# Engineering Learned Heuristics to Improve Clustering for Multilevel Graph Partitioning

**Simeon Schrape**
Karlsruhe Institute of Technology, Germany

**Nikolai Maas**
Karlsruhe Institute of Technology, Germany

**Kenneth Langedal**
Faculty of Mathematics and Computer Science, Heidelberg University, Germany

**Daniel Seemaier**
Karlsruhe Institute of Technology, Germany

## Abstract

Balanced Graph Partitioning is a classical optimization problem where quality guarantees are computationally infeasible, and practical solvers therefore rely on manually engineered heuristics. Yet, the problem has also proven difficult for approaches that rely heavily on machine learning – especially since applications often need to partition graphs of huge scale in a short amount of time. Instead, we demonstrate how to achieve practical improvements with a more careful approach that uses machine learning to improve heuristic decisions within the state-of-the-art solver Mt-KaHyPar.

We use a pre-trained neural network to predict a score for each edge, which then guides clustering decisions in the first phase of the partitioning (the coarsening). Combined with corresponding adjustments to the clustering algorithm and an efficient implementation of the neural network logic, we improve the overall solution quality while preserving the efficiency and scalability of the original algorithm. Our detailed evaluation on more than 180 graphs shows an average quality improvement of 2% on a class of graphs with beneficial properties, and unchanged quality on all remaining graphs. Moreover, our improvements generalize to a set of instances from the literature that are much larger than the graphs used during training.

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

**Keywords and phrases** Graph Partitioning, Graph Algorithms, Machine Learning, Neural Networks

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2026.25

**Supplementary Material** *Software*: `https://github.com/kahypar/mt-kahypar/tree/sea2026` [25] archived at `swh:1:dir:0285e232ceaf8b004e75d01d1e5f4e6984770663`
*Dataset*: `https://zenodo.org/records/19387774`

**Funding** *Kenneth Langedal*: Supported by DFG grant SCHU 2567/5-1.

## 1 Introduction

Graphs provide a widely used language for modeling relations between objects. In many applications, partitioning a graph into $k$ roughly equal-sized blocks while minimizing the number of edges cut by the partition is a crucial subtask [4, 9]. This Balanced Graph Partitioning problem is NP-hard and even NP-hard to approximate. Consequently, practical solvers rely on heuristics to partition large graphs quickly.

Modern general-purpose graph partitioners achieve high solution quality through *multilevel graph partitioning* (MGP). MGP proceeds in three phases: coarsening, initial partitioning, and refinement. During coarsening, the input graph is transformed into a hierarchy of successively smaller graphs by contracting edges, typically using matchings or clusterings. An initial partitioning algorithm then computes a $k$-way partition on the coarsest graph.


© Simeon Schrape, Nikolai Maas, Kenneth Langedal, and Daniel Seemaier;
licensed under Creative Commons License CC-BY 4.0

Finally, during uncoarsening, the partition is projected back to finer levels and refined using local search algorithms. The multilevel approach is effective because refinement can act at different granularities. At coarse levels, moving a single vertex corresponds to moving an entire cluster of vertices in the input graph, which helps escape from poor local minima.

To this end, the quality of the coarsening hierarchy is crucial, as it determines which structures are preserved across levels. Poor contractions can irrevocably conceal *good* cuts, thereby misguiding both initial partitioning and subsequent refinement. A common intuition is to contract heavy edges preferentially, since the cut of any initial partition is upper-bounded by the total edge weight of the coarsest graph. However, minimizing this total weight alone does not capture whether contractions preserve the *right* structures across levels. At the same time, coarsening offers no clear optimization objective that reliably measures the quality of a given hierarchy. As a result, although there was substantial progress on refinement methods over the previous decades (where the optimization objective is explicit), developing stronger coarsening algorithms remains challenging. Many modern partitioners still employ comparatively simple, locally driven coarsening heuristics.

We address this gap by using machine learning to guide coarsening decisions. We train a neural model that predicts a score for each edge, with higher scores indicating a higher likelihood of the edge being cut in the final partition. Unlike recent graph partitioning methods that adopt an end-to-end machine learning approach, our method is the first to combine the state-of-the-art partitioner Mt-KaHyPar with a pre-trained neural network for coarsening. Moreover, we carefully implemented our neural network logic and feature computation to minimize the inference overhead. As a result, we are the first to demonstrate that machine learning can boost the practical performance of state-of-the-art partitioners, evaluated on a large set of diverse benchmark instances.

**Contributions.** We combine the strengths of hand-crafted algorithms and machine learning techniques by integrating learned coarsening decisions into the state-of-the-art multilevel partitioner Mt-KaHyPar. To this end, we train, integrate and evaluate several machine learning models and study their behaviour in an end-to-end setting. Our experiments show that the resulting approach generalizes well to previously unseen graphs, improving solution quality by 0.3–1.4% over the Mt-KaHyPar baseline. With an efficient AVX2-based implementation of our best-performing model, we only incur a 14% running time overhead. These improvements are achieved in a mature setting: Mt-KaHyPar is the result of a long sequence of previous publications and engineering efforts [1, 20, 22, 26, 35, 45, 46]. Thus, there is a high bar to overcome for additional improvements.

## 2 Preliminaries

Let $G = (V, E, c, \omega)$ be an undirected graph with vertex set $V$, edge set $E$, node weights $c : V \to \mathbb{N}_{>0}$, and edge weights $\omega : E \to \mathbb{N}_{>0}$. We set $n := |V|$ and $m := |E|$. We extend $c$ and $\omega$ to sets, i.e., $c(V' \subseteq V) := \sum_{v \in V'} c(v)$ and $\omega(E' \subseteq E) := \sum_{e \in E'} \omega(e)$. $N(v) := \{u \mid \{u, v\} \in E\}$ denotes the neighbors of $v \in V$ and $E(V', V'') := \{\{u, v\} \in E \mid u \in V', v \in V''\}$ denotes the set of all edges between two sets of nodes $V' \subseteq V$ and $V'' \subseteq V$. The Balanced Graph Partitioning problem asks for $k$ *blocks* of nodes $\Pi := \{V_1, \ldots, V_k\}$ that partition $V$, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balance constraint* demands that $\forall i \in \{1, \ldots, k\}$: $c(V_i) \leq L_{\max} := (1 + \varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter $\varepsilon > 0$. The objective is to minimize $\text{cut}(\Pi) := \sum_{i<j} \omega(E(V_i, V_j))$ (weight of all cut edges). A *clustering* $\mathcal{C} := \{C_1, \ldots, C_b\}$ is also a partition of $V$, where the number of blocks $b$ is not given in advance (there is also no balance constraint).

## 3 Related Work

There has been extensive research on graph partitioning, so we refer the reader to recent surveys [4, 7] for a broader overview and focus here on work closely related to our contributions. As discussed above, modern general-purpose graph partitioners mostly follow the multilevel paradigm to obtain high-quality solutions. Within this framework, the coarsening phase is crucial, as it determines which structures are preserved across levels. However, coarsening offers no clear optimization objective. Consequently, most practical partitioners rely on simple, locally driven coarsening rules, such as contracting heavy-edge matchings (e.g., [11, 17, 27, 28, 32, 49]) or greedy node clusters (e.g., [2, 10, 20, 23, 37, 38, 51]). In both cases, contraction decisions are based on local neighborhood information, and thus do not explicitly incorporate global structural properties of the graph. Schlag et al. [26] mitigate this restriction to some extent by enriching matching decisions with a more global perspective by first computing communities using a community detection algorithm (Louvain clustering) on the input graph, and then restricting matches to intra-community edges. Another class of coarsening schemes [12, 42, 43] is inspired by algebraic multigrid systems. Here, nodes are divided into fractions, and different fractions of a node can belong to different clusters.

One of the most widely used refinement algorithm is the linear-time FM algorithm by Fiduccia and Mattheyses [15], which greedily moves boundary nodes according to their gain. To escape local minima, FM permits sequences that include negative-gain moves, hoping to unlock subsequent positive-gain moves that yield a net improvement. However, standard FM remains *constrained* by the balance constraint, as it only considers node moves that keep the partition feasible. More recently, several authors [18, 35, 44] have proposed *unconstrained* refinement schemes that allow temporary balance violations and restore feasibility afterward. In particular, Maas et al. [35] propose an unconstrained variant of FM refinement that also allows moves into overloaded blocks. To be more precise, the algorithm first estimates the cost of removing excess weight from each block. During the FM search, these estimates are then incorporated into the gain computations, so that a move into an overloaded block is penalized by the anticipated rebalancing cost. After the search terminates, a dedicated rebalancing step restores feasibility.

