Linear-Time Multilevel Graph Partitioning via Edge Sparsification

Lars Gottesbüren
Google Research, Zürich, Switzerland

Nikolai Maas
Karlsruhe Institute of Technology, Germany

Dominik Rosch
Karlsruhe Institute of Technology, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Daniel Seemaier
Karlsruhe Institute of Technology, Germany

Abstract

The current landscape of balanced graph partitioning is divided into high-quality but expensive multilevel algorithms and cheaper approaches with linear running time, such as single-level algorithms and streaming algorithms. We demonstrate how to achieve the best of both worlds with a *linear time multilevel algorithm*. Multilevel algorithms construct a hierarchy of increasingly smaller graphs by repeatedly contracting clusters of nodes. Our approach preserves their distinct advantage, allowing refinement of the partition over multiple levels with increasing detail. At the same time, we use *edge sparsification* to guarantee geometric size reduction between the levels and thus linear running time.

We provide a proof of the linear running time as well as additional insights into the behavior of multilevel algorithms, showing that graphs with low modularity are most likely to trigger worst-case running time. We evaluate multiple approaches for edge sparsification and integrate our algorithm into the state-of-the-art multilevel partitioner KaMinPar, maintaining its excellent parallel scalability. As demonstrated in detailed experiments, this results in a 1.49× average speedup (up to 4× for some instances) with only 1% loss in solution quality. Moreover, our algorithm clearly outperforms state-of-the-art single-level and streaming approaches.

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

Keywords and phrases Graph Partitioning, Graph Algorithms, Linear Time Algorithms, Graph Sparsification

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.32

Related Version *Full Version*: `https://arxiv.org/abs/2504.17615` [30]

Supplementary Material *Software (Source Code)*: `https://github.com/KaHIP/KaMinPar/commit/73eeaa2371c6826e166c9ca1383996f14d8c7a2a` [27]
archived at `swh:1:dir:2ec38dbccb676136f71dc1796d6a06b450c48c6d`

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882500).

1 Introduction

Balanced graph partitioning aims to divide a graph into blocks of roughly equal size while minimizing the number of edges cut by the partition. As this is a crucial subtask in many applications [8, 14], it is of considerable interest to compute high-quality partitions within a minimal amount of time. Unfortunately, this goal seems unattainable from the viewpoint of

complexity theory – even approximating balanced graph partitioning to a constant factor is NP-hard [4]. Consequently, heuristic approaches are used in practice, covering a wide spectrum of options along the running time versus quality trade-off.

In the high-quality category, the most successful approaches use the multilevel framework. By repeatedly contracting clusters of nodes, multilevel algorithms first construct a hierarchy of increasingly smaller graphs in the *coarsening phase*. On the smallest graph, more expensive heuristics can be used to find a good *initial partition*. Finally, the *uncoarsening phase* undoes the contractions in reverse order while further improving the partition quality via local search algorithms (this is called *refinement*). Overall, multilevel partitioning combines a good initial solution with iterative refinement on a series of summarized graph representations with increasingly finer granularity. This has proven highly successful in practice, consistently achieving better solution quality than alternative approaches on real-world inputs [13, 26, 36]. However, due to lacking constraints on the size of the contracted representations, current multilevel implementations have superlinear running time.

On the other hand, *single-level* algorithms that use only a fixed number of passes on the input graph can run in linear time [50]. This is motivated by applications where the partitioning time is a potential bottleneck. For example, graph partitioning is used in various domains to efficiently distribute workloads across parallel machines [7, 11, 48]. This requires the graph partitioning step to be less expensive than the downstream computation. Taking this to the extreme, *streaming* approaches only consider a small part of the graph at once, assigning nodes greedily while using only a minimal representation of the partition state [17, 21, 31]. However, the running time guarantees of single-level and streaming algorithms come at the cost of inferior solution quality when compared to multilevel algorithms [6, 26, 53].

Contributions. In this work, we show that the described trade-off can be avoided by constructing a *linear time multilevel algorithm*. Our coarsening algorithm enforces that the graph shrinks by a constant factor with every successive contraction step, using edge sparsification to reduce the number of edges if necessary. We prove that this guarantees $\mathcal{O}(n + m)$ expected total work for n nodes and m edges, without any assumptions on the input graph. Our analysis provides a framework to understand the running time behavior of a broad class of existing multilevel algorithms. In addition, we demonstrate that graphs with low modularity are most likely to trigger worst-case running time behavior, while graphs with high modularity might already allow linear running time without using edge sparsification.

We integrate our approach into the KaMinPar shared-memory graph partitioner [29], preserving its excellent scaling behavior while guaranteeing linear work. For instance classes that approximate the worst case, our algorithm achieves practical speedups of up to 4× (1.49× in the geometric mean) over a baseline KaMinPar configuration – which is the fastest available shared-memory multilevel partitioner according to Ref. [26]. Despite this, the loss in partition quality is only 1% on average. Our algorithm outperforms both the single-level partitioner PuLP [50] and the state-of-the-art streaming partitioner CUTTANA [31], achieving 24% and 66% smaller average cuts, respectively, as well as a faster running time.

2 Preliminaries

Notation and Definitions. Let $G = (V, E, c, \omega)$ be an undirected graph with node weights $c : V \to \mathbb{N}_{>0}$, edge weights $\omega : E \to \mathbb{N}_{>0}$, $n := |V|$, and $m := |E|$. We extend c and ω 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 $E(v) := \{e \mid v \in e\}$ denotes the edges incident to v. We

are looking for k *blocks* of nodes $\Pi \coloneqq \{V_1, \dots, V_k\}$ that partition V, i.e., $V_1 \cup \dots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balance constraint* demands that $\forall i \in \{1, \dots, k\}$: $c(V_i) \leq L_{\max} \coloneqq (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter $\varepsilon > 0$. The objective is to minimize $\text{cut}(\Pi) \coloneqq \sum_{i<j} \omega(E_{ij})$ (weight of all cut edges), where $E_{ij} \coloneqq \{\{u,v\} \in E \mid u \in V_i, v \in V_j\}$. A *clustering* $\mathcal{C} \coloneqq \{C_1, \dots, C_b\}$ is also a partition of V, where the number of blocks b is not given in advance (there is also no balance constraint).

Multilevel Graph Partitioning. Virtually all high-quality, general-purpose graph partitioners are based on the multilevel paradigm, which consists of three phases. During coarsening, the algorithms construct a hierarchy $\mathcal{H} = \langle G =: G_1, G_2, \dots, G_\ell \rangle$ of successively coarser representations of the input graph G. Coarse graphs are built by either computing node clusterings or matchings and afterwards *contracting* them. A clustering $\mathcal{C} = \{C_1, \dots, C_b\}$ is contracted by replacing each cluster C_i with a coarse node c_i with weight $c(c_i) = c(C_i)$. For each pair of clusters C_i and C_j, there is a coarse edge $e = \{c_i, c_j\}$ with weight $\omega(e) = \omega(E_{ij})$ if $E_{ij} \neq \emptyset$, where E_{ij} is the set of all edges between clusters C_i and C_j. Once the number of coarse nodes falls below a threshold (typically, kC for some tuning constant C), *initial partitioning* computes an initial solution of the coarsest graph G_ℓ. Subsequently, contractions are undone, projecting the current solution to finer graphs and refining it. The total running time of a multilevel partitioner is the cumulative time for coarsening, initial partitioning, and refinement across all levels of the hierarchy $\mathcal{H}$.

3 Related Work

There has been a lot of research on graph partitioning, thus we refer the reader to surveys [8, 13] for a general overview and only focus on work closely related to our contributions here. As described above, modern general-purpose, high-quality graph partitioners such as Mt-Metis [38], Mt-KaHIP [3], Mt-KaHyPar [26], KaMinPar [29], and Jet [25] are mostly based on the multilevel paradigm, which constructs a hierarchy of coarser graphs during the coarsening phase.

