Deep Multilevel Graph Partitioning

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Daniel Max Manfred Seemaier
aus Nürtingen

Tag der mündlichen Prüfung: 8. Mai 2026

1. Referent: Prof. Dr. Peter Sanders
2. Referent: Prof. Dr. Ümit V. Çatalyürek

To my mother, Marion Seemaier-Keller.

Abstract

The balanced graph partitioning problem asks for a partition of the vertices of a graph into k disjoint blocks of roughly equal weight while minimizing the total weight of cut edges. This is a fundamental problem in computer science with a wide range of applications. For example, in parallel processing, one can often model computation and communication of an application as a graph. A balanced partition of this graph then distributes workload evenly across processing elements (PEs), while a small cut reduces inter-processor communication.

Unfortunately, the graph partitioning problem is NP-hard and even NP-hard to approximate within any constant factor. At the same time, the graphs that arise in practice and require partitioning are often massive, which makes exact methods infeasible and motivates the use of heuristics. Among these, the *multilevel scheme* has proven particularly successful for obtaining high-quality partitions. It proceeds in three phases. During *coarsening*, the algorithm builds a hierarchy of successively coarser graphs, typically by contracting vertex sets. Next, an *initial partition* is computed on the coarsest level. Finally, during *uncoarsening*, the contractions are undone level by level, projecting the coarse partition onto the next finer graph. On each level, the partition is improved using local search algorithms (also called *refinement*). The scheme can partition a graph directly into k blocks (called *direct k-way partitioning*) or compute a k-way partition via *recursive bipartitioning*.

A vast body of literature studies heuristic algorithms for graph partitioning, and a wide range of sequential and parallel partitioners (both shared-memory and distributed-memory) has emerged from this work. Nevertheless, several challenges remain. Most of the research targets partitioning into a relatively small number of blocks k, typically $k \leq 256$. With modern parallel architectures offering ever more processing elements, substantially larger values of k are increasingly relevant. Unfortunately, state-of-the-art partitioners often struggle in this regime, producing highly imbalanced and thus infeasible or otherwise low-quality partitions, or suffering from excessive running times. Moreover, although modern parallel partitioners are fast in practice, they require considerable memory to store the input graph along with expensive auxiliary data structures, limiting the size of graphs that can be handled on a single machine. In addition, while linear running time is frequently observed empirically, they lack worst-case performance guarantees on arbitrary input graphs. All of these issues arise in both shared-memory and distributed-memory settings.

In this dissertation, we propose several techniques to address these limitations. We introduce *deep multilevel graph partitioning* (Deep MGP), a multilevel scheme that extends the coarsening phase *deep* into the initial partitioning phase. Deep MGP coarsens the input graph to a small size independent of k and then computes an initial bipartition. During uncoarsening, as blocks grow, it applies recursive bipartitioning to create additional blocks, keeping block sizes roughly constant until k blocks are obtained. In this way, Deep MGP combines the merits of direct k-way partitioning and recursive bipartitioning, yielding a stronger, more flexible, and arguably more elegant multilevel framework.

We further present KaMinPar, a shared-memory parallel implementation of Deep MGP that deliberately relies on simple, highly efficient building blocks. In particular, we use size-constrained label propagation both to compute clusterings during coarsening and as a

refinement algorithm during uncoarsening. Empirically, KaMinPar is at least 2× faster than current state-of-the-art shared-memory partitioners — even for small k — while obtaining competitive partition quality. Somewhat surprisingly, it can even outperform a single-level partitioner that attains reasonable quality. For very large k, KaMinPar is up to $\approx 8\times$ faster than the fastest competing algorithm based on direct k-way partitioning. For unweighted input graphs, KaMinPar can further guarantee balanced partitions by introducing rebalancing as an explicit component in multilevel partitioning.

During the development of KaMinPar, we found that memory consumption is the primary bottleneck limiting scalability to larger graphs. We therefore present and evaluate a set of optimizations that substantially reduce its memory footprint. Concretely, we design parallel label propagation clustering and graph contraction algorithms that use only $\mathcal{O}(n)$ auxiliary space rather than $\mathcal{O}(np)$, where n is the number of vertices and p is the number of processing elements. Combined with integrating an existing compressed graph representation, these changes reduce peak memory usage by up to 16× while preserving partition quality and maintaining similar running times. In particular, our optimizations enable partitioning a randomly generated graph with *one trillion* edges in under 8 minutes on *a single machine*, using about 900 GiB of memory. We call the resulting algorithm TeraPart.

Subsequently, we propose a configuration of KaMinPar that runs in strictly linear time and performs linear work, without making any assumptions about the input graph. To the best of our knowledge, this is the first multilevel partitioner with worst-case linear running time, while prior algorithms require $\mathcal{O}((n+m)\log n)$ work or more. In addition to using linear time building blocks for the multilevel scheme, this requires the construction of a coarse graph hierarchy whose total size is linear in the size of the input graph. To this end, we show that the coarsening scheme implemented in KaMinPar guarantees a geometric reduction of the number of vertices from level to level, and use graph sparsification to bound the density of coarse graphs. By benchmarking on graphs that show a considerable increase in density on coarser levels, we show that sparsification yields a 1.5× average speedup (up to 4× on some graphs) over the non-sparsifying baseline, while reducing average partition quality by only 1%. Moreover, this configuration outperforms state-of-the-art single-level partitioners, producing high-quality partitions in less time.

Lastly, we consider partitioning in distributed memory and propose dKaMinPar, the distributed-memory counterpart of KaMinPar. Our distributed partitioner is likewise based on Deep MGP and uses a batch-synchronous implementation of size-constrained label propagation for both coarsening and uncoarsening. In contrast to previous works, we enforce stricter balance constraints on the cluster weights during coarsening, and use explicit rebalancing to ensure balanced partition outputs. We further integrate the same compressed graph representation as in TeraPart to obtain xTeraPart. In addition, we adapt an unconstrained local search algorithm recently proposed for GPU partitioning to the distributed setting. The resulting approach is effective on mesh-like graphs as well as highly skewed web and social graphs, while computing balanced, high-quality partitions on up to 8 192 CPU cores and graphs with up to 16 trillion edges. Moreover, through unconstrained refinement, it substantially improves partition quality over previous distributed partitioners, narrowing the long-standing quality gap between sequential and shared-memory partitioners on the one hand and distributed-memory partitioners on the other.

Deutsche Zusammenfassung

Das balancierte Graphpartitionierungsproblem besteht darin, die Knoten eines Graphen in k nicht-überlappende Blöcke zu partitionieren und dabei das Gesamtgewicht der geschnittenen Kanten zu minimieren. Dies ist ein fundamentales Problem in der Informatik mit einem breiten Spektrum an Anwendungen. Zum Beispiel können Arbeitslast und Kommunikation in parallelen Anwendungen oft als Graph modelliert werden. Eine balancierte Partitionierung des Graphen induziert dann eine lastbalancierte Verteilung der Eingabedaten auf mehrere Prozessoren, während ein kleiner Kantenschnitt die Kommunikation zwischen den Prozessoren reduziert.

Leider ist das Graphpartitionierungsproblem NP-schwer und sogar NP-schwer, es innerhalb eines konstanten Faktors zu approximieren. Gleichzeitig sind die Graphen, die in der Praxis auftreten und partitioniert werden müssen, oft sehr groß, was exakte Verfahren unpraktikabel macht und den Einsatz von Heuristiken motiviert. Unter diesen hat sich das *Mehrstufenschema* als besonders erfolgreich erwiesen, um Partitionen hoher Qualität zu berechnen. Es gliedert sich in drei Phasen. Während der *Vergröberung* konstruiert der Algorithmus eine Hierarchie sukzessive gröberer Graphen, typischerweise durch Kontraktion von Knotenmengen. Anschließend wird auf der gröbsten Stufe eine *initiale k-Wege-Partition* berechnet (*direkte k-Wege-Partitionierung*). Schließlich werden während der *Verfeinerung* die Kontraktionen Stufe für Stufe rückgängig gemacht, wobei die grobe Partition auf den jeweils nächst feineren Graphen projiziert wird. Auf jeder Stufe wird die Partition mithilfe lokaler Suchverfahren verbessert.

Eine umfangreiche Literatur untersucht heuristische Algorithmen für die Graphpartitionierung, und aus diesen Arbeiten ist eine große Bandbreite sequentieller und paralleler Partitionierer (sowohl für den gemeinsam genutzten Speicher als auch für den verteilten Speicher) hervorgegangen. Dennoch bleiben mehrere Herausforderungen ungelöst. Die meisten Arbeiten konzentrieren sich darauf, Graphen in eine relativ kleine Anzahl an Blöcken (Parameter k) zu partitionieren, üblicherweise $k \leq 256$. Moderne parallele Computerarchitekturen bieten aber immer mehr Prozessoren, sodass viel größere Werte für k zunehmend relevant werden. Allerdings funktionieren aktuelle Partitionierer in diesem Teil des Parameterraums nicht sehr gut, und sie produzieren oft unbalancierte oder schlechte Partitionen, oder haben eine sehr hohe Laufzeit. Darüber hinaus sind moderne Partitionierer zwar schnell in der Praxis, benötigen aber viel Arbeitsspeicher, um den Eingabegraphen sowie teure Hilfsdatenstrukturen zu verwalten. Dies limitiert die maximale Größe der Graphen, die auf einer einzelnen Maschine partitioniert werden können. Zudem zeigen moderne Partitionierer auf vielen Eingabegraphen zwar ein lineares Laufzeitverhalten, es gibt aber keine strikten Garantien für beliebige Eingabegraphen. All diese Probleme treten sowohl bei parallelen Partitionierern im geteilten wie auch im verteilten Speicher auf.

In dieser Dissertation stellen wir mehrere Techniken vor, die diese Herausforderungen angehen. Wir führen *Deep Multilevel Graph Partitioning* (Deep MGP) als neues mehrstufiges Partitionierungsschema ein, das die Vergröberungsphase tief in die initiale Partitionierungsphase fortsetzt. Deep MGP vergröbert den Eingabegraphen immer auf eine kleine Größe, die unabhängig von k ist, und berechnet dann eine initiale Bipartition. Während der

Verfeinerungsphase wachsen diese Blöcke von Stufe zu Stufe, und werden durch rekursive Bipartitionierung in kleinere Blöcke zerteilt, sodass die Größe eines einzelnen Blocks konstant bleibt (bis alle k Blöcke erreicht wurden). Deep MGP kombiniert in dieser Art die Vorteile von direkter k-Wege-Partitionierung und rekursiver Bipartitionierung und ergibt ein stärkeres, flexibleres und unter Umständen eleganteres Partitionierungsschema.

Wir präsentieren zudem KaMinPar, eine parallele Implementierung von Deep MGP im gemeinsam genutzten Speicher, die bewusst auf einfache, aber effiziente Bausteine setzt. Insbesondere verwenden wir den größenbeschränkten Label-Propagation-Algorithmus sowohl für das Berechnen der Cluster während der Vergröberungsphase, als auch als lokalen Suchalgorithmus während der Verfeinerungsphase. Empirische Daten zeigen, dass KaMinPar mindestens 2× schneller als andere Partitionierer auf dem aktuellsten Stand der Technik ist — sogar für kleine k — und dabei Partitionen mit kompetitiver Qualität berechnet. Bemerkenswerterweise ist KaMinPar sogar schneller als ein einstufiger Partitionierer, der (in der Kategorie der nicht-mehrstufigen Algorithmen) gute Qualität erreicht. Für sehr große k ist KaMinPar bis zu $\approx 8\times$ schneller als andere Partitionierer, die auf direkter k-Wege-Partitionierung basieren. Wenn der Eingabegraph ungewichtet ist, kann KaMinPar schließlich balancierte Lösungen garantieren, indem wir die Rebalancierung als eine eigenständige Komponente eines mehrstufigen Algorithmus aufgreifen und integrieren.

Während der Entwicklung von KaMinPar ist uns aufgefallen, dass der primäre Flaschenhals, um größere Graphen zu partitionieren, der Speicherverbrauch ist. Wir präsentieren und erproben deshalb eine Reihe an Optimierungen, um diesen substantiell zu reduzieren. Konkret entwerfen wir parallele Label-Propagation und Graphkontraktionsalgorithmen, die $\mathcal{O}(n)$ anstelle von $\mathcal{O}(np)$ zusätzlichen Speicherplatz benötigen, wobei n die Anzahl der Knoten und p die Anzahl der Prozessoren bezeichnet. Kombiniert mit einer bereits existierenden komprimierten Graphrepräsentation reduzieren diese Optimierungen der Speicherplatzbedarf um bis zu 16×, während die Partitionierungsqualität erhalten bleibt und ähnliche Laufzeiten erreicht werden. Insbesondere erlauben unsere Optimierungen einen zufällig erstellten Graphen mit *einer Billion* Kanten in unter 8 Minuten auf *einer einzelnen Maschine* zu partitionieren, wofür ca. 900 GiB Arbeitsspeicher benötigt werden. Wir bezeichnen den resultierenden Algorithmus als TeraPart.

Anschließend stellen wir eine Konfiguration von KaMinPar vor, die strikt lineare Zeit und Arbeit benötigt, ohne dabei irgendwelche Annahmen über den Eingabegraphen zu machen. Nach bestem Wissen ist dies der erste mehrstufige Partitionierer mit linearer Laufzeit im schlimmsten Fall, während vorherige Partitionierer eine Laufzeit von $\mathcal{O}((n+m)\log n)$ oder größer benötigen. Neben Bausteinen mit linearer Laufzeit für das Mehrstufenschema erfordert dies eine Hierarchie an Graphen, deren Gesamtgröße linear in der Eingabegröße ist. Dazu zeigen wir, dass das Vergröberungsschema von KaMinPar eine geometrische Reduktion der Knotenzahl von Stufe zu Stufe garantiert, und verwenden Sparsifizierungstechniken, um die Dichte der gröberen Graphen zu begrenzen. Wir experimentieren auf einer Menge an Graphen, deren Dichte auf gröberen Stufen deutlich ansteigt und zeigen, dass die Sparsifizierung zu einem durchschnittlichen Speedup von 1.5× gegenüber einer Variante ohne Sparsifizierung führt. Gleichzeitig wird die Lösungsqualität um nur 1% reduziert. Diese Konfiguration schneidet besser ab als einstufige Algorithmen auf dem aktuellsten Stand der Technik, indem sie bessere Partitionen schneller berechnet.

Zuletzt betrachten wir die Partitionierung im verteilten Speicher und präsentieren dKaMinPar, das Gegenstück zu KaMinPar im verteilten Speichermodell. Unser verteilter Partitionierer basiert ebenfalls auf Deep MGP und verwendet größenbeschränkte Label-Propagation sowohl für die Vergröberungs- als auch für die Verfeinerungsphase. Im Gegensatz

zu früheren Arbeiten erzwingen wir strengere Balancierungsbedingungen für die Gewichte der Cluster während der Vergröberungsphase. Für die Verfeinerungsphase entwerfen wir einen verteilten Algorithmus, um die Balancierung der finalen Partition für ungewichtete Eingabegraphen sicherzustellen. Wir binden außerdem dieselbe komprimierte Graphrepräsentation wie in TeraPart ein und erhalten so xTeraPart. Darüber hinaus integrieren wir einen unbeschränkten lokalen Suchalgorithmus, der kürzlich für das Partitionieren auf GPUs vorgestellt wurde, in dKaMinPar. Der daraus resultierende Ansatz ist sowohl auf netzartigen als auch auf Web- und Sozialgraphen effizient und berechnet balancierte Partitionen mit hoher Qualität auf bis zu 8 192 Prozessorkernen und Graphen mit bis zu 16 Billionen Kanten. Mit dem unbeschränkten lokalen Suchalgorithmus erreichen wir außerdem eine deutlich verbesserte Lösungsqualität, und reduzieren so die Lücke zwischen der erreichbaren Qualität mit sequentiellen und parallelen Algorithmen im gemeinsam genutzten Speicher auf der einen Seite und verteilten Partitionierern auf der anderen Seite.

Acknowledgements

First and foremost, I would like to thank my advisor Peter Sanders, who welcomed me into his research group and guided me over the years. I also thank Ümit Çatalyürek for serving as a co-reviewer on my dissertation committee.

I am grateful to my current colleagues Jannick Borowitz, Stefan Hermann, Ashlin Iser, Moritz Laupichler, Nikolai Maas, Niccoló Rigi-Luperti, Matthias Schimeck, Dominik Schreiber, Tim Niklas Uhl, Stefan Walzer, and Marvin Williams for the many work- and non-work-related discussions we had over the years. In particular, I thank Jannick Borowitz, Nikolai Maas, Tim Niklas Uhl, and Marvin Williams for proofreading parts of this dissertation and for their many helpful comments and suggestions. I also thank my former colleagues Michael Axtmann, Timo Bingmann, Daniel Funke, Lars Gottesbüren, Demian Hespe, Tobias Heuer, Lukas Hübner, Lorenz Hübschle-Schneider, Florian Kurpicz, Sebastian Lamm, Hans-Peter Lehmann, Tobias Maier, and Sascha Witt for contributing to the collegial atmosphere that made our time in the group so enjoyable.

I want to give special thanks to Sebastian Schlag and Christian Schulz, who introduced me to graph partitioning almost a decade ago by supervising my bachelor's thesis. Without them, I most likely would not have started working on this problem, and subsequently, would not have written this dissertation. Moreover, I would like to express special thanks to Lars Gottesbüren, Tobias Heuer, and Nikolai Maas, whose close collaboration and shared interest in graph partitioning had a particularly strong impact on my work over the years.

I would like to thank my co-authors Ümit Çatalyürek, Adil Chhabra, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Florian Kurpicz, Nikolai Maas, Henning Meyerhenke, Dominik Rosch, Daniel Salwasser, Peter Sanders, Sebastian Schlag, Dominik Schweisgut, Christian Schulz, and Dorothea Wagner for their collaboration on graph-partitioning papers during my time as a doctoral student.

Lastly, I would like to thank all of the bright students with whom I had the chance to work over my academic career. In particular, I would like to thank Manuel Haag, Cedrico Knoesel, Daniel Salwasser, Dominik Rosch, Tobias Kempf, and Simeon Schrape for their excellent work and subsequent roles as student research assistants.

Funding. Part of this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. Furthermore, the authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`). This project has received funding from the European Research Council (ERC)[1] under the European Unions Horizon 2020 research and innovation program (grant agreement No. 882500).

[1] 

Table of Contents

1 Introduction

Partitioning a graph into k blocks of roughly equal weight while minimizing the total weight of cut edges is a fundamental problem in computer science, with applications spanning parallel processing, route planning, image processing, VLSI design, and many more [Bul+16]. In many of these settings, graph partitioning serves as a tool for graph decomposition. In parallel processing on graph-structured data, for example, the input graph is distributed over multiple *processing elements* (PEs), such as compute nodes of a high-performance cluster, CPU cores, or GPUs. Vertices then represent computational workload, so that a balanced partition aims to distribute work evenly across PEs. Edges model communication, so that minimizing the edge cut corresponds to reducing inter-PE communication.

Perhaps one of the most canonical examples arises in distributed scientific computing. When solving differential equations numerically, a continuous domain is discretized into a grid or mesh, and unknowns are associated with vertices. Iterative solvers then repeatedly apply stencil-like updates in which each vertex computes its next value from its previous value and those of its neighbors. To execute this workload on a cluster with p PEs without suffering from work imbalance, the mesh must be partitioned into p roughly equal-sized blocks, while keeping the edge cut small to limit communication between PEs. Since the graphs arising in this application are typically meshes or grids, early partitioning algorithms were designed primarily for such graphs.

In contrast, modern applications increasingly involve graphs from other domains, such as web graphs, social networks or citation graphs. A prominent example comes from machine learning on graph-structured data, where graph neural networks (GNNs) have emerged as a widely used class of models. The core computational primitive in many GNN architectures is the message-passing layer. Starting from initial vertex feature vectors, each layer updates every vertex representation by aggregating its own feature vector with those of its neighbors and then applying a function to the aggregate (typically a neural network) [Ham]. Figure 1.1 shows an example of a single message-passing layer. Since training large-scale GNNs is computationally expensive, distributed training is often essential. This again induces a balanced graph partitioning problem as described above, and empirical data suggests that partition quality (i.e., the number of cut edges) can strongly influence training performance [Mer+25].

Unfortunately, balanced graph partitioning is NP-hard [GJ79] and even NP-hard to approximate [AR06], so that optimal solutions can only be computed for small graphs or graphs that admit a small cut. In practice, however, graphs requiring partitioning are often huge, so that even quadratic time algorithms are often infeasible. Subsequently, heuristics are prevalent in the field. For roughly three decades, multilevel approaches have been the dominant technique for obtaining high-quality partitions in a short amount of time. Multilevel graph partitioning splits the computation into the three phases *coarsening*, *initial partitioning*, and *uncoarsening*. During coarsening, the algorithm constructs a hierarchy of successively coarse representations of the input graph, typically by computing a clustering or matching of the graph and subsequently contracting the clusters or matched edges. Once the graph is small, an initial partition is computed. During uncoarsening, this partition is finally

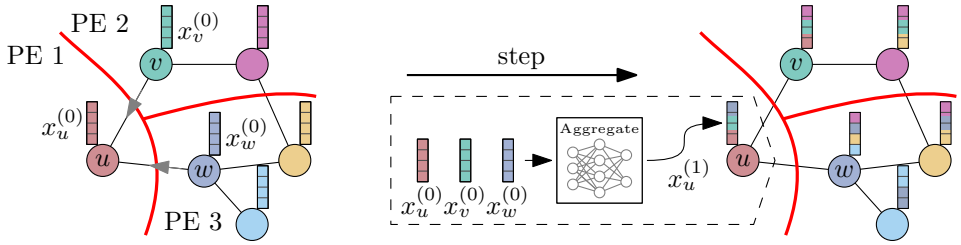

Figure 1.1 Example of a single message-passing layer. The graph is distributed over PEs 1, 2, and 3. Each vertex v aggregates its initial feature vector $x_v^{(0)}$ with those of its neighbors to produce $x_v^{(1)}$. When adjacent vertices reside on different PEs, this aggregation requires inter-PE communication (illustrated by gray arrows for vertex u). Colors in the feature vectors indicate which vertices contribute to the aggregate.

projected back onto the successively finer levels and refined using local search algorithms.

Despite the maturity of existing multilevel graph partitioners, several fundamental limitations remain that, taken together, restrict the applicability of current tools. First, most existing algorithms focus on partitioning into a small number of blocks, typically $k \leq 256$. When k is much larger, they often produce imbalanced partitions, low-quality solutions, or incur excessive running times. The reason for this is that the coarsest graph in a multilevel hierarchy must contain at least $C \cdot k$ vertices, for some constant $C \geq 1$, so that when k is large, coarsening cannot reduce the graph sufficiently, limiting the effectiveness of the multilevel approach. This problem is even more prevalent in parallel partitioners, where initial partitioning can incur a scalability bottleneck. Second, many partitioners lack robustness in practice. They are not always guaranteed to compute balanced solutions, even on unweighted input graphs. This is particularly a problem when k is large. Additionally, they often consume excessive amounts of memory, which is often the limiting factor when partitioning graphs of increasing sizes on a single-node machine. Moreover, while the building blocks used in coarsening, initial partitioning, and refinement are often (near-)linear-time algorithms, the overall complexity also depends on the properties of the constructed graph hierarchy. If coarsening stagnates early, the coarsest graph can remain large even for small k. Conversely, coarse graphs may become substantially denser than the input graph, leading to non-linear asymptotic running time. Third, graphs that exceed the main memory of a single machine require distributed processing. However, current distributed partitioners often scale well only in restricted regimes (e.g., for mesh-like graphs), and may perform poorly on graphs with a skewed degree distribution, such as web graphs or social networks.

There is no existing partitioner that simultaneously achieves high solution quality, guaranteed balance (for unweighted input graphs), scalable performance for large k, controlled memory consumption, and robustness across diverse graph classes. Addressing these limitations in isolation is insufficient. For instance, a partitioner that scales to large k but only with excessive memory consumption or limited parallel scalability would still leave a significant gap.

1.1 Main Contributions

In this dissertation, we tackle all of these problems by advancing the practical state-of-the-art in balanced graph partitioning along four directions: (i) scalable shared-memory partitioning that preserves high solution quality at large core counts and for large number of blocks k through *deep multilevel graph partitioning*, (ii) memory-efficient multilevel methods that

push partitioning to considerably larger graphs by reducing the memory footprint of several core steps, (iii) a linear-time multilevel algorithm that avoids pathological slowdowns by controlling the density of coarse graphs through sparsification, and (iv) distributed deep multilevel graph partitioning that extends the deep multilevel graph partitioning scheme to distributed-memory systems. To the best of our knowledge, the resulting partitioner, KaMinPar, is the fastest multilevel partitioner publicly available across a large and diverse set of benchmark instances, and can partition considerably larger graphs than previous algorithms, into a considerably larger number of blocks k.

This work is based on the four conference publications Ref. [Got+21b; Got+25; Sal+25; SS23b], the brief announcement Ref. [SS24], and the journal publication Ref. [Çat+23]. The resulting implementation is publicly available as open-source software[1].

Deep Multilevel Graph Partitioning. In the first part of the dissertation, we introduce Deep MGP as a generic meta-heuristic for graph partitioning. Deep MGP continues the multilevel scheme deep into initial partitioning, recursively bipartitioning the blocks of coarse graphs as they are projected to the finer graphs of the hierarchy. In contrast to previous approaches, the number of blocks in coarse partitions is therefore not static, but linked to the number of vertices in the coarse graphs (until k blocks are obtained). To exploit all of the available parallelism, multiple diversified attempts are performed on coarse levels to maintain the invariant that parallel tasks performed by p processors work on graphs with at least pC vertices, for some constant tuning parameter $C \geq 1$. We engineer the partitioner KaMinPar, which uses Deep MGP to achieve scalability to both large k and a considerable number of parallel cores while guaranteeing balanced solutions for unweighted input graphs. KaMinPar is based on rather simple building blocks, using parallel label propagation for both coarsening and refinement and a pool of simple, greedy heuristics for initial bipartitioning. Balance is guaranteed through a greedy balancing algorithm which employs parallelism over the blocks of the partition. Empirical data indicate a very favorable quality-performance tradeoff, achieving considerable speedups over state-of-the-art partitioners for small k. For large k, we achieve even greater speedups over other partitioners.

Tera-Scale Multilevel Graph Partitioning. While we developed and evaluated KaMinPar, we noticed that it required less memory than competing algorithms, even though we did not focus on memory efficiency. However, the memory available in a single machine was also the bottleneck in KaMinPar for scaling to larger graphs: it could either partition a graph within just a few minutes on a high-end CPU, or exceed the available memory. Thus, we present and study several optimizations to significantly reduce its footprint. We integrate these optimizations into KaMinPar, obtaining TeraPart. We devise parallel label propagation clustering and graph contraction algorithms that use $O(n)$ auxiliary space instead of $O(np)$, where n is the number of vertices in the graph. Moreover, we employ an existing compressed graph representation that enables iterating over a neighborhood by on-the-fly decoding at speeds close to the uncompressed graph. Combining these optimizations yields up to a 16-fold reduction in peak memory consumption, while retaining the same solution quality and similar speed. This configuration can partition a graph with *one trillion* edges in under 8 minutes *on a single machine* using around 900 GiB of RAM. To the best of our knowledge, this is the first work to employ the multilevel framework at this scale, which is vital to achieving

[1] At `github.com/KaHIP/KaMinPar`.

low edge cuts. Finally, we present a version of shared-memory parallel FM refinement with sparse gain tables that uses $O(m)$ space instead of $O(nk)$, reducing peak memory by factor 5.8 on medium-sized graphs without affecting running time.

Linear-Time Multilevel Graph Partitioning. The current landscape of balanced graph partitioning is split between high-quality but expensive multilevel algorithms and cheaper approaches with linear running time, such as single-level and streaming algorithms. In the third part of this dissertation, we show how to achieve the best of both worlds by constructing a multilevel algorithm with linear (expected) work. The central idea is to enforce geometric size reduction between hierarchy levels not only in the number of vertices, but also in the number of edges through graph sparsification techniques. This yields $\mathcal{O}(n + m)$ expected work without any assumptions on the input graph, while preserving the multilevel advantage of refinement across multiple levels of granularity. We integrate this approach into KaMinPar. On graphs that approximate multilevel worst case instances, we obtain speedups up to 4× at only about 1% average loss in solution quality. Moreover, the resulting algorithm outperforms single-level and streaming approaches, computing better partitions faster.

Distributed Deep Multilevel Graph Partitioning. The last part of the dissertation extends KaMinPar to the distributed-memory model, obtaining dKaMinPar. It scales to (at least) 8 192 cores (on randomly generated graphs) while achieving partitioning quality comparable to widely used sequential and shared-memory graph partitioners. In comparison, previous distributed graph partitioners scale only in more restricted scenarios and often induce a considerable quality penalty compared to non-distributed partitioners. When partitioning into a large number of blocks, they even produce infeasible solutions that violate the balance constraint. dKaMinPar achieves its robustness by a scalable distributed implementation of the deep-multilevel scheme for graph partitioning. Crucially, this includes new algorithms for balancing during refinement and coarsening. We further improve dKaMinPar following two directions: firstly, by adapting a recent refinement algorithm originally designed for graph partitioning on GPUs, we further improve its solution quality, thereby shrinking the gap in solution quality between distributed and the highest-quality shared-memory partitioners considerably. Secondly, by integrating the same compression techniques as in TeraPart, dKaMinPar can handle graphs with up to 16 trillion edges on 128 HoreKa compute nodes with 256 GiB each in just under 10 minutes. These graphs are 64 times larger than what is feasible for competing algorithms.

1.2 Outline

This dissertation is structured as follows. In Chapter 2, we introduce the notation and basic concepts used throughout the dissertation and describe our experimental methodology. Chapter 3 reviews the literature on balanced graph partitioning, with a focus on practical heuristics. The four main contributions then build on each other in a natural progression. Chapter 4 introduces the deep multilevel graph partitioning scheme and KaMinPar, its shared-memory implementation. Subsequently, Chapter 5 proposes a set of optimizations that reduce KaMinPar's memory footprint, while Chapter 6 presents a configuration of KaMinPar with provable linear running time based on graph sparsification during coarsening. Chapter 7 finally extends deep multilevel graph partitioning to the distributed setting, drawing on techniques from Chapters 4 and 5. Each chapter concludes with a discussion of results and directions for future work. Chapter 8 concludes the dissertation.

2 Preliminaries

In this section, we introduce the basic definitions, concepts and notation that will be used throughout this thesis.

2.1 Graph Related Definitions and Concepts

Let $G = (V, E, c, \omega)$ be a weighted, undirected graph where $V = \{0, \dots, |V| - 1\}$ is the set of vertices and $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ is the set of edges. The functions $c : V \to \mathbb{N}_{>0}$ and $\omega : E \to \mathbb{N}_{>0}$ assign weights to the vertices and edges of G. These functions are extended to sets by setting $c(V') := \sum_{u \in V'} c(u)$ for $V' \subseteq V$ and $\omega(E') := \sum_{e \in E'} \omega(e)$ for $E' \subseteq E$. We denote the number of vertices in G (the *order* of G) as $n := |V|$ and the number of edges in G (the *size* of G) as $m := |E|$. We say that two vertices $u, v \in V$ are *adjacent* if $\{u, v\} \in E$. Likewise, two edges $e, e' \in E$ are adjacent if $e \cap e' \neq \emptyset$. An edge $e \in E$ is incident to vertex $u \in V$ if $u \in e$. The set $N(u) := \{v \mid \{u, v\} \in E\}$ denotes the *neighbors* of vertex $u \in V$ and $E(u) := \{e \mid u \in e\}$ is the set of its incident edges. The *degree* of a vertex $u \in V$ is the number of its neighbors, denoted $d(u) := |N(u)|$. The *maximum vertex degree* of a graph is $\Delta(G) := \max_{u \in V} d(u)$. A subgraph of G is a graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E \cap \{\{u', v'\} \mid u', v' \in V'\}$. Given a set $V' \subseteq V$, we say that $G' = (V', E')$ is the subgraph *induced* by V' if $E' = E \cap \{\{u', v'\} \mid u', v' \in V'\}$ and write $G' := G[V']$.

In a *directed* graph, an edge e is given as an ordered pair $e = (u, v)$. We call u the *source vertex* and v the *target vertex* of e. Unless stated otherwise, we assume all graphs to be undirected throughout this dissertation.

A *matching* $M \subseteq E$ in a graph $G = (V, E)$ is a set of edges such that no two edges are incident to a common vertex. A matching M is *maximal* if no additional edge can be added to M without violating the matching property. The matching is a *maximum* (weighted) matching if $\omega(M) \geq \omega(M')$ for all matchings M'.

Contracting a set of vertices $X \subseteq V$ constructs a new graph $G' = (V', E', c', \omega')$ by collapsing $X = \{x_1, \dots, x_\ell\}$ into a single vertex x with weight $c'(x) = c(X)$. For all other vertices $v \in V \setminus X$, $c'(v) = c(v)$. For each edge $\{x_i, v\} \in E$ with $x_i \in X$ and $v \in V \setminus X$, there is an edge $\{x, v\} \in E'$ with weight $\omega(\{x, v\}) = \sum_{\{x_i, v\} \in E_{X, V \setminus X}} \omega(\{x_i, v\})$ where $E_{X, V \setminus X}$ are the cut edges between X and $V \setminus X$. For all other edges $\{u, v\} \in E$ with $u, v \in V \setminus X$, $\omega'(\{u, v\}) = \omega(\{u, v\})$.

Given two integers $a < b$, we use $a..b := \{a, \dots, b\}$. Given a number $N \in \mathbb{N}$, we use the notation $[N] := \{0, \dots, N - 1\}$.

Balanced Graph Partitioning. The *balanced graph partitioning problem* asks for a partition of V into *blocks* $\Pi := \{V_1, \dots, V_k\}$ such that $\bigcup_{i \in [k]} V_i = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *number of blocks* k is given as a fixed input parameter. The *balance constraint* demands

that $c(V_i) \leq L_{\max} := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some *imbalance parameter* $\varepsilon > 0$.[1] We call a block V_i *overloaded* if its weight exceeds this limit, i.e., $c(V_i) > L_{\max}$. We say that a partition is *imbalanced* and *violating the balance constraint* if at least one of its blocks is overloaded. Otherwise, the partition is *balanced*. The objective is to minimize $\texttt{cut}(\Pi) := \sum_{i<j} \omega(E_{ij})$, where $E_{ij} := \{\{u,v\} \in E \mid u \in V_i, v \in V_j\}$ denotes the set of all *cut edges*. We say that blocks i and j are *adjacent* if $E_{ij} \neq \emptyset$. When a vertex $u \in V_i$ has a neighbor $v \in V_j$, $i \neq j$, we call u a *boundary vertex* and say that u is *adjacent* to block j. Consequently, the set of all boundary vertices forms the *boundary* of Π. Given $\Pi = (V_1, \ldots, V_k)$, we can obtain a new partition $\Pi' = (V_1', \ldots, V_k')$ by *moving* a vertex $u \in V_i$ from its current block i to some target block j, i.e., $V_i' := V_i \setminus \{u\}$, $V_j' := V_j \cup \{u\}$ and $V_\ell' = V_\ell$ for all other blocks $i \neq \ell \neq j$. The *gain* of this move is $g_j(u) := \texttt{cut}(\Pi) - \texttt{cut}(\Pi')$. We say that a move is *feasible* if it does not overload the target block. Otherwise, it is an *infeasible* move. The *gain of a vertex* u is the maximum gain of any feasible move, i.e., $g(u) := \max_{j \in [k], c(V_j)+c(u) \leq L_{\max}} g_j(u)$. Some algorithms base their decision on the *relative gain* of a move, which is defined as

$$g_{\text{rel}}(u) := \begin{cases} g(u) \cdot c(u) & \text{if } g(u) \geq 0, \\ g(u)/c(u) & \text{if } g(u) < 0, \end{cases}$$

i.e., the maximum gain $g(u)$ scaled by the vertex weight $c(u)$. We use $\Pi[u]$ to denote the current block of vertex u in Π, i.e., $\Pi[u] = i$ if and only if $u \in V_i$.

A *clustering* $\mathcal{C} := \{C_1, \ldots, C_\ell\}$ is also a partition of V, but ℓ is not given in advance and the blocks are usually referred to as *clusters*. Also, there is typically no balance constraint. A *size-constrained clustering* is a clustering where the weight of each cluster is bound by some parameter U. *Contracting a clustering* constructs the graph obtained after contracting each vertex set C_i.

Complexity of Graph Partitioning. The balanced graph partitioning problem is NP-complete [GJ79; HR73] and even NP-complete to approximate within any finite factor [AR06]. If we define k in terms of n, that is, set $k = n/C$, the problem is solvable in polynomial time for $C = 2$ only by computing a maximum weighted matching. For $C > 2$, the problem remains NP-complete [GJ79]. Given these results, it is unsurprising that most practical graph partitioning algorithms are heuristic in nature. Graphs requiring partitioning are often very large, so that even quadratic algorithms are infeasible in practice.

Graph Representation. We represent a graph $G = (V, E)$ in shared-memory using the *compressed sparse-row* (CSR) format. This representation consists of two arrays: $\mathcal{P}$ of size $n+1$ and $\mathcal{E}$ of size $\sum_{v \in V} d(v) = 2m$. The array $\mathcal{E}$ contains the concatenated adjacency lists of all vertices, while $\mathcal{P}$ stores pointers into $\mathcal{E}$ such that the neighbors of a vertex $v \in V$ are given by

$$\{\mathcal{E}[\mathcal{P}[v]], \mathcal{E}[\mathcal{P}[v]+1], \ldots, \mathcal{E}[\mathcal{P}[v+1]-1]\}.$$

[1] For general vertex weights, the balance constraint as stated above can make feasibility nontrivial, since even deciding whether a balanced partition exists is NP-complete. In much of the graph partitioning literature, this issue is ignored and ε is implicitly assumed to be large enough so that feasibility is not a concern. However, a more robust alternative is to incorporate the heaviest vertex into the balance constraint, e.g., by setting $L_{\max} := (1+\varepsilon)\frac{c(V)}{k} + \max_{v \in V} c(v)$ or $L_{\max} := \max\{(1+\varepsilon)\frac{c(V)}{k}, \frac{c(V)}{k} + \max_{v \in V} c(v)\}$.

Thus, each undirected edge is represented as two directed edges in opposite directions. For weighted graphs, we additionally store vertex weights and edge weights in separate arrays $\mathcal{C}$ (of size n) and $\mathcal{W}$ (of size $2m$).

2.2 Multilevel Graph Partitioning

Many high-quality graph partitioners employ the multilevel scheme, which consists of three phases: During the *coarsening phase*, the algorithms build a hierarchy of successively smaller graphs where each graph is a coarse approximation of the previous one. Coarse graphs are built by either computing vertex clusters or matchings and afterwards contracting them. Contracting a clustering $\mathcal{C} = \{C_1, \ldots, C_\ell\}$ entails contracting all clusters simultaneously. Likewise, a matching can be contracted by contracting the endpoints of the matched edges. Once the number of vertices of a coarse graph falls below a certain threshold or the coarsening algorithm converges, *initial partitioning* computes a partition of the coarsest graph. Finally, *refinement* subsequently undoes the contractions performed during coarsening. In each uncontraction, the partition is first projected to the finer graph and then improved using *local improvement* algorithms.

Generally, there are two ways to partition a graph into k blocks using the multilevel scheme, namely *direct k-way* partitioning and *recursive bipartitioning*. The former coarsens the graph until $\Omega(k)$ vertices are left – usually kC vertices where C is an input parameter – and then computes a k-way partition of the coarsest graph. The latter first computes a bipartition $\Pi = \{V_1, V_2\}$ and then recurses on the induced subgraphs $G[V_1]$ and $G[V_2]$ by partitioning V_1 into $\lceil \frac{k}{2} \rceil$ and V_2 into $\lfloor \frac{k}{2} \rfloor$ blocks. Note that many partitioners based on direct k-way partitioning use recursive bipartitioning for initial partitioning on the coarsest graph.

2.3 Shared-Memory Parallelism

The first part of this dissertation describes algorithms for graph partitioning in the shared-memory model.

Computational Model. To analyze the running time complexity of our algorithms, we use the *Parallel Random Access Machine* (PRAM) model. We consider a fixed number of processing elements (PEs), typically denoted by p, operating on shared memory. More precisely, we use the aCRQW (*asynchronous concurrent-read queued-write*) variant [San+19], which does not assume global lock-step synchronization between PEs. Instead, the running time accounts for contention on shared-memory accesses. Concurrent reads are permitted without additional cost, whereas concurrent writes are *queued*, so that if x PEs attempt to write to the same memory location, these writes are serialized and incur $\Omega(x)$ time. Write conflicts are resolved *arbitrarily*, i.e., when multiple writes target the same memory location, there is no guarantee which value is stored. We primarily consider the following complexity measures:

- Work: The total number of operations performed by all processors, equivalent to the sequential runtime if executed on a single processor.
- Time ($T(p)$): The number of parallel steps required to complete the computation with p processors.
- Span: The longest sequence of dependent computational steps in the algorithm. This represents a theoretical lower bound on the parallel execution time and corresponds to $T(p)$ as $p \to \infty$.

Our algorithms depend on fundamental parallel operations – namely, reductions and prefix sums – for efficient data aggregation and distribution. A *reduction* computes an associative function (e.g., a sum) over an array and, using a standard tree-reduction approach, can be implemented in $\mathcal{O}\left(\frac{n}{p} + \log(n)\right)$ time, with a span of $\mathcal{O}(\log(n))$ and total work $\mathcal{O}(n)$. Similarly, a *prefix sum* computes the cumulative result of an associative operation over an array under the same time, span and work complexities.

We also use atomic concurrent updates of memory cells via *compare-and-swap* operations, denoted `CAS(m, x, y)`. This operation writes the value `y` to the memory cell `m` only if its current value is `x`. The update occurs atomically, i.e., if multiple processors concurrently attempt swaps using the same expected value `x` with a different new value, only one will succeed. The operation returns a boolean indicating whether the swap was successful. Consequently, this allows efficient, race-free parallelizations without relying on traditional locks.

TBB. We parallelize our shared-memory codes using the oneAPI Thread Building Blocks (oneTBB) library [TBB].

2.4 Distributed-Memory Parallelism

Chapter 7 describes graph partitioning algorithms for the distributed-memory model. In this architecture, processing elements (PEs) have their own memory and communicate with each other by passing messages via a network. This section introduces the fundamentals of the model and describes common communication patterns that arise in practice.

Model of Communication. Communication in distributed-memory systems follows a message-passing model, where data transfer occurs explicitly between processes. We assume that all p processors are connected by a full-duplex, single-ported communication network. A widely used approach to cost modeling in such systems is the linear-affine communication model, which accounts for both fixed and variable costs associated with data transfer. The total communication cost of sending a message with b words between two compute nodes is $\alpha + \beta b$, where α represents the message startup latency cost, and β represents the per-word cost of communication, which depends on network bandwidth. Common communication patterns that we use in this dissertation are [San+19]:

- Broadcast: A single processor contributes a value, which is delivered to every other processor.
- (All-)Reduce: Every processor contributes a value, and an associative operation combines these values. In a *Reduce* operation, the final result is stored on a designated root processor. In an *All-Reduce* operation, every processor receives the final result.
- (All-)Gather: Every processor i contributes a value x_i, which are collected in an ordered sequence $\langle x_1, x_2, \ldots, x_p \rangle$. In a *Gather* operation, the final sequence is only stored on the root processor. In an *All-Gather* operation, every processor receives the final sequence.
- Scatter: A single processor contributes p values $x_1, \ldots, x_p$, which are sent to other processors such that processor i receives value x_i.
- All-to-all: Every processor contributes a unique data segment, and these segments are exchanged so that each processor receives a distinct segment from every other processor.

Additionally, point-to-point operations send a single message from one processor to another. The cost of these collective and point-to-point operations is summarized in Table 2.1.

Table 2.1 Communication costs of collective operations under the linear-affine communication cost model, based on Ref. [San+19]. Here, p is the number of processors, b is the message size per processor, α is the message startup latency and β is the per-byte transfer cost.

Operation	Cost	MPI Operations
Point-to-point	$\mathcal{O}(\alpha + \beta b)$	`MPI_Send()`, `MPI_Recv()`
Broadcast	$\mathcal{O}(\alpha \log(p) + \beta b)$	`MPI_Bcast()`
Reduce	$\mathcal{O}(\alpha \log(p) + \beta b)$	`MPI_Reduce()`
All-Reduce	$\mathcal{O}(\alpha \log(p) + \beta b)$	`MPI_Allreduce()`
Gather	$\mathcal{O}(\alpha \log(p) + \beta pb)$	`MPI_Gather()`
Scatter	$\mathcal{O}(\alpha \log(p) + \beta pb)$	`MPI_Scatter()`
All-Gather	$\mathcal{O}(\alpha \log(p) + \beta pb)$	`MPI_Allgather()`
All-to-All	$\mathcal{O}(p(\alpha + \beta b))$	`MPI_Alltoall()`

MPI. We implement our algorithms using the Message Passing Interface (MPI) [Gab+04; Mes23]. This interface implements communication procedures for all primitives listed in Table 2.1.

Evaluating Scalability. When evaluating distributed algorithms, we often focus on their *scalability*, i.e., their performance as the number of processors increases. To assess this, we conduct two types of experiments: *weak scaling* and *strong scaling*. In weak scaling experiments, the input size grows proportionally with the number of processors. Specifically, we use artificially generated graphs where the number of vertices scales with the number of processors while maintaining a fixed average degree. In contrast, strong scaling experiments increase the number of processors while operating on a fixed input graph. We utilize both real-world and artificially generated graphs for these experiments.

2.5 Experimental Methodology

This section outlines the general experimental setup and methodology used throughout this thesis. This includes details on result aggregation, the types of plots used for evaluation, the graph benchmark sets employed, and the computational environments in which the experiments were conducted.

Result Aggregation and Visualization. In our experiments, we generally perform multiple repetitions (unless stated otherwise, 5 repetitions) for each graph partitioner and benchmark instance with different seeds for random number generation. We use the arithmetic mean to aggregate the edge cut, running time, and imbalance on a per-instance basis. Occasionally, partitioners produce partitions that violate the specified balance constraint. In such cases, we discard all violating repetitions and aggregate results only over those that satisfy the balance constraint. If all repetitions result in imbalanced partitions, we classify the instance as *imbalanced* and thus *infeasible*. We further impose per-instance time limits, specified in the respective sections. If a partitioner exceeds this limit, we include the specified time limit in the running time aggregate for the repetition. If all repetitions exceed the time limit (or fail due to other cases, such as segmentation faults caused by bugs in the implementations), we classify the instance as *failed* and thus *infeasible*.

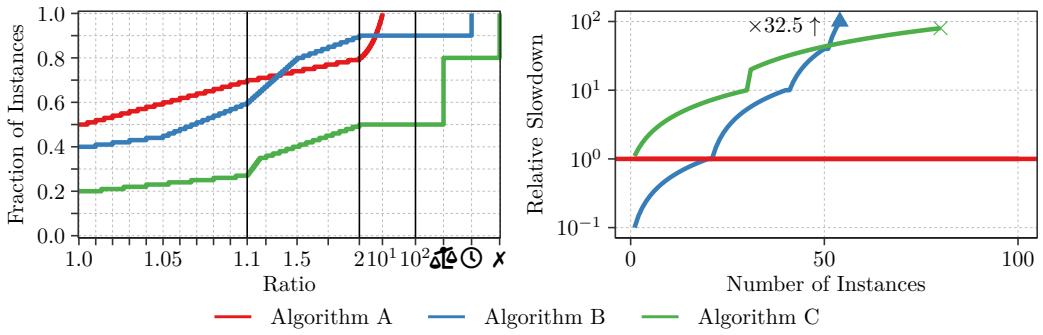

Figure 2.1 Example for a performance profile (left) and a slowdown plot (right).

When aggregating over multiple instances, we always report geometric means for performance metrics such as edge cuts or running times unless stated otherwise. This ensures that each instance has a fair influence on the overall aggregate. When we subsequently refer to *average* performance metrics (e.g., edge cuts or edge cut ratios, running times or running time ratios), we mean geometric means throughout.

In addition to these overall aggregates, we use various plotting methods to provide further insights into our results. We introduce the most important plotting methods next.

Performance Profiles. When evaluating the partition quality (i.e., edge cuts) of different partitioners, relying solely on aggregated values such as averages or medians can be misleading. An algorithm that occasionally finds very good solutions but performs inconsistently overall may appear favorable in such summaries, even if another algorithm is more stable and consistently near-optimal. To account for this, we use *performance profiles* [DM02], which provide a more detailed comparison by capturing how the performance of an algorithm degrades relative to the best observed solution.

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

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

on the y-axis and the ratio 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).

Occasionally, competing graph partitioners may produce partitions that violate the specified balance constraint, experience crashes, or exceed the given per-instance timeout. We integrate these instances into the performance profiles by including markers on the x-axis: ⚖ (for the fraction of imbalanced partitions), 🕓 (for timeouts), and ✗ (for crashes).

Figure 2.1 (left) shows a performance profile on synthetic cut data as an example. Algorithm A finds the best edge cuts for 50% of the instances, whereas Algorithm B and Algorithm C do so for 40% and 20%, respectively. These shares can sum to more than 100% when multiple algorithms find the same best cut for some instances. Moreover, the plot shows that Algorithm A finds cuts within 1.1× of the best cut (found by any of the three

algorithms) on 70% of the instances and within 2× on 80%. The remaining cuts are within 10× of the best cut found. Algorithm B reaches the 2× threshold on 90% of the instances, but exceeds the given time limit on 10%. Algorithm C produces imbalanced partitions for 30% and crashes for 10%. Note the different scales on the x-axis and that the third segment uses a logarithmic scale.

Speedup and Slowdown Plots. To compare the relative running time between partitioners, we use slowdown and speedup plots. An example for a slowdown plot is shown in Figure 2.1 (right). In both plot variations, one algorithm is used as the baseline (Algorithm A in the example). For the other algorithms, we sort the per-instance slowdown resp. speedup over Algorithm A and plot the resulting points on the y-axis. If some partitioner is *much* slower resp. faster than the baseline on some instances, we cut the y-axis and indicate this using ▲ (Algorithm B). In this case, we annotate the plot with the geometric mean slowdown resp. speedup over all instances (Algorithm B is 32.5× slower than Algorithm A on average). Finally, we use × to indicate the presence of instances on which a partitioner crashed (Algorithm C).

Benchmark Sets. Throughout this thesis, we use several benchmark sets to evaluate our algorithms that cover a wide range of graph types and sizes. They are summarized here for orientation. Detailed descriptions and statistics are provided in the respective chapters where the sets are introduced.

- Benchmark Set A (listed in Tables 4.1 and 4.2) is the primary benchmark set for our shared-memory algorithms presented in Chapter 4 and Chapter 5.
- Benchmark Set B is a subset of Benchmark Set A, containing the 20 largest graphs by edge count (bolded in Tables 4.1 and 4.2). These graphs are used in Section 4.4.1.2 and for scalability experiments.
- Benchmark Set C (listed in Table 5.1) consists of large web graphs and is used for evaluation in Chapter 5.
- Benchmark Set D (listed in Table 6.1) contains graphs that are challenging for typical graph coarsening algorithms (see Chapter 6). This benchmark set is used in Section 6.3.
- Benchmark Set E (listed in Tables 7.1 and 7.2) is used for evaluation of our distributed-memory algorithm presented in Chapter 7.

Across these benchmark sets, we roughly distinguish between *regular*[2] and *irregular* graphs based on their maximum degree Δ. For instance, graphs that model social networks or web graphs often exhibit a few high-degree vertices (hubs) that connect to many other vertices, leading to an *irregular* structure. On the other hand, graphs derived from meshes or grids typically have a more uniform degree distribution, resulting in a *regular* structure. The sets generally include real-world as well as randomly generated graphs.

Machines. We use various machines to execute our experiments and list them here for reference. Machine A is equipped with an AMD EPYC 7702P 64-core processor, clocked at 2 GHz (with a turbo boost up to 3.35 GHz) with 256 MiB shared L3 cache and 1 TiB of main memory. Machine B refers to compute nodes of the bwUniCluster HPC system, where each

[2] We are aware that the term *regular* is commonly used for graphs in which every vertex has the same degree. In this thesis, we use the term only as a coarse label for degree heterogeneity as described here.

Table 2.2 Overview of machines and their software environment used in the experimental evaluation.

Machine	CPU	RAM	Software Environment
A	1× AMD EPYC 7702P (1× 64 cores)	1 TiB	Rocky Linux 9.4 Linux 5.14.0 GCC 14.2.0
B (bwUniCluster)	2× Intel Xeon Gold 6230 (2× 20 cores)	{96, 192} GiB	RHEL 8.8 Linux 4.18.0 GCC 14.1.0
C (HoreKa)	2× Intel Xeon Platinum 8368 (2× 38 cores)	256 GiB	RHEL 8.8 Linux 4.18.0 GCC 13.3.0
D	1× AMD EPYC 9684X (1× 96 cores)	1.5 TiB	Ubuntu 22.04 LTS Linux 5.15.0 GCC 13.2.0
E	2× Intel Xeon Gold 6530 (2× 32 cores)	3 TiB	Rocky Linux 9.5 Linux 5.15.0 GCC 13.2.0

compute node is equipped with Intel Xeon Gold 6230 processors (2 sockets with 20 cores each), clocked at 2.1 GHz (with a turbo boost up to 3.9 GHz) and featuring 27.5 MiB of cache per CPU. These nodes have either 96 GiB or 192 GiB of main memory. Similarly, Machine C refers to compute nodes of the HoreKa HPC system, where each node is equipped with Intel Xeon Platinum 8368 processors (2 sockets with 38 cores each), clocked at 2.4 GHz (with a turbo boost up to 3.4 GHz) with 57 MiB of shared L3 cache per CPU. These machines are equipped with 256 GiB of main memory and are interconnected via an InfiniBand 4X HDR 200 Gbit/s network, which provides approximately 1 microsecond latency. Machine D is equipped with a single 96-core AMD EPYC 9684X processor clocked at 2.55 GHz (with a turbo boost up to 3.7 GHz) featuring 1 152 MiB of shared L3 cache. This machine is equipped with 1 536 GiB of main memory. Finally, Machine E is a dual-socket machine with 3 TiB of RAM. Table 2.2 summarizes all machines used, along with their respective software environment.

