High-Quality Multi-Constraint Hypergraph Partitioning via Greedy Rebalancing

Nikolai Maas
Karlsruhe Institute of Technology, Germany

Abstract

Multi-constraint hypergraph partitioning is a generalization of balanced partitioning, where the vertex set of a hypergraph is partitioned such that the inter-block connectivity of hyperedges is minimized while balancing the vertices with regard to d distinct constraints. A prominent class of applications is data distribution tasks, where this allows to achieve good load balance for d different kinds of resources and simultaneously minimize the communication volume.

Although the best approaches for single-constraint partitioning are usually complex (multilevel) algorithms with many components, we show that replacing only one component already leads to high-quality multi-constraint partitions: the rebalancing step, which restores balance for a partition that has (hopefully) small connectivity but violates the constraints. We design a multi-constraint rebalancing algorithm based on greedy local search, proving that balance is always restored for $d = 2$ and bounded maximum weight. The key is to ensure monotonically decreasing global imbalance by choosing an imbalance metric where there is always a balance-improving move available. Integrating our algorithm into the state-of-the-art partitioner Mt-KaHyPar, we demonstrate an 11.5 % geometric mean connectivity reduction compared to the next best competitor (Metis) and better reliability regarding partition balance, even though the majority of inputs is outside of the theoretical guarantee.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Hypergraph Partitioning, Multi-Constraint Partitioning, Graph Algorithms, Multilevel Algorithms, Local Search, Vector Scheduling, Multidimensional Load Balancing

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.12

Related Version *Full Version*: `https://arxiv.org/abs/2605.28333` [33]

Supplementary Material *Software (Source Code)*: `https://github.com/kahypar/mt-kahypar/tree/esa2026`, archived at `swh:1:dir:090b1736ccfc167c7c37154a0617c5031e0fc168`
Dataset (Benchmark Set & Experimental Results): `https://zenodo.org/records/19630135`

1 Introduction

Traditional balanced hypergraph partitioning asks for a partition of the vertices of a hypergraph $H = (V, E)$ into k disjoint blocks $V_1, \ldots, V_k \subseteq V$ of roughly equal size $|V_i| \leq (1+\varepsilon)\frac{|V|}{k}$ (the balance constraint) while minimizing the inter-block connectivity of the hyperedges. This is an essential task for many applications, including distributed databases [3, 12], VLSI circuit design [15, 24], and scientific simulations [17]. However, a single balance constraint is often insufficient to model application requirements. For example, multi-phase simulations balance the compute load in each phase while minimizing dependencies between processors. If phases are treated as separate partitioning problems, expensive data migrations are required in between – which can be avoided by computing a global partition with multiple distinct balance constraints [30, 39]. Another application is distributed processing tasks that need to balance both compute and memory requirements. As a concrete example, high-quality partitions significantly improve the efficiency of distributed GNN training [31, 46].

Multi-constraint partitioning models these use cases with d-dimensional vertex weights, thereby expressing d different kinds of balance constraints at once. Despite its practical relevance, there is not much existing work on multi-constraint partitioning. Current approaches mostly use straightforward adjustments of single-constraint partitioning algorithms

and restrict local search procedures to disregard moves that violate any of the d constraints [5, 7, 28, 38]. We believe that these *constrained* search algorithms have a severe limitation; since d balance constraints could shrink the subspace of feasible solutions significantly, it becomes hard for the search to escape from local minima. Instead, we propose to use *unconstrained* search algorithms that allow temporary balance violations, guiding the search through regions of infeasible solutions instead of around them. We already achieved improvements for traditional partitioning via unconstrained search techniques [29, 34], with similar findings by other authors [18, 37]. In this work, we will show the effectiveness of this technique in the multi-constraint setting.

However, allowing balance violations means that the algorithm must restore balance afterwards, thereby alternating between feasible and infeasible solutions. This step is known as *rebalancing* and it is a much more difficult task in the multi-constraint setting than with a single constraint [28]. At its core, it is equivalent to a multi-dimensional job scheduling problem – but we must simultaneously consider the optimization objective. We therefore develop a rebalancing algorithm that efficiently and reliably restores partition balance. The algorithm quantifies the current imbalance with a metric based on the L_1-norm and greedily performs single-vertex moves that minimize the impact on quality while monotonically reducing imbalance. In addition, we use a secondary heuristic to handle cases where the greedy approach can not make progress. It moves a small set of vertices in a way that reduces block-internal imbalances between dimensions, thereby unlocking new move options.

Contributions. We propose a new rebalancing algorithm for multi-constraint partitioning and integrate it into the state-of-the-art parallel hypergraph partitioner Mt-KaHyPar [19], while also implementing multi-constraint support for all components of Mt-KaHyPar. We prove that our rebalancing always restores balance for $d = 2$ and bounded maximum weights. Experiments on 77 hypergraph instances and 67 graph instances show that our algorithm still finds a feasible solution with over 99 % reliability for $d > 2$ and larger weights, whereas state-of-the-art competitors are below 90 %. Our algorithm computes the best solution for 79 % of instances, achieves an 11.5 % geometric mean quality improvement compared to the next best competitor, and it is the fastest multilevel algorithm when using 16 threads.

2 Preliminaries

Hypergraphs and Multi-Constraint Partitioning. For a given dimensionality $d \in \mathbb{N}$, let $H = (V, E, c, \omega)$ be a *weighted hypergraph* with a vertex set V, a set of hyperedges $E \subseteq \mathcal{P}(V)$, multi-dimensional vertex weights $c\colon V \mapsto [0,1]^d$, and edge weights $\omega\colon E \mapsto \mathbb{R}_{>0}$. We extend c and ω to sets, i.e., $c(V' \subseteq V) \coloneqq \sum_{v \in V'} c(v)$ and $\omega(E' \subseteq E) \coloneqq \sum_{e \in E'} \omega(e)$. The vertices of a hyperedge e are called the *pins* of e. A *k-way partition* $\Pi = \{V_1, \ldots, V_k\}$ of a hypergraph H is a set of k disjoint *blocks* that partition V, i.e., $V = V_1 \cup \cdots \cup V_k$. When considering a k-way partition Π, we always assume that the vertex weights are normalized such that $\frac{1}{k} c(V)_j = 1$ for each dimension $j \in [d]$, as this simplifies notation. We say that Π is *ε-balanced* if $c(V_i)_j \leq 1+\varepsilon$ for each $i \in [k]$ and $j \in [d]$. The *connectivity* of a hyperedge e is $\lambda(e) \coloneqq |\{V_i \in \Pi \mid V_i \cap e \neq \emptyset\}|$. *Multi-constrained balanced hypergraph partitioning* asks for an ε-balanced k-way partition of H that minimizes the *connectivity* $(\lambda - 1)(\Pi) \coloneqq \sum_{e \in E} (\lambda(e) - 1)\omega(e)$. We also refer to the connectivity as the solution quality of the partition. The problem of finding any balanced partition, ignoring quality, is equivalent to the vector scheduling problem with k identical machines. Vector scheduling admits an EPTAS for constant d [6] (with doubly-exponential dependence on d), but constant factor approximations for arbitrary d are hard [11].

Multilevel Algorithms. A common approach for (hyper-)graph partitioning in practice is the multilevel scheme. Multilevel algorithms construct a sequence of increasingly smaller hypergraphs by contracting clusters or matchings (*coarsening phase*), followed by computing an *initial partition* on the smallest hypergraph. Afterwards, the contractions are undone in reverse order while projecting the current partition to the next larger hypergraph (*uncoarsening phase*). At each step, the partition is further improved via local search algorithms (*refinement*). While approximating balanced partitioning within a constant factor is NP-hard in general [2], multilevel algorithms often find high-quality solutions in practice [10, 19].