Graph Coarsening. Early multilevel partitioners, like Chaco [32] and Metis [36], primarily employed coarsening strategies based on contracting graph matchings. While effective for mesh-like graphs due to high matching coverage (often 85-95% [35]), these strategies struggle with graphs exhibiting irregular structures, such as scale-free networks. On these graphs, small maximal matchings can result in much slower coarsening and potentially a linear number of levels. Subsequent developments addressed this limitation. Mt-Metis [39] introduced 2-hop matchings, extending small maximal matchings by further pairing nodes that have some degree of overlap in their neighborhoods until $\geq 75\%$ of nodes are contracted. This technique was subsequently also implemented by other partitioners [18, 25]. Alternative strategies focus on accelerating coarsening by grouping multiple nodes. These include methods based on cluster contraction [44, 3, 26, 29] and pseudo-matchings where nodes can match with multiple neighbors [1]. While enabling faster node reductions, they often constrain the weight of the clusters to ensure that finding a balanced initial partition is feasible. This can be problematic on graphs with highly connected hubs (e.g., the center of a star graph), potentially limiting the achievable coarsening ratio.

Graph Sparsification. Graph sparsification techniques aim to approximate a given graph with a sparser one (called *sparsifier*), typically containing substantially fewer edges while

preserving specific structural properties important for downstream tasks. This allows handling massive data sets where considering the full graph is computationally infeasible, as well as speeding up a variety of algorithms on graphs or matrices [2, 9, 22]. For graph partitioning, preserving cut properties (and thus approximately preserving the partition objective) is particularly relevant. An *ε-cut sparsifier* guarantees that every cut in the sparsifier has a weight within a $1\pm\varepsilon$ factor of the original cut. Benczúr and Karger showed that such sparsifiers with $O(n \log n/\varepsilon^2)$ edges exist for any graph and gave near-linear time constructions [9].

There are several approaches to construct sparsifiers. Spielman and Srivastava [51] introduced sparsification based on *effective resistance*, which often yields high-quality sparsifiers and preserves spectral properties closely related to cuts. However, this method can be computationally demanding. Alternatively, various heuristic sampling techniques exist, such as uniform edge sampling, k-neighbor sampling, and Forest Fire sampling [40, 42], which uses an analogy to a spreading wildfire. Chen et al. [16] provide a comparative study, suggesting that Forest Fire sampling outperforms uniform sampling for preserving cut-related properties.

KaMinPar. We integrate the techniques described in this paper into the KaMinPar [29] framework. KaMinPar is a shared-memory parallel multilevel graph partitioner. Its coarsening and uncoarsening phases are based on the *size-constrained label propagation* [44] algorithm, which is parameterized by a maximum cluster size (resp. block weight) U. In the coarsening resp. uncoarsening phase, each node is initially assigned to its own cluster resp. to its corresponding block of the partition. The algorithm then proceeds in rounds. In each round, the nodes are visited in some order. A node u is moved to the cluster resp. block K that contains the most neighbors of u without violating the size constraint U, i.e., $c(K) + c(u) \leq U$. The algorithm terminates once no nodes have been moved during a round or a maximum number of rounds has been exceeded. The coarsening further implements a 2-hop clustering strategy [29], which reduces the number of coarse nodes further whenever label propagation alone yields a node reduction factor less than 2. Since each round of size-constrained label propagation runs in linear time, and there is only a constant number of rounds, KaMinPar achieves linear time per hierarchy level for coarsening and uncoarsening.

The original paper [29] shows that KaMinPar achieves overall linear-time complexity under two key assumptions: (i) a constant node reduction factor between hierarchy levels, and (ii) bounded average degree for coarse graphs. While we will demonstrate in Theorem 1 that KaMinPar's coarsening strategy satisfies assumption (i), the inability to guarantee assumption (ii) results in a worst-case running time with an extra $log(n)$ factor.

4 Linear Time Multilevel Graph Partitioning

Multilevel algorithms construct a hierarchy $\mathcal{H} = \langle G =: G_1, G_2, \ldots, G_\ell \rangle$ of successively coarser representations of the input graph G. Each level of $\mathcal{H}$ is considered twice, during coarsening (to construct the next level) and during refinement (to improve the current partition). Assuming linear time for the coarsening and refinement on each level (see Section 3), the total sequential running time is $\Theta(\sum_{i=1}^{\ell} |V_i| + |E_i|)$. Without additional constraints on the number and size of the levels, the worst-case running time might be $\Theta(nm)$ or worse.

To obtain better guarantees, we need a geometric size reduction per level.[1] As a first step, we require that $|V_{i+1}| \leq \gamma |V_i|$ for some constant $\gamma < 1$ that is independent of G. If this is the case, the coarsened graph has constant size after a logarithmic number of steps, which

[1] In general, any series with a sum of $\mathcal{O}(1)$ works – a geometric series is, however, the most straightforward.

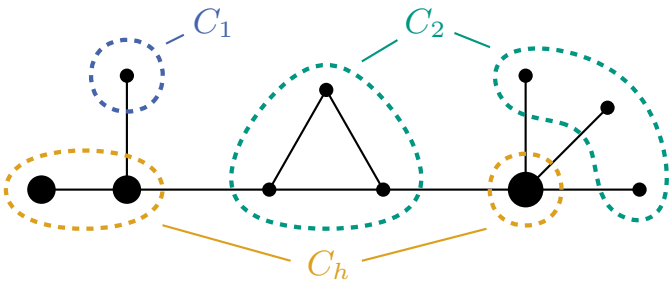

Figure 1 Illustration of Theorem 1, with examples of the different cluster types. Note that the green cluster to the right is created by 2-hop clustering.

already achieves a running time of $\mathcal{O}(n + m \log n)$. Combined with a similar guarantee for the number of edges, we get a linear total running time.

4.1 Reducing the Number of Nodes

As discussed in Section 3, the coarsening algorithms used in practice start by computing either a matching or a clustering of adjacent nodes. Typically, a maximum allowed node weight U is enforced for clusters. We use the term *size-constrained label propagation* to refer to a broad class of coarsening algorithms that form clusters of adjacent nodes and use a weight constraint. We require one essential property. In the resulting clustering, a node v never forms a singleton cluster as long as there is any adjacent cluster K with $c(K) + c(v) \leq U$.

Due to the weight constraint, size-constrained label propagation by itself is not sufficient for reducing the number of nodes (consider, e.g., a star graph). To solve this, partitioners use *2-hop clustering* as a second step, forming clusters of nodes that are not adjacent but instead have a common neighbor cluster. In the following, we provide the first formal proof that this guarantees a constant factor node reduction.

Consider a (non-isolated) node v in a singleton cluster $S = \{v\}$. Formally, we will assume that the algorithm assigns a *favorite cluster* K_S to S, out of the clusters adjacent to S. The 2-hop clustering then merges any nodes with the same favorite cluster, as long as this does not violate the weight constraint. Note that only considering favorite clusters is more restrictive than general 2-hop clustering, but is already sufficient for our purpose.

▶ **Theorem 1.** *The number of clusters obtained by size-constrained label propagation and 2-hop clustering with a maximal cluster weight $U \geq 2\frac{c(V)}{|V|}$ is at most*

$$|\mathcal{C}| \leq \frac{1}{2}|V| + \frac{c(V)}{U}$$

on any graph without isolated nodes.

Proof. We divide the set of clusters $\mathcal{C}$ into multiple subsets (see Figure 1 for an illustration). C_h is the set of *heavy* clusters with weight larger than $\frac{1}{2}U$. C_1 is the set of singleton clusters with weight at most $\frac{1}{2}U$ and C_2 is the set of clusters with multiple nodes and weight at most $\frac{1}{2}U$. Note that $\mathcal{C} = C_h \cup C_1 \cup C_2$. Let $r := \frac{1}{|C_2|}\sum_{K \in C_2} |K|$ be the average number of nodes for clusters in C_2. In combination, this results in the inequality $|V| \geq |C_h| + |C_1| + r|C_2|$.

Each singleton cluster $S \in C_1$ is only adjacent to clusters with weight larger than $U - c(S)$, and thus only clusters in C_h – otherwise, the node would have joined the lighter adjacent cluster. Consider the favorite cluster $K_S \in C_h$ of S. Due to 2-hop coarsening, there is no other cluster in C_1 with the same favorite (otherwise, 2-hop clustering would have joined them).