We integrate our algorithms into the Mt-KaHyPar [20] partitioning framework. Mt-KaHyPar constructs the multilevel hierarchy via community-aware coarsening [26], clustering the graph via one round of size-constrained label propagation [37] at each level. For refinement, Mt-KaHyPar traditionally relied on a parallelized constrained variant of FM, but introduced unconstrained FM refinement [35] recently.

Recent work has also started exploring the use of machine learning as part of the partitioning pipeline. These methods range from guided coarsening to full end-to-end machine learning methods. An example of the former, by Cai et al. [8], uses graph neural networks (GNN) to learn edge-weight assignments to produce improved coarsened graphs. End-to-end machine learning frameworks incorporate the balanced graph partitioning objective directly into a differential loss function. Examples of such end-to-end approaches include the Generalizable Approximate Graph Partitioning (GAP) framework by Nazi et al. [39] and the bi-objective approach by Wei et al. [50]. Closer to the non-learning methods, Gatti et al. [16] integrate a GNN reinforcement-learning (RL) agent within a multilevel framework to guide refinement. Another RL method by Barrett et al. [5] formulates partitioning as a stand-alone RL combinatorial optimization problem, introducing new exploration strategies rather than a multilevel hierarchy.

Machine learning methods are commonly built using neural network architectures that can be trained to map input features to desired outputs. A widely used baseline is the dense feed-forward neural network, also called a multilayer perceptron (MLP), which consists of a sequence of fully connected layers interleaved with nonlinear activation functions. A problem with MLP models on their own is that they operate on fixed-size feature vectors, which are not suited for the irregular structures found in graphs. In contrast, graph neural networks (GNNs) extend neural learning to graph-structured data. Rather than treating inputs as independent vectors, GNNs propagate and aggregate information along graph edges, allowing node representations to be iteratively updated based on the features of neighboring nodes and the graph topology. Commonly used architectures for the Balanced Graph Partitioning problem and other combinatorial graph problems are the Graph Convolutional Network [30] (GCN) and the Graph Sample and Aggregate [24] (GraphSAGE).

## 4 Guiding Coarsening Algorithms with Machine Learning

Current clustering approaches for multilevel coarsening are mostly based on greedily assigning nodes to neighboring clusters. The core idea of our approach is to systematically restrict the clustering to avoid contracting edges that might be cut by high-quality solutions, using a machine learning model $\mathcal{M}$ to rate individual edges. While predicting all edges correctly is itself NP-hard, any predictions better than random might already provide an improvement.

### 4.1 Edge-Restricted Clustering

In the following, we assume a machine learning model $\mathcal{M}$ which predicts values between 0 and 1, with 0 meaning that an edge can be freely contracted and 1 meaning that an edge should definitively not be contracted. Let $p_{\mathcal{M}} \colon E \to [0,1]$ be the function that maps an edge to its prediction. We extend it to any set of edges $E'$ by defining $p_{\mathcal{M}}(E') \coloneqq \sum_{e \in E'} p_{\mathcal{M}}(e)$. Our basic idea is to use a threshold $t$ for each clustering round and only contract an edge $e$ if $p_{\mathcal{M}}(e) \leq t$.

Our full algorithm builds upon the size-constrained label propagation used by the default configuration of Mt-KaHyPar [20, 22] (although we only consider plain graphs instead of hypergraphs), which we summarize in the following. The algorithm initially places every node in its own singleton cluster. Then, it iterates in parallel over all nodes, using a random order. For each node $u$, it greedily assigns $u$ to the neighboring cluster which has the strongest connection to $u$ by incident edges and where the total node weight of the cluster is below a predefined size constraint. The clustering terminates if either the number of clusters is smaller than $n$ by a factor of 2.5, or after all nodes have been visited.

For our modified algorithm, consider an unweighted graph $G$ and let $\mathcal{C}$ be the current clustering. Our first modification of the algorithm states that $u$ may be assigned to a cluster $C \in \mathcal{C}$ only if $p_{\mathcal{M}}(E(\{u\}, C)) \leq t \cdot |N(u) \cap C|$. Thus, we simply consider the average prediction for all edges leading to a cluster when deciding whether the cluster is eligible.

Apart from forbidding some contractions, we also penalize clusters which are close to the threshold but still below it, to make it more likely that another adjacent cluster is chosen. In the standard algorithm, the rating for cluster $C$ is $r(u, C) \coloneqq |N(u) \cap C|$. Let $\delta_C \coloneqq \frac{1}{t}\left(t - \frac{p_{\mathcal{M}}(E(u,C))}{|N(u) \cap C|}\right)$ be the relative distance to the threshold for an eligible cluster. Our second modification of the algorithm replaces the standard rating function with the penalized variant $r_\alpha(u, C) \coloneqq \delta_C^\alpha \cdot |N(u) \cap C|$. The parameter $\alpha$ allows to interpolate between a linear penalty, quadratic penalty, or other exponents.

## 4.2 Ensuring A Sufficient Contraction Factor

Our modified clustering algorithm can lead to problems if a majority of the edges has a high prediction, i.e., should not be contracted. Depending on the choice of $t$, a large part of the nodes might not have any eligible adjacent cluster and, consequently, is not contracted. This means that the coarsened graph is almost as large as the input graph, which might substantially degrade the overall running time of the multilevel algorithm.

To prevent the described problem, we use $\ell$ separate clustering rounds with increasing thresholds $t_1, \ldots, t_\ell$ (in practice, we use equidistant values in the interval $[0.2, 0.8]$ and $\ell = 5$). After each round, we check whether the desired reduction in node count was achieved. If yes, we terminate the clustering. If not, we start a new clustering round with a larger threshold, hopefully allowing for additional contractions. Note that we keep all clusters and only try to find new clusters for nodes that could not be clustered in the previous round. Therefore, consecutive rounds are faster since less nodes are considered. Also, for each round after the first we only aim to achieve a factor 1.75 reduction in node count instead of 2.5 – less aggressive clustering is beneficial for solution quality if a majority of edges has a high prediction (see Section 6.2).

Moreover, we introduce a *two-hop clustering* step after the regular clustering rounds. Two-hop clustering is a well-known technique [21, 23, 33] which clusters nodes that have no common neighbor but are adjacent to the same cluster, thereby allowing to contract nodes even if their neighbor clusters are overweight or not eligible due to edge predictions.

## 4.3 Predictions for Coarse Graphs

Our machine learning model is only applicable to the input graph, because we train on unweighted graphs while the coarse graphs have node and edge weights. To mitigate this, we instead use the accumulated predictions from the previous level. Let $C_1$ and $C_2$ be two adjacent clusters with corresponding nodes $c_1$ and $c_2$ in the coarse graph. Then we use $p_{\mathcal{M}}(\{c_1, c_2\}) \coloneqq p_{\mathcal{M}}(E(C_1, C_2))$ for the prediction of the coarse edge. Our modified clustering algorithm still works as described above if we also account for edge weights, i.e., we use the summed weight of the edges incident to a cluster instead of the number of neighbors. Thereby, we can still use the original predictions. Also note that the coarser levels are less important for solution quality, which makes a potential loss of accuracy more tolerable.

## 4.4 Neural Network Integration

Integrating a neural network model to guide coarsening decisions adds significant complexity compared to locally driven coarsening heuristics. At inference time, computing a set of edge predictions from the trained model involves multiple dense matrix-matrix multiplications. For large graphs with few features, these matrix multiplications consist of a tall-and-skinny matrix with many rows ($n$) and few columns (features), along with a much smaller matrix of trainable weights that depends on the number of hidden channels in the model.

While highly optimized libraries for matrix multiplication are widely available, they are typically optimized for large square-like matrices, and performance can degrade significantly as the matrix sizes approach the inner kernel size used to perform the computation. These *kernels* compute a small segment of the resulting matrix, which is approximately two by four registers in size, or 16 by four elements when using half precision and 256-bit registers [19]. Kernel code is often written for specific CPU architectures, resulting in variations in optimal shapes across different CPUs.