3 Related Work

Multilevel Partitioning. We refer to recent surveys [8, 10] for a broad overview on hypergraph partitioning and multilevel partitioning in particular. Within the multilevel framework, strong refinement algorithms are crucial for improving the partition during uncoarsening and thereby obtaining a high-quality solution. The FM algorithm by Fiduccia and Mattheyses [16] is one of the most widely used refinement algorithms as it offers a good trade-off between quality and running time. It greedily moves vertices prioritized by their current gain, allowing moves with negative gains in order to escape local minima (afterwards rolling back to the best observed solution). However, recent work [18, 29, 34, 37] demonstrated that it is beneficial to also allow temporary balance violations. Refinement algorithms based on this principle include Jet [18], as well as unconstrained variants of FM and label propagation refinement [34].

Our implementation extends Mt-KaHyPar [19, 22] (default configuration) for multi-constraint partitioning. Mt-KaHyPar is a scalable shared-memory hypergraph partitioner with quality comparable to the best sequential algorithms [19], full support for unconstrained refinement [34] and configurations with different time to quality trade-offs [19, 20, 21].

Multi-Constraint Partitioning. In comparison to single-constraint partitioning, literature on the multi-constraint setting is sparse. Metis supports multi-constraint partitioning on graph inputs [28], with key components including an adjusted tie breaking during coarsening and a multi-queue approach for bipartitioning and refinement. Metis uses rebalancing, but forbids balance-violating moves once balance is achieved. These techniques were then transferred to ParMetis [38, 40], with new contributions focusing on a scalable two-pass refinement algorithm. The idea is to compute move candidates and then select a subset that avoids (too large) balance violations. The authors observed that small imbalances can be beneficial for solution quality [38]. Barat et al. implement multi-constraint support for Scotch via "small variations" of the baseline algorithm [7]. However, the implementation is not public.

Some more recent graph partitioning algorithms operate outside of the multilevel framework. This includes GD [4], a multi-constraint partitioner based on projected gradient descent by Avdiukhin et al., and PuLP [42], a single-level graph partitioner designed for scalability. PuLP does not implement general multi-constraint partitioning, but it supports the special case of balancing both node weights and edge weights simultaneously.

For hypergraphs, Karypis [27] applies ideas from multi-constraint Metis to the hMetis hypergraph partitioner. Selvakkumaran et al. [41] consider resource-aware FPGA placement, demonstrating good performance for approaches that apply postprocessing (i.e., rebalancing) to a hMetis partition. Yet, both results are not integrated into the public hMetis package. Zoltan [13] supports geometric mesh partitioning with multi-dimensional weights (RCB algorithm), but not general multi-constraint partitioning. To the best of our knowledge, there are only two public implementations of general multi-constraint hypergraph partition-

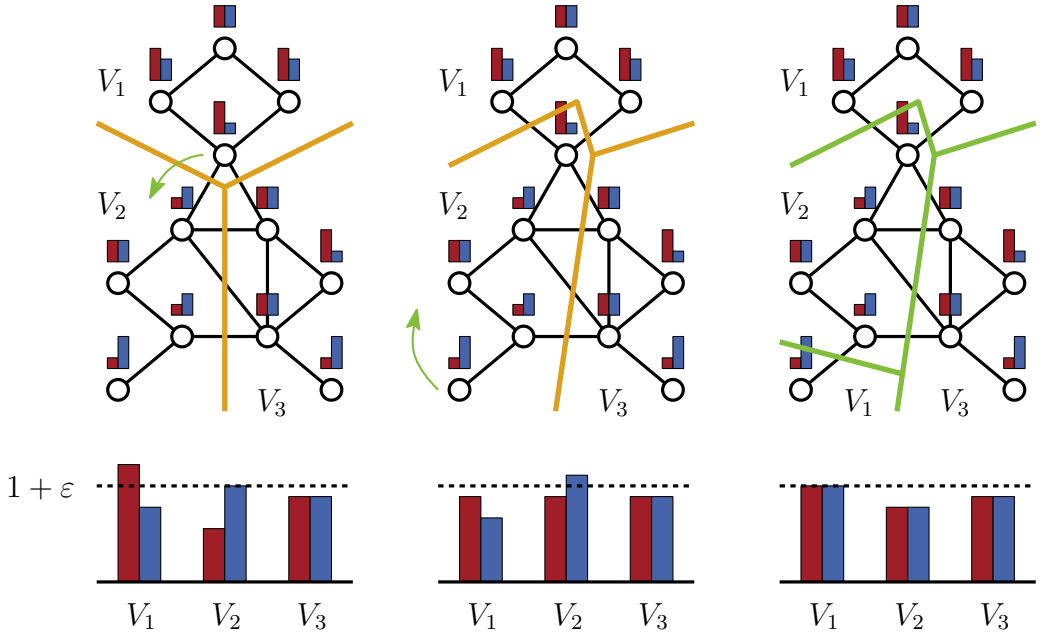


Figure 1 Illustration of rebalancing via the L_1^u imbalance. Any move away from V_1 also overloads the target block (left). However, there is a sequence of two moves (applied from left to right) that monotonically decreases the L_1^u imbalance (with $u = 1 + \varepsilon$) and results in a balanced partition.

ing. PaToH [5, 49] implements multi-constraint partitioning by generalizing the standard algorithmic components, but does not use any specialized algorithms. TritonPart [9] is designed for VLSI applications, including multi-dimensional weights. However, the publication only evaluates single-constraint inputs, so its actual capabilities are unclear.

4 Rebalancing for Multi-Constraint Partitioning

The primary design goals for our rebalancing algorithm are that (i) it should restore balance with high reliability and (ii) it should minimize the impact on partition connectivity (so it finds connectivity improvements when combined with unconstrained refinement). For the second goal, it is generally preferable to move only a small number of vertices. The remainder of this section therefore develops new techniques to restore balance by iteratively moving single vertices, which becomes more difficult with increasing dimension d.

4.1 A More General Imbalance Metric

For the following discussion, remember that we assume all vertex weights are normalized such that the average block weight is 1 in each dimension (Section 2). We further use $\delta := \max_{v \in V, j \in [d]} c(v)_j$ to denote the maximum vertex weight. For balanced partitioning, a common assumption is that weights are small ($\delta \leq \varepsilon$) – otherwise, finding any feasible solution is already NP-hard [23, 25]. In the single-constraint case, this makes rebalancing a simple task; a δ-balanced assignment can be found by iteratively moving vertices from overloaded blocks to a block with load at most 1. However, this approach is already insufficient for $d = 2$. There might be no block which simultaneously has small weight in both dimensions, and thus every possible move away from an overloaded block would overload another block (e.g., Figure 1). Instead, we propose to allow moves to overloaded blocks as long as they reduce the overall imbalance. We formalize this with an imbalance metric based on the L_1 norm.