Algorithm 1 Graph Coarsening with Sparsification.

```
1 i ← 1, G_i ← G                                              // Input: graph G
2 while G_i not small enough do
3 |   G'_{i+1} ← Coarsen(G_i)
4 |   m̂ ← min{τ_e · |E(G_i)|, τ_d · (|E(G_i)|/|V(G_i)|) · |V(G'_{i+1})|}
5 |   if |E(G'_{i+1})| > ρ · m̂ then G_{i+1} ← Sparsify(G'_{i+1}, m̂)
6 |   else G_{i+1} ← G'_{i+1}
7 |   i += 1
8 return ⟨G_1, ..., G_i⟩                                      // Output: hierarchy H = ⟨G_1, ..., G_i⟩
```

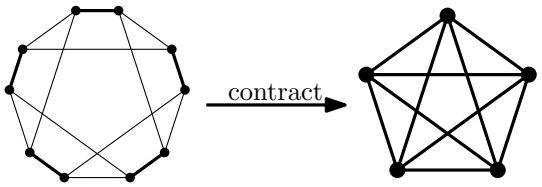

Figure 2 Contracting the bolded edges leads to increased density on the coarse graph.

Consequently, $|C_1| \leq |C_h|$. Moreover, $c(S)+c(K_S) > U$ gives, when summed over all clusters and combined with the definition of C_h, the inequality $c(V) \geq \sum_{K\in C_h} c(K)+\sum_{K\in C_1} c(K) = \sum_{K\in C_1}(c(K)+c(K_S)) + \sum_{K\in C_h\setminus\{K'_S|K'\in C_1\}} c(K) > U|C_1| + \frac{1}{2}U(|C_h|-|C_1|)$. Rearranged, this is $|C_h|+|C_1| \leq 2\frac{c(V)}{U}$.

Combining all inequalities, we get

$$\begin{aligned}
|\mathcal{C}| &= |C_h| + |C_1| + |C_2| \\
&\leq \frac{1}{r}|V| + (1-\frac{1}{r})|C_h| + (1-\frac{1}{r})|C_1| \\
&\leq \frac{1}{r}|V| + 2(1-\frac{1}{r})\frac{c(V)}{U} \\
&\leq \frac{1}{2}|V| + \frac{c(V)}{U}
\end{aligned}$$

For the final step, we use the observation that $xa+(1-x)b \leq \frac{1}{2}a+\frac{1}{2}b$ for $b \leq a$ and $x \leq \frac{1}{2}$. Since $r \geq 2$ and $U \geq 2\frac{c(V)}{|V|}$, we can apply this with $x = \frac{1}{r}$, $a = |V|$ and $b = 2\frac{c(V)}{U}$. ◀

Isolated nodes (i.e., nodes without a neighbor) are a special case as standard clustering algorithms do not handle them. Therefore, they are omitted from Theorem 1. However, it is trivial to either remove isolated nodes and reinsert them in the uncoarsening, or alternatively cluster them with each other (we do the latter).

Note that the precondition $U \geq 2\frac{c(V)}{|V|}$ is no limitation for the applicability of Theorem 1. In practice, much larger values are used for U (in our case $U = \frac{c(V)}{160k}$, see Section 4.3). However, the theorem does not include clustering approaches which limit the *number* of nodes in a cluster (e.g., allowing only matchings). We note that in this case similar, but weaker, bounds can be obtained with an analogous line of reasoning.

4.2 Reducing the Number of Edges via Sparsification

As discussed above, geometric node reduction in coarsening strategies can still lead to superlinear total running time due to increasing graph density at coarser levels (e.g., Figure 2). Achieving linear time necessitates preventing this phenomenon. Since common clustering algorithms can not guarantee a geometric reduction in the number of edges, we propose an alternative strategy: sparsifying the contracted graph when the edge count does not shrink at a sufficient rate. To validate this concern, we first show that the issue can arise on important classes of graphs using the example of Erdős-Rényi graphs. Subsequently, we will detail our sparsification approach.

Consider the coarsening hierarchy of a sparse Erdős-Rényi graph $G_0 = G(n_0, c/n_0)$ with n_0 nodes and edge probability $p_0 := c/n_0$ for some constant c. Assume that coarsening halves the number of nodes at each level by contracting pairs of nodes, and that coarse graphs also behave like Erdős-Rényi graphs. In other words, $G_i = G(n_i, p_i)$ with $n_i = n_{i-1}/2$ and $p_i \approx 1 - (1 - p_{i-1})^4$ for $i > 0$ (there is an edge between two coarse nodes if any of the four potential edges between the corresponding nodes in G_{i-1} existed). Note that $n_i = n_0/2^i$ and $p_i = 1 - (1 - p_0)^{4^i} = 1 - (1 - c/n_0)^{4^i} \approx 1 - e^{-4^i \cdot c/n_0}$. Let $i = \alpha \log(n_0)$, then $p_i \approx 1 - e^{-cn_0^{2\alpha - 1}} \xrightarrow{n_0 \to \infty} 0$ for $\alpha < \frac{1}{2}$, i.e., there are $\Theta(\log n_0)$ sparse levels. On these,

$$\frac{\mathbb{E}[m_{i+1}]}{\mathbb{E}[m_i]} = \frac{1 - (1 - p_i)^4}{p_i} \frac{n_i(n_i - 2)/8}{n_i(n_i - 1)/2} \xrightarrow{n_0 \to \infty} \frac{1 - (1 - p_i)^4}{4p_i} \approx \frac{1 - (1 - 4p_i)}{4p_i} = 1,$$

since $n_i = n_0/2^i \geq n_0/2^{\alpha \log n_0} > \sqrt{n_0} \to \infty$ and $(1 - p_i)^4 \approx 1 - 4p_i$ for small p_i. Thus, the number of coarse edges remains relatively constant, leading to overall $\mathcal{O}(m_0 \log(n_0))$ time.

To achieve linear time, we therefore limit the number of edges through sparsification as outlined in Algorithm 1. Let $G'_{i+1} = (V_{i+1}, E'_{i+1})$ denote the current graph before sparsification, obtained by contracting the previous graph G_i (line 1). We obtain $G_{i+1} = (V_{i+1}, E_{i+1})$ by sparsifying the edges of G'_{i+1} so that the size of E_{i+1} is bounded by a threshold $\hat{m}$, defined as

$$\hat{m} := \min\{\tau_e \cdot |E_i|, \tau_d \cdot \frac{|E_i|}{|V_i|} \cdot |V_{i+1}|\}.$$

Here, τ_e is the *edge threshold* parameter, limiting the coarse edge count relative to the current graph's edge count, and τ_d is the *density threshold* parameter, likewise limiting the average degree of the coarse graph. Since sparsification itself introduces computational overhead, we only apply it if the potential edge reduction is significant. Specifically, we trigger sparsification only if $|E'_{i+1}| > \hat{m}$ *and* the target edge count $\hat{m}$ represents a substantial reduction from the current edge count $|E'_{i+1}|$, quantified by the condition $|E'_{i+1}|/\hat{m} \geq \rho$, where $\rho \geq 1$ is a tunable constant (line 1). Once triggered, we use one of the following sampling algorithms to reduce the edge count to $\hat{m}$ (in expectation), before adding the sparsified graph to the hierarchy (line 1). Since our goal is to achieve overall linear time, we only consider linear time sparsification algorithms. Further, sparsification must be fast in practice for speedups to be attainable.

Uniform Sampling: UR. As a simple baseline, we consider uniform random sampling. Each edge $e \in E'_{i+1}$ is selected independently with probability $p := \hat{m}/|E'_{i+1}|$, resulting in expected $\hat{m}$ edges. Note that this approach is oblivious to edge weights – although heavier edges have larger influence on the partitioning objective and are thus likely more important.

