-K, while the edge cuts are comparable (see Figure 31 in Appendix A).

Mt-KaHyPar-Q-F (5.22s) is faster than KaFFPa-EcoS (10.51s) and produces better edge cuts by 2.9% in the median. The differences between the edge cuts computed by Mt-KaHyPar-Q-F and KaFFPa-Strong (162.83s) are not statistically significant ($Z = -2.3101$ and $p = 0.02088$). Out of all

Table 4. Geometric mean running times of different sequential (hyper)graph partitioning algorithm and Mt-KaHyPar-D/-Q-F with an increasing number of threads on set set M_G (left) and M_{HG} (right)

Seq. Algo.	$t[s]$	Threads	Mt-KaHyPar -D	Mt-KaHyPar -Q-F	Seq. Algo.	$t[s]$	Threads	Mt-KaHyPar -D	Mt-KaHyPar -Q-F
Metis-K	0.39	16	0.45	4.23	PaToH-D	1.17	16	0.74	3.98
Metis-R	0.55	10	0.55	5.22	PaToH-Q	5.86	10	0.88	5.08
KaFFPa-Fast	1.69	8	0.61	5.74	Mondriaan	6.62	8	1.08	5.58
Scotch	1.84	4	0.98	8.98	KaHyPar-CA	28.14	4	1.83	9.07
KaFFPa-FastS	1.88	2	1.69	15.64	*r*KaHyPar	46.10	2	3.33	16.04
KaFFPa-EcoS	10.51	1	3.00	28.56	*k*KaHyPar	48.98	1	6.24	29.52
KaFFPa-Eco	10.94				hMetis-K	73.75			
KaFFPa-Strong	162.83				hMetis-R	93.21			
KaFFPa-StrongS	201.99								

tested algorithms, KaFFPa-StrongS (201.99s) is the only algorithm producing slightly better edge cuts than Mt-KaHyPar-Q-F (median improvement is 1%). However, this comes at the cost of a 38.66 times longer running time on average, making the quality improvement questionable in practice.

As we have seen, Mt-KaHyPar-D is faster than most of the sequential algorithms using ten threads. This raises the question of whether or not the result still holds when we use fewer threads. We therefore compare the running times of Mt-KaHyPar-D/-Q-F with an increasing number of threads to the different sequential algorithms on set M_G and M_{HG} in Table 4.[10] On set M_G, Mt-KaHyPar-D is faster than most of the sequential algorithms using two threads. Metis-K is still faster than Mt-KaHyPar-D, but their running times become comparable when we use 16 threads. The sequential time of Mt-KaHyPar-Q-F is almost an order of magnitude faster than the running time of the best sequential partitioner KaFFPa-StrongS, and it becomes faster than KaFFPa-EcoS when we use 4 threads. On set M_{HG}, we have to run Mt-KaHyPar-D with 8 threads to achieve comparable speed to PaToH-D. However, this number decreases to 2 threads when we compare their running times on the larger instances of set L_{HG} [49, see Figure 4.17]. The sequential time of Mt-KaHyPar-D is comparable to PaToH-Q, and Mt-KaHyPar-Q-F is significantly faster than its sequential counterpart *k*KaHyPar when we use only one thread.

Comparison to Parallel Systems. Figure 18 compares Mt-KaHyPar to the hypergraph partitioners Zoltan (distributed-memory), BiPart (deterministic shared-memory), and PaToH (sequential) on set L_{HG}. Note that PaToH-D is fast enough to conduct the experiments on set L_{HG} in a reasonable time frame, while this is not the case for any of the other sequential partitioners. Despite the fact that Zoltan has been shown to outperform BiPart (see Table 3), we have included it for a direct comparison to our deterministic configuration Mt-KaHyPar-SDet.

The median improvement of Mt-KaHyPar-SDet (geometric mean running time 3.14s) over BiPart (29.19s) – the only existing competitor for deterministic partitioning – is 200%, while it is almost an order of magnitude faster. Our deterministic algorithm also outperforms Zoltan (12.63s, median improvement is 12%) and the Wilcoxon signed-ranked test reveals that there is no statistically significant difference between the solutions produced by Mt-KaHyPar-SDet and PaToH-D (51.2s, $Z = 1.7314$ and $p = 0.08337$).

Mt-KaHyPar-D (4.64s) is slightly slower than Mt-KaHyPar-SDet, but it computes solutions that are 23% resp. 6.6% better than those of Zoltan resp. PaToH-D in the median and is still significantly faster than both algorithms. When flow-based refinement is used (Mt-KaHyPar-D-F, not shown

[10]Note that increasing the number of threads does not affect the solution quality of Mt-KaHyPar-D/-Q-F, as shown in Figure 14.

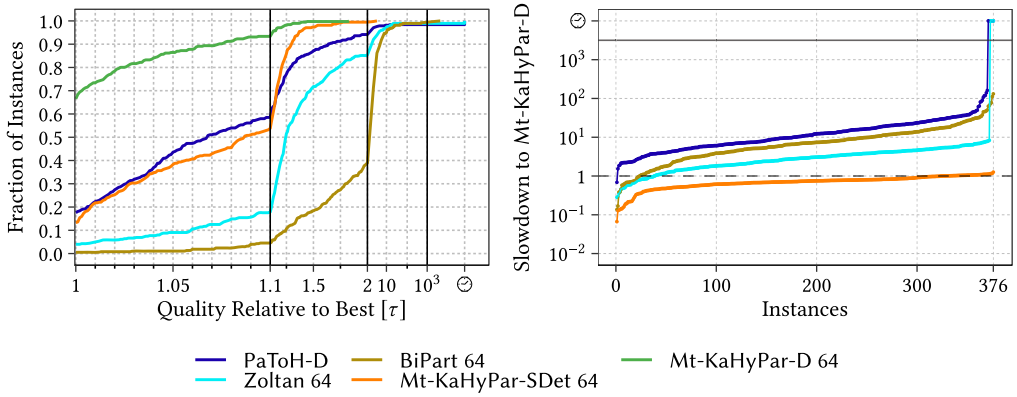

Fig. 18. Performance profiles and running times comparing Mt-KaHyPar to PaToH and Zoltan on set L_{HG}.