▶ **Definition 1.** *For a given partition* $\Pi = \{V_1, \ldots, V_k\}$ *of a hypergraph* H *and an upper weight limit* $u \in \mathbb{R}$*, we define the* L_1^u *imbalance of block* V_i *as* $L_1^u(V_i) := \sum_{j \in [d]} \max\{c(V_i)_j - u, 0\}$ *and the total* L_1^u *imbalance as* $L_1^u(\Pi) := \sum_i L_1^u(V_i)$*.*

We can observe that all blocks have maximum weight at most u in all dimensions if and only if $L_1^u(\Pi) = 0$. As illustrated in Figure 1, the high-level idea of our rebalancing algorithm is to iteratively apply moves that reduce the L_1^u imbalance, for an appropriately chosen value of u. In the 2-dimensional case, we can show that such a move always exists under certain conditions.

▶ **Theorem 2.** *Let* $d = 2$ *and let* Π *be a partition of a hypergraph* H*. If* $u \geq 1 + \delta$ *and there is a block* $V_1 \in \Pi$ *with* $\|c(V_1)\|_\infty > u + \delta$*, then it is possible to move a vertex from* V_1 *to another block such that the* L_1^u *imbalance is reduced.*

Proof. Without loss of generality, we assume $c(V_1)_1 > u + \delta$. Let V_2 be any block with $c(V_2)_1 \leq 1$. First, we consider the case where V_1 contains a vertex v with $c(v)_1 > c(v)_2$. Moving v to V_2 does not overload the first dimension (due to $u \geq 1+\delta$) and therefore increases the L_1^u imbalance by at most $c(v)_2$. Combined with the reduced imbalance of V_1, the total change in imbalance is at most $c(v)_2 - c(v)_1 < 0$. In the case where $c(v)_2 \geq c(v)_1$ holds for all vertices $v \in V_1$, we get $c(V_1)_2 \geq c(V_1)_1 > u + \delta$ by summation. Let v be a vertex assigned to V_1 with $c(v)_1 > 0$. Since removing v from V_1 reduces the imbalance in both dimensions, the change in imbalance when moving v to V_2 is at most $c(v)_2 - (c(v)_1 + c(v)_2) = -c(v)_1 < 0$. ◀

Using Theorem 2, we can construct a rebalancing algorithm that guarantees 2δ-balance. Note that this is only a factor two worse than the one-dimensional bound of δ.

▶ **Corollary 3.** *If* $d = 2$*, then the following algorithm always results in a* 2δ*-balanced partition: While the partition is not* 2δ*-balanced, apply any move that reduces the* $L_1^{1+\delta}$ *imbalance.*

Proof. The partition is not 2δ-balanced if there is a block V_1 with $\|c(V_1)\|_\infty > 1 + 2\delta$, which implies with Theorem 2 that a move decreasing the $L_1^{1+\delta}$ imbalance always exists. The algorithm terminates since the sequence of imbalance values is strictly decreasing and the number of possible configurations is finite. ◀

Unfortunately, no similar guarantee holds for $d > 2$; see our full paper [33] for a detailed discussion. However, we will show in Section 6 that rebalancing with the L_1^u imbalance still works well in practice.

4.2 Greedy L_1^u Rebalancing

For our full algorithm, we build on the greedy rebalancing scheme that has proven successful for single-constraint partitioning [34]. Initially, greedy rebalancing selects a set of move candidates $M \subseteq V$ by collecting all vertices in overloaded blocks. For each vertex $v \in M$, a set of eligible target blocks $T(v) \subseteq \Pi$ is determined (in the single-constraint case, blocks that do not become overloaded when adding v) and the best target block is selected from $T(v)$ based on a rating function $r(v, \cdot)$. These steps are straightforward to do in parallel. Then, the algorithm iteratively moves the vertices in M, greedily selecting the vertex with the best rating in each step. In practice, this is done with a concurrent relaxed priority queue [47], extracting the current best moves in parallel while updating the priorities of neighbors after a move is applied. The rebalancing terminates when either balance is restored or all remaining moves in M do not improve balance.

For multi-constraint rebalancing, we consider all moves that reduce the total L_1^u imbalance – even if the target block is overloaded. To define our rating function, we first need to introduce the *gain* of a move. Assuming we move a vertex v to the block V_i, let Π' be the partition that results from applying this move. We define the *connectivity gain* as $\Delta(\lambda-1)(v,V_i) \coloneqq (\lambda-1)(\Pi) - (\lambda-1)(\Pi')$ and the *balance gain* analogously as $\Delta L_1^u(v,V_i) \coloneqq L_1^u(\Pi) - L_1^u(\Pi')$. Our rating then computes the ratio of connectivity gain to balance gain (high is better):

$$r(v,V_i) \coloneqq \begin{cases} \frac{\Delta(\lambda-1)(v,V_i)}{\Delta L_1^u(v,V_i)}, & \text{if } \Delta(\lambda-1)(v,V_i) < 0 \\ \Delta(\lambda-1)(v,V_i) \cdot \Delta L_1^u(v,V_i), & \text{if } \Delta(\lambda-1)(v,V_i) \geq 0 \end{cases}$$

This is effectively equivalent to the single-constraint case [34], optimizing the rebalancing progress relative to the quality penalty (note that we expect a negative connectivity gain). However, allowing moves that overload the target block also has drawbacks. As vertices from newly overloaded blocks are not contained in M, the algorithm can not restore balance for these blocks. Instead, we run up to ten repeated rounds of greedy rebalancing, stopping early if the balance is restored or a round is unable to improve the imbalance.[1]

An important parameter is the choice of u. A simple option is $u = 1+\varepsilon$, but Theorem 2 tells us to prefer $u = 1+\varepsilon-\delta$, where δ is the maximum vertex weight. However, real-world instances might contain some vertices with large weight relative to the average block weight. Choosing u too small because of this can cause problems, in particular since Theorem 2 also requires that $u \geq 1+\delta$ (and therefore $2\delta \leq \varepsilon$). Fortunately, it is often practically feasible to ignore a number of overly heavy vertices and effectively work with a smaller δ'. We use a parameterized threshold t that limits the considered weight (defined as a fraction of the average block weight), define $\delta'_j \coloneqq \min\{t, \max_v c(v)_j\}$ for $j \in [d]$ and choose per-dimension bounds $u = (1+\varepsilon-\delta'_1, \ldots, 1+\varepsilon-\delta'_d)^\top$; extending Definition 1 in the natural way to $u \in \mathbb{R}^d$. Concretely, we found that $t = 0.0025$ works well, see Section 6.2. During multilevel partitioning, rebalancing is applied at different levels of the coarsening hierarchy (Section 2). We compute δ' separately for each level, since the maximum vertex weight might be different.

Moreover, race conditions can lead to the parallel execution of moves that worsen the imbalance in combination (this is a non-issue for $d = 1$ as overloading the target block can be avoided with atomic operations). To prevent this, we add a rollback that restores the intermediate state with lowest imbalance after each round, based on a sequential move order. This is equivalent to the rollback of FM refinement, but using the L_1^u imbalance as objective.