3 Related Work

In this chapter, we survey the extensive body of work on balanced graph partitioning developed over the past decades. We focus on techniques for *general-purpose* partitioning, that is, methods which require no information other than the graph's topology and its vertex or edge weights and that are not fundamentally tailored towards a specific application domain. Because most practical approaches are heuristic, graph partitioning is naturally viewed as a bicriteria problem in which partition quality (i.e., the number of cut edges) must be traded off against the time required to compute the partition. We mostly focus on works that are based on the *multilevel scheme*, which forms the basis for virtually all general-purpose partitioners that achieve high quality. This scheme consists of three phases: coarsening, initial partitioning, and uncoarsening (also called refinement).

For a more comprehensive review of the field, we refer the reader to existing survey articles and monographs [BS13; Bul+16; Çat+23; NF98]. The survey by Fjällström [NF98] from the late 1990s reviews early multilevel techniques and parallelization efforts, as well as geometric partitioning methods. A more recent survey due to Buluç et al. [Bul+16] covers works up to the early 2010s and also discusses real-world applications of balanced graph partitioning. The most recent survey by Çatalyürek et al. [Çat+23] focuses primarily on advances in shared-memory and distributed parallel partitioning, streaming partitioning, problem variations, and additionally reviews techniques specific to hypergraph partitioning.

References and Contributors. Parts of the paragraphs on acyclic graph partitioning and GPU-based graph partitioning in Section 3.6 are adapted from our survey paper [Çat+23]. The adapted text was originally written by the author of this dissertation, with editing from the other co-authors of the survey. All remaining text in this chapter was specifically written for this dissertation by the author of this dissertation.

Structure. The remainder of this chapter is structured as follows. In Section 3.1, we give a high-level overview of multilevel graph partitioning. The multilevel scheme offers a large design space, as concrete implementations depend on the combination of building blocks used for coarsening, initial partitioning, and refinement. We survey these building blocks in the sequential, shared-memory and distributed-memory settings in Section 3.2, Section 3.3, and Section 3.4, respectively. Due to the heuristic nature of the field, new algorithmic contributions are typically validated empirically and are thus often accompanied by practical implementations. This led to a rich ecosystem of partitioning tools, which we review in Section 3.5. Finally, in Section 3.6, we briefly discuss problem variations that are closely related but not central to our contributions.

3.1 Multilevel Graph Partitioning

The multilevel scheme originally emerged in the context of algebraic multigrid solvers [BMR83] during the 1980s. Hendrickson and Leland [HL95a] were the first to apply it to the graph partitioning problem. Nowadays, virtually all high-quality general-purpose graph partitioners

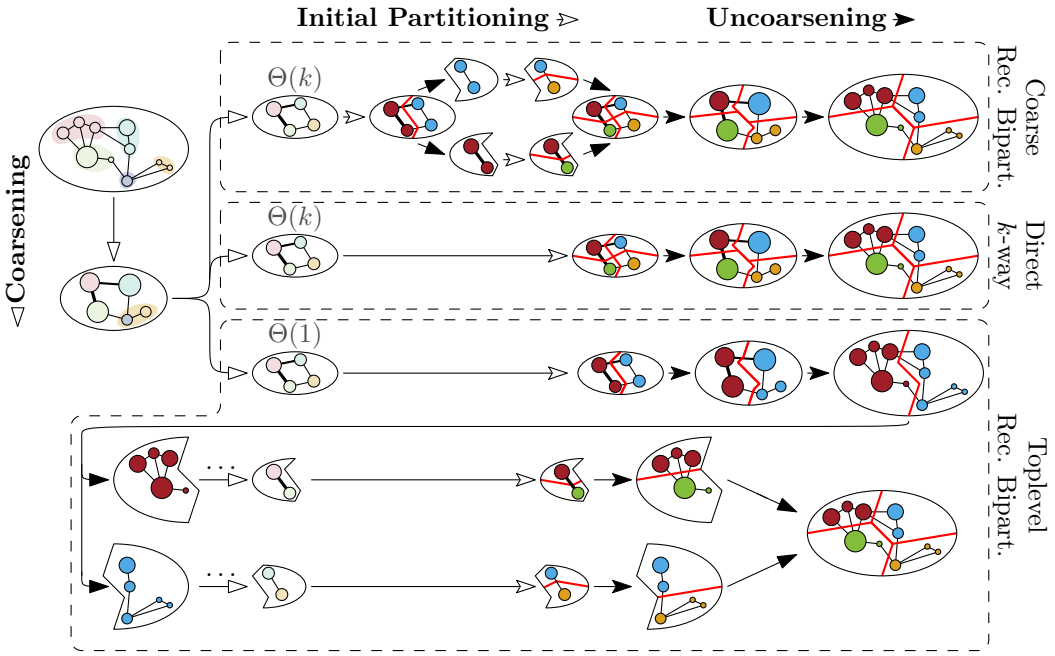

Figure 3.1 Illustration of multilevel graph partitioning schemes: initial partitioning via recursive bipartitioning of the coarsest graph (top), direct k-way partitioning (middle), and recursive multilevel bipartitioning (bottom).

are based on the multilevel scheme, see Section 3.5. Multilevel partitioners first construct a hierarchy of successively coarser, yet structurally similar representations of the input graph. Once the graph is sufficiently small, an initial partition is computed on the coarsest level and then projected back to finer levels, where it is successively refined by refinement algorithms usually based on local search. This scheme offers several advantages over single-level approaches. First, coarsening aggregates the weights of contracted vertices as well as the total edge weights between aggregates, so that a partition computed on the coarsest level can be projected to the original graph while preserving balance and edge cut. Second, moving a single vertex on a coarse level corresponds to moving an entire cluster of vertices on finer levels, helping refinement algorithms to escape poor local minima more effectively. Finally, since the coarsest graph is small while (hopefully) capturing much of the original graph structure, higher-quality initial partitions can be computed through more expensive algorithms that would be infeasible to apply to the input graph directly.

Recursive Bipartitioning and Direct k-way Partitioning. In general, there are two ways to partition a graph into k blocks, illustrated in Figure 3.1. In *recursive bipartitioning* (Figure 3.1, bottom), the multilevel scheme is first used to compute a partition $\{V_1, V_2\}$. The algorithm then recurses by partitioning the block-induced subgraphs $G[V_1]$ and $G[V_2]$ into $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$ blocks, respectively, until k blocks are obtained. If k is not a power of two, the bipartitioning steps must produce blocks with unequal weights. There are several drawbacks to recursive bipartitioning [Akh+17; HL95a]. First, even if the bipartitioning algorithm is optimal, the combined k-way partition can be arbitrarily poor [ST97]. Second, it restricts the view of the refinement algorithm used during the uncoarsening phase to two blocks. Lastly, enforcing balance is more challenging since early bipartitions must be computed with smaller ε (see discussion below). On the other hand, *direct k-way partitioning*

(Figure 3.1, middle) coarsens the graph only once down to $\Omega(k)$ vertices (typically down to kC vertices, where C is a tuning parameter). Then, an initial partitioning algorithm partitions the coarsest graph into k blocks directly. This usually gives better solution quality even empirically [Akh+17]. Note that many multilevel graph partitioners based on the direct k-way partitioning scheme use recursive bipartitioning to compute an initial partition of the coarsest graph (Figure 3.1, top) [Akh+17; Got+21a; KK98c; LK16; SS11a].

Parallelization. The multilevel scheme can be parallelized by using parallel algorithms in the coarsening and refinement phases. We review these algorithms in Section 3.2 and Section 3.4. On coarse levels, however, the contracted graphs can be too small to keep all PEs busy, and scalability can be limited by excessive synchronization overheads. To mitigate this issue, Chevalier and Pellegrini [CP08] replicate coarse graphs, thereby enforcing a lower bound on the graph size per core. Each replica is coarsened independently (and replicated again if necessary) until an initial partition is computed. During uncoarsening, the best partition among the replicas is selected at each level and propagated to the next finer level, while the remaining candidates are discarded. This approach can improve solution quality without sacrificing efficiency, since assigning many PEs to very small graphs typically yields no additional speedup, as discussed above. Other approaches use parallelism on the coarsest level by performing independent initial partitioning attempts [ASS20; Dev+06; HSS10; LK13; MSS17]. If initial partitioning is performed via recursive bipartitioning, parallelism can further be used for independent attempts at computing the same bipartition, and by assigning independent bipartitioning tasks to different PEs [KK99; LK13]. If the bipartitioning tasks themselves use a multilevel cycle, using multiple PEs per bipartitioning task can improve efficiency [Got+24a].

Maintaining the Balance Constraint. Computing a balanced partition of a weighted graph with

$$L_{\max} \coloneqq (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$$

as the balance constraint is NP-complete via a reduction from standard scheduling on identical parallel machines [GJ79]. Therefore, many multilevel partitioners incorporate techniques that prevent the formation of heavy vertices during the coarsening process by penalizing the contraction of vertices with large weights [ÇA99] or enforcing a strict upper bound for vertex weights [HS17]. This makes it easier for initial partitioning to find a feasible initial partition. However, if the input graph already contains non-uniform vertex weights, there are usually no hard guarantees that the final partition adheres to $L_{\max}$ [Çat+23]. To avoid the NP-complete scheduling problem, Heuer et al. [HMS21] propose relaxing the balance constraint to

$$L'_{\max} = \left(\frac{4}{3} - \frac{1}{3k}\right) L^{\mathtt{OPT}}_{\max},$$

where $L^{\mathtt{OPT}}_{\max}$ denotes the smallest achievable maximum block weight, and the multiplicative factor corresponds to the approximation factor of the *longest processing time* [Gra69] algorithm. Thus, a partition adhering to $L'_{\max}$ can be computed in polynomial time. Alternatively, as noted in Section 2.1, the balance constraint might be relaxed to

$$L_{\max} \coloneqq \max\left\{(1+\varepsilon)\frac{c(V)}{k}, \frac{c(V)}{k} + \max_{v \in V} c(v)\right\}.$$

In the recursive bipartitioning setting, using the input imbalance parameter ε for each bipartition can produce blocks in the final k-way partition with weight up to approx. $(1+\varepsilon)^{\log_2(k)} \lceil c(V)/k \rceil$. Therefore, KaHyPar [HMS21; Sch+16] ensures that a k-way partition obtained via recursive bipartitioning is balanced by adapting the imbalance ratio for each bipartition individually. Let $G[V']$ be the subgraph of the current bipartition that should be partitioned recursively into $k' \leq k$ blocks. Then,

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

is the imbalance ratio used for the bipartition of $G[V']$. If each bipartition is ε'-balanced, then the final k-way partition is ε-balanced.

Iterated Global Search Strategies. To further improve partition quality, Walshaw [Wal04] proposes iterated *V-cycles*. First, a partition is computed using the multilevel scheme described above. Subsequent multilevel cycles repeat coarsening and uncoarsening, but restrict contractions to vertices that are assigned to the same block. As a result, the current partition can be projected onto the coarsest level and reused as the initial partition. Since coarsening is usually randomized, each cycle produces a different coarsening hierarchy, allowing refinement during uncoarsening to find additional improvements.

Sanders and Schulz [SS11a] further propose *F-cycles* and *W-cycles*. Like V-cycles, they restrict contractions to vertices that are assigned to the same block, so that the current partition can be reused as the initial partition. In an F-cycle, during uncoarsening, an independent V-cycle is performed at each level of the hierarchy. A W-cycle follows the same structure, but replaces the V-cycles by recursive W-cycles. Although these strategies can yield slightly better quality per cycle, they are also more expensive than standard V-cycles. Sanders and Schulz report that under equal time budgets, where the faster methods can afford more cycle repetitions, F- and W-cycles do not provide a consistent advantage over standard V-cycles.

n-level Multilevel Graph Partitioning. In standard multilevel graph partitioning, each coarsening step reduces the number of vertices by a constant factor, yielding a hierarchy of $\Theta(\log(n))$ levels. Osipov and Sanders [OS10] instead propose an n-level scheme that contracts (and, during uncoarsening, uncontracts) exactly one edge per level, resulting in $\Theta(n)$ hierarchy levels. To make this approach practical, two engineering challenges must be addressed. First, contractions must be supported efficiently by the underlying graph data structure. Second, refinement during uncoarsening must be *localized*, i.e., it must improve the partition in time sublinear in n per level. Empirically, the n-level scheme can improve solution quality, but it is also reported to be up to an order of magnitude slower than standard multilevel approaches [Got+22; OS10].

3.2 Coarsening

The coarsening phase of multilevel graph partitioning constructs a hierarchy of successively coarse graph representations by contracting edges. Typically, this is done by contracting a sequence of matchings [ÇA99; CP08; Dev+06; HL95a; KK98a; KK99; LK13; PR96; Sch+16; WC07] or by contracting clusters obtained from a graph clustering algorithm [AK06; ASS20; ÇA99; Çat+12; Got+21a]. In the following, we use the term *matching-based* coarsening for schemes that contract only pairs of adjacent vertices, and *cluster-based* coarsening for schemes

that contract arbitrary sets of vertices. Consequently, matching-based schemes reduce the number of vertices by at most a factor of two per hierarchy level, while cluster-based schemes achieve larger reduction factors.

3.2.1 Matching-based Coarsening

Matching-based coarsening schemes compute a matching on the current graph and contract the endpoints of each matched edge into a single coarse vertex on the next hierarchy level. In one of the first multilevel partitioners, Hendrickson and Leland [HL95a] compute a random *maximal* (i.e., not necessarily *maximum*) matching agnostic to edge weights. To better preserve the structure of the graph, one would ideally contract the edges of a *maximum-weight* matching. The rationale for this is that the total edge weight of a coarse graph, which bounds the edge cut induced by the initial partition, is reduced by the weight of the matched edges on each hierarchy level. An exact maximum-weight matching can be computed in $\mathcal{O}(m\sqrt{n}\log(nN))$ time for integer edge weights bounded by N, e.g., using the algorithm of Duan et al. [DPS18]. Alternatively, a strongly polynomial-time algorithm due to Gabow runs in $\mathcal{O}(n(m + n\log(n)))$ time [Gab18]. However, these running time bounds are far above the typical near-linear running time of modern multilevel graph partitioners. In addition, multilevel partitioners are heuristic in nature, so that coarsening does not require an optimal maximum-weight matching. Approximation algorithms can break the current $\Theta(m\sqrt{n})$ barrier of exact matching algorithms. Duan and Pettie [DP14] present a $(1-\varepsilon)$-approximate maximum weight matching algorithm with running time $\mathcal{O}\big(m\varepsilon^{-1}\log(\varepsilon^{-1})\big)$. Despite this, for the reasons mentioned above, practical partitioners typically use even simpler linear-time approximation algorithms or simply greedy heuristics to obtain a sufficiently heavy matching.

Heavy-Edge Matching. Karypis and Kumar [KK95b; KK98a] propose the *heavy-edge matching* heuristic, which can be implemented in a single pass over the edges. The algorithm iterates over the vertices in random order and matches each vertex u with an unmatched neighbor v that maximizes the incident edge weight $\omega(u, v)$. Note that this heuristic provides no approximation guarantee. Their experiments show that this heuristic typically yields smaller edge cuts than random matching, as used, for example, by Hendrickson and Leland [HL95a]. They further propose *light edge matching*, which prefers low-weight edges, and *heavy clique matching*, which aims to contract dense subgraphs. However, these alternatives yield less favorable trade-offs between running time and cut quality than heavy-edge matching [KK95b]. The heavy-edge matching heuristic is also used by various other graph partitioners [CP08; Dav+20; Gil+24; KK99; LK13; PR96; WC00].

Global Paths Algorithm. Maue and Sanders [MS07] propose the *Global Paths Algorithm* (GPA), which computes a $\frac{1}{2}$-approximation of a maximum-weight matching. Note that there are many matching algorithms with the same [Avi83; DH03a; Pre99] or better approximation factors [DH03b; DP14]; we highlight this algorithm since it is used by Holtgrewe et al. [HSS10] as well as by Sanders and Schulz [SS11a] specifically for multilevel graph partitioning. Holtgrewe et al. [HSS10] report better solution quality due to GPA than heavy-edge matching.

GPA first sorts all edges by decreasing weight and incrementally builds a collection of vertex-disjoint paths and even-length cycles. Edges that connect inner-path vertices or close paths to odd-length cycles are discarded. In a second step, the algorithm computes an optimal maximum-weight matching independently on each resulting path or cycle via dynamic programming, and returns the union of these matchings.

Two-Hop Matching. A major shortcoming of matching-based coarsening schemes is that they are ineffective on graphs that do not admit sufficiently large maximal matchings. For example, consider a star graph, where all vertices have degree 1 except for the center vertex, which has degree $n-1$. A maximum matching only contains a single edge, yielding a coarsening hierarchy with $\Theta(n)$ levels. To address this issue, LaSalle et al. [LaS+15] propose *two-hop matching*, which also allows matching vertices that are not adjacent but share a common neighbor (i.e., are two hops apart). The algorithm works as follows. First, it computes a standard matching, e.g., using the heavy-matching heuristic. If this matching covers at least 75% of the vertices, it is returned and no two-hop phase is performed. Otherwise, the remaining unmatched vertices are grouped into *leaves* (degree-one vertices), *twins* (vertices with identical neighborhoods), and *relatives* (vertices that share at least one common neighbor). The algorithm then matches leaves, followed by twins, and finally relatives, until 75% of the vertices are matched. In their experimental evaluation, LaSalle et al. [LaS+15] demonstrate that two-hop matching can yield considerable speedups of up to 7×, especially on graphs featuring a skewed degree distribution, such as web or social graphs. Gilbert et al. [Gil+24] also use heavy-edge matching with additional two-hop matching for their GPU partitioner.

Davis et al. [Dav+20] propose a related matching strategy based on *brotherly matching*. After performing one iteration of heavy-edge matching, the algorithm processes the remaining unmatched vertices as follows. For each unmatched vertex u, it selects the neighbor v maximizing the incident edge weight $\omega(u, v)$ and uses v as *matchmaker* by matching all currently unmatched vertices of v pairwise. If the number of unmatched neighbors of v is odd, the remaining vertex is matched with v. Since v is already matched with some other vertex w (otherwise, u would not have been unmatched), this yields a 3-way or 4-way matching, depending on whether w is also a matchmaker with an odd number of unmatched neighbors. In the latter case, the 4-way matching is split into two 2-way matches.

Edge-Rating Functions. Holtgrewe et al. [HSS10] propose more sophisticated rating functions to guide the matching algorithm instead of plain edge weights. They obtain the best results with the ratings

$$\text{expansion}^\star(e = \{u, v\}) \coloneqq \frac{\omega(e)}{c(u) \cdot c(v)}, \quad \text{expansion}^{\star 2}(e = \{u, v\}) \coloneqq \frac{\omega(e)^2}{c(u) \cdot c(v)}$$

and

$$\text{innerOut}(e = \{u, v\}) \coloneqq \frac{\omega(e)}{\omega(E(v)) + \omega(E(u)) - 2\omega(e)}.$$

Sanders and Schulz [SS11a] also use these rating functions in combination with the GPA algorithm for graph coarsening.

Parallelization in the Shared-Memory Setting. LaSalle and Karypis [LK13] discuss several shared-memory parallelizations of heavy-edge matching. We briefly outline their best-performing variant, which essentially computes a standard heavy-edge matching subject to race conditions, while resolving conflicts in a subsequent pass. The matching is represented using a shared array M, so that $M[v]$ stores the matched neighbor of vertex v. Initially, $M[v] = v$ for all vertices v. The algorithm iterates over vertices in parallel. For each vertex v, an unmatched neighbor u is selected according to the heavy-edge heuristic and tentatively recorded by setting $M[v] = u$ and $M[u] = v$. These writes happen unprotected and are subject to race conditions. Thus, the process can produce asymmetric matchings, where

$M[v] = u$ but $M[u] \neq v$. A subsequent cleanup pass resolves such conflicts. To this end, each thread once more iterates over its vertices, and resets $M[v] = v$ for each vertex v where $M[M[v]] \neq v$. LaSalle and Karypis [LK13] report that these conflicts occur rarely in practice and therefore have negligible impact on the resulting matching. They also consider alternative parallelization schemes based on locks or on explicit inter-thread coordination that mimics the distributed-memory request resolution (see below), but find these variants to be less efficient.

Parallelization in the Distributed Setting. In the distributed setting, each PE only stores a subgraph of the global input graph. To exchange information between adjacent vertices stored on different PEs, explicit communication is required.

Walshaw et al. [WCE97] parallelize heavy-edge matching using a two-pass scheme. In the first pass, each PE computes a heavy-edge matching on its local subgraph, restricting choices to locally owned neighbors. In the second phase, the remaining unmatched vertices may match across PEs. To this end, PEs first exchange the set of still unmatched interface vertices. Each PE then selects matching partners for its unmatched interface vertices among the available non-local neighbors and communicates the selection. If both endpoints of an edge select each other, the match is accepted.

Karypis and Kumar [KK96; KK97; KK99] parallelize the heavy-edge matching in the distributed setting by performing multiple passes. In each pass, every PE iterates over its local unmatched vertices and selects for each vertex u a matching partner v according to the heavy-edge heuristic. If v is also a local vertex, the match is committed immediately and both vertices are marked as *matched*. Otherwise, the PE buffers a *match request* to the owner of v. To avoid circular match requests, passes alternate between buffering only requests to larger (i.e., $u < v$) resp. smaller (i.e., $u > v$) vertices. After each pass, PEs exchange the buffered requests and resolve conflicts (e.g., when a vertex receives multiple requests) arbitrarily. Rejected vertices remain unmatched and are reconsidered in the next pass. In their earlier work [KK96], Karypis and Kumar reduce conflicts by computing a graph coloring and processing vertices in color-wise phases within each pass. However, they drop the coloring step in later work [KK97]. Chevalier and Pellegrini [CP08] use a similar parallelization approach of heavy-edge matching and report convergence typically within 5 passes.

3.2.2 Clustering-based Coarsening

Recall that clustering-based coarsening contracts more than two vertices into one coarse vertex. In the following, we write $\mathcal{C}[v]$ for the cluster containing a vertex v, i.e., $v \in \mathcal{C}[v]$.

Çatalyürek and Aykanat [ÇA99] propose a randomized agglomerative clustering algorithm for (hyper)graph coarsening. The algorithm starts from singleton clusters, i.e., $\mathcal{C}[v] = \{v\}$ for all vertices $v \in V$. It then iterates over the vertices in random order. For each vertex $u \in V$ that is still in a singleton cluster (i.e., $|\mathcal{C}[u]| = 1$), the algorithm scores each neighbor $v \in N(u)$ by

$$\frac{|\{\{u, w\} \in E \mid w \in \mathcal{C}[v]\}|}{c(\mathcal{C}[v]) + c(u)},$$

that is, it counts the edges connecting u to vertices in $\mathcal{C}[v]$ and normalizes by the total weight of the resulting cluster. Finally, u is assigned to the neighboring cluster with maximum score.

Abou-Rjeili and Karypis [AK06] generalize matching-based coarsening to a cluster-based variant by allowing a vertex to match with a neighbor that is already matched with another

Algorithm 1 Size-Constrained Label Propagation.

Input : Weighted graph $G = (V, E, c, \omega)$, size constraint U
Output : Clustering $\mathcal{C}$

```
1  for u ∈ V do
2      C[u] ← {u}
3  while iteration limit not reached and clustering not converged do
4      for u ∈ V in some order do
5          R := new Hash Map()
6          for v ∈ N(u) do
7              // Unless stated otherwise, score(u, v) := ω({u, v}).
8              R[C[v]] ← R[C[v]] + score(u, v)
9          K ← argmax_{K ∈ R.keys(), c(K ∪ {u}) ≤ U} R[K]      // Find best cluster
10         K ← K ∪ {u}                                          // Move u to cluster K
```

vertex, thereby forming clusters larger than two vertices. They propose several edge-ordering criteria based on edge weights, vertex weights and degrees, and the weighted core number of vertices. The criteria can be applied globally, by sorting all edges according to the chosen score, or locally, by iterating over vertices in random order and sorting the vertex's incident edges according to that score. To limit the vertex reduction factor between consecutive levels, they terminate a clustering iteration once the number of coarse vertices drops below half of the current number of vertices. Moreover, they cap the weight of any cluster at $c(V)/20$. Gottesbüren et al. [Got+21a] also cap the vertex reduction rate between coarsening levels by terminating the clustering process once the number of vertices shrinks by a factor of 2.5.

Size-Constrained Label Propagation. Meyerhenke et al. [MSS14] propose *size-constrained label propagation*, a clustering algorithm derived from the label propagation algorithm of Raghavan et al. [RAK07]. This algorithm is now used by various multilevel partitioners [ASS20; Got+24a; MSS14; MSS17]. The algorithm is parameterized by a maximum cluster weight U (the size constraint) and is outlined in Algorithm 1. Initially, each vertex is assigned to its own singleton cluster (line 2). The algorithm then performs multiple iterations over the vertices (line 3). During an iteration, vertices are visited in some order (line 4). For each vertex u, the algorithm selects a target cluster K that maximizes the total weight of edges between u and its neighbors currently in K, subject to the size constraint $c(K) + c(u) \leq U$ (line 9). The algorithm terminates once the clustering has converged or a maximum number of iterations is reached. To reduce running time, Meyerhenke et al. propose an *active set* strategy. Initially, all vertices are marked *active*. After a vertex is processed, it is marked *inactive* and is only reactivated if one of its neighbors changes its cluster assignment. During subsequent iterations, inactive vertices are skipped.

Meyerhenke et al. [MSS14] parameterize the algorithm as follows. Vertices are ordered in increasing degree order, ensuring that by the time the algorithm visits high-degree vertices, low-degree neighbors often only span few clusters. They report an 8% improvement in solution quality and 20% improvement in running time due to the ordering. Moreover, they perform up to 10 iterations, but terminate early if fewer than 5% of the vertices were moved during an iteration. The size constraint U is set to $\frac{L_{\max}}{18}$. The same algorithm can be used for refinement by setting U to $L_{\max}$ and initially assigning vertices to clusters representing

blocks. In this case, the algorithm resembles greedy refinement, see Section 3.4.1.

Note that a single round of size-constrained label propagation is similar to the agglomerative clustering algorithm by Çatalyürek and Aykanat [ÇA99] described above. The key difference is that Meyerhenke et al. enforce a hard upper bound on cluster weights, whereas Çatalyürek and Aykanat bias vertex assignments toward lighter clusters by scaling ratings inversely with cluster weight.

Wang et al. [Wan+14] also use size-constrained label propagation within a multilevel graph partitioner (that performs no refinement during uncoarsening). In addition to enforcing a size constraint, they downweight edges incident to heavy vertices similar to the approach by Çatalyürek and Aykanat [ÇA99]. Concretely, for a vertex u, a neighbor v contributes $\texttt{score}(u, v) \coloneqq \omega(\{u, v\})/c(v)$ to the score of cluster $\mathcal{C}[v]$ (see Algorithm 1, line 8).

Parallel Size-Constrained Label Propagation. Staudt and Meyerhenke [SM13; SM16] parallelize label propagation in the shared-memory setting by processing vertices in parallel in their natural order. They argue that the resulting race conditions caused by multiple threads moving adjacent vertices concurrently can be beneficial, as they introduce an additional source of randomization. They further introduced the active set strategy to improve running time, which is also used by Ref. [MSS14] as described above.

Akhremtsev et al. [ASS20] employ parallel size-constrained label propagation during the coarsening phase, and further tune the parallelization in several ways. First, they sort vertices by degree using a parallel sorting algorithm. They then process vertices in *work packages*, each containing a set of vertices whose total number of incident edges is roughly $\max\{1\,000, \sqrt{m}\}$. Threads repeatedly pull work packages from a shared global queue and process the contained vertices sequentially. To implement the active set strategy, neighbors of moved vertices are scheduled for the next iteration by inserting them into work packages for the subsequent iteration. Second, they maintain cluster weights using atomic compare-and-swap operations, ensuring that the size constraint U is adhered to.

An important tuning knob is the data structure used to aggregate cluster ratings (the Rating Map in Algorithm 1), especially in the parallel setting. Since this data structure must be thread-local, memory efficiency and cache locality (e.g., when using a simple array of size n, indexed by cluster IDs) must be balanced with computational overhead (e.g., when using a hash table). Akhremtsev et al. [ASS20] implement this with a thread-local vector of size n.[1] Heuer [Heu22] uses small thread-local hash tables when the number of entries is estimated to be small (below $2^{15}/3$), and otherwise switches to a larger thread-local hash table while capping the maximum number of different clusters considered at a tunable constant. LaSalle et al. [LaS+15] similarly use a thread-local small hash table for low-degree vertices, but switch to a thread-local vector of size n for high-degree vertices.

In contrast to the above approaches, Meyerhenke et al. [MSS17] propose a *relaxed* distributed-memory parallelization of size-constrained label propagation. Each iteration is split into $b \coloneqq \max\{\alpha, \beta/p\}$ phases, with tuning parameters $\alpha = 8$ and $\beta = 128$. In phase i, each PE processes roughly a $1/b$ fraction of its local vertices according to the sequential Algorithm 1. After each phase, neighboring PEs exchange the updated cluster assignments of interface vertices. To reduce synchronization overhead, communication from phase i is overlapped with computation from phase $i + 1$. The size constraint U is relaxed and enforced only *locally*. To this end, each PE maintains cluster weights based solely on its local

[1] See the source code of Mt-KaHIP v1.00 at `github.com/KaHIP/mt-KaHIP`.

vertices. Since clusters may span multiple PEs (interface vertices may join the clusters of ghost vertices), the global weight of a cluster can therefore grow to as much as pU.

Community Detection. Heuer and Schlag [HS17] guide coarsening using community detection. The key idea is to first compute communities that capture global structure, and then use this information to constrain the otherwise local decisions made during coarsening. Specifically, they compute a modularity-maximizing clustering using the Louvain method [Blo+08] and restrict subsequent coarsening (matching-based in their implementation) such that only vertices belonging to the same community may be contracted. They report improvements in solution quality by 1.1%–5.7%, depending on the instance family. Gottesbüren et al. [Got+21a] also use community detection to improve the solution quality of their shared-memory partitioner. They use the parallel Louvain method of Staudt and Meyerhenke [SM13; SM16] to compute the communities.

Ensemble Clustering. Meyerhenke et al. [MSS14] propose *ensemble clustering* to further improve partition quality. Instead of computing and contracting a single clustering, their approach computes multiple clusterings (an *ensemble*) by running the same clustering algorithm multiple times with different random seeds. A new clustering is then derived by clustering vertices together if and only if they appear in the same cluster in *all* clusterings of the ensemble (i.e., the intersection of the clusterings). This has two intuitive effects. First, the resulting clusters tend to be smaller, which slows down coarsening and yields more hierarchy levels. This can be beneficial for solution quality [AK06]. Second, the clusters form a stronger consensus about which vertices should be contracted, potentially making clustering more robust.

3.3 Initial Partitioning

Recall that there are two common approaches for computing an initial k-way partition on the coarsest graph. An initial partitioner may either compute k blocks directly (*direct k-way partitioning*) or obtain a k-way solution via recursive (multilevel) bipartitioning. Akhremtsev et al. [Akh+17] report better results through recursive bipartitioning than direct k-way partitioning in preliminary experiments.

Graph Growing Techniques. Karypis and Kumar [KK95b] propose two combinatorial approaches for computing an initial bipartition $\{V_1, V_2\}$. Both methods work by growing the first block V_1 around a random seed vertex v, until it reaches half of the total vertex weight, and assigning all remaining vertices to V_2. Since the performance of these approaches depends on the seed vertex v, they are repeated several times from different seeds. The first variant, called *graph growing*, grows the block in a breadth-first manner, repeatedly adding neighboring vertices until the block reaches half of the total vertex weight. The second variant, called *greedy graph growing* (GGG), also starts from a random seed but, in each step, selects the neighboring vertex whose addition to the block yields the smallest increase in cut. This can be implemented efficiently by storing vertices adjacent to the growing block in a priority queue keyed by gain. In their experiments, GGG consistently yields better cuts than the BFS-style growing strategy and also outperforms spectral graph bisection (briefly reviewed in Section 3.6.1).

Pool Partitioning. Many partitioners compute an initial partition by running a *pool* (also called *portfolio*) of heuristics (random bipartitioning, (greedy) graph growing variants, and size-constrained label propagation) [Got+24a; Sch+22]. Each heuristic is executed repeatedly with different random seeds (e.g., both Schlag et al. [Sch+16] and Gottesbüren et al. [Got+21a] run each heuristic 20 times), and the best (i.e., lowest edge cut, or smallest imbalance, if all partitions are imbalanced) partition found by any heuristic is returned.

To reduce the running time of pool-based initial partitioning, Gottesbüren et al. [Got+22] propose an adaptive selection strategy. The key observation is that, on a fixed instance, some heuristics consistently produce partitions that are far worse than the current best and thus unlikely to improve in additional repetitions. More precisely, each heuristic h is first executed 5 times. Assuming that the resulting cut values are normally distributed, the sample mean μ_h and standard deviation σ_h are computed. Let $\Pi^\star$ denote the best initial partition found so far (by any heuristic). Further repetitions are no longer assigned to h once

$$\mu_h - 2\sigma_h > \texttt{cut}(\Pi^\star).$$

This decision is based on the fact that, if cut values are normally distributed, roughly 95% of the cut values fall into the interval $\mu_h \pm 2\sigma_h$. Hence, if the optimistic bound $\mu_h - 2\sigma_h$ exceeds the best cut, additional runs of h are unlikely to yield an improvement.

3.4 Refinement

Next, we review refinement algorithms. We first look at simple greedy methods, which only move vertices that immediately improve the current partition. We then turn to more sophisticated local-search algorithms that incorporate a limited form of lookahead, allowing sequences of moves that may temporarily worsen the cut if later moves yield a net improvement. Finally, we look at *unconstrained refinement* algorithms, which temporarily relax the balance constraint.

3.4.1 Greedy Refinement

Arguably one of the simplest forms of partition refinement considers only single-vertex moves and performs a move only if it improves the current cut while preserving the balance constraint. To be more precise, such algorithms iterate over all vertices in some order (e.g., randomized [Got+21a] or sorted by degree [MSS14] or gain [KK95b; LK13]). For each vertex v, they compute the gain of moving v to each adjacent block V_i such that $c(V_i)+c(v) \leq L_{\max}$. If there exists an improving feasible move, v is moved to the block. In the literature, this procedure is sometimes described as a variant of size-constrained label propagation, where clusters correspond to blocks and the size constraint is set to $L_{\max}$ [ASS20; Got+21a; MSS14; MSS17; Sch13]. Typically, multiple passes of greedy refinement are performed to improve solution quality.

Parallelization in the Shared-Memory Setting. Although greedy refinement is conceptually simple, parallelizing it introduces two main challenges. First, the balance constraint must be preserved when multiple vertices are moved concurrently, possibly to the same target block. Second, computed gain values can become inaccurate when adjacent vertices are moved concurrently. In particular, two individually improving moves may interact negatively and worsen the cut overall, e.g., when swapping two adjacent vertices connected via a heavy edge as illustrated by Figure 3.2 (left). See Figure 3.2 (right) for another example.

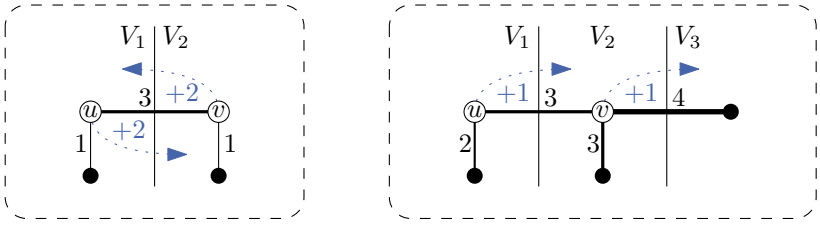

Figure 3.2 Illustration of inaccurate gains when moving adjacent vertices concurrently. **Left:** vertices u and v swap blocks. Each individual move has a gain of 2, but in combination, they increase the edge cut from 3 to 5. **Right:** vertex u is moved from block V_1 to block V_2, while vertex v is moved from block V_2 to block V_3. Each individual move has a gain of 1, but in combination, they increase the edge cut from 7 to 8.

In the shared-memory setting, LaSalle and Karypis [LK13] propose two strategies to mitigate these issues. Their first strategy uses locking to prevent conflicts. Before moving a vertex v, both the source and target blocks are locked, as well as all vertices adjacent to v. This prevents all conflicts, but induces considerable overhead. Their more fine-grained alternative restricts each pass to moves in a single direction by alternating between allowing moves only from larger to smaller block IDs and vice versa. This restriction prevents adjacent vertices from swapping blocks, but does not prevent all cases of move conflicts, as illustrated by Figure 3.2 (right). To enforce balance, they additionally collect moves proposed by threads in a buffer and roll back those that would violate the balance constraint.

Akhremtsev et al. [ASS20] and Gottesbüren et al. [Got+21a] enforce the balance constraint by updating the block weights via atomic compare-and-swap operations before moving a vertex. Akhremtsev et al. [ASS20] accept that gains can become inaccurate due to concurrent moves. In contrast, Gottesbüren et al. [Got+21a] propose *attributed gains* to validate the resulting gain after moving a vertex, and revert the move if the gain becomes negative.

Parallelization in the Distributed Setting. In the distributed setting, these issues become even more challenging due to limited synchronization and delayed visibility of remote moves. The approaches discussed below split refinement into multiple phases. In each computation phase, PEs move vertices locally without communication. Information (i.e., block assignments of interface vertices, changes in block weights) is exchanged only between computation phases.

Karypis and Kumar [KK99] compute a vertex coloring and perform refinement in color-wise phases. During each computation phase, PEs only move vertices that belong to the same color class. This guarantees that no two moved vertices are adjacent. Balance is enforced only with respect to locally stored (and possibly outdated) block weights. To this end, each PE maintains a copy of the current global block weights $c(V_i)$ for each $V_i \in \Pi$ and records the local weight changes $\Delta(V_i)$ induced by its moves to or from block V_i. A vertex is only moved to block V_i if $c(V_i) + \Delta(V_i) + c(v) \leq L_{\max}$. The global block weights $c(V_i)$ are updated between computation phases (e.g., by performing an all-reduce operation over the $\Delta(V_i)$). Since $\Delta(V_i)$ only accounts for PE-local moves, the balance constraint is not strictly enforced. In their subsequent work [KK97], Karypis and Kumar avoid computing a graph coloring and instead restrict moves to one direction per computation phase, as described above.

Meyerhenke et al. [MSS17] ignore conflicts from concurrently moved vertices and handle the balance constraint in the same manner as Karypis and Kumar [KK99], i.e., do not strictly enforce it but accept possibly imbalanced and thus infeasible solutions. Each refinement pass is split into $b = \max\{\alpha, \beta/p\}$ phases, where α and β are tuning parameters set to $\alpha = 8$

and $\beta = 128$. In each phase, the algorithm processes roughly one $1/b$-th of the vertices and greedily moves them to the best feasible target block. To reduce synchronization overhead, communication from phase i is overlapped with computation from phase $i + 1$.

Slota et al. [Slo+17] approximate balance by alternating between a *refinement phase* and a *balancing phase*. In both phases, vertices are greedily moved based on their gain to feasible target blocks. Similar to the approaches described above, each PE maintains a copy of the global block weights $c(V_i)$, which are only updated between refinement passes, and records local weight changes $\Delta(V_i)$ during a pass. However, instead of checking $c(V_i) + \Delta(V_i) + c(v) \leq L_{\max}$ to determine whether V_i is a feasible target block for vertex v, Slota et al. [Slo+17] approximate the block's global weight by

$$c'(V_i) \coloneqq c(V_i) + \alpha \cdot \Delta(V_i),$$

where α is a tuning parameter set to

$$\alpha \coloneqq p \cdot \left(\frac{3}{4} \cdot \frac{i}{I} + \frac{1}{4} \right).$$

Here, p is the number of PEs used for partitioning, I is the number of refinement passes, and i is the current refinement pass. Each PE then checks whether $c'(V_i) + c(v) \leq L_{\max}$ before moving v to block V_i. In a balancing pass, vertex gains are further scaled by $L_{\max}/c'(V_i)$ to bias moves towards underloaded blocks.

More recently, several authors [Kab+17; Mar+17] propose probabilistic methods that preserve balance only in expectation. Kabiljo et al. [Kab+17] also split refinement into multiple phases. In each computation phase, PEs collect candidate vertices to be moved between blocks. The intended move counts S_{ij} (number of vertices proposed to move from block V_i to block V_j) are then aggregated on a root PE and broadcast to all PEs. Each PE executes a proposed move from V_i to V_j with probability

$$\frac{\min\{S_{ij}, S_{ji}\}}{S_{ij}},$$

so that, in expectation, the same number of vertices is moved from V_i to V_j as from V_j to V_i. Martella et al. [Mar+17] follow a similar idea, but formulate balance in terms of edge counts per block rather than vertex counts. To remain consistent with the rest of this chapter, we restate their scheme for vertex balance. Concretely, they aggregate the number of vertices $\Delta(V_j)$ that should be moved by any PE to block V_j on a root PE and broadcast the aggregate to all PEs. Each PE then executes a proposed move to block V_j with probability

$$\frac{L_{\max} - c(V_j)}{\Delta(V_j)},$$

so that, in expectation, V_j is not overloaded.

3.4.2 Kernighan-Lin Algorithm

One of the earliest widely successful refinement heuristics for graph partitioning is due to Kernighan and Lin [KL70] (*KL refinement*). They consider the balanced bisection setting, i.e., improving a given bipartition $\Pi = \{V_1, V_2\}$ of a graph $G = (V, E, \omega)$ with edge weights ω and $L \coloneqq |V_1| = |V_2|$. Let $\Pi^{\texttt{OPT}} = \{V_1^{\texttt{OPT}}, V_2^{\texttt{OPT}}\}$ denote an optimal balanced bipartition. Then there exist sets $X \subseteq V_1$ and $Y \subseteq V_2$ such that

$$V_1^{\texttt{OPT}} = (V_1 \setminus X) \cup Y \quad \text{and} \quad V_2^{\texttt{OPT}} = (V_2 \setminus Y) \cup X,$$

i.e., one can transform Π into $\Pi^{\texttt{OPT}}$ by swapping the vertices of X and Y. Hence, refinement can be viewed as identifying an exchange (X, Y) that reduces the edge cut. Since finding an optimal exchange is NP-hard, the KL algorithm instead constructs X and Y using a greedy multi-pass heuristic as follows. At the start of a pass, X and Y are initialized as empty lists. Additionally, for each vertex $v \in V_j$, $j \in \{1, 2\}$, the gain value $g(v)$ due to moving v to the other block is precomputed. Then, during the $1 \leq i \leq L$-th iteration, the algorithm selects the pair of vertices $x \in V_1$ and $y \in V_2$ maximizing the reduction in edge cut $G_i \coloneqq g(x) + g(y) - 2\omega(\{x, y\})$ (called *swap gain*) when swapping x and y. Note that we assume $\omega(\{x, y\}) = 0$ for $\{x, y\} \notin E$. Moreover, note that possibly $G_i < 0$. Subsequently, x resp. y are appended to X resp. Y, removed from V_1 resp. V_2, and the gain values $g(\cdot)$ of the remaining vertices are updated to reflect the swap. Note that each vertex is only swapped once during this process. After L iterations, all vertices are appended to either X or Y, and the algorithm determines $1 \leq \ell \leq L$ such that the total swap gain

$$G \coloneqq \sum_{i=1}^{\ell} G_i$$

is maximized. If $G > 0$, the first ℓ vertices in X and Y are swapped and the algorithm continues with another pass. Otherwise, the swap is not performed and the algorithm terminates. Picking a single swap gain maximizing pair of vertices can be implemented in time $\mathcal{O}(n \log(n))$, so that a single pass of the algorithm can be implemented in time $\mathcal{O}\big(n^2 \log(n)\big)$.

3.4.3 Fiduccia-Mattheyses Algorithm

While KL refinement can produce strong improvements, it is computationally expensive. Fiduccia and Mattheyses [FM82] (*FM refinement*) reduce the runtime complexity to linear time per pass in the bipartition setting by replacing vertex swaps with single-vertex moves. To be more precise, their algorithm alternates between the two blocks, each time selecting the vertex with the maximum gain and moving it to the other block. Similar to KL refinement, each vertex is only moved once during a pass. After a pass, the algorithm rolls back the move sequence to the intermediate state that achieved the lowest edge cut. Selecting individual vertices rather than pairs of vertices enables efficient selection using a priority queue keyed by gain. In particular, Fiduccia and Mattheyses show that a single pass of the algorithm can be implemented in linear time via a novel bucket-based priority queue.

Since then, numerous authors have proposed improvements to the FM algorithm, making it one of the most widely used refinement algorithms today. The original publication [FM82] considers only the bipartite case. To refine a k-way partition, the bipartite algorithm can be used to repeatedly refine selected pairs of blocks individually [CL98]. Hendrickson and Leland [HL95b] extend the FM algorithm directly to k-way refinement by maintaining $k(k-1)$ priority queues, one for each ordered pair $(V_{\text{from}}, V_{\text{to}})$ of blocks. Each queue stores vertices that can be moved from block V_{from} to block V_{to}. Then, each iteration performs the highest-gain move across all queues that preserves the balance constraint, and subsequently updates the affected gains across all queues. However, since vertices can be contained in multiple priority queues, updating all affected gain values requires time $\mathcal{O}(km)$. This makes this variant computationally more expensive than the original linear-time algorithm.

Karypis and Kumar [KK98c] also propose a direct extension to k-way refinement, but maintain only a single, global priority queue containing vertices from all blocks. To reduce the number of vertices considered for movement, a vertex is inserted only if the total weight

of its incident edges to other blocks is at least the total weight of its incident edges to vertices in its current block. Thus, the global queue only contains a subset of the partition's boundary vertices. The algorithm repeatedly extracts the highest-gain vertex and moves it to the block that maximizes its gain while preserving balance. Rather than processing the entire queue, they terminate the search after x consecutive moves that did not improve the edge cut, and then roll back the last x moves to the best intermediate state found during refinement.

Träff [Trä06] proposes a direct extension to k-way refinement which retains linear-time complexity when using the bucket-based priority queues. The core idea is to maintain one priority queue Q_{V_i} per block V_i, storing only vertices currently in V_i and keyed by the maximum gain of the vertex. Additionally, one priority queue *per vertex* Q_u of size k stores the total weight of edges between u and neighbors in each block. This queue is used to determine the best target block of vertex u efficiently when moving u. Finally, a priority queue Q of size k tracks the current maximum key of each block-specific queue Q_{V_i}. In each iteration, the algorithm extracts the best block index V_i from Q, removes the corresponding best vertex u from Q_{V_i}, and moves it to the target block that maximizes its gain using Q_u. This move might violate the balance constraint; in this case, the next move is selected from the overloaded block, rather than based on Q. Subsequently, affected gains in Q_{V_i}, $V_i \in \Pi$, are updated and the keys in Q and Q_v, $v \in V$, are adjusted accordingly.

Highly Localized FM Refinement. Osipov and Sanders [OS10] develop a *highly localized* variant of FM refinement for their n-level partitioner (see Section 3.1). Their algorithm works as follows. For each block V_i, the algorithm maintains a priority queue Q_{V_i} containing vertices that are *not* in V_i, but have at least one neighbor in V_i. This contrasts with the variant by Träff [Trä06], where vertices are stored in the queue corresponding to their current block. Crucially, *only the endpoints* of the uncontracted edge are initially inserted into the queues. Note that if neither vertex is a boundary vertex, the queues remain empty and the search terminates immediately. As before, the queues are keyed by maximum gain, and in each iteration, the maximum gain move that preserves the balance constraint is performed before updating the remaining queue entries. Additionally, neighbors of the moved vertex that are also boundary vertices become eligible and are inserted into the queues. Each vertex is moved at most once.

To determine when to stop a highly localized search, Osipov and Sanders derive an *adaptive stopping criterion* based on a random-walk model. They assume that the most recent p gain values that did not yield an improvement are identically and independently distributed random variables with expectation μ and variance σ^2. The search terminates once

$$p\mu^2 > \alpha\sigma^2 + \beta,$$

where α and β are tuning parameters. They set $\alpha \in \{1, 4\}$, depending on algorithm configuration, and $\beta = \log(n)$.

Akhremtsev et al. [Akh+17] also use localized FM refinement in their n-level hypergraph partitioner KaHyPar. Beyond their hypergraph-specific contributions, they observe that repeatedly recomputing gain values for vertices touched by the localized passes on each level incurs a substantial running time overhead. Thus, they propose *gain caching*: the algorithm maintains a gain table of size $\Theta(nk)$, i.e., one entry per vertex and block, storing the gain of assigning the vertex to that block. The table is kept consistent during uncoarsening and after vertex moves by updating entries. Consequently, they require $\Theta(nk)$ additional memory.

Sanders and Schulz [SS11a] build on the idea of highly localized FM refinement and

develop *multi-try FM* for the standard multilevel scheme with $\Theta(\log(n))$ hierarchy levels. Multi-try FM repeatedly starts a localized FM search from a single boundary vertex. During a search, any vertex that becomes eligible for movement, i.e., is inserted into a priority queue, is considered *touched*. Touched vertices are unavailable for future searches during the same FM pass. Once the search terminates (using the adaptive stopping criterion described above), the algorithm initializes a new localized search from an *untouched* boundary vertex. This process continues until every boundary vertex has been touched. The intuition behind multi-try FM is that many small, localized searches can find improvements that a global FM pass may miss, since global variants may spend many negative-gain moves in unrelated regions before reaching an area where an improving sequence exists. Arguably even more important, multi-try FM paves the way towards parallelizing FM refinement. Global priority queues introduce sequential dependencies, whereas the highly localized searches can be executed concurrently on disjoint regions of the graph with substantially less synchronization.

Parallelization in the Shared-Memory Setting. Akhremtsev et al. [ASS20] parallelize localized multi-try FM by running many independent, highly localized searches in parallel and committing improving move sequences to the shared partition only in a *subsequent sequential* step. More precisely, at the beginning of each pass, all current boundary vertices are collected in a shared queue T. Each thread repeatedly extracts a boundary vertex $v \in T$ and performs a localized FM search around v as described above. During these searches, the shared partition remains unchanged. Instead, each thread simulates moves using thread-local data structures (e.g., a priority queue, local block-weight deltas, and a small hash table for locally changed vertex assignments). To prevent conflicting searches, threads acquire exclusive ownership of vertices they touch by atomically setting per-vertex flags via CAS operations on a shared array. Within a pass, flags are only set and never cleared, so each vertex participates in at most one localized search. Each thread records the best move sequence found by its searches (possibly empty if no improvement was found). Once T is empty, all recorded sequences are committed via a sequential post-processing step. While each non-empty recorded move sequence improves the partition in isolation, multiple sequences can conflict due to the balance constraint or because concurrent moves of adjacent vertices invalidate the computed gains. Let $M_i = \{B_{i1}, B_{i2}, \dots\}$ be the set of move sequences produced by thread i. The subsequent commit step sequentially processes the sequences in the order $M_1, M_2, \dots$, recomputes the gain values against the shared partition, and applies the feasible prefix of each sequence that yields the best actual improvement.

Gottesbüren et al. [Got+21a] propose several improvements to the shared-memory parallelization scheme of Akhremtsev et al. [ASS20]. First, they employ a shared gain cache to avoid repeatedly recomputing gain values from scratch, analogous to the gain cache used in KaHyPar [Akh+17]. To keep the shared gain cache consistent while searches perform thread-local tentative moves, each thread additionally maintains a small hash-table based gain cache that stores deltas to the shared gain cache. Second, instead of deferring all updates to the commit step, threads apply improving move sequences to the shared partition as soon as they are found, reducing both the number and severity of conflicts. However, they still record the applied moves and run a rollback-style post-processing step that recomputes the gains against the shared partition and applies the best prefix. In contrast to Akhremtsev et al. [ASS20], who do this sequentially, they have also parallelized this step by recomputing the exact gain of each move in the sequence in parallel and then using a parallel prefix sum to find the optimal prefix [Got+21a].

Parallelization in the Distributed Setting. Chevalier and Pellegrini [CP08] propose FM refinement for distributed-memory graph partitioning via *band graph refinement*. The core idea is to restrict refinement to a narrow region around the cut that is small enough to be replicated on every PE. To be more precise, given a distributed partitioned graph, they construct the *distributed band graph* by keeping only vertices whose distance to the cut is at most a small constant. They report good results already for a band width of 3. All remaining vertices are contracted into one vertex per block and PE. Since they only consider the bipartitioning case, this yields two contracted vertices per PE. Each PE then gathers a complete copy of the band graph and runs independent FM refinement (using the implementation from Scotch [PR96]) from slightly perturbed initial states. The best refined partition across all PEs is finally projected back to the distributed graph. This approach is only feasible if the partition cut (and consequently, the band graph) is much smaller than the distributed graph. Fortunately, this is the case for 2D and 3D meshes [CP08; LRT79].