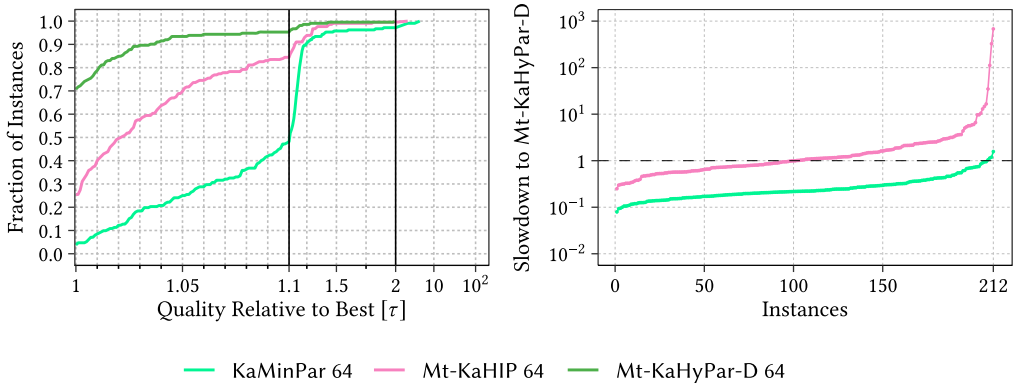

Fig. 19. Performance profiles and running times comparing Mt-KaHyPar to KaMinPar and Mt-KaHIP on set L_G.

in the plots), we achieve a median improvement over Zoltan of 34%. This shows that Mt-KaHyPar can partition extremely large hypergraph with high solution quality, which was previously only possible with sequential codes on medium-sized instances.

Figure 19 compares Mt-KaHyPar to the parallel graph partitioner KaMinPar (shared-memory) and Mt-KaHIP (shared-memory, also implements a parallel version of the FM algorithm) on set L_G. We can see that Mt-KaHyPar-D (10.8s) computes on most of the instances the best solutions. The median improvement of Mt-KaHyPar-D over Mt-KaHIP (13.69s) is 2.1%, while it is also slightly faster. Out of all tested algorithms, KaMinPar (2.69s) is the only algorithm that is faster than Mt-KaHyPar-D, but the edge cuts produced by KaMinPar are worse than those of Mt-KaHyPar-D by 9.9% in the median. On larger graph instances, KaMinPar is the method of choice when speed is more important than quality, and Mt-KaHyPar should be used if one aims for high solution quality.

Limitations. In this study, we partitioned (hyper)graphs in up to 128 blocks with an allowed imbalance of $\varepsilon = 3\%$. We want to point out that there are still settings where the results of this evaluation do not apply. For example, KaMinPar is specifically designed for partitioning graphs into a large number of blocks (e.g., $k \in O(\sqrt{n})$). In this setting, existing algorithms struggle to find balanced solutions or do not complete in a reasonable time frame [48]. We are integrating KaMinPar's deep multilevel partitioning scheme in Mt-KaHyPar and hope to offer support for very large k in

the near future. Another limitation is the restriction of our algorithms to running in-memory on a single machine, and thus instances are restricted to the size of currently available RAM. Finally, partitioning (hyper)graphs with a tight balance constraint (e.g., $\varepsilon \approx 0$) poses additional challenges for traditional refinement algorithms as this drastically reduces the set of possible moves.

13 CONCLUSION

We have presented the *first* set of shared-memory algorithms for partitioning hypergraphs. Our solver Mt-KaHyPar produces solutions on par with the best sequential codes, while it is faster than most of the existing parallel algorithms. We demonstrated this achievement in our extensive experimental evaluation with 25 sequential and parallel graph and hypergraph partitioners tested on over 800 (hyper)graphs. We contributed parallel formulations for all phases of the multilevel scheme: a parallel clustering-based coarsening algorithm guided by the community structure of the input hypergraph obtained via a parallel community detection algorithm, initial partitioning via parallel recursive bipartitioning using work-stealing, the first fully-parallel FM implementation, and a parallelization of flow-based refinement. Perhaps the most suprising result is the efficient parallelization of the n-level partitioning scheme, even though we showed that traditional multilevel algorithms can compute comparable solutions when flow-based refinement is used. Furthermore, we presented multiple techniques to accurately (re)compute gain values for concurrent node moves, which had not been addressed in parallel partitioning algorithms before. We also proposed data structure optimizations for plain graphs, making Mt-KaHyPar the state-of-the-art solver for graph partitioning. Additionally, we devised a deterministic version of our multilevel algorithm based on the synchronous local moving scheme.

Given that quality improvements often come at the cost of significantly longer running times, it may be interesting to evaluate the quality-time trade-off of existing tools for applications before advancing the field of high-quality partitioning. For instances that do not fit into the main memory of a single machine, translating the techniques presented in this work into the distributed-memory setting is also an important area for future research. We see further algorithmic improvements in a localized version of flow-based refinement that runs after each batch uncontraction in the n-level scheme as well asimproving clustering decisions in the coarsening phase.

APPENDIX

A COMPARISON TO OTHER SYSTEMS

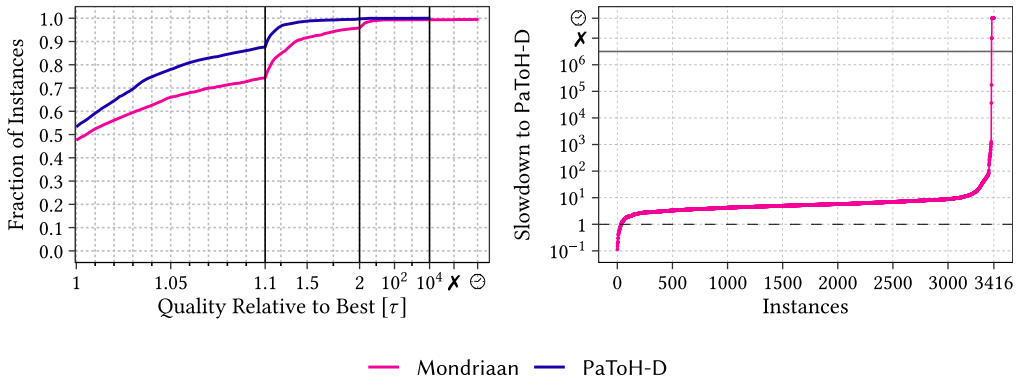

Fig. 20. Performance profiles and running times comparing PaToH-D and Mondriaan on set M_{HG}.