4.3 A Fallback to Fix Internal Imbalance

If $d > 2$ or if heavy vertices are present, the L_1^u imbalance still exhibits imbalanced local minima: states where some blocks are overloaded, yet no single move exists that improves the L_1^u imbalance. This is usually due to *internal imbalance* of blocks, i.e., one dimension has larger weight than the remaining dimensions. With high internal imbalance, a block might be overloaded in one dimension although its weight is below average if summed over all dimensions. We address this with a fallback heuristic designed to reduce internal imbalance, which is triggered if a rebalancing round finds no balance-improving move. The heuristic selects a small subset of vertices S_i from each overloaded block V_i, which is moved to other blocks even if this worsens the current L_1^u imbalance. For this, the algorithm first chooses for each vertex $v \in V_i$ the target block $t(v)$ where moving v to $t(v)$ worsens the L_1^u imbalance the least. Then, we prioritize the vertices based on the following rating (higher is better):

[1] Including all vertices in M is not a solution either, since changes to the partition balance require a (very expensive) global update of the target blocks – which is implicitly handled when using multiple rounds.

$$s(v) := \frac{1}{\|c(v)\|_1} \cdot \frac{c(v)_\ell}{\sum_{j \neq \ell} c(v)_j} \cdot \left(1 + \frac{\Delta L_1(v, t(v))}{\|c(v)\|_1}\right),$$

where $\ell = \arg\max_{j \in [d]} c(V_i)_j$ is the dimension where the current block has maximum weight. The first term penalizes heavy vertices, as it is usually sufficient to move a relatively small amount of weight. The second term selects vertices with a high weight in the overloaded dimension, so that the move reduces the internal imbalance of V_i substantially. Finally, the third term attempts to minimize the increase in L_1^u imbalance (note that $\Delta L_1(v, t(v))$ is negative and the weight sum $\|c(v)\|_1$ is an upper bound for its absolute value). We also consider alternative ratings in the full paper [33], concluding that the exact choice of the rating function has only small influence on the overall performance of the fallback.

The vertex with the best rating is iteratively added to S_i until the accumulated weight is large enough that removing S_i from V_i makes the block balanced. Then, all selected vertices are moved to their target blocks and we apply L_1^u rebalancing again. If the partition is still imbalanced, the rebalancing stops, as it seems unlikely that balance can be restored. Note, it is possible that this results in a state with worse imbalance than before the fallback. We use the same rollback technique as discussed before to restore the state with lowest imbalance.

5 Multilevel Multi-Constraint Partitioning

In the following, we summarize how we integrate multi-constraint support into Mt-KaHyPar. Due to space limitations, we refer the interested reader to previous publications [19, 22, 34] for additional context on the algorithmic details of Mt-KaHyPar. From a high-level view, we make only minimal semantic changes to the algorithmic components other than the rebalancing; instead we change the data structures and low-level operations that handle vertex weights. While our description assumes normalized weights, the actual implementation normalizes the weights on the fly whenever comparisons between dimensions are necessary.

Coarsening and Initial Partitioning. Mt-KaHyPar uses size-constrained label propagation clustering, where vertices are visited in random order and greedily assigned to a neighboring cluster that minimizes the heavy-edge rating function [19]. A cluster weight limit of $c_{\max} = \frac{c(V)}{160k}$ is enforced to prevent skewed weight distributions in the coarse hypergraph. We simply enforce this limit separately per constraint, i.e., if a vertex u wants to join a cluster $C \subseteq V$, we require that $c(C)_i + c(u)_i \leq \frac{1}{160}$ for each $i \in [d]$. Similarly, when computing an initial solution via recursive bipartitioning of the coarsest hypergraph, we use the d-dimensional balance criterion but leave the portfolio of bipartitioning algorithms otherwise unchanged. In the multi-constraint setting it can be difficult (or impossible) to find balanced bipartitions [28]; in such cases we rely on our rebalancing algorithm to restore balance during uncoarsening.

Uncoarsening and Refinement. While the engineering of refinement algorithms is a topic with many facets, we want to highlight some aspects that are of particular relevance to the multi-constraint setting. First, refinement algorithms are typically designed with the assumption that the input partition is already balanced. Consequently, finding a balanced solution early leads to better results in general, as it provides more refinement opportunities. We therefore apply rebalancing after initial partitioning and after each uncontraction step, if the partition is not yet balanced. Note that uncoarsening results in a finer hypergraph with smaller weights, possibly allowing the rebalancing to succeed even if this was impossible before. Second, refinement algorithms based on iterative vertex moves have two options for escaping local minima: (i) allow moves that worsen the connectivity, as is done in FM

refinement, and (ii) allow moves that cause a temporary violation of the balance constraint, applying rebalancing afterwards. In the multi-constraint setting, moves cause a balance violation more easily, which makes it particularly useful to include option (ii).

Therefore, we adapt the unconstrained refinement algorithms of Mt-KaHyPar [34] to the multi-constraint setting: unconstrained label propagation and unconstrained FM refinement. The former essentially alternates between rounds of label propagation and rebalancing, and is easily adapted by replacing the previous rebalancing with our new multi-constraint rebalancing. However, unconstrained FM refinement incorporates two techniques that are difficult to generalize. First, it estimates penalties for balance-violating moves via a lookup in a weight sequence. Second, it interleaves sequences of balance-violating moves with balance-restoring moves in a reordering step, possibly discovering additional feasible solutions at intermediate points of the sequence. Both techniques rely on the monotone behavior of a sequence of scalar weights, which has no direct multi-dimensional equivalent. Consequently, our current implementation of unconstrained FM does not support these techniques.

Postprocessing. Mt-KaHyPar initially removes degree zero vertices and reinserts them after the partitioning in a postprocessing step. In the multi-constraint setting, we need to be careful to not violate the partition balance. We sort all removed vertices in decreasing order of their L_1 weight. We then iterate over the vertices and assign each vertex v to the block V_i that maximizes $c(v)^\top \cdot (\mathbb{1}_d + \varepsilon_d - c(V_i))$. This ensures that the vertex weight is aligned with the free capacity of V_i and performed well in preliminary experiments. If necessary, we apply rebalancing afterwards.

Scalability and Efficient Implementation

In principle, the discussed adaptations have no impact on the scalability of Mt-KaHyPar (excluding the new rebalancing), as we mostly replace one-dimensional operations on weights with d-dimensional operations. If d is a small constant, this does not affect the asymptotic behavior of the algorithm. However, we need an efficient data layout to minimize overhead in practice. Mt-KaHyPar stores vertex weights in an array of size n as part of the basic (hyper-)graph data structure. For d constraints, we instead use an array of size $n \cdot d$ with d consecutive entries for each vertex. Since operations typically need to access all weight dimensions of a vertex simultaneously, this is a cache-efficient layout.

Many procedures need to perform calculations on vertex weights, and sometimes store the results. A naive approach might create a new allocation for each intermediate value. As this would cause significant overhead, we instead combine multiple techniques for a more efficient implementation. First, we can use pointers for any weights that are already stored elsewhere. To store results, however, creating a copy is necessary, and storing partial results can prevent redundant recomputations. In these cases, we use a pre-allocated copy buffer. Thereby, only a single initial allocation is required (usually one allocation per thread), which is then reused throughout the procedure. Furthermore, calculations often involve multiple steps with intermediate results. For example, to calculate whether moving a vertex would overload the target block, we need the sum of the current block weight and the vertex weight. In such cases, we do not store intermediate results at all but instead implement the calculation in a single loop, which both saves copies and improves data locality. This optimization is called *loop fusion* and it is a well-known technique to improve the efficiency of multiple consecutive matrix or vector operations [36, 48], making it a natural optimization for our use case.[2]

[2] While commonly used compilers such as `gcc` do not apply loop fusion automatically, we still can avoid doing it manually. We implement basic operations only once with a generic programming technique known as *expression templates* in the C++ community [43, 44], which allows to express calculations naturally while emitting efficient assembly instructions as a single fused loop.