Holtgrewe et al. [HSS10] follow a similar idea. In their approach, the number of blocks equals the number of PEs, and each PE exclusively owns one block. Bipartite refinement passes are scheduled via matchings in the quotient graph. To refine a selected pair of blocks, the corresponding PEs perform bounded breadth-first searches from their boundary vertices and exchange the discovered vertices so that both PEs obtain the local subgraph around the common border. They then execute a bipartite FM search restricted to this boundary region, and apply the better cut found by either PE.

3.4.4 Unconstrained Refinement

As with any local search heuristic, the effectiveness of refinement algorithms hinges on their ability to avoid becoming trapped in poor local minima. Simple greedy heuristics such as label propagation terminate once no single-vertex move yields a cut improvement. More complex heuristics, such as Kernighan-Lin [KL70] (KL) or Fiduccia-Mattheyses [FM82] (FM) refinement, can find improvements that require a sequence of moves and only net a cut improvement in combination. The multilevel scheme further mitigates this issue to some extent, since moving a single vertex on a coarse level translates into a coordinated move of large vertex sets on finer levels. However, most common refinement algorithms remain *constrained* by the balance constraint, i.e., they start from an initially balanced partition and enforce the balance constraint throughout the search. In contrast, several authors propose local search heuristics that can be classified as *unconstrained*. These heuristics allow temporary violations of the balance constraint and subsequently rebalance the partition.

Caldwell et al. [CKM00] propose a multi-pass heuristic for graphs with non-unit vertex weights that relaxes balance in the first pass so that every vertex is movable, then tightens the constraint over subsequent passes while performing greedy rebalancing between passes. Dutt and Theny [DT97] allow temporary imbalance if an *estimator* predicts a net improvement after a later rebalancing step, thereby enabling moves of heavy vertices or entire tightly connected clusters that would otherwise be blocked by the balance constraint. These schemes emerged when single-level algorithms were still prevalent in the field. With the rise of multilevel partitioners, refinement largely standardized around constrained variants of label propagation [RAK07], KL [KL70], FM [FM82] or flow-based algorithms [GHS22; HSS19; SS11a] that maintain the balance constraint at all times.

More recently, several works have revisited unconstrained refinement within multilevel frameworks. They found that permitting balance-violating moves, followed by an explicit rebalancing phase, can substantially improve solution quality, especially on graphs with a skewed degree distribution, such as complex networks.

Jet. Gilbert et al. [Gil+24] introduce *Jet*, a GPU-oriented refinement algorithm enabling wide synchronous parallelism. The Jet algorithm first constructs a set M of tentative *move candidates*. For a vertex v in block V_i, let $\text{conn}(v, V')$ be the summed weight of edges between v and vertices in block V'. Let $V_j = \arg\max_{V'} \text{conn}(v, V')$ be the block with the strongest connection to v. Vertex v enters M if

$$\text{conn}(v, V_j) - \text{conn}(v, V_i) \geq -\lfloor \tau \cdot \text{conn}(v, V_i) \rfloor,$$

where the *temperature* parameter $\tau \in [0, 1]$ controls the tolerance for negative gains. Note that $\tau = 0$ admits only non-negative gains, whereas $\tau = 1$ admits all vertices. In contrast to constrained refinement algorithms, M may include balance-violating moves. Candidates are subsequently gain-sorted, before positive-gain moves are applied in parallel and then locked to avoid oscillations in the next iteration. A separate rebalancing step repairs any balance violations. This process is iterated until several subsequent iterations fail to improve the cut.

Empirically, Jet achieves partition quality comparable to constrained FM refinement on regular, mesh-like graphs, while outperforming it on irregular graphs (e.g., with power-law degree distributions) [Gil+24; MGS]. Unlike FM, Jet does not require fine-grained locking and communication to enable parallelism, but instead relies on bulk-synchronous rounds. This maps well to GPU architectures [Gil+24], but also to distributed-memory [SS24] or deterministic shared-memory [KGM] implementations.

Unconstrained Label Propagation. Maas et al. [MGS] propose an unconstrained variant of label propagation refinement. In unconstrained LP, active vertices move greedily to the blocks with the strongest connection, while temporarily ignoring balance constraint violations. In contrast to Jet, only positive-gain moves are admitted. There is also no prioritization of moves. Vertices are processed in parallel in their natural order. After each round, a subsequent rebalancing step restores partition feasibility. During the first round, all vertices are active and considered for movement. In subsequent rounds, only vertices adjacent to moved vertices that did not themselves move in the previous round are activated. This pruning strategy is more restrictive than the standard *active set* strategy of label propagation, where vertices that moved in the previous round are still considered for movement if one of their neighbors moved afterwards. The authors note that this avoids oscillations and leads to faster convergence. The algorithm iterates until convergence, and finally restores the best feasible partition found during the iterations.

Unconstrained FM. Maas et al. [MGS] further propose an *unconstrained FM* algorithm based on parallel FM refinement [Got+21a], see also Section 3.4.3. The core idea is to allow moves during FM refinement that temporarily violate the balance constraint, while *penalizing* the gain of such moves using an estimate of the subsequent rebalancing cost. More precisely, for each block V_i, the algorithm initializes a set of exponentially spaced buckets $B_{i,s}$ for $s \geq 1$. A subset of the block's vertices v are assigned to bucket $B_{i,s}$ based on their relative gains, i.e., $s = \lceil \log_{3/2} g(v)/c(v) \rceil$. To quickly estimate the penalty of overloading V_i by moving vertex u into it, the algorithm identifies the smallest ℓ such that $\sum_{k=1}^{\ell} c(B_{i,k}) \geq c(V_i) + c(u) - L_{\max}$.

As in parallel constrained FM, each thread performs an independent search around a set of seed vertices, tracking tentative moves in thread-local data structures. Once a thread found an improvement (including the estimated rebalancing costs), the move sequence is committed to a global move sequence $\mathcal{M}$ and applied to the global partition, making the moves visible to the other threads. After all threads terminate, a subsequent rebalancing

algorithm computes a sequence of rebalancing moves $\mathcal{R}$. Finally, $\mathcal{M}$ and $\mathcal{R}$ are interleaved and the prefix yielding the best (balanced) partition is selected.

3.4.5 Other Refinement Algorithms

Helpful Sets. Diekmann et al. [DMP94] propose the *helpful set* technique to improve the cut between two blocks. Given a bipartition $\Pi = \{V_1, V_2\}$, they define the *helpfulness* of a vertex set $S \subseteq V_i$ (for $i \in \{1, 2\}$) as the decrease in edge cut obtained by moving all vertices of S to the opposite block. The algorithm maintains a threshold τ, initially set to $\tau = \texttt{cut}(\Pi)$, and searches for a set S whose helpfulness is at least τ. If such a set S with helpfulness $\tau' \geq \tau$ is found, the algorithm moves S to the other block and then searches on the opposite side for a set of the same size whose helpfulness is at least $-\tau' + 1$. If the second set exists, the vertices are also moved to the opposite block (restoring balance) and the threshold τ is doubled. Otherwise, the vertices of S are moved back to their original block and τ is halved. The process terminates once $\tau < 1$ or no set with positive helpfulness can be found.

Parallel Hill Climbing. LaSalle and Karypis [LK16] propose a parallel hill-climbing refinement algorithm for Mt-Metis. In Mt-Metis, each thread owns a subset of the graph's vertices. The algorithm maintains one priority queue T_t per thread t, which is initialized by the thread's owned boundary vertices. Each vertex v is keyed by a priority $p(v)$ that approximates its gain:

$$p(v) \coloneqq \frac{d_{\text{ext}}(v)}{\sqrt{\Delta(v)}} - d_{\text{int}}(v),$$

where $d_{\text{ext}}(v)$ and $d_{\text{int}}(v)$ denote the total weight of edges from v to neighbors in other blocks and to neighbors in its own block, respectively, and $\Delta(v)$ denotes the number of distinct blocks that contain neighbors of v. The algorithm then repeatedly extracts the highest ranked vertex v from T_t. If the maximum gain of v to a block V_i with $c(V_i) + c(v) \leq L_{\max}$ is positive, the vertex is moved immediately. Otherwise, the algorithm attempts to *construct a hill* around v, i.e., a set of vertices whose combined move yields a positive gain. To this end, a new empty hill H is initialized and v is inserted into a priority queue Q. Vertices from Q are added to H, and neighbors in the same block are inserted into Q. As soon as the combined gain is positive, H is moved. If the hill exceeds some maximum size, it is discarded. To reduce running time, each edge is only traversed once in each direction by any thread. Finally, to avoid oscillations (see Figure 3.2, left), refinement is performed in alternating passes in which moves are restricted to a single direction with respect to block IDs (from lower to higher IDs or vice versa).

Tabu Search. Rolland et al. [RPG96] apply the Tabu Search metaheuristic to balanced graph partitioning. In contrast to the refinement algorithms described above (i.e., greedy, KL, and FM refinement), where each vertex may only be moved once per refinement pass, Tabu Search allows vertices to move multiple times. To prevent oscillations, a moved vertex becomes *tabu* for a fixed time t, meaning that it may not be moved again for the next t vertex moves (algorithm iterations). More precisely, their algorithm performs $5n$ iterations and maintains a tabu list $T[\cdot]$ that stores, for each vertex, the last iteration in which it was moved. In iteration i, the algorithm selects a highest-gain vertex v that is either admissible under the tabu rule (i.e., $i - T[v] \geq t$, where $t = 3\sqrt{n}$), or improves the best feasible partition found so far. The selected vertex is then moved and $T[v]$ is set to i. Similar to KL and FM

refinement, Tabu Search can escape local minima since the selected move may worsen the current partition. In the end, the best feasible partition found during the search is returned.

Benlic and Hao [BH11] use Tabu Search in a multilevel graph partitioner. In their experiments, they report better solution quality than METIS and SCOTCH (which use variants of FM refinement), and they further improve upon many of the best previously reported cuts for instances from the Walshaw benchmark set [Wal23].

Diffusive Methods. Several authors have proposed diffusive methods to compute initial partitions from scratch or to refine existing ones [MMS09a; MMS09b; Pel07a]. The general idea works as follows [MMS09a]. An initial load is placed on a set of source vertices, e.g., vertices of a single block or center vertices of an initial partition. During each iteration, load spreads to neighboring vertices proportionally to the load difference across each edge. Additionally, there is a drain vector that removes a small amount of load from every non-source vertex and replenishes it at the sources. This process converges to a non-uniform steady state, which can be obtained by solving a linear system involving the graph Laplacian. The resulting load values indicate how well-connected each vertex is to the source set, and can be used to guide move decisions or partition assignment. Meyerhenke et al. [MMS09a] further present a diffusive method for fast local refinement around the boundary of a partition, avoiding solving the full linear system.

Flow-based Refinement Algorithms. Sanders and Schulz [SS11a] propose a flow-based refinement algorithm to improve the cut between two blocks (k-way partitions are refined via scheduling on pairs of blocks). The core idea is to extract a small *corridor* subgraph around the current cut and to compute an improving minimum s-t-cut within this corridor.

To construct the corridor, the algorithm performs two BFS searches starting from the boundary vertices of the two blocks, so that each BFS only expands to vertices of one block. All vertices reached by either search are added to the corridor. Each BFS stops once the total weight of the added vertices exceeds some prescribed limit (see below). All vertices outside the corridor are contracted into one vertex per side (called s and t), and the edges connecting s resp. t to corridor vertices are represented using infinite-capacity edges. Note that this construction differs from the band graphs discussed above, which are constructed by limiting the BFS depth (i.e., distance to the cut) rather than explicitly capping the total vertex weight collected from each block. Subsequently, a maximum s-t-flow is computed to obtain a minimum cut within the corridor graph. Balance is handled through the choice of the two BFS weight limits. In the simplest variant, the limits are chosen conservatively so that *every* (minimum) cut in the corridor induces a balanced partition of the full graph. In the *adaptive flow* variant, these limits are adjusted by a binary-search like procedure. If the current minimum cut would violate the balance constraint, the corridor is tightened (reducing the limits); otherwise, it is expanded to allow larger improvements. However, this is more expensive, as it requires solving multiple flow problems.

Gottesbüren et al. [Got+20] improve the running time of flow-refinement by building on the (Hyper)FlowCutter [GHW19; HS18]. If the current minimum cut violates balance, they avoid recomputing a flow from scratch. Instead, they enlarge the terminal vertex set of the smaller side by fixing all vertices on the smaller side to remain on that side, and additionally fix a single *piercing vertex* from the larger side into the smaller-side terminal set. Moving the piercing vertex to the smaller side forces the minimum cut to differ from the previous one, and the smaller side must strictly grow due to the additional terminals. The process terminates once a balanced minimum cut is found.

Gottesbüren et al. [GHS22] parallelize flow-based refinement in the shared-memory setting. There are two layers of parallelism: first, multiple block pairs can be refined in parallel. Second, the flow algorithm itself can be parallelized.

3.5 Multilevel Partitioning Frameworks

Next, we survey publicly available multilevel graph partitioning frameworks. We first discuss sequential partitioners, and then turn to shared-memory and finally distributed-memory partitioners. Note that there also exists a vast landscape of hypergraph partitioners (see Section 3.6), such as hMetis [Kar+99], PaToH [ÇA99], and KaHyPar [Sch+22], which can naturally also be used to partition graphs. Since techniques tailored towards hypergraph partitioning specifically are outside the scope of this thesis, we only discuss hypergraph partitioners that are particularly relevant to our contributions and experimental evaluations here.

3.5.1 Sequential Graph Partitioners

Over the last three decades, numerous sequential multilevel graph partitioners have been proposed.

Chaco. Hendrickson and Leland [HL95a] were the first to apply the multilevel scheme in its modern form to graph partitioning. Their seminal paper was accompanied by Chaco [HL95a], the first multilevel partitioning framework. Chaco performs coarsening by repeatedly contracting the edges of a maximal matching until the graph is sufficiently small (in their experiments, they coarsen down to 200 vertices). Matchings are constructed greedily and do not take edge weights into account. On the coarsest graph, an initial partition is computed via spectral methods and subsequently improved using FM refinement. The same refinement algorithm is used during uncoarsening. One multilevel cycle computes 2, 4, or 8 blocks (using 1, 2, or 3 eigenvectors during initial partitioning), so that a general k-way partition is obtained through recursive partitioning.

METIS. During the late 1990s, Karypis and Kumar [KK95b; KK97; KK98a; KK98c] introduced the sequential multilevel graph partitioner Metis, which remains one of the most widely used partitioners in practice. In its multilevel pipeline, Metis coarsens the graph via heavy-edge matching [KK95b; KK97]. To improve the performance on graphs with a skewed-degree distribution, later versions of Metis adopted the two-hop matching scheme of Mt-Metis [Kar13; LaS+15]. Early versions computed k-way partitions exclusively via recursive bipartitioning on the input graph [KK95b; KK97], whereas later versions also support direct k-way initial partitioning and k-way refinement [KK98c]. In the latter case, an initial k-way partition on the coarsest level is obtained using the previous recursive bipartitioning implementation. During uncoarsening, partitions are refined using a tuned k-way extension of the FM algorithm [KK98c]. Notably, Metis also supports multi-constraint partitioning [KK98b].

Scotch. Pellegrini and Roman [BCP18; Pel07b; PR96] propose Scotch, a graph partitioner based on recursive bipartitioning. The published descriptions provide only limited algorithmic detail. The coarsening phase is implemented via heavy-edge matching [BCP18]. For initial bipartitions, the authors mention *random* and *greedy* approaches [PR96]. Refinement is

based on the FM algorithm, with improvements to the bucket-based priority queue required for linear time. Experiments on 5 different graphs indicate similar solution quality to Chaco and Metis. Scotch also supports multi-constraint partitioning [BCP18].

KaSPar. Osipov and Sanders [OS10] present the n-level partitioner KaSPar, which contracts only a single edge per hierarchy level, yielding a coarsening hierarchy with $\Theta(n)$ levels. At each level, the next edge to be contracted is chosen according to the expansion$^\star(\cdot)$ rating function stated in Section 3.2.1. Initial partitioning is delegated to Scotch [PR96]. During uncoarsening, KaSPar refines the projected partition using a highly localized FM search around the uncontracted edge.

Party. Monien et al. [MPD00; MS04] propose Party, a multilevel partitioner based on recursive bipartitioning. Its coarsening phase is matching-based and builds on the linear-time 1/2-approximation algorithm due to Preis [Pre99], but additionally restricts matches to pairs of vertices whose combined weight does not exceed the sum of the lightest and the heaviest vertex in the graph. Coarsening proceeds aggressively until the graph is reduced to only two vertices. For refinement, Party introduces the helpful-set technique (described in Section 3.4.5). It offers limited shared-memory parallelism by performing independent bipartitions that arise during recursive bipartitioning in parallel. However, the coarsening or refinement algorithms are not parallelized. In their experiments, Monien et al. report that Party typically computes better partitions than METIS and Jostle, but is also slower.

KaHIP. Sanders and Schulz [MSS14; Sch13; SS11a; SS13] develop the KaHIP partitioning framework, which implements a broad variety of techniques for all phases of the multilevel scheme. For coarsening, KaHIP supports both matching-based approaches [SS11a], using random matching or the GPA algorithm with the expansion$^{\star 2}(\cdot)$ or innerOuter$(\cdot)$ rating, and cluster-based approaches [MSS14], based on size-constrained label propagation. Once the graph is sufficiently small, KaHIP computes an initial k-way partition via recursive bipartitioning. This recursive scheme itself follows a multilevel cycle and uses Greedy Graph Growing on the coarsest level [KK95b; Sch13]. For the uncoarsening phase, KaHIP offers several refinement algorithms, including greedy refinement (via size-constrained label propagation), multi-try 2-way FM refinement on block pairs, k-way FM refinement, flow-based refinement [SS11a], and ILP-based refinement [HNS18]. To further improve solution quality, KaHIP also implements V-, F-, and W-cycles, and a memetic meta-heuristic [SS12].

Given the large space of algorithmic choices and tuning parameters, KaHIP provides a number of configuration presets that trade solution quality against running time. In our experiments, we only use the *Fastsocial* configuration, which combines size-constrained label propagation for coarsening (up to 10 iterations), multilevel recursive bipartitioning for initial partitioning (size-constrained label propagation for coarsening, BFS-style bipartitioning, and 2-way FM during uncoarsening), and size-constrained label propagation for refinement (up to 25 iterations). Stronger configurations additionally use techniques such as ensemble clustering, stronger refinement (see above), and perform multiple V-cycles, but at substantially higher running time. On social networks, Meyerhenke et al. [MSS14] report that KaHIP in the Fastsocial configuration computes cuts that are by 3.9% smaller than those of METIS, while being 3.8× slower. Stronger configurations compute up to 16.7% smaller cuts than METIS, at the cost of up to 741.0× higher running time. The strongest configuration of KaHIP achieves the best average solution quality among all publicly available multilevel partitioners in Ref. [Got+24a], but is expensive (a median edge-cut improvement of 1% over

the strongest configuration of Mt-KaHyPar at 38.7× higher running time). However, we note that this conclusion is benchmark dependent. In particular, Ref. [Sch+22] benchmarks on a different set of graphs and observes slightly better quality for KaHyPar, especially on complex networks.

Beyond standard vertex partitioning, KaHIP solves several related partitioning problems. It can compute edge partitions [Sch+19], which assigns edges (rather than vertices) to blocks, and minimizes the number of *vertex replications* (number of blocks containing edges incident to a vertex). It further includes algorithms for computing vertex separators [SS16a], and algorithms for perfectly-balanced partitioning [SS13] (i.e., standard partitioning with $\varepsilon = 0\%$).

KaHyPar. Schlag et al. [Sch+22; Sch20] present the (hyper)graph partitioner KaHyPar, which also introduces several techniques that are of independent interest in the graph partitioning setting. KaHyPar follows the n-level scheme, contracting exactly one vertex pair per level [Sch+16]. To guide coarsening, it first computes a Louvain clustering of the input (hyper)graph and then restricts contractions to vertices within the same cluster (i.e., *community*) [HS17]. Initial partitioning is performed via multilevel recursive bipartitioning [Akh+17], where each bipartition is computed with a pool of heuristics, including random bipartitioning, BFS-style (Hyper)graph Growing, Greedy (Hyper)graph Growing, and size-constrained label propagation. To ensure the feasibility of the bipartitions, KaHyPar adapts the ε used during bipartitioning (as discussed in Section 3.1), and it further *prepacks* heavy vertices before bipartitioning [HMS21]. During uncoarsening, KaHyPar uses highly localized FM searches around the uncontracted vertex pairs. Its FM implementation maintains a gain cache to avoid repeatedly recomputing gains from scratch. In addition, flow-based refinement is used [Got+20; HSS19]. In their experimental evaluation, Schlag et al. [Sch+22] show that KaHyPar computes very high-quality partitions, which are slightly better or on-par with the best configuration of KaHIP, depending on benchmark set, using a comparable amount of time.

3.5.2 Shared-Memory Parallel Graph Partitioners

Next, we survey shared-memory parallel multilevel graph partitioners. Over the last years, state-of-the-art shared-memory partitioners have matured to achieve solution qualities that are on-par with the best sequential partitioners, while substantially reducing running time on modern multi-core CPUs.

Mt-METIS. LaSalle and Karypis [LaS+15; LK13; LK16] propose Mt-METIS, a shared-memory parallelization of METIS. Like METIS, Mt-METIS coarsens the graph using heavy-edge matching, and augments it with two-hop matching to ensure progress on graphs that tend to admit only small maximum matchings on coarser levels (e.g., graphs with a skewed-degree distribution such as web and social graphs) [LaS+15]. Initial partitioning is delegated to the sequential METIS algorithm. The first version only employs greedy refinement. LaSalle and Karypis report that Mt-METIS achieves higher speedups over METIS than the distributed partitioners ParMETIS and Pt-Scotch, while observing comparable partition quality across all four partitioners. In subsequent work, they introduce a parallel hill-climbing refinement algorithm [LK16], which can escape local minima to some extent. In their experiments, they report a reduction in mean edge cut by 6.3% over greedy refinement, at an average increase in running time of 41%.

Mt-KaHIP. Akhremtsev et al. [Akh19; ASS20] present Mt-KaHIP, a shared-memory parallelization of the KaHIP partitioner. Coarsening is based on a shared-memory implementation of size-constrained label propagation, while initial partitioning is delegated to the sequential KaHIP. To this end, each thread obtains a copy of the coarsest graph and runs KaHIP with a different random seed. The best partition over all threads is selected. During uncoarsening, Mt-KaHIP first applies a parallel greedy refinement based on size-constrained label propagation, followed by parallelized localized FM refinement. In their experimental evaluation, Akhremtsev et al. [ASS20] report that Mt-KaHIP achieves small edge cuts than Mt-METIS, ParHIP (fast configuration), and KaHIP (fastsocial configuration) by 18.7%, 12.2%, and 7.2% on average, respectively. While Mt-KaHIP is more than twice as fast as ParHIP, it is also more than 4× slower than Mt-METIS on average. However, Mt-KaHIP still computes higher-quality partitions when giving Mt-METIS the same time budget for independent repetitions.

Mt-KaHyPar. Gottesbüren et al. [Got+24a; Got23; Heu22] propose Mt-KaHyPar, a shared-memory parallelization of KaHyPar. Although Mt-KaHyPar was originally developed as a hypergraph partitioner, many of its algorithmic contributions are also relevant to graph partitioning. In addition, it provides an explicit graph partitioning mode that reduces overheads due to hypergraph-specific data structures [Got+24a]. Mt-KaHyPar performs coarsening using a single round of size-constrained label propagation [Got+21a]. Initial partitioning follows a multilevel recursive bipartitioning scheme and employs a pool of bipartitioning heuristics with an adaptive pool selection technique (see Section 3.3) [Got+22]. For refinement, it combines size-constrained label propagation with parallel (unconstrained [MGS]) multi-try FM and paralle flow-based refinement [GHS22]. The framework supports both the classic $\Theta(\log(n))$-level multilevel hierarchy and an $\Theta(n)$-level variant [Got+22], and it additionally offers a deterministic mode [GH22; KGM]. Extensive experiments suggest that Mt-KaHyPar is the highest-quality publicly available hypergraph partitioner [Got+24a]. For graphs, Ref. [Got+24a] reports that the strongest KaHIP attains about 1% smaller edge cuts on average. However, subsequent improvements by unconstrained refinement [MGS] likely reverse this gap.

3.5.3 Distributed Graph Partitioners

Below, we review multilevel frameworks for distributed memory partitioning. All of the partitioners discussed here split the input graph across p processing elements (PE), such that each PE only stores a local subgraph of the global input graph. A vertex contained in the local subgraph of a PE is *owned* by that PE and may be adjacent to vertices owned by other PEs.

Jostle. Walshaw and Cross [WC00] present the distributed-memory parallel multilevel graph partitioner Jostle, which computes a partition with one block per PE. For coarsening, Jostle uses a dual-phase parallel variant of heavy-edge matching [KK95b]. In the first phase, vertices are only matched with other local vertices (i.e., owned by the same PE). Only in the second phase, remaining unmatched vertices are matched with non-local neighbors [WCE97]. There is no separate initial partitioning step. Instead, Jostle continues coarsening until the coarsest graph contains one vertex per block (PE), which directly induces an initial partition. To improve scalability once the graph becomes sufficiently small, Jostle copies the full graph onto each PE and continues sequentially. During uncoarsening, Walshaw and Cross [WC00] employ a multilevel balancing schedule that permits larger imbalance on coarse

levels and gradually tightens the balance constraint until the final constraint is reached on the input level. To rebalance between levels, they propose a diffusion-inspired approach that computes a balancing flow between blocks. Refinement then proceeds by exchanging vertices so that the prescribed flow is satisfied while improving the cut. To this end, a sequential KL-style algorithm is used on the boundary area between pairs of blocks, with multiple pairs being processed in parallel.

ParMETIS. Karypis and Kumar [KK97; KK99] introduce ParMETIS, a distributed-memory parallelization of the sequential METIS algorithm. ParMETIS coarsens the graph using a distributed variant of heavy-edge matching. Once the graph becomes too small for the current number of PEs, it is *folded* onto half of the processors by redistributing the graph accordingly [KK97]. This process is repeated recursively. Initial partitioning works via recursive bipartitioning. At the coarsest level, each PE holds a replica of the coarsest graph and independently computes deterministic bipartitions, with each PE following a single path in the bipartitioning tree. During uncoarsening, ParMETIS applies a distributed implementation of greedy refinement, see Section 3.4.1.

PT-Scotch. Chevalier and Pellegrini [CP08; Pel09] present Pt-Scotch, a distributed-memory partitioner based on recursive bipartitioning. A primary use case of Pt-Scotch is computing graph orderings via nested dissection. In its multilevel scheme, coarsening is performed via heavy-edge matching. To improve parallel efficiency on coarse levels, Pt-Scotch also applies a *folding* step. Once the coarse graph has fewer than 100 vertices per PE, the distributed graph is replicated and redistributed across two disjoint PE groups of half the original size. Folding is applied recursively until each PE holds a full copy of the coarsest graph. Each PE then computes an initial bipartition using the Scotch algorithm. During uncoarsening, Pt-Scotch selects the better of the candidate partitions due to the folding process, and refines each level using sequential FM restricted to the band graph around the current cut (see Section 3.4.3). Reported experiments indicate that Pt-Scotch is roughly 4× slower than ParMETIS, but computes higher-quality orderings (comparable to those of Scotch) that can reduce end-to-end running times, for example in parallel sparse matrix factorizations.

KaPPa. Holtgrewe et al. [HSS10] engineer the distributed-memory partitioner KaPPa, which always computes a partition into $k = p$ blocks. During coarsening, KaPPa employs the parallel matching scheme of Manne and Bisseling [MB07], where each PE first computes a matching on its local subgraph (in their case, using the GPA algorithm [MS07]), and subsequently matches edges that connect vertices on different PEs (which form the *gap graph*). Coarsening continues until the graph has only $\Theta(n/k^2)$ vertices per PE. For initial partitioning, the coarsest graph is replicated on every PE and partitioned sequentially using METIS or Scotch. Each PE uses a different random seed, and the best initial partition is selected. The graph is then redistributed between PEs, so that each PE stores all vertices that belong to one block. Refinement uses the 2-way FM algorithm, which is applied to the boundary area between block pairs. For a selected block pair, the boundary region is gathered onto the corresponding PEs and refined sequentially. However, multiple block pairs are refined in parallel. Their experiments show that KaPPa generally produces higher-quality partitions than ParMETIS, whose cuts are on average 18%–30% larger, depending on the chosen KaPPa configuration. However, ParMETIS is also more than 10× faster than even the fastest KaPPa configuration. They report scalability up to at least 1 024 PEs.

ParHIP. Meyerhenke et al. [MSS17] present a distributed-memory parallelization of KaHIP, called ParHIP. ParHIP uses a distributed implementation of size-constrained label propagation for both coarsening and refinement. In both phases, the size or balance constraints are not strictly enforced, see Section 3.2.2 and Section 3.4.1 for details. Initial partitioning is delegated to a sequential algorithm, depending on the chosen algorithm configuration, after replicating the coarsest graph on each PE. ParHIP offers two configurations that trade solution quality for running time. The *Fast* configuration uses KaHIP for initial partitioning and performs two V-cycles. The *Eco* configuration uses the evolutionary mode [SS12] of KaHIP for initial partitioning with a time budget of $2^{11}\,\mathrm{s}/p$, where p denotes the number of PEs used for partitioning, and performs five V-cycles.

In their experiments, ParHIP-Eco generally yields the best partitions, but is also much slower than ParHIP-Fast (on all instances) and ParMETIS (on mesh-like instances). Compared to ParMETIS, ParHIP achieves 19.2% and 27.4% smaller edge cuts on average with its Fast and Eco configuration, respectively. ParMETIS is more than 5× faster than ParHIP-Fast on mesh-like graphs, but more than 2× slower on social and web graphs. This is likely because ParMETIS uses matching based coarsening, which tends to get stuck on graphs with a skewed degree distribution [LaS+15]. They further report scalability results on synthetic geometric graphs, Delaunay triangulations, and web graphs, including weak scalability up to at least 2 048 PEs and strong scalability up to 128 PEs on most real-world graphs (up to 1 024 PEs on one graph). To the best of our knowledge, ParHIP-Eco achieves the highest-quality partitions in the distributed setting, but lags behind high-quality shared-memory partitioners [Got+24a].

3.6 Other Computational Models and Problem Variations

The textbook balanced graph partitioning problem does not capture all aspects relevant to real-world applications. Therefore, additional aspects, such as multiple constraints [ÇA99; KK98b; SKK00; Slo+17; SMR14] or objectives [Slo+17; SMR14], acyclic graphs [Her+17; MPS17; MPS20; Pop+21], matrix partitioning, process mapping, or various ways of modeling communication costs have been considered. We provide a brief overview below.

Graph Partitioning with Multiple Constraints. In multi-constraint graph partitioning, vertices have $\ell \geq 1$ weight components, i.e., c is a vector-valued function $c : V \to \mathbb{N}^\ell$. Accordingly, $L_{\max} \in \mathbb{N}^\ell$, and a partition is *balanced* if and only if each block satisfies all constraints component-wise. Multilevel partitioners that support multi-constraint partitioning include METIS [KK98b], PaToH [ACU08], and Scotch [BCP18].

Karypis and Kumar [KK98b] extend METIS to multi-constraint vertex weights. During coarsening, they continue to compute a heavy-edge matching, but resolve tie situations using the *balanced-edge* criterion, which prefers contractions whose resulting weight vectors are more uniform across dimensions. For initial partitioning via multilevel bipartitioning, they generalize Greedy Graph Growing. Starting from $V_1 = V$ and $V_2 = \emptyset$, the algorithm maintains ℓ priority queues keyed by gain, one per weight dimension. Each vertex is inserted into the queue corresponding to its largest weight component. In each step, the algorithm identifies the currently heaviest (normalized) component of $c(V_1)$ and moves the highest-gain vertex from the corresponding queue to V_2. This process terminates once at least one component of $c(V_2)$ exceeds half of the total weight in that component. Refinement during uncoarsening is based on a multi-constraint variant of FM refinement. Instead of one queue per block, the algorithm maintains 2ℓ queues (one per block and weight dimension), organized

analogously but initialized only with boundary vertices. Moves are selected from the queue associated with the currently heaviest (normalized) weight component. If the bipartition is imbalanced, a rebalancing pass using the same selection principle but initialized with all vertices is performed. For k-way refinement, they employ greedy boundary refinement, which moves boundary vertices to the block maximizing gain while preserving balance.

Both Aykanat et al. [ACU08] and Barat et al. [BCP18] limit the maximum weight of coarse vertices in any dimension during coarsening. The latter additionally repeat initial partitioning on finer levels if the current initial partition is imbalanced. During uncoarsening, they use variants of FM refinement that only allow balance-preserving moves.

Acyclic Graph Partitioning. If the input graph is a directed acyclic graph (DAG), one is often interested in partitions with an acyclic quotient graph. Moreira et al. [MPS17] show that computing a perfectly balanced graph partition remains NP-complete even under this acyclicity constraint, and that no constant-factor approximations exist for $k \geq 3$. Herrmann et al. [Her+17; Her+19] and Moreira et al. [MPS17; MPS20] propose multilevel heuristics for DAG partitioning. To prevent cycles forming during coarsening, these approaches either start from an acyclic partition and restrict contractions to intra-block edges (effectively performing a V-cycle) [MPS17; MPS20], or integrate the acyclicity constraint directly into the clustering process [Her+17; Her+19]. Initial partitions are computed by splitting vertices along a topological order, or by first computing potentially cyclic bipartitions and subsequently restoring acyclicity [Her+19]. The latter strategy is conceptually related to the unconstrained refinement techniques reviewed in Section 3.4.4, in that it temporarily relaxes a constraint to obtain a good cut, and enforces feasibility only in a post-processing step. Refinement is typically based on FM, but further restricted to preserve acyclicity after each move. To the best of our knowledge, unconstrained refinement methods have not yet been explored for this setting, but appear promising since the additional constraint further restricts the solution landscape. The approaches by Herrmann et al. [Her+19] yield higher-quality partitions in only a fraction of the running time compared to the evolutionary approach by Moreira et al. [MPS20]. Popp et al. [Pop+21] extend these techniques to directed acyclic hypergraph partitioning, where each hyperedge has a single *head* pin while all other pins are *tails*.

Hypergraph Partitioning. Hypergraphs generalize graphs by allowing edges (called *hyperedges*) to connect an arbitrary number of vertices, referred to as *pins* of the hyperedges. In *hypergraph partitioning*, the goal is to divide the vertex set into k blocks subject to a balance constraint. Common objective functions are the *(hyperedge) cut*, i.e., the total weight of all hyperedges whose pins lie in more than one block, and the *connectivity metric*, which additionally scales the contribution of a cut hyperedge e by $(\lambda(e) - 1)$, where $\lambda(e)$ is the number of blocks which contain pins of e. Note that if the hypergraph is a graph (i.e., every hyperedge has exactly two pins), both objectives coincide with the usual edge-cut objective from graph partitioning. Thus, hypergraph partitioning is strictly more general, and, in general, cannot be reduced to graph partitioning [IWW93]. Nevertheless, many heuristics from graph partitioning carry over successfully, although gain computation is typically more expensive due to the larger hyperedges. Successful multilevel hypergraph partitioners include the sequential hMetis [Kar+99], PaToH [ÇA99], and KaHyPar [Sch+22], the shared-memory parallel partitioners ParPaToH [Çat+12] and Mt-KaHyPar [Got+24a], and the distributed-memory partitioner Zoltan [Dev+06]. According to Ref. [Got+24a], Mt-KaHyPar computes the highest-quality partitions on average.

Edge Partitioning. Edge partitioning asks for a partition of the edges (rather than the vertices) into k blocks, subject to a balance constraint. The objective is usually to minimize the *vertex cut*, given by $\sum_{v \in V}(I(v) - 1)$, where $I(v)$ denotes the number of blocks that contain at least one edge incident to v. Note the similarity to the connectivity metric in hypergraph partitioning. This problem variant arises naturally in vertex-centric graph processing frameworks, where normal vertex partitioning can lead to load imbalance and communication overhead on skewed, power-law graphs [Mal+10].

There is a variety of single-level approaches to solve the edge partitioning problem, e.g., Ref. [BLV14; Han+19; May+18; Rah+14; Zha+17]. Li et al. [Li+17] reduce edge partitioning to vertex partitioning by constructing a (lossy) gadget graph that over-approximates the vertex cut objective. Moreover, edge partitioning can be reduced to hypergraph partitioning via the following construction. Create one hypergraph vertex for each graph edge, and for each original vertex v, add a hyperedge containing exactly the hypergraph vertices corresponding to the edges incident to v. Under this transformation, minimizing the connectivity metric in this hypergraph corresponds directly to minimizing the vertex cut in the original graph [Li+17]. With this transformation, high-quality multilevel hypergraph partitioners obtain higher quality than many techniques developed specifically for edge partitioning [Sch+19].

3.6.1 GPU-based Graph Partitioning

Due to their high computational power, modern GPUs have become an important tool for accelerating data-parallel applications. However, due to the highly irregular structure of graphs, it remains challenging to design graph algorithms that efficiently utilize the architecture of modern GPUs.

Spectral Graph Partitioning. The availability of efficient eigensolvers on GPUs has led to a recent re-emergence of spectral techniques for graph partitioning on GPU systems [ABR20; Ace+21; NM16]. These techniques were first developed by Donath and Hoffman [DH72; DH73] and Fiedler [Fie75] to compute graph bisections in the 1970s. Subsequently, these techniques have been improved [Bop87; BS93; HL95b; PSL90; Sim91] and extended to partition a graph into more than two blocks using multiple eigenvectors [AY95; HL95b].

The core idea of spectral graph partitioning is as follows [Fie75]. Let $G = (V = [n], E)$ be an undirected and unweighted graph with Laplacian matrix $L \in \mathbb{R}^{n \times n}$ defined by

$$L_{ij} = \begin{cases} d(i) & \text{if } i = j, \\ -1 & \text{if } i \neq j \text{ and } (i,j) \in E, \\ 0 & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Consider a bisection $\Pi = \{S, V \setminus S\}$ of V where $|S| = |V \setminus S|$, encoded by a vector $x \in \{-1, 1\}^n$ with $x_i = -1$ for $i \in S$ and $x_i = 1$ for $i \in V \setminus S$. Note that we have $x^T \mathbf{1} = 0$, where $\mathbf{1} = (1, \ldots, 1)^T$. The cut size can be written as the quadratic form

$$\texttt{cut}(\Pi) = \frac{1}{4} \sum_{(i,j) \in E} (x_i - x_j)^2 = \frac{1}{4} x^T L x.$$

Fiedler showed that the second smallest eigenvalue λ_2 of L (the *Fiedler value*) admits the characterization

$$\lambda_2 = \min_{x \neq 0,\, x^T \mathbf{1} = 0} \frac{x^T L x}{x^T x},$$

and that the minimum occurs exactly when x is an eigenvector corresponding to λ_2 (a *Fiedler vector*). Spectral partitioning therefore relaxes the discrete constraint $x \in \{-1, 1\}^n$ to $x \in \mathbb{R}^n$ with $x^T\mathbf{1} = 0$, computes a Fiedler vector x, and rounds it to a discrete bisection, e.g.,

$$S = \{i \in V \mid x_i < \tau\},$$

for a tunable threshold τ (commonly $\tau = 0$, or chosen according to balance).

GPU-based Spectral Graph Partitioners. Naumov and Moon [NM16] present an implementation of spectral graph partitioning for single GPU systems as part of the nvGRAPH library. More recently, Acer et al. [ABR20; Ace+21] proposed the multi-GPU partitioner Sphynx. Both partitioners precondition the matrix and use the LOBPCG [Kny01] eigenvalue solver. The eigenvectors are subsequently used to embed the graph into a multidimensional coordinate space, which is then used to derive a partition of the graph. In nvGRAPH, this is achieved by using a k-means clustering algorithm on the embedded graph, whereas Sphynx uses the geometric graph partitioner Multi-Jagged [Dev+16] which supports multi-GPU systems. According to Ref. [Ace+21], the approach by Acer et al. outperforms nvGRAPH across partition balance, cut sizes and running times (when run on a single GPU system). Compared against ParMETIS [KK99], it obtains approx. 20% resp. 70% larger cuts on graphs featuring a regular resp. irregular degree distribution. However, the authors report a significant speedup of approx. 19× for irregular graphs using 24 GPUs over ParMETIS using 168 MPI processes across four compute nodes. Additionally, Acer et al. report the influence of several matrix preconditioners on different classes of graphs.

GPU-based Multilevel Graph Partitioners. To the best of our knowledge, the earliest multilevel graph partitioning algorithms tailored to GPUs are due to Goodarzi et al. [GBG16]. They implement coarsening using an algorithm akin to the heavy-edge matching employed by METIS [KK98a]. Once small enough, the coarsest graph is then transferred to the CPU and initially partitioned using Mt-METIS [LK13]. For refinement, vertices are distributed across GPU threads, each of which determines the gain-maximizing move for its assigned vertices. To avoid conflicting moves that worsen the edge cut in combination, refinement proceeds in alternating rounds that restrict moves to blocks with increasing or decreasing block IDs. For each target block, candidate moves are collected in a global buffer, sorted by gain, and finally applied in order, selecting the highest-rated moves first. Their follow-up paper [Goo+19] introduces several improvements. First, they use Warp Segmentation [KGB15] to accelerate the heavy-edge matching computation during coarsening. Second, they move initial partitioning to the GPU and compute it via a greedy region-growing procedure. For refinement, boundary vertices are again distributed across GPU threads, and each thread determines the best target block for its assigned vertices, inserting the candidate moves into a global buffer. The algorithm then selects the ℓ highest rated candidates from this buffer (for a small constant ℓ). Because these moves might conflict, it evaluates all 2^ℓ combinations in parallel (distributed across thread groups), chooses the best combination, and applies it to the partition. Overall, their GPU-based approach is reported to be around 1.9× faster than Mt-METIS (benchmarked on a Nvidia Kepler K40 GPU resp. Xeon E5540 CPU), while producing slightly worse partition quality on average.

Auer and Bisseling [AB12] present a fine-grained shared-memory parallel algorithm for graph coarsening and apply this algorithm in the context of graph clustering to obtain a fast

greedy heuristic for maximizing modularity in weighted undirected graphs. The algorithm is suitable for both multi-core CPUs and GPUs.

Gilbert et al. [Gil+21] present performance-portable graph coarsening algorithms. In particular, the authors study a GPU parallelization of the heavy edge coarsening method. They evaluate their coarsening method using a multilevel spectral graph partitioning algorithm as the primary use case. In subsequent work [Gil+24], they extend their approach to a fully GPU-based multilevel graph partitioner called JET by introducing unconstrained refinement techniques for the refinement phase. Somewhat surprisingly, their unconstrained refinement method outperforms the arguably more sophisticated FM algorithm that is commonly used in high-quality CPU-based multilevel partitioners, making the technique of independent interest. We therefore review it in Section 3.4.4. To the best of our knowledge, JET is the first GPU-based graph partitioner that is highly competitive with CPU-based partitioners in both running time and solution quality.

3.6.2 Single-Level and Streaming Techniques

Beyond multilevel partitioners, a wide range of single-level techniques for graph partitioning have been proposed. In addition, *streaming* partitioners avoid storing the entire graph in memory and instead assign vertices to blocks on-the-fly while streaming the graph. Since these approaches typically yield partitions that are not competitive with multilevel partitioners, we restrict our review to the algorithms included in our experimental evaluations. A broader overview focusing on recent developments is available in Ref. [Çat+23].

Single-Level Label Propagation. Slota et al. [Slo+17; SMR14] propose the parallel single-level graph partitioners PuLP (shared-memory) and XtraPuLP (distributed-memory), both of which are based on size-constrained variants of label propagation. In the original setting, graphs are unweighted (i.e., $c(\cdot) = \omega(\cdot) = 1$ and $n = c(V)$).

PuLP proceeds in two or three phases. First, it computes an initial k-way partition by assigning vertices uniformly at random to the k blocks and then applying a degree-weighted label propagation variant to refine it. The label propagation algorithm roughly follows Algorithm 1, but sets

$$\texttt{score}(u, v) := d(v)$$

in line 8. Moreover, it enforces a *lower* size constraint $L_{\min} := (1-\varepsilon)\frac{n}{k}$, so that a vertex may be moved from its current block only if the remaining block weight stays at least $L_{\min}$. An upper bound is not enforced. The purpose of this phase is to form clusters around high-degree vertex hubs. Second, PuLP alternates between balancing and refining label propagation rounds. In a balancing round, moves are restricted to non-overloaded target blocks, and scores are biased toward lighter blocks by setting

$$\texttt{score}(u, v) := d(v) \cdot \left(\frac{L_{\max}}{c(\mathcal{C}[v])} - 1 \right).$$

In a refining pass, the algorithm uses the standard (unbiased) scoring function (i.e., $\texttt{score}(\cdot) = 1$) while enforcing the upper size constraint $U := L_{\max}$. Lastly, PuLP uses similar label propagation variants to balance the edges per block and minimize the maximum per-block edge cut. This phase can be disabled (which we always do when comparing against PuLP).

XtraPuLP extends the same overall approach to the distributed-memory setting by interleaving label propagation rounds with communication rounds that synchronize updated

block assignments of ghost vertices. To avoid communication within a label propagation round, it approximates global block weights using local information, see Section 3.2.2.

Buffered Streaming Graph Partitioner. Faraj and Schulz [FS22] introduce the buffered streaming graph partitioner HeiStream. Given a constant batch size β, the algorithm loads the next β vertices of the input graph (together with their incident edges) into memory and constructs a model graph $\mathcal{B}$. This model graph contains the subgraph induced by the β batch vertices, as well as k auxiliary vertices that represent the existing partial partition, obtained by contracting all previously processed vertices of each block into a single vertex. Vertices that appear later in the input graph are either ignored (simple model) or approximated by contracting such vertices to a random neighbor in $\mathcal{B}$ (advanced model). The model $\mathcal{B}$ is then partitioned using a multilevel scheme whose coarsening and refinement phases employ size-constrained label propagation. For initial partitioning, HeiStream employs Fennel [Tso+14], which greedily minimizes a combined objective capturing both edge cut and imbalance. After partitioning, the current model $\mathcal{B}$ is discarded and the next β vertices are read to build the next model graph. Experimental results indicate that HeiStream yields significantly smaller edge cuts than earlier streaming partitioners. In particular, it yields $4\times$ smaller edge cuts than plain Fennel on average.

More recently, Hajidehi et al. [HSS24] present the buffered streaming partitioner Cuttana, which combines two orthogonal ideas to improve partition quality. First, Cuttana does not assign vertices immediately upon arrival, but instead maintains a constant-size priority queue of unassigned vertices. Whenever the queue is full, it evicts and assigns the vertex with the highest *buffer score*, which estimates how informative the current partial partition is for that vertex (i.e., counts already assigned neighbors, with a bias for high-degree vertices). Intuitively, this mechanism partially compensates for unfavorable input orders by allowing limited reordering within the queue. Second, Cuttana introduces *subpartitioning*. Each vertex is assigned not only to one of the k final blocks, but additionally to one of $k' \coloneqq \ell k$ subblocks for some $\ell > 1$. During streaming, the algorithm constructs the quotient graph induced by the k'-way subpartition in memory and subsequently employs greedy refinement on this (much) smaller graph to improve the induced k-way partition. Experimental results indicate that Cuttana produces better partitions than HeiStream for graphs where the stream order has little locality.

4 Deep Multilevel Graph Partitioning

One particularly prominent application of balanced graph partitioning is the distribution of workloads across parallel machines. In this context, the graph serves as an abstraction of the underlying computational problem: each vertex represents a unit of work, and each edge models a communication requirement between these units. The goal is to assign tasks to k processors such that each processor receives a balanced workload, and the overall communication cost – corresponding to the cut edges – is minimized.

Modern high-performance computing, however, pushes k into the millions – a domain where traditional graph partitioning methods falter. Most existing research has focused on small values of k (typically $2 \leq k \leq 256$), and when applied to large k, these algorithms often yield imbalanced partitions and large edge cuts, or suffer from excessive running times. This is due to the fact that most high-quality general-purpose graph partitioners employ the multilevel scheme and derive a k-way partition either through direct k-way partitioning or recursive bipartitioning. In direct k-way partitioning, the coarsening phase stops when there are still $\Omega(k)$ vertices – a size that becomes impractical as k increases. Moreover, since parallel partitioners often compute initial partitions sequentially, this can become a bottleneck for scalability. Recursive bipartitioning performs $\log(k)$ multilevel cycles, successively bipartitioning the block-induced subgraphs from the previous cycle. These approaches are constrained to 2-way refinement, which limits solution quality. When k is large, the bipartitioning steps must either enforce very small imbalances or risk failing to produce a balanced k-way partition.

Contributions. We introduce a novel approach to multilevel graph partitioning, which we call *deep multilevel graph partitioning* (Deep MGP). Deep MGP extends the multilevel scheme deep into initial partitioning and integrates aspects of direct k-way and recursive bipartitioning. The scheme can be instantiated with concrete (parallel) building blocks for coarsening and uncoarsening, k-way refinement (also called *local improvement*), and bipartitioning of small graphs. We also include *balancing* as an explicit component. Deep MGP performs only one multilevel cycle and always coarsens the graph down to just $2C$ vertices, where C is an input parameter. Bipartitioning is done during uncoarsening so that it is always applied to graphs with about $2C$ vertices until the desired number k of blocks is reached. This approach mitigates the aforementioned problems. To exploit all the available parallelism, the invariant is maintained that parallel tasks performed by p processing elements work on graphs with at least pC vertices. Maintaining this invariant during coarsening enables multiple diversified attempts with minimal overhead. Under certain simplifying assumptions, Deep MGP for k-partitioning an n-vertex graph with bounded degree can be done in time $\mathcal{O}\big((n/p)\max(1, \log(kC/n)) + \log^2 n\big)$, i.e., with linear work and polylogarithmic span unless k is very large. We further present a scalable shared-memory parallel implementation of Deep MGP, named KaMinPar. Developed in `C++20` and leveraging oneAPI Thread Building Blocks (oneTBB) for parallelization, KaMinPar integrates well-established techniques from prior graph partitioning systems, such as size-constrained label propagation for coarsening and refinement, along with a pool of greedy initial bipartitioning heuristics. Our careful and

efficient implementation leads to substantial performance gains and improved robustness over previous partitioners even for small values of k. In particular, we improve cache locality during label propagation and enhance clustering robustness by adapting METIS's [LaS+15] two-hop matching. We show that the resulting clustering scheme yields a geometric reduction in the number of vertices from one hierarchy level to the next, allowing coarsening to proceed until the graph becomes arbitrarily small. This stands in contrast to prior partitioners that use size-constrained label propagation for clustering during coarsening, which can stagnate early on some graphs. Additionally, we optimize coarse graph contraction by employing a buffered approach that avoids redundant edge deduplication and eliminates the need for a global hash table. Finally, we introduce a greedy balancing algorithm to ensure balanced partitions for unweighted input graphs.

References and Contributors. This chapter is mostly based on the conference publication Ref. [Got+21b] published together with Lars Gottesbüren, Tobias Heuer, Peter Sanders and Christian Schulz. The text was mainly written by the author of this dissertation with editing by Lars Gottesbüren, Tobias Heuer, Peter Sanders and Christian Schulz. The proof of Theorem 1 was written by Peter Sanders. Theorem 3 is based on the conference publication [Got+25] published together with Lars Gottesbüren, Nikolai Maas, Dominik Rosch and Peter Sanders. The theorem was originally written by Dominik Rosch as part of his bachelor's thesis [Ros24] and later refined by Nikolai Maas. The ideas described here were developed together by all authors. The algorithms were implemented by the author of this dissertation.

Structure. The remainder of this chapter is organized as follows. In Section 4.1, we review work that is closely related to our contributions. Subsequently, Section 4.2 introduces the deep multilevel graph partitioning scheme as a novel partitioning approach. This introduction is independent of any specific building blocks. Thus, we describe a concrete instantiation of Deep MGP in Section 4.3 by introducing a shared-memory parallel graph partitioner based on Deep MGP, KaMinPar. Finally, we present an experimental evaluation of KaMinPar in Section 4.4, where we compare its performance against state-of-the-art partitioners in Section 4.4.1 and further evaluate its components in Section 4.4.2.

4.1 Related Work

We introduce most of the related work in Chapter 3. In this section, we focus on issues closely related to the main contributions of this chapter.

Graph partitioners that are able to partition large real-world graphs can be categorized into three types of systems: streaming [Abb+18; FS22; Mar+17; NU13; SK12; Tso+14], single-level [Slo+17; SMR16] and parallel multilevel algorithms [ASS20; CP08; KK98c; MSS17]. Streaming graph partitioners operate in a model in which the vertices v and their neighborhood arrive one at a time and the algorithm has to assign v to a block immediately. Single-level systems such as PuLP [Slo+17; SMR14] directly partition the input graph into k blocks and use local search heuristics to further improve solution quality. Streaming and single-level systems significantly sacrifice solution quality to achieve higher speed [FS22; Mar+17; SMR16; Tso+14]. Most modern high-quality graph partitioners are therefore based on the multilevel scheme. Well-known software packages based on this approach include the sequential graph partitioners KaHIP [SS11a], Metis [KK98a] and Scotch [PR96], the shared-memory graph

partitioners Mt-KaHIP [ASS20] and Mt-Metis [LK13; LK16], and the distributed graph partitioners PT-Scotch [CP08], ParHIP [MSS17] and ParMetis [KK99].