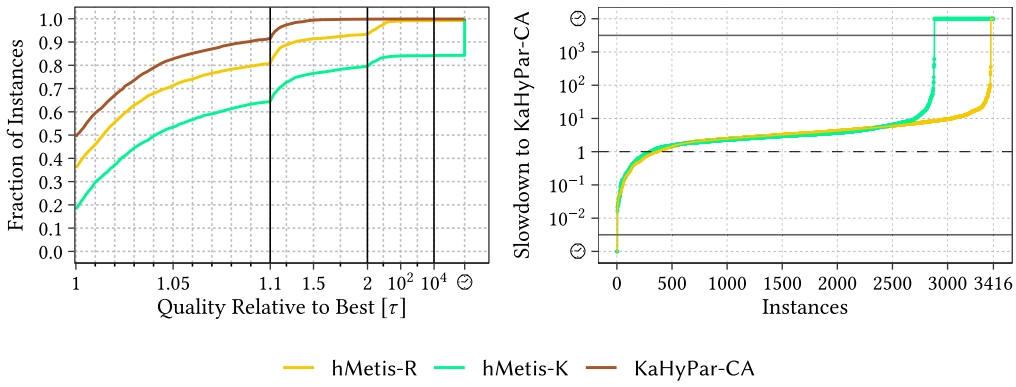

Fig. 21. Performance profiles and running times comparing KaHyPar-CA and hMetis on set M_{HG}.

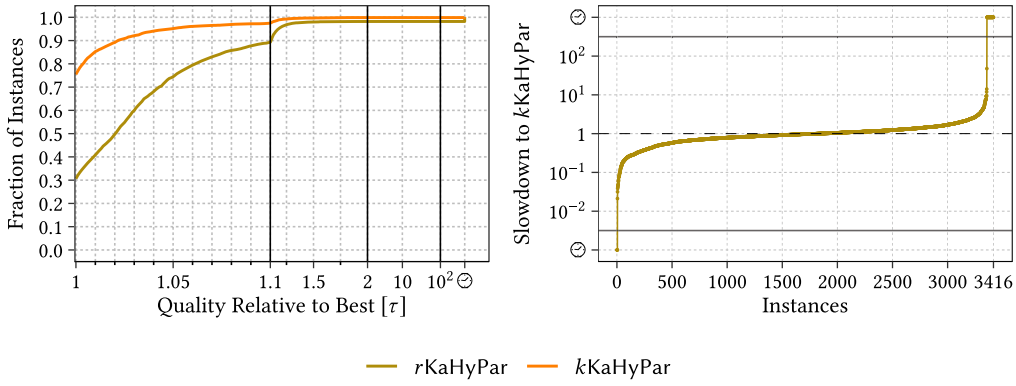

Fig. 22. Performance profiles and running times comparing rKaHyPar and kKaHyPar on set M_{HG}.

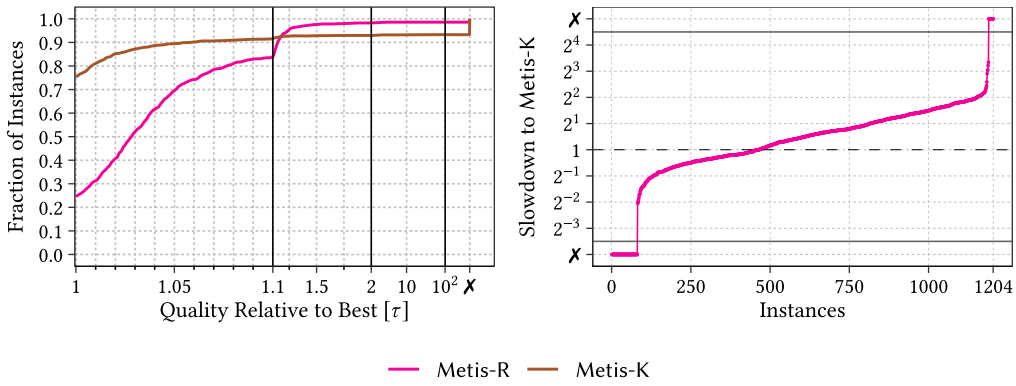

Fig. 23. Performance profiles and running times comparing Metis-R and Metis-K on set M_G.

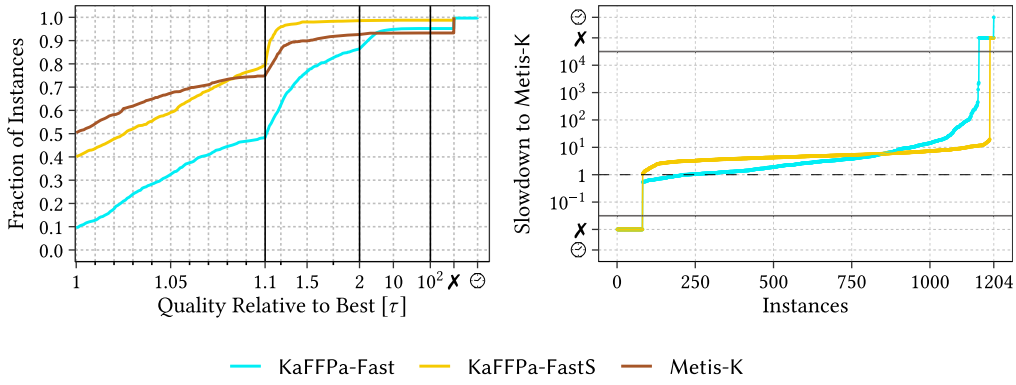

Fig. 24. Performance profiles and running times comparing Metis-K and KaFFPa-Fast/-FastS on set M_G.

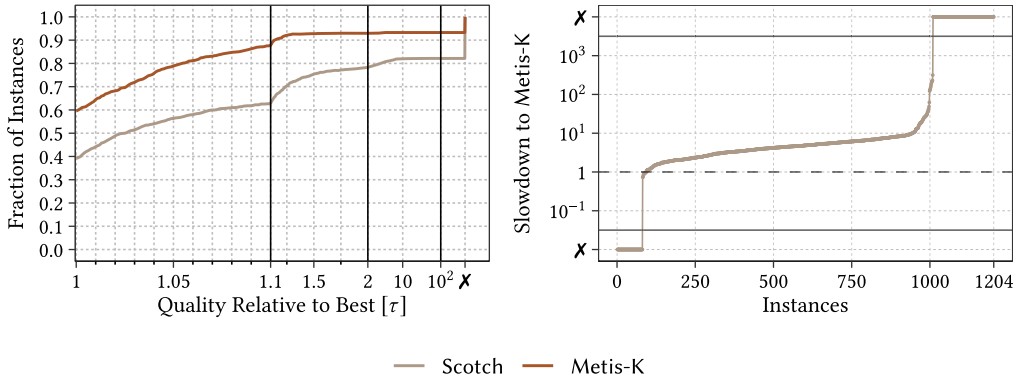

Fig. 25. Performance profiles and running times comparing Metis-K and Scotch on set M_G.