Atomic Operations. Maintaining partition weight limits in a parallel setting involves atomic operations to synchronize changes from different threads. Mt-KaHyPar uses `add-and-fetch` operations, optimistically adding the vertex weight to the block weight while using the return value to cancel the move if the weight limit is violated due to a race condition (restoring the weight via `sub-and-fetch`). We use one `add-and-fetch` per dimension and, in case of constrained refinement, cancel the move if this exceeds the weight limit in the current dimension. This guarantees that the weight limits are not exceeded, although moves might be canceled unnecessarily due to race conditions. We do not attempt to prevent this as it should be rare in practice.

6 Experimental Evaluation

We implemented our approach within the Mt-KaHyPar framework (v1.5.3), the source code is available at `https://github.com/kahypar/mt-kahypar/tree/esa2026`. In the following, we evaluate the effectiveness of our rebalancing algorithm and unconstrained refinement (Section 6.2) and we compare our approach with the state of the art (Section 6.3).

6.1 Setup

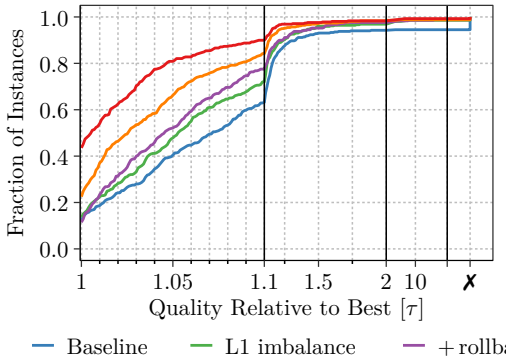
The code is compiled using gcc 14.2 with flags `-O3 -march=native`. We perform all experiments on a machine with an AMD EPYC 9684X processor (one socket with 96 cores), clocked at 2.55-3.7 GHz with 1536 GB RAM and 1152 MB L3 cache. We run all parallel algorithms with 96 threads, except where specified otherwise.

Benchmark Sets. We created our benchmarks [32] by adopting sets from the literature and extending the (hyper-)graphs with additional weight dimensions. Since a common application is the simultaneous balancing of vertices and edges [42], we always use unit weights for the first dimension and the vertex degree as the weight for the second dimension. Set V consists of 27 hypergraphs derived from VLSI circuit design; namely from the ISPD98 VLSI Circuit Benchmark Suite [1] and the DAC 2012 Routability-Driven Placement Contest [45] (the latter instances are significantly larger). These hypergraphs have natural weights that correspond to the area of electrical components, which we include as a third weight dimension. Set H comprises 50 hypergraphs from the benchmark set of Heuer and Schlag [26] (originally 488 instances). We selected instances with a high variance in vertex degrees, as these result in a more challenging weight distribution. Since most competitors do not support hypergraph inputs, we also include Set I and Set R from Maas et al. [34, 35], consisting of large graph instances. Set I contains graphs with high degree variance, while Set R contains graphs with more regular degrees. We exclude the four largest instances from Set I (`twitter2010`, `friendster`, `sk2005`, `it2004`) since they cause overflows when using 32 bit integers to represent weights. See Table 1 for a summary of key metrics.

Methodology. In the following evaluation, we allow an imbalance of $\varepsilon = 0.03$. We use $k \in \{2, 5, 8, 11, 16, 27, 32\}$ and 5 random seeds for each instance (i.e., combination of hypergraph and k), aggregating connectivity and running time with the arithmetic mean over the seeds. We consider the result imbalanced only if none of the seeds yielded a balanced solution. When aggregating over multiple instances, we use the geometric mean. Reported running times exclude file I/O and parsing.

Table 1 Overview of benchmark sets. We show pin counts for the smallest and largest hypergraph in the set (graph edges count as two pins), the number of instances excluded due to heavy vertices (determined separately for each value of k), and the instances where Corollary 3 guarantees balance – which is the case if both $d = 2$ and $2\delta \leq \varepsilon$ hold.

Set	Type	#	d	Min # Pins	Max # Pins	Excluded	Corollary 3 applies
V	hypergraph	27	3	50.6 k	4,9 M	81 (43 %)	–
H	hypergraph	50	2	3.5 k	55.8 M	40 (11 %)	101 (28.9 %)
I	graph	34	2	10.7 M	1.7 B	6 (3 %)	147 (61.8 %)
R	graph	33	2	25.4 M	1.1 B	–	231 (100 %)



Algorithm	Conn.	Time	Balanced
Baseline	1.0	1.0	94.50 %
L_1^u imbalance	0.996	1.118	98.56 %
+ rollback	0.993	1.145	98.80 %
+ weight red.	0.964	1.099	98.80 %
+ fallback	0.953	1.129	99.28 %

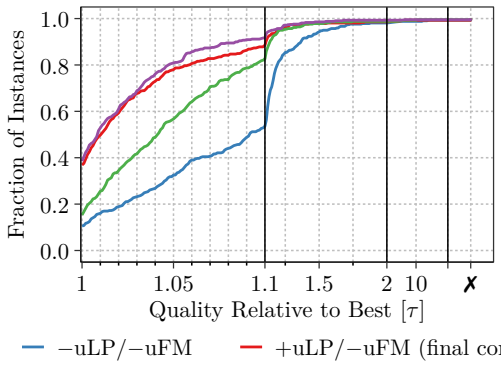
Figure 2 Ablation study of the components of our rebalancing algorithm on Set V and Set H. We compare the resulting connectivity (left) and list geometric means of the connectivity and total running time relative to the baseline, as well as the fraction of balanced results (right).

We use performance profiles [14] to compare the solution quality of different algorithms. A performance profile plots for each algorithm A the fraction of instances where A is within a factor τ of the best found solution, i.e., $\frac{1}{|\mathcal{I}|}|\{q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I) \mid I \in \mathcal{I}\}|$, where $\mathcal{I}$ is the set of instances, $\mathcal{A}$ is the set of considered algorithms and q_A the solution quality (connectivity) of algorithm A on a given instance. We mark infeasible results with ✗.

Depending on the distribution of vertex weights, instances might not admit a balanced solution. E.g., an instance is trivially infeasible if a singular vertex exceeds the maximum allowed block weight – which becomes more likely for larger values of k. To avoid this, we exclude instances with any vertex weight above 70% of the average block weight. Note that this is much more relaxed than Corollary 3 (which requires $\delta \leq \frac{1}{2}\varepsilon = 1.5\,\%$), leading to the inclusion of many instances where we can not provide a theoretical guarantee (see Table 1).

6.2 Impact of Rebalancing and Refinement

We evaluate the effectiveness of our rebalancing algorithm in an ablation study, presented in Figure 2. We incrementally add the components proposed in Section 4, while all other parts of the algorithm use our final configuration. The baseline is a multi-constraint version of the standard rebalancing algorithm of Mt-KaHyPar, which allows moves only if the target block does not become overloaded. Using the L_1^u imbalance metric (with $u = 1 + \varepsilon$) reduces the fraction of instances with no balanced solution by a factor of four (1.4 % instead of 5.5 %), demonstrating its effectiveness for multi-constraint partitioning. However, we pay a 12 % geometric mean running time overhead since multiple rebalancing rounds can now be necessary. Adding a rollback mechanism further increases the reliability of finding



Algorithm	Conn.	Time	Balanced
−uLP/−uFM	1.000	1.000	99.28 %
+uLP/−uFM	0.910	1.179	99.28 %
−uLP/+uFM	0.933	1.115	99.52 %
+uLP/+uFM	0.902	1.280	99.52 %

Figure 3 Comparing refinement algorithms on Set V and Set H. We use either constrained or unconstrained label propagation (uLP) and FM refinement (uFM), resulting in 4 configurations. We compare the resulting connectivity (left) and list geometric means of the connectivity and total running time relative to the constrained baseline, as well as the fraction of balanced results (right).

balanced solutions and also provides a minor improvement in solution quality. As suggested by Theorem 2, the imbalance metric works even better with a slightly reduced target block weight u. Setting a threshold of $t = 0.0025$ (see Section 4.2), this provides a substantial improvement both in solution quality (3 %) and running time (4.5 %). We found in preliminary experiments that the exact choice of t is mostly inconsequential, as long as t is significantly smaller than ε. Finally, our fallback algorithm further increases the reliability to over 99 % and also improves solution quality (1 %) at a moderate cost in running time (3 %).