The matching [CP08; Dev+06; KK99; LK13; WC07] and clustering algorithms [ASS20; Çat+12; Got+21a] used by different sequential partitioners in the coarsening phase are well-suited for parallelization, sometimes with only minor quality losses [Bir+13; Çat+12]. Refer to Section 3.2 for a broader overview on (parallel) coarsening techniques. Recall that coarsening stops once the graph has been reduced to a fixed number of vertices per block, typically kC. The tuning parameter C balances the trade-off between giving the initial partitioner enough flexibility to find high-quality initial partitions and limiting the cost of this phase. To compute initial partitions, parallel partitioners either call sequential multilevel algorithms with different random seeds [ASS20; CP08; Dev+06; HSS10] or use parallel recursive bipartitioning [Got+21a; KK99; LK13; LK16]. In the former case, the graph is copied to each PE and the best partition obtained from all independent runs is projected back to the coarsest graph. In the latter case, parallelism is achieved by either splitting the thread pool into two evenly-sized groups and assigning each to one of the two recursive calls [KK99; LK13], or dynamically assigning the threads to the recursive calls with a task scheduler [Got+21a]. To obtain an initial bipartition of the coarsest graph, portfolio approaches composed of different bipartitioning algorithms such as random bipartitioning, breadth-first searches from randomly selected seed nodes, label propagation and greedy graph growing [KK98a] are often used [ÇA99; Got+21a; Sch+16; SS11a]. Refer to Section 3.3 for a broader discussion of initial partitioning approaches.

Most parallel partitioners use the label propagation heuristic to improve the solution quality during the refinement phase [ASS20; Dev+06; Got+21a; LK13; MSS17; WC07]. More advanced techniques are based on parallel variants of the FM local search [FM82] that are widely used in sequential partitioners. PT-Scotch [CP08], KaPPa [HSS10] and Jostle [WC07] use sequential 2-way FM refinement on two adjacent blocks of the partition. Mt-KaHIP [ASS20] and Mt-KaHyPar [Got+21a] implement a parallel k-way FM algorithm based on the *localized multi-try* FM of the sequential graph partitioner KaHIP [SS11a]. Mt-Metis [LK16] uses greedy refinement (FM with only positive gain moves), and hill-scanning, a simplification of localized FM where small groups of vertices, whose individual gains are negative, are moved if their combined gain is positive. In contrast to multi-try FM, all vertices of a group are moved to the same target block. In the parallel setting, vertices can change their block concurrently, which requires synchronization to ensure that the balance constraint is not violated [LK13]. Existing systems either explicitly communicate their local changes and reject moves that would violate the balance constraint [Dev+06; LK13; MSS17] or use atomic *compare-and-swap* instructions to maintain block weights [ASS20; Got+21a]. Refer to Section 3.4 for a more general discussion on (parallel) refinement algorithms.

Limitations of Existing Systems for Large k. As stated above, the coarsening phase of MGP usually stops when kC vertices are left. For large k, this breaks the assumption that the coarsest graph is small, since $kC \approx n$ in the worst case. Thus, expensive initial partitioners are infeasible at this level. In particular, performing initial partitioning via a sequential k-way partitioner induces a severe sequential bottleneck in existing parallel partitioners, e.g., Mt-KaHIP [ASS20]. Many partitioners use multilevel recursive bipartitioning for the coarsest graph [Akh+17; Got+21a; KK98c; LK13; SS11a], which produces good initial partitions as long as n is *sufficiently larger* than k [KK98c]. This results in a sequential running time of $\mathcal{O}(T \log k)$, where T is the running time of the bipartitioning algorithm. When this is used within a parallel algorithm, initial partitioning can become a major bottleneck [ASS20].

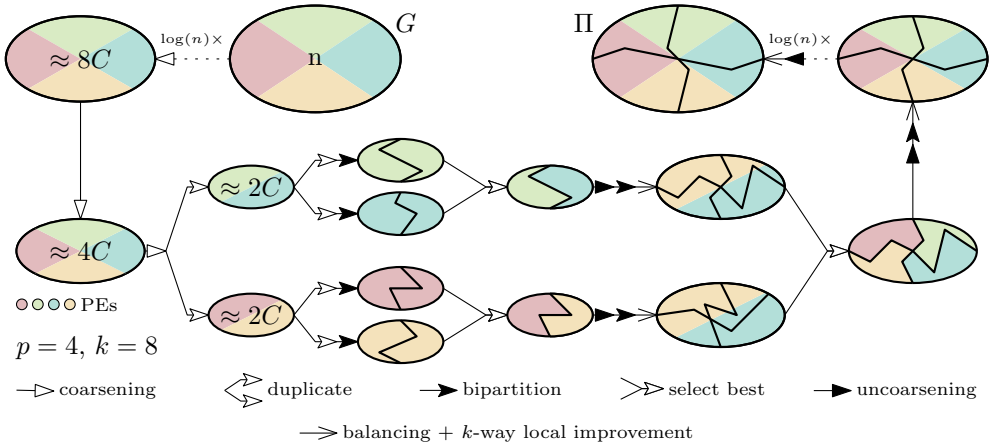

Figure 4.1 Illustrative example of the deep multilevel graph partitioning scheme. Partitioning a graph G with n vertices into $k = 8$ blocks (partition Π) using $p = 4$ PEs. During coarsening, small graphs are duplicated to maintain load $\geq pC$ on each PE. Blocks are recursively bipartitioned during uncoarsening. Colors indicate work performed by each PE, and arrowheads indicate the type of the operations.

Furthermore, a feasible solution for the k-way graph partitioning problem must satisfy the balance constraint that usually depends on the average block weight $\frac{c(V)}{k}$. Thus, increasing the number of blocks leads to a tighter balance constraint and drastically reduces the space of feasible solutions. Therefore, a partitioner handling larger values of k has to employ techniques to ensure that the balance constraint is not violated.

4.2 Deep Multilevel Graph Partitioning

In this section, we introduce our first major contribution: a new partitioning scheme called *Deep Multilevel Graph Partitioning* (Deep MGP). Deep MGP extends the coarsening phase deep into the initial partitioning phase. Roughly speaking, the process starts by coarsening the input graph until only $2C$ vertices remain, where C is a constant tuning parameter called *contraction limit*. The coarsest graph is then bipartitioned into two blocks. Then, uncoarsening starts, where Deep MGP maintains the key invariant that a coarse graph with n' vertices has $k' = \min\left\{k, \text{ceil}_2\left(\frac{n'}{C}\right)\right\}$ blocks[1]. This ensures that each instance of flat bipartitioning operates on a graph with approximately $2C$ vertices, assuming that the number of vertices in each block roughly doubles when unrolling a single coarsening level. As the hierarchy levels are unrolled, the graph is ultimately divided into $\min\left\{k, \text{ceil}_2\left(\frac{n}{C}\right)\right\}$ blocks. In the case where $k > \text{ceil}_2\left(\frac{n}{C}\right)$, further bipartitioning occurs on the input graph until k blocks are obtained. If k is not a power of two, necessarily inhomogeneous block weights are maintained during steps where $k' < k$ to produce k balanced blocks after the last bipartitioning step.

Essentially, this approach combines the merits of direct k-way partitioning and recursive bipartitioning. Like direct k-way partitioning, Deep MGP coarsens and uncoarsens the graph only once and allows the use of k-way refinement algorithms throughout the coarsening hierarchy. This yields similar performance to direct k-way partitioning for the (traditional)

[1] We use $\text{ceil}_2(x)$ to denote the smallest power of two that is $\geq x$

Algorithm 2 `partition`

Input : $G = (V, E)$, k, $\varepsilon > 0$, const. C
Output : k'-way partition Π of G

1 **if** $|V(G)| > 2C$ *and coarsening has not converged* **then** `// Recursion`
2 $\quad G_c \coloneqq \texttt{coarsen}(G)$
3 $\quad \Pi_c \coloneqq \texttt{partition}(G_c, k, \varepsilon, C)$
4 $\quad \Pi \coloneqq \texttt{project}(G, \Pi_c)$
5 **else** `// Base case`
6 $\quad \Pi \coloneqq \{V\}$
7 $k' \coloneqq \begin{cases} k & \text{if } |V| = n \\ \text{ceil}_2(|V|/C) & \text{otherwise} \end{cases}$
8 $k' \coloneqq \max\{\min\{k, k'\}, 2\}$
9 **while** $|\Pi| < k'$ **do** `// Obtain` k' `blocks`
10 $\quad \Pi \coloneqq \texttt{bipartitionBlocks}(G, \Pi, k)$
11 $\Pi \coloneqq \texttt{refine}(G, \texttt{balance}(G, \Pi))$
12 **return** Π

small values of k commonly found in the literature (roughly $2 \leq k \leq 256$), and avoids the $\log(k)$ factor of recursive bipartitioning. On the other hand, Deep MGP forces that possibly expensive initial (bi)partitioning algorithms are only applied to small graphs, eliminating the potential bottleneck for large values of k or parallel graph partitioners.

The remainder of this section provides a detailed description of Deep MGP. We initially simplify the explanation by assuming that k is a power of two but later generalize it. Finally, we discuss parallelization and analyze the algorithm's running time.

4.2.1 Algorithmic Overview

Deep MGP starts by progressively coarsening the input graph $G_1 = (V_1, E_1)$, building a hierarchy of increasingly coarse representations $G_2, \ldots, G_\ell$ of G_1. Each level in this hierarchy is generated by clustering vertices in G_i and contracting these clusters to form the next coarser graph G_{i+1}. This process continues until one of two conditions is met:

1. The coarsest graph G_ℓ has at most $2C$ vertices, where C is the predefined contraction limit, or
2. the coarsening procedure converges, meaning further contraction does not significantly reduce the graph's order.

Crucially, the order of G_ℓ is independent of k. In Algorithm 2, this process is implemented through the recursion in lines 2–3. From this point, the process continues through a sequence of the following operations. The current graph is first partitioned into smaller blocks through recursive initial bipartitioning. If necessary, the partition is then rebalanced to ensure that all blocks adhere to the balance constraint. A k-way refinement algorithm is subsequently used to improve partition quality before the partition is projected onto the previous graph $G_{\ell-1}$, where the process is repeated. Throughout the uncoarsening process, two key invariants are maintained:

Algorithm 3 `bipartitionBlocks`

Input : G, k'-way partition Π, k, const. R
Output : $2k'$-way partition Π'

```
1 Π' := ∅
2 foreach V_i ∈ Π do
3     ε' := ((1 + ε) c(V) / (|Π| c(V_i)))^(1/log2(k/|Π|)) − 1
4     G_i := G[V_i]
5     // Compute R bipartitions of V_i
6     Π_1, ..., Π_R := bipartition(G_i, ε', R)
7     // Select lowest (balanced) edge cut
8     Π' := Π' ∪ lowest(G_i, Π_1, ..., Π_R)
```

(P) A coarse graph $G_i = (V_i, E_i)$ is partitioned into $k_i := \text{ceil}_2\left(\frac{|V_i|}{C}\right)$ blocks (bounded by 2 and k), and
(B) a k_i-way partition of G_i fulfills the balance constraint (before being projected to G_{i-1}).

An idealized coarsening algorithm produces a hierarchy in which the number of vertices is halved at each successive level, and the coarsest graph G_ℓ has $2C$ vertices. In this scenario, a single bipartitioning of G_ℓ is sufficient to initially satisfy invariant (P). As uncoarsening progresses and the number of vertices doubles at each step, preserving the invariant requires that each block undergoes exactly one additional bipartitioning. In the more realistic case, however, coarsening may reduce the number of vertices by a factor significantly larger than 2. To ensure that invariant (P) holds in such cases, we employ recursive initial bipartitioning. Specifically, whenever the partition Π_i of G_i does not satisfy invariant (P), each block in Π_i is recursively bipartitioned until the required k_i blocks are obtained. This process is implemented in lines 7–10 of Algorithm 2, while a single initial bipartitioning step is implemented in Algorithm 3. Note that we use the adapted ε' of Ref. [Sch+16] here, since using the non-adapted ε for each bipartition could overload blocks [Sch+16].

Uncoarsening and initial bipartitioning can cause violations of the balance invariant (B), as will be discussed in Section 4.2.4. If this is the case, we run a balancing algorithm afterwards. The resulting partition – which then satisfies both invariants (P) and (B) – is then improved using a k-way local improvement algorithm (Algorithm 2, line 11).

If $k > \text{ceil}_2\left(\frac{|V_1|}{C}\right)$, the partition computed by the process described above has less than k blocks. In this case, we perform an additional round of recursive initial bipartitioning to obtain k blocks, and run balancing and k-way local improvement once more on the final partition.

4.2.2 Parallelization

We parallelize the partitioning method described above using parallel coarsening, local improvement and balancing algorithms. On very coarse levels, we maintain the invariant that parallel tasks performed by p PEs work on graphs with at least pC vertices. This is achieved by running initial partitioning on more and more copies of the coarsened graph, as illustrated by Figure 4.1. We diversify this search by using randomized coarsening, initial bipartitioning and local improvement algorithms. More precisely, we follow the algorithm

described above until the coarsest graph G_C has pC vertices left. To uphold the invariant that tasks performed by p PEs work on graphs with at least pC vertices, we obtain two copies G_C^r and G_C^ℓ of G_C, and split PEs into two groups with $p' = \frac{p}{2}$ PEs each. Note that grouping the PEs is only conceptual and the partitioning scheme does not require any static PE-to-graph assignment. If $p' > 1$, we continue by coarsening G_C^r with PEs of the first group and G_C^ℓ with PEs of the second group, until each graph has $p'C$ vertices left. We proceed in this fashion recursively, until we have obtained p graphs with $2C$ vertices each after $\log_2(p)$ recursion levels.

Each of these graphs is then bipartitioned using a single PE. Let G_C^r and G_C^ℓ with respective bipartitions Π_C^r and Π_C^ℓ be two such graphs that are copies of G_C on the previous recursion level. We use the better bipartition of Π_C^r and Π_C^ℓ (i.e., if only one of these partitions is feasible, we use that one, otherwise the one with the lower edge cut) as partition Π_C of G_C. We proceed on G_C as before, i.e., we bipartition each block of Π_C if applicable, and apply the balancing and local improvement algorithm. This process is repeated for all $\log_2(p)$ recursion levels.

4.2.3 Handling General k

The simplified description of Deep MGP given above only considers the case where the number of blocks k is a power of two. In this case, recursive bipartitioning results in a perfect binary bipartitioning tree, i.e., all blocks participate in all bipartitioning steps. Moreover, on each level of the bipartitioning tree, all blocks must adhere to the same maximum block weight. In the general case where k can be an arbitrary integer, the bipartitioning tree is no longer perfect and blocks are subject to different maximum block weights. Thus, we associate each of the $i = 0, \ldots, 2^\ell - 1$ blocks after the ℓ-th bipartitioning step with its *final block count* $f_{V_i^\ell}$, that is, the number of blocks that V_i^ℓ will be subdivided into in the final partition. Given these counts, we compute the maximum block weight L_i^ℓ of block V_i^ℓ as

$$L_i^\ell := \max\left\{ f_{V_i^\ell} \cdot (1+\varepsilon)\frac{c(V)}{k}, f_{V_i^\ell} \cdot \frac{c(V)}{k} + \max_v c(v) \right\}.$$

Initially, there is only one block $V_0^0 := V$ and we set $f_{V_0^0} := k$. Bipartitioning the i-th block on level ℓ produces two sub-blocks $V_{2i}^{\ell+1}$ and $V_{2i+1}^{\ell+1}$ on level $\ell+1$, for which we define their final block counts recursively as $f_{V_{2i}^{\ell+1}} := \lceil \frac{f_{V_i^\ell}}{2} \rceil$ and $f_{V_{2i+1}^{\ell+1}} := \lfloor \frac{f_{V_i^\ell}}{2} \rfloor$. Thus, the block counts on some level ℓ are either $\lfloor k/2^\ell \rfloor$ or $\lceil k/2^\ell \rceil$, see Figure 4.2 for an example. More precisely, we observe that

$$f_{V_i^\ell} = \begin{cases} \lceil k/2^\ell \rceil & \text{if } \overleftarrow{\sigma}(i) \leq (k \bmod 2^\ell), \\ \lfloor k/2^\ell \rfloor & \text{otherwise,} \end{cases}$$

where $\overleftarrow{\sigma}(i)$ denotes the permutation obtained by reversing the binary ℓ-digit representation of i. Thus, we can compute the final block counts efficiently through constant time bit operations and do not have to store the tree explicitly. Once we have computed a $k' := \text{floor}_2(k)$-way partition, there are $k - k'$ blocks B with $f_B = 2$ and $2k' - k$ blocks B with $f_B = 1$. During the next and final bipartitioning step, we obtain a k-way partition by only bipartitioning those blocks with $f_B = 2$.

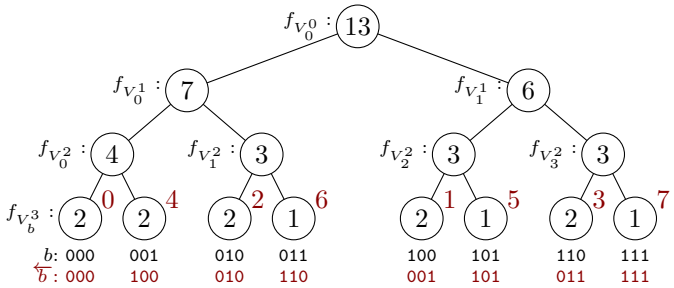

Figure 4.2 Bipartitioning tree for a $k = 13$-way partition. Vertices are labeled with their *final block count* $f_{V_i^\ell}$. When working on a $k' = 2^\ell$-way partition, we set the maximum weight L_i^ℓ of block i to $\max\{f_{V_i^\ell} \cdot (1+\varepsilon)\frac{c(V)}{k}, f_{V_i^\ell} \cdot \frac{c(V)}{k} + \max_v c(v)\}$.

4.2.4 Maintaining the Balance Constraint

Since MGP implementations usually employ coarsening algorithms that do not guarantee strictly uniform vertex weights, maintaining the balance constraint used in other partitioning systems, $L := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$, becomes an NP-complete problem [GJ79] on coarse levels. To mitigate this problem, we use $L_{\max} := \max\{(1+\varepsilon)\frac{c(V)}{k}, \frac{c(V)}{k} + \max_v c(v)\}$ as balance constraint instead. This ensures that a feasible partition always exists, and that it can be found with simple greedy algorithms. Both claims are based on the fact that the average block weight of a partition is $\frac{c(V)}{k}$ and thus, there is always at least one block V_i with $c(V_i) \leq \frac{c(V)}{k}$. In the multilevel setting, projecting a partition to a finer graph can violate the balance constraint due to the change in $\max_v c(v)$. However, the overload per block is bounded by $\max_v c(v)$, which implies that a balancing algorithm only needs to move a small number of vertices out of a block to restore the balance constraint.

4.2.5 Running Time

Next, we analyze the running time of parallel (Deep) MGP using highly idealized assumptions. We do not claim that the results hold for our implementation but use this simplified analysis to give a quantitative expression to the qualitative reasoning that Deep MGP is scalable if its components are scalable. The analysis also allows us to compare the asymptotic performance of different approaches to parallel MGP without having to discuss which particular implementations of the basic operations can or cannot avoid certain difficult cases. We assume:

(1) k is a power of two and we have unit vertex and edge weights,
(2) $n > Cp \log p$,
(3) coarsening a graph halves the number of vertices,
(4) coarsening, uncoarsening and balancing a graph with n vertices takes time $\mathcal{O}(n/p + \log n)$,
(5) sequential bipartitioning takes linear time.

We effectively ignore edges here. This implies the assumption that vertices have bounded degree and that the degrees remain bounded when the graph is shrunk.

▶ **Theorem 1.** *Under the assumptions made above, Deep MGP requires time*

$$\mathcal{O}\left(\frac{n}{p} \max\left(1, \log \frac{kC}{n}\right) + \log^2 n\right).$$

Proof. By (3), MGP goes through $\log(n/C)$ levels so that these overhead terms "$\log n$" in (4) sum to $\mathcal{O}\big(\log^2 n\big)$ – we ignore these overhead from now on. While more than pC vertices are left, the graph shrinks geometrically with the levels so that the total remaining cost for (un)coarsening and balancing from (4) is linear – $\mathcal{O}(n/p)$.

When at most pC vertices are left, replication and selection of the best partition keeps the number of vertices at each level at $\Theta(pC)$. There are $\log p$ such levels incurring total cost $\mathcal{O}(C \log p)$ for (un)coarsening and balancing. By (2) this cost is bounded by $\mathcal{O}(n/p)$.

For the cost of bipartitioning we consider three cases:

Case (a) $k \leq p$**:** Each PE performs $\log k$ bipartitions with total cost $C \log k$. By (2) this is bounded by $\mathcal{O}(n/p)$.

Case (b) $p < k \leq n/C$**:** Each PE performs $\log p + k/p$ bipartitions with total cost $C(k/p + \log p)$. Once more, by (2) this is bounded by $\mathcal{O}(n/p)$.

Case (c) $k > n/C$**:** In this case, Deep MGP first performs an n/C-way partitioning into blocks of size about C. By the above analysis, this takes time $\mathcal{O}\big(n/p + \log^2 n\big)$. Then the remaining blocks are partitioned into $k/(n/C) = kC/n$ blocks using recursive bipartitioning in time $\mathcal{O}(C \log(kC/n))$. Summing over all blocks assigned to a PE we get additional cost $n \log(kC/n)/p$. ◀

▶ **Corollary 2.** *Under the assumptions made above, parallel multilevel graph partitioning requires time*

- $\mathcal{O}\Big(\frac{n}{p} + kC + \log^2 n\Big)$ *if direct k-way partitioning is used,*
- $\mathcal{O}\Big(\frac{n}{p} \log k + \log^2 n\Big)$ *if recursive bipartitioning is used, and*
- $\mathcal{O}\Big(\frac{n}{p} \max\big(1, \log \frac{kC}{n}\big) + \log^2 n\Big)$ *if deep multilevel is used.*

In the following discussion, we ignore the overhead terms $\mathcal{O}\big(\log^2 n\big)$. In the "traditional" case where k is *small* (more precisely, $k \leq \frac{n}{C}$), the overhead factor $\mathcal{O}\big(\max\big(1, \log \frac{kC}{n}\big)\big)$ of Deep MGP vanishes. Thus, its asymptotic running time becomes $\mathcal{O}\Big(\frac{n}{p}\Big)$, which is the same running time as direct k-way partitioning and asymptotically faster than recursive bipartitioning by a factor of $\mathcal{O}(\log k)$. On the other hand, Deep MGP does not suffer from the sequential $\mathcal{O}(kC)$ bottleneck of direct k-way partitioning for *large* k. For $k \geq \frac{n}{C}$, its asymptotic running time becomes $\mathcal{O}\Big(\frac{n}{p} \log \frac{kC}{n}\Big)$, thus reducing the $\mathcal{O}(\log k)$ factor of recursive bipartitioning to $\mathcal{O}\big(\log \frac{kC}{n}\big)$.

4.3 Implementation: KaMinPar

In this section, we present the key components of KaMinPar, our shared-memory parallel implementation of deep multilevel graph partitioning. Recall that Deep MGP relies on three main components:

1. **Parallel Coarsening:** Coarsening is performed using a parallel implementation of size-constrained label propagation, enhanced with two-hop clustering to achieve provable geometric graph shrinkage down to $\leq 2C$ vertices.
2. **Sequential Bipartitioning:** Small graphs are bipartitioned through a nested, sequential multilevel bipartitioning cycle. This involves further coarsening via sequential size-constrained label propagation, followed by bipartitioning with a pool of greedy heuristics. During the sequential uncoarsening phase, bipartitions are refined through 2-way FM.

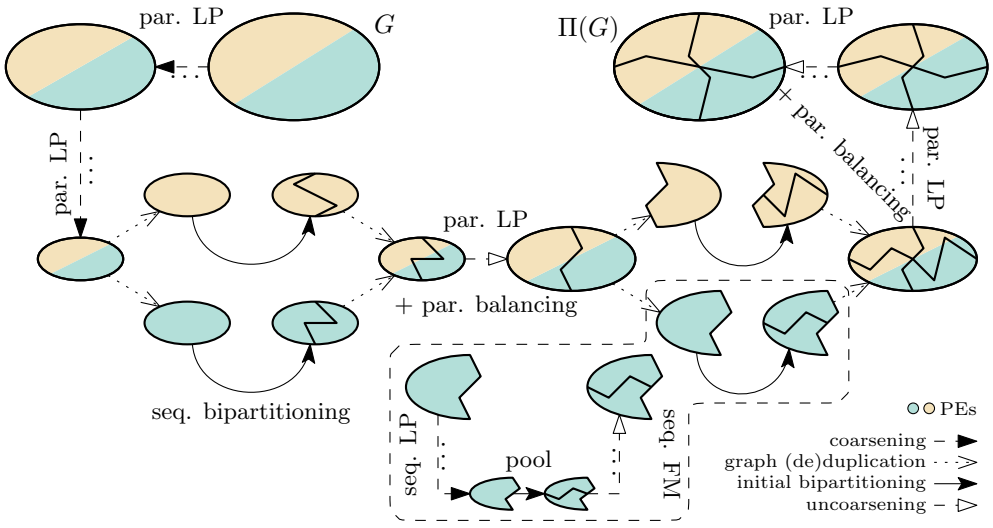

Figure 4.3 Overview of the components implemented in KaMinPar. Coarsening and refinement are achieved using parallel label propagation, complemented by a parallel balancing algorithm during uncoarsening. Bipartitions are generated through a nested, fully sequential multilevel approach: the already small graphs undergo further shrinking through coarsening based on sequential label propagation, followed by a pool of greedy sequential bipartitioning heuristics. During uncoarsening of the bipartitioned graph, two-way FM further refines it. The colors represent the workload distributed across PEs.

3. **Parallel Uncoarsening:** The k-way partition is further refined using parallel label propagation during uncoarsening. Balanced partitions are ensured by employing a greedy balancing algorithm, which leverages parallelism across partition blocks.

Figure 4.3 provides an overview of how these components are integrated into the deep multilevel scheme. We now turn towards the specifics of each component as implemented in KaMinPar, describing their parallelization and analyzing their running times.

4.3.1 Coarsening

We use the size-constrained label propagation algorithm [MSS14] to compute a clustering for contraction while ensuring that no cluster exceeds the upper weight bound U. In the literature [SS11a], a common choice for U is given by

$$U = f \cdot \frac{c(V)}{k},$$

where $f \leq 1$ is a tuning constant and k is the number of blocks in the final partition. However, this definition implies that the coarsest graph contains at least $c(V)/U = k/f$ vertices. This is not suitable for deep multilevel graph partitioning since our goal is to reduce the graph to $2C$ vertices regardless of k. On the other hand, when k is large, it is beneficial to keep U small on finer levels since excessively heavy clusters could hinder initial partitioning from finding good, balanced bipartitions. To address these constraints, we aim to define U such that it starts small on finer levels and gradually increases as the graph shrinks, i.e., it increases inversely to the number of blocks that will be produced during uncoarsening. As an additional constraint, recall from our discussion in Section 2.2 that, even without

considering edge-cut minimization, finding a balanced partition of a weighted graph with a small imbalance ε is NP-complete in general. Consequently, we propose a relaxation of the maximum block weight to

$$L_{\max} \coloneqq \max\left\{(1+\varepsilon)\left\lceil\frac{c(V)}{k}\right\rceil, \left\lceil\frac{c(V)}{k}\right\rceil + \max_{v\in V} c(v)\right\}.$$

We can further incorporate this observation into our choice of U. Let $G_i = (V_i, E_i, c_i, \omega_i)$ for $i \geq 1$ denote the current (coarse or input) graph that should be contracted to build the next coarse graph G_{i+1}. In KaMinPar, we define

$$U_i \coloneqq \varepsilon\left\lceil\frac{c(V)}{k_i}\right\rceil,$$

where

$$k_i = \min\{k, |V_i|/C\}$$

denotes the number of blocks that Deep MGP will produce for G_i during uncoarsening, k is the number of blocks in the final partition, and ε is the imbalance parameter from the problem formulation. This choice guarantees that for G_{i+1},

$$\begin{aligned}\left\lceil\frac{c(V)}{k_{i+1}}\right\rceil + \max_{v\in V_{i+1}} c_{i+1}(v) &\leq \left\lceil\frac{c(V)}{k_{i+1}}\right\rceil + \max_{v\in V_i} c_i(v) \leq \left\lceil\frac{c(V)}{k_{i+1}}\right\rceil + U_i = \left\lceil\frac{c(V)}{k_{i+1}}\right\rceil + \varepsilon\left\lceil\frac{c(V)}{k_i}\right\rceil \\ &\leq \left\lceil\frac{c(V)}{k_{i+1}}\right\rceil + \varepsilon\left\lceil\frac{c(V)}{k_{i+1}}\right\rceil = (1+\varepsilon)\left\lceil\frac{c(V)}{k_{i+1}}\right\rceil,\end{aligned}$$

thereby providing sufficient leeway to achieve balanced partitions on weighted coarse graphs using simple greedy balancing algorithms (more precisely, any vertex from an overloaded block can always be moved to some other block without overloading it). Note that the same statement would hold if U_i were enforced already when clustering G_{i-1}. However, U_i depends on $|V_i|$, which is not known until G_i has been constructed.

Since the choice of U_i depends on the parameters ε and C, it is natural to ask whether U_i might become so restrictive that it prevents further contractions. For example, consider an unweighted input graph $G_1 \coloneqq G = (V, E)$. We must have $U_1 \geq 2$ to start coarsening, since otherwise, no contraction is possible. This condition holds when $C \geq 2\varepsilon^{-1}$, because then

$$U_1 \geq \varepsilon\frac{|V|}{\min\{k, |V|/C\}} \geq \varepsilon C \geq 2.$$

As we will later show in Theorem 3 and Theorem 4, choosing $C > 2\varepsilon^{-1}$ is indeed sufficient to guarantee geometric shrinkage of the input graph down to arbitrarily small coarse graphs.

In our implementation, we set $C = 2\,000$, which consequently limits us to $\varepsilon > \frac{1}{1\,000}$. Note that this bound is significantly smaller than typical values of ε found in literature. Moreover, this does not fundamentally constrain our partitioning scheme, as even smaller imbalances could be accommodated by increasing C. Note that for very small imbalances that enforce perfect balance, alternative techniques such as those presented in Ref. [SS13] are often preferable.

Parallel Label Propagation. We perform 5 rounds of label propagation, terminating early if no vertices were moved during the previous round. However, on typical graph instances, early termination does not occur, and all 5 rounds are completed. To further improve the

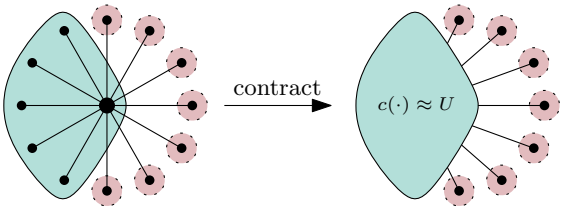

Figure 4.4 Star-graphs serve as an (extreme) example of graphs requiring two-hop clusters for effective coarsening. During the first coarsening phase, some leaves form a cluster with the center of the star until the maximum cluster weight U is reached. Afterwards, no further clusters can be formed, causing the coarsening process to terminate while the graph remains arbitrarily large.

running time, we only visit a vertex if one of its neighbors changed its cluster in the previous round [SM16]. We parallelize the algorithm by iterating over all vertices in parallel. When a vertex is moved to a new cluster, we update the respective cluster weights using atomic fetch-and-add operations. Note that this approach does not strictly enforce the weight limit, which could be violated if multiple threads move a vertex to the same cluster at the same time. However, this is unproblematic in practice, since the weight limit violations are usually small. Additionally, we introduce a balancing algorithm in Section 4.3.4.1 to fix (small) balance violations during the uncoarsening phase.

Cluster Rating Aggregation. To determine the best cluster for a vertex, we iterate over its neighbors to compute the aggregated edge weights, referred to as the *rating*, for each neighboring cluster. To compute these aggregates, we follow the approach used in Mt-Metis [LaS+15]: for vertices with low degree, we accumulate ratings in compact, thread-local hash tables with a fixed capacity. For higher-degree vertices that exceed this capacity, we use a sparse thread-local array of size $\mathcal{O}(n)$ instead. We refer to this hybrid data structure as *rating map*. In the worst case, where each thread allocates its own $\mathcal{O}(n)$ array, this implies a total memory footprint of $\mathcal{O}(np)$ for p threads.

Iteration Order and Cache Locality. The performance of label propagation is improved when vertices are processed in increasing degree order [ASS20; MSS14]. Consequently, prior work follows this approach. However, this ordering is not cache-efficient and can significantly impact running time. Additionally, it lacks diversification through randomization. To address these issues, we propose a degree bucket ordering strategy. Instead of strictly sorting vertices by degree, we group them into exponentially spaced degree buckets: bucket i contains all vertices with degrees in the range $2^i \leq d < 2^{i+1}$. Once the buckets are formed, we reorder the graph in memory by arranging vertices according to their bucket number. This organization helps preserve the natural locality present in the graph structure. Note that we only apply this reordering step to the input graph, while keeping the natural order of coarse vertices on coarse graphs. For vertex traversal, we split degree buckets into much smaller chunks and randomize traversal on an inter-chunk and intra-chunk level. This part is analogous to the randomization in Metis' matching algorithm [KK98a].

Two-hop Clustering. We observed that size-constrained label propagation alone is unable to shrink some irregular graph instances sufficiently. These graphs often feature a dense core comprising relatively few vertices that are connected to most other vertices in the

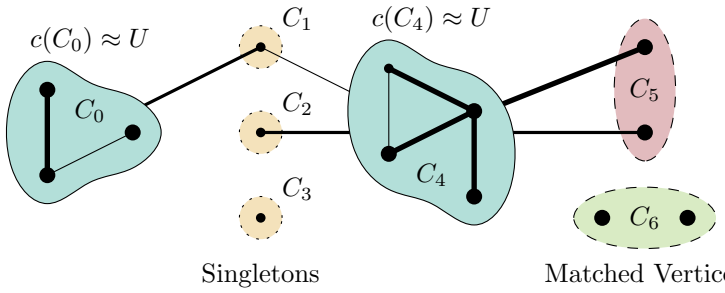

Figure 4.5 Label propagation based clustering scheme of KaMinPar. The thickness of edges and the size of vertices indicate their relative weight. Vertices are moved to the clusters to which they have the strongest connection and that do not exceed the maximum cluster weight U (C_0 and C_4). If a round of label propagation terminates with more than $n/2$ non-empty clusters remaining, additional clusters are formed through two-hop clustering: vertices which have not been moved to another cluster are matched with other such vertices that have their strongest connection to the same cluster (C_5). Consequently, note that C_1 and C_2 are not matched, since C_1 has its strongest connection to C_0 while C_2 is only connected to C_4. Isolated vertices are matched together (C_6), while any other vertices remain unmatched (C_3).

graph, while the majority of vertices are loosely connected to each other, apart from their links to the core. Star graphs, as illustrated in Figure 4.4, represent an extreme example of this structure. To address this issue, we implement a technique inspired by the two-hop matching algorithm of Mt-Metis [LaS+15]. During label propagation, if a vertex u cannot be moved to any neighboring cluster due to the size constraint, we store the highest-rated neighboring cluster as u's *favored cluster*. If, after termination, the graph has shrunk by less than 50%, we merge singleton clusters that share the same favored cluster until the graph is reduced by 50%. Note that we do not follow any particular order when merging the singleton clusters. Isolated vertices are handled similarly by conceptualizing a virtual vertex (and thus favored cluster) connected to all isolated vertices. We experimented with two approaches for creating two-hop clusters. In the first approach, vertices are added to two-hop clusters until the clusters reach the maximum weight U. In the second approach, two-hop clusters are limited to a maximum of two vertices. Preliminary experiments showed that limiting two-hop clusters to two vertices resulted in slightly better partition quality. We illustrate the resulting clustering scheme in Figure 4.5.

Coarsening Guarantees. In Theorem 1, we analyzed the running time of Deep MGP under the assumption that the number of vertices decreases geometrically during coarsening. With the introduction of two-hop clustering, we can now prove that the coarsening scheme implemented in KaMinPar inherently satisfies this assumption. However, we note that the overall worst-case running time of KaMinPar remains superlinear, since our coarsening scheme does not bound the density of coarse graphs. We will address this issue in Chapter 6.

▶ **Theorem 3.** *Let $G = (V, E, c)$ be a vertex-weighted graph without isolated vertices, and let $\mathcal{C}$ be a clustering of G computed by size-constrained label propagation. Let U be the size constraint. Then, the number of clusters is bounded by*

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

Proof. We divide the set of clusters $\mathcal{C}$ into multiple subsets. 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 vertices 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 vertices 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 vertex 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, two-hop clustering would have joined them). 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 $c(V) \geq \sum_{K\in C_h} c(K) + \sum_{K\in C_1} c(K) = \sum_{K\in C_1}(c(K) + c(K_S)) + \sum_{K\in C_h\setminus\{K'_S | K'\in C_1\}} c(K) \geq U|C_1| + \frac{1}{2}U(|C_h| - |C_1|)$. Rearranged, this is $|C_h| + |C_1| \leq 2\frac{c(V)}{U}$.

Combining all inequalities, we get

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

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 vertices (i.e., vertices without a neighbor) are a special case as standard clustering algorithms do not handle them. Therefore, they are omitted from Theorem 3. However, it is trivial to either remove isolated vertices and reinsert them in the uncoarsening, or alternatively cluster them with each other (we do the latter, as described above).

▶ **Corollary 4.** *Let C be the contraction limit and ε be the imbalance parameter from the problem formulation. If $C > 2\varepsilon^{-1}$, the coarsening scheme of KaMinPar shrinks the number of vertices in coarse graphs geometrically.*

Proof. Let $G = (V, E)$ denote the current (coarse) graph and assume (without loss of generality) that there are no isolated vertices. Recall that we define the maximum cluster weight for G as $U = \varepsilon\left\lceil\frac{c(V)}{\min\{k, |V|/C\}}\right\rceil$. Let $\gamma \coloneqq \frac{1}{2} + \frac{1}{\varepsilon C}$. Since $C > 2\varepsilon^{-1}$, $\gamma < 1$. By plugging U into Theorem 3, we obtain

$$|\mathcal{C}| \leq \frac{1}{2}|V| + \frac{\min\{k, |V|/C\}}{\varepsilon} \leq \frac{1}{2}|V| + \frac{1}{\varepsilon C}|V| = \left(\frac{1}{2} + \frac{1}{\varepsilon C}\right)|V| = \gamma|V|.$$

Since the clusters of G become the coarse vertices of the next coarse graph, the number of vertices is thus reduced by a factor of $1/\gamma > 1$ at each level. ◀

Running Time and Memory Requirement. The proposed label propagation algorithm has a parallel runtime complexity of $\mathcal{O}\left(\frac{m}{p} + \Delta\right)$ with span $\mathcal{O}(\Delta)$, since it does not employ parallelism over large neighborhoods. The total computational work is linear, $\mathcal{O}(n + m)$. In the worst case, each thread allocates the sparse $\mathcal{O}(n)$ table for gain aggregation, leading to an overall memory footprint of $\mathcal{O}(np)$. While this is feasible for many real-world graphs,

scalability can be limited for graphs with high maximum degree (e.g., irregular social networks) due to the $\mathcal{O}(\Delta)$ span. Furthermore, consuming $\mathcal{O}(np)$ memory can be problematic with the growing core counts in modern CPUs. In Section 5.4.1, we will therefore present an alternative parallelization scheme that is better suited for clustering graphs featuring vertices with large degrees on a large number of threads p.

4.3.2 Contraction

After computing a clustering $\mathcal{C}$ of $G = (V, E)$, the next step is to contract $\mathcal{C}$ and construct the coarse graph $G_{\mathcal{C}} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ in which each vertex represents an entire cluster of G. This contraction process is detailed in Algorithm 4 and follows four primary steps:

1. The first step (lines 2 to 9) constructs a mapping $\mathcal{M}$ from the vertices in G to the coarse vertices in $G_{\mathcal{C}}$. We initialize $\mathcal{M}$ with 0 and mark non-empty clusters by setting $\mathcal{M}[\mathcal{C}[v]] = 1$ for all $v \in V$. A prefix sum over $\mathcal{M}$ (line 5) then maps cluster IDs to coarse vertex IDs. Finally, we obtain the fine-to-coarse vertex mapping by concatenating $\mathcal{M}$ and $\mathcal{C}$ in line 8.
2. The next step is to sort the vertices of G by their corresponding coarse vertex, which we do in lines 11 to 17 using standard parallel integer sorting. Afterwards, $\mathcal{B}$ stores the sorted vertices, while $\mathcal{I}$ marks the starting position of each coarse vertex in $\mathcal{B}$. Thus, the vertices corresponding to a cluster C are $\{\mathcal{B}[\mathcal{I}[C]], \mathcal{B}[\mathcal{I}[C]+1], \ldots, \mathcal{B}[\mathcal{I}[C+1]-1]\}$.
3. We then obtain the coarse edges by traversing all outgoing edges of a cluster, using thread-local rating maps $\mathcal{R}_t$ for edge weight aggregation and coarse edge deduplication (lines 19 to 28). This step is parallelized over coarse vertices using $\mathcal{I}$ and $\mathcal{B}$ from the previous step. Since the position of edges in $E_{\mathcal{C}}$ depends on the vertex degrees of preceding coarse vertices – information we do not yet have – we cannot directly store the result in $E_{\mathcal{C}}$. One approach to solving this problem would be to perform this step twice, obtaining vertex degrees during the first pass and writing the edges during the second pass. However, this incurs a significant running time overhead. Instead, we store the coarse edges in auxiliary thread-local edge buffers $\mathcal{L}_t$, tracking their position in the shared array $\mathcal{D}$ (line 26).
4. Finally, we compute the coarse vertex degrees using the information recorded in $\mathcal{D}$ and copy the edges from $\mathcal{L}_t$ to $E_{\mathcal{C}}$ (lines 30 to 35).

4.3.3 Initial Bipartitioning

Following the coarsening phase, the next building block of Deep MGP is initial bipartitioning. Ideally, the graphs subject to bipartitioning should be relatively small, containing approximately $2C$ vertices. However, we note that their actual size depends on the rate of coarsening. The label propagation scheme described above can rapidly shrink graphs with well-defined cluster structures, which may force bipartitioning to be performed on considerably larger graphs. The algorithms in this section are implemented sequentially and we only employ parallelism over the blocks subject to bipartitioning. Recall that on the coarsest graphs, Deep MGP provides sufficient parallel workloads by replicating the graphs.

Let $G = (V, E)$ be the input graph to be partitioned into k blocks, and let $B = (V_B, E_B)$ be a block-induced subgraph of a k'-way partitioned coarse representation G' of G. We extend G' to more blocks by recursively bipartitioning B. Each bipartition is computed following a nested multilevel cycle, as illustrated in Figure 4.3. Coarsening is performed using size-constrained label propagation, but we only perform a single round of label propagation on each level to reduce running time. Moreover, this enables a small optimization where

Algorithm 4 `ContractClustering`$(G, \mathcal{C})$

Input : Graph $G = (V, E)$, $n = |V|$, clustering $\mathcal{C}[\cdot]$.
Output : Coarse graph $G_{\mathcal{C}} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ obtained by contracting $\mathcal{C}$.

```
1  // Compute fine to coarse mapping M
2  M ← [0, ..., 0]                                          // Array of size n + 1
3  for u ∈ V do parallel
4      M[C[u]] ← 1
5  M ← ExclusivePrefixSum(M)
6  n_C ← M[n]                                              // Number of coarse vertices
7  for u ∈ V do parallel
8      C[u] ← M[C[u]]
9  M ← C
10 // Integer sort V by M
11 I ← [0, ..., 0]                                         // Array of size n_C + 1
12 for u ∈ V do parallel
13     I[M[u]] atomic+= 1
14 I ← InclusivePrefixSum(I)
15 B ← [0, ..., 0]                                         // Array of size n
16 for u ∈ V do parallel
17     B[atomic-- I[M[u]]] ← u
18 // Compute coarse edges
19 L_t ← []                                  // Thread-local buffer for coarse edges
20 D ← []                                    // Concurrent buffer to index L_t
21 R_t ← []                                  // Thread-local rating map
22 for b := 0 to n_C do parallel                                   // Thread t
23     for u ∈ B[I[b]..I[b + 1] − 1] do
24         for e = {u, v} ∈ E with M[u] ≠ M[v] do
25             R_t[M[v]] += ω(e)
26         D.concurrentPushBack(|R_t|, |L_t|, t)
27         L_t.append(R_t)                          // Append copy of R_t to L_t
28         R_t.clear()
29 // Construct coarse graph
30 V_C ← ExclusivePrefixSum(D_1)                  // Prefix sum over coarse degrees
31 m_C ← V_C[n_C]                                  // Number of coarse edges
32 E_C ← [0, ..., 0]                                       // Array of size m_c
33 for b := 0 to n_C do parallel                            // Copy L_t to E_C
34     (d, i, t') ← D[b]
35     E_C[V_C[b]..V_C[b + 1] − 1] ← L_{t'}[i..i + d].keys
36 return G_C = (V_C, E_C)
```

we interleave the contraction of the current clustering with the computation of the next clustering during edge weight aggregation. We set the maximum cluster weight U' to

$$U' \coloneqq fL_2,$$

where f is a constant scaling factor (we follow KaHIP [SS11a] and set $f = \frac{1}{12}$) and L_2 is the maximum block weight in the bipartition. If both blocks have equal weight, we set $L_2 \coloneqq (1+\varepsilon')\lceil c(B)/2 \rceil$, where ε' is based on the adaptive epsilon of Ref. [Sch+16],

$$\varepsilon' \coloneqq \left((1+\varepsilon) \frac{c(V)}{k' \cdot c(V_B)} \right)^{\frac{1}{\lceil \log_2(k/k') \rceil}} - 1.$$

This formulation restricts ε to smaller values at higher recursion levels, such that if each bipartition is ε'-balanced, the resulting partition is ε-balanced. When block weights are uneven (i.e., when k is not a power of two), we adjust the maximum block weights according to their final block counts, as discussed in Section 4.2.3. Note that initial bipartitioning is not guaranteed to produce balanced partitions. Consequently, we can only ensure balance through an additional k-way balancing algorithm during uncoarsening. We describe this step in Section 4.3.4.1.

To bipartition the coarsest graph, we use a pool of the following heuristics: random bipartitioning, greedy graph growing [KK98a], and breadth-first search expansions based on the following strategies: (a) alternating block expansion, (b) always expanding the lighter block, (c) expanding one block until it reaches the maximum block weight, (d) always selecting the block with the larger frontier next, (e) always selecting the block with the smaller frontier next. Each algorithm is executed several times with different random seeds and the best bipartition with the lowest edge cut is selected. More precisely, when $k < 2^{11}$, we perform between 5 and 50 repetitions with each bipartitioner, using the adaptive selection criterion from Mt-KaHyPar [Got+21a]. This criterion estimates the performance of each bipartitioning heuristic by tracking the mean (μ) and variance (σ^2) of bipartition cuts obtained so far by the respective heuristic. After the minimum number of repetitions, additional repetitions are performed only if $\sigma^2 > \left(\frac{\mu - c}{2}\right)^2$, where c is the best edge cut found by any heuristic so far. For larger values of k, we reduce the number of repetitions per heuristic to between 2 and 4 (instead of 5 and 50).

During uncoarsening, we use 2-way FM for refinement [FM82]. We always run 5 rounds of FM, but terminate early if a round improves the bipartition by less than 0.01%. A single round terminates after visiting all vertices, or if 100 consecutive moves fail to improve the bipartition.

4.3.4 Uncoarsening

After initial bipartitioning, we project the resulting partition onto the next finer graph. The projected partition may violate the balance constraint due to changes in $\max_v c(v)$ and because initial bipartitioning does not necessarily produce balanced partitions. If balance is violated, we restore it using a k-way balancing algorithm which employs parallelism over the blocks of the partition. Since the maximum overload of any block is bounded by the maximum vertex weight of the previous graph, we expect that only a few vertices per overloaded blocks need to be moved, thus making our parallelization approach efficient. Finally, we refine the partition using size-constrained label propagation as a k-way refinement step.

4.3.4.1 Balancing

The balancing step redistributes excess weight from overloaded blocks by greedily moving vertices based on their *relative gain*. The algorithm works in three phases and is summarized in Algorithm 5.

Algorithm 5 `Rebalance`(G, Π)

Input : Graph $G = (V, E)$, imbalanced partition $\Pi = (V_1, \ldots, V_k)$ of G, maximum block weight $L_{\max}$, number of threads p.
Output : Balanced partition $\Pi' = (V'_1, \ldots, V'_k)$ of G.

```
1  // Initialize per-block thread-local priority queues
2  P_{B,t} ← new PQ()                          // ∀ block B ∈ Π, thread t ∈ [p]
3  for u ∈ V do parallel                                          // Thread t
4      DisplacingPush(P_{Π[u],t}, Π[u], u)          // Π[u]: block containing u
5  for B ∈ Π do parallel
6      // Merge thread-local priority queues into one double-ended
         priority queue
7      P_B ← Shortest max prefix of P_{B,1}, ..., B_{B,p} with total vertex weight ≥ o(B)
8      // Move vertices
9      while o(B) > 0 do
10         (r, u) ← (P_B.maxKey(), P.maxVertex()) ; P_B.deleteMax()
11         r' ← g_rel(u)        // Recompute rel. gain, denote target block T
12         // Try to move u from B to T subject to L_max
13         if r' ≥ r and Π.move(u, T, L_max) then
14             for unmarked neighbor v ∈ N(u) ∩ B do
15                 DisplacingPush(P_B, B, v) ; Mark v
16         else
17             P_B.push(u, r')
18 return Π
19 Function DisplacingPush(P, B, u)
20     if c(P) < o(B) or g_rel(u) > P.minKey() then
21         P.push(u, g_rel(u))
22         while c(P) − c(P.minVertex()) ≥ o(B) do
23             P.deleteMin()
```

During the first phase (lines 1 to 4), we iterate over all vertices in parallel. Each thread t maintains a thread-local priority queue $P_{B,t}$ for each overloaded block B (i.e., one with $c(B) > L_{\max}$). This queue stores vertices from B, prioritized by their *relative gain*, defined as

$$g_{\text{rel}}(v) \coloneqq \begin{cases} g(v) \cdot c(v) & \text{if } g(v) \geq 0, \\ g(v)/c(v) & \text{if } g(v) < 0, \end{cases}$$

where $g(v)$ is the *absolute gain* of v, i.e., the largest reduction in edge cut when moving vertex v to any block T with $c(T) + c(v) \leq L_{\max}$. Prioritizing vertices by relative gain rather than absolute gain encourages moving fewer heavy vertices instead of many lighter ones if doing so results in a smaller expected edge cut increase. To keep the priority queues small, each thread retains just enough vertices in $P_{B,t}$ to cover the block's overload, defined as

$$o(B) \coloneqq c(B) - L_{\max}.$$

More precisely, we add vertices unconditionally to $P_{B,t}$ until $c(P_{B,t}) \geq o(B)$. After reaching this threshold, a newly encountered vertex from B is added only if its relative gain exceeds that of the lowest-ranked vertex in $P_{B,t}$. Subsequently, vertices are removed so that $c(P_{B,t}) \geq o(B)$ but $c(P_{B,t}) - c(v) < o(B)$ for the lowest-ranked vertex $v \in P_{B,t}$ (lines 19 to 23).

In the second phase (line 7), we employ parallelism over (overloaded) blocks B to merge the thread-local priority queues $P_{B,1}, \ldots, P_{B,p}$ from the first phase into a single double-ended priority queue P_B. Here, p refers to the number of threads. Since the third phase will require access to both the highest- and lowest-ranked vertices in P_B, we use a double-ended priority queue, implemented as two binary heaps with inverse orderings. During merging, we also maintain the invariant that P_B contains just enough vertices to cover B's overload.

The third phase (lines 9 to 17) is responsible for the vertex moves and also employs parallelism over the overloaded blocks B. Iteratively, we extract the highest-ranked vertex v from P_B (line 10). Before moving v, we recompute its relative gain, as its original block may no longer be viable due to other vertex moves. If its relative gain remains unchanged or improves, we attempt the move. This involves a compare-and-swap loop on the target block's weight to prevent overloading due to concurrent moves. If the move succeeds, we insert v's neighboring vertices in block B into P_B following the same procedure as before (line 15). To keep running times low, we only consider each vertex once for insertion. If the relative gain of v worsened, we simply reinsert v into P_B (line 17).

Running Time. We assume that there are no isolated vertices in the graph, i.e., $n < m$. Let $\hat{o}$ be the maximum overload of any block, bounding the sizes of the priority queues. The first phase runs in time

$$\mathcal{O}\left(\underbrace{\left(\frac{m}{p} + \Delta\right)}_{\text{Gain computation}} + \underbrace{\frac{n}{p}\log(\hat{o})}_{\text{PQ inserts}}\right),$$

where the $\log(\hat{o})$ factor arises from the thread-local priority queue operations, and the span Δ accounts for the sequential processing of large neighborhoods during gain computations.

Merging the thread-local priority queues during the second phase requires time

$$\mathcal{O}\left(\left(\frac{k}{p} + 1\right)(p + \hat{o}\log(\hat{o}p))\right),$$

as we can identify the top $\hat{o}$ elements across p PQs in sequential time $\mathcal{O}(p + \hat{o}\log(p\hat{o}))$ and we only employ parallelism over the blocks of the partition. This can be achieved by building a heap H consisting of the highest-ranked element of each PQ (time $\mathcal{O}(p)$), then extracting $\hat{o}$ elements from H (time $\mathcal{O}(\hat{o}\log(p))$). After each extraction, the element is also removed from its original PQ (time $\mathcal{O}(\hat{o}\log(\hat{o}))$) and the new highest-ranked element from that PQ is inserted into H (time $\mathcal{O}(\hat{o}\log(p))$). Since we only employ parallelism over the blocks, we obtain span $\mathcal{O}(p + \hat{o}\log(p\hat{o}))$.