The performance drop for smaller matrices is particularly noticeable in popular deep learning frameworks like PyTorch, as highlighted by a similar study for graph coloring [31]. To mitigate some of the added computational cost, we implement a custom matrix kernel tailored to the dimensions of our model. Like other high-performance matrix multiplication libraries, we also leverage the single instruction, multiple data (SIMD) extensions of the x86 instruction set. Specifically, we use the AVX2 extension and 256-bit registers. Aside from the tailored shape of our kernel, the overall implementation is otherwise very similar to other publicly available kernels, such as those described by Goto and Geijn [19].

Using this custom kernel, we implement the computation of edge predictions in two phases. The first phase computes the global graph features and the specific features of every node (see Section 5.2), storing the results in an array with one feature vector per node. Since the computations are independent, we simply do this in parallel over the nodes. The second phase computes the actual edge predictions by reading the node features of both endpoints from the feature array, writing the model input features into the input matrix and then running our custom matrix multiplication kernel. We parallelize the second phase in batches of 256 edges per parallel task, which provides a reasonable trade-off between fine-grained parallelism and large enough input matrices ($256 \times 32$).

# 5 Neural Network Architecture and Training

While graph neural networks are a natural framework for learning directly on graph-structured data, we instead adopt a dense feed-forward neural network (MLP) using hand-crafted features. This choice is partly motivated by known limitations with GNN architectures [34] and the computational overhead of repeated neighborhood aggregation. Furthermore, using an MLP-based model establishes a baseline for future work to explore GNN-based architectures.

Our model $\mathcal{M}$ for predicting edge values uses a supervised deep neural network architecture with an input layer of 32 neurons, followed by three hidden layers with 48, 32, and 16 neurons, respectively. The output layer is a single neuron with a sigmoid activation function. As inputs, we use features that describe the two endpoints of the considered edge and features that relate to the graph as a whole, which are detailed in Section 5.2. Before feeding the features into the neural network, each feature is normalized to a mean of 0 and a standard deviation of 1 with a linear scaling (the normalization parameters are based on the training data). These architectural choices primarily aim to minimize the model's computational cost. Preliminary testing did not show a significant increase in prediction accuracy with larger models. There are also practical considerations of AVX2 and our matrix kernel that prevent us from reducing the number of neurons below 16 without sacrificing computational efficiency.

## 5.1 Training

Unfortunately, there is no clear way to measure how decisions in the coarsening stage affect the solution quality of the final partitioning result.[1] Our approach instead aims to predict whether an edge is likely to be cut by high-quality solutions. In practice, we generate training data by repeatedly running an expensive partitioning algorithm on small training instances (using $k \in \{4, 8, 12, 16, 20, 24, 28, 32, 48, 64, 96, 128\}$ and 10 repetitions per value of $k$). On the resulting set of 120 high-quality solutions, we simply compute the fraction of solutions that cut a given edge $e$ and train $\mathcal{M}$ to predict this fraction.

[1] Multilevel partitioning algorithms are highly non-linear combinatorial algorithms that have multiple stages with complex dependencies.

Our algorithm for generating high-quality solutions is a variant of the memetic partitioning algorithm proposed by Andre et al. [3]. The idea is to use a memetic approach as a meta-heuristic on top of a multilevel algorithm, generating an initial population with ordinary partitioning runs and then using multilevel mutation and recombination operations to find new solutions. In our approach, we replace KaHyPar with Mt-KaHyPar as the underlying multilevel algorithm. Overall, this allowed us to generate high-quality training data with a time requirement between multiple minutes and a few hours per graph in the training set.

To facilitate efficient training of $\mathcal{M}$, we limit our dataset to a subset of these labeled edges rather than processing the full edge sets of all graphs. We constructed a fixed dataset of one million edges to serve as the basis for training by randomly sampling an equal number of edges from each graph in the training set. This approach guarantees that every graph contributes equally to the dataset, preventing larger graphs from dominating the training process. The collected data was subsequently split into training (70%), validation (15%), and testing (15%) sets. This strategy challenges the model $\mathcal{M}$ to generalize in two distinct ways: it must correctly classify unseen edges within the training graphs (evaluated via the test set) and it must adapt to entirely new graph instances (evaluated by the model's performance within the Mt-KaHyPar partitioner).

For the training objective, we utilized the mean squared error (MSE) loss function. Optimization was performed using the Adam optimizer [29] with an initial learning rate of 0.001 and a weight decay of $1 \times 10^{-5}$. To enhance model performance, we implemented a learning rate scheduler which reduces the learning rate by a factor of 10 if the validation does not improve significantly in 10 epochs. We also employed early stopping with a patience of 10 and neuron dropout with a probability of 50% to mitigate overfitting.

## 5.2 Features

As shown in Table 1, we consider features that mostly consist of simple metrics which can be computed without much time overhead (e.g., statistics relating to the degree sequence of the graph). For predicting a given edge $\{u, v\}$, we include global features that describe the graph as a whole, node features for both $u$ and $v$ that relate to the node and the structure of its neighborhood, and edge features that relate to both $u$ and $v$ at once.

In addition, we exploit a preprocessing step of Mt-KaHyPar to get access to additional features. Mt-KaHyPar pre-computes a modularity clustering [20, 22] with a parallel version of the Louvain algorithm [6]. Then, contractions during the coarsening phase are only allowed within the same community (this was originally developed for KaHyPar [26], see Section 3).

We exploit this by extracting multiple features that characterize the computed Louvain clustering, describing the structure of the communities and the achieved modularity. Since the Louvain algorithm computes a hierarchical clustering [6], we extract features from multiple levels of the clustering: the final level as well as the two levels before the final level. The features are computed separately for each of the three levels (entries in Table 1 are marked accordingly). Note that these features would be comparatively expensive to compute again, but since the community detection is already a part of the regular partitioning algorithm we can access them without additional overhead. Moreover, Table 2 shows features that require more computation time, since computing them is (at least) equivalent to counting the triangles contained in the graphs – which is particularly expensive if the graph contains many nodes with high degree. The idea is that these features are more expensive but might also improve the prediction accuracy. However, they are not included in our final configuration since the improvements turned out to be rather insubstantial (see Section 6.1).

**Table 1** Features that can be computed with small overhead. Features either describe the whole graph (`global`), an endpoint $u$ of the considered edge (`node`), or both endpoints (`edge`). The features related to Louvain clustering are replicated for three levels of the Louvain hierarchy (indicated by $\{0, 1, 2\}$). All features are scaled to have a mean of 0 and a standard deviation of 1, see Section 5.

| Type | Feature Name | Description |
|---|---|---|
| `global` | $n$ | Number of nodes |
| `global` | $m$ | Number of edges |
| `global` | Global Node Degrees | Multiple statistical features relating to the global degree sequence (e.g., average, median, skew, and entropy) |
| `global` | Irregularity | Standard deviation of degrees divided by average degree |
| `global` | Communities $\{0, 1, 2\}$ | # communities in the Louvain clustering |
| `global` | Modularity $\{0, 1, 2\}$ | Modularity value of the Louvain clustering |
| `node` | Degree | Node degree of $u$ |
| `node` | Degree Quantile | Quantile of node degree in the global degree sequence |
| `node` | Neighbor Degrees | Multiple statistical features relating to the neighbor degrees (e.g., average, median, skew, and entropy) |
| `node` | Degree Deviation | $\chi^2$ distance of the neighbor degrees to the average degree |
| `node` | Degree 1 Neighbors | # neighbors with degree 1 |
| `node` | Min Contracted Degree | Minimum rate of # outgoing edges divided by node weight if a part of the neighborhood is contracted onto $u$ |
| `edge` | Strawman Similarity | Defined as $((d(u) - 1)(d(v) - 1))^{-1}$ |
| `edge` | Same Community $\{0, 1, 2\}$ | Whether $u$ and $v$ are in the same Louvain community |

**Table 2** Features that are more expensive to compute (excluded from the final configuration).

| Type | Feature Name | Description |
|---|---|---|
| `node` | Local Clustering Coefficient | # edges in the neighborhood divided by # possible edges |
| `node` | Triangles | # triangles containing the node |
| `node` | External Edges | # edges from the neighborhood to an external node |
| `node` | Neighborhood Modularity | Contribution the neighborhood of the node makes to total modularity if it is considered a single community |
| `node` | Max Neighbor Modularity | Modularity contribution of a subset of the neighborhood with maximum modularity |

## 6 Experiments

**Setup.** We implemented our approach within the Mt-KaHyPar [20] framework and compiled it using `gcc` 14.2.0 with flags `-O3 -mtune=native -march=native`[2]. The code is parallelized using TBB [40]. All experiments are performed 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 64 threads.