Algorithm 2 Weighted Forest Fire: graph $G = (V, E)$, burn ratio ν, probability p. The only difference to the original Forest Fire [41] algorithm is highlighted blue.

```
1  S ← new Array() of size |E|                                   // Scores
2  b ← 0                                                         // Number of burnt edges
3  while b ≤ ν|E| do in parallel
4      Q ← new FIFO(); Q.push(random node from V)                 // BFS queue
5      T ← new Set()                                             // Visited nodes
6      while Q ≠ ∅ do
7          u ← Q.pop()
8          while N(u) \ T ≠ ∅ do
9              Sample v from N(u) \ T with prob. ω({u,v}) / Σ_{v∈N(u)\T} ω({u,v})
10             T.insert(v); Q.push(v)
11             S[{u,v}] atomic+= 1; b atomic+= 1
12             Break with prob. p
13 return S
```

Weighted Threshold Sampling: T-Weight. To incorporate edge weights, we consider a weighted threshold sampling strategy. First, we identify the weight threshold $\omega_t := \omega(e_t)$ corresponding to the $\hat{m}$-th heaviest edge e_t in G'_{i+1}. This can be done in expected time $\mathcal{O}(|E'_{i+1}|)$ using the quickselect algorithm. Based on ω_t, we partition E'_{i+1} into three disjoint sets $E'^{<}_{i+1}$, $E'^{=}_{i+1}$, and $E'^{>}_{i+1}$, for coarse edges with weight smaller than, equal to, or larger than ω_t. Edges in $E'^{<}_{i+1}$ are discarded, while edges in $E'^{>}_{i+1}$ are kept. To reach the target size $\hat{m}$, we further sample edges from $E'^{=}_{i+1}$ uniformly with probability $p := \frac{\hat{m}-|E'^{>}_{i+1}|}{|E'^{=}_{i+1}|}$.

(Weighted) Forest Fire Sampling: T-(W)FF. We further include a variation of threshold sampling that uses *Forest Fire* [42] scores rather than edge weights, as this performed well as a cut-preserving sparsifier in Ref. [16]. We include a brief description for self-containment and extend the algorithm to take edge weights into consideration (Algorithm 2). The algorithm computes edge scores by simulating fires spreading through the graph via multiple traversals starting from random nodes. When visiting a node u, the number of neighbors X to be visited is drawn from a geometric distribution paramterized by a tunable parameter p. The standard forest fire algorithm subsequently samples X distinct nodes (without replacement) from u's unvisited neighbors. We incorporate edge weights by making this sampling weight-dependent: the probability of selecting neighbor v of u is proportional to the edge weight $\omega(\{u, v\})$ relative to the total edge weight between u and its unvisited neighbors (line 9). Note that this modification (marked blue in Algorithm 2) is the only difference to Ref. [41]. Each edge traversal during this process increments the *burn* score of the edge (line 11). The algorithm stops scheduling fires once the cumulative burn score `b` exceeds $\nu|E|$ (line 3) for some *burn ratio* $\nu > 0$. After computing the edge importance scores, we use the weighted threshold sampling strategy to sparsify the graph (using the importance scores instead of edge weights).

4.3 Putting it Together

Based on the discussed insights, we propose a linear time multilevel algorithm that builds upon KaMinPar [29]. We leverage the existing clustering and refinement algorithms

available in KaMinPar, whose running time is linear in the size of the current hierarchy level (see Section 3). We introduce two necessary changes to achieve linear time for the overall algorithm. Most importantly, we introduce edge sparsification as discussed in Section 4.2, ensuring the number of edges shrinks geometrically. In addition, we replace the coarsening and initial partitioning used by the default configuration of KaMinPar with a more traditional approach. This is because the default configuration is amenable to scenarios where expensive bipartitioning happens on a relatively large graph, adding a $\Omega(n \log n)$ term to the running time in the worst-case. We refer to our technical report [30] for details.

Instead, we use size-constrained label propagation with subsequent recursive bipartitioning, following other state-of-the-art multilevel algorithms [3, 28, 44]. Similar to Mt-KaHyPar [26], the cluster weight limit is $U = \frac{c(V)}{160k}$ and we limit the node reduction per coarsening step to at most 2.5×. As this is combined with 2-hop clustering, Theorem 1 guarantees a geometric shrink factor until a size of $|V_i| = 320k$ is reached. The coarsening terminates at $160k$ nodes or if the current shrink factor is too small, thereby resulting in a graph with size $\mathcal{O}(k)$.[2] The recursive bipartitioning then requires total time $\mathcal{O}(k \log k)$, which is linear under the extremely weak assumption that $k \log k \in \mathcal{O}(n + m)$.

So far, we have argued from a sequential point of view. In the parallel setting, the consequence is that our algorithm needs only linear work. With regards to scalability, the sparsification algorithms described in Section 4.2 lend themselves to a rather straightforward parallelization. Combined with the excellent scalability of the coarsening and refinement algorithms of KaMinPar [29] and the fact that initial partitioning is insignificant for the total running time, we maintain the scalability of default KaMinPar while reducing the required work.

5 Quantifying Worst-Case Instances

As discussed, we need edge sparsification to achieve linear time if coarsening does not shrink the number of edges geometrically. However, it would be useful to understand for which graphs sparsification is required and for which it is not – both for theoretical insights into the structure of worst-case instances and to allow empirical estimates. Given a clustering C of a graph G, we are thus interested in the number of edges of G', where G' is the graph created by contracting C. If $|E(G')| \approx |E(G)|$ for clusterings computed by the coarsening algorithm, sparsification is required to further reduce the number of edges.

Intuitively, this is the case for graphs with low locality – edges might lead anywhere and are thus hard to contain in small clusters. For many random graph models, this is rather easy to decide. For example, sparse Erdős-Rényi graphs are highly non-local, thus necessitating sparsification (see Section 4.2). On the other hand, for random geometric graphs (i.e., unit disk graphs) coarsening algorithms reduce the number of edges very efficiently. However, to classify real-world instances or more complex graph models, a general criterion is needed.

Classification via Modularity. We propose that the *modularity* of a graph allows to estimate whether sparsification is necessary.[3] Modularity was introduced by Newman and Girvan to evaluate the quality of a clustering with regards to community structure [46], and modularity based community detection algorithms are used in many applications [33, 54, 55]. Given

[2] Note that sparsification ensures $\mathcal{O}(k)$ edges – although this is not necessary for linear time.
[3] There are also multiple other locality metrics, but these are less useful. For example, the clustering coefficient is based on the number of triangles. However, this does not result in any useful bounds.

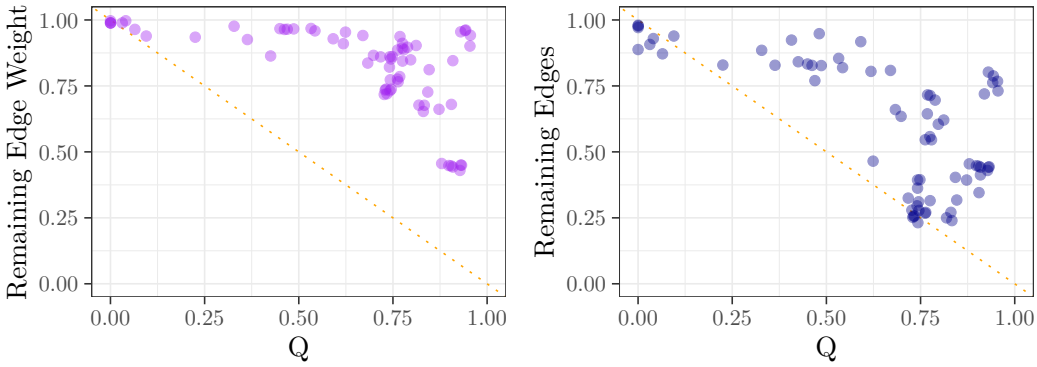

Figure 3 Remaining total edge weight (**left**) and number of edges (**right**) after one coarsening step, compared to the modularity of the graph. The y-values are denoted as a fraction of the initial value. Based on Lemma 2, we expect most points (i.e., graphs) to be in the upper right half.

a clustering $C = \{V_1, \ldots, V_{|C|}\}$, let $e_{ij} := \frac{1}{2|E|}|E(V_i, V_j)|$ denote the fraction of edges that connect cluster i and cluster j (only counted in one direction). Further, let $a_i := \sum_j e_{ij}$ be the fraction of edges with one endpoint in cluster i. The modularity of the clustering is then defined as $Q_C := \sum_i \left(e_{ii} - a_i^2\right)$, where $Q_C \in [-\frac{1}{2}, 1]$. The modularity $Q \in [0, 1]$ of the graph itself is the maximum modularity of all possible clusterings. As demonstrated in the following, modularity is a good fit for our purpose.