In the third phase, we again parallelize over blocks. This could lead to load imbalance if edges are distributed unevenly across blocks. In the following, we assume that there is some balance; more precisely, that the vertices assigned to a single block are incident to at most $(1+\delta)\frac{m}{k}$ edges, for some $\delta > 0$. The third phase then repeatedly extracts vertices from its thread-local double-ended priority queue, recomputes its relative gain, and reinserts the vertex or moves the vertex and attempts to insert its neighboring vertices. Computing gain values for all vertices of a block once requires time $\mathcal{O}\big((1+\delta)\frac{m}{k}\big)$, while removing or inserting

a vertex into the queue requires time $\mathcal{O}(\log(\hat{o}))$. Recall that vertices are reinserted into the queue if their relative gain has worsened. Since vertices are only moved from overloaded to non-overloaded blocks, this can only happen when the original target block of a vertex is no longer viable, i.e., at most $\min\{k, \Delta\}$ times (since the gain for all non-adjacent blocks is the same). Overall, we therefore obtain time

$$\mathcal{O}\left(\underbrace{\frac{k}{p}\min\{k,\Delta\}(1+\delta)\frac{m}{k}}_{\text{Gain computations}} + \underbrace{\frac{k}{p}\min\{k,\Delta\}\hat{o}\log(\hat{o})}_{\text{PQ operations}}\right).$$

If we assume uniform vertex weights, overloads are small – specifically, $\hat{o} \in \Theta(1)$. With the additional assumption that $\delta \in \Theta(1)$, the running time simplifies to

$$\mathcal{O}\left(\frac{m}{p}\min\{k,\Delta\}\right).$$

Putting it all together, we obtain runtime

$$\mathcal{O}\left(\frac{m}{p}\min\{k,\Delta\} + \Delta + p\right)$$

with span $\mathcal{O}(\Delta + p)$. While this analysis may appear pessimistic for large k and large Δ, we observed that vertices are only rarely reinserted into the priority queue during the third phase, so the worst-case factor $\min\{k, \Delta\}$ does not become a practical bottleneck.

4.3.4.2 Refinement

Label Propagation Refinement. We refine the partition using the same parallelization of size-constrained label propagation as described in Section 4.3.1. Initially, vertices are assigned to clusters corresponding to the partition blocks. In contrast to the coarsening phase, we strictly enforce the maximum cluster weight (set to the maximum block weight $L_{\max}$) using an atomic compare-and-swap instruction. For a small number of blocks (in our implementation, $k \leq 128$), we align each block weight to its own cache line to avoid false sharing when updating block weights after a vertex move. Note that this is not necessary during the coarsening phase, since the number of cluster weights (initially, n) is usually much larger there. We perform up to 5 rounds of label propagation but terminate early if no vertex is moved during a round. In practice, however, the early termination condition is rarely triggered, and the algorithm almost always performs all 5 rounds.

Following the analysis from Section 4.3.1, refinement takes time $\mathcal{O}\left(\frac{m}{p} + \Delta\right)$ with span $\mathcal{O}(\Delta)$. In the worst case, each thread has to allocate a vector of size $\mathcal{O}(k)$ for thread-local gain aggregations. Thus, its memory requirement is $\mathcal{O}(kp)$.

FM Refinement. For some parts of our experimental evaluation in Section 4.4, we use additional FM refinement [FM82]. This is arguably one of the most widely used refinement algorithms, see also Section 3.4.3. To this end, we have re-implemented a simplified version of the parallelized [ASS20; Got+21a] multi-try [SS11a] FM algorithm, which we outline below.

The algorithm starts by collecting all border vertices of the current partition into a queue Q. Then, each thread t starts a round of local search by pulling (up to) 10 *seed* vertices from Q and inserting them into a thread-local gain-based priority queue P_t. The priority of each vertex is its highest gain to any block that would not become overloaded when moving

the vertex to the block. Note that, as usual in FM refinement, this gain might be negative. Throughout the algorithm, each vertex may only be contained in one priority queue at a time, which is ensured by requiring threads to acquire *ownership* over a vertex before inserting it into P_t. This is implemented by maintaining one lock per vertex. Vertices currently owned by other threads are ignored. After initializing P_t, threads independently and asynchronously simulate the vertex moves. After each move, the gains of owned neighboring vertices are updated, while unlocked neighbors are acquired and inserted into P_t. Whenever the simulated vertex moves yield a positive total gain value, the moves are applied to the partition so that they become visible to the other threads. Threads stop their current local search once the adaptive stopping criterion of Ref. [OS10] indicates that finding further improvements is unlikely. All owned vertices are then unlocked, so that they can be acquired by other threads, and the thread starts a new round until Q is empty.

We highlight two potential downsides of this implementation. First, while each thread restricts its local moves to those that do not violate the balance constraint, multiple threads in combination can cause balance violations by moving vertices to the same block concurrently. We observed that these violations are usually small in practice, so that we do not try to prevent them, but re-run our balancing algorithm described in Section 4.3.4.1 afterwards to ensure a balanced output. Secondly, recall that threads compute their sets of vertex moves in isolation and apply them unconditionally to the shared partition once the total sum of gain values is positive. This might cause conflicts if neighboring vertices are moved concurrently by different threads. The implementations of Ref. [ASS20] and Ref. [Got+21a] present techniques to resolve these conflicts, while we do not. This simplifies the algorithm. However, as shown in Section 4.4, we still achieve competitive results.

4.4 Experiments

In this section, we present a detailed experimental evaluation of our shared-memory graph partitioner KaMinPar. We first discuss the methodology and general setup of the experiments. Next, we compare its performance against state-of-the-art sequential and shared-memory parallel graph partitioners in Section 4.4.1. Unlike previous work on graph partitioning, we look at both small and large values of k, ranging from 2 to 2^{20}. Finally, we evaluate KaMinPar's algorithmic components and their scalability in Section 4.4.2.

Setup. We have implemented KaMinPar in `C++` and used Intel's oneAPI TBB library [TBB] for parallelization. The binaries are built using `GCC` with flags `-O3 -march=native`. We perform our scalability experiments on Machine A and all other experiments on Machine B. The specifications of both machines are provided in Table 2.2. Machine B is a cluster of compute nodes equipped with either 96 GiB or 192 GiB of RAM, which we allocate depending on the value of k used in the respective experiment: 96 GiB RAM nodes for $k \in \{2, 4, 8, 16, 32, 64\}$, and 192 GiB RAM nodes for $k \in \{2^{11}, 2^{14}, 2^{17}, 2^{20}\}$. To mimic the performance on typical commodity machines, we only use 10 cores on a single socket of Machine B. We indicate this by annotating parallel partitioners with a numerical suffix reflecting the number of cores used to obtain the results. For example, we denote KaMinPar 10 when using 10 cores. Sequential partitioners have no numerical suffix.

We generally set $\varepsilon = 3\%$ as the maximum allowed imbalance for all experiments. This imbalance value is commonly used in the literature [Got+21a; KK98c; Sch+16; SS11a]. We perform 5 repetitions for each instance (combination of a graph and a number of blocks k) with different random seeds and average over measured quantities (e.g., edge cuts and

Table 4.1 Regular graphs of Benchmark Set A (**bold**: subset B), characterized by the number of vertices (n), the number of undirected edges (m), and the maximum degree (Δ).

Graph	$n/10^3$	$m/10^3$	Δ	Graph	$n/10^3$	$m/10^3$	Δ
Artificial [Bad+12; Fun+18]							
Rgg2e26-2D	67 109	574 554	45	**Rgg2e26-3D**	67 109	377 952	34
Del2e24-2D	16 777	50 332	26				
Biology [DH11]							
KmerV1r	214 004	232 705	8	**KmerA2a**	170 372	179 942	40
KmerP1a	138 896	148 465	40	KmerU1a	64 678	66 394	35
KmerV2a	53 500	57 076	39	Cage15	5 155	47 022	46
Finite Element [Bad+12; DH11]							
Queen4147	4 147	162 676	80	**HV15R**	2 017	162 358	492
CubeCoupDt6	2 165	62 521	67	Bump2911	2 911	62 409	194
Flan1565	1 565	57 921	80	MLGeer	1 504	54 688	73
DielFilterV3	1 103	44 102	269	LongCoupDt6	1 470	42 809	755
ChannelB050	4 802	42 681	18	Audikw1	944	38 354	344
Serena	1 391	31 570	248	Geo1438	1 438	30 859	56
Hook1498	1 498	29 710	92	BoneS10	915	27 277	80
AfShell10	1 508	25 582	34	Ldoor	952	22 785	76
Optimization [Bad+12]							
Nlpkkt240	27 994	373 239	27	**Nlpkkt200**	16 240	215 993	27
Road [Bad+12]							
EuropeOSM	50 912	54 055	13	AsiaOSM	11 951	12 712	9
Semiconductor [Bad+12; DH11]							
Stokes	11 450	257 981	1 728	VasStokes4M	4 382	97 709	1 139
VasStokes2M	2 147	48 352	1 307	NV2	1 454	25 637	83

running times) using the arithmetic mean. When aggregating results over multiple instances, we use the geometric mean for all quantities.

Instances. We evaluate running times and partition quality on a benchmark set consisting of the 72 graphs listed in Tables 4.1 and 4.2. We refer to this benchmark set as Benchmark Set A. The set includes randomly generated as well as real-world graphs sourced from various application domains. The graphs vary significantly in size, ranging from 5.4 million to 1.8 billion undirected edges and 65.5 thousand to 214.0 million vertices, with maximum degrees spanning from just 8 to 10.3 million. We roughly characterize the graphs as *regular*[2] or *irregular* based on their maximum degree, since we found that this property can considerably impact the performance of different partitioners. Regular graphs (type R, listed in Table 4.1) have a relatively small maximum degree, while irregular graphs (type I, listed in Table 4.2) contain a small number of high-degree vertices that account for a considerable portion of the total edges. For example, in the Mawi2015 graph, nearly half of all edges are incident to a single vertex. Such vertex hubs are commonly found in web- and social graphs, as well as random graph models that mimic them. Overall, 39 of the 72 graphs in the benchmark set are classified as irregular, while the remaining 33 are classified as regular.

[2] We are aware that the term *regular* is commonly used for graphs in which every vertex has the same degree. In this thesis, we use the term only as a coarse label for degree heterogeneity as described here.

Table 4.2 Irregular graphs of Benchmark Set A (**bold**: subset B), characterized by the number of vertices (n), the number of undirected edges (m), and the maximum degree (Δ).

Graph	$n/10^3$	$m/10^3$	Δ	Graph	$n/10^3$	$m/10^3$	Δ
Artificial [Bad+12; Fun+18; KGB15]							
RMat2e25	27 090	268 416	24 179	**Rhg10e8**	10 000	199 592	96 981
Kron2e21	2 097	91 041	213 904	Kron2e20	1 049	44 619	131 503
Rhg2e23	8 389	32 082	415 850	Kron2e19	524	21 781	80 674
RMat2e16	66	16 777	26 068				
Brain [RA15]							
Bn87128519	862	169 368	7 397	Bn87126525	976	146 109	8 009
Bn87122310	924	94 371	8 135	Bn87123142	847	53 825	8 356
Bn87117515	892	48 670	4 546				
Compression [Jez15]							
Proteins1GB9	14 538	74 309	660 097	Proteins1GB7	2 826	43 857	373 750
Dna1GB9	3 233	25 285	342 348	English1GB7	802	13 063	100 816
Sources1GB9	2 792	12 188	37 881	Sources1GB7	899	7 272	32 091
Semiconductor [DH11]							
Circuit5M	5 558	26 984	1 290 500				
Social [Lab; Les; RA15]							
Friendster	65 608	1 806 067	5 214	**Twitter2010**	41 652	1 202 513	2 997 487
Sinaweibo	58 656	261 321	278 489	Orkut	3 073	117 185	33 313
Hollywood2011	2 181	114 493	13 107	LiveJournal	4 037	34 681	14 815
Flickr	1 715	15 555	27 236	IMDB2021	2 996	5 369	833
Web [DH11; Lab]							
Sk2005	50 636	1 810 063	8 563 816	**It2004**	41 292	1 027 475	1 326 744
Webbase2001	118 142	854 810	816 127	**Uk2005**	39 460	783 027	1 776 858
Arabic2015	22 744	553 903	575 628	IndoChina2004	7 415	150 985	256 425
Mawi2015	22 915	24 562	10 327 637				
Wiki [Lab]							
EnWiki2022	6 492	144 589	231 674	DeWiki2013	1 532	33 093	118 246
FrWiki2013	1 352	31 037	148 758	ItWiki2013	1 017	23 430	91 517
EsWiki2013	973	21 185	145 310				

For our large k and scalability experiments, we only use the 20 largest graphs (by number of edges) of Benchmark Set A. We denote this subset as Benchmark Set B. The included graphs are **bolded** in Tables 4.1 and 4.2.

4.4.1 Experimental Evaluation Against the State of the Art

In this section, we assess the performance of KaMinPar by comparing it to state-of-the-art graph partitioners, thus positioning it within the broader landscape of partitioners. We first evaluate KaMinPar using small values of k, specifically $k \in \{2, 4, 8, 16, 32, 64\}$, in Section 4.4.1.1. These values are commonly used when benchmarking graph partitioners, so that existing partitioners are well-optimized for them. This allows for a fair and direct comparison. Subsequently, we extend our evaluation to significantly larger values of k, specifically $k \in \{2^{11}, 2^{14}, 2^{17}, 2^{20}\}$, in Section 4.4.1.2. We anticipate considerable performance advantages in this setting due to deep multilevel graph partitioning.

We compare our algorithm against several partitioners that cover the state-of-the-art for

sequential and shared-memory parallel graph partitioning. A more detailed description of each partitioner is available in Chapter 2. Here, we only provide a brief summary for each partitioner:

- Metis [KK98a] (v5.1.0): A well-established sequential multilevel graph partitioner which uses heavy-edge and two-hop matching for coarsening and FM refinement during uncoarsening. Metis is generally considered to be one of the fastest sequential general-purpose multilevel graph partitioners.
- Mt-Metis [LK13; LK16] (v0.7.2): The multi-threaded variant of Metis. It employs a parallelized version of heavy-edge matching and two-hop matching for coarsening. We evaluate three configurations differing in refinement strategies and partitioning modes: in the direct k-way mode, Mt-Metis invokes the sequential Metis partitioner to compute a k-way initial partition. The initial partition is improved using greedy refinement (denoted Mt-Metis-G) or parallel hill-climbing refinement (denoted Mt-Metis-HC) during uncoarsening. Parallel hill-climbing achieves improved partition quality but is also slightly slower [LK16]. Additionally, we evaluate its recursive bipartitioning mode, denoted Mt-Metis-R. This mode is particularly relevant as a baseline for large k partitioning.
- KaHIP [SS11a] (v3.18): A sequential graph partitioner designed for high quality partitioning. KaHIP implements various algorithms for coarsening and refinement, which can be configured via *presets*. We evaluate its `fsocial` preset (denoted KaHIP-fsocial), which balances speed and partition quality, using size-constrained label propagation for both coarsening and refinement. The initial partition is computed through direct k-way partitioning based on recursive bipartitioning of the coarsest graph. Note that KaHIP also offers stronger presets with improved partition quality, but those are also much slower (up to orders of magnitude [Got+24a]).
- Mt-KaHIP [ASS20] (v1.0): The multi-threaded variant of KaHIP, which also utilizes parallel size-constrained label propagation for both coarsening and refinement. Initial partitioning is performed by direct k-way partitioning via sequential KaHIP. We include two configurations differing in refinement methods: Mt-KaHIP-LP only uses label propagation, while Mt-KaHIP-FM uses additional parallel FM refinement.
- PuLP [SMR14] (v0.11): A multi-threaded single-level graph partitioner optimized for partitioning social networks. We include it since it is very fast and finds partitions of comparable quality to Metis and KaHIP on social graphs [SMR14], making it a high-quality representation of single-level partitioners. Additionally, it is also based on parallel label propagation, which is arguably the most important building block of KaMinPar.
- Mt-KaHyPar [Got+21a] (v1.4): A multi-threaded partitioner that was originally designed for high-quality hypergraph partitioning, but has recently been extended to support an optimized graph partitioning mode [Heu22]. We include the configuration featuring parallel FM refinement (denoted Mt-KaHyPar-FM). As with KaHIP, Mt-KaHyPar also supports techniques aimed towards higher partition quality, notably n-level coarsening [Got+22], parallel flow-based refinement [GHS22] and parallel unconstrained FM refinement [MGS]. These techniques improve partition quality beyond KaMinPar's capabilities. However, these techniques are also orthogonal to our contributions and thus not the focus of this work. Moreover, they are much slower.

  Moreover, we briefly compare against the GPU-focused Jet [Gil+24] partitioner[3], whose

[3] Commit hash `9ca1fd0` in `github.com/sandialabs/Jet-Partitioner`.

Table 4.3 Geometric mean running time and solution quality of KaMinPar and competitors on Benchmark Set A for $k \in \{2, 4, 8, 16, 32, 64\}$. To ensure comparability, running time aggregates only include instances for which all partitioners produced a result. The number of such instances, along with the total number of instances, is shown in the last row. Solution quality is measured relative to KaMinPar (lower is better) and includes only instances for which the respective algorithm successfully computed a partition. Since different sets of instances are excluded for each competitor, solution quality cannot be directly compared between them. Note that imbalanced partitions are included in the relative cut aggregates for competitors. Column #Inf. shows each algorithm's robustness by counting the instances for which a partitioner crashed or computed an imbalanced partition.

	T (all)		T ($m \geq 10^8$)		T ($m \geq 10^9$)			
Algorithm	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.	Rel. Cut	#Inf.
KaMinPar 10	**1.44 s**	**1.00**	**2.78 s**	**1.00**	**5.99 s**	**1.00**	1.00	**0**
KaMinPar-FM 10	3.97 s	2.76	6.87 s	2.47	8.21 s	1.37	**0.84**	**0**
Mt-Metis-HC 10	3.86 s	2.68	6.92 s	2.49	22.51 s	3.76	0.98	212
Mt-Metis-G 10	3.09 s	2.15	5.87 s	2.11	20.06 s	3.35	1.01	162
Mt-Metis-R 10	5.49 s	3.82	11.53 s	4.15	41.12 s	6.87	1.03	152
Mt-KaHyPar-FM 10	8.24 s	5.73	15.82 s	5.69	53.04 s	8.86	0.88	60
Mt-KaHIP-FM 10	7.18 s	4.99	13.11 s	4.72	29.73 s	4.97	0.92	12
Mt-KaHIP-LP 10	5.28 s	3.67	9.00 s	3.24	25.19 s	4.21	1.00	13
PuLP 10	1.62 s	1.13	3.12 s	1.12	14.41 s	2.41	2.58	65
Metis	7.11 s	4.95	13.83 s	4.98	47.48 s	7.93	1.07	24
KaHIP-fsocial	18.54 s	12.89	37.60 s	13.53	206.86 s	34.56	1.03	31
#Instances	302 of 432		152 of 234		12 of 48			

refinement algorithm is tailored to bulk-synchronous machine models (see Section 3.4.4 for a description). The official Jet implementation uses Kokkos [ETS14; Tro+22] for portability across CPUs and GPUs, which introduces some abstraction overhead compared to the other purely CPU-focused implementations. We later extend this comparison to Jet's refinement algorithm via our distributed partitioner in Section 7.3.

We exclude the distributed graph partitioners ParMetis [KK99] and Pt-Scotch [CP08] from our comparisons, as they are slower than Mt-Metis while producing partitions of similar quality [LK13]. Additionally, we omit ParHIP [MSS17] since it is outperformed by Mt-KaHIP [ASS20].

4.4.1.1 Running Time and Solution Quality for Small k

In Figure 4.6, Figure 4.7 and Table 4.3, we present a detailed comparison of partition quality and running time between KaMinPar and competing sequential and multi-threaded partitioners. Alongside the standard KaMinPar, which uses only label propagation for refinement, we also evaluate a variant equipped with additional parallel FM refinement (denoted as KaMinPar-FM). To this end, we have re-implemented the parallelized multi-try FM algorithm [SS11a] following the work of Gottesbüren et al. [Got+21a]. We describe our implementation in Section 4.3.4.2. We include this evaluation to demonstrate that the

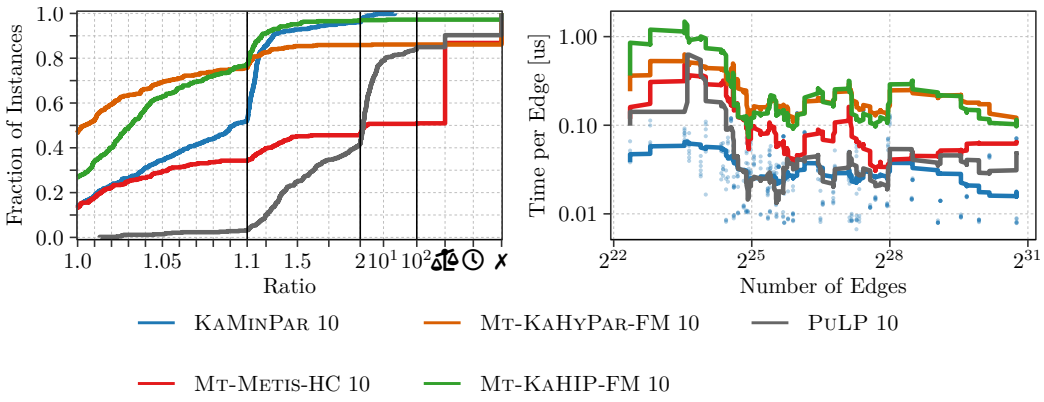

Figure 4.6 Partition quality (**left**) and running times (**right**) for KaMinPar 10 and multi-threaded competitors on Benchmark Set A for $k \in \{2, 4, 8, 16, 32, 64\}$. Running times are plotted as per-instance time-per-edge (only shown for KaMinPar) with a right-aligned rolling geometric mean over 50 instances (shown for all partitioners).

superior partition quality achieved by competitors employing stronger refinement techniques arises from orthogonal improvements, and that incorporating FM refinement into KaMinPar yields comparable partition quality.

As indicated in Table 4.3, not all competitors manage to partition all graphs in our benchmark set. On some instances, they run out of memory or crash due to segmentation faults. Whenever we argue running times in the following discussion, we therefore only aggregate over instances for which *all* competitors computed a partition, allowing a total ranking of the algorithms with comparable absolute geometric mean running times. Whenever we compare the solution quality (i.e., edge cuts) of two partitioners, we aggregate over instances for which both partitioners computed a solution, effectively ignoring the balance constraint. Note that KaMinPar always computes balanced solutions.

Comparison Against Multi-Threaded Multilevel Partitioners. As shown in Table 4.3, KaMinPar 10 is the fastest algorithm in our comparison, achieving a geometric mean running time of 1.44 s. In contrast, the fastest competing multilevel algorithm, Mt-Metis-G 10, has a geometric mean running time of 3.09 s, which is 2.15 times slower than KaMinPar 10. Additionally, KaMinPar 10 is more than 3 times faster than Mt-KaHIP-LP 10 (5.28 s). Since both of these competitors also employ simple greedy refinement techniques, their edge cuts are comparable. The geometric mean edge cut of KaMinPar 10 is 0.90% and 0.12% lower than those of Mt-Metis-G 10 and Mt-KaHIP-LP 10, respectively. However, note that out of 432 instances, Mt-Metis-G 10 crashes on 41 instances and computes imbalanced partitions for 121 instances. As shown in Figure 4.11 (left), these balance violations can be quite severe, with imbalances up to 161% instead of the allowed imbalance of $\varepsilon = 3\%$. This makes a fair comparison difficult.

A notable observation is the performance penalty incurred by recursive bipartitioning. Mt-Metis-R 10, which uses recursive bipartitioning, has a geometric mean running time of 5.49 s, which is 1.78 times slower than Mt-Metis-G 10 (3.09 s) and 3.82 times slower than KaMinPar 10 (1.44 s). This is most likely because recursive bipartitioning repeats the multilevel cycle $\log(k)$ times to compute a k-way partition, whereas direct k-way partitioning (Mt-Metis-G) and deep multilevel partitioning (KaMinPar) only perform a single multilevel cycle. Furthermore, we observe that Mt-Metis-R 10 produces worse edge cuts. Its average

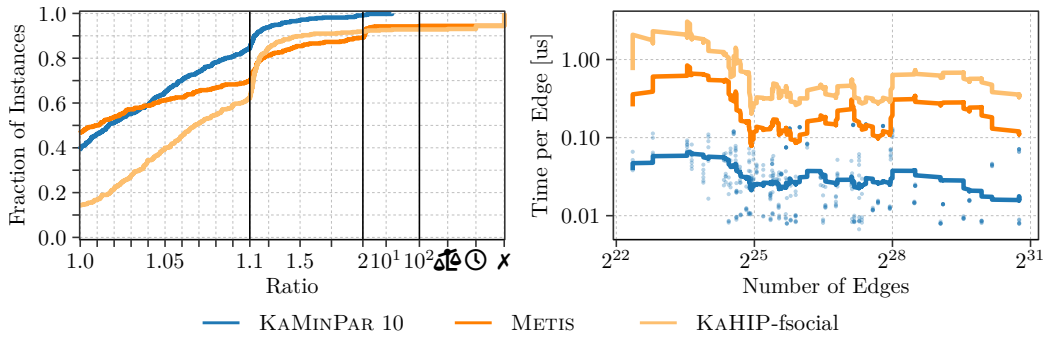

Figure 4.7 Partition quality (**left**) and running times (**right**) for KaMinPar 10 and sequential competitors on Benchmark Set A for $k \in \{2, 4, 8, 16, 32, 64\}$. Running times are plotted as per-instance time-per-edge (only shown for KaMinPar) with a right-aligned rolling geometric mean over 50 instances (shown for all partitioners).

edge cut is 1.82% larger than that achieved by Mt-Metis-G 10, and 3.00% larger than that of KaMinPar 10. We attribute this to the fact that Mt-Metis-R is limited to 2-way refinement, and thus lacks a broader view on the k-way partition. Note that deep multilevel graph partitioning does not exhibit these restrictions.

Evaluating multi-threaded multilevel competitors with stronger refinement techniques, we observe even greater speedups: KaMinPar 10 is 2.68, 4.99, and 5.73 times faster than Mt-Metis-HC 10 (3.86 s, parallel hill-climbing refinement), Mt-KaHIP-FM 10 (7.18 s, parallel FM refinement), and Mt-KaHyPar-FM 10 (8.24 s, parallel FM refinement), respectively. As can be seen in Figure 4.6 (right), these speedups persist across both smaller and larger graphs. However, these competitors also achieve better partition quality. On average, Mt-Metis-HC 10, Mt-KaHIP-FM 10 and Mt-KaHyPar-FM 10 compute 2.22%, 8.70%, and 13.20% lower edge cuts than KaMinPar 10, respectively. This trend can also be observed in the performance profile Figure 4.6 (left), where Mt-KaHyPar-FM 10 finds the best partitions for nearly half of all instances, while Mt-KaHIP-FM 10 finds the best partitions for almost 30% of the instances. Meanwhile, the edge cuts found by KaMinPar 10 are at least 10% larger than the best edge cuts found on more than half of all instances. A fair comparison against Mt-Metis-HC 10 remains difficult, as it often crashes or produces partitions violating the balance constraint even with the stronger refinement algorithm. More precisely, out of 432 instances, it crashes on 57 instances. Out of the remaining 375 instances, it computes imbalanced partitions for 155 instances. Yet, these (often imbalanced) partitions are only 2% smaller than the (balanced) partitions computed by KaMinPar 10, on average. Mt-KaHyPar 10 does not produce imbalanced solutions, but crashes on 60 instances due to memory limitations.

Furthermore, note that stronger refinement techniques such as FM are orthogonal to the other components of multilevel graph partitioners. Indeed, equipping KaMinPar 10 with additional FM refinement (denoted KaMinPar-FM 10 in Table 4.3) improves its edge cuts by 19.27% on average, while increasing its running time by a factor of 2.76. With this configuration, KaMinPar-FM 10 outperforms Mt-KaHIP-FM 10 and Mt-KaHyPar-FM 10 in *both* partition quality and running time simultaneously, producing 10.02% and 6.92% lower edge cuts while being 1.81 and 2.08 times faster, on average, respectively.

Comparison Against Sequential Multilevel Partitioners. Results for sequential multilevel graph partitioners are summarized in Table 4.3 and Figure 4.7. Using 10 cores, KaMinPar 10

(geometric mean running time 1.44 s) achieves a moderate speedup of 4.95 over the sequential Metis algorithm (7.11 s) and a considerable speedup of 12.89 over KaHIP-fsocial (18.54 s). Metis is well-optimized and, as a sequential algorithm, does not suffer overheads due to parallelization. However, its matching-based coarsening scheme only halves the number of vertices at each level of the graph hierarchy, whereas KaMinPar can shrink the graph much faster. Since KaHIP-fsocial employs similar building blocks – using label propagation for both coarsening and refinement – we attribute the observed speedup (which exceeds the number of cores used by KaMinPar) primarily to improved cache efficiency during coarsening from vertex reordering and a faster graph contraction algorithm, confer Section 4.3.1. Additionally, KaHIP-fsocial lacks two-hop clustering, causing its coarsening to converge early on some irregular graphs. Looking at partition quality, KaMinPar finds moderately lower edge cuts when aggregating over the entire benchmark set: by 6.79% and 3.38% on average compared to Metis and KaHIP-fsocial, respectively. However, as can be seen in Figure 4.7 (left), Metis finds the best partitions for roughly 45% of the instances, whereas KaMinPar 10 and KaHIP-fsocial only find the best partitions for 40% and 15% of the instances, respectively. Consequently, we conclude that neither partitioner consistently outperforms the others. Rather, the relative quality obtained by each partitioner depends on the type of the graph, warranting a more focused comparison in subsequent paragraphs.

Comparison Against Multi-Threaded Single-Level Label Propagation. Perhaps surprisingly, KaMinPar 10 also exhibits a moderate speed advantage (factor of 1.13) over the single-level partitioner PuLP 10, with geometric mean running times of 1.44 s and 1.62 s, respectively. This speedup likely stems from PuLP's initial random vertex assignment followed by an unconstrained label propagation phase for community detection (see Section 3.6.2 for a more detailed description of PuLP). During this phase, the absence of a size constraint often leads to imbalanced partitions. As a result, PuLP requires a subsequent balancing phase, using additional label propagation iterations to enforce the maximum block weight $L_{\max}$. At the typical imbalance parameter of $\varepsilon = 3\%$, this balancing phase can be computationally expensive, as it might require many iterations. On some instances, we have observed that PuLP terminates without reaching a balanced partition, see Figure 4.11 (left). As a single-level partitioner, PuLP further lacks the coarser perspective on the solution landscape that the multilevel scheme provides. Consequently, one would anticipate significantly higher edge cuts. Table 4.3 confirms this, showing that PuLP produces edge cuts that are, on average, 2.58 times larger than those of KaMinPar (and, by extension, the other multilevel algorithms by similar factors). This is further illustrated in Figure 4.6 (left). PuLP 10's edge cuts are at least twice as large as the best edge cuts found on more than 50% of all instances. On roughly 5% of instances, it computes partitions with edge cuts at least one order of magnitude larger than the best partition found by one of the multilevel algorithms. In the context of multilevel graph partitioning, where computationally expensive techniques often only improve solution quality by a few percentage points, this is considered a lot. Therefore, PuLP's partition quality is not competitive with the multilevel approaches that are the focus of this work. On the other hand, our results indicate that KaMinPar can achieve the same running time without this trade-off in solution quality.

Detailed Partition Quality Evaluation. Figures 4.6 and 4.7 show that all multilevel algorithms, including KaMinPar, compute partitions with best-in-class edge cuts for some benchmark instances but perform subpar on others. This performance variability suggests different strengths and weaknesses of each competitor depending on graph characteristics.

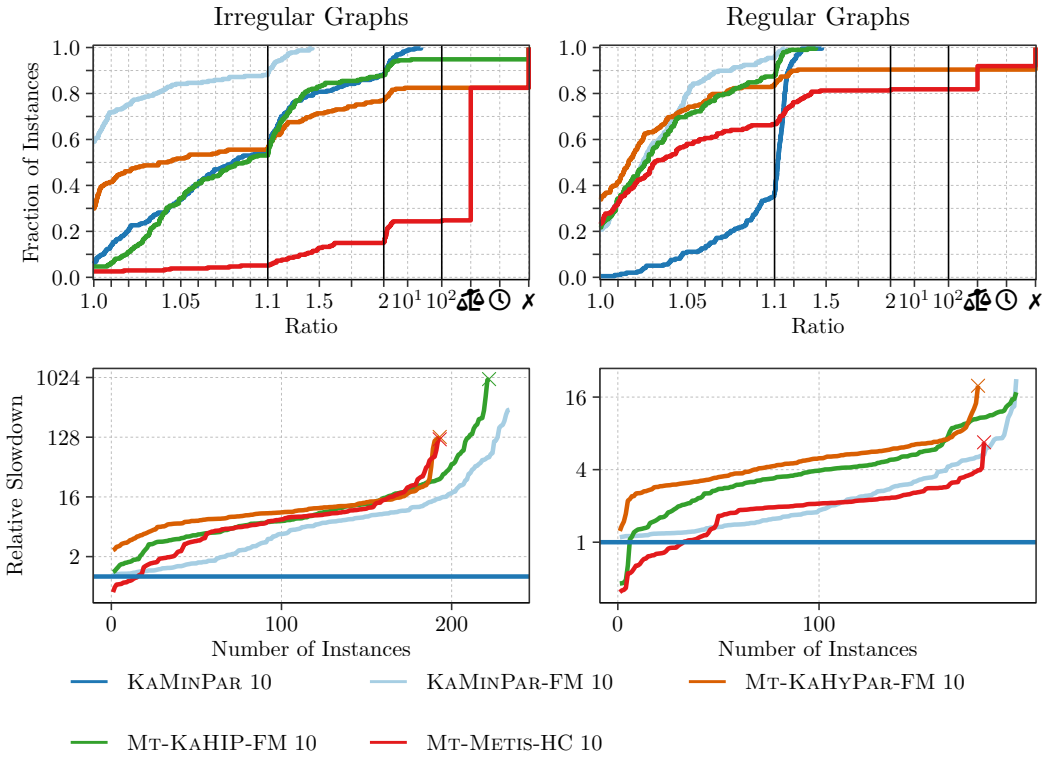

Figure 4.8 Comparison of partition quality and relative running times between KaMinPar 10, KaMinPar-FM 10, and multi-threaded competitors on Benchmark Set A for $k \in \{2, 4, 8, 16, 32, 64\}$. Results are shown separately for *irregular* graphs (**left**, characterized by large maximum degrees) and *regular* graphs (**right**, characterized by small maximum degrees), following the classification between Table 4.1 and Table 4.2. Running times are plotted relative to KaMinPar 10 on a logarithmic scale.

We therefore conduct a more focused analysis of KaMinPar's relative performance to competitors across different graph classes (Figure 4.8) and types (Figure 4.9 and Figure 4.10) next, using the classification in Table 4.1 and Table 4.2. For clarity, we limit the comparison to the strongest configuration of each multi-threaded competitor, i.e., Mt-Metis-HC 10, Mt-KaHyPar-FM 10, and Mt-KaHIP-FM 10, and benchmark against KaMinPar 10 both with and without additional FM refinement.

Focusing on irregular graphs (Figure 4.8, left), KaMinPar 10 (i.e., without FM refinement) computes partitions of similar quality to competitors for a considerable fraction of instances. While the best edge cuts are mostly found by KaMinPar with FM refinement (59.0% of the instances) and Mt-KaHyPar-FM 10 (29.9% of the instances), it computes edge cuts which are at most 10% higher than the best edge cut found by any algorithm for 54.7% of the instances, which is in line with Mt-KaHIP-FM 10 and Mt-KaHyPar-FM 10 (53.0% and 55.6%, respectively). However, it produces much higher cuts on a few instances, as indicated by the longer tail in the performance profile (note that the third section of the x-axis has a logarithmic scale). Breaking instances further down into domain classes, Figure 4.9 reveals that these instances are primarily from the Compression class, i.e., graphs deduced from the text recompression technique [Jez15]. Although few in number, they have a considerable impact on the geometric mean edge cut, which is 9.21% and 16.02% larger than that of Mt-KaHIP-FM 10 and Mt-KaHyPar-FM 10, respectively. However, with FM

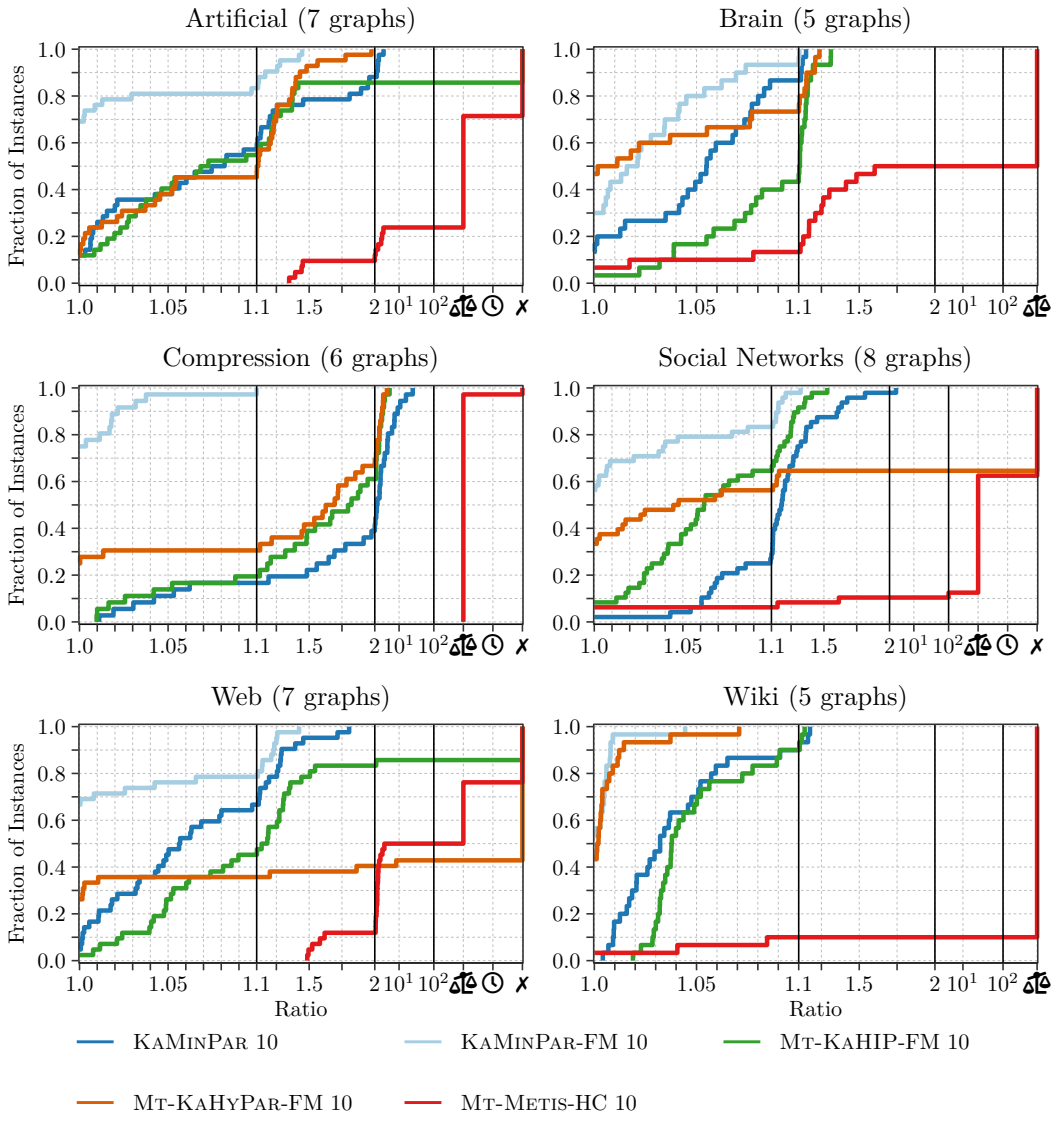

Figure 4.9 Comparison of partition quality between KaMinPar 10, KaMinPar-FM 10, and multi-threaded competitors on the **irregular** graphs of Benchmark Set A for $k \in \{2, 4, 8, 16, 32, 64\}$. Each performance profile includes only graphs of a specific type, following the classification of Table 4.2. We omit the *irregular semiconductor* category since it only includes a single graph.

refinement, KaMinPar seems to perform considerably better than the competing algorithms on these graphs. In general, FM refinement drastically improves the partition quality of KaMinPar on irregular graphs, reducing its geometric mean edge cut across all instances by 28.19%. Consequently, KaMinPar-FM 10 achieves edge cuts that are 14.06% and 18.43% lower than those of Mt-KaHyPar-FM 10 and Mt-KaHIP-FM 10, respectively.

Mt-Metis-HC 10 performs the worst on this subset. It computes balanced partitions for only 24.8% of the irregular instances, while computing imbalanced partitions for 57.7% and crashing on 17.5% of the instances. As shown in Figure 4.9, these performance issues span most domain categories. Only within the Brain and Web categories does it compute

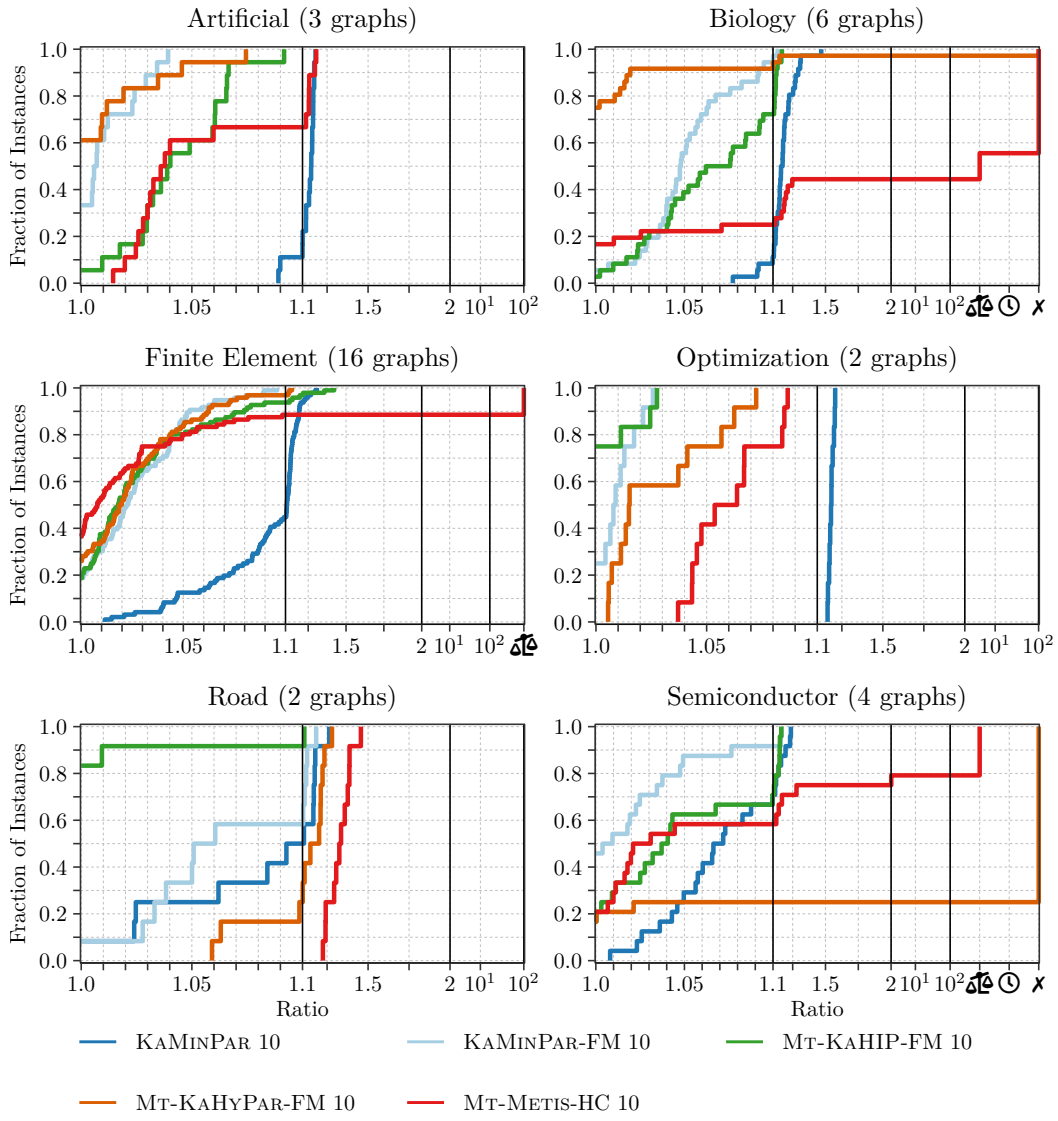

Figure 4.10 Comparison of partition quality between KaMinPar 10, KaMinPar-FM 10, and multi-threaded competitors on the **regular** graphs of Benchmark Set A for $k \in \{2, 4, 8, 16, 32, 64\}$. Each performance profile includes only graphs of a specific type, following the classification of Table 4.1.

balanced partitions for at least half of the instances. However, even ignoring that most of its partitions are imbalanced, its geometric mean edge cut is 2.04% and 33.68% larger than that of KaMinPar without and with FM refinement, respectively.

As shown in Figure 4.8 (bottom left), the running time of KaMinPar is a considerable improvement over competitors. More precisely, KaMinPar 10 (KaMinPar-FM 10) is 4.51 (1.22), 7.75 (2.10), and 7.77 (2.11) times faster than Mt-Metis-HC 10, Mt-KaHyPar-FM 10, and Mt-KaHIP-FM 10, respectively, on irregular graphs.

On regular graphs, KaMinPar 10 computes larger average edge cuts than all competitors: by 6.89%, 10.23%, and 8.14% compared to Mt-Metis-HC 10, Mt-KaHyPar-FM 10, and Mt-KaHIP-FM 10, respectively. Notably, Mt-Metis-HC 10 performs much better on

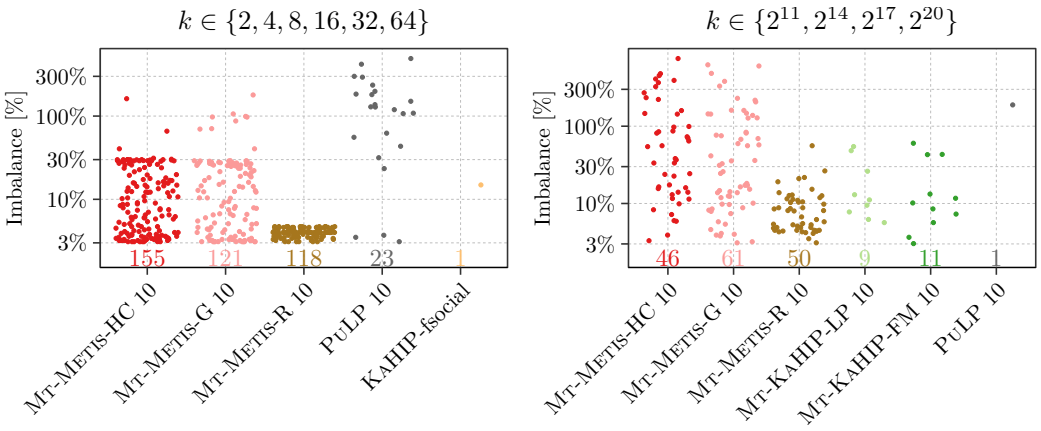

Figure 4.11 Distribution of imbalance percentages for partitions exceeding the $\varepsilon = 3\%$ balance constraint, for smaller $k \in \{2, 4, 8, 16, 32, 64\}$ on Benchmark Set A (**left**) and larger $k \in \{2^{11}, 2^{14}, 2^{17}, 2^{20}\}$ on Benchmark Set B (**right**). Recall that we always perform 5 repetitions per instance and partitioner, and consider an instance imbalanced if all 5 repetitions yielded an imbalanced partition. We plot the *minimum* imbalance out of the 5 repetitions for such instances and only include partitioners which produced at least one imbalanced result. The numeric labels above the x-axis indicate the total number of imbalanced instances produced by each partitioner (out of 432 resp. 80 instances).

regular rather than irregular graphs, computing feasible partitions for 81.8% of the instances and finding the best partitions for 21.7% of the instances. This is expected, as the authors of Mt-Metis primarily evaluated such graphs in their papers [LK13; LK16]. While the relative differences in average edge cuts to competitors on regular and irregular graphs appear similar, the performance profiles reveal distinct patterns. As highlighted above, KaMinPar computes partitions of similar quality on many irregular graphs but struggles on a few specific graphs. Conversely, on regular graphs, KaMinPar consistently computes partitions of worse quality across nearly all tested graphs (compare Figure 4.8, left and right).

With FM refinement, KaMinPar-FM 10 computes partitions that, on average, have edge cuts 2.46% and 1.29% smaller than those produced by Mt-Metis-HC 10 and Mt-KaHIP-FM 10, respectively, while Mt-KaHyPar-FM 10 achieves an average edge cut that is slightly (by 0.28%) smaller. Looking at Figure 4.10, we can see that in this configuration, edge cuts are generally on-par with competitors, except for graphs in the Biology and Road categories. Here, Mt-KaHyPar-FM 10 and Mt-KaHIP-FM 10, respectively, compute considerable better cuts for a large fraction of instances.

Note that FM refinement can be computationally expensive, especially on irregular graphs. This can be seen in Figure 4.8 (bottom), where the configuration using FM refinement is up to 333 times slower than the one using only label propagation for refinement. We observe slowdowns greater than a factor of 100 for larger values of k on the Proteins1GB7 and Proteins1GB9 graphs of the Compression class, as well as on Mawi2015 of the Web class. However, looking at Figure 4.9, quality improvements also appear most prominent on Compression graphs. Aggregating slowdowns on a per-class basis, the geometric mean slowdowns due to FM refinement are 28.9, 8.3, 8.1, 3.9, 3.0, and 1.7 on irregular Compression, Social, Wiki, Artificial, Web, and Brain graphs. For regular graph classes, the geometric mean slowdowns are 4.2, 2.7, 2.3, 2.0, 1.4, and 1.2 on Optimization, Biology, Finite Element, Semiconductor, Artificial, and Road graphs.

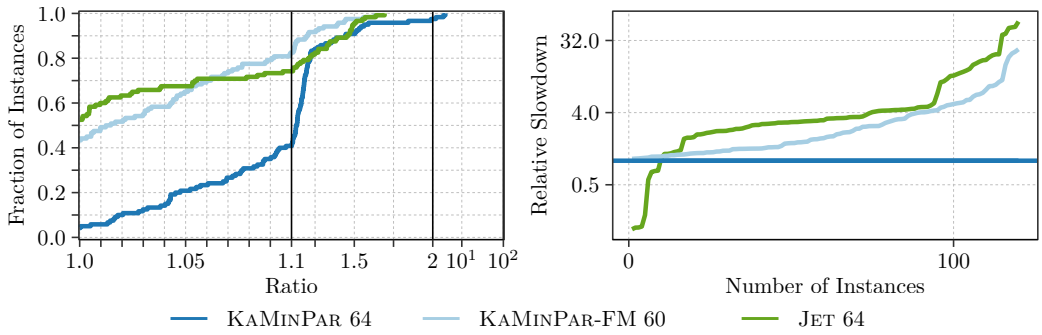

Figure 4.12 Partition quality (**left**) and relative running times (**right**) for KaMinPar, KaMinPar-FM, and Jet for $k \in \{2, 4, 8, 16, 32, 64\}$ on 64 cores.

Scalability Comparison Against Multi-Threaded Multilevel Partitioners. Above, we compared KaMinPar against competing multi-threaded multilevel partitioners using only 10 cores. To contrast its strong scaling behavior with Mt-METIS-HC and Mt-KaHIP-FM, we additionally perform one experiment on Machine A using either 4 or 64 cores. We restrict this experiment to the larger graphs of Benchmark Set B, use the same values of k as above, and set a time limit of 60 min. Note that we evaluate the strong scaling behavior of KaMinPar in isolation and in greater detail in Section 4.4.2.2.

On average across this benchmark set, Mt-METIS-HC 4 and Mt-METIS-HC 64 are slower than KaMinPar 4 and KaMinPar 64 by factors of 3.3 and 5.9, respectively. Similarly, Mt-KaHIP-FM 4 and Mt-KaHIP-FM 64 are slower than KaMinPar 4 and KaMinPar 64 by factors of 5.6 and 7.1, respectively. This indicates that KaMinPar shows better strong scaling behavior than both of the competitors. The average edge cuts obtained by Mt-METIS-HC increase by 3.9% when going from 4 to 64 cores, while they decrease by 2.5% and 2.1% for Mt-KaHIP-FM and KaMinPar, respectively. Note that these aggregates only span instances for which the respective partitioner obtained a balanced partition within the given time limit. For Mt-METIS-HC 4 and Mt-METIS-HC 64, this is the case for 75 and 70 out of 120 instances, respectively. In comparison, both Mt-KaHIP-FM 4 and Mt-KaHIP-FM 64 compute balanced partitions for 115 instances, whereas KaMinPar 4 and KaMinPar 64 always compute balanced partitions in less than 6 min.

Comparison Against Jet. Finally, we compare KaMinPar against the GPU-focused Jet [Gil+24] partitioner on Benchmark Set B, using 64 cores of Machine A[4] and the same values of k as above. This is in contrast to Ref. [Gil+24], where Jet is compared against several CPU-based partitioners on heterogeneous hardware, running Jet on an Nvidia A100 GPU and the CPU-based partitioners on a 32-core Ryzen Threadripper 3970x. In this setting, Jet achieves similar partition quality to Mt-KaHyPar-FM 32, while being at least one order of magnitude faster on 70% of the benchmark instances [Gil+24]. Compared to an older version of KaMinPar 32, it is at least ≈2.8× faster on 50% of the instances (on average, roughly 2.6–2.9× faster depending on graph class) [Gil+24]. However, the refinement phase of Jet is 3.5–9.2× slower on CPU than on GPU [Gil+24] (on average across all instances, 5.6×), suggesting that on comparable hardware, Jet is slower than KaMinPar 32.