**Benchmark Sets.** We use several distinct graph datasets for our evaluation, as summarized in Table 3. Set A consists of 118 instances with small to medium size (29 thousand edges to 53 million edges). It includes graphs from the SuiteSparse Matrix Collection [13] and

[2] A link to the (open source) implementation is omitted to preserve anonymity, but it will be included in the published version of the paper, together with the benchmark sets and experimental results.

the Network Repository [41], which are derived from scientific simulations, social networks, website link structures and other real-world data sources. We also added 18 instances from the Walshaw graph partitioning benchmark set [47] and 10 instances from the DAC 2012 Routability-Driven Placement Contest [48], which are derived from semiconductor design. Moreover, the set includes synthetic graphs from random graph models such as the R-MAT and Erdős–Rényi models, random geometric graphs, and Delaunay graphs. Set B is a subset of Set A, comprising 69 instances. We further divided Set B into three subsets based on the mean value of the training labels; see Section 6.1 for details.

In addition, we use two benchmark sets that were introduced by Maas et al. [35, 36]. Set I consists of 38 large, highly irregular graphs, and Set R consists of 33 large, fairly regular graphs. Here, irregularity is measured as the standard deviation of the degree distribution relative to the average degree.

**Table 3** Overview of graph benchmark sets that are used in the experimental evaluation.

| Name | Size | Subset of | Min # Edges | Max # Edges | Reference |
|---|---|---|---|---|---|
| Set A | 118 | – | 29 k | 53 M | Appendix A |
| Set B | 69 | Set A | 45 k | 53 M | Table 8 |
| Set $B_{1\text{-}20}$ | 20 | Set B | 100 k | 34 M | – |
| Set $B_{21\text{-}45}$ | 25 | Set B | 45 k | 53 M | – |
| Set $B_{46\text{-}69}$ | 24 | Set B | 119 k | 27 M | – |
| Set I | 38 | – | 5.4 M | 1.8 B | [36] |
| Set R | 33 | – | 12 M | 575 M | [36] |

**Baseline and Competitors.** We consider the following baseline algorithms and competitors to evaluate the performance of our approach.

**Mt-KaHyPar.** We use two baselines based on Mt-KaHyPar. The first corresponds to the current default configuration of Mt-KaHyPar and employs the recently introduced unconstrained refinement technique [35]. The second baseline uses traditional (constrained) FM refinement and corresponds to the previous default configuration of Mt-KaHyPar. Here, the coarsening phase has a larger impact on quality since constrained refinement is less likely to make large changes to the initial solution and is thereby more dependent on the solution quality of coarser levels. Overall, this allows us to evaluate the impact of our approach on both the current state-of-the-art as well as other common configurations. In both baselines, we include small adjustments to the coarsening and preprocessing phase that match the algorithmic changes proposed in Section 4, so that we can evaluate the impact of our machine learning model in isolation (see Appendix B for the impact on the baseline).

**METIS**. Perhaps the most widely known graph partitioning system, we include Metis [27] in both the recursive bisection configuration (Metis-R) and the multilevel $k$-way configuration (Metis-K). Note that Metis is a sequential algorithm, and thus we run it on a single core (we refer to previous comparisons with Mt-KaHyPar [20] for ParMetis and Mt-Metis, as comparing existing partitioners is not the primary goal of this work).

**E2E**. The end-to-end bi-objective approach by Wei et al. [50] is the most recent ML method at the time of writing. Furthermore, it is among the few learning-based approaches that consider the Balanced Graph Partitioning problem directly, rather than a minimum normalized cut. E2E also scales linearly in the size of the graph, where other alternatives such as ECORD by Barret et al. [5] need $\mathcal{O}(n^2)$ running time. We configure E2E with the same hyperparameters as the authors used in their experiments, with a learning rate of

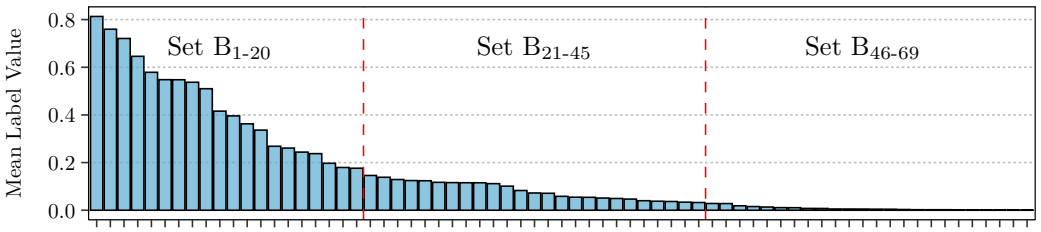


**Figure 1** Distribution of mean label values over the Set B. Each bar represents the arithmetic mean over the edge labels for one graph in the benchmark set.

0.005 and $\alpha = 1$. We also use 500 training epochs with an early stopping patience of 100. Note that E2E is fundamentally different from our approach. Our method performs training up-front, known as inductive learning, whereas E2E trains a new model for each input graph, a form of transductive learning.

**Methodology.** We use $k \in \{4, 8, 32, 64\}$ and, as is standard in the literature, $\epsilon = 0.03$ for all experiments. We run each instance (i.e., combination of graph and $k$) with five different seeds for randomization and a time limit of 1 hour, reporting the arithmetic mean over the seeds for the cut size and running time. When aggregating over multiple instances, we use the geometric mean. In aggregated running times, we include runs with imbalanced partitions and replace the value for any run that exceeded the time limit with the time limit itself.

**Performance Profiles.** We use *performance profiles* [14] to compare the relative solution quality of competing algorithms. Let $\mathcal{A}$ be the set of algorithms, $\mathcal{I}$ the set of instances, and $c_A(I)$ the cut of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. We plot one line for each algorithm $A$, showing the fraction of instances ($y$-axis) which are within a factor of $\tau$ ($x$-axis) of the best found solution, i.e., $c_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} c_{A'}(I)$. An algorithm performs better if it achieves lower $\tau$ values for higher fractions of instances (i.e., the line is in the top left). Additionally, we use $\times$ (for imbalanced partitions) and $\circledcirc$ (for out-of-time) as markers on the $x$-axis.

## 6.1 Model Selection and Analysis

During the training, we observed substantially different model performance depending on the input graph. Some instances yield cut improvements of more than 10% over the baseline, while there are also graphs without any improvement. A property which can partially explain this is the uneven label distribution in the training data (i.e., the frequency of edge cuts in high-quality partitions, see Section 5.1) – the mean over the labels varies substantially on individual graphs. Based on this, we divided Set B into three subsets of approximately equal size: high, medium, and low label values (denoted as Set $B_{1\text{-}20}$, Set $B_{21\text{-}45}$, and Set $B_{46\text{-}69}$). Figure 1 visualizes the resulting distribution of mean label values.

**Models.** We evaluate multiple models by varying both the training data and the features, as shown in Table 4. Models are either trained only on Set $B_{1\text{-}20}$ or on all graphs of Set B. We use three different feature sets: Feature Sets A and B include only computationally cheap features (see Table 1), while Feature Set C also includes more expensive features (see Table 2). To select a concrete subset of 32 features, we applied a Random Forest regressor to rank the features by importance. Feature Sets A and C were selected with the regressor trained on Set $B_{1\text{-}20}$, Feature Set B was selected with Set B as training input.