In Figure 3, we compare the performance of constrained and unconstrained refinement (see Section 5), specifically label propagation (LP) and FM as used by default Mt-KaHyPar. We also include hybrid constrained/unconstrained configurations. The result clearly demonstrates the importance of unconstrained refinement for high-quality partitioning. All configurations with unconstrained refinement achieve substantially lower connectivity values compared to −uLP/−uFM (7-10 % in the geometric mean), though at the cost of increased running time. Comparing the unconstrained algorithms, LP performs better than FM, and is already close to the best quality when combined with standard FM – unconstrained FM adds only a marginal improvement at a notable running time cost. This is likely because two techniques are missing compared to the single-constrained case: imbalance penalties and move sequence reordering (see Section 5). For our final configuration, we use the hybrid variant with unconstrained LP refinement.

6.3 Comparison to the State of the Art

We compare our algorithm (denoted as Mt-KaHyPar-MC), to Metis [28] (v5.2.1), ParMetis [38] (v4.0.3), PaToH [49] (v3.2) and PuLP [42] (v0.2), which is a mostly comprehensive list of publicly available partitioners with (partial) multi-constraint support. For Scotch [7], hMetis [27] and GD [4], we are not aware of a public implementation. We exclude Zoltan [13], as it is limited to geometric partitioning, and TritonPart [9], as the paper provides no data for the claimed multi-constraint support.[3] We evaluated both the direct k-way and recursive bisection mode for Metis, and the default and quality configuration for PaToH. We only show direct k-way Metis and default PaToH, as these outperform the alternative configurations [33]. We use the recommended default settings for all remaining partitioners.

[3] The implementation also has non-trivial dependencies and generally a complicated setup process.

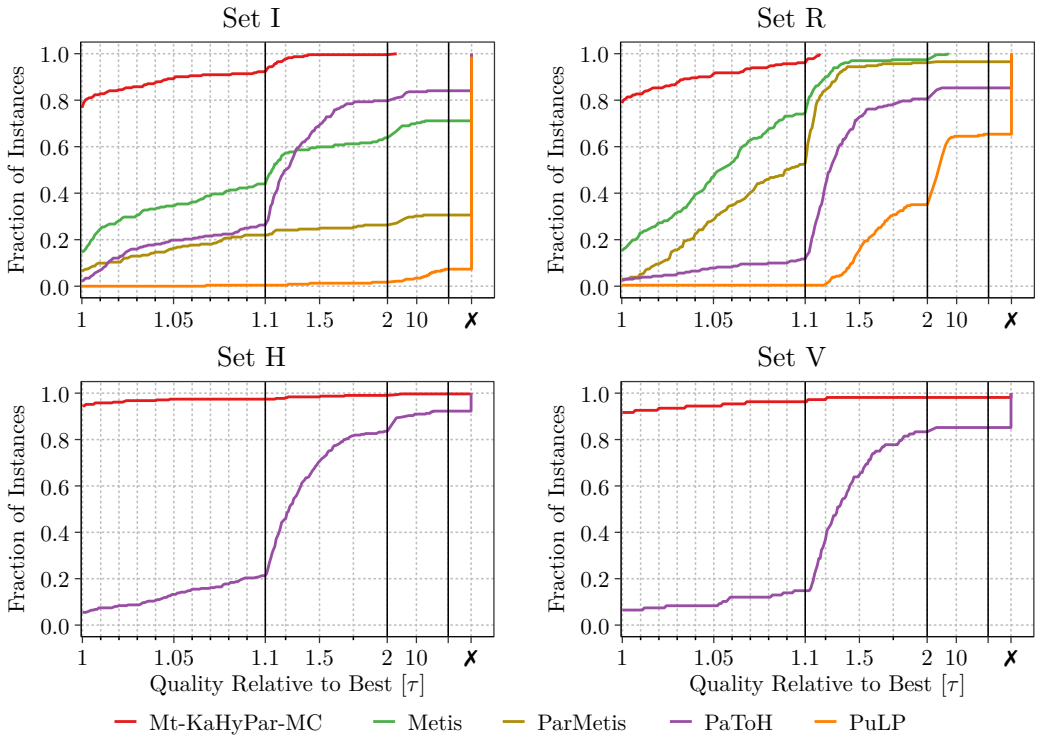


Figure 4 Comparing our solution quality to state-of-the-art competitors on graph benchmark sets (top) and hypergraph benchmark sets (bottom). Infeasible results or crashes are marked with ✗.

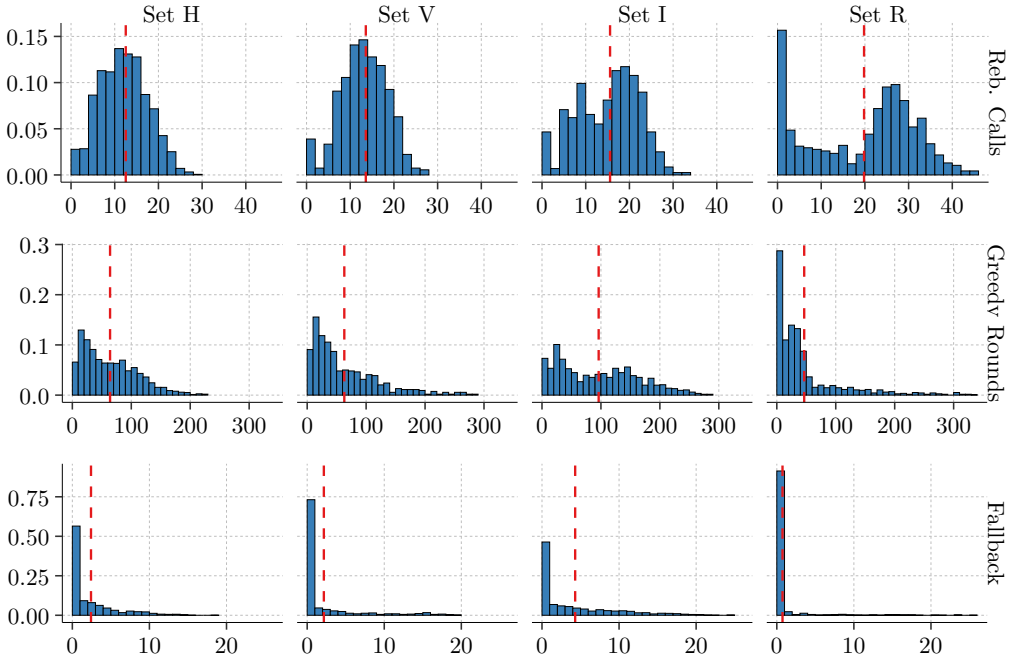


Figure 5 Histograms of rebalancing calls (top), greedy rebalancing rounds (middle) and fallback invocations during rebalancing (bottom). The y-axis depicts the fraction of runs (each seed constitutes a separate data point) and the x-axis the total count per run. The mean is marked with a red line.