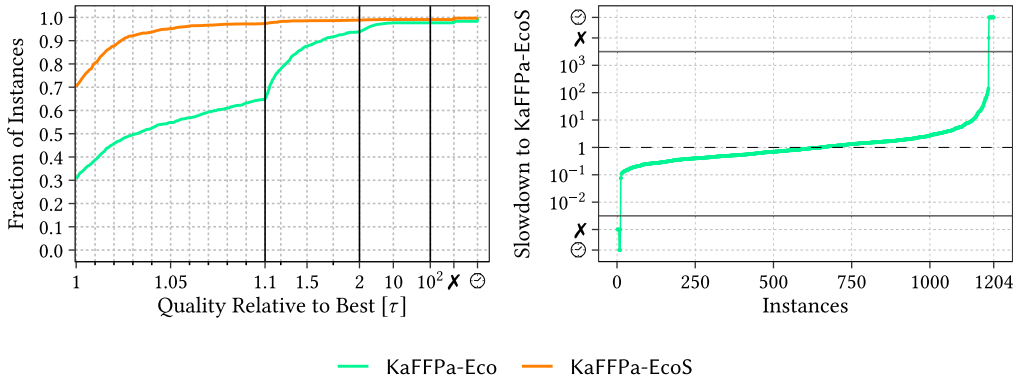

Fig. 26. Performance profiles and running times comparing KaFFPa-Eco and KaFFPa-EcoS on set M_G.

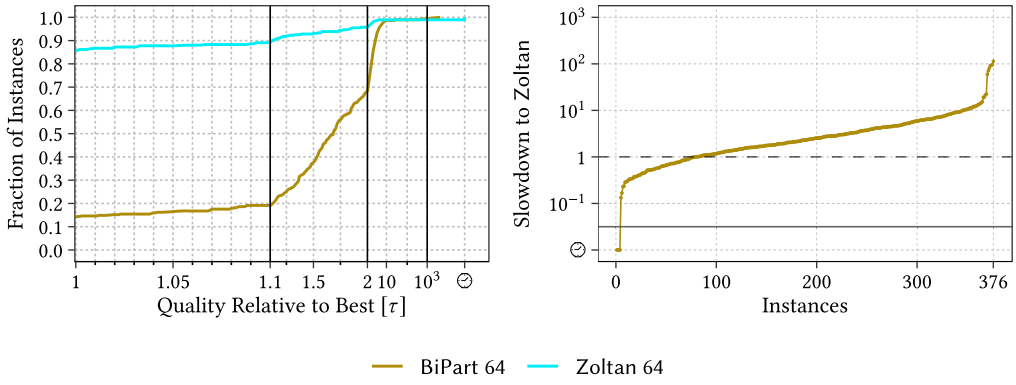

Fig. 27. Performance profiles and running times comparing Zoltan and BiPart on set L_{HG}.

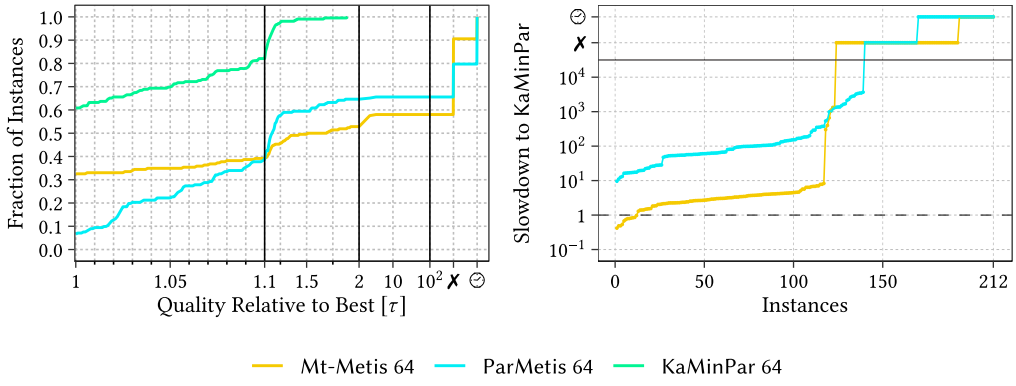

Fig. 28. Performance profiles and running times comparing KaMinPar to Mt-Metis and ParMetis on set L_G.

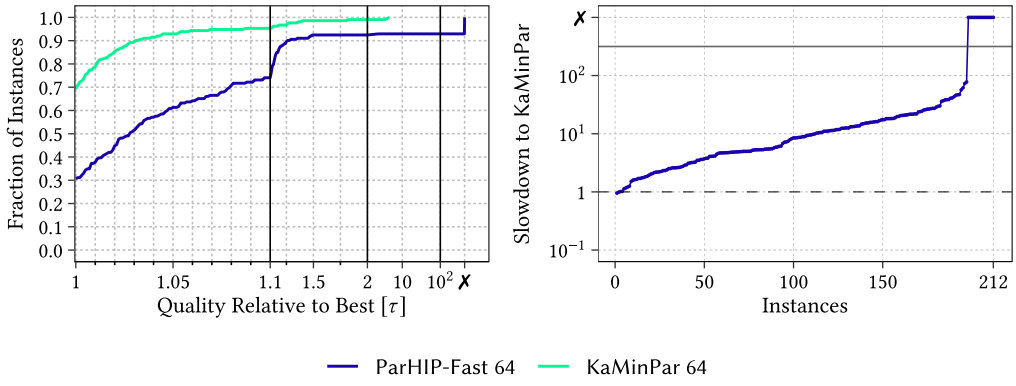

Fig. 29. Performance profiles and running times comparing KaMinPar to ParHIP-Fast on set L_G.