**Table 4** Overview of the machine learning models, their input features and their training data.

| Model | Feature Set | Training Set |
|---|---|---|
| Data20-FeatSrc20 (ML-Clustering) | Feature Set A | Set $B_{1\text{-}20}$ |
| Data69-FeatSrc20 | Feature Set A | Set B |
| Data20-FeatSrc69 | Feature Set B | Set $B_{1\text{-}20}$ |
| Data69-FeatSrc69 | Feature Set B | Set B |
| ML-Clustering-Heavy | Feature Set C | Set $B_{1\text{-}20}$ |
| Fold Models | Feature Set A | Subset of Set A |

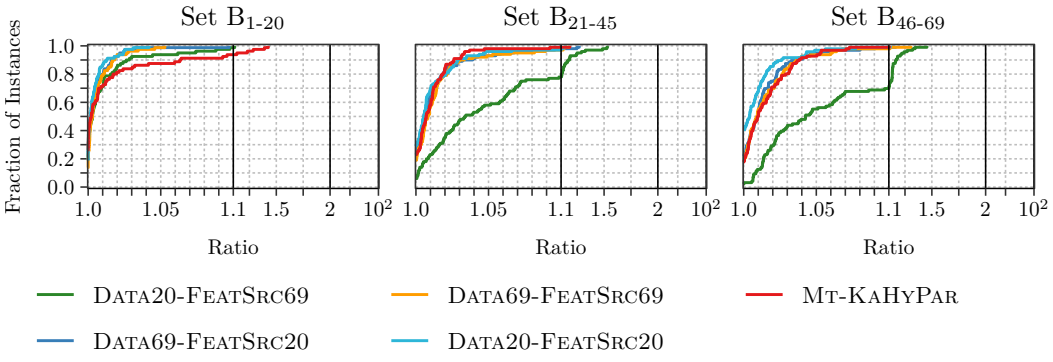


**Figure 2** Comparison of partition quality between our approach and the unconstrained baseline across three datasets. Left: Evaluation on Set $B_{1\text{-}20}$. Middle: Evaluation on Set $B_{21\text{-}45}$. Right: Evaluation on Set $B_{46\text{-}69}$. See Table 4 for details on the models.

**Model Evaluation.** Figure 2 shows the end-to-end performance of the models that use cheap features on each subset of Set B. First, we observe that all models perform significantly better on Set $B_{1\text{-}20}$ compared to the other subsets. This means that our approach yields the most improvement on graphs with high cut values, while there are only minor changes on other instances. Note that this is not simply an artifact of the training process. While we can generate training data with higher label values by adjusting the parameters for the data generation, we did not observe any improvement in model performance when we attempted this in preliminary experiments. The likely reason is that these graphs have an inherently different structure which limits the achievable performance of our model.

Comparing the models, Data20-FeatSrc69 performs substantially worse than the remaining models, possibly due to some kind of overfitting. Data20-FeatSrc20 achieves the best performance by a slight margin. This is surprising insofar that this model uses the smaller data set for both feature selection and training. It also indicates that it is possible to train on graphs where substantial quality improvements are feasible, without a negative impact on instances outside of the training distribution – an unexpected but highly welcome property. In the following, we use Data20-FeatSrc20 as the default model, denoted as ML-Clustering.

We provide a comparison to ML-Clustering-Heavy on Set $B_{1\text{-}20}$ in Figure 3. Despite having access to more expensive features, ML-Clustering-Heavy achieves only a small improvement of 1.5% in the geometric mean when using constrained refinement. Moreover, this does not translate to any improvement when using state-of-the-art unconstrained refinement. Since ML-Clustering-Heavy incurs a relative running time overhead of 33% on Set B and does not provide any quality improvement on Set $B_{21\text{-}45}$ or Set $B_{46\text{-}69}$, we use ML-Clustering for our final configuration.

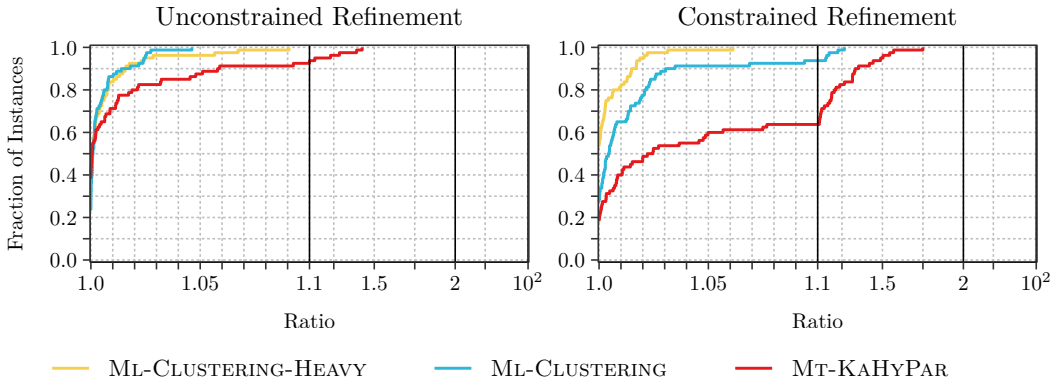


**Figure 3** Comparison of partition quality between ML-CLUSTERING (cheap features) and ML-CLUSTERING-HEAVY (expensive features) on Set $B_{1\text{-}20}$.

**Table 5** Percentage change in cut sizes compared to MT-KAHYPAR for each fold across unconstrained and constrained refinement strategies.

| Fold | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Unconstrained (%) | -0.88 | -0.23 | -0.43 | -0.67 | +0.02 | -0.03 | -0.26 | -1.67 |
| Constrained (%) | -2.79 | -1.74 | -1.71 | -0.23 | -0.54 | -1.55 | -0.43 | -2.89 |

***k*-Fold Cross-Validation.** To evaluate the generalization capabilities of our approach, we conducted an 8-fold cross-validation on Set A. We evenly divided the graph set into eight folds via random assignment. We trained eight distinct models, where each model utilized seven folds for training and the remaining fold for validation. Edge sampling was performed as described in Section 5.1. Following training, each model generated inference labels for the unseen fold, which were subsequently used for the guided partitioning algorithm. This approach allows us to simulate the full integration of the model into the partitioner.

We compare the results to the MT-KAHYPAR baseline algorithm in Table 5. The results exhibit some variance between the folds, but the general trend confirms the model's generalization capability; some folds match the baseline quality while others demonstrate a clear improvement (up to 1.67% for unconstrained refinement and up to 2.89% for constrained refinement). As expected, we see larger improvements when using constrained refinement.

## 6.2 Algorithm Parameter Tuning

Our guided coarsening algorithm (see Section 4) includes multiple parameters that control how the predicted edge values are used in the clustering. The most important parameters are the number of clustering subrounds, the target shrink factor for the secondary rounds

**Table 6** Parameter evaluation on Set A, comparing the geometric mean differences in edge cut and running time relative to the final configuration (5 subrounds, shrink factor 1.75 and $\alpha = 2$).

| Parameter | Subrounds | | | Shrink Factor | | | Penalty | | |
|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 7 | 1.5 | 2 | 2.5 | $\alpha = 0$ | $\alpha = 1$ | $\alpha = 3$ |
| Cut Change (%) | +0.11 | +0.14 | +0.06 | +0.09 | +0.13 | +0.10 | +0.06 | +0.16 | +0.20 |
| Time Delta (%) | -0.39 | -0.62 | +0.59 | +0.56 | -0.73 | -0.84 | -0.29 | +0.67 | +0.29 |

and the penalty parameter $\alpha$. We evaluated different choices for these parameters with the unconstrained refinement strategy and the ML-Clustering model on Set A. The results are summarized in Table 6.

As shown in the table, the parameters chosen for the final configuration provide optimal solution quality with only a minimal increase in running time when compared to other choices. However, the overall impact is very small (less than 1% in the geometric mean in each case). This indicates that our method is robust with regard to the details of the clustering algorithm, while it also seems unlikely that further parameter tuning can achieve meaningful improvements.

## 6.3 Final Evaluation

**Quality.** Figure 4 (top) evaluates the performance of our approach (ML-Clustering) on Set A, comparing it to the baseline configuration of Mt-KaHyPar, to Metis and to the state-of-the-art ML method E2E. Our approach achieves consistently the best quality, providing an improvement over baseline Mt-KaHyPar (0.34% and 1.4% geometric mean difference when using unconstrained or constrained refinement, respectively). Both algorithms produce substantially better solutions than Metis (roughly 20% geometric mean difference[3]), while E2E is unable to find a balanced partition on most instances, and usually does not achieve a competitive cut ratio even if the solution is feasible. The results are similar even if we include partitions with large imbalance to accommodate the different balance definition of E2E (see Appendix B, Figure 7).