As can be seen in Figure 4.12, our experiment confirms this. On Machine A, we observe

[4] We ran this experiment after the discontinuation of Machine B.

Table 4.4 Results of our experiment for large values of k with different parallel partitioners on Benchmark Set B. The last two columns show the geometric mean running time and edge cuts relative to KaMinPar over all instances that do not crash (timeout instances are additionally excluded in edge cut comparison).

Algorithm	#Timeout	#Crash	#Imb.	#Feasible	Rel. Time	Rel. Cut
KaMinPar 10	**0**	**0**	**0**	**80**	**1.00**	1.00
Mt-Metis-HC 10	6	28	46	0	8.90	1.03
Mt-Metis-G 10	**0**	17	61	2	3.83	1.05
Mt-Metis-R 10	**0**	27	50	3	4.74	1.04
Mt-KaHyPar-FM 10	**0**	73	**0**	7	120.24	**0.95**
Mt-KaHIP-FM 10	9	36	11	24	26.12	1.02
Mt-KaHIP-LP 10	19	26	9	26	27.77	1.06
PuLP 10	6	43	1	30	31.64	1.18

that Jet 64 is on average 3.7× slower than KaMinPar 64, while producing 8.1% smaller edge cuts. Compared to KaMinPar-FM 64, it is 1.5× slower while producing 2.4% larger edge cuts on average across this benchmark set. However, as shown in Figure 4.12, Jet 64 finds considerably better partitions for some instances, and is also up to 4× faster than KaMinPar 64 on a few instances.

4.4.1.2 Running Time and Solution Quality for Large k

In the previous section, we have compared the performance of KaMinPar against several other partitioners using relatively small block counts. We now focus on much larger block counts, more specifically $k \in \{2^{11}, 2^{14}, 2^{17}, 2^{20}\}$. We limit our evaluation to Benchmark Set B (20 largest graphs of Benchmark Set A by edge count, bolded in Tables 4.1 and 4.2). We only benchmark parallel partitioners to avoid excessive running times. As before, we set $\varepsilon = 3\%$ and impose a per-instance time limit of 90 min. Note that this time limit is more than 10 times longer than the longest running time of KaMinPar (443 s on Friendster for $k = 2^{20}$) on any instance. The results are summarized in Table 4.4, where we consider a run of an algorithm for an instance as *feasible* if the algorithm terminates in the given time limit and the produced partition satisfies the balance constraint $L_{\max} := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$.

Comparison Against Mt-Metis. Out of the 80 evaluated instances (20 graphs times 4 values of k), Mt-Metis-HC 10, Mt-Metis-G 10, and Mt-Metis-R 10 were only able to produce feasible partitions on 0, 2, and 3 instances, respectively. All configurations produce imbalanced partitions for the majority of instances, and often crash or exceed the time limit. As shown in Figure 4.11 (right), Mt-Metis-HC 10 and Mt-Metis-G 10 generally produce greater balance violations than Mt-Metis-R 10. This makes a fair comparison against all three configurations difficult, since KaMinPar always satisfies the balance constraint. Note that we ignore these balance violations when comparing partition quality in the following.

Even for these large values of k, the recursive bipartitioning mode of Mt-Metis remains, on average, slower than its direct k-way mode with greedy refinement by a factor of 1.64×. However, as shown in Figure 4.13 (left), this relationship eventually reverses: at $k = 2^{17}$, recursive bipartitioning becomes faster by a factor of 1.08×, and at $k = 2^{20}$, by 1.82×. This trend is consistent with our expectations, as the initial partitioning phase in direct

k-way partitioning increasingly dominates running time at large k, introducing a sequential bottleneck. On the other hand, recursive bipartitioning suffers from the $\log(k)$ overhead at smaller k.

Hill-climbing refinement does not seem to scale well to large k. While we observe an improvement in edge cut by 3.5% of Mt-Metis-HC 10 over Mt-Metis-G 10 across instances for which both algorithms did not crash, it more than doubles its running time (increase by a factor of 2.42×). Moreover, hill-climbing runs into the 90 min time limit on 6 instances and generally causes more frequent crashes (28 compared to 17 crashed instances).

Finally, we compare Mt-Metis against KaMinPar. All three configurations show greater slowdowns over KaMinPar for larger values of k compared to the smaller values of k used in Section 4.4.1.1. More precisely, on the graphs of Benchmark Set B, the slowdowns increase from 4.52× (small k) to 4.74× (large k) for Mt-Metis-R 10, from 2.61× to 3.83× for Mt-Metis-G 10, and from 3.00× to 8.90× for Mt-Metis-HC 10. These results are consistent with our expectations regarding the improved scalability of deep multilevel graph partitioning when k is large. On the other hand, the edge cuts produced by KaMinPar are slightly better than those of any Mt-Metis configuration, as shown in Table 4.4. In particular, we note that hill-climbing refinement does not yield better edge cuts, contrary to the observations at smaller k, confer Table 4.3.

Comparison Against Mt-KaHyPar. While the running time of Mt-Metis without hill-climbing refinement remains reasonable, we observe much greater slowdowns for the other multilevel partitioners. For Mt-KaHyPar-FM 10, memory consumption increases drastically for large k. We assume that this is due to the fact that its FM algorithm maintains a gain table of size $\mathcal{O}(nk)$, which quickly becomes infeasible for large k. As a result, it crashes due to insufficient memory on all instances with $k > 2^{11}$ and 13 out of 20 instances with $k = 2^{11}$. On instances on which it does not run out of memory, its running time is more than two orders of magnitude slower than that of KaMinPar, confer Table 4.4. Conversely, it achieves edge cuts that are on average 5% smaller than those of KaMinPar. We attribute this improvement to the use of FM refinement. The improvement is, however, more pronounced at smaller values of k (a 12% reduction as shown in Table 4.3), suggesting that the benefit of FM refinement diminishes as k increases, similar to hill-climbing refinement. We further analyse this effect in Section 4.4.2.6.

Comparison Against Mt-KaHIP. Mt-KaHIP also struggles with large values of k. It crashes on 36 (with FM refinement) resp. 26 (without FM refinement) out of 80 instances, and exceeds the time limit on 9 resp. 19 instances. Since the FM implementation of Mt-KaHIP-FM does not cache gain values, it does not suffer from the same $\mathcal{O}(nk)$ memory bottleneck as Mt-KaHyPar-FM. Yet, we observe a slowdown exceeding 20-fold compared to KaMinPar 10 regardless of the refinement algorithm used, while yielding slightly larger edge cuts on average, see Table 4.4. As before, FM refinement does not yield substantial benefits at large values of k.

Comparison Against PuLP. Recall that PuLP operates by performing label propagation (without a strict size constraint) on a random initial k-way partition, followed by additional label propagation rounds to balance the partition. For large k and a relatively small allowed imbalance of $\varepsilon = 3\%$, this often requires many iterations, resulting in more pronounced slowdowns compared to other approaches that already aim to find a balanced initial partition. In our experiment, PuLP 10 exceeded the time limit on 6 instances and was, on average,

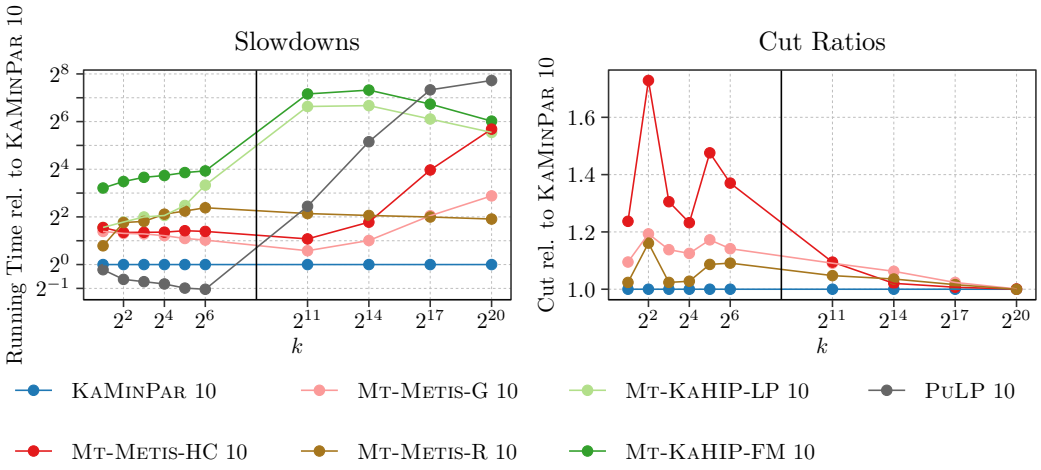

Figure 4.13 **Left**: Running time of competing parallel graph partitioners relative to KaMinPar 10 on the graphs of Benchmark Set B for small ($k \in \{2, 4, 8, 16, 32, 64\}$) and large ($k \in \{2^{11}, 2^{14}, 2^{17}, 2^{20}\}$) values of k. To provide a robust comparison, each curve aggregates only graphs where the respective partitioner produced a result (including timeouts) for all values of k. This excludes Mt-KaHyPar-FM 10, which ran out of memory on all instances with $k = 2^{20}$. **Right**: The corresponding cut ratios (ignoring balance violations) relative to KaMinPar 10, additionally excluding instances where a partitioner exceeded the time limit for some k. This further excludes Mt-KaHIP-LP 10 and Mt-KaHIP-FM 10, which failed or exceeded the time limit on all instances with $k = 2^{20}$.

more than 30 times slower than KaMinPar 10 across the 37 instances where it did not crash (out of a total of 80 instances). On those instances where PuLP 10 successfully produced a partition, it typically satisfied the balance constraint – except for one instance. For smaller values of k, imbalanced outputs occurred more frequently; see Figure 4.11 (left vs. right). This is due to the reduced benchmark set: the graphs for which PuLP 10 produced imbalanced partitions at smaller k were relatively small and thus not part of Benchmark Set B. The edge cuts produced by PuLP 10 are considerably larger than those of KaMinPar 10 – by a factor of 1.18 – and similarly larger than those of the other multilevel partitioners. Notably, this ratio is smaller than at lower k (2.58), indicating that while the multilevel approach remains crucial for achieving high-quality cuts at larger k, its advantage over single-level methods reduces as k increases.

Performance Trends for Increasing k. In Figure 4.13, we plot relative running times (as slowdowns) and partition quality (as cut ratios) of parallel competitors with respect to KaMinPar 10 as the number of blocks k increases. Since competitors frequently crash or exceed the 90-minute time limit for larger k, the cut ratio plot only spans graphs for which the respective competitor successfully produced a partition across all tested values of k. The running time plot further includes graphs for which the respective competitor timed out on some graphs for some k; in this case, the time limit is used as the running time for that instance. This ensures a consistent benchmark set at each value of k and avoids artifacts due to varying graph subsets. However, it also limits the number of graphs included in the

aggregates: for Mt-Metis-{HC, G, R}, only 9,[5] 14,[6] and 9 graphs[7] are included in the running time aggregates, while 3, 14 and 9 graphs are included in the cut ratio aggregates, respectively. For Mt-KaHIP-{LP, FM}, 3 and 2 graphs are included in the running time aggregates, respectively. However, neither configuration of Mt-KaHIP was able to compute a partition for any of the benchmark graphs within the time limit for $k = 2^{20}$, so the partitioner is excluded from the cut ratio plot. Moreover, Mt-KaHyPar-FM 10 ran out of memory on all instances at $k = 2^{20}$, so it is excluded from both plots. For PuLP, 4 graphs are included in the running time aggregates. Since it always exceeded the time limit for $k = 2^{20}$, it is also omitted from the cut ratio plot. We stress that due to the limited number of graphs included in this analysis, drawing definitive conclusions remains challenging.

Looking at Figure 4.13 (left), we can observe that both Mt-Metis-{HC, G} 10 and Mt-KaHIP-{LP, FM} 10 exhibit substantial slowdowns as k increases. This observation is expected, as both approaches are based on direct k-way partitioning, which computes the initial k-way partitioning on a coarse graph of size $\Omega(k)$. At larger values of k, this step becomes a performance bottleneck. The relative running time for Mt-KaHIP-{LP, FM} 10 appears to decrease at larger k, but this is not indicative of improved performance. Instead, the running time is capped by the time limit, preventing further increases while KaMinPar 10's running time continues to grow. Mt-Metis-R 10 is based on recursive bipartitioning, and thus exhibits running time overheads at lower k. However, at $k = 2^{17}$ onwards, it outperforms the other configurations. In particular, its relative running time remains relatively constant for larger k, and even decreases slightly from 4.41 times slower than KaMinPar 10 at $k = 2^{11}$ to 3.76 at $k = 2^{20}$. This is generally expected, as both approaches effectively converge to flat recursive bipartitioning for very large k, which accounts for the bulk of the running time. More precisely, with $k = \alpha n/2$ and $\alpha \in (0, 1]$, the slowdown

$$\frac{T_{\text{rb}}}{T_{\text{deep}}} = \frac{n/p \log(k)}{n/p \log(kC/n)} = 1 + \frac{\log(n/C)}{\log(\alpha C/2)}$$

converges towards $\log(n/2)/\log(C/2)$ as $\alpha \to 1$.

Looking at partition quality (Figure 4.13, right), we observe that the edge cuts produced by all partitioners eventually converge as k increases. As noted above, this suggests that stronger refinement algorithms, such as hill-climbing refinement, show diminishing returns for larger values of k. Moreover, all three variations of multilevel graph partitioning (Deep MGP, direct k-way MGP and recursive bipartitioning MGP) achieve essentially the same partition quality for the largest value of k in our experiment. This raises the question of whether multilevel algorithms are still necessary to obtain high-quality partitions when k is very large. To address this question, Manuel Haag compared KaMinPar with two single-level techniques in his bachelor thesis [Haa21]: single-level label propagation and a *path refiner* that searches for a path in the quotient graph of a given partition and then moves vertices along this path, from block to block, so that the overall move sequence reduces the edge cut. He found that KaMinPar computes lower edge cuts than single-level label propagation based partitioning

[5] Graphs included in the running time aggregates for Mt-Metis-HC: Arabic2015, **HV15R**, Nlpkkt200, Nlpkkt240, Rgg2e26-2D, Rgg2e26-3D, **Rhg10e8**, **Sinaweibo**, and Webbase2010. Bolded: also included in the cut ratio aggregates.

[6] Graphs included in the running time aggregates for Mt-Metis-G: Arabic2015, Bn87128519, HV15R, Kmer2Aa, KmerV1r, Nlpkkt200, Nlpkkt240, Queen4147, Rgg2e26-2D, Rgg2e26-3D, Rhg10e8, Sinaweibo, Stokes, and Webbase2010. The same graphs are included in the cut ratio aggregates.

[7] Graphs included in the running time aggregates for Mt-Metis-R: HV15R, KmerA2a, KmerV1r, Nlpkkt200, Nlpkkt240, Queen4147, Rgg2e26-2D, Rhg10e8, and Stokes. The same graphs are included in the cut ratio aggregates.

Table 4.5 Comparison of Deep MGP, direct k-way partitioning and recursive bipartitioning with small (left, on Benchmark Set A) and large (right, on Benchmark Set B) values of k. Running times and edge cuts are relative to Deep MGP. #Imb. counts the number of imbalanced partitions produced by the respective partitioning scheme, while #Fail counts the number of timeouts and crashes due to insufficient memory.

	$k \in \{2, 4, 8, 16, 32, 64\}$				$k \in \{2^{11}, 2^{14}, 2^{17}, 2^{20}\}$			
Algorithm	Rel. T	Rel. Cut	#Imb.	#Fail	Rel. T	Rel. Cut	#Imb.	#Fail
Deep MGP	1.00	1.00	0	0	1.00	1.00	0	0
Direct k-way	0.88	1.02	0	0	10.41	1.00	0	28
RB	2.49	1.03	0	0	3.20	1.01	35	0

up to $k \leq n/16$, and better cuts than single-level partitioning with additional path refinement up to $k \leq n/32$, indicating that, as long as k is not extremely large, multilevel algorithms still provide a quality advantage.

Conclusion. Our experiments show that KaMinPar can reliably compute feasible and high-quality partitions in a reasonable amount of time even if the number of blocks k is very large. Multilevel partitioners struggle with larger running times, especially when using expensive refinement algorithms such as hill-climbing [LK16] or the FM algorithm [FM82]. Moreover, they often produce infeasible partitions, or fail to balance the partition within the 90 min time limit.

4.4.2 Experimental Evaluation of Algorithmic Components

We now evaluate the algorithmic components and tuning parameters of KaMinPar. As before, we evaluate KaMinPar on the 72 graphs of Benchmark Set A, which are listed in Tables 4.1 and 4.2, and we use compute nodes of Machine B unless stated otherwise (confer Table 2.2). For our scalability experiments in Section 4.4.2.2, we use Machine A and only the 20 largest graphs of Benchmark Set A (by number of edges, denoted Benchmark Set B), which are bolded in the benchmark tables.

4.4.2.1 Partitioning Schemes

In Section 4.4.1, we compared KaMinPar with partitioners based on direct k-way partitioning (Mt-KaHyPar, Mt-KaHIP, and Mt-Metis-{G, HC}) and one partitioner based on recursive bipartitioning (Mt-Metis-R). This provides an indirect comparison between deep multilevel partitioning and classical multilevel schemes, but the results are confounded by differences in implementation details and in the general building blocks used for coarsening, initial partitioning, and refinement. For a better comparison, we implement and benchmark direct k-way partitioning and recursive bipartitioning within the KaMinPar framework.[8]

The implementation for direct k-way partitioning roughly follows the implementation of Mt-KaHIP [ASS20] and uses the same parallel coarsening algorithm as the Deep MGP implementation in KaMinPar. In this case however, the graph is coarsened until at most $C \cdot k$ vertices remain (instead of $2 \cdot C$ vertices as in Deep MGP). At that point, each thread independently computes an initial k-way partition by using sequential recursive bipartitioning.

[8] This experiment was performed on the successor of Machine B (AMD EPYC 9454 compute nodes).

The best (balanced) partition is subsequently selected and refined during uncoarsening, using the same parallel refinement algorithms as before.

Recursive bipartitioning on the other hand is implemented as follows. First, the framework computes a $k' \coloneqq \min\{k, \tau \cdot \text{ceil}_2(p)\}$-way partition by recursively bipartitioning the graph in parallel. In our experiments, $p = 10$ and $\tau = 25$, hence $k' = \min\{k, 400\}$. During this phase, each bipartitioning step uses the same parallel coarsening and uncoarsening algorithms as the other partitioning schemes. Parallelism on the coarsest graph is used for independent attempts at computing an initial bipartition. After a bipartition is computed, the block-induced subgraphs are extracted and two parallel tasks for further bipartitioning are spawned. Note that the code is parallelized using Intel TBB, which implements a work-stealing approach to balance load between threads. Once this process has computed a k'-way partition, and $k' < k$, the blocks are further bipartitioned using the same sequential algorithms as used during the initial bipartitioning step of Deep MGP, see Section 4.3.3. The reason for this two-phased approach is as follows. Our parallel coarsening and refinement algorithms are not optimized for running many times concurrently or on very small graphs. On the other hand, we only want to switch to our sequential code paths once there are sufficiently many blocks to avoid load imbalances. Note that recursive bipartitioning does not use k-way refinement, and in particular, does not use our k-way rebalancing algorithm.

The results are summarized in Table 4.5. For small values of k, all three schemes achieve similar partition quality, with both direct k-way partitioning and recursive bipartitioning producing edge cuts that are within roughly 2% of those of Deep MGP. We attribute the slightly better quality of Deep MGP to its diversified initial partitioning scheme across multiple hierarchy levels, as described in Section 4.2.1. Direct k-way partitioning is slightly faster than Deep MGP in this setting. This is due to initial partitioning: since we do not limit the vertex reduction rate between subsequent hierarchy levels during coarsening, the coarsest graph can have considerably fewer than $C \cdot k$ vertices. On the other hand, the initial bipartitioning steps during Deep MGP can be applied to coarse graphs with considerably more than $2C$ vertices per block for the same reason, incurring running time overheads. Recursive bipartitioning, on the other hand, is a factor of 2.49 slower than Deep MGP, reflecting the additional $\log(k)$ factor in its runtime.

For larger values of k, Deep MGP is by far the fastest partitioning scheme. Direct k-way partitioning becomes more than one order of magnitude slower since its sequential initial partitioning phase turns into a bottleneck. We have observed the same behaviour for Mt-KaHIP and Mt-Metis-{G, HC} in Section 4.4.1.2. Moreover, it often exceeds the available memory due to invoking initial partitioning on a very large graph. Recursive bipartitioning roughly follows the same trend as we observed for Mt-Metis-R in Figure 4.13, where its relative slowdown over Deep MGP eventually converges as k grows. Note that recursive bipartitioning computes some imbalanced partitions due to the lack of k-way refinement.

4.4.2.2 Scalability

Figure 4.14 illustrates the scalability of KaMinPar on Benchmark Set B, using $p \in \{1, 4, 16, 64\}$ cores of Machine A. To give each major part of KaMinPar a fair share of the total running time, we use $k \in \{2^{11}, 2^{14}, 2^{17}, 2^{20}\}$. In particular, this means that we also allocate a fair share of the total running time to initial bipartitioning. For comparison, we also consider the scalability for smaller, more *classical* values of k ($k \in \{2, 4, 8, 16, 32, 64\}$) in Figure 4.15. Note that in these cases, initial bipartitioning is inherently non-scalable on 64 cores, since we use the additional parallelism for more bipartitioning attempts on diversified coarse graph hierarchies. Each instance's speedup is plotted as a point, while the

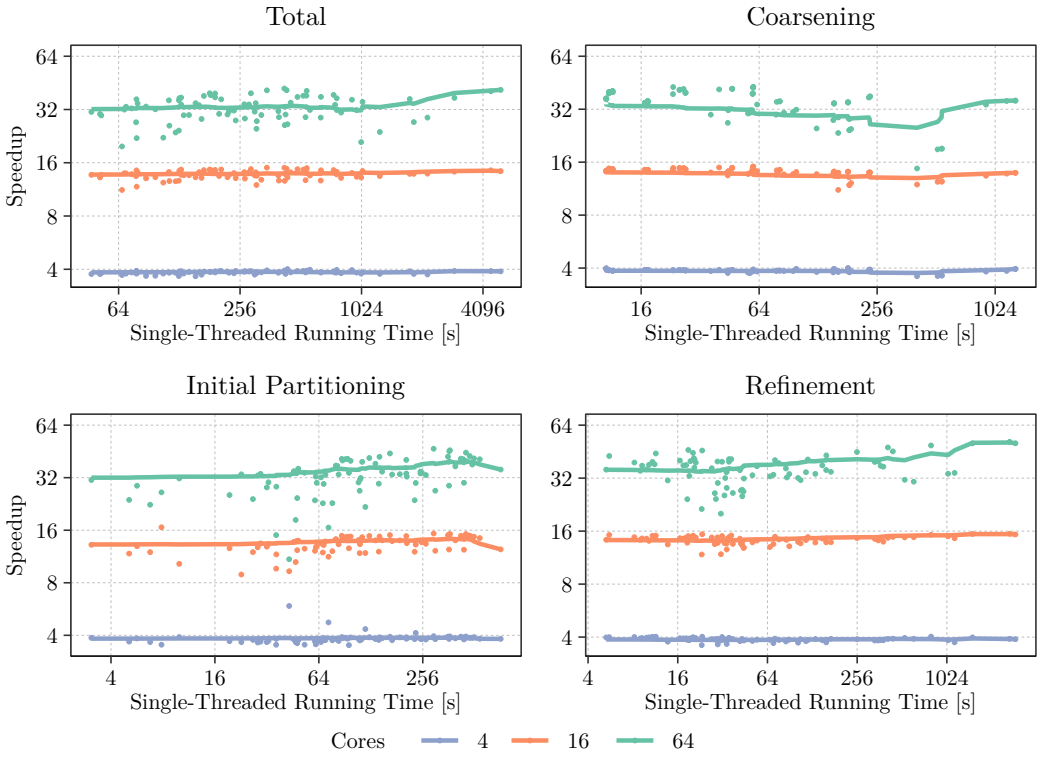

Figure 4.14 Self-relative speedups for the different components of KaMinPar on Benchmark Set B using $k \in \{2^{11}, 2^{14}, 2^{17}, 2^{20}\}$.

cumulative geometric mean speedup over instances with a single-threaded running time of at least t seconds is shown as a line. Note that in the remainder of this section, the term *initial bipartitioning* refers to bipartitioning on graphs with more than $2pC$ vertices only.

For large values of k, the overall geometric mean speedup of KaMinPar is 3.8 for $p = 4$, 13.7 for $p = 16$, and 32.1 for $p = 64$. The geometric mean speedups for coarsening, initial partitioning and refinement are 34.1, 32.0 and 35.7 for $p = 64$. While the speedups remain close to ideal up to $p = 16$ cores, they decline at $p = 64$ cores. We attribute this to memory bandwidth limitations, as KaMinPar avoids computationally expensive arithmetic operations, thereby making performance predominantly memory-bound. Thus, perfect speedups are not possible.

For smaller values of k, the speedups tend to be lower, with overall geometric mean speedups of 3.8 for $p = 4$, 12.5 for $p = 16$, and 24.0 for $p = 64$. For $p = 64$, the geometric mean speedups for coarsening and refinement are 30.1 and 33.4, respectively. We attribute this trend to several factors. First, we spend an increased fraction of time on initial bipartitioning in replicated graphs, which inherently does not scale. During the coarsening phase, the cluster weight limit on finer levels is higher for smaller k, resulting in larger and thus fewer clusters, which increases contention. Similarly, fewer blocks during refinement result in greater contention when updating block weights.

Lastly, we examine how partition quality evolves with increasing thread counts. Figure 4.16 presents the edge cuts corresponding to Figure 4.15 (i.e., for small k). Overall, we observe a slight improvement in partition quality as the number of cores increases. With 64 cores, the geometric mean cut is 2.42% smaller than when only using a single core.

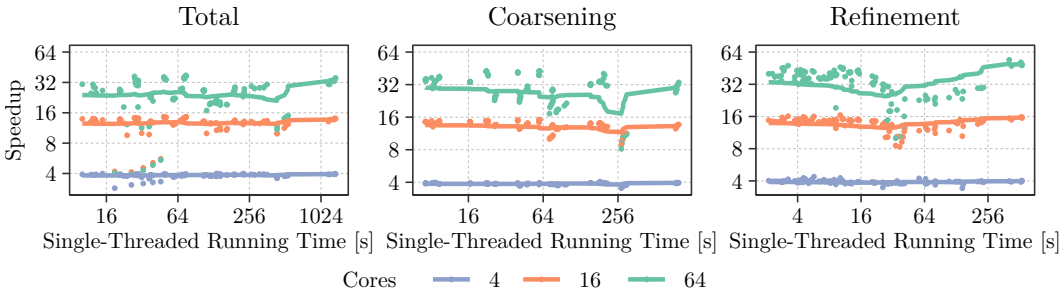

Figure 4.15 Self-relative speedups for the different components of KaMinPar on Benchmark Set B using $k \in \{2, 4, 8, 16, 32, 64\}$.

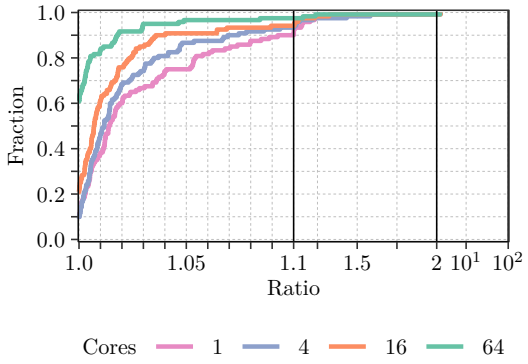

Figure 4.16 Partition quality of KaMinPar on Benchmark Set B and $k \in \{2, 4, 8, 16, 32, 64\}$ using $p \in \{1, 4, 16, 64\}$ threads.

This improvement is expected, as the additional parallelism is used to generate more initial bipartitions on diversified coarse graphs, thus increasing the chance to find a better bipartition. This matches the observations made for Mt-KaHIP [ASS20], but is different than Mt-KaHyPar [Got+24a], where the best results are obtained on smaller core counts.

4.4.2.3 Impact of Graph-Reordering

Recall from Section 4.3.1 that KaMinPar sorts the vertices of the input graph into exponentially spaced degree buckets and reorders the graph accordingly. This preprocessing step is performed before the coarsening phase. As shown in Figure 4.17, reordering improves overall partition quality for both regular and irregular graphs, though its impact is more pronounced on irregular graphs. On average, edge cuts are reduced by 2.8%, with a smaller improvement of 0.6% for regular graphs and a more significant improvement of 4.7% for irregular graphs. However, these improvements come at the cost of increased running times. Overall, the running time increases by 8.7% on average, with an increase of 3.7% for regular graphs and 13.1% for irregular graphs. This performance overhead stems from two main factors. First, the reordering process itself requires additional computation effort to rearrange the graph in memory, accounting for 5.2% of the total running time (4.8% for regular graphs and 5.6% for irregular graphs). Second, reordering the vertices can disrupt some locality that might exist in the natural vertex order, negatively impacting cache efficiency. For irregular graphs, coarsening after reordering is 2.8% slower, and refinement is 5.2% slower. Interestingly, for

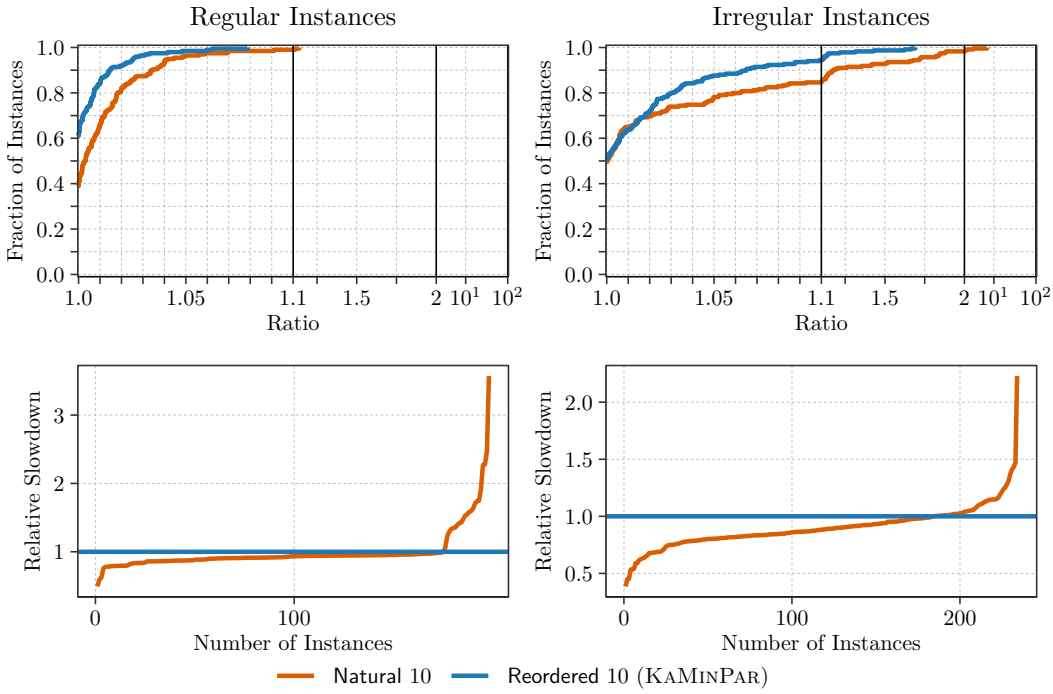

Figure 4.17 Impact of vertex reordering by degree buckets on running time and edge cuts for $k \in \{2, 4, 8, 16, 32, 64\}$. Natural refers to KaMinPar without vertex reordering (i.e., label propagation iterates over vertices in their natural order), whereas Reordered applies the reordering. Results are shown separately for regular (left, graphs of Table 4.1) and irregular (right, Table 4.2) graphs.

regular graphs, coarsening is 5.4% slower, but uncoarsening is 8.9% faster.

Despite the running time overheads, the 4.7% edge cut reduction on irregular graphs justifies the preprocessing step. Note that in the context of multilevel graph partitioning, even a small percentage improvement in edge cuts across a large and diverse benchmark set is often considered substantial.

4.4.2.4 Impact of Two-Hop Clustering

Recall that the coarsening phase of KaMinPar uses size-constrained label propagation with additional two-hop clustering, see Section 4.3.1. As shown there, two-hop clustering is required to obtain provable bounds on the vertex reduction rate between coarsening levels. This raises the natural question whether two-hop coarsening is cost free, or whether it comes at the cost of reduced partition quality or increased running time.

We therefore benchmark KaMinPar with and without two-hop clustering. As can be seen in Figure 4.18, two-hop clustering has no measurable effect on most benchmark instances. This is expected, since it is only triggered if the number of coarse vertices on the next hierarchy level would not be reduced by at least a factor of two otherwise. When it is triggered, the impact on running time can, however, be substantial, as shown in Figure 4.18 (right): on 44 out of 432 instances, two-hop clustering yields speedups exceeding 1.25×. The most extreme case is on the Mawi2015 graph with $k = 64$, where it achieves a 164× speedup (and a 55× speedup with $k = 2$). For all other affected graphs, speedups remain below 5×. All graphs for which two-hop clustering is triggered are classified as irregular by Table 4.2.

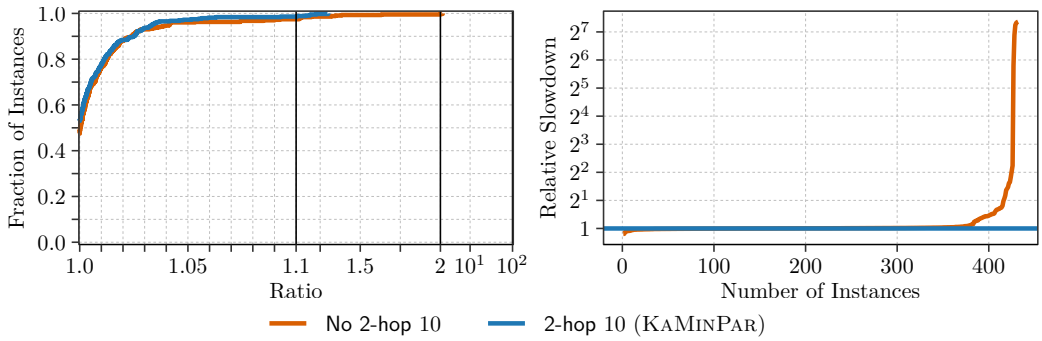

Figure 4.18 Impact of coarsening with and without 2-hop clustering on solution quality (left) and running time (right).

Finally, the performance profile Figure 4.18 (left) reveals that two-hop clustering does not degrade partition quality.

4.4.2.5 Running Time Analysis

So far, we have only reported the total running time of KaMinPar. In Figure 4.20, we decompose the total running time into the contributions of clustering, coarse graph construction, initial partitioning, balancing and refinement. The graphs in the plot are sorted by edge count along the y-axis, with the largest graphs at the top. In the following discussion, we state the average fraction of running time spent in each phase in parentheses. Note that we aggregate the per-instance fractions with arithmetic means, so that the phase shares sum up to 100%.

First, the breakdown reveals that, for large graphs, running time is dominated by the coarsening phase (62.6% of the total running time on average), specifically clustering the input graph (45.7%) and constructing the first level of the graph hierarchy (11.7%). Constructing the remaining coarsening levels is comparably cheap (5.1%). The reason for this is that we do not constrain the vertex reduction rate during coarsening (beyond enforcing the maximum cluster weight). As a consequence, the graphs are often already much smaller than the input graph after the first clustering and contraction step. For smaller graphs, the cost of initial partitioning (14.2%) becomes increasingly prominent. This is due to the same reason: when graphs shrink too fast, initial bipartitioning may be invoked on graphs substantially larger than $2C$ vertices per block. In the most extreme cases, the first coarsening level reduces the graph very aggressively, causing initial bipartitioning to be invoked on the input graph directly. This happens, e.g., on the HV15R graph, where initial partitioning dominates overall running time.

Balancing contributes only a negligible fraction (1.3%) of the total running time for all graphs. This is consistent with our discussion in Section 4.2.4, where we argue that balance violations that occur during deep multilevel graph partitioning are typically small. Lastly, we observe that refinement (13.3%) is faster than clustering, although both steps use the same size-constrained label propagation algorithm. This is because there are (much) fewer labels during refinement (one label per block) than during clustering (one label per vertex). For this reason, the uncoarsening phase in KaMinPar is generally much faster than the coarsening phase. This would change when deploying stronger refinement algorithms, such as FM refinement. The remaining running time (denoted Rest, 8.8%) is spent on reordering the graph during preprocessing (see Sections 4.3.1 and 4.4.2.3), projecting the partition

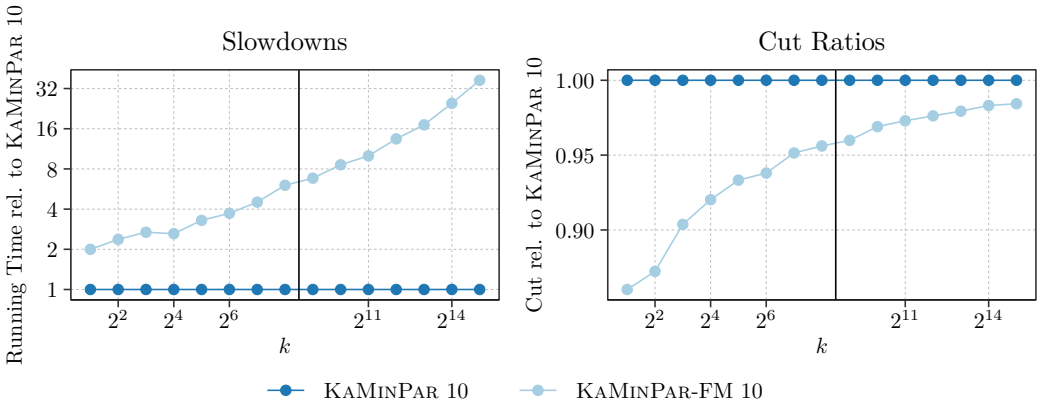

Figure 4.19 Impact of FM refinement for small and large values of k. Since the running time of FM refinement grows considerably with k, we only benchmark up to $k = 2^{17}$.

from coarser to finer graphs, memory allocations during partitioning, and other pre- and postprocessing steps (e.g., the removal of isolated vertices during preprocessing, which are then greedily assigned to blocks during postprocessing).

4.4.2.6 Impact of FM Refinement for Large k

The results observed in Section 4.4.1.2 suggest that stronger refinement algorithms, such as Mt-Metis' hill-climbing refinement [LK16] or (parallel) FM refinement [FM82], yield diminishing quality improvements as k grows. However, so far we have only compared stronger refinement algorithms implemented in other partitioners for larger values of k. For a cleaner apples-to-apples evaluation, we now benchmark KaMinPar with and without additional FM refinement for smaller and larger values of k. Figure 4.19 confirms our previous observation: FM refinement is mostly beneficial for smaller values of k. As k increases, the average improvement in edge cut decreases. Since the running time of FM refinement depends on the boundary size of the partition, its running time increases substantially at the same time. For instance, at $k = 2$, FM refinement improves edge cuts by 14.0% on average, while increasing running times by only 2.0×. However, at $k = 2^{15}$, the average edge cut is only improved by 1.6%, at the cost of a 36.9× higher average running time. Thus, FM refinement should not be used for large values of k.

4.5 Conclusion

We presented a new graph partitioning scheme that successfully combines the merits of classical direct k-way partitioning and recursive bipartitioning. Similar to direct k-way partitioning, Deep MGP coarsens and uncoarsens the graph only once, and allows the use of k-way local improvement algorithms. Yet, it does not suffer scalability problems if k is large and has a better asymptotic running time than recursive bipartitioning. Our experimental evaluation shows that our shared-memory parallel implementation KaMinPar of Deep MGP runs efficiently on up to 64 cores, while achieving comparable results to established multilevel graph partitioners and being at least 2× faster if k is small. Surprisingly, KaMinPar is on average even faster than the single-level partitioner PuLP, which also uses label propagation as a core building block. Furthermore, our evaluation shows that KaMinPar is much faster

than other graph partitioners based on direct k-way partitioning when k is large, while consistently producing balanced solutions.

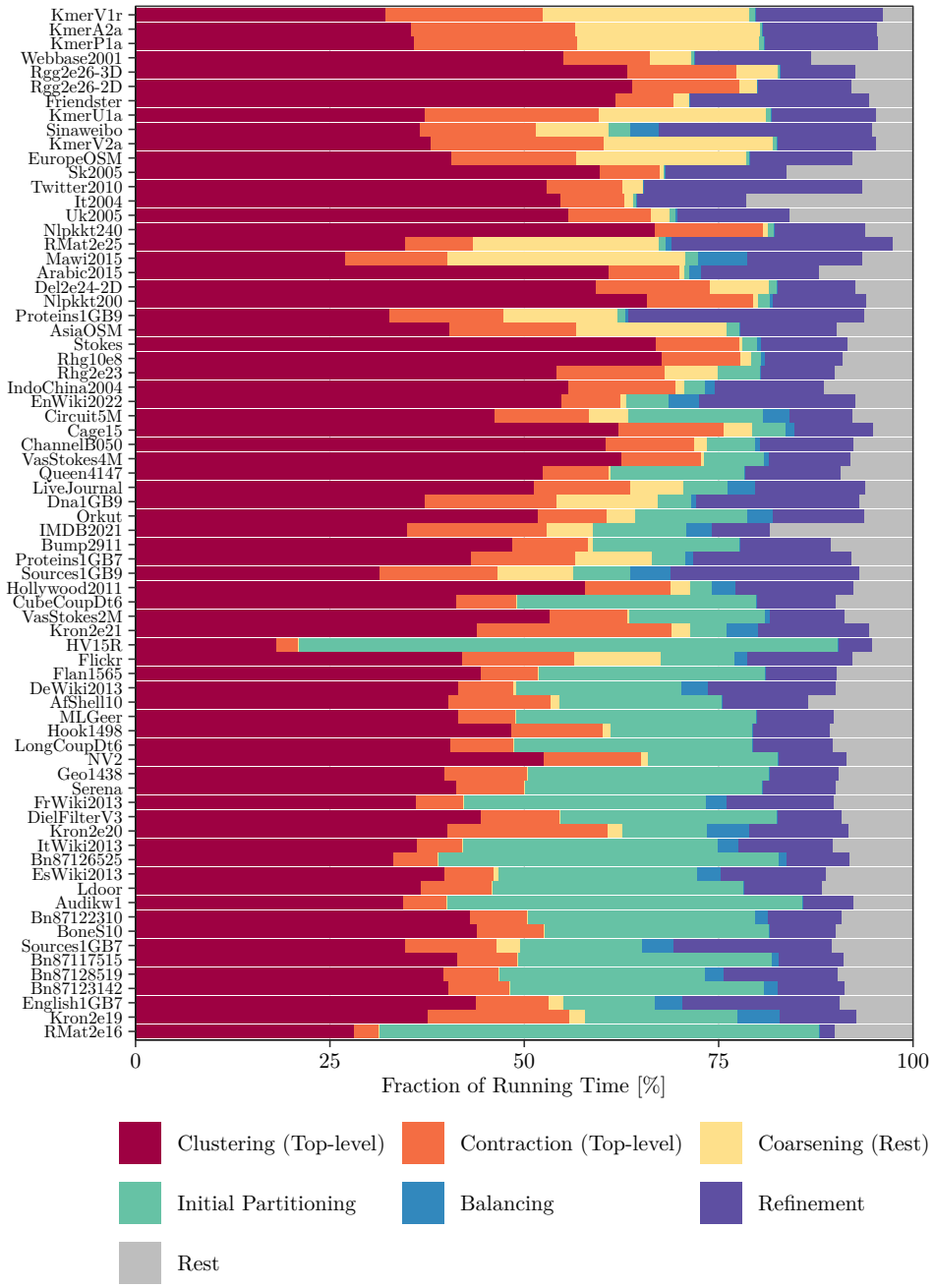

Figure 4.20 Breakdown of KaMinPar's total running time into coarsening, initial partitioning, balancing and refinement phases for Benchmark Set A with $k \in \{2, 4, 8, 16, 32, 64\}$, using 10 cores of Machine B. The coarsening phase is further subdivided into top-level clustering (Clustering (Top-level)), top-level contraction (Contraction (Top-level)), and clustering and contraction of coarse graphs (Coarsening (Rest)). The Rest category mostly accounts for pre- and post-processing tasks such as graph reordering, removal and reintegration of isolated vertices, and memory allocations and deallocations. Graphs are ordered by descending number of vertices.

5 Tera-Scale Multilevel Graph Partitioning

Recent work has seen tremendous progress in speeding up partitioning algorithms through parallelism. In the previous chapter, we have introduced KaMinPar, a shared-memory parallel graph partitioner that is $\geq 2\times$ faster than previous parallel multilevel partitioners. As we have shown in our experimental evaluation, KaMinPar is able to partition a graph with 1.8 G edges in roughly two minutes, using only 10 cores. Its current obstacle in scaling to much larger graphs is its high memory usage due to auxiliary data structures and storing the graph itself in memory. In this chapter, we tackle this obstacle and present TeraPart, a memory-efficient multilevel graph partitioning method that is designed to scale to much larger graphs.

Contributions. We present and study several optimizations to significantly reduce KaMinPar's memory footprint. We devise parallel label propagation clustering and graph contraction algorithms that use $\mathcal{O}(n)$ auxiliary space instead of $\mathcal{O}(np)$, where n is the number of vertices in the graph and p is the number of processors. Moreover, we employ an existing compressed graph representation that enables iterating over a neighborhood by on-the-fly decoding at speeds close to the uncompressed graph. Combining these optimizations yields up to a 16-fold reduction in peak memory, while retaining the same solution quality and similar speed. This configuration can partition a graph with *one trillion* edges in under 8 minutes *on a single machine* using 96 cores and around 900 GiB of RAM. This is the first work to employ the multilevel framework at this scale, which is vital to achieving low edge cuts. Finally, we present a version of shared-memory parallel FM local search that uses $\mathcal{O}(m)$ space instead of $\mathcal{O}(nk)$, reducing peak memory by a factor of 6.3 on medium-sized graphs without affecting running time.

References and Contributors. This chapter is based on a conference publication [Sal+25] published together with Daniel Salwasser, Lars Gottesbüren, and Peter Sanders. The ideas described here were developed together by all authors. The implementation was mostly done by Daniel Salwasser as part of his bachelor thesis [Sal24] and for our publication. In our weekly meetings with Daniel Salwasser, we discussed new ideas and designed the experimental evaluation study for the algorithms. The space-efficient parallel FM local search algorithm was implemented by the author of this dissertation.

Structure. The remainder of this chapter is organized as follows. In Section 5.1, we review related work that is closely related to our contributions, before analyzing the current memory consumption of KaMinPar in Section 5.2. The following sections introduce algorithms to reduce the memory consumption: Section 5.3 describes the in-memory graph compression techniques implemented in TeraPart and provides details on the parallel compression routine. Section 5.4 introduces changes to the label propagation clustering and contraction algorithms implemented in KaMinPar to reduce their memory footprint, while Section 5.5

describes a novel approach at gain tables to drastically reduce the memory footprint of parallel FM refinement. Finally, we present an experimental analysis of TeraPart in Sections 5.6 and 5.6.3 before concluding this chapter in Section 5.7.

5.1 Related Work

We introduce the general related work for graph partitioning in Chapter 2. Here, we only focus on related work that is closely related to the contributions of this chapter. We highlight additional context on the multilevel scheme and briefly recall the relevant components of KaMinPar.

In-Memory Graph Compression. Multilevel algorithms are typically implemented as in-memory algorithms, i.e., the complete graph and all required auxiliary data structures are stored in main memory. Storing the full graph in memory limits the maximum graph size that can be processed on a single shared-memory machine. This limitation can be mitigated by storing a compressed representation of the input graph instead of a plain edge list or adjacency array. To be practical in high-performance settings, such a scheme must support fast parallel encoding and, in particular, low-overhead parallel decoding. Moreover, many algorithms require random access on the vertices of the graph, which must also be supported by the encoding scheme.

Shun et al. [SDB15; Shu20] integrate such a compressed representation into the Ligra framework. They compress each adjacency list by first sorting neighbors in ascending order and storing the gaps between consecutive neighbors (*gap encoding*), and then encoding these gaps using *variable-length* k-bit codes, including a run-length encoded byte-code variant inspired by Kourtis et al. [KGK10]. On a range of large real-world and synthetic graphs, this reduces the in-memory size to about half of the uncompressed representation. Despite the additional decoding work, the authors report an average speedup of about 14% on 40 hardware threads (includes hyperthreads) across several graph processing tasks,[1] arguing that most parallel graph algorithms are memory-bound, so that improved memory-bandwidth utilization can more than compensate for the additional work.

Semi-External Graph Partitioning. An alternative approach to reducing the memory consumption of graph processing workloads is to use (semi-)external algorithms instead of purely in-memory algorithms. In the context of graph partitioning, this has been explored by Akhremtsev et al. [ASS15], who design semi-external and external variants of the multilevel scheme based on size-constrained label propagation for both clustering and refinement. Once the contracted graph becomes small enough to fit into main memory, the in-memory partitioner KaHIP [SS11a] is invoked to compute the initial partition. Their experiments report running times for the semi-external implementation that are competitive with in-memory KaHIP, although this is achieved largely by offsetting I/O overheads through orthogonal algorithmic tuning (e.g., different size-constraints during coarsening and fewer label propagation rounds). Moreover, their multilevel algorithm relies exclusively on label propagation based refinement in the (semi-)external phase. In contrast, more powerful refinement algorithms such as parallelized FM [ASS20; Got+21a] explore local regions of the graph adaptively and therefore require frequent random access to adjacency lists of

[1] Breadth-first search, betweenness centrality computation from a source vertex, graph radii estimation, connected components, and Bellman-Ford shortest paths [SDB15].

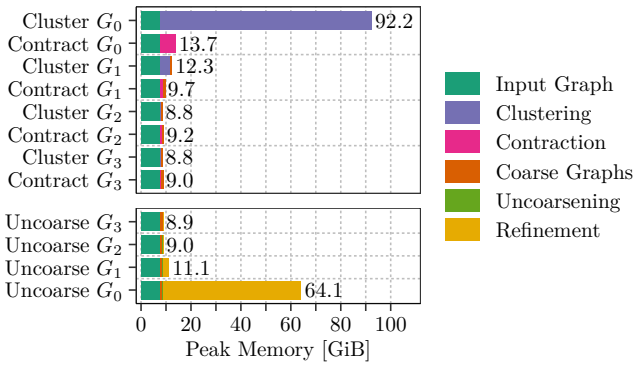

Figure 5.1 Memory consumption during the different phases of the KaMinPar algorithm. Only the top level and three coarse levels are shown, as the memory consumption is barely reduced for the following levels. The measurements were carried out for webbase2001 with $p = 96$ cores and $k = 64$ blocks.

neighboring vertices, which appears difficult to support efficiently in the semi-external model without incurring prohibitive random I/O.

KaMinPar. We implement our optimizations in the state-of-the-art shared-memory parallel multilevel graph partitioner KaMinPar introduced in Chapter 4. For coarsening, it uses label propagation clustering, with additional two-hop matching [LaS+15] to ensure coarsening progress on irregular graphs. Starting from each vertex in its own cluster, label propagation visits vertices in random order in parallel; a vertex joins a cluster containing the plurality of its neighbors. For initial bipartitioning, it uses a portfolio of randomized sequential greedy graph growing heuristics and 2-way FM [FM82] refinement. In the refinement stage, KaMinPar uses size-constrained label propagation [MSS17], starting with the given partition, and optionally shared-memory parallel localized k-way FM refinement [ASS20; Got+21a].

5.2 Memory Analysis

In order to understand the areas for potential improvement, we analyze which component of KaMinPar uses how much memory at which stage of the partitioning process. Figure 5.1 breaks down the memory usage per stage and level using the webbase2001 [RA15] graph with $k = 64$ blocks and $p = 96$ cores as an example. For this analysis, we enabled the optional FM refinement described in Section 4.3.4.2.

The top three memory peaks occur when working on the top-level graph G_0: (i) clustering in the coarsening stage, (ii) FM refinement, and (iii) contracting the clustering to construct G_1. In the coarsening stage, KaMinPar uses 92.2 GiB, 84.5 GiB of which for auxiliary data structures in the clustering algorithm and 7.7 GiB for the input graph. Two-phase label propagation (which we will introduce in Section 5.4.1) reduces the auxiliary memory during clustering to 2.8 GiB. By compressing the input graph (see Section 5.3.1) we only need 2.9 GiB to store it. Note that further memory savings from compressing coarser graphs (level ≥ 1) are negligible, so that we only compress the input graph. During FM refinement, the auxiliary data structure uses 55.1 GiB of RAM, which we reduce to 5.6 GiB by employing our sparse gain table optimization (see Section 5.5). Note that label propagation refinement uses a negligible amount of memory, as it is proportional to k, rather than n. Finally, one-

pass contraction (see Section 5.4.2) reduces the auxiliary memory from 6.0 GiB to 1.4 GiB. Combining the optimizations reduces the peak memory usage on webbase2001 to 5.7 GiB when using label propagation refinement and to 9.5 GiB when using optional FM refinement.