Partition Balance. Figure 4 (top) shows the solution quality and fraction of feasible solutions on Set I and Set R. Mt-KaHyPar-MC always computes a feasible solution, while the competitors have lower reliability, in particular on Set I (see Table 2 for exact values). Specifically, PuLP finds a feasible solution for less than 9 % of inputs on Set I, likely because it is designed for more relaxed edge balance; the original paper uses 0.5 [42]. Note that the infeasible results include a few crashes in the case of PuLP, PaToH and ParMetis.[4]

We provide statistics about the behavior of our rebalancing in Figure 5. On average, rebalancing the partition once requires between 2.5 (Set R) and 6 (Set I) greedy rounds. On Set R, rebalancing is used more often (on average 20× per run), but it tends to succeed with few rounds and without our secondary fallback heuristic. In general, on more than half of the inputs the fallback is never used, though there is a tail of instances with different behavior.

Partition Connectivity. Mt-KaHyPar-MC finds the best solution of all considered algorithms on 79 % of instances for both Set I and Set R. The difference to the next competitor is larger on Set I (geometric mean: 1.16× versus Metis, 1.27× versus PaToH) than on Set R (1.09× versus Metis). This is consistent with previous results for unconstrained refinement on these instances [34]. Metis tends to find the best results among the competitors, in particular on Set R. On Set I, PaToH finds more balanced solutions than Metis though often with worse connectivity. We also compare our algorithm to PaToH (the only competitor that supports hypergraph inputs) on Set H and Set V in Figure 4 (bottom), with similar findings: Mt-KaHyPar-MC achieves significantly better solution quality and more often finds a balanced solution, even though most instances are outside the guarantees from Corollary 3. These results are restricted to $d \leq 3$; we include experiments with $d = 6$ in the full paper [33].

Table 2 Geometric mean running times on Set I and Set R, excluding instances where any algorithm crashed (40 out of 239 on Set I, none on Set R), and fraction of balanced results (excluding crashes). We also show the running time of our algorithm with 1 and 16 instead of 96 threads.

Algorithm	Time (Set I)	Balanced (Set I)	Time (Set R)	Balanced (Set R)
Mt-KaHyPar-MC	3.50 s	100.00 %	1.47 s	100.00 %
with 16 threads	7.22 s		3.17 s	
with 1 thread	61.34 s		27.94 s	
Metis	15.60 s	71.12 %	7.21 s	100.00 %
ParMetis	13.32 s	34.98 %	4.40 s	96.54 %
PaToH	86.46 s	89.86 %	70.38 s	85.28 %
PuLP	0.69 s	8.77 %	0.50 s	65.37 %

Running Time. While running time is not the primary focus of this work, we aim to preserve the scalability of default Mt-KaHyPar. The geometric mean running times presented in Table 2 show that Mt-KaHyPar-MC (with all 96 threads) is the second fastest algorithm after PuLP. This is unsurprising, as PuLP uses single-level partitioning, trading solution quality for a lower running time. Compared to all multilevel competitors, i.e., ParMetis with 96 threads, Metis, and PaToH (both single-threaded), our algorithm is still faster when using 16 threads. However, we also note that using 96 threads only leads to a 18.4× speedup, whereas default Mt-KaHyPar achieves a speedup of 20.5× for 64 threads [19]. Figure 6

[4] PuLP produced a segmentation fault in 30 out of 35 runs on rmat-n16m24, PaToH had allocation errors on uk2005, webbase2001 and mawi2015, ParMetis crashed on uk2005, webbase2001, mawi2015 and sinaweibo and exceeded the time limit on rmat-n25m28.

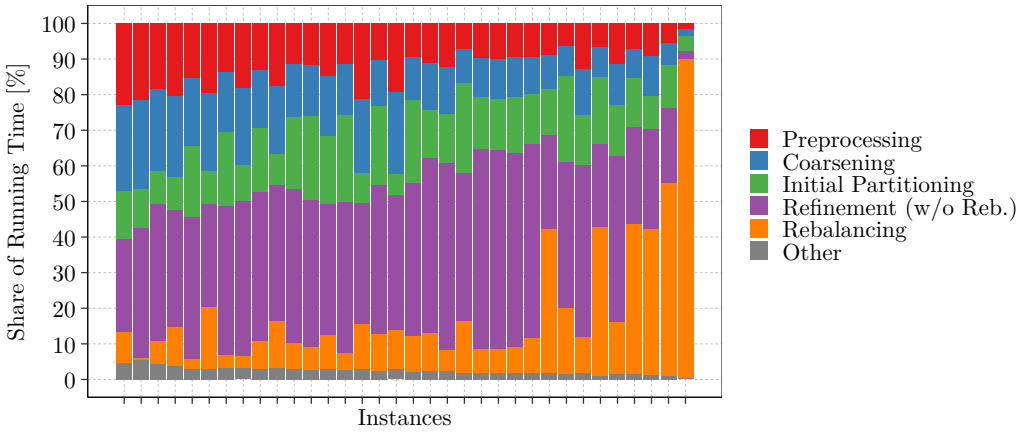


Figure 6 Running time of components, as fraction of total time. Each bar corresponds to a single instance (hypergraph and k) from the larger instances of Set V, in ascending order of total time.

provides insight into the running time of different parts of the algorithm. We can observe a high variance in rebalancing time: while it is usually below 20 % of total time, one instance is dominated by rebalancing. On this instance, our algorithm could not find a balanced solution, so the long time is likely due to many failed rebalancing attempts. This indicates opportunities for future work to reduce the overhead of failed rebalancing attempts and improve the scalability.

7 Conclusion

We achieve a substantial improvement in quality and reliability over the current state of the art in multi-constraint hypergraph partitioning, using unconstrained local search algorithms that escape local optima by allowing temporary balance violations. This is enabled by a new multi-constraint rebalancing algorithm that restores balance with high reliability, providing guarantees for $d = 2$ and bounded maximum weight. Yet, there is room for further improvements. A multi-constraint variant of the sequence reordering technique for unconstrained FM refinement should be possible, although it might have weaker guarantees. How to generalize penalties for balance-violating moves is less clear and might require a new kind of approximation. Furthermore, we plan to investigate the feasibility of strong balance guarantees for $d > 2$ and to minimize the running time overhead of rebalancing.

References

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

2 Konstantin Andreev and Harald Räcke. Balanced Graph Partitioning. In *16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 120–124, 2004. doi:10.1145/1007912.1007931.

3 Ankita Atrey, Gregory van Seghbroeck, Higinio Mora, Bruno Volckaert, and Filip De Turck. UnifyDR: A Generic Framework for Unifying Data and Replica Placement. *IEEE Access*, 8:216894–216910, 2020. doi:10.1109/ACCESS.2020.3041670.

4 Dmitrii Avdiukhin, Sergey Pupyrev, and Grigory Yaroslavtsev. Multi-Dimensional Balanced Graph Partitioning via Projected Gradient Descent. *Proceedings of the VLDB Endowment*, 12(8):906–919, 2019. doi:10.14778/3324301.3324307.

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

6 Nikhil Bansal, Tim Oosterwijk, Tjark Vredeveld, and Ruben van der Zwaan. Approximating Vector Scheduling: Almost Matching Upper and Lower Bounds. *Algorithmica*, 76(4):1077–1096, 2016. doi:10.1007/S00453-016-0116-0.