Our approach achieves a substantially larger improvement for the configuration with constrained refinement (right). Most likely, this is because we generally achieve a superior coarsening structure, but unconstrained refinement is capable of finding good solutions even if less structure is preserved, thereby reducing the available space for improvement. Finally, we can observe in Figure 4 (bottom) that the quality improvement is quite substantial on Set $B_{1\text{-}20}$ – demonstrating that our model achieves large quality improvements on graphs with beneficial properties while behaving similar to the baseline on other instances. Consequently, we achieve heuristic improvements on some instances without losing quality on out-of-distribution inputs.[4]

**Running Time.** Table 7 shows the geometric mean running time of the considered algorithms. Baseline Mt-KaHyPar is the fastest algorithm, while our approach adds 14% running time overhead on average. Metis-K has a similar running time (Metis-R is about 2× slower) using only a single thread, which makes it more efficient in terms of compute resources – at the cost of substantially worse solution quality. E2E is not competitive on the considered instances, usually requiring multiple minutes per run and even exceeding the time limit of 1 hour in a few cases.

To analyze the overhead of our approach in more detail, we provide a breakdown of the running time by different parts of the algorithm in Figure 5. On 90% of graphs in Set A, the sum of feature computation and model inference require at most 20% of the total running time. An exception exists for a few instances which contain nodes with extremely large

[3] Also, there were multiple instances where Metis-K produced slightly imbalanced partitions (a block with one node more than allowed), which seems to be caused by inexact rounding in the implementation. We consider these partitions to still be balanced in our presentation of results.

[4] Note that the model is trained only on a small subset of edges from the 20 graphs of Set $B_{1\text{-}20}$, i.e., most of the benchmark set consisting of 118 graphs is not part of the training data.

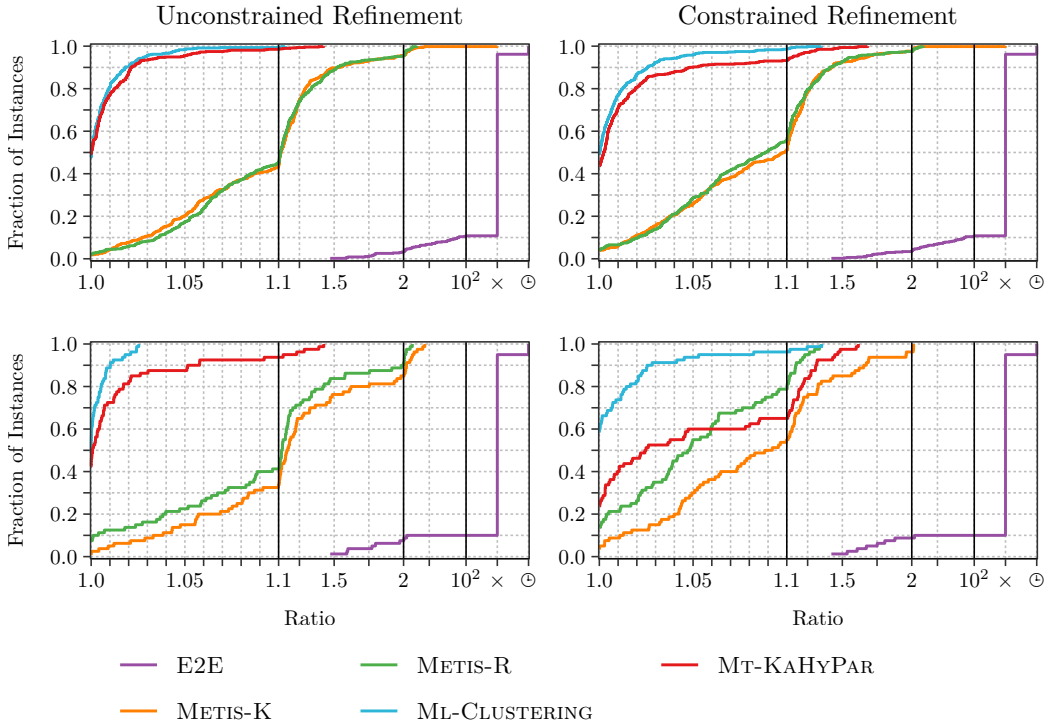


**Figure 4** Comparison of ML-Clustering against the Mt-KaHyPar baseline and competitors E2E and Metis. Top: Partition quality evaluation on Set A. Bottom: Partition quality on Set $B_{1\text{-}20}$.

**Table 7** Geometric mean values for running time and cut on Set A. Cut values are relative to Mt-KaHyPar.

| Model | Set A | | Set $B_{1\text{-}20}$ | |
|---|---|---|---|---|
| | Time | Rel. Cut (%) | Time | Rel. Cut (%) |
| ML-Clustering | 0.41 s | -0.34 | 0.80 s | -2.00 |
| Mt-KaHyPar | 0.36 s | 0.00 | 0.78 s | +0.00 |
| ML-Clustering (Constrained) | 0.39 s | +3.98 | 0.73 s | +15.76 |
| Mt-KaHyPar (Constrained) | 0.34 s | +5.42 | 0.71 s | +25.13 |
| Metis-R | 0.68 s | +20.05 | 1.18 s | +21.66 |
| Metis-K | 0.38 s | +20.34 | 1.20 s | +35.96 |
| E2E | 158.91 s | +608.14 | 203.89 s | +79.25 |

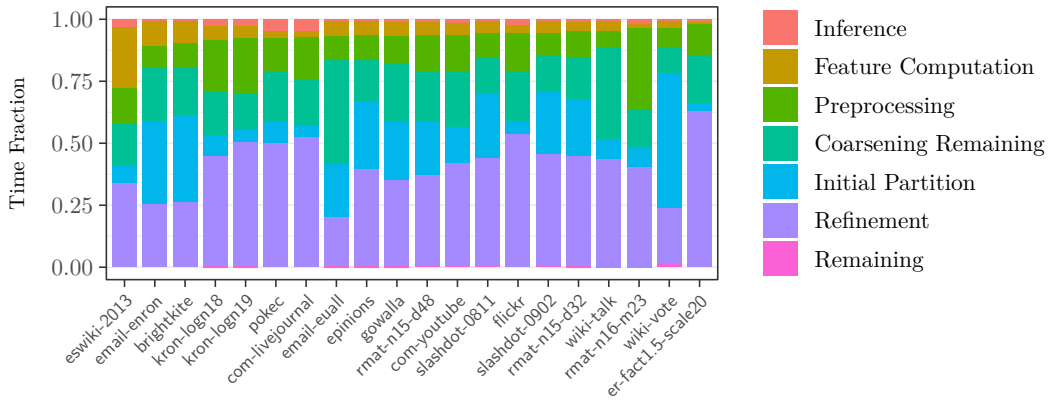


**Figure 5** Normalized running times of Mt-Clustering on Set $B_{1\text{-}20}$ with $k = 8$. The $x$-axis is sorted by the sum of inference and feature computation time relative to the total running time.

degree, notably the eswiki-2013 graph (27.8% of total time for $k = 8$). Here, the feature computation is more expensive due to the need to sort all node neighborhoods by degree. Considering the arithmetic mean of relative contributions over all runs, feature computation and inference contribute 10.9% running time overhead on Set A and 6.5% on Set $B_{1\text{-}20}$. The remaining part of the total overhead of 14% can be explained by the fact that the clustering algorithm needs to use more rounds if some contractions are forbidden by the edge predictions, thereby increasing the coarsening time.

Overall, these results show that our guided coarsening approach can be integrated into a state-of-the-art multilevel partitioner with small overhead. There are also plausible options to further improve efficiency in future work, for example a portfolio approach that selectively enables guided coarsening on instances where it is likely to yield substantial quality improvements. This is particularly compelling since the overhead is significantly smaller on these instances (Set $B_{1\text{-}20}$).

**Evaluation on Large-Scale Graphs.** To assess the robustness of our approach, we also evaluate the partition quality on Set I and Set R, which contain significantly larger graph instances. The results are presented in Figure 6. Consistent with our previous findings, we observe a modest but noticeable improvement in solution quality on Set I, while the performance on Set R is similar to the baseline. This demonstrates that our approach generalizes even to graphs that are much larger than any of the training input.