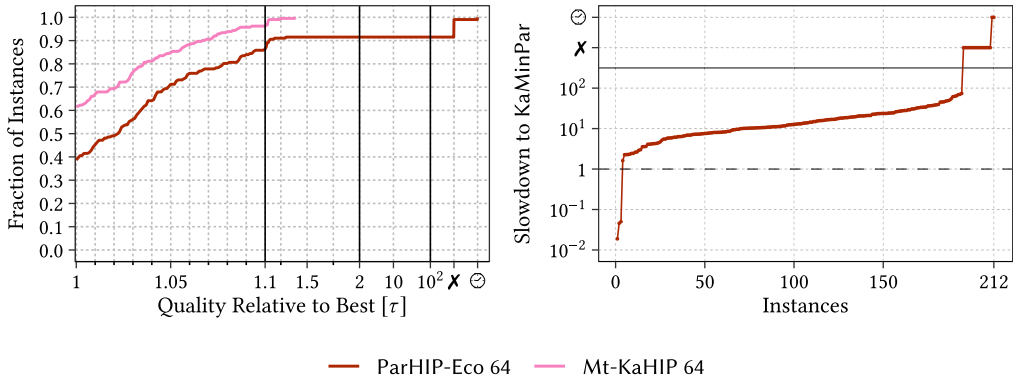

Fig. 30. Performance profiles and running times comparing Mt-KaHIP to ParHIP-Eco on set L_G.

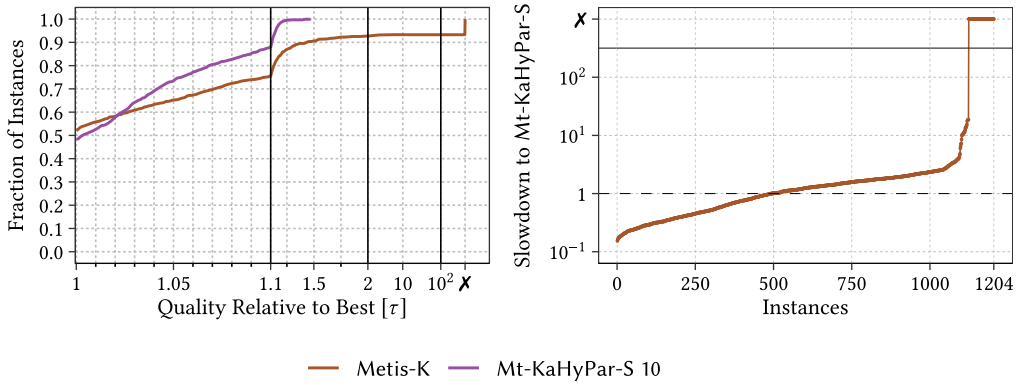

Fig. 31. Performance profiles and running times comparing Mt-KaHyPar-S (**S**peed, Mt-KaHyPar-D without FM refinement) to Metis-K on set M_G.

REFERENCES

[1] Amine Abou-Rjeili and George Karypis. 2006. Multilevel algorithms for partitioning power-law graphs. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. https://doi.org/10.1109/IPDPS.2006.1639360

[2] Yaroslav Akhremtsev. 2019. *Parallel and External High Quality Graph Partitioning*. Dissertation. Karlsruhe Institute of Technology.

[3] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2017. Engineering a direct k-way hypergraph partitioning algorithm. In *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 28–42. https://doi.org/10.1137/1.9781611974768.3

[4] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. 2017. High-quality shared-memory graph partitioning. In *European Conference on Parallel Processing (Euro-Par)*. Springer, 659–671. https://doi.org/10.1007/978-3-319-96983-1_47

[5] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. 2004. MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation. *The International Journal of Universal Computer Science* 10, 12 (2004), 1562–1596. https://doi.org/10.3217/jucs-010-12-1562

[6] Charles J. Alpert. 1998. The ISPD98 circuit benchmark suite. In *International Symposium on Physical Design (ISPD)*. 80–85. https://doi.org/10.1145/274535.274546

[7] Charles J. Alpert, Jsen-Hsin Huang, and Andrew B. Kahng. 1997. Multilevel circuit partitioning. In *34th Conference on Design Automation (DAC)*. 530–533. https://doi.org/10.1145/266021.266275

[8] Charles J. Alpert and Andrew B. Kahng. 1995. Recent directions in netlist partitioning: A survey. *Integration: The VLSI Journal* 19, 1–2 (1995), 1–81. https://doi.org/10.1016/0167-9260(95)00008-4

[9] Richard J. Anderson and João C. Setubal. 1995. A parallel implementation of the push-relabel algorithm for the maximum flow problem. *J. Parallel and Distrib. Comput.* 29, 1 (1995), 17–26. https://doi.org/10.1006/jpdc.1995.1103
[10] Robin Andre, Sebastian Schlag, and Christian Schulz. 2018. Memetic multilevel hypergraph partitioning. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 347–354. https://doi.org/10.1145/3205455.3205475
[11] Pablo Andres-Martinez and Chris Heunen. 2019. Automated distribution of quantum circuits via hypergraph partitioning. *Physical Review A* 100, 3 (2019), 1–11.
[12] Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. 2008. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel Distributed Computing* 68, 5 (2008), 609–625. https://doi.org/10.1016/j.jpdc.2007.09.006
[13] David Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner (Eds.). 2013. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*. Contemporary Mathematics, Vol. 588. American Mathematical Society.
[14] Stephen T. Barnard and Horst D. Simon. 1993. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *6th SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*. 711–718.
[15] Niklas Baumstark, Guy E. Blelloch, and Julian Shun. 2015. Efficient implementation of a synchronous parallel push-relabel algorithm. In *23rd European Symposium on Algorithms (ESA)*, Vol. 9294. Springer, 106–117. https://doi.org/10.1007/978-3-662-48350-3_10
[16] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. 2014. The SAT Competition 2014. http://www.satcompetition.org/2014/
[17] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *PPoPP 2012*. https://doi.org/10.1145/2145816.2145840
[18] Vincent D. Blondel, Jean Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 10 (2008).
[19] Robert L. Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. 2009. Parallel programming must be deterministic by default. *Usenix HotPar* 6 (2009).
[20] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. 2008. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering* 20, 2 (2008), 172–188. https://doi.org/10.1109/TKDE.2007.190689
[21] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. Recent advances in graph partitioning. In *Algorithm Engineering - Selected Results and Surveys*. Vol. 9220. 117–158. https://doi.org/10.1007/978-3-319-49487-6_4
[22] Michael J. Campbell and Thomas D. V. Swinscow. 2009. *Statistics at Square One*. BMJ Publishing Group.
[23] Ümit V. Çatalyürek and Cevdet Aykanat. 2001. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *15th International Parallel and Distributed Processing Symposium (IPDPS)*. 118. https://doi.org/10.1109/IPDPS.2001.925093
[24] Ümit V. Çatalyürek and Cevdet Aykanat. 2001. A hypergraph-partitioning approach for coarse-grain decomposition. In *ACM/IEEE Conference on Supercomputing*. ACM, 28. https://doi.org/10.1145/582034.582062
[25] Ümit V. Çatalyürek and Cevdet Aykanat. 2011. *PaToH: Partitioning Tool for Hypergraphs*.
[26] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. 2012. Multithreaded clustering for multi-level hypergraph partitioning. In *26th International Parallel and Distributed Processing Symposium (IPDPS)*. 848–859. https://doi.org/10.1109/IPDPS.2012.81
[27] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. 2022. More recent advances in (hyper)graph partitioning. *Computing Research Repository (CoRR)* abs/2205.13202 (2022). arXiv:2205.13202
[28] Ümit V. Catalyurek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (1999), 673–693. https://doi.org/10.1109/71.780863
[29] Boris V. Cherkassky and Andrew V. Goldberg. 1997. On implementing the push-relabel method for the maximum flow problem. *Algorithmica* 19, 4 (1997), 390–410. https://doi.org/10.1007/PL00009180
[30] Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Comput.* 34, 6–8 (2008), 318–331. https://doi.org/10.1016/j.parco.2007.12.001
[31] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. 2010. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment* 3, 1 (2010), 48–57. https://doi.org/10.14778/1920841.1920853
[32] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (11 2011), 1:1–1:25. https://doi.org/10.1145/2049662.2049663