▶ **Lemma 2.** *For a given clustering, the total fraction of edges that connect nodes within the same cluster is bounded by*

$$Q_C \leq \sum_i e_{ii} \leq Q_C + \alpha_C$$

where $\alpha := \max_i a_i$ is the maximum fraction of edges with endpoints in the same cluster.

Proof. Since $Q_C = \sum_i \left(e_{ii} - a_i^2\right)$, the left side of the inequality follows immediately. Further, $\sum_i e_{ii} = Q_C + \sum_i a_i^2 \leq Q_C + \sum_i (\max_j a_j) a_i = Q_C + \max_i a_i$. Note that $\sum_i a_i = 1$ since each edge is connected to exactly two clusters. ◀

Lemma 2 provides a lower bound of $1 - Q_C - \alpha_C$ for the fraction of edges connecting different clusters. Unfortunately, this does not correspond directly to the edges of G' – if multiple edges connect the same pair of clusters (we say that the edges are *parallel*), they are combined into a single edge, thereby further reducing the number of remaining edges. The actual bound is thus $1 - Q_C - \alpha_C - p_C \leq \frac{|E(G')|}{|E(G)|}$, where p_C is the number of parallel edges.

Let us consider the case where the clusters are small, such as during the first steps of multilevel coarsening. Then, $1 - Q$ is an approximate lower bound for the number of edges. If clusters are small, any cluster is only incident to a small fraction of all edges and edges are unlikely to be parallel, i.e., both α_C and p_C are small. Moreover, Q_C is almost certainly smaller than Q as achieving maximum modularity often necessitates large clusters [23]. Consequently, we expect that $1 - Q \lesssim 1 - Q_C - \alpha_C - p_C \leq \frac{|E(G')|}{|E(G)|}$ if clusters are small, making $1 - Q$ an accurate bound.

Empirical Effect of Modularity. To verify whether $1 - Q$ is a useful bound in practice, we provide an empirical evaluation on a set of 71 large graphs which is used in multiple recent

works on graph partitioning [37, 43, 49]. Figure 3 shows computed modularity scores of the graphs in relation to the fraction of edges that remains after one step of our multilevel coarsening algorithm (see Section 4.1). Since computing the modularity of a graph is itself NP-hard [12], we calculate approximate scores with the well-known Louvain algorithm [10], using the implementation from NetworKit [5, 52].

With regards to the total edge weight of G' (left, corresponds to inter-cluster edges in G), $1 - Q$ is almost a strict lower bound. For most graphs, the fraction of remaining weight is much larger than $1 - Q$. The actual number of edges after combining parallel edges (right) is often significantly smaller than the weight. However, $1 - Q$ is still a mostly accurate bound. Therefore, modularity is indeed useful to predict the coarsening behavior of multilevel algorithms – graphs with low modularity are likely to be worst-case instances.

6 Experiments

Setup. We implemented the described sparsification algorithms within the KaMinPar [29] framework and compiled it using `gcc` 14.2.0 with flags `-O3 -mtune=native -march=native`. The code is parallelized using TBB [47]. All experiments are performed on a machine equipped with two 32-core Intel Xeon Gold 6530 processors (2.1 GHz) and 3 TB RAM running Rocky Linux 9.5. We only use one of the two processors (i.e., 32 cores) to avoid NUMA effects.

Competitors. In Section 6.3, we compare our algorithm against PuLP [50] (v1.1) and Cuttana [31] (commit `ed0c182` in the official GitHub repository[4]). PuLP is a linear time single-level partitioner which focuses on shared-memory scalability and low memory usage. PuLP starts by growing (unbalanced) initial clusters from a random assignment and then performs multiple iterations of alternating balancing and cut minimization. Each phase is implemented with a fixed number of label propagation rounds. The authors showed that PuLP achieves considerable speedups in comparison to multiple well-known multilevel algorithms. Cuttana is a streaming partitioner which improves upon the solution quality of previous streaming approaches with a node buffering technique. Instead of assigning each node greedily, it uses a buffer of fixed size to delay assignment until more information on the neighborhood is available or the buffer overflows. In addition, Cuttana uses a more fine-grained subpartition which allows to further refine the solution quality after the initial assignment. We use the default settings for PuLP and configure Cuttana following the parameters described by the authors, i.e., $\frac{\mathcal{K}'}{\mathcal{K}} = 4\,096$, $D_{\max} = 1\,000$ and max_qsize $= 10^6$.

Instances. We focus on graphs for which coarsening increases edge density substantially. This happens on 17 out of 71 graphs of a benchmark set used in previous works on graph partitioning [43, 37, 49] (mostly real-world k-mer and social graphs, and graphs deduced from text recompression [34]). We only use these graphs since sparsification is not triggered on the remaining instances, thus leaving the algorithm unchanged. Note that there is no running time overhead in this case. We further include 6 social graphs from the Sparse Matrix Collection [19] and generate random graphs: Erdős-Rényi graphs (using KaGen [24]), as well as Chung-Lu [45], Planted Partition, and R-MAT [15] graphs (using NetworKit [5]). These graphs are inherently non-local, thus especially challenging for linear-time partitioning. Overall, the benchmark set comprises 39 graphs (see Ref. [30]) with 511 K to 1.8 G undirected edges. The graphs deduced from text recompression feature node weights. All other graphs are

[4] `https://github.com/cuttana/cuttana-partitioner`

unweighted. Tuning experiments are performed on a subset containing 8 randomly drawn graphs spanning different types.

Methodology. We consider an *instance* as the combination of a graph and a number of blocks k. We set the imbalance tolerance to $\varepsilon = 3\%$, use $k \in \{3, 7, 8, 16, 37, 64\}$ and perform 5 repetitions for each instance using different seeds. Results (running time, edge cut) are averaged arithmetically per instance over these repetitions. When aggregating across multiple instances, we use the geometric mean to ensure that each instance has equal influence.

Performance Profiles. To compare the edge cuts of different algorithms, we use *performance profiles* [20]. Let $\mathcal{A}$ be the set of all algorithms we want to compare, $\mathcal{I}$ the set of instances, and $\text{cut}_A(I)$ the edge cut of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm A, we plot the fraction of instances

$$\mathcal{P}_A(\tau) \coloneqq \frac{|\{I \in \mathcal{I} \ : \ \text{cut}_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} \text{cut}_{A'}(I)\}|}{|\mathcal{I}|}$$

on the y-axis and τ on the x-axis. Achieving higher fractions at lower τ-values is considered better. In particular, $\mathcal{P}_A(1)$ denotes the fraction of instances for which algorithm A performs best, while $\mathcal{P}_A(\tau)$ for $\tau > 1$ illustrates the robustness of the algorithm. For example, an algorithm A with $\mathcal{P}_A(1) = 0.49$ but $\mathcal{P}_A(1.01) = 1.0$ (i.e., never more than 1% worse than the best) might be preferable to an algorithm B with $\mathcal{P}_B(1) = 0.51$ that only achieves $\mathcal{P}_B(\tau) = 1.0$ at much larger τ (indicating much worse partitions on some inputs).

6.1 Parameter Study

We begin our evaluation by tuning the parameters introduced in Section 4.2. Recall that these are the edge and density thresholds τ_e and τ_d, which control the number of coarse edges, and the minimum reduction factor ρ, which controls whether sparsification is triggered on a given hierarchy level. We use the tuning benchmark subset and $k = 16$ for this experiment to limit computational costs.

The results are shown in Figure 4, where we plot geometric mean edge cuts and running times relative to the KaMinPar baseline without sparsification for $\tau_e \in \{1/4, 1/2, 1\}$, $\tau_d \in \{1/2, 1, 2\}$ and $\rho \in \{1, 2, 3, 4, 8\}$. We observe similar speedups of up to 1.63× for weighted threshold sampling (T-Weight) and uniform sampling (UR). T-Weight achieves the highest speedup (1.63×) at $\tau_e = 1/2$, $\tau_d = 1/2$ and $\rho = 3$, while UR achieves the same speedup at slightly different parameters ($\tau_e = 1/4$, $\tau_d = 1$ and $\rho = 2$). With these parameters, edge cuts increase by 12.6% and 27.6% for T-Weight and UR, respectively. Surprisingly, more aggressive sparsification (i.e., smaller τ_e, τ_d and ρ) does not achieve larger speedups. This is likely due to several factors. First, the sparsification process itself introduces computational overhead which can counteract potential speedups, particularly when the size reduction is modest. Second, excessive sparsification degrades partition quality considerably, thereby increasing the workload required for the refinement algorithm to converge to a local optimum. Hence, moderate sparsification seems favorable.