Given this analysis, a natural question is whether KaMinPar is particularly wasteful and whether other partitioning algorithms, such as Mt-METIS [LK13; LK16] are better. We found that Mt-METIS uses $2\times-4\times$ more memory than KaMinPar, confer Figure 5.4 (middle).

5.3 Graph Representation

The graphs are stored in the compressed sparse row (CSR) format. In the CSR format, the edges are stored in one contiguous array $\mathcal{E}$ of size $2m$, i.e., each undirected edge is represented by two directed edges in opposite directions. Additionally, an array $\mathcal{P}$ of size $n+1$ stores the beginning of each neighborhood, i.e., $\mathcal{E}[\mathcal{P}[u] : \mathcal{P}[u+1]]$ stores the neighbors of vertex u.

To reduce the memory footprint of the input graph, we use a compressed representation instead of the CSR format. We further want to reduce the running time and memory space required to compress the input graph, which is why we compress the input graph in parallel in a single I/O-pass.

5.3.1 Compression Scheme

We employ compression techniques that enable fast decoding of each vertex's neighborhood. This is inspired by the success of Ligra+[SDB15] for graph processing workloads (which do not include graph partitioning). Ligra+ uses gap encoding [BV04] combined with variable-length integer encoding (VarInt). With gap encoding, one sorts each neighborhood by increasing ID and stores only the difference to the previous neighbor ID. Since the gaps are often small, they can often be represented using much less than 8 bytes. VarInt is a variable-length byte codec, where the first 7 bits in a byte are used to represent part of a number and the final one bit (the continuation bit) indicates whether the first 7 bits of the following byte belong to the current or a new number. Additionally, we combine this with interval encoding [BV04], which lets us achieve less than one byte per compressed edge on some graphs, in contrast to Ligra+. With interval encoding, one stores consecutive intervals $\{x, x+1, x+2, \ldots, x+\ell-1\}$ with $\ell \geq 3$ as a tuple (x, ℓ) rather than ℓ bytes representing gaps of 1 each with gap encoding. This is particularly impactful in graphs with high locality in the neighbor IDs. For edge weights, we only employ gap encoding combined with the VarInt codec because interval encoding is unlikely to yield better compression. Since edge weights are not sorted, we store an additional sign bit. To enable parallel iteration over the neighborhood of a high-degree vertex (degree larger than 10 000), we split its neighbors into chunks of fixed length (size 1 000) that are encoded and decoded independently [SDB15].

Similar to the CSR format, the compressed neighborhoods are concatenated into one array storing the edges, and for each vertex we store a pointer to the beginning of its neighborhood. We store the edge weights of a neighborhood interleaved with its gaps and intervals. Moreover, due to compression, vertex degrees cannot be deduced from the range begin pointers. Hence, we store the first edge ID of a neighborhood as a VarInt at the start of the neighborhood to deduce the degrees. By storing the first edge ID instead of the degree, we can obtain the ID of each edge in a neighborhood during iteration, which is required by some parts of the KaMinPar code.

Algorithm 6 Original Label Propagation Round

```
1 for u ∈ V do parallel
2     R ← rating map
3     for v ∈ N(u) do
4         R[C[v]] ← R[C[v]] + ω(uv)
5     C[u] ← arg max_{c ∈ R.keys()} R[c]
```

5.3.2 Parallel Compression and Single-Pass I/O

The size of the uncompressed graph may exceed the available memory, so that loading the uncompressed graph and then compressing it from memory is not an option. Rather we want to perform the compression during I/O.

One issue with CSR storage is that the size of the edge array must be known beforehand to allocate to the correct size. However, the size of the compressed edges is only known after loading and compressing them. The naive solution is to load the graph twice, computing the compressed size in the first pass, and storing the compressed graph in the second pass. However, often the graph I/O takes a similar time as partitioning itself, so ideally we want to load the graph only once. Growing arrays such as `std::vector` are not suitable either, since they require twice the memory to copy the data when growing.

To avoid having to know the correct amount of memory in advance, we overcommit memory [WMS09], that is, we compute an upper bound on the memory consumption of the compressed edge array and request that amount. Moreover, we only touch the memory that we actually use. This technique works because the operating system only assigns virtual pages to physical pages when the corresponding page is touched. Thus, only the memory of the compressed edge array plus at most one page is physically backed by memory.

To further speed up the I/O, we compress neighborhoods in parallel. However, to store a compressed neighborhood in the edge array, we have to know the size of the previous compressed neighborhoods, so that it can be written to the correct position. This is challenging, as this is essentially a prefix sum problem, which requires scanning the input twice when performed in parallel, and we only want to compress the neighborhoods once. We overcome this by using the two-pass approach locally. The threads work on packets of consecutive vertices that contain a similar number of edges and first compress them into a thread-local buffer. A thread that finished a packet waits until all preceding packets have found their memory requirement and updated the current edge array position accordingly. Then it increases the end position by the size of its buffer, marks the packet as finished, and proceeds to copy its buffer into the memory range preceding that position.

5.4 Graph Coarsening

In the coarsening stage, we first compute a disjoint clustering of the current level's graph using label propagation [RAK07] and then contract this clustering by merging all vertices in the same cluster. Our goal is to reduce the amount of auxiliary memory needed by these algorithms during their execution.

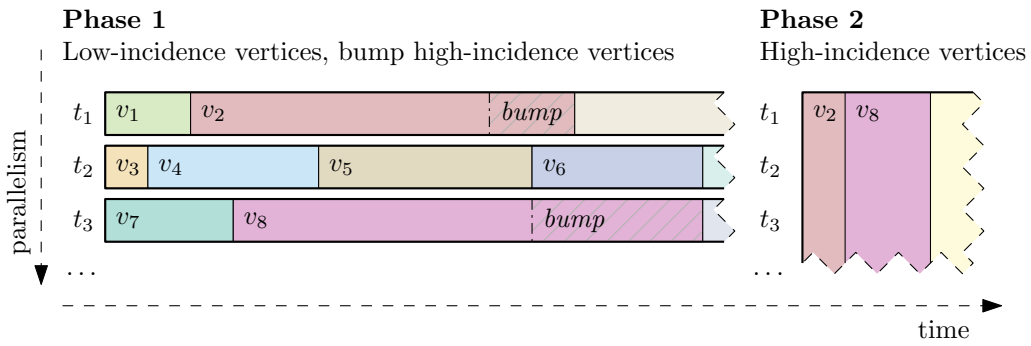

Figure 5.2 Two-Phase Label Propagation. In the first phase, vertices are processed in parallel: thread t_1 scans the neighborhoods of vertices v_1 and v_2, thread t_2 of v_3, v_4, v_5, and v_6, and so on. If a vertex is incident to many different clusters (threshold T_{bump}, e.g., v_2 and v_8), it gets *bumped* to the second phase. In the second phase, bumped vertices are processed one at a time, but with parallelism employed over their neighborhoods.

5.4.1 Label Propagation Clustering

Recall from Section 4.3.1 that we use the label propagation [RAK07] algorithm to cluster the graph during the coarsening phase. We represent the clustering as an array $\mathcal{C}$ of size n, mapping vertices to their assigned cluster. Initially, each vertex starts in its own cluster, i.e., $\forall u \in V : \mathcal{C}[u] = u$. Then, we iterate through the vertices in parallel, determine the best cluster c to join for a given vertex u, and update the clustering $\mathcal{C}[u] \leftarrow c$. See Algorithm 6 for pseudocode of one round of label propagation. Each vertex is considered once per round. We perform five rounds before contracting the clustering.

Rating Maps. For a given vertex u, each cluster has an associated *rating*: The sum of edge weights from u to the cluster. To determine the best cluster to join, we iterate through $N(u)$ and aggregate the ratings in a *rating map* data structure. The rating map can be implemented as either a hash table mapping cluster IDs to ratings or as a *sparse array*, i.e., an array $\mathcal{A}$ of size n storing the rating for cluster c at position c and a vector L storing non-zero entries in $\mathcal{A}$, which is used to reset $\mathcal{A}$ after a vertex is processed. Hash tables use less memory (proportional to the maximum degree) but tend to have slower lookups in general, which is why sparse arrays are often preferred. Each thread needs its own rating map since we parallelize over the vertices. Hence, the rating maps are the single largest contributor to the peak memory footprint of clustering in Figure 5.1. However, for vertices with small degree, the repeated local probing in hash tables can be faster due to better cache efficiency than random accesses to the sparse array [Got+21a; Got+21b]. More precisely, the hash table size is proportional to the number of unique clusters in the neighborhood $\text{nc}(u) \coloneqq |\{\mathcal{C}[v] \mid v \in N(u)\}|$. The hash table remains efficient as long as this number is small.

Two-Phase Label Propagation. This motivates a simple adaptation to substantially reduce memory and preserve performance. We split the label propagation round into two phases as shown in Algorithm 7. First, we try to process all vertices in parallel with small fixed-capacity hash tables (no dynamic growth). While processing a vertex u, if $\text{nc}(u)$ exceeds a threshold T_{bump}, we stop and *bump* it to the second phase. In the second phase, we process the few bumped vertices sequentially with a sparse array as the rating map and employ parallelism over the edges. As we only use one sparse array, we need just $O(n + p \cdot T_{\text{bump}})$ instead

Algorithm 7 Two-Phase Label Propagation Round

```
1  for u ∈ V do parallel                                          // First Phase
2      R ← empty hash table [thread-local, fixed-capacity]
3      for v ∈ N(u) do
4          R[C[v]] ← R[C[v]] + ω(uv)
5          if |R| ≥ T_bump then
6              Bump u and continue with next vertex
7      C[u] ← arg max_{c ∈ R.keys()} R[c]
8  A ← allocate zero-initialized array of size n
9  for each bumped vertex u do                                    // Second Phase
10     for v ∈ N(u) do parallel
11         R_t[C[v]] ← R_t[C[v]] + ω(uv)
12         if |R_t| ≥ T_bump then
13             FlushRatingMap(A, R_t, L_t)
14     FlushRatingMap(A, R_t, L_t) ∀t ∈ [p]
15     C[u] ← arg max_{c ∈ ⋃_{t∈[p]} L_t} A[c]
16     A[c] ← 0 ∀c ∈ ⋃_{t∈[p]} L_t
17 Function FlushRatingMap(A, R_t, L_t)
18     for (c, w) ∈ R_t do
19         w_prev ← A[c] +=^atomic w
20         if w_prev = 0 then
21             Append c to thread-local vector L_t
22     R_t.clear()
```

of $O(n \cdot p)$ memory, where p is the number of processors. The term $p \cdot T_{\text{bump}}$ is negligible compared to n on current machines and graphs where reducing memory usage matters. Since the degree of a bumped vertex is at least T_{bump}, there is sufficient work to justify parallelism over the edges.

Parallel Sparse Array. With parallelism over edges there is a race condition on the cluster ratings in the sparse array $\mathcal{A}$. Therefore, we use atomic fetch-add instructions to aggregate the scores safely, see line 19 of Algorithm 7. We implement the list L of non-zero entries as thread-local buffers L_t. To prevent duplicate cluster IDs in $L = \bigcup_{t \in [p]} L_t$, only the thread that raises the rating of cluster c from $\mathcal{A}[c] = 0$ to $\mathcal{A}[c] > 0$ tracks c in its buffer. The atomic fetch-add instruction returns the value just *before* the operation, which we check in line 20. Another concern is contention. We only process vertices with high $\text{nc}(u)$ in the second phase, so that rating contributions are spread across many clusters and thus memory locations, which helps to some extent. However, it is still possible that few clusters receive a majority of the atomic increments. To reduce contention, we leverage the hash tables from the first phase as intermediate buffers, to perform fewer atomic updates overall. Once a hash table reaches capacity or all neighbors are traversed, we flush the hash table by applying its entries

to $\mathcal{A}$ with the atomic fetch-add instruction.

Running Time. Recall from Section 4.3.1 that our original parallelization of size-constrained label propagation runs in $\mathcal{O}\left(\frac{m}{p} + \Delta\right)$ time with span $\mathcal{O}(\Delta)$ and requires $\mathcal{O}(np)$ additional memory for auxiliary data structures. With the two-phase label propagation scheme described above, we reduce the auxiliary memory to $\mathcal{O}(n + pT_{\text{bump}})$. However, when implementing as described, this does not improve the $\mathcal{O}(\Delta)$ span: arbitrarily large neighborhoods may still be processed sequentially during the first phase of the algorithm whenever the neighbors occupy only a small number of distinct clusters. Moreover, up to m/T_{bump} vertices can be bumped to the second phase. In our experiments, we fix T_{bump} to a constant. Overall, this variant performed best during preliminary testing.

We also analyze a slightly more streamlined variant, in which vertices v are bumped to the second phase solely based on their degree, i.e., whenever $d(v) \geq T_{\text{bump}}$. To ensure sufficient parallelism in the second phase, it is more robust to choose $T_{\text{bump}} \in \Theta(p)$. Then, the first phase runs in time $\mathcal{O}\left(\frac{m}{p} + \min\{p, \Delta\}\right)$ with span $\mathcal{O}(\min\{p, \Delta\})$. In the second phase, at most $n_{\text{bump}} \leq \frac{m}{p}$ vertices are processed sequentially, since every bumped vertex has degree at least p, yielding time $\mathcal{O}\left(\frac{m}{p}\right)$ and span $\mathcal{O}\left(\frac{m}{p}\right)$. Overall, the variant runs in time $\mathcal{O}\left(\frac{m}{p} + \min\{\Delta, p\}\right)$ with span $\mathcal{O}\left(\frac{m}{p} + \min\{p, \Delta\}\right)$ and uses $\mathcal{O}\left(n + p^2\right)$ additional memory. While this worst-case analysis appears pessimistic, practical performance is often superior. In graphs with small maximum degree (such as meshes), vertices are rarely bumped to the second phase, effectively removing the sequential bottleneck. In scale-free graphs (such as social networks), the number of bumped high-degree vertices n_{bump} is typically much smaller than the conservative upper bound of m/p. Crucially, since $p \ll \Delta$ in these skewed graphs, capping the sequential scanning of neighbors at p (rather than Δ) often improves load balancing considerably in practice.

5.4.2 Contraction

Given the clustering $\mathcal{C}$, we want to construct the CSR representation $(\mathcal{P}', \mathcal{E}')$ of the contracted graph $G' = (V', E')$ to obtain the next level in the multilevel hierarchy. In Section 4.3.2, we described an efficient implementation that uses auxiliary memory of size $\Theta(|E'|)$ to temporarily store $\mathcal{E}'$ out of order. Before we improve the memory efficiency of this process, we briefly restate the main steps of the operation. Let $\mathcal{C}^{-1}[c] = \{u \in V \mid \mathcal{C}[u] = c\}$ be the set of vertices in the cluster with ID c. Recall that the clusters in G correspond to vertices in G' and intra-cluster edges $(u, v) \in E$ with $\mathcal{C}[u] \neq \mathcal{C}[v]$ in G become edges $(\mathcal{C}[u], \mathcal{C}[v])$ in G'. The weight of an edge (a, b) in G' is

$$\omega'(a, b) = \sum_{u \in \mathcal{C}^{-1}[a], v \in \mathcal{C}^{-1}[b], (u,v) \in E} \omega(u, v),$$

i.e., the sum of weights of duplicate edges after remapping (u, v) to $(\mathcal{C}[u], \mathcal{C}[v])$. For each cluster a, we compute its outgoing edges in G' by iterating over the vertices $u \in \mathcal{C}^{-1}[a]$ and scanning all outgoing edges (u, v) of u in G, adding $\omega(u, v)$ to the edge weight $\omega'(a, \mathcal{C}[v])$. Note the similarity to rating aggregation in label propagation.

Two-Phase Aggregation. In Figure 5.1, we observe that after label propagation coarsening and FM refinement, contraction is the third largest contributor to the peak memory. Although sparse arrays only require $n' = |V'| \ll n$ number of entries in the contraction step, this

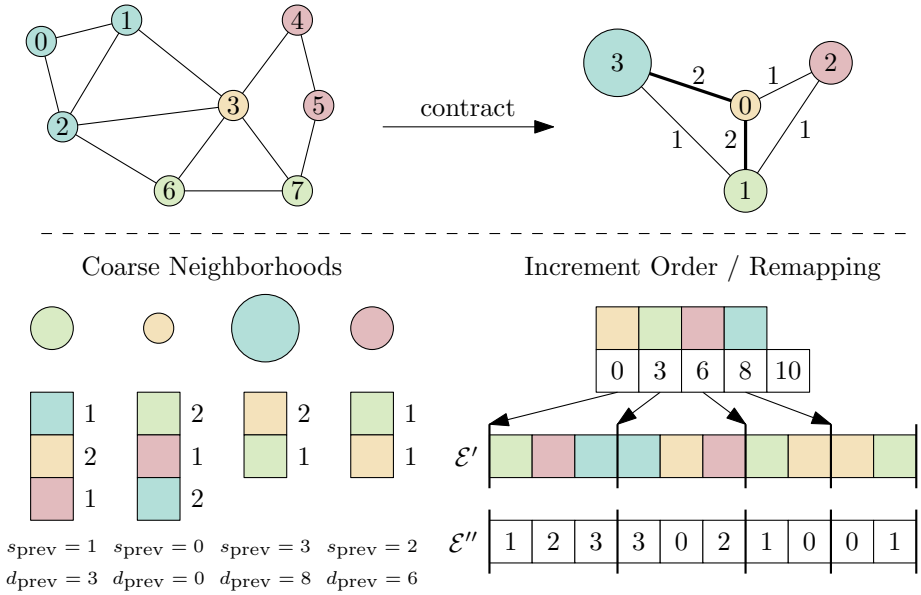

Figure 5.3 Illustration of the contraction step. After computing a coarse neighborhood, we increment the counters s and d to obtain the start position d_{prev} in $\mathcal{E}'$ and new coarse vertex ID s_{prev}. Once all neighborhoods are inserted, we remap the old cluster IDs (denoted as colors) to the new coarse vertex IDs (numbers).

is a good avenue for optimization. Therefore, we also apply our two-phase aggregation approach from Section 5.4.1 here. In the first phase, we employ parallelism over the clusters. Within a cluster c, we scan through its contained vertices $u \in \mathcal{C}^{-1}[c]$ and their neighborhoods sequentially. For each neighbor $v \in N(u)$, we add $\omega(u, v)$ to the edge weight $\omega'(c, C[v])$. In the second phase, we process one cluster at a time, but employ parallelism over $\mathcal{C}^{-1}[c]$, and potentially the neighborhoods if the degree in G is sufficiently high.

One-Pass Contraction. The second piece of auxiliary memory used in contraction as described in Section 4.3.2 is a set of temporary buffers storing E' during aggregation; before the edges are copied to $\mathcal{E}'$. The difficulty with using CSR in the parallel setting is the necessity to know each vertex degree and offset $\mathcal{P}'$ (prefix sum over the degrees), before edges can be written to $\mathcal{E}'$ in parallel. In the sequential setting, one can simply compute the prefix sum at the same time as appending edges. In the parallel setting however, the degrees of vertices whose edges should appear earlier in $\mathcal{E}'$ may not be known yet. Once all coarse edges are computed, we can compute the prefix sum over the degrees to obtain the offsets $\mathcal{P}'$, and then copy the edges to the appropriate locations in $\mathcal{E}'$. In the remainder of this section, we describe how to avoid the buffers, so that we store the coarse graph only once, in CSR format, and only compute the coarse edges once. Note that a simpler approach is to compute the coarse edges twice: Once to count the degrees, and a second time to place the edges in $\mathcal{E}'$. However, this is not an option as it effectively doubles the running time of the contraction algorithm.

Similar to the single-pass I/O in Section 5.3.2, we employ virtual memory overcommitment to allocate $2m$ entries for $\mathcal{E}'$ without physical memory backing, as we do not know the true $m' = |E'|$ yet. We append newly computed coarse edges to $\mathcal{E}'$, thus only $2m'$ entries plus at most one page of memory are physically allocated. Note that we use $2m'$, as we compute

and store each undirected edge as two directed edges in opposite directions.

To some extent, our approach to resolve the prefix sum issue mimics the sequential approach, but uses atomic instructions to fix race conditions and uses buffering to hide contention from the use of atomics. Let $d \leftarrow 0$ be an index counting the number of edges already inserted to $\mathcal{E}'$ and $s \leftarrow 0$ denote the number of coarse vertices already processed. At the end we will have $d = 2m'$ and $s = n$. See also Figure 5.3 for an illustration.

The following applies to the first phase, where coarse vertices are processed in parallel. For each coarse vertex $u' \in V'$, we start computing its coarse edges and store them in a thread-local hash table R_t. As soon as $|R_t| \geq T_{\text{bump}}$, we bump u' to the second phase. Otherwise, R_t contains all edges of u'. We increment $d \overset{\text{atomic}}{+=} |R_t|$ and capture the value d_{prev} immediately before the increment. Then we copy R_t to the range $\mathcal{E}'[d_{\text{prev}} : d_{\text{prev}} + |R_t|]$, as illustrated at the bottom of Figure 5.3.

Note that due to parallelism, the neighborhoods in $\mathcal{E}'$ are out of order, i.e., are not stored in the same order as the coarse vertex IDs. Thus, one can either store a begin and end pointer to the neighborhood of each vertex (which requires twice the memory), or relabel the vertices, so that the neighborhoods of consecutive vertex IDs are consecutive in $\mathcal{E}'$. To this end, we increment $s \overset{\text{atomic}}{+=} 1$ for each coarse vertex processed, in a transaction combined with the update $d \overset{\text{atomic}}{+=} |R_t|$ as they must be consistent. Again, we capture s_{prev} as the value before the transaction and save the beginning of the neighborhood in $\mathcal{P}'[s_{\text{prev}}] \leftarrow d_{\text{prev}}$. Additionally, we store the new vertex ID s_{prev} so that we can remap the endpoints of the coarse edges at the end, thus avoiding to shuffle the neighborhoods in $\mathcal{E}'$.

To synchronously update d and s in a transaction, we employ the double-width compare-and-swap instruction [Int24]. The two counters are stored as a 128-bit integer, with d stored in the lower 64 bits and s in the upper bits. We implement the transaction as a compare-and-swap loop, where we extract, update and repack the values in the loop body.

In order to perform fewer compare-and-swap instructions and thus reduce contention, we increment the dual counter for several coarse vertices at once. To implement this, we store the neighbors of multiple coarse vertices in another fixed-capacity buffer B_t. Once the buffer reaches capacity, or no coarse vertices are left to process, we increment d by $|B_t|$ and s by the number of coarse neighborhoods stored in the buffer. In the second phase where coarse high-degree vertices are processed sequentially, there is no need to atomically update d and s.

5.5 Space-Efficient Gain Tables

A common performance optimization in refinement algorithms is a *gain table*. For each pair of a vertex $u \in V$ and a block V_i, we cache the value $\omega(u, V_i) = \sum_{(u,v)\in E, v\in V_i} \omega(u, v)$ in the table, which we call the *affinity* of a vertex to a block. Let $\Pi(u)$ denote the block to which u is assigned. With cached affinity values, the gain of moving u to V_i can be computed as $\omega(u, V_i) - \omega(u, \Pi(u))$. After moving a vertex u from block V_s to V_t, we update the affinity for each neighbor v of u as $\omega(v, V_s) \mathrel{-}= \omega(u, v)$ and $\omega(v, V_t) \mathrel{+}= \omega(u, v)$ using atomic fetch-add instructions. This optimization is standard practice for algorithms such as FM [FM82] where vertices' gains will be inspected substantially more often than moves performed, such that the overhead of updating the gain table after a move is lower than computing the gain from scratch, each time the vertex is inspected.

Unfortunately, the standard implementation of gain tables uses k entries per vertex for a total of $O(nk)$ memory, which is infeasible for very large graphs and even moderate values of k. Instead, we would like to use $O(m)$ memory. The idea is to use the standard implementation with k entries only for vertices with $\deg(v) > k$. For vertices with lower degree, we use tiny

Table 5.1 Graphs of Benchmark Set C, characterized by the number of vertices (n), the number of undirected edges (m), and the average (d) and maximum (Δ) degree.

Graph	$n/10^3$	$m/10^3$	d	Δ	Ref.
Hyperlink	3 563 603	112 243 992	71	45 676 832	[Meu+15]
Eu2015	1 070 557	80 528 516	150	20 252 259	[Lab]
Uk2014	787 801	42 464 216	107	8 605 492	[Lab]
Clueweb12	978 408	37 372 179	76	75 611 696	[Lab]
Gsh2015	988 491	25 690 705	51	58 860 305	[Lab]

linear-probing hash tables with fixed capacity $\Theta(\deg(v))$, as the vertex can be adjacent to at most $\deg(v)$ different blocks. The affinity value for non-adjacent blocks is zero, which we do not store explicitly. This leads to a memory footprint of $O(\sum_{v \in V} \min(\deg(v), k)) \subset O(m)$.

The updates for affinity values of low-degree vertices cannot be applied with atomic fetch-add instructions, as values that drop to zero must be removed. Since we anticipate far more queries than deletions, we move up elements to close gaps in the probing order upon deletion [San+19]. Therefore, positions in the hash table are not stable such that each hash table must be protected by a spinlock. Somewhat surprisingly, these overheads have a minor impact on running time, as we demonstrate in the experiments.

The memory for all affinity values is allocated in one contiguous array. Note that the affinity values of a vertex are upper bounded by its total incident edge weight U. To reduce the memory footprint of the gain table further, we choose a variable width w (8, 16, 32 or 64 bits) for the entries of each vertex as the smallest value $w > \log_2(U)$. Furthermore, for each vertex, we keep a pointer to the beginning of the slice of memory storing its affinity values.

5.6 Experiments

We have integrated the described algorithms into the KaMinPar partitioner introduced in Chapter 4. We denote the resulting algorithm as TeraPart. We compile all codes using GCC-13.2.0 with flags `-march=native -msse4.1 -mcx16` and use Intel TBB [TBB] for parallelization.

Setup. We perform our experiments on a machine equipped with a 96-core AMD EPYC 9684X processor clocked at 2.55 GHz (with boost up to 3.70 GHz), and with 1 152 MB L3 cache (Machine D in Table 2.2). The machine is further equipped with 1 536 GiB of main memory and runs Ubuntu 24.04. Unless stated otherwise, we obtain our results using all 96 cores.

Instances. To cover a diverse range of medium sized graphs, we evaluate our optimizations on Benchmark Set A introduced in Section 4.4, see Tables 4.1 and 4.2. Additionally, we perform experiments on several huge web graphs from the Laboratory for Web Algorithmics [Bol+11; Bol+18; BV04], namely gsh-2012, clueweb12, uk-2014 and eu-2015, as well as the hyperlink(-2012) [Meu+15] graph. We refer to this set of graphs as Benchmark Set C. We converted these graphs to undirected graphs by adding missing reverse edges and removing any self-loops. Details for these graphs are shown in Table 5.1.

To perform experiments on tera-scale graphs, we further use 2D random geometric (denoted rgg2D) and hyperbolic (denoted rhg) graphs. We use the KaGen [Fun+18] graph generator to generate these graphs. rgg2D graphs do not have any high-degree vertices and

resemble mesh-like graphs, whereas rhg graphs have a skewed power-law degree distribution and model real-world social networks. We generate these graphs with average degree $d = 256$ and power-law exponent $\gamma = 3.0$ (rhg only).

Methodology. As before, we consider the combination of a graph and the number of blocks k as an *instance*. Unless stated otherwise, we use $\varepsilon = 3\%$, $k \in \{8, 37, 64, 91, 128, 1\,000, 30\,000\}$ (i.e., arbitrary small and large values of k), perform 5 repetitions for each instance using different seeds, and use all 96 cores of our shared-memory machine. We aggregate running times, peak memory consumption and edge cuts for the same instance using the arithmetic mean over all seeds. When aggregating over multiple instances, we use the geometric mean for all metrics.

To measure the peak memory consumption of our partitioners (i.e., KaMinPar, TeraPart and variations thereof), we instrumented the code to record all dynamic allocations during execution. For competing partitioners (i.e., Mt-METIS and ParMETIS), we measure the process' Resident Set Size (RSS). In the case of KaMinPar, these two metrics can diverge because the algorithm uses `mmap` for graph I/O. Consequently, RSS may include the memory-mapped input file in the peak memory count. Our instrumentation avoids this distortion, reflecting the actual peak memory consumption more accurately. Therefore, we rely on manually tracked memory for self-comparisons. However, to ensure fairness, we use RSS for all algorithms when comparing against Mt-METIS or ParMETIS.

All graphs are stored on disk in an uncompressed binary format. Compression happens on-the-fly during the streaming process into memory, introducing some overhead compared to directly loading the graphs into memory. To keep this overhead low, we have parallelized the compression process as described in Section 5.3.2. For instance, it takes 2 905 s and 572 s to load the Eu2015 graph sequentially with and without on-the-fly graph compression from a RAID 0 with two Micron 7450 PRO NVMe SSDs. However, with 96 cores, I/O times reduce to 179 s and 177 s, respectively. Since the overhead with parallel I/O is thus negligible, we generally exclude I/O times from the measurements. Including I/O times would also be unfair to the competitors Mt-METIS and ParMETIS, since they read graphs in a text format and thus exhibit overheads due to tokenizing the input string.

To compare the edge cuts of different algorithms, we use performance profiles [DM02]. They are introduced in Section 2.5.

5.6.1 In-Memory with Label Propagation Refinement

We first evaluate the impact of our proposed optimizations using KaMinPar with label propagation refinement as the baseline. We obtain TeraPart by enabling our optimizations one after the other in the following order:

(i) two-phase label propagation (Section 5.4.1),
(ii) graph compression (Section 5.3.1) and
(iii) TeraPart: one-pass cluster contraction (Section 5.4.2).

Medium-Sized Graphs. In Figure 5.4, we evaluate the impact of the proposed optimizations on running time and peak memory using the medium-sized graphs from Benchmark Set A. Figure 5.5 additionally evaluates partition quality. We enable our optimizations one after the other to transform our baseline KaMinPar into TeraPart.

First, two-phase label propagation improves both memory consumption and execution time of KaMinPar. It reduces the average memory usage by 25.2%, from 3.42 GiB to

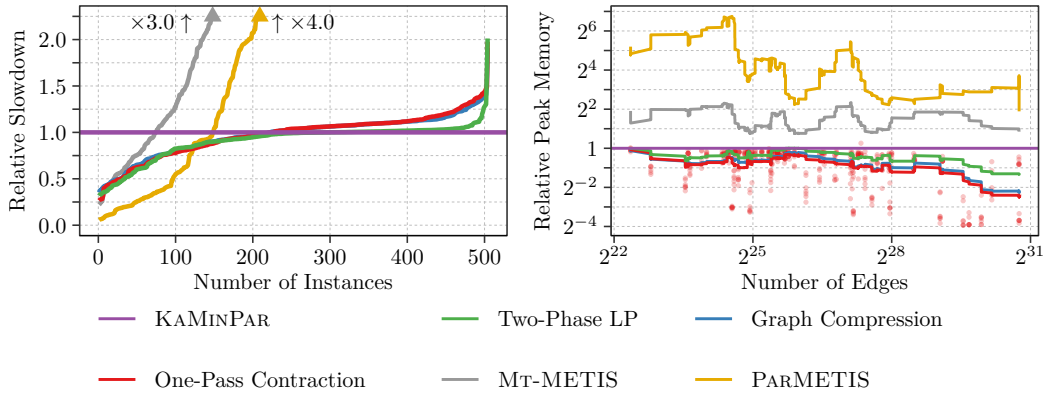

Figure 5.4 Relative running times (left) and peak memory (right) on Benchmark Set A for TeraPart relative to KaMinPar when enabling the following optimizations one after the other: (i) two-phase label propagation, (ii) graph compression, and (iii) one-pass cluster contraction. We also include Mt-METIS and ParMETIS for reference. For peak memory, we plot the per-instance ratios for TeraPart with a right-aligned rolling geometric mean over 50 instances (shown for all algorithms).

2.56 GiB, while also decreasing running time by 11.7%, from 1.53 s to 1.35 s. The memory savings are due to now allocating only a single array of size $\mathcal{O}(n)$ for clustering rating aggregation instead of p such arrays. As stated before, this reduces the overall memory from $\mathcal{O}(np)$ to $\mathcal{O}(n)$ for this step. The speedup stems from the improved load balance in the second phase of two-phase label propagation. Previously, irregular graphs with vertices of very high degree incurred sequential bottlenecks, since parallelism was exploited only across neighborhoods, not within them. Indeed, looking only at graphs that feature a high maximum degree (more specifically, $\Delta(G) \geq T_{\text{bump}} = 10\,000$), we observe a more pronounced improvement in running time (by 23.7%).

Gap and interval encoding achieve an average compression factor of 3.1 across the entire benchmark set, although compression ratios vary greatly depending on the application domains of the graphs. For instance, compression ratios range from slightly below 1 for Kmer graphs [DH11] to 5.7 for graphs derived from finite element simulations. When we enable graph compression in our experiment (i.e., compare Graph Compression against Two-Phase LP in Figure 5.4), these ratios translate to a further 23.7% reduction in peak memory. Unfortunately, decoding the compression at runtime is not free. On average, it increases running time by 6.2%. However, combined with two-phase label propagation, we still get a speedup over the KaMinPar baseline by 5.9%, while reducing peak memory by 43.0%.

Finally, we obtain TeraPart by switching to our one-pass contraction algorithm, which further reduces both peak memory consumption (to 1.78 GiB) and running time (to 1.43 s) by 8.7% and 0.7%, respectively. With all three optimizations enabled, TeraPart consumes 47.9% less memory than KaMinPar, while being 6.5% faster. Originally, we only set out to reduce the memory consumption of KaMinPar. Perhaps surprisingly, we can conclude from our experiments that TeraPart not only optimizes the memory consumption of KaMinPar by cutting it almost in half, but also improves its running time slightly.

As shown in Figure 5.4 (right), the memory savings of TeraPart are more pronounced on the larger graphs of the benchmark set. Indeed, considering only graphs with $m \geq 10^8$ edges (182 out of 504 instances), average peak memory reduction increases to 64.3%, and further increases to 81.1% when considering only graphs with $m \geq 10^9$ edges (28 instances).

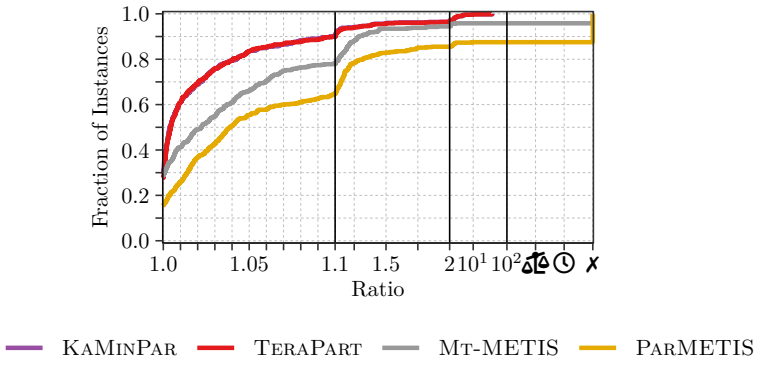

Figure 5.5 Comparison of solution quality on Benchmark Set A between our baseline KaMinPar, the optimized TeraPart (One-Pass Contraction in Figure 5.4), and the competitors Mt-METIS and ParMETIS. Instances for which Mt-METIS or ParMETIS did not produce a result are marked with ✗. Note that the curves of KaMinPar and TeraPart lie on top of each other, indicating that our optimizations do not affect solution quality.

Note that the largest graphs (by edge count) of the benchmark set are classified as irregular, i.e., feature high maximum degrees. Since two-phase label propagation only reduces memory usage and running time on these graphs, observing larger improvements is expected.

Note that our optimizations should not change the behaviour by which KaMinPar computes a partition. Thus, we would expect partition quality for TeraPart and KaMinPar to be the same. To confirm this expectation, we look at the performance profile in Figure 5.5. We see that the curves for both algorithms lie on top of each other, indicating that our optimizations indeed do not affect the solution quality of the partitioner. More precisely, the average edge cuts of both partitioners are within 0.04% of each other, which is insubstantial.

To contrast our results against well established graph partitioners, we also include Mt-METIS and ParMETIS in Figures 5.4 and 5.5. Mt-METIS fails to produce any partition for the three largest graphs (Twitter2010, Friendster and Sk2005) in our medium-sized benchmark set for all values of k. For the remaining graphs, we find that Mt-METIS is on average 4.0 times slower than KaMinPar while consuming 2.7 times more memory than even our unoptimized KaMinPar baseline. Compared to the optimized TeraPart, Mt-METIS is 4.3 times slower while consuming 4.3 times more memory on average. We further observe that Mt-METIS does not always respect the balance constraint, producing imbalanced partitions for 314 out of 504 instances. However, looking at Figure 5.5, we see that even when ignoring the infeasibility of the partitions, TeraPart still finds partitions of similar quality that are also balanced. The distributed partitioner ParMETIS fails to compute partitions for 63 instances and computes imbalanced partitions for 228 instances. On the instances on which it does compute a partition, it is 3.1 times slower than TeraPart while consuming 24.5 times more memory.

Multi-threaded Scalability. Figure 5.6 shows the scalability of TeraPart for $\varepsilon = 3\%$ and $k \in \{64, 30\,000\}$ on the medium-sized graphs of Benchmark Set A using $p \in \{12, 24, 48, 96\}$ cores of our shared-memory machine. The overall geometric mean speedup is 9.1 for $p = 12$, 14.4 for $p = 24$, 19.8 for $p = 48$, and 22.4 for $p = 96$. The speedup achieved with 96 cores over 48 is somewhat limited when averaging over all instances. This is mostly due to the

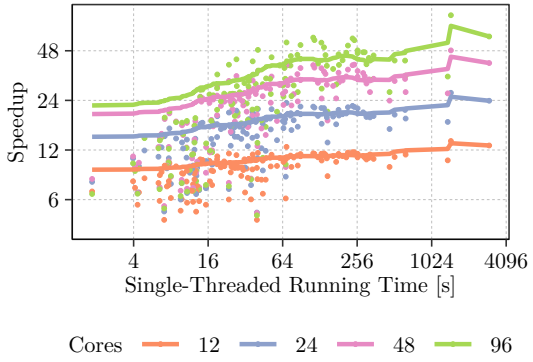

Figure 5.6 Self-relative speedups for TeraPart with $p \in \{12, 24, 48, 96\}$ cores and $k \in \{64, 30\,000\}$. We sort instances i by their sequential running time t^i and plot the cumulative geometric mean speedup of all instances with sequential running time $t^i \geq t$ at position t.

initial partitioning phase, which can only make full use of the available parallelism once the graph is partitioned into a sufficiently large number of blocks $k' \geq p$. On larger graphs which require at least 64 s of sequential processing time (48 out of 144 instances), our speedups increase to 10.8, 19.2, 30.5, and 39.6 for 12, 24, 48, and 96 cores, respectively. For the 15 instances which require at least 256 s of sequential processing time, we measure a speedup of 42.9 on $p = 96$ cores. Note that TeraPart does not perform any expensive arithmetic operations and is limited by memory bandwidth. Thus, perfect speedups are not possible.

Huge Web Graphs. Memory savings are arguably more important for huge graphs, and the graphs in Benchmark Set A are rather small compared to the terabytes of memory available in modern high-end servers. Thus, we now turn towards the much larger web graphs of Benchmark Set C (up to 112 G undirected edges, confer Table 5.1) to further showcase the scalability of TeraPart. Compression ratios (presented in Figure 5.8) for these graphs range from 5 for Hyperlink to just over 11 for Eu2015. Interval encoding is crucial for these graphs, as gap encoding alone only achieves more moderate compression ratios of 2.7 to 3.4.

In terms of running time (shown in Figure 5.7, left), we generally observe the same pattern as before, but more pronounced, with two-phase label propagation being the most impactful optimization. In particular, we observe a speedup of 4.6× for the Eu2015 graph when enabling two-phase label propagation (which reduces to 3.1× after enabling graph compression and one-pass cluster contraction). Graph compression generally increases running time, while one-pass contraction slightly decreases it. Overall, we observe speedups ranging from 1.2× on Clueweb12 to 3.1× on Eu2015 over the KaMinPar baseline.

TeraPart further only uses a fraction of the memory required by KaMinPar, as shown in Figure 5.7 (right). There, we highlight $k = 30\,000$. For smaller k, the graphs shrink faster and thus memory savings by one-pass contraction are less pronounced (7.5% vs 40.7% reduction), while the influence of two-phase label propagation and graph compression remains similar. On Gsh2015, Clueweb12, Uk2014 and Eu2015, KaMinPar uses 12.9, 12.5, 15.7 and 15.7 times more memory than TeraPart (averaged over all k), respectively. For the largest graph in the benchmark set, Hyperlink, KaMinPar would require roughly 2.4 TiB RAM (measured on a different machine with 4 TiB of main memory), and thus runs out of memory on our 1.5 TiB machine. In contrast, TeraPart is able to partition this graph with just

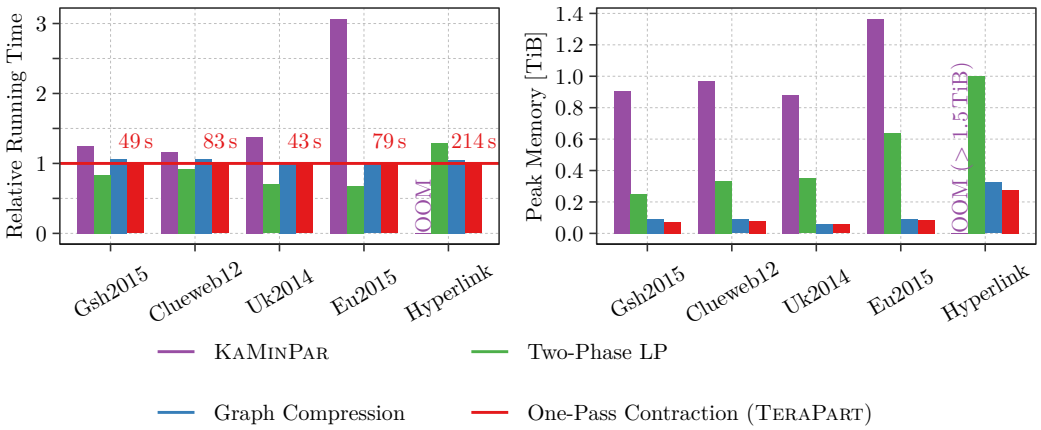

Figure 5.7 Relative running times (left, all k values) and peak memory (right, $k = 30\,000$) on Benchmark Set C for TeraPart relative to KaMinPar when enabling the following optimizations one after the other: (i) two-phase label propagation, (ii) graph compression, and (iii) one-pass cluster contraction. Edge cuts for TeraPart are listed in Table 5.2.

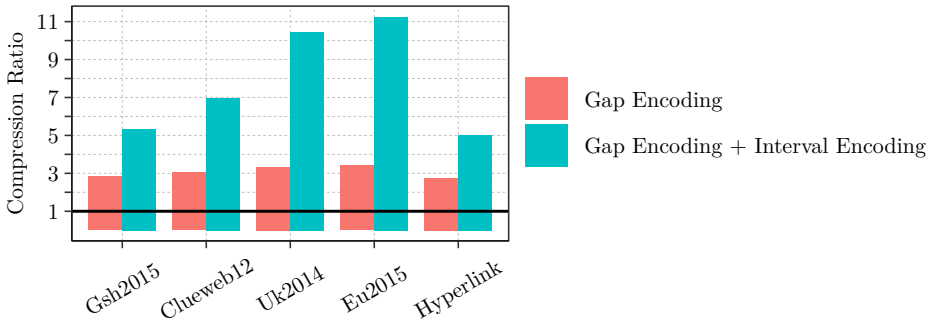

Figure 5.8 Compression ratios of the huge web graphs, with just gap encoding and gap encoding plus interval encoding.

279 GiB–306 GiB of memory, depending on k. Moreover, it is able to do so in less than 4 min on our 96 core CPU.

Scaling to a Trillion Edges. To further explore the scalability of TeraPart, we generate synthetic tera-scale rgg2D and rhg graphs using the aforementioned properties. The generated rgg2D resp. rhg graphs have 8.59 billion vertices and 1.10 resp. 1.01 trillion (undirected) edges, which would require 16.1 TiB resp. 14.8 TiB to be stored as uncompressed CSR. TeraPart is able to compress these graphs down to just 1 194 GiB resp. 608 GiB (compression ratio of 14.2 resp. 26.3), allowing for in-memory partitioning on a single shared-memory machine. Partitioning into $k = 30\,000$ blocks takes just 663 s (rgg2D) resp. 467 s (rhg) using 96 cores while cutting 1.48% resp. 0.45% of the edges. In total, TeraPart requires 1.46 TiB and 886 GiB memory to partition the rgg2D and rhg graphs, respectively, allocating 304 GiB and 278 GiB of memory to auxiliary data structures.

Table 5.2 Edge cuts corresponding to Figure 5.7 for $k = 64$. We report the edge cut for TeraPart (-LP) as percentage of total edges cut and the edge cut of TeraPart-FM relative to TeraPart-LP.

Graph	Algorithm	Cut	Time	Memory
Gsh2015	TeraPart-LP	3.09%	46 s	68 GiB
	TeraPart-FM	0.92×	355 s	125 GiB
Clueweb12	TeraPart-LP	10.99%	71 s	71 GiB
	TeraPart-FM	0.94×	1 468 s	170 GiB
Uk2014	TeraPart-LP	0.13%	56 s	55 GiB
	TeraPart-FM	0.95×	65 s	102 GiB
Eu2015	TeraPart-LP	0.32%	77 s	87 GiB
	TeraPart-FM	0.96×	196 s	158 GiB
Hyperlink	TeraPart-LP	1.68%	191 s	279 GiB
	TeraPart-FM	0.87×	2 204 s	538 GiB

5.6.2 In-Memory with Parallel FM Refinement

We now equip TeraPart with FM refinement and evaluate the space-efficient gain tables introduced in Section 5.5. In this section, we use TeraPart-LP to refer to the baseline variant of TeraPart with label propagation only, and TeraPart-FM to refer to TeraPart additionally equipped with FM refinement using the space-efficient gain tables.

Figure 5.9 reports the results for running times and memory usage across Benchmark Set A. As can be seen in Figure 5.9 (right), our space-efficient gain tables reduce the memory consumption of FM refinement considerably. On average, they require 2.8× less memory than FM refinement with the standard $\mathcal{O}(nk)$ memory gain tables. Despite their higher per-operation cost (hash table lookups instead of array accesses, spin locks instead of lightweight atomic operations) in theory, Figure 5.9 (left) shows only a small runtime overhead. The average running time increases by just 1.1% compared to the full table baseline. Moreover, when we restrict the analysis to graphs for which TeraPart-FM uses more than 8 GiB of memory, we find that our space-efficient gain tables reduce peak memory by a factor of 6.3 compared to the full gain tables, on average, while the penalty in running time vanishes (to -0.25%, i.e., a slight speedup). For the largest graph (in terms of n) in Benchmark Set A (KmerV1r), the standard gain table runs out of memory for $k = 1\,000$. With space-efficient gain tables, we can successfully partition this graph using only 14.7 GiB of memory.

Using no gain table at all (i.e., gains are repeatedly recomputed from scratch instead) is not a viable option, as shown in Figure 5.9 (left). Although this configuration uses no additional memory, it is 2.6 times slower than TeraPart-FM on average. While this might be tolerable for some applications, we also observe that its performance is substantially less robust: We observe slowdowns of at least one order of magnitude on 65 out of 504 instances, and it exceeds our one-hour time limit on 16 instances.

We plot the corresponding solution quality in Figure 5.10. Since all three FM configurations produce similar edge cuts, we only plot one curve. Unsurprisingly, TeraPart-FM finds better cuts than TeraPart-LP or Mt-METIS on 80% of the instances. On 50% of the instances, FM refinement reduces the edge cut by at least 4.5% compared to just label propagation refinement. While Mt-METIS finds better cuts than TeraPart-FM on roughly 15% of the instances, we note once more that most of its partitions do not respect

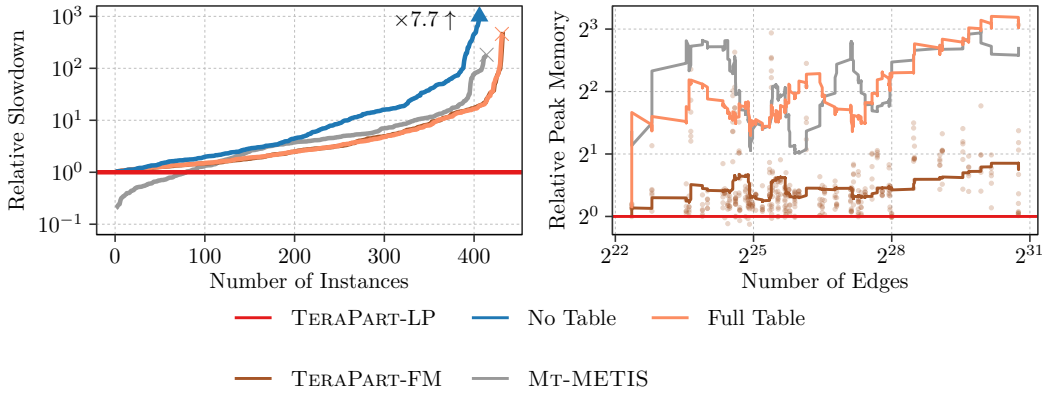

Figure 5.9 Relative running time (left) and peak memory (right) for TeraPart on Benchmark Set A, equipped with FM refinement using no gain cache (No Table), the full $\mathcal{O}(nk)$ memory gain table (Full Table), and the proposed space-efficient gain table (TeraPart-FM). For peak memory, we plot the per-instance ratios (only shown for TeraPart-FM) with a right-aligned rolling geometric mean over 50 instances. For reference, we include TeraPart without FM refinement (i.e., only label propagation refinement, denoted TeraPart-LP) and Mt-METIS. We use $k \in \{8, 37, 64, 91, 128, 1\,000\}$ since FM refinement for larger values of k yields diminishing returns, see Section 4.4.2.6. Note that in the slowdown plot, the curves of TeraPart-FM and Full Table mostly lie on top of each other.

the balance constraint (314 out of 504); we ignore this in the performance profile, i.e., count all partitions as balanced. Both TeraPart-{LP, FM} always produce balanced partitions.

With our space-efficient gain tables, FM refinement also scales to the much larger graphs of Benchmark Set C, see Table 5.2. Here, edge cut improvements range from 4% for Eu2015 to 13% for the Hyperlink graph. Mt-METIS was already unable to partition the smallest graph (Clueweb12) of this benchmark set.

5.6.3 Alternative Approaches

Besides optimizing the multilevel framework for memory efficiency as done in this paper, there are two existing approaches that reduce the burden on RAM: Semi-external memory algorithms and streaming algorithms. These approaches do not hold the graph in memory. Instead, the vertex neighborhoods are loaded one at a time from network or SSD, processed and then dropped again. As streaming algorithms perform just one pass over the graph, it is well known that they achieve sub-par solution quality compared to multilevel algorithms. We found that HeiStream [Chh+24; FS22] (which produces lower edge cuts than more basic streaming algorithms) cuts 3.1× (rgg2D) to 14.8× (rhg) more edges than TeraPart on our generated tera-edge graphs for $k = 30\,000$.

Semi-external memory algorithms perform multiple passes over the graph and can use $O(n)$ space for auxiliary data. Thus, it is possible to implement label propagation and the multilevel framework in semi-external memory [ASS15], whereas sophisticated heuristics such as FM [FM82] seem difficult. We are aware of one semi-external memory algorithm by Akhremtsev et al. [ASS15]. This algorithm is an order of magnitude slower than TeraPart, see Table 5.3. As their source code is not available, we compare against results on four graphs from their paper. To enable a fair comparison, we benchmarked TeraPart on an equivalent machine hosting two 8-core Intel Xeon E5-2650v2 clocked at 2.6 GHz. We also note that

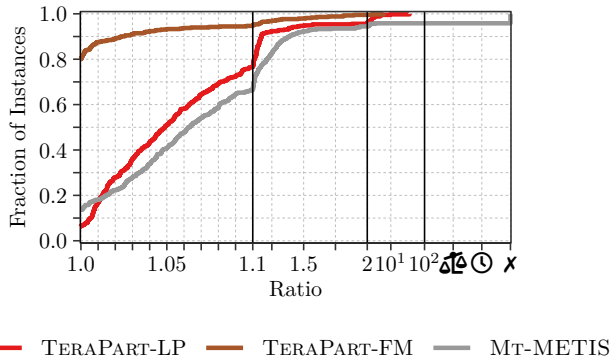

Figure 5.10 Solution quality of TeraPart with (TeraPart-FM) and without (TeraPart-LP) FM refinement. For reference, we include Mt-METIS. Since all three FM configurations produce similar edge cuts, we only plot TeraPart-FM in the performance profile (right).

Table 5.3 Comparing TeraPart against the semi-external memory partitioning algorithm (SEM) from [ASS15] with $k = 16$ and $\varepsilon = 3\%$.

Graph	Algorithm	Cut [M]	Time [s]	Memory [GiB]
Arabic2005	TeraPart	1.88	5.5	1.44
	SEM	2.28	36.0	1.85
Uk2002	TeraPart	1.45	3.2	1.05
	SEM	1.54	39.7	1.53
Sk2005	TeraPart	15.68	18.3	3.62
	SEM	22.25	203.4	7.76
Uk2007	TeraPart	4.09	27.8	6.81
	SEM	4.55	209.1	8.58

semi-external algorithms are orthogonal to our work, and combining the two approaches is mutually beneficial to enable processing even larger graphs, though we leave this direction for future work.