[33] Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. 2013. Hypergraph sparsification and its application to partitioning. In *42nd International Conference on Parallel Processing (ICPP)*. 200–209. https://doi.org/10.1109/ICPP.2013.29

[34] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Çatalyürek. 2006. Parallel hypergraph partitioning for scientific computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. https://doi.org/10.1109/IPDPS.2006.1639359

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

[36] Andreas E. Feldmann. 2013. Fast balanced partitioning is hard even on grids and trees. *Theoretical Computer Science* 485 (2013), 61–68. https://doi.org/10.1016/j.tcs.2013.03.014

[37] Charles M. Fiduccia and Robert M. Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *19th Conference on Design Automation (DAC)*. 175–181. https://doi.org/10.1145/800263.809204

[38] Lester Randolph Ford and Delbert R. Fulkerson. 1956. Maximal flow through a network. *Canadian Journal of Mathematics* 8 (1956), 399–404. https://doi.org/10.4153/CJM-1956-045-5

[39] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. 2018. Communication-free massively distributed graph generation. In *32nd International Parallel and Distributed Processing Symposium (IPDPS)*. 336–347. https://doi.org/10.1109/IPDPS.2018.00043

[40] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Vol. 174. W. H. Freeman.

[41] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. 1976. Some simplified NP-complete graph problems. *Theoretical Computer Science* 1, 3 (1976), 237–267. https://doi.org/10.1016/0304-3975(76)90059-1

[42] Andrew V. Goldberg and Robert Endre Tarjan. 1988. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)* 35, 4 (1988), 921–940. https://doi.org/10.1145/48014.61051

[43] Lars Gottesbüren and Michael Hamann. 2022. Deterministic parallel hypergraph partitioning. In *Euro-Par 2022: Parallel Processing - 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22–26, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13440)*. Springer, 301–316. https://doi.org/10.1007/978-3-031-12597-3_19

[44] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. 2020. Advanced flow-based multilevel hypergraph partitioning. *18th International Symposium on Experimental Algorithms (SEA)* (2020). https://doi.org/10.4230/LIPIcs.SEA.2020.11

[45] Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. 2019. Evaluation of a flow-based hypergraph bipartitioning algorithm. In *27th European Symposium on Algorithms (ESA)*. 52:1–52:17. https://doi.org/10.4230/LIPIcs.ESA.2019.52

[46] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. 2022. Parallel flow-based hypergraph partitioning. In *20th International Symposium on Experimental Algorithms (SEA) (LIPIcs, Vol. 233)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:21. https://doi.org/10.4230/LIPIcs.SEA.2022.5

[47] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2022. Shared-memory n-level hypergraph partitioning. In *24th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 131–144. https://doi.org/10.1137/1.9781611977042.11

[48] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. 2021. Deep multilevel graph partitioning. In *29th European Symposium on Algorithms (ESA) (LIPIcs, Vol. 204)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 48:1–48:17. https://doi.org/10.4230/LIPIcs.ESA.2021.48

[49] Lars Gottesbüren. 2022. *Parallel and Flow-Based High-Quality Hypergraph Partitioning*. Ph. D. Dissertation. Karlsruhe Institute of Technology.

[50] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2021. Scalable shared-memory hypergraph partitioning. In *23rd Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 16–30. https://doi.org/10.1137/1.9781611976472.2

[51] Johnnie Gray and Stefanos Kourtis. 2021. Hyper-optimized tensor network contraction. *Quantum* 5 (2021), 410. https://doi.org/10.22331/q-2021-03-15-410

[52] Lars W. Hagen, Dennis J.-H. Huang, and Andrew B. Kahng. 1997. On implementation choices for iterative improvement partitioning algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 16, 10 (1997), 1199–1205. https://doi.org/10.1109/43.662682

[53] Michael Hamann and Ben Strasser. 2018. Graph bisection with Pareto optimization. *ACM Journal of Experimental Algorithmics (JEA)* 23 (2018). https://doi.org/10.1145/3173045

[54] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. 2018. Distributed graph clustering using modularity and map equation. In *European Conference on Parallel Processing (Euro-Par)*. 688–702. https://doi.org/10.1007/978-3-319-96983-1_49