## 7 Conclusion

Current state-of-the-art graph partitioners achieve their performance through decades of algorithm engineering, whereas purely end-to-end machine learning approaches have so far struggled to match their solution quality and running time efficiency. We demonstrate that these approaches are not in conflict, but that machine learning can yield practical gains when it is used to improve specific heuristic decisions inside a strong classical partitioner. By integrating a pre-trained neural model into Mt-KaHyPar, we guide the coarsening phase of multilevel partitioning with per-edge scores that indicate how likely an edge is to be cut by a high-quality solution. This tackles a long-standing problem of classical coarsening, where there is no reliable local objective to guide contraction decisions. By learning which contractions are unlikely to conceal high-quality solutions, we construct hierarchies that better

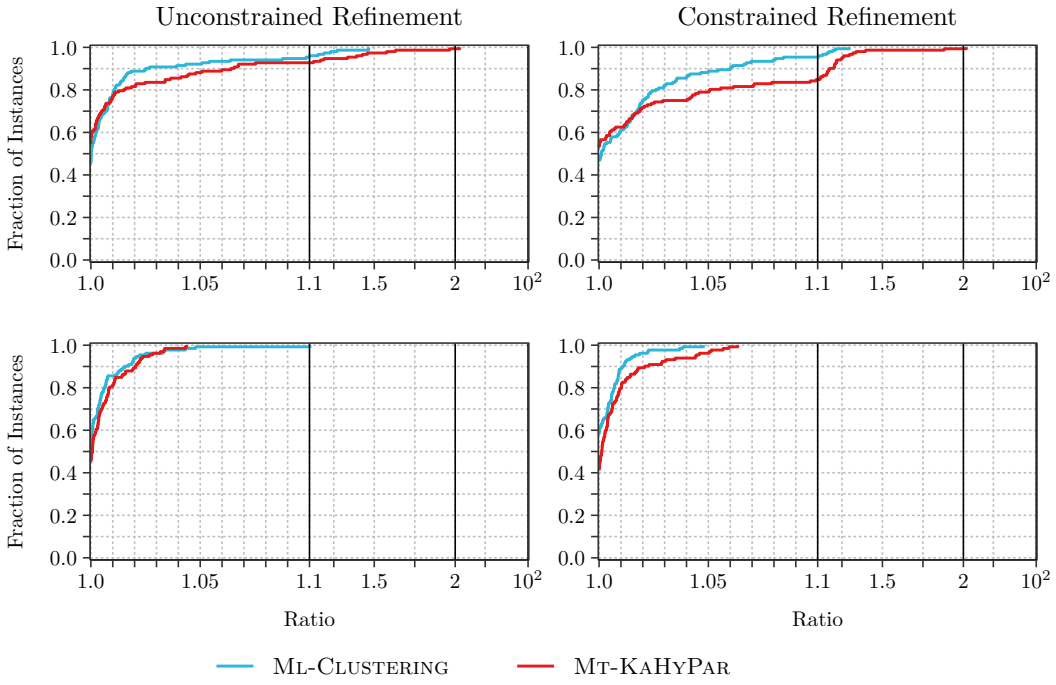


**Figure 6** Partition quality of ML-CLUSTERING on the large graph instances in comparison to the baseline. The top shows the evaluation on Set I while the bottom shows the evaluation on Set R.

preserve good cuts. Our guided coarsening improves solution quality over the MT-KAHYPAR baseline by 0.3%–1.4%, depending on benchmark set, while increasing running time by only 14% on average. These results suggest that a pragmatic path forward for machine learning in graph partitioning is to develop tightly integrated, performance-aware learned heuristics that augment mature multilevel partitioners.

## References


1. Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a Direct $k$-way Hypergraph Partitioning Algorithm. In *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 28–42, 2017. doi:10.1137/1.9781611974768.3.
2. Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-Quality Shared-Memory Graph Partitioning. *IEEE Transactions on Parallel Distributed Systems*, 31(11):2710–2722, 2020. doi:10.1109/TPDS.2020.3001645.
3. Robin Andre, Sebastian Schlag, and Christian Schulz. Memetic Multilevel Hypergraph Partitioning. In *Genetic and Evolutionary Computation Conference, (GECCO)*, pages 347–354, 2018. doi:10.1145/3205455.3205475.
4. David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph Partitioning and Graph Clustering*, volume 588. American Mathematical Society, 2013. doi:10.1090/conm/588.
5. Thomas D. Barrett, Christopher W.F. Parsonson, and Alexandre Laterre. Learning to Solve Combinatorial Graph Partitioning Problems via Efficient Exploration. arXiv preprint, 2022. doi:10.48550/arXiv.2205.14105.
6. Vincent D. Blondel, Jean Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008. doi:10.1088/1742-5468/2008/10/P10008.

7 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.

8 Chen Cai, Dingkang Wang, and Yusu Wang. Graph Coarsening with Neural Networks. In *9th International Conference on Learning Representations (ICLR)*, 2021.

9 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.

10 Umit V. Catalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. Multithreaded Clustering for Multi-level Hypergraph Partitioning. In *26th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 848–859. IEEE Computer Society, 2012. doi:10.1109/IPDPS.2012.81.

11 Cédric Chevalier and François Pellegrini. PT-Scotch: A Tool for Efficient Parallel Graph Ordering. *Parallel Computing*, 34(6-8):318–331, 2008. doi:10.1016/j.parco.2007.12.001.

12 Cédric Chevalier and Ilya Safro. Comparison of Coarsening Schemes for Multilevel Graph Partitioning. In *3rd Conference on Learning and Intelligent Optimization (LION)*, volume 5851, pages 191–205. Springer, 2009. doi:10.1007/978-3-642-11169-3_14.

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

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 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.

16 Alice Gatti, Zhixiong Hu, Tess Smidt, Esmond G. Ng, and Pieter Ghysels. Graph Partitioning and Sparse Matrix Ordering using Reinforcement Learning and Graph Neural Networks. *Journal of Machine Learning Research*, 23(303):1–28, 2022. URL: https://jmlr.org/papers/v23/21-0644.html.

17 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.

18 Michael S. Gilbert, Kamesh Madduri, Erik G. Boman, and Sivasankaran Rajamanickam. Jet: Multilevel Graph Partitioning on GPUs, 2023. doi:10.48550/arXiv.2304.13194.

19 Kazushige Goto and Robert A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008. doi:10.1145/1356052.1356053.

20 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.

21 Lars Gottesbüren, Nikolai Maas, Dominik Rosch, Peter Sanders, and Daniel Seemaier. Linear-Time Multilevel Graph Partitioning via Edge Sparsification. In *33th European Symposium on Algorithms (ESA)*, volume 351, pages 32:1–32:20, 2025. doi:10.4230/LIPIcs.ESA.2025.32.

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 Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *31th Conference on Neural Information Processing Systems (NIPS 2017)*, pages 1–11. Curran Associates, 2017.

25 Tobias Heuer, Lars Gottesbüren, Nikolai Maas, and Simeon Schrape. Mt-KaHyPar. Software, version 1.5.3., swhId: swh:1:dir:0285e232ceaf8b004e75d01d1e5f4e6984770663 (visited on 2026-05-29). URL: https://github.com/kahypar/mt-kahypar/tree/sea2026, doi:10.4230/artifacts.26212.

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 and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. doi:10.1137/S1064827595287997.

28 George Karypis and Vipin Kumar. Parallel Multilevel k-Way Partitioning Scheme for Irregular Graphs. *Siam Review*, 41(2):278–300, 1999. doi:10.1137/S0036144598334138.

29 Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2017. doi:10.48550/arXiv.1412.6980.

30 Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations (ICLR)*, 2017.

31 Kenneth Langedal and Fredrik Manne. Graph Neural Networks as Ordering Heuristics for Parallel Graph Coloring. In *27st Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 56–67. Society for Industrial and Applied Mathematics, 2025. doi:10.1137/1.9781611978339.5.

32 Dominique Lasalle and George Karypis. Multi-threaded Graph Partitioning. In *27th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 225–236, 2013. doi:10.1109/IPDPS.2013.50.

33 Dominique LaSalle, Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. Improving Graph Partitioning for Modern Graphs and Architectures. In *5th Workshop on Irregular Applications - Architectures and Algorithms (IA3)*, pages 14:1–14:4. ACM, 2015. doi:10.1145/2833179.2833188.

34 Andreas Loukas. What Graph Neural Networks Cannot Learn: Depth vs Width. In *8th International Conference on Learning Representations (ICLR)*, 2020.

35 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.

36 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.