Larger ρ seems beneficial for maintaining partition quality. At $\rho = 4$ (i.e., only sparsify if reducing the number of edges by a factor of ≥ 4), both T-Weight and UR show similar speedups as with smaller ρ and partition quality close to the baseline. We therefore pick $\tau_e = \tau_d = 1/2$ and $\rho = 4$ for subsequent experiments, where T-Weight and UR achieve speedups of 1.56× and 1.59×, while increasing edge cuts by 0.9% and 5.3%, respectively.

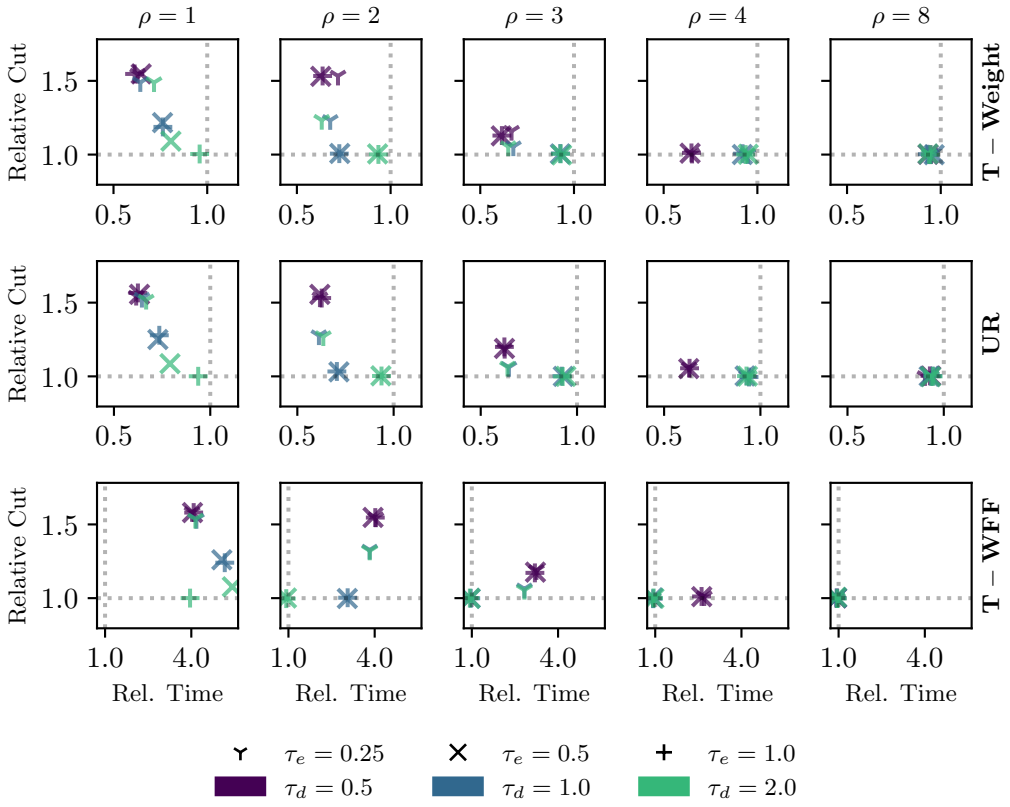

Figure 4 Relative cut and running time of KaMinPar with weighted threshold sampling (T-Weight), uniform sampling (UR), or threshold sampling via Weighted Forest Fire scores (T-WFF) versus baseline (KaMinPar without sparsification) on the tuning benchmark set with $k = 16$.

Lastly, we look at threshold sampling using Weighted Forest Fire (T-WFF) scores. For T-WFF itself, we use $p = 0.6$ and $\nu = 0.5$, since these parameters performed best during preliminary experimentation. We observe the fastest running times using parameters that do not trigger sparsification, suggesting that T-WFF does not provide practical speedups. At $\tau_e = \tau_d = 1/2$ and $\rho = 4$, T-WFF is 2.62× slower while incurring a 1.0% increase in cut size.

6.2 Effects of Sparsification

Next, we evaluate the proposed sparsification techniques on the full benchmark set. As can be seen in Figure 5, T-Weight (geometric mean running time 1.43 s) and UR (1.40 s) achieve similar speedups of 1.49× resp. 1.52× over the baseline (no sparsification, 2.13 s). T-Weight achieves considerably better partition quality (increase in average edge cut by 1.5%) than UR (increase by 5.5%). T-WFF outperforms T-FF, but is not competitive: its partition quality is slightly worse (increase by 3.9%) while much slower (7.04 s). We thus focus on T-Weight.

Looking at Figure 6, we can see that sparsification reduces the number of edges on coarse graphs considerably. Without sparsification, the graphs on the first hierarchy levels (i.e., after the first coarsening step) contain, on average, 75% of the edges of the input graphs, but only 39% of the nodes. With sparsification, the average edge count reduces to 28%.

As shown in Figure 7 (left), the speedup from T-Weight sparsification varies considerably

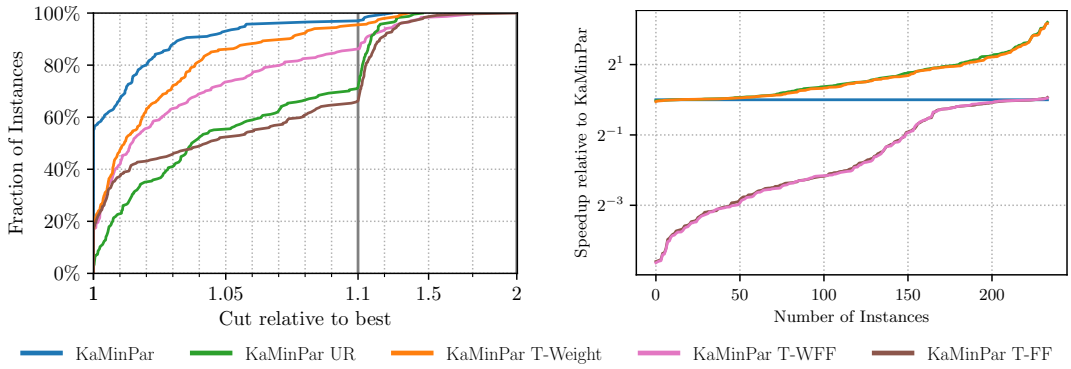

Figure 5 Partition quality as *performance profile* (**left**) and speedup over baseline (no sparsification, **right**) of sparsification algorithms: weighted threshold sampling (T-Weight), uniform sampling (UR), and threshold sampling via (Weighted) Forest Fire scores (T-(W)FF).

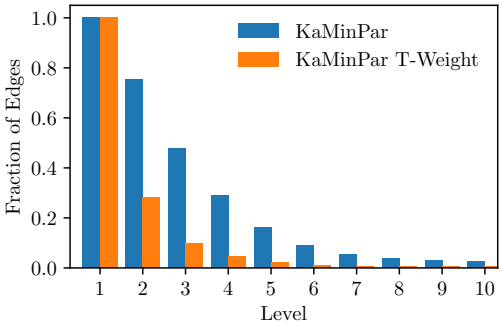

Figure 6 Relative geometric mean number of edges per hierarchy level (levels 1-10), comparing no sparsification against T-Weight (weighted threshold sampling) sparsification. Edge counts are relative to the input graphs. The final value is propagated for hierarchies shorter than 10 levels.