[55] Scott Hauck and Gaetano Borriello. 1995. An evaluation of bipartitioning techniques. In *16th Conference on Advanced Research in VLSI (ARVLSI)*. 383–403.
[56] Bruce Hendrickson and Robert W. Leland. 1995. A multi-level algorithm for partitioning graphs. In *Supercomputing*. ACM, 28. https://doi.org/10.1145/224170.224228
[57] Alexandra Henzinger, Alexander Noe, and Christian Schulz. 2020. ILP-based local search for graph partitioning. *ACM Journal of Experimental Algorithmics (JEA)* 25 (2020), 1–26. https://doi.org/10.1145/3398634
[58] Tobias Heuer. 2015. *Engineering Initial Partitioning Algorithms for Direct k-way Hypergraph Partitioning*. Bachelor Thesis. Karlsruhe Institute of Technology.
[59] Tobias Heuer. 2018. *High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations*. Master Thesis. Karlsruhe Institute of Technology.
[60] Tobias Heuer. 2022. *Scalable High-Quality Graph and Hypergraph Partitioning*. Ph. D. Dissertation. Karlsruhe Institute of Technology. https://doi.org/10.5445/IR/1000152872
[61] Tobias Heuer, Nikolai Maas, and Sebastian Schlag. 2021. Multilevel hypergraph partitioning with vertex weights revisited. In *19th International Symposium on Experimental Algorithms (SEA) (LIPIcs, Vol. 190)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:20. https://doi.org/10.4230/LIPIcs.SEA.2021.8
[62] Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2019. Network flow-based refinement for multilevel hypergraph partitioning. *ACM Journal of Experimental Algorithmics (JEA)* 24, 1 (09 2019), 2.3:1–2.3:36. https://doi.org/10.1145/3329872
[63] Tobias Heuer and Sebastian Schlag. 2017. Improving coarsening schemes for hypergraph partitioning by exploiting community structure. In *16th International Symposium on Experimental Algorithms (SEA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 21:1–21:19. https://doi.org/10.4230/LIPIcs.SEA.2017.21
[64] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. 2010. Engineering a scalable high quality graph partitioner. In *24th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1–12. https://doi.org/10.1109/IPDPS.2010.5470485
[65] T. C. Hu and K. Moerder. 1985. Multiterminal flows in a hypergraph. In *VLSI Circuit Layout: Theory and Design*. IEEE, Chapter 3, 87–93.
[66] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. 2017. Social hash partitioner: A scalable distributed hypergraph partitioner. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1418–1429. https://doi.org/10.14778/3137628.3137650
[67] Gökçehan Kara and Can C. Özturan. 2019. Graph coloring based parallel push-relabel algorithm for the maximum flow problem. *ACM Trans. Math. Software* 45, 4 (2019), 46:1–46:28. https://doi.org/10.1145/3330481
[68] George Karypis. 2003. Multilevel hypergraph partitioning. In *Multilevel Optimization in VLSICAD*. Springer, 125–154.
[69] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1999. Multilevel hypergraph partitioning: Applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7, 1 (1999), 69–79. https://doi.org/10.1109/92.748202
[70] George Karypis and Vipin Kumar. 1996. Parallel multilevel k-way partitioning scheme for irregular graphs. In *ACM/IEEE Conference on Supercomputing*. 35. https://doi.org/10.1109/SC.1996.32
[71] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997
[72] George Karypis and Vipin Kumar. 1998. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel and Distrib. Comput.* 48, 1 (1998), 96–129. https://doi.org/10.1006/jpdc.1997.1404
[73] George Karypis and Vipin Kumar. 2000. Multilevel k-way hypergraph partitioning. *VLSI Design* 2000, 3 (2000), 285–300. https://doi.org/10.1155/2000/19436
[74] Alexander V. Karzanov. 1974. Determining the maximal flow in a network by the method of preflows. In *Soviet Mathematics Doklady*, Vol. 15. 434–437.
[75] Enver Kayaaslan, Ali Pinar, Ümit V. Çatalyürek, and Cevdet Aykanat. 2012. Partitioning hypergraphs in scientific computing applications through vertex separators on graphs. *SIAM Journal on Scientific Computing* 34, 2 (2012). https://doi.org/10.1137/100810022
[76] Brian W. Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal* 49, 2 (2 1970), 291–307. https://doi.org/10.1002/j.1538-7305.1970.tb01770.x
[77] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-efficient graph processing on GPUs. In *International Conference on Parallel Architectures and Compilation (PACT)*. 39–50. https://doi.org/10.1109/PACT.2015.15
[78] K. Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. 2014. SWORD: Workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal* 23, 6 (2014), 845–870. https://doi.org/10.1007/s00778-014-0362-1
[79] University of Milano Laboratory of Web Algorithms. [n. d.]. Datasets. http://law.di.unimi.it/datasets.php