37 Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning Complex Networks via Size-Constrained Clustering. In *13th International Symposium on Experimental Algorithms (SEA)*, pages 351–363. Springer, 2014. doi:10.1007/978-3-319-07959-2_30.

38 Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel Graph Partitioning for Complex Networks. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2625–2638, 2017. doi:10.1109/TPDS.2017.2671868.

39 Azade Nazi, Will Hang, Anna Goldie, Sujith Ravi, and Azalia Mirhoseini. GAP: Generalizable Approximate Graph Partitioning Framework. arXiv preprint, 2019. doi:10.48550/arXiv.1903.00614.

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

41 Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *29th Conference on Artificial Intelligence (AAAI)*, 2015. doi:10.1609/AAAI.V29I1.9277.

42 Ilya Safro, Dorit Ron, and Achi Brandt. Multilevel algorithms for linear ordering problems. *ACM J. Exp. Algorithmics*, 13, 2008. doi:10.1145/1412228.1412232.

43 Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. *ACM J. Exp. Algorithmics*, 19(1), 2014. doi:10.1145/2670338.

**44** 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. ACM, 2024. doi:10.1145/3626183.3660257.

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

**46** Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-Quality Hypergraph Partitioning. *ACM Journal of Experimental Algorithmics*, 27:1.9:1–1.9:39, 2022. doi:10.1145/3529090.

**47** Alan J. Soper, Chris Walshaw, and Mark Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004. doi:10.1023/B:JOGO.0000042115.44455.F3.

**48** 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.

**49** Chris Walshaw, Mark Cross, and Martin G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997. doi:10.1006/jpdc.1997.1407.

**50** Pengcheng Wei, Yuan Fang, Zhihao Wen, Zheng Xiao, and Binbin Chen. An end-to-end bi-objective approach to deep graph partitioning. *Neural Networks*, 181:106823, 2025. doi:10.1016/j.neunet.2024.106823.

**51** 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.


## A Details on the Benchmark Sets

**Table 8** Graphs in Set B, by graph class.

| Class | # | Graphs |
|---|---|---|
| Collaboration | 6 | citation-citeseer, coauthors-citeseer, copapers-citeseer, coauthors-DBLP, copapers-DBLP, cond-mat |
| Social | 14 | com-amazon, com-DBLP, com-livejournal, com-youtube, email-enron, email-euall, imdb-2021, brightkite, gowalla, epinions, slashdot-0811, slashdot-0902, flickr, pokec |
| Web | 7 | as-skitter, cnr-2000, in-2004, berk-stan, google, notre-dame, stanford |
| Wiki | 3 | eswiki-2013, wiki-talk, wiki-vote |
| Brain | 2 | bn-M87117515, bn-M87123142 |
| Road | 4 | osm-italy, osm-luxembourg, road-CA, road-TX |
| Simulation | 8 | 4elt, M6, hugebubbles-0, hugetrace-0, hugetric-0, thermal2, venturi-level3, ecology1 |
| Semiconductor | 4 | G3-circuit, superblue7, superblue11, superblue19 |
| Walshaw | 6 | bcsstk29, bcsstk30, fe-ocean, fe-sphere, wave, wing |
| Artificial | 14 | delaunay-n18, delaunay-n19, delaunay-n20, er-fact1.5-scale20, kron-logn18, kron-logn19, rgg-n15, rgg-n17, rgg-n19, rhg-n17, rhg-n18, rmat-n15-d32, rmat-n15-d48, rmat-n16-m23 |
| Other | 1 | kkt-power |

**Table 9** Graphs in Set A which are not contained in Set B, by graph class.

| Class | # | Graphs |
|---|---|---|
| Collaboration | 4 | astro-ph, hep-ph, cond-mat-2003, cond-mat-2005 |
| Social | 3 | amazon-0302, DBLP-2011, libimseti |
| Web | 6 | amazon, amazon-2008, as-22july06, deu, eu-2005, p2p-Gnutella04 |
| Wiki | 4 | dewiki-2013, frwiki-2013, itwiki-2013, wiki-topcats |
| Simulation | 6 | geo-1438, hook-1498, serena, bone-s10, ldoor, packing-b050 |
| Semiconductor | 11 | circuit-5M, nv2, cant-spmv, scircuit-spmv, superblue2, superblue3, superblue6, superblue9, superblue12, superblue14, superblue16 |
| Artificial | 2 | smallworld, ba-n22-d2 |
| Walshaw | 12 | 144, auto, brack2, fe-body, fe-rotor, fe-tooth, finan512, m14b, memplus, t60k, vibrobox, wing-nodal |
| Other | 1 | wordassociation-2011 |

## B Additional Experimental Results

In the following, we provide an additional quality comparison under large imbalance (Figure 7). Moreover, our baseline slightly differs from the main version of Mt-KaHyPar: we include two-hop coarsening (see Section 4.2), and the community detection preprocessing step is always enabled since it is required for the feature computation (default Mt-KaHyPar uses a heuristic to disable the preprocessing on mesh graphs). Figure 8 shows that our modified baseline actually has better quality, but also higher running time.

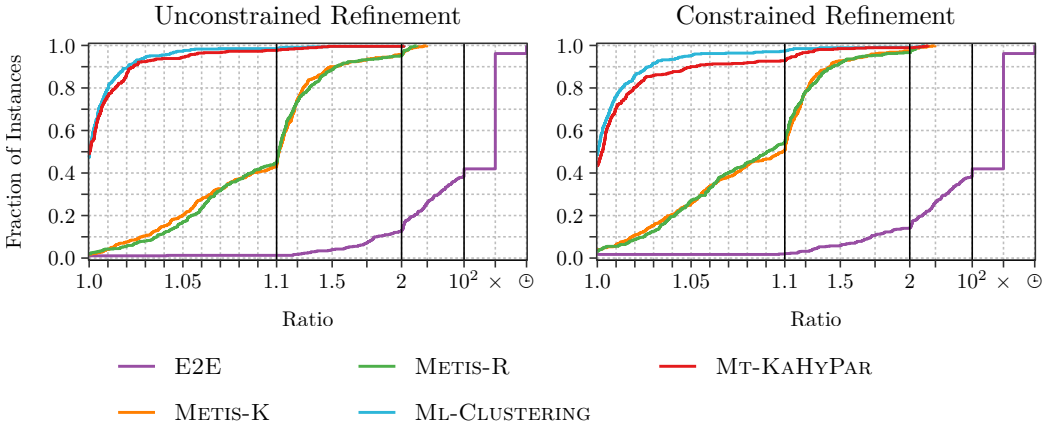


**Figure 7** Comparison of Ml-Clustering against the Mt-KaHyPar baseline and competitors E2E and Metis on Set A, allowing extreme imbalances of up to $\epsilon = 1$ so that more of the often imbalanced outputs of E2E can be included.

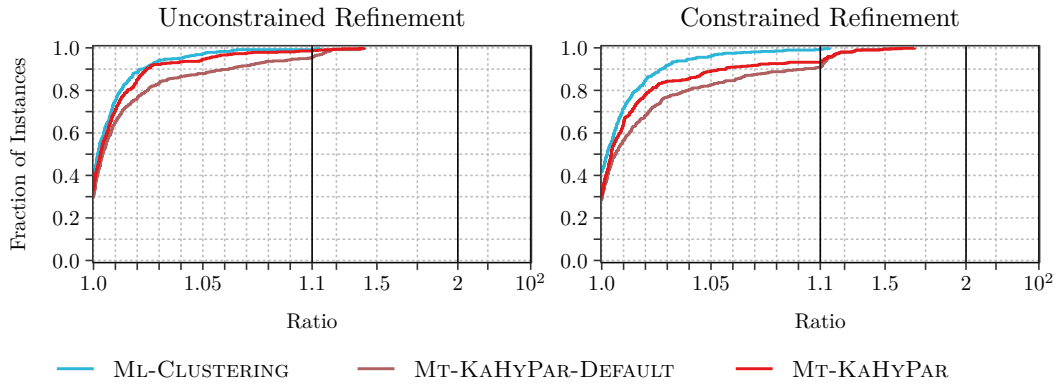


**Figure 8** Comparing the quality of Ml-Clustering and our baseline (Mt-KaHyPar) to the main version of Mt-KaHyPar (-Default) on Set A. Geometric mean running times are 0.41 s for Ml-Clustering, 0.36 s for our baseline and 0.28 s for Mt-KaHyPar-Default.