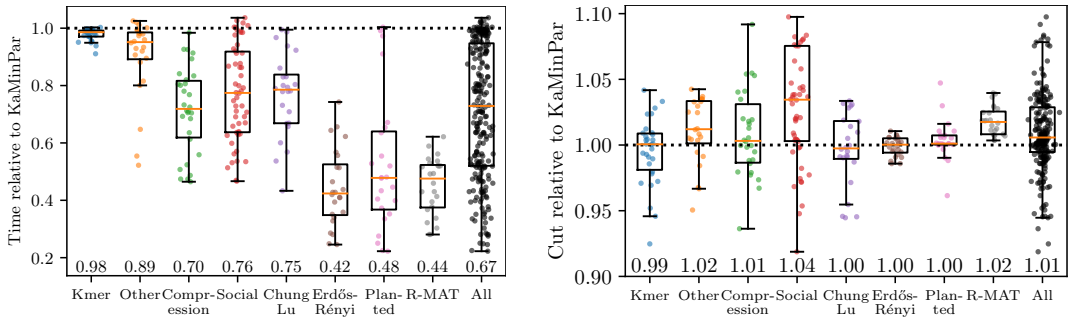

Figure 7 Comparison of KaMinPar with T-Weight sparsification relative to the baseline without sparsification grouped by graph class (lower is faster resp. higher-quality). **Left:** Relative running times with the geometric mean relative time annotated per class. **Right:** Relative cuts with the geometric mean relative cut annotated per class.

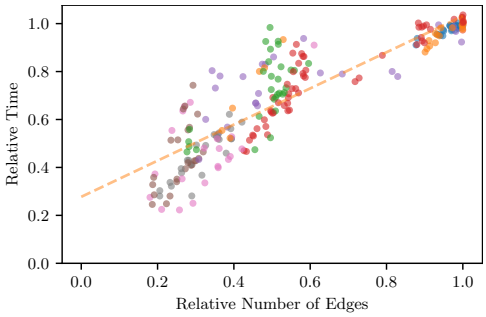

Figure 8 Relationship between running time of KaMinPar T-Weight relative to KaMinPar without sparsification and the hierarchy size ratio (number of total edges across all hierarchy levels after sparsification relative to no sparsification). Note the strong correlation (coefficient ≈ 0.893).

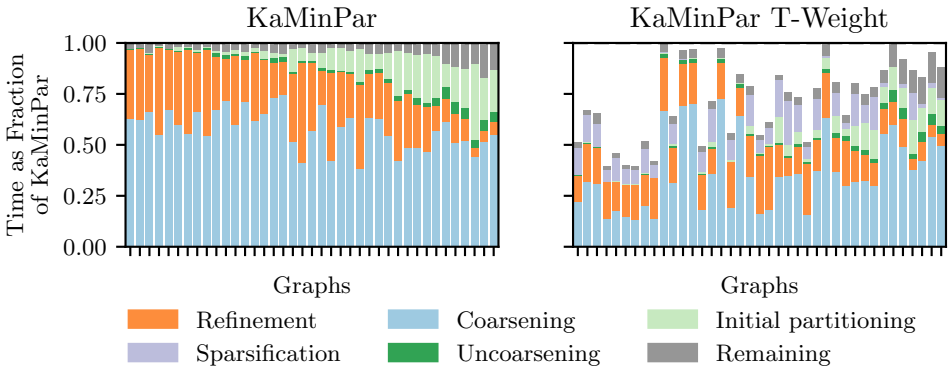

Figure 9 Relative running time attribution for KaMinPar without sparsification (**left**) and with T-Weight sparsification (**right**) using $k = 16$. Graphs are sorted by the total running time of KaMinPar without sparsification in descending order.

with graph class, with moderate cut size increases on real-world grahs (Figure 7, right). Non-local graphs with extremely low modularity (Erdős-Rényi, Planted Partition, R-MAT) show substantial gains (average speedup $> 2\times$, up to $4\times$), while real-world text recompression ($\approx 1.43\times$) and social graphs ($\approx 1.32\times$) exhibit moderate speedups. K-mer graphs see negligible benefit. This variation correlates strongly with the reduction in graph hierarchy size (number of edges across all hierarchy levels): instances with greater reduction achieve faster relative running times (Figure 8). The observed speedups stem primarily from reduced time in the coarsening and refinement phases, see Figure 9. Without sparsification, these phases consume on average 55% (1.17 s) and 23% (0.48 s) of the total partitioning time (2.13 s), respectively. Sparsification reduces these to 0.65 s and 0.32 s, respectively. This improvement comes at low cost, as the sparsification step itself averages only 0.10 s out of 1.62 s when triggered (94% of the instances).

6.3 Comparison against Competing Partitioners

Finally, we compare KaMinPar with T-Weight sparsification against alternative linear-time partitioners: single-level PuLP [50] and streaming Cuttana [31]. We found that Cuttana is rather slow and does not support node weights. Thus, we limit our benchmark set to unweighted graphs with $m \leq 2^{26}$ edges (18 out of 39 graphs).

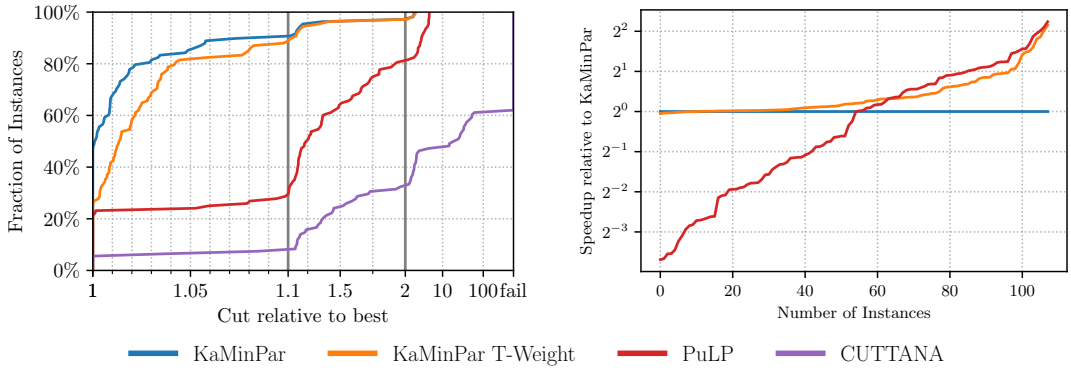

Figure 10 Partition quality (**left**) and relative running time (**right**) on the reduced benchmark set and all k values for KaMinPar without and with T-Weight sparsification, PuLP and Cuttana. Speedups are plotted relative to KaMinPar without sparsification. Cuttana is omitted from the speedup plot since it is $\geq 72\times$ slower than all other algorithms.

As shown in Figure 10 (left), KaMinPar with T-Weight sparsification computes considerably better partitions with average cuts 30% and 66% smaller than those of PuLP and Cuttana, respectively. Compared to KaMinPar *without* sparsification, edge cuts are slightly larger (cutting 1% more edges on average), but are still within a factor of 1.10 to the best cut found on 88% of all instances (vs. 90% for non-sparsifying KaMinPar). In contrast, PuLP and Cuttana compute edge cuts within factors 1.26 and 1.93 to the best cuts found on only *half of the instances*, respectively. PuLP computes the best partitions for 21% of the instances, predominantly Erdős-Rényi and Planted Partition graphs. Cuttana crashes on 39% of the instances (we exclude these instances in pairwise aggregates).

Through sparsification, the geometric mean running time of KaMinPar reduces from 0.48 s to 0.37 s. PuLP (0.63 s) is slower than non-sparsifying KaMinPar on average, but proves faster on 53 (resp. 48 vs. sparsifying KaMinPar) and twice as fast on 23 (resp. 2) out of 108 instances. Cuttana is $72\times$ slower than PuLP (and thus the other algorithms), although we note that comparing running times fairly is difficult since Cuttana interleaves computation with graph I/O from SSD (I/O times are excluded for the other algorithms).

7 Conclusion

Current graph partitioning algorithms can be classified into high-quality but superlinear multilevel algorithms, and cheaper linear time approaches such as single-level partitioning and streaming partitioning. We demonstrate both in theory and in practice that it is possible to achieve the best of both worlds at once. Our linear time multilevel algorithm uses edge sparsification to constrain the size of subsequent coarser levels, which provably guarantees linear work while maintaining scalability to many cores. We minimize quality loss by choosing appropriate thresholds for triggering the sparsification step and, if triggered, removing the edges with lowest weight. As a result, our multilevel algorithm is faster than state-of-the-art single-level and streaming approaches while consistently computing better solutions – making multilevel the preferable choice even if extremely short running time is required.