7 Rémi Barat, Cédric Chevalier, and François Pellegrini. Multi-criteria Graph Partitioning with Scotch. In *8th Workshop on Combinatorial Scientific Computing (CSC)*, pages 66–75, 2018. doi:10.1137/1.9781611975215.7.

8 Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering*, volume 9220, pages 117–158. Springer, 2016. doi:10.1007/978-3-319-49487-6_4.

9 Ismail Bustany, Grigor Gasparyan, Andrew B. Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. An Open-Source Constraints-Driven General Partitioning Multi-Tool for VLSI Physical Design. In *International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023. doi:10.1109/ICCAD57390.2023.10323975.

10 Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, et al. More Recent Advances in (Hyper)Graph Partitioning. *ACM Computing Surveys*, 55(12):253–253, 2023. doi:10.1145/3571808.

11 Chandra Chekuri and Sanjeev Khanna. On Multidimensional Packing Problems. *SIAM Journal on Computing*, 33(4):837–851, 2004. doi:10.1137/S0097539799356265.

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

13 Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Catalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006. doi:10.1109/IPDPS.2006.1639359.

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

15 Shantanu Dutt and Wenyong Deng. Cluster-Aware Iterative Improvement Techniques for Partitioning Large VLSI Circuits. *ACM Transactions on Design Automation of Electronic Systems*, 7(1):91–121, 2002. doi:10.1145/504914.504918.

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

17 Jonas Fietz, Mathias J. Krause, Christian Schulz, Peter Sanders, and Vincent Heuveline. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In *18th European Conference on Parallel Processing (Euro-Par)*, volume 7484 of *Lecture Notes in Computer Science*, pages 818–829, 2012. doi:10.1007/978-3-642-32820-6_81.

18 Michael S. Gilbert, Kamesh Madduri, Erik G. Boman, and Siva Rajamanickam. Jet: Multilevel Graph Partitioning on Graphics Processing Units. *SIAM Journal of Scientific Computing.*, 46(5):700, 2024. doi:10.1137/23M1559129.

19 Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable High-Quality Hypergraph Partitioning. *ACM Transactions on Algorithms*, 20(1):9:1–9:54, 2024. doi:10.1145/3626527.

20 Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel Flow-Based Hypergraph Partitioning. In *20th International Symposium on Experimental Algorithms (SEA)*, volume 233, pages 5:1–5:21, 2022. doi:10.4230/LIPIcs.SEA.2022.5.

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

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

23 Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep Multilevel Graph Partitioning. In *29th European Symposium on Algorithms (ESA)*, pages 48:1–48:17, 2021. doi:10.4230/LIPIcs.ESA.2021.48.

24 Wenzhong Guo, Genggeng Liu, Guolong Chen, and Shaojun Peng. A Hybrid Multi-Objective PSO Algorithm with Local Search Strategy for VLSI Partitioning. *Frontiers of Computer Science*, 8(2):203–216, 2014. doi:10.1007/S11704-014-3008-Y.

25 Tobias Heuer, Nikolai Maas, and Sebastian Schlag. Multilevel Hypergraph Partitioning with Vertex Weights Revisited. In *19th International Symposium on Experimental Algorithms (SEA)*, pages 8:1–8:20, 2021. doi:10.4230/LIPIcs.SEA.2021.8.

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

27 George Karypis. Multilevel Algorithms for Multi-Constraint Hypergraph Partitioning. Technical report, University of Minnesota, 1999.

28 George Karypis and Vipin Kumar. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *ACM/IEEE Conference on Supercomputing (SC)*, page 28, 1998. doi:10.1109/SC.1998.10018.

29 Robert Krause, Lars Gottesbüren, and Nikolai Maas. Deterministic Parallel High-Quality Hypergraph Partitioning. In *3rd Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 222–236. SIAM, 2025. doi:10.1137/1.9781611978759.17.

30 Alice Lasserre, Jean Marie Couteyen-Carpaye, Abdou Guermouche, and Raymond Namyst. Multi-Criteria Mesh Partitioning for an Explicit Temporal Adaptive Task-Distributed Finite-Volume Solver. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 976–985. IEEE, 2024. doi:10.1109/IPDPSW63119.2024.00168.

31 Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 443–458. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/ma.

32 Nikolai Maas. Benchmark Sets and Experimental Results for "High- Quality Multi-Constraint Hypergraph Partitioning via Greedy Rebalancing" , April 2026. doi:10.5281/zenodo.19630135.

33 Nikolai Maas. High-Quality Multi-Constraint Hypergraph Partitioning via Greedy Rebalancing, 2026. doi:10.48550/arXiv.2605.28333.

34 Nikolai Maas, Lars Gottesbüren, and Daniel Seemaier. Parallel Unconstrained Local Search for Partitioning Irregular Graphs. In *26st Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 32–45. SIAM, 2024. doi:10.1137/1.9781611977929.3.

35 Nikolai Maas, Lars Gottesbüren, and Daniel Seemaier. Benchmark Sets and Experimental Results for "Parallel Unconstrained Local Search for Partitioning Irregular Graphs" , May 2025. doi:10.5281/zenodo.15386627.

36 Kiminori Matsuzaki and Kento Emoto. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *21st Symposium on Implementation and Application of Functional Languages (IFL)*, volume 6041, pages 72–89, 2009. doi:10.1007/978-3-642-16478-1_5.

37 Peter Sanders and Daniel Seemaier. Brief Announcement: Distributed Unconstrained Local Search for Multilevel Graph Partitioning. In *36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 443–445, 2024. doi:10.1145/3626183.3660257.

38 Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning. In *6th European Conference on Parallel Processing (Euro-Par)*, volume 1900, pages 296–310, 2000. doi:10.1007/3-540-44520-X_39.

39 Kirk Schloegel, George Karypis, and Vipin Kumar. Graph Partitioning for Dynamic, Adaptive and Multi-phase Scientific Simulations. In *International Conference on Cluster Computing (CLUSTER)*, pages 271–273. IEEE Computer Society, 2001. doi:10.1109/CLUSTR.2001.959987.

40 Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel Static and Dynamic Multi-Constraint Graph Partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002. doi:10.1002/CPE.605.

41 Navaratnasothie Selvakkumaran, Abhishek Ranjan, Salil Raje, and George Karypis. Multi-Resource Aware Partitioning Algorithms for FPGAs with Heterogeneous Resources. In *41th Design Automation Conference (DAC)*, pages 741–746, 2004. doi:10.1145/996566.996768.

42 George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. PuLP: Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks. In *IEEE International Conference on Big Data*, pages 481–490. IEEE, 2014. doi:10.1109/BIGDATA.2014.7004265.

43 David Vandevoorde and Nicolai Josuttis. C++ Templates: The Complete Guide, 2002.

44 Todd Veldhuizen. Expression Templates, 1995.

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

46 Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. In *10th International Conference on Learning Representations (ICLR)*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=kSwqMH0zn1F.

47 Marvin Williams, Peter Sanders, and Roman Dementiev. Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues. In *29th European Symposium on Algorithms (ESA)*, volume 204, pages 81:1–81:17, 2021. doi:10.4230/LIPIcs.ESA.2021.81.

48 Field G. Van Zee and Robert A. van de Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, 2015. doi:10.1145/2764454.

49 Ümit V. Catalyürek and Cevdet Aykanat. Hypergraph-Partitioning-based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. doi:10.1109/71.780863.