[80] Jesper Larsson Träff. 2006. Direct graph k-partitioning with a Kernighan–Lin like heuristic. *Operations Research Letters* 34, 6 (Nov. 2006), 621–629. https://doi.org/10.1016/j.orl.2005.10.003
[81] Dominique Lasalle and George Karypis. 2013. Multi-threaded graph partitioning. In *27th International Parallel and Distributed Processing Symposium (IPDPS)*. 225–236. https://doi.org/10.1109/IPDPS.2013.50
[82] Dominique LaSalle and George Karypis. 2016. A parallel hill-climbing refinement algorithm for graph partitioning. In *45th International Conference on Parallel Processing (ICPP)*. 236–241. https://doi.org/10.1109/ICPP.2016.34
[83] Dominique LaSalle, Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. 2015. Improving graph partitioning for modern graphs and architectures. In *5th Workshop on Irregular Applications - Architectures and Algorithms IA3*. 14:1–14:4. https://doi.org/10.1145/2833179.2833188
[84] Eugene L. Lawler. 1973. Cutsets and partitions of hypergraphs. *Networks* 3, 3 (1973), 275–285. https://doi.org/10.1002/net.3230030306
[85] Edward A. Lee. 2006. The problem with threads. *Computer* 39, 5 (2006), 33–42. https://doi.org/10.1109/MC.2006.180
[86] Thomas Lengauer. 1990. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons. https://doi.org/10.1017/S0263574700015691
[87] J. Leskovec and A. Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data
[88] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. 2021. BiPart: A parallel and deterministic hypergraph partitioner. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 161–174. https://doi.org/10.1145/3437801.3441611
[89] Zoltán Á. Mann and Pál A. Papp. 2014. Formula partitioning revisited. In *5th Pragmatics of SAT Workshop*. 41–56. https://doi.org/10.29007/9skn
[90] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. 2008. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. In *22nd International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1–13. https://doi.org/10.1109/IPDPS.2008.4536237
[91] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2625–2638. https://doi.org/10.1109/TPDS.2017.2671868
[92] Mark E. J. Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical Review* 69 (2 2004). Issue 2.
[93] Vitaly Osipov and Peter Sanders. 2010. n-level graph partitioning. In *18th European Symposium on Algorithms (ESA)*. Springer, 278–289. https://doi.org/10.1007/978-3-642-15775-2_24
[94] David A. Papa and Igor L. Markov. 2007. Hypergraph partitioning and clustering. In *Handbook of Approximation Algorithms and Metaheuristics*. https://doi.org/10.1201/9781420010749.ch61
[95] François Pellegrini and Jean Roman. 1996. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking (HPCN)*, Vol. 1067. Springer, 493–498. https://doi.org/10.1007/3-540-61142-8_588
[96] Chuck Pheatt. 2008. Intel threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.
[97] Jean-Claude Picard and Maurice Queyranne. 1980. On the structure of all minimum cuts in a network and applications. *Combinatorial Optimization II* (1980), 8–16. https://doi.org/10.1007/BF01581031
[98] R. A. Rutman. 1964. An algorithm for placement of interconnected elements based on minimum wire length. In *Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS)*. ACM, 477–491.
[99] Youssef Saab. 1995. A fast and robust network bisection algorithm. *IEEE Trans. Comput.* 44, 7 (1995), 903–913. https://doi.org/10.1109/12.392848
[100] Laura A. Sanchis. 1989. Multiple-way network partitioning. *IEEE Trans. Comput.* 38, 1 (1989), 62–81. https://doi.org/10.1109/12.8730
[101] Peter Sanders and Christian Schulz. 2011. Engineering multilevel graph partitioning algorithms. In *19th European Symposium on Algorithms (ESA)*. Springer, 469–480. https://doi.org/10.1007/978-3-642-23719-5_40
[102] Peter Sanders and Christian Schulz. 2012. Distributed evolutionary graph partitioning. In *12th Workshop on Algorithm Engineering & Experiments (ALENEX)*. 16–29. https://doi.org/10.1137/1.9781611972924.2
[103] John E. Savage and Markus G. Wloka. 1991. Parallelism in graph-partitioning. *J. Parallel and Distrib. Comput.* 13, 3 (1991), 257–272. https://doi.org/10.1016/0743-7315(91)90074-J
[104] Sebastian Schlag. 2020. *High-Quality Hypergraph Partitioning*. Ph. D. Dissertation. Karlsruhe Institute of Technology. https://doi.org/10.5445/IR/1000105953
[105] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. k-way hypergraph partitioning via n-level recursive bisection. In *18th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 53–67. https://doi.org/10.1137/1.9781611974317.5

[106] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2022. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics (JEA)* (Mar. 2022). https://doi.org/10.1145/3529090 Just accepted.

[107] C. Schulz. 2013. *High Quality Graph Partitioning*. Ph. D. Dissertation. Karlsruhe Institute of Technology.

[108] Daniel G. Schweikert and Brian W. Kernighan. 1972. A proper model for the partitioning of electrical circuits. In *9th Conference on Design Automation (DAC)*. ACM, 57–62. https://doi.org/10.1145/800153.804930

[109] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456. https://doi.org/10.14778/3025111.3025125

[110] Yossi Shiloach and Uzi Vishkin. 1982. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms* 3, 2 (1982), 128–146. https://doi.org/10.1016/0196-6774(82)90013-X

[111] Christian L. Staudt and Henning Meyerhenke. 2016. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (01 2016), 171–184. https://doi.org/10.1109/TPDS.2015.2390633

[112] Guy L. Steele. 1990. Making asynchronous parallelism safe for the world. In *POPL 90*, Frances E. Allen (Ed.). ACM Press, 218–231. https://doi.org/10.1145/96709.96731

[113] Aleksandar Trifunovic and William J. Knottenbelt. 2004. Parkway 2.0: A parallel multilevel hypergraph partitioning tool. In *19th International Symposium on Computer and Information Sciences (ISCIS)*, Vol. 3280. Springer, 789–800. https://doi.org/10.1007/978-3-540-30182-0_79

[114] Aleksandar Trifunovic and William J. Knottenbelt. 2004. Towards a parallel disk-based algorithm for multilevel k-way hypergraph partitioning. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS.2004.1303286

[115] Brendan Vastenhouw and Rob H. Bisseling. 2005. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.* 47, 1 (2005), 67–95. https://doi.org/10.1137/S0036144502409019

[116] Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. 2012. The DAC 2012 routability-driven placement contest and benchmark suite. In *49th Conference on Design Automation (DAC)*. ACM, 774–782. https://doi.org/10.1145/2228360.2228500

[117] C. Walshaw. 2003. *An Exploration of Multilevel Combinatorial Optimisation*. Springer, 71–124.

[118] C. Walshaw. 2004. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research* 131, 1–4 (2004), 325–372. https://doi.org/10.1023/B:ANOR.0000039525.80601.15

[119] Chris Walshaw and Mark Cross. 2000. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM Journal on Scientific Computing* 22, 1 (2000), 63–80. https://doi.org/10.1137/S1064827598337373

[120] Chris Walshaw and Mark Cross. 2000. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Comput.* 26, 12 (2000), 1635–1660. https://doi.org/10.1016/S0167-8191(00)00046-6

[121] Chris Walshaw, Mark Cross, and Martin G. Everett. 1997. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel and Distrib. Comput.* 47, 2 (1997), 102–108. https://doi.org/10.1006/jpdc.1997.1407

[122] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics*. Springer, 196–202. https://doi.org/10.1007/978-1-4612-4380-9_16

[123] Samuel Williams, Leonid Oliker, Richard W. Vuduc, John Shalf, Katherine A. Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM Press, 38. https://doi.org/10.1145/1362622.1362674

[124] Hannah H. Yang and D. F. Wong. 1996. Efficient network flow based min-cut balanced partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 15, 12 (1996), 1533–1540. https://doi.org/10.1007/978-1-4615-0292-0_41

[125] Wenyin Yang, Guojun Wang, Kim-Kwang Raymond Choo, and Shuhong Chen. 2018. HEPart: A balanced hypergraph partitioning algorithm for big data applications. *Future Generation Computer Systems* 83 (2018), 250–268. https://doi.org/10.1016/j.future.2018.01.009

[126] Boyang Yu and Jianping Pan. 2015. Location-aware associated data placement for geo-distributed data-intensive applications. In *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 603–611. https://doi.org/10.1109/INFOCOM.2015.7218428

Received 20 February 2023; revised 18 September 2023; accepted 24 September 2023