References

1 Amine Abou-Rjeili and George Karypis. Multilevel Algorithms for Partitioning Power-Law Graphs. In *20th International Conference on Parallel and Distributed Processing (IPDPS)*. IEEE Computer Society, 2006. doi:10.1109/IPDPS.2006.1639360.

2 Nesreen K. Ahmed, Jennifer Neville, and Ramana Rao Kompella. Network Sampling: From Static to Streaming Graphs. *ACM Transactions on Knowledge Discovery from Dat*, 8(2):7:1–7:56, 2013. doi:10.1145/2601438.

3 Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-Quality Shared-Memory Graph Partitioning. In *24th European Conference on Parallel Processing (Euro-Par)*, pages 659–671. Springer, August 2018. doi:10.1007/978-3-319-96983-1_47.

4 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.

5 Eugenio Angriman, Alexander van der Grinten, Michael Hamann, Henning Meyerhenke, and Manuel Penschuck. Algorithms for Large-Scale Network Analysis and the NetworKit Toolkit. In *Algorithms for Big Data: DFG Priority Program 1736*, pages 3–20. Springer, 2023. doi:10.1007/978-3-031-21534-6_1.

6 Amel Awadelkarim and Johan Ugander. Prioritized Restreaming Algorithms for Balanced Graph Partitioning. In *26th Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1877–1887. ACM, 2020. doi:10.1145/3394486.3403239.

7 Cevdet Aykanat, Berkant Barla Cambazoglu, Ferit Findik, and Tahsin M. Kurç. Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids. *Journal of Parallel and Distributed Computing*, 67(1):77–99, 2007. doi:10.1016/J.JPDC.2006.05.005.

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

9 András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *28th Symposium on Theory of Computing (STOC)*, pages 47–55. ACM, 1996. doi:10.1145/237814.237827.

10 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.

11 Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Scalable Matrix Computations on Large Scale-Free Graphs Using 2D Graph Partitioning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 50:1–50:12. ACM, 2013. doi:10.1145/2503210.2503293.

12 Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing Modularity is Hard, 2006. doi:10.48550/arXiv.physics/0608255.

13 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.

14 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.

15 Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *4th International Conference on Data Mining (ICDM)*, pages 442–446. SIAM, 2004. doi:10.1137/1.9781611972740.43.

16 Yuhan Chen, Haojie Ye, Sanketh Vedula, Alex M. Bronstein, Ronald G. Dreslinski, Trevor N. Mudge, and Nishil Talati. Demystifying Graph Sparsification Algorithms in Graph Properties

Preservation. *Proceedings of the VLDB Endowment*, 17(3):427–440, 2023. doi:10.14778/3632093.3632106.

17 Adil Chhabra, Florian Kurpicz, Christian Schulz, Dominik Schweisgut, and Daniel Seemaier. Partitioning Trillion Edge Graphs on Edge Devices, 2024. doi:10.48550/arXiv.2410.07732.

18 Timothy A. Davis, William W. Hager, Scott P. Kolodziej, and S. Nuri Yeralan. Algorithm 1003: Mongoose, a Graph Coarsening and Partitioning Library. *ACM Transactions on Mathematical Software*, 46(1), 2020. doi:10.1145/3337792.

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

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

21 Marcelo Fonseca Faraj and Christian Schulz. Buffered Streaming Graph Partitioning. *ACM Journal of Experimental Algorithmics (JEA)*, 27:1.10:1–1.10:26, 2022. doi:10.1145/3546911.

22 Sebastian Forster and Tijn de Vos. Faster Cut Sparsification of Weighted Graphs. *Algorithmica*, 85(4):929–964, 2022. doi:10.1007/s00453-022-01053-4.

23 Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *National Academy of Sciences*, 104(1):36–41, 2007. doi:10.1073/pnas.0605965104.

24 Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free Massively Distributed Graph Generation. In *32nd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 336–347. IEEE Computer Society, 2018. doi:10.1109/IPDPS.2018.00043.

25 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.

26 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.

27 Lars Gottesbüren, Nikolai Maas, Dominik Rosch, Peter Sanders, and Daniel Seemaier. KaMinPar. Software, swhId: swh:1:dir:2ec38dbccb676136f71dc1796d6a06b450c48c6d (visited on 2025-09-03). URL: https://github.com/KaHIP/KaMinPar/commit/73eeaa2371c6826e166c9ca1383996f14d8c7a2a, doi:10.4230/artifacts.24666.

28 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.

29 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.

30 Lars Gottesbüren, Nikolai Maas, Dominik Rosch, Peter Sanders, and Daniel Seemaier. Linear-time multilevel graph partitioning via edge sparsification, 2025. doi:10.48550/arXiv.2504.17615.

31 Milad Rezaei Hajidehi, Sraavan Sridhar, and Margo I. Seltzer. CUTTANA: Scalable Graph Partitioning for Faster Distributed Graph Databases and Analytics. *Proceedings of the VLDB Endowment*, 18(1):14–27, 2024. doi:10.14778/3696435.3696437.

32 Bruce Hendrickson and Robert Leland. A Multi-Level Algorithm For Partitioning Graphs. In *ACM/IEEE Conference on Supercomputing*, pages 28–es. ACM, 1995. doi:10.1145/224170.224228.

33 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.

34 Artur Jez. Faster Fully Compressed Pattern Matching by Recompression. *ACM Transactions on Algorithms (TALG)*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.

35 George Karypis and Vipin Kumar. Analysis of Multilevel Graph Partitioning. In *ACM/IEEE Conference on Supercomputing*, pages 29–es. ACM, 1995. doi:10.1145/224170.224229.

36 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.

37 Robert Krause, Lars Gottesbüren, and Nikolai Maas. Deterministic Parallel High-Quality Hypergraph Partitioning, 2025. doi:10.48550/arXiv.2504.12013.

38 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.

39 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.

40 Jure Leskovec and Christos Faloutsos. Sampling from Large Graphs. In *12th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 631–636. ACM, 2006. doi:10.1145/1150402.1150479.

41 Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph Evolution: Densification and Shrinking Diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1):2–es, 2007. doi:10.1145/1217299.1217301.

42 Gerd Lindner, Christian L. Staudt, Michael Hamann, Henning Meyerhenke, and Dorothea Wagner. Structure-Preserving Sparsification of Social Networks. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 448–454. ACM, 2015. doi:10.1145/2808797.2809313.

43 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.

44 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.

45 Joel C. Miller and Aric A. Hagberg. Efficient Generation of Networks with Given Expected Degrees. In *8th International Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 115–126. Springer, 2011. doi:10.1007/978-3-642-21286-4_10.

46 Mark E. J. Newman and Michelle Girvan. Finding and Evaluating Community Structure in Networks. *Physical Review E*, 69, February 2004. doi:10.1103/PhysRevE.69.026113.

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

48 Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Conference on Scientific and Statistical Database Management (SSDBM)*, pages 22:1–22:12. ACM, 2013. doi:10.1145/2484838.2484843.

49 Daniel Salwasser, Daniel Seemaier, Lars Gottesbüren, and Peter Sanders. Tera-Scale Multilevel Graph Partitioning, 2024. doi:10.48550/arXiv.2410.19119.

50 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.

51 Daniel A. Spielman and Nikhil Srivastava. Graph Sparsification by Effective Resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011. doi:10.1137/080734029.

52 Christian L. Staudt and Henning Meyerhenke. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):171–184, 2016. doi:10.1109/TPDS.2015.2390633.

53 Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *7th International*

Conference on Web Search and Data Mining (WSDM), pages 333–342. ACM, 2014. doi:10.1145/2556195.2556213.

54 Jianshu Weng and Bu-Sung Lee. Event Detection in Twitter. In *5th International Conference on Weblogs and Social Media*. AAAI, 2011. doi:10.1609/icwsm.v5i1.14102.

55 Tom Chao Zhou, Hao Ma, Michael R. Lyu, and Irwin King. UserRec: A User Recommendation Framework in Social Tagging Systems. In *24th AAAI Conference on Artificial Intelligence*, pages 1486–1491. AAAI, 2010. doi:10.1609/AAAI.V24I1.7524.