5.7 Conclusion

We presented TeraPart, the first multilevel algorithm capable of partitioning trillion-edge graphs on a single shared-memory machine. Showcasing TeraPart's excellent efficiency, it does so in just under 8 minutes. Moreover, its distributed version xTeraPart can partition 16-trillion edge graphs in about 10 minutes. TeraPart consists of a collection of techniques to reduce the memory consumption of the coarsening and uncoarsening stages of the multilevel framework. By splitting label propagation into two phases, we are able to reduce its auxiliary memory from $\mathcal{O}(np)$ down to $\mathcal{O}(n)$, and simultaneously alleviate a previous load balancing bottleneck. We apply the same two-phase technique to the graph contraction algorithm, and eliminate a copy of the coarse graph that was necessary due to the previous parallelization scheme. With our space-efficient gain tables, we reduce the

memory consumption for FM refinement from $\mathcal{O}(nk)$ down to $\mathcal{O}(m)$, resulting in a 5.8× peak memory reduction for graphs over 8 GiB. The last piece of the puzzle is storing the graph in a compressed format and decoding neighborhoods on-the-fly when they are required. Based on our experiments, the next avenue for optimization is to further reduce the memory for the input graph. We are both interested in improving the compression ratios, as well as integrating our techniques with a semi-external algorithm.

6 Linear-Time Multilevel Graph Partitioning

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. In this chapter, we demonstrate how to achieve the best of both worlds with a *linear-time multilevel algorithm*. Recall that multilevel algorithms construct a hierarchy of increasingly smaller graphs by repeatedly contracting sets of vertices, before computing an initial partition on the smallest graph, and then refining this partition on the successively larger graphs. Due to the lack of constraints on the size of the contracted representations, current multilevel implementations have superlinear worst-case running time. This can be problematic for 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 [Ayk+07; BDR13; SW13]. This requires the graph partitioning step to be less expensive than the downstream computation. Our approach preserves the distinct advantage of multilevel algorithms, 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 worst-case running time. We evaluate multiple approaches for edge sparsification and integrate our algorithm into KaMinPar. On a set of benchmark graphs where existing multilevel partitioners exhibit superlinear behavior, we demonstrate in detailed experiments that this results in a 1.49× average speedup (up to 4× for some instances) with only 1% loss in solution quality compared to our baseline configuration. Moreover, our algorithm clearly outperforms state-of-the-art single-level and streaming approaches, computing better partitions faster.

Contributions. In this chapter, 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. This guarantees $\mathcal{O}(n + m)$ expected total work for n vertices and m edges, without any assumptions on the input graph.

We integrate our approach into the KaMinPar partitioner, presented in Chapter 4, 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× (1.5× in the geometric mean) over a baseline KaMinPar configuration – which is, to the best of our knowledge, the fastest available shared-memory multilevel partitioner [Got+24a] (see also Section 4.4.1). Despite this, the loss in partition quality is only 1% on average. Our algorithm outperforms both the single-level partitioner PuLP [SMR14] and the state-of-the-art streaming partitioner Cuttana [HSS24], achieving 24% and 66% smaller average cuts, respectively, as well as a faster running time.

References and Contributors. This chapter is based on a conference publication published together with Lars Gottesbüren, Nikolai Maas, Dominik Rosch, and Peter Sanders [Got+25].

The text was mainly written by Nikolai Maas, Dominik Rosch and the author of this dissertation with editing by Lars Gottesbüren and Peter Sanders. The ideas described here were developed jointly by all authors. The initial implementation of the algorithms was done by Dominik Rosch as part of his bachelor thesis [Ros24], with further running time optimizations by the author of this dissertation.

Structure. The remainder of this chapter is organized as follows. In Section 6.1, we briefly review related work on coarsening strategies and their implications on linear time multilevel graph partitioning, as well as graph sparsification techniques. Section 6.2 subsequently discusses the current running time bottlenecks in KaMinPar, and describes edge sparsification techniques to achieve worst-case linear time. Section 6.3 evaluates the sparsification techniques and their impact on empirical running time and solution quality, while Section 6.4 concludes this chapter.

6.1 Related Work

We introduce most of the general work in Chapter 3 and only focus on work closely related to the contributions of this chapter here. As described before, most modern general-purpose, high-quality graph partitioners are based on the multilevel scheme, which constructs a hierarchy of coarse graphs during the coarsening phase.

Graph Coarsening. Early multilevel partitioners, like Chaco [HL95a] and Metis [KK98a], primarily employed coarsening strategies based on contracting graph matchings. While effective for mesh-like graphs due to high matching coverage (often 85–95% [KK95a]), 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 [LaS+15] introduced 2-hop matchings, extending small maximal matchings by further pairing vertices that have some degree of overlap in their neighborhoods until $\geq 75\%$ of vertices are contracted. This technique was subsequently also implemented by other partitioners [Dav+20; Gil+24]. Alternative strategies focus on accelerating coarsening by grouping multiple vertices. These include methods based on cluster contraction [ASS20; Got+21b; Got+24a; MSS14] and pseudo-matchings where vertices can match with multiple neighbors [AK06]. While enabling faster vertex 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 [ANK13; BK96; FdV23]. 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 [BK96].

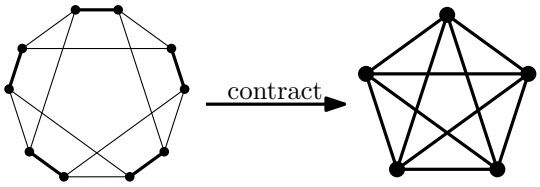

Figure 6.1 Contracting the bolded edges leads to increased density on the coarse graph.

There are several approaches to construct sparsifiers. Spielman and Srivastava [SS11b] 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 [LF06; Lin+15], which uses an analogy to a spreading wildfire. Chen et al. [Che+23] provide a comparative study, suggesting that Forest Fire sampling outperforms uniform sampling for preserving cut-related properties.

6.2 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 $G = (V, E)$. Each level of $\mathcal{H}$ is considered twice, once during coarsening (to construct the next level) and once during refinement (to improve the current partition). Assuming linear time for the coarsening and refinement on each level, the total sequential running time is $\Theta\left(\sum_{i=1}^{\ell} |V_i| + |E_i|\right)$. 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 (in general, any series with a sum of $\mathcal{O}(1)$ works – a geometric series is, however, the most straightforward). 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 already achieves a running time of $\mathcal{O}(|V| + |E| \log |V|)$. However, this is still superlinear in the worst case due to the potential increase in graph density at coarser levels (see Figure 6.1 for an example). To prevent this, we combine this with a similar guarantee for the number of edges to get linear total running time.

The remainder of this section is structured as follows. Since we build our linear-time multilevel graph partitioner on top of the KaMinPar framework introduced in Section 4.3, we first discuss the current bottlenecks in KaMinPar that prevent linear-time guarantees regardless of hierarchy size, and describe how to resolve them. We then propose edge sparsification strategies that guarantee a geometric reduction of the number of coarse edges. Finally, we combine these components into a complete linear-time multilevel partitioning algorithm.

6.2.1 Non-linear Bottlenecks in KaMinPar.

Recall that the default mode of KaMinPar uses deep multilevel graph partitioning for coarsening and initial partitioning. Unfortunately, as discussed in the following, deep multilevel partitioning without additional modifications is unfit to achieve linear running time. The problem is that expensive bipartitioning algorithms might be applied to relatively large graphs due to a combination of two effects. First, the coarsening algorithm applies

no size constraints or early stopping during the clustering (except for the block weight). This can result in a massive size reduction at each coarsening step. Second, recall from Section 4.2.1 that the deep multilevel scheme applies bipartitioning steps over multiple levels if the current level G_i is below a size threshold. In this case, bipartitioning is applied to the next larger level G_{i-1} to improve scalability for large k. While this is not a problem in itself, the fact that there is no bound on the size of G_{i-1} means that bipartitioning might be applied to a graph with $\Theta(n)$ vertices, possibly even to the input graph. Note that we have sidestepped this problem in Section 4.2.5 by introducing simplified assumptions for coarsening. However, in practice, we have observed notable slowdowns due to this effect. Since bipartitioning is crucial for solution quality, it needs to include the more expensive FM refinement algorithm in practice. FM refinement has $\Omega(n \log n)$ running time, thereby adding an $\Omega(n \log n)$ term to the total running time in the worst case.

These problems could be mitigated by changing the coarsening phase of KaMinPar such that initial partitioning only occurs on levels of size $\Theta(k)$. However, in this chapter, we avoid these problems by replacing the deep multilevel scheme with a more traditional approach based on direct k-way partitioning via recursive bipartitioning. This approach closely follows other state-of-the-art multilevel algorithms, such as Mt-KaHIP [ASS20] and Mt-KaHyPar [Got+21a]. More precisely, we coarsen the graph using the same size-constrained label propagation algorithm with 2-hop clustering as described in Section 4.3.1. Similar to Mt-KaHyPar [Got+24a], we set the cluster weight limit to $U = \frac{c(V)}{160k}$ and limit the vertex reduction rate per coarsening level to at most 2.5×. This is done by counting the number of empty clusters C_e during label propagation and aborting the algorithm early if $|V_i|/(|V_i| - C_e) \geq 2.5$. By Theorem 3, this guarantees a geometric reduction of vertices per coarsening level until a size of $\Theta(k)$ vertices is reached. We terminate coarsening at $160k$ vertices. Together with edge sparsification, which we will discuss in the following section, this process guarantees $\Theta(k)$ edges as well. We can then compute an initial k-way partition by recursively bipartitioning the coarsest graph until k blocks are obtained. This is done by using the bipartitioning algorithms described in Section 4.3.3, which require total time $\mathcal{O}(k \log k)$. This is linear under the extremely weak assumption that $k \log k \in \mathcal{O}(n + m)$. We further disable the balancing algorithm (see Section 4.2.4) implemented in KaMinPar since its running time is not linear.

6.2.2 Reducing the Number of Edges via Sparsification

As discussed above, geometric vertex reduction in coarsening strategies can still lead to super-linear total running time due to increasing graph density at coarser levels (e.g., Figure 6.1). To validate this concern, we first show that the issue can arise on important classes of graphs using the example of Erdős-Rényi graphs. To achieve linear time, we must thus prevent this phenomenon. Since common clustering algorithms cannot 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. 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 vertices and edge probability $p_0 \coloneqq c/n_0$ for some constant c. Assume that coarsening halves the number of vertices at each level by contracting pairs of vertices, 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 vertices if any of the four potential edges between the corresponding vertices 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 $i = \alpha \log(n_0)$,

Algorithm 8 Graph Coarsening with Sparsification.

Input : Input graph $G_1 = (V(G_1), E(G_1))$, edge threshold τ_e, density threshold τ_d, minimum reduction factor ρ.
Output : Graph hierarchy $\langle G_1, \dots, G_i \rangle$.

```
1 i ← 1
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⟩
```

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 $n_i = n_0/2^i \geq n_0/2^{\alpha \log n_0} > \sqrt{n_0} \to \infty$ for $n_0 \to \infty$ and $(1-p_i)^4 \approx 1 - 4p_i$ for small p_i, 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 8. 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 3). 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

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

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 5). 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 5). Since our goal is to achieve overall linear time, we only consider linear time sparsification algorithms. Further, sparsification must be fast in practice to attain speedups. Note that Dominik Rosch evaluated additional sparsification approaches in his bachelor's thesis [Ros24]: Effective-resistance sparsification [SS11b], unbiased threshold sampling, and k-Neighbor sampling [SWT16]). However, these are outperformed by the approaches listed below.

(Weighted) Uniform Random Sampling: (W)UR. As a simple baseline, we consider (weighted) uniform random edge sampling. Given the target sample size $\hat{m}$, the unweighted variant selects each edge $e \in E'_{i+1}$ independently with probability

$$p_U \coloneqq \frac{\hat{m}}{|E'_{i+1}|}.$$

Thus, $\mathbb{E}[|E_U|] = p_U|E'_{i+1}| = \hat{m}$, where E_U denotes the sampled edge set. This approach is oblivious to edge weights, although heavier edges have larger influence on the partitioning objective and are thus likely more important. The weighted variant samples edges proportionally to their weight. Specifically, each edge $e \in E'_{i+1}$ is selected independently with probability

$$p_W(e) \coloneqq \min\{1,\ p'_W(e)\} \qquad \text{where} \qquad p'_W(e) \coloneqq \omega(e)\frac{\hat{m}}{\omega(E'_{i+1})},$$

which yields $\mathbb{E}[|E_W|] = \sum_{e \in E'_{i+1}} p_W(e) \leq \hat{m}$, where E_W denotes the sampled edge set. Note that the expected number of sampled edges may be smaller than $\hat{m}$ if there are edges with $p'_W(e) > 1$.

Unweighted uniform sampling yields an unbiased estimator for a *scaled* cut value (see Theorem 5). To make the weighted sampling strategy unbiased, we further rescale the weights of the sampled edges by setting

$$\omega_W(e) \coloneqq \max\{1,\ p'_W(e)\}.$$

Note that we round $\omega_W(e)$ to the nearest integer in our implementation (since it only supports integer edge weights), which introduces a small bias.

▶ **Theorem 5.** *Let $G = (V, E, c, \omega)$ be a graph, and let Π be a k-way partition of V with cut*

$$\texttt{cut}(\Pi) = \sum_{e \in E(\Pi)} \omega(e),$$

where $E(\Pi)$ denotes the set of edges cut by Π. Let $\texttt{cut}_U(\Pi)$ resp. $\texttt{cut}_W(\Pi)$ be the cut values in the sampled graphs obtained by unweighted resp. weighted uniform sampling (using the original weights $\omega(\cdot)$ resp. the rescaled weights $\omega_W(\cdot)$). Then,

$$\mathbb{E}[\texttt{cut}_U(\Pi)] = \frac{\hat{m}}{|E'_{i+1}|} \cdot \texttt{cut}(\Pi) \qquad \textit{and} \qquad \mathbb{E}[\texttt{cut}_W(\Pi)] = \frac{\hat{m}}{\omega(E'_{i+1})} \cdot \texttt{cut}(\Pi).$$

Proof. For unweighted uniform sampling, let E_U denote the set of sampled edges. Then,

$$\mathbb{E}\left[\texttt{cut}_U(\Pi)\right] = \mathbb{E}\left[\sum_{e \in E(\Pi) \cap E_U} \omega(e)\right] = \sum_{e \in E(\Pi)} p_U \cdot \omega(e) = \frac{\hat{m}}{|E'_{i+1}|} \cdot \texttt{cut}(\Pi).$$

For weighted uniform sampling, define

$$L \coloneqq \left\{ e \in E'_{i+1} \mid \omega(e) \leq \frac{\omega(E'_{i+1})}{\hat{m}} \right\} \qquad \text{and} \qquad H \coloneqq E'_{i+1} \setminus L$$

as the sets of lightweight edges L and heavy edges H. Let E_W denote the set of sampled

edges. Then,

$$\begin{aligned}
\mathbb{E}\left[\mathtt{cut}_W(\Pi)\right] &= \mathbb{E}\left[\sum_{e\in E(\Pi)\cap E_W} \omega_W(e)\right] \\
&= \mathbb{E}\left[\sum_{e\in E(\Pi)\cap(E_W\cap L)} \omega_W(e)\right] + \mathbb{E}\left[\sum_{e\in E(\Pi)\cap(E_W\cap H)} \omega_W(e)\right] \\
&= \sum_{e\in E(\Pi)\cap L} \underbrace{p_W(e)}_{=p'_W(e)} \cdot \underbrace{\omega_W(e)}_{=1} + \sum_{e\in E(\Pi)\cap H} \underbrace{p_W(e)}_{=1} \cdot \underbrace{\omega_W(e)}_{=p'_W(e)} \\
&= \sum_{e\in E(\Pi)} p'_W(e) \\
&= \frac{\hat{m}}{\omega(E'_{i+1})}\mathtt{cut}(\Pi).
\end{aligned}$$

◀

Weighted Threshold Sampling: T-Weight. To incorporate edge weights, we consider a weighted threshold sampling strategy. First, we identify the weight threshold $\omega_t \coloneqq \omega(e_t)$ corresponding to the $\hat{m}$-th heaviest edge e_t in G'_{i+1}. This can be done in expected time $\mathcal{O}\big(|E'_{i+1}|\big)$ 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

$$p \coloneqq \frac{\hat{m} - |E'^{>}_{i+1}|}{|E'^{=}_{i+1}|}.$$

(Weighted) Forest Fire Sampling: T-(W)FF. We further include a variation of threshold sampling that uses *Forest Fire* [Lin+15] scores rather than edge weights, as this performed well as a cut-preserving sparsifier in Ref. [Che+23]. We include a brief description for self-containment and extend the algorithm to take edge weights into consideration (Algorithm 9). The algorithm computes edge scores by simulating fires spreading through the graph via multiple traversals starting from random vertices. When visiting a vertex u, the number of neighbors X to be visited is drawn from a geometric distribution parameterized by a tunable parameter p. The standard forest fire algorithm subsequently samples X distinct vertices (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 10). Note that this modification (marked blue in Algorithm 9) is the only difference to Ref. [LF06]. Each edge traversal during this process increments the *burn* score of the edge (line 14). 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).

Algorithm 9 Weighted Forest Fire: graph $G = (V, E)$, burn ratio ν, probability p. The difference from the original Forest Fire [LF06] 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()
5      Q.push(random vertex from V)                              // BFS queue
6      T ← new Set()                                             // Visited vertices
7      while Q ≠ ∅ do
8          u ← Q.pop()
9          while N(u) \ T ≠ ∅ do
10             Sample v from N(u) \ T with prob. ω({u,v}) / Σ_{v∈N(u)\T} ω({u,v})
11             T.insert(v)
12             Q.push(v)
13             S[{u,v}] +=(atomic) 1
14             b +=(atomic) 1
15             break with prob. p
16 return S
```

6.2.3 Putting it Together

In the remainder of this chapter, we refer to our baseline configuration as described in Section 6.2.1 *without additional edge sparsification* as lKaMinPar. When using edge sparsification, we indicate the sparsification method as a suffix, i.e., lKaMinPar-UR for uniform sampling, lKaMinPar-T-Weight for threshold sampling based on edge weights, and lKaMinPar-T-(W)FF for threshold sampling based on (weighted) Forest Fire scores.

Further Optimization. In Algorithm 8, we construct the sparsified graph G_{i+1} from the coarsened graph G'_{i+1}. For UR and T-Weight sampling, however, the sampling decisions depend only on the number of coarse edges in G'_{i+1} and their weights, not on its topology. In practice, we have found it slightly faster to construct the sparsified G_{i+1} by re-running the contraction algorithm and sampling edges on-the-fly. We therefore use this variant for UR and T-Weight, but follow Algorithm 8 for T-(W)FF. Note that this optimization also reduces peak memory usage, since the unsparsified coarse graph can be deallocated before allocating memory for the sparsified graph.

Moving to the Parallel Setting. 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 6.2.2 lend themselves to a rather straightforward parallelization. Combined with the excellent scalability of the coarsening and refinement algorithms of KaMinPar (see Section 5.4 and Section 4.3.4.2) 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.

6.3 Experiments

We have integrated the described algorithms into the KaMinPar partitioner introduced in Chapter 4. We compile all codes using GCC-14.2.0 with flags `-O3 -march=native` and use Intel TBB [TBB] for parallelization.

Setup. We perform our experiments on a single machine equipped with two 32-core Intel Xeon Gold 6530 processors (clocked at 2.1 GHz) and 3 TiB of main memory running Rocky Linux 9.5 (Machine E in Table 2.2). We only use one socket of the machine (i.e., 32 cores) to avoid NUMA effects.

Competitors. In Section 6.3.3, we compare our algorithm against PuLP [SMR14] (v1.1) and Cuttana [HSS24] (commit `ed0c182` in the official GitHub repository[1]). PuLP is a linear-time single-level partitioner which focuses on shared-memory scalability and low memory usage. In Section 4.4.1.1, we have seen that PuLP is slightly slower (1.13×) than KaMinPar. Cuttana is a linear-time streaming partitioner which improves upon the solution quality of previous streaming approaches with a vertex buffering technique. We refer to Section 3.6.2 for more detailed descriptions of PuLP and Cuttana. We use the default settings for PuLP and configure Cuttana using 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, as sparsification is not triggered otherwise. This happens on 17 out of 71 graphs of Benchmark Set A, which is also used in other works on graph partitioning [KGM; MGS] and in Chapters 4 and 5. The affected graphs are mostly real-world k-mer and social graphs, as well as graphs derived from text recompression [Jez15]. In this chapter, we restrict the benchmark set to these graphs since sparsification is not triggered on the remaining graphs, thus leaving the algorithm unchanged. Note that the running time overhead is negligible in this case, and the solution quality remains unchanged. We further include 6 social graphs from the Sparse Matrix Collection [DH11] and generate random graphs: Erdős-Rényi graphs (using KaGen [Fun+18]), as well as Chung-Lu [MH11], Planted Partition, and R-MAT [CZF04] graphs (using NetworKit [Ang+22]). These graphs are inherently non-local and thus especially challenging for linear-time partitioning. Overall, the benchmark set comprises the 39 graphs listed in Table 6.1 with 131 K to 214 M vertices and 511 K to 1.8 G undirected edges. We refer to this set as Benchmark Set D. The graphs deduced from text recompression feature edge weights, while all other graphs are unweighted. Tuning experiments are performed on a subset containing 8 randomly drawn graphs spanning different types. These are highlighted in Table 6.1.

Methodology. As before, 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. Running times and edge cuts are aggregated per instance using the arithmetic mean. When aggregating across multiple instances, we use the geometric mean to ensure that each instance has equal influence.

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

Table 6.1 Graphs of Benchmark Set D (**bold**: tuning subset), characterized by the number of vertices (n), the number of undirected edges (m), and the maximum degree (Δ).

Graph	$n/10^3$	$m/10^3$	Δ	Graph	$n/10^3$	$m/10^3$	Δ
ChungLu [Ang+22]							
chungLuL2e30	134 218	600 116	19 342 891	chungLuS2e30	4 194	19 843	1 344 103
chungLuL2e25	4 194	12 221	2 979 744	chungLuS2e25	131	511	51 402
Compression [Jez15]							
Proteins1GB9	14 538	74 309	660 097	Proteins1GB7	2 826	43 857	373 750
Dna1GB9	3 233	25 285	342 348	Sources1GB9	2 792	12 188	37 881
Sources1GB7	899	7 272	32 091				
Erdő-Rényi [Fun+18]							
er2e29	16 777	536 871	110	er2e28	8 389	268 435	109
er2e25	2 097	33 554	67	er2e24	2 097	16 777	39
Kmer [DH11]							
kmerV1r	214 004	232 705	8	kmerA2a	170 372	179 942	40
kmerP1a	138 896	148 465	40	kmerU1a	64 678	66 394	35
kmerV2a	53 500	57 076	39				
Other [DH11; SS16b]							
baN22	4 194	8 389	11 958	kktPower	2 063	6 482	95
debrG18	1 049	2 097	4	**debrG17**	524	1 049	4
Planted [Ang+22]							
planted2e29-5	4 194	536 871	349	plantedS2e26-5	262	67 114	633
plantedL2e26-5	1 049	67 106	184	planted2e25-7	4 194	33 561	40
R-MAT [HS20; KGB15]							
rmatN24M29	16 777	533 965	2 006 665	rmatN25M28	27 090	268 416	24 179
rmatN24M28	16 777	267 659	1 109 319	rmatN23M27	8 389	133 683	680 044
Social [Les; RA15]							
Friendster	65 608	1 806 067	5 214	Twitter2010	41 652	1 202 513	2 997 487
Sinaweibo	58 656	261 321	278 489	Orkut	3 073	117 185	33 313
LiveJournal	4 037	34 681	14 815	Flickr	1 715	15 555	27 236
wikiTalk	2 394	4 660	100 029	**comDBLP**	317	1 050	343
coAuthorsDBLP	299	978	336				

6.3.1 Parameter Study

We begin our evaluation by tuning the parameters introduced in Section 6.2.2. 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, and only consider the weighted sampling strategies.

The results are shown in Figure 6.2, 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 up to 1.80× for weighted uniform sampling (WUR). T-Weight achieves the highest speedup at $\tau_e = 1/2$, $\tau_d = 1/2$ and $\rho = 3$, while WUR achieves its highest speedup at slightly different parameters ($\tau_e = 1/4$, $\tau_d = 1/2$ and $\rho = 4$). With these parameters, edge cuts increase by 12.7% and 2.0% for T-Weight and WUR, respectively. Surprisingly, more aggressive sparsification (i.e., smaller τ_e, τ_d and ρ)

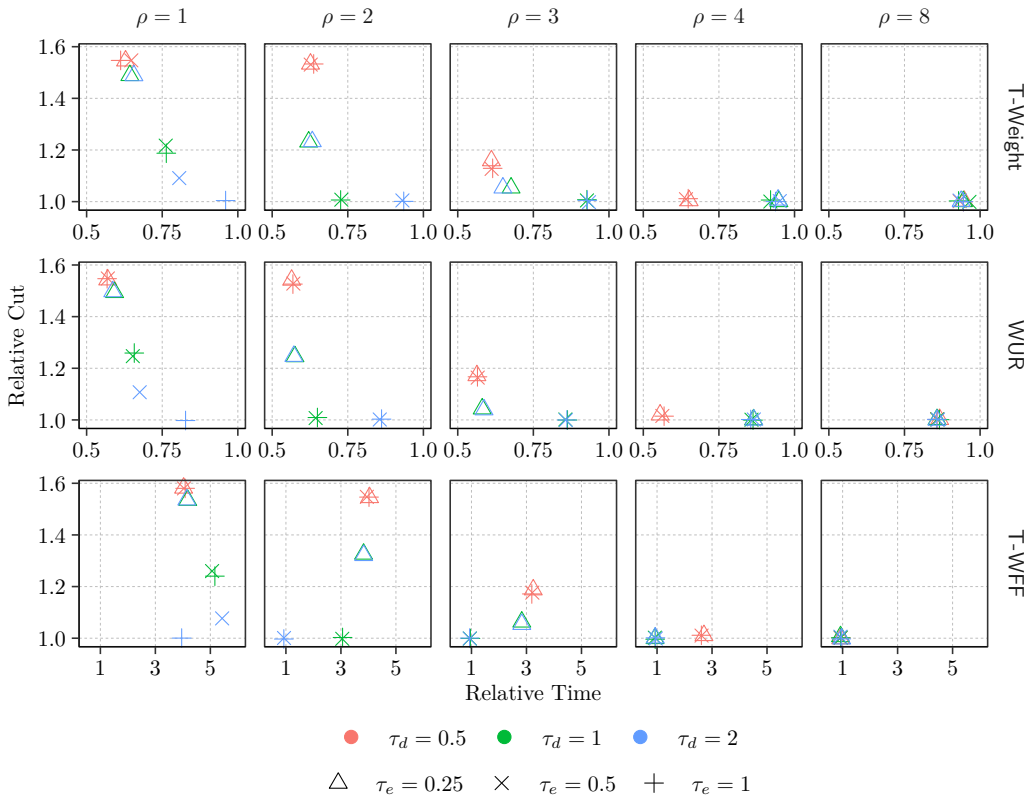

Figure 6.2 Relative cut and running time of LKAMINPAR with weighted threshold sampling (T-Weight), uniform sampling (UR), and threshold sampling via weighted Forest Fire scores (T-WFF) versus the baseline (LKAMINPAR without sparsification) on the tuning benchmark set with $k = 16$.

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 WUR 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 WUR achieve speedups of 1.56× and 1.78×, while increasing edge cuts by 0.9% and 1.5%, respectively.

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.

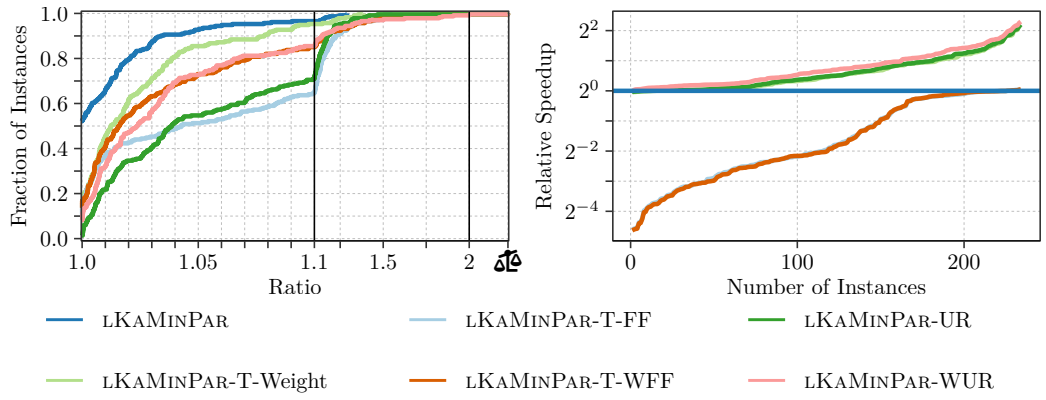

Figure 6.3 Partition quality as performance profile (**left**) and speedup over baseline (no sparsification, **right**) of sparsification algorithms: weighted threshold sampling (T-Weight), (weighted) uniform sampling ((W)UR), and threshold sampling via (weighted) Forest Fire scores (T-(W)FF).

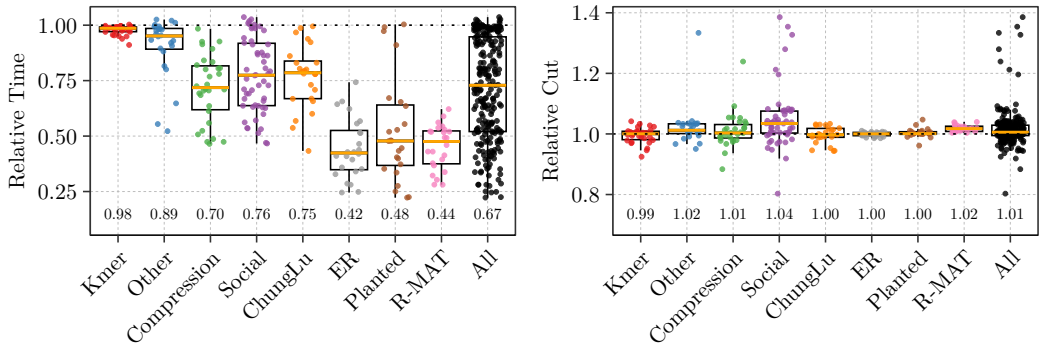

Figure 6.4 Comparison of LKAMINPAR-T-Weight relative to the baseline (LKAMINPAR without sparsification) grouped by graph class (lower is faster resp. higher-quality). **Left:** Relative running times with the geometric mean relative time annotated per class. **Right:** Relative cuts with the geometric mean relative cut annotated per class.

6.3.2 Effects of Sparsification

Next, we evaluate the proposed sparsification techniques on the full benchmark set. The results are shown in Figure 6.3. Sparsification via T-Weight (geometric mean running time 1.43 s) and WUR (1.40 s) achieve speedups of 1.49× resp. 1.69× over the baseline (2.13 s). Looking at Figure 6.3 (right), we can see that WUR is slightly faster than T-Weight and UR. This is likely because WUR samples fewer edges than T-Weight and UR, see the discussion in Section 6.2.2. T-Weight achieves considerably better partition quality (increase in geometric mean edge cut by 1.5%) than WUR (increase by 4.3%), which in turn achieves better partition quality than the unweighted UR (increase by 5.5%). Although slightly slower, T-Weight is thus preferable to (W)UR. T-WFF achieves better solution quality than its unweighted counterpart T-FF, but still performs worse than T-Weight (increase in geometric mean edge cut by 3.9%) while being much slower (7.04 s). We thus exclude UR, T-WFF and T-FF from further experiments and solely focus on T-Weight.

Looking at Figure 6.5, we can see that sparsification reduces the number of edges on coarse graphs considerably. Without sparsification, the graphs on the first hierarchy levels

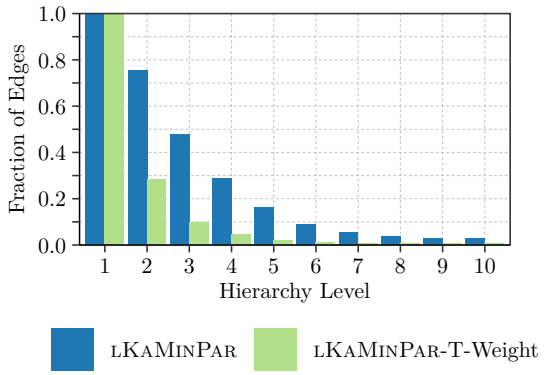

Figure 6.5 Relative geometric mean number of edges per hierarchy level (levels 1-10), comparing no sparsification against T-Weight (weighted threshold sampling) sparsification. Edge counts are relative to the input graphs. The final value is propagated for hierarchies shorter than 10 levels.

(i.e., after the first coarsening step) contain, on average, 75.5% of the edges of the input graphs, but only 39.7% of the vertices. This translates to an increase in density by a factor of 1.9×, which further increases to factors of 3.0× and 31.0× on the subsequent and final levels, respectively. With sparsification, the average edge count reduces to 28.2%, resulting in a density increase of only 0.7× on level 1. The density increase remains moderate, reaching only 0.6× and 3.4× on the subsequent and final levels, respectively.

As can be seen in Figure 6.4 (left), the speedup achieved through T-Weight sparsification varies considerably across graph classes, while the associated cut size increases are moderate even on most real-world graphs (Figure 6.4, right). First, we observe that the geometric mean edge cut increases by only 1% across all instances. Unsurprisingly, randomly generated graphs see almost no increase, except for R-MAT graphs, where cut size increases by 2%. Note that R-MAT graphs are sparsified aggressively (Figure 6.4, left) and are more structured than Erdős-Rényi graphs. The most substantial cut increases occur on social graphs (by 4% on average), although only one graph (out of the nine graphs) shows a cut size increase exceeding 10% (Sinaweibo for all values of k, up to 38.5% for $k = 3$). In the text compression class, only one *instance* shows a cut increase by more than 10% resp. 20%, while k-mer graphs are mostly unaffected by sparsification. Overall, we observe a cut increase exceeding 5% (resp. 10%) on 19.2% (resp. 6.7%) of all real-world graphs. These results indicate that even simple sparsification algorithms are sufficient to maintain the partition quality of multilevel partitioners on real-world input instances. Regarding running times, highly non-local instances such as Erdős-Rényi, Planted Partition and R-MAT graphs benefit the most, with average speedups exceeding 2× and reach up to 4× on some instances. In contrast, real-world text recompression (≈ 1.43×) and social (≈ 1.32×) graphs show more moderate speedups, while k-mer graphs see almost no benefit.

The variation in speedups correlates closely with the reduction in the overall size of the graph hierarchy, measured as the total number of edges across all hierarchy levels. As illustrated in Figure 6.6 (left), instances that exhibit greater size reductions achieve faster relative running times. However, the attained speedups remain smaller than the size reductions for almost all instances, as indicated by the points lying above the $x = y$ diagonal in the plot. As mentioned earlier, this is expected. Sparsification itself introduces computational overhead and degrades partition quality, thereby increasing the workload

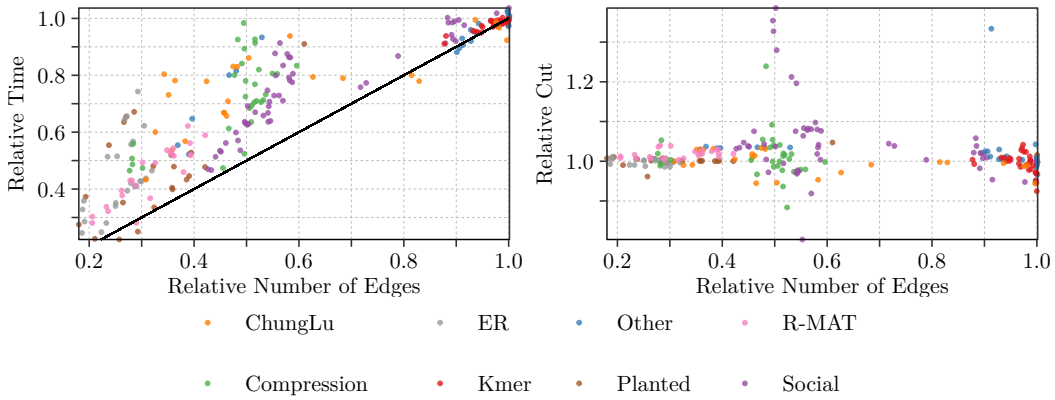

Figure 6.6 Relationship between running time (**left**) and cut size (**right**) of lKaMinPar-T-Weight relative to the lKaMinPar baseline without sparsification and the hierarchy size ratio (number of total edges across all hierarchy levels after sparsification relative to no sparsification). The $x = y$ diagonal is shown as a solid black line for reference. Note the strong correlation for running time (coefficient ≈ 0.893).

of the refinement algorithm. Nonetheless, the observed speedups stem primarily from the reduced time in the coarsening and refinement phases, as detailed in Figure 6.7. 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. The sparsification step itself averages 0.10 s out of 1.62 s when triggered, which happens on 94% of the instances. On the other hand, the relative edge cut increase seems mostly unrelated to the hierarchy reduction factor, as can be seen in Figure 6.6, right. Even for real-world graphs, there is no clear trend.

6.3.3 Comparison against Competing Partitioners

Finally, we compare lKaMinPar-T-Weight against alternative linear-time partitioners: the single-level partitioner PuLP [SMR14] and streaming partitioner Cuttana [HSS24]. We found that Cuttana is much slower than the other algorithms and does not support weighted graphs. To account for this, we limit our benchmark set to the unweighted graphs with $m \leq 2^{26}$ edges (18 out of 39 graphs).

As shown in Figure 6.8 (left), lKaMinPar-T-Weight computes considerably better partitions with average cuts 30% and 66% smaller than those of PuLP and Cuttana, respectively. This is expected given the general advantage of multilevel partitioners over single-level partitioners and the relatively small cut increases that we have observed in Section 6.3.2. Indeed, compared to lKaMinPar *without* sparsification, the edge cuts of lKaMinPar-T-Weight are only slightly larger (cutting 1% more edges on average) and are still within a factor of 1.10 to the best cut found on 88% of all instances (vs. 90% for non-sparsifying lKaMinPar). 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. Interestingly, PuLP computes the best partitions for 21% of the instances, predominantly Erdős-Rényi and Planted Partition graphs. We further note that Cuttana crashes on 39% of the instances. We omit these instances from the pairwise comparisons.

Sparsification reduces the geometric mean running time of lKaMinPar-T-Weight from 0.48 s to 0.37 s. PuLP (0.63 s) is slower than non-sparsifying lKaMinPar on average, but

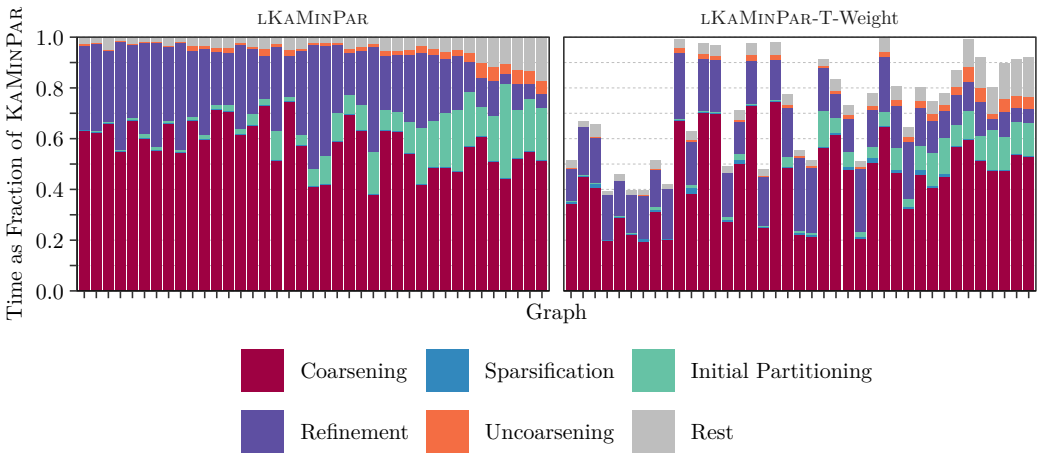

Figure 6.7 Relative running time attribution for lKaMinPar without sparsification (**left**) and with T-Weight sparsification (**right**) using $k = 16$. Graphs are sorted by the total running time of lKaMinPar without sparsification in descending order.

proves faster on 53 instances (resp. 48 instances vs. lKaMinPar-T-Weight) and twice as fast on 23 instances (resp. 2 instances) out of 108 instances. This is likely due to the following reason: recall that PuLP starts partitioning by refining (improving balance and edge cut) a random (possibly imbalanced) partition via label propagation. The convergence speed towards a balanced partition depends on the graph topology, so that PuLP is very fast on graphs that are easy to balance (e.g., Erdős-Rényi graphs) while much slower on graphs such as real-world social networks, where obtaining a good balanced partition is more challenging. Cuttana is 72× slower than PuLP (and thus the other algorithms) making its running time uncompetitive. However, we note that comparing running times fairly is difficult since Cuttana interleaves computation with graph I/O from SSD. For all other algorithms, we ignore I/O times.

6.4 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 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.

For future work, we plan to investigate sparsification as a speedup technique for additional components of multilevel graph partitioning. For example, in KaMinPar, computing a clustering of the input graph is the dominant cost for many instances (compare Figure 4.20 in Section 4.4.2.5). This is because when assigning a vertex to a cluster, the label propagation

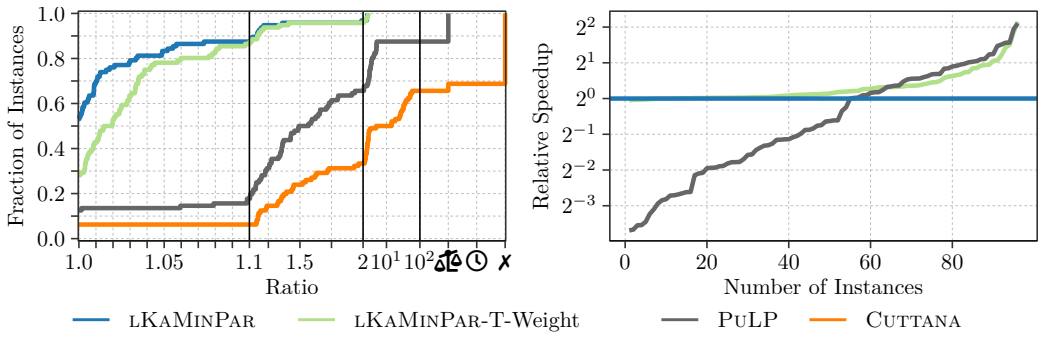

Figure 6.8 Partition quality (**left**) and relative running time (**right**) on the reduced benchmark set and all k values for KaMinPar without and with T-Weight sparsification, PuLP and Cuttana. Speedups are plotted relative to KaMinPar without sparsification. Cuttana is omitted from the speedup plot since it is more than 72× slower than all other algorithms.

algorithm must access the current cluster assignment of its neighbors. If the graph exhibits poor locality, this can cause many cache misses. Sparsification could reduce this cost by only considering a subset of the vertex's neighbors.

7 Distributed Deep Multilevel Graph Partitioning

Parallelizing multilevel graph partitioning has proven challenging over several decades. While shared-memory graph partitioners have recently matured to achieve high quality and reasonable scalability [ASS20; GHS22; Got+21b; LK13], current distributed-memory partitioners [KK99; MSS17; Slo+17] induce a severe quality deterioration and often are not able to consistently achieve feasible (i.e. balanced) partitions. In particular, high quality partitioners do not scale to the number of processing elements (PEs) available in large supercomputers. This situation is exacerbated by the fact that often the number of blocks k should increase linearly in the number of PEs. Previous systems are not able to directly handle large k, running into even bigger problems with achieving feasibility.

In this chapter, we present dKaMinPar which addresses all these issues. Its basis is a distributed-memory adaptation of the deep multilevel graph partitioning concept introduced in Chapter 4, which continues the multilevel approach deep into the initial partitioning phase. This makes the large k case much easier and eliminates a scalability bottleneck due to initial partitioning. Our coarsening and refinement algorithms are based on the label propagation approach previously used in several partitioners [KK99; MSS17; Slo+17], see Section 3.2.2. Label propagation [MSS14; RAK07] greedily moves vertices to other blocks (resp. clusters) when such moves reduce the cut and do not violate the balance constraint (resp. maximum cluster weight). This is simple, fast, effective and robust even for complex networks. We develop a distributed-memory version with improved scalability, e.g., by using improved irregular all-to-all primitives. Perhaps the main algorithmic innovations are new scalable distributed techniques that allow us to maintain the balance constraint. During coarsening, a maximum cluster weight is approximated by unwinding contractions that lead to overweight clusters. During uncoarsening, block weight constraints are achieved by finding, selecting and applying globally "best" block moves. Additionally, we implement a distributed-memory version of the Jet [Gil+24] algorithm, which has previously been shown to achieve similar quality as FM refinement [Gil+24], to improve the solution quality of dKaMinPar, shrinking the gap between the highest-quality distributed-memory and shared-memory partitioners considerably. Lastly, we integrate the compressed graph representation described in Chapter 5 into dKaMinPar to obtain xTeraPart, the distributed-memory version of TeraPart. With this, we can partition tera-scale randomly generated graphs with up to 2^{40} edges on only 8 compute nodes, equipped with 2 TiB of main memory in total.

Contributions. We present a scalable MPI-based distributed-memory implementation of the deep multilevel graph partitioning scheme. In its basic version, we achieve simplicity by using label propagation for both coarsening and refinement. We improve solution quality by adapting the Jet [Gil+24] algorithm to the distributed-memory model, thereby considerably closing the quality gap between high-quality shared-memory and distributed-memory partitioners. We perform an extensive evaluation on both large real-world networks and huge synthetic networks from 3 input families to showcase scalability up to (at least) 8 192 machine cores and 16 *trillion* undirected graph edges. Our algorithm works both for

complex networks and for a large number of blocks, settings in which previous systems often fail.

References and Contributors. This chapter is mostly based on the conference publication Ref. [SS23b] and the brief announcement Ref. [SS24], both papers published together with Peter Sanders. xTeraPart is based on the conference publication Ref. [Sal+25] published together with Daniel Salwasser, Lars Gottesbüren and Peter Sanders. The text was mostly written by the author of this dissertation, with editing from Peter Sanders. All parts due to Ref. [SS23b; SS24] were implemented by the author of this dissertation. The graph compression scheme described in Section 5.3.1 was implemented by Daniel Salwasser. The experimental evaluation presented in this chapter was performed by the author of this dissertation.

Structure. The remainder of this chapter is organized as follows. Since we have already introduced the relevant related work for distributed graph partitioning throughout Chapter 3, Section 7.1 directly describes the distributed-memory adaptation of the Deep MGP scheme introduced in Chapter 4. Section 7.2 subsequently describes the building blocks of our distributed graph partitioner based on Deep MGP, which we call dKaMinPar resp. xTeraPart depending on its configuration. We then present the experimental evaluation in Section 7.3 and conclude the chapter in Section 7.4.

7.1 Distributed Deep Multilevel Graph Partitioning

In this section, we adapt the Deep MGP scheme to the distributed setting. As in Chapter 4, we first describe the partition scheme itself. Afterwards, we present the distributed-memory graph partitioner dKaMinPar based on the distributed Deep MGP scheme by describing the components used for *coarsening*, *initial partitioning*, *refinement* and *balancing* in Section 7.2.

Recall that Deep MGP follows the traditional multilevel graph partitioning scheme, but coarsens the graph down to a small size independent of k. After partitioning the coarsest graph into two blocks, it maintains the invariant that each block of the current partition contains roughly C vertices, for some input parameter C (the *contraction limit*). This is maintained through recursive bipartitioning on each level until there are k blocks.

Overview. To adapt Deep MGP to the distributed setting, we consequently use distributed algorithms for graph clustering and contraction, and partition refinement (see Section 7.2). Initially, the input graph is distributed over all p PEs. During coarsening, we maintain the invariant that a graph distributed across p PEs contains at least pC vertices: when violated, we split PEs into groups that work independently on copies of the coarse graph. During uncoarsening, we maintain the invariant that a (coarse) graph with n' vertices is partitioned into roughly $\min\{k, n'/C\}$ blocks. Figure 7.1 illustrates distributed Deep MGP by partitioning an input graph into $k = 8$ blocks using $p = 4$ PEs, while Algorithm 10 provides a more formal description. In the following, we assume p and k to be powers of 2 to keep the description simple. General values of k are supported as described in Section 4.2.3, while general values of p imply that some PEs may need to process one additional block compared to other PEs.

Distributed Deep MGP. We repeatedly coarsen the input graph until only $2C$ vertices are left, building a hierarchy $G =: G_1, G_2, \ldots, G_\ell$ of successively coarse graphs. On each level,

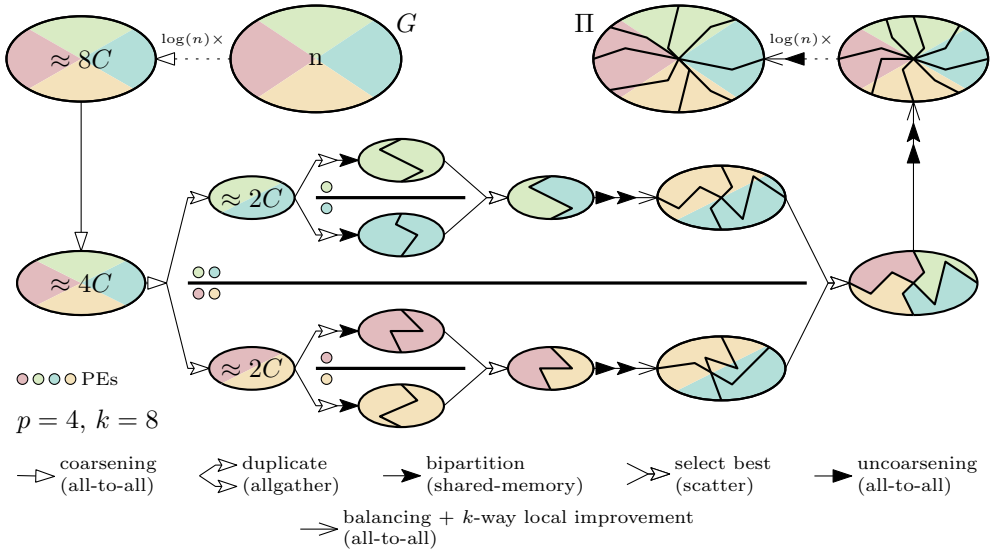

Figure 7.1 Distributed Deep MGP using $p = 4$ PEs to partition the input graph G into $k = 8$ blocks. Unpartitioned graphs are labeled with their number of vertices. During initial partitioning and uncoarsening, blocks are recursively bipartitioned into smaller blocks. Bold horizontal lines illustrate PE groups working independently.

we check whether the current graph G_i contains at least pC vertices. If this is not the case, we split the p PEs into two subgroups $1..\frac{p}{2}$ and $\frac{p}{2}+1..p$, and distribute one full copy of G_i over each group. This process improves scalability on coarse levels, and, with randomized coarsening algorithms, leads to more diversification. The subgroups continue this procedure recursively until roughly $2C$ vertices are left in the coarsest graph (distributed over 2 PEs). Finally, each pair of PEs obtains a full copy of the coarsest graph in-memory and uses a shared-memory parallel (or possibly sequential) partitioner to compute a bipartition of the graph. In Figure 7.1, we copy and redistribute the graph once at $2C$ vertices (going from 4 PEs to 2 PEs), and then create one copy for each PE.

During the uncoarsening phase, the best partition (within each PE group) is selected and projected onto $G_{\ell-1}$ by assigning fine vertices to the blocks of their corresponding coarse vertices. From here, we maintain two invariants during uncoarsening:

1. The current partition is feasible, which we ensure using the distributed balancing algorithm described in Section 7.2.3, and
2. each block contains roughly C vertices (until there are k blocks).

To maintain the latter invariant, assume that the current graph (distributed across p PEs) with $n_i := |V(G_i)|$ vertices is partitioned into $k' < k$ blocks. In this case, we assign k'/p blocks to each individual PE, and create full copies of the respective block-induced subgraphs on the assigned PEs. This can be achieved using all-to-all communication between PEs. These subgraphs are then partitioned into $\text{ceil}_2(n_i/C)/k'$ blocks using a shared-memory parallel (or sequential) partitioner. Afterwards, we update the partition of the distributed graph using all-to-all communication and subsequently improve it using a local search refinement algorithm. We repeat this process until we obtain a k-way partition. Note that if $k > n_1/C$, the partition computed on the finest graph does not have enough blocks. In this case, we distribute and partition the block-induced subgraphs once more to obtain the remaining blocks.

Algorithm 10 `DeepMGP`(G, k, p): Deep multilevel with k blocks on p PEs.

Input : $G = (V, E)$, number of blocks k, number of PEs p, constant C
Output : k-way partition Π of G

```
if |V(G)| > 2 · C then                                   // Deep coarsening
    if |V(G)| < C · p then          // Duplicate graph if too small for p PEs
        c := p/ceil_2(|V(G)|/C)                          // Number of copies
        G := Replicate(G, c)        // Replicate graph c times and group PEs
        p := p/c                                  // Number of PEs per group
    // Standard multilevel graph partitioning:
    G_c := Coarsen(G)
    Π_c := DeepMGP(G_c, k, p)
    Π := Project(G, Π_c)
    Π := BalanceAndRefine(G, Π)
else                                                     // Base case
    Π := {V}                                  // Trivial 1-way partition
k' := min{k, ceil_2(|V|/C)}
if |Π| < k' then                                         // Extend partition
    G'_1, ..., G'_{|Π|/p} := DistributeBlocks(G, Π)      // G'_i s are local graphs
    // Partition each local graph into min{k'/|Π|, K} blocks:
    for i := 1 to |Π|/p do
        Π'_i := KaMinPar(G'_i, k'/|Π|)
    // Combine local partitions Π'_i of G'_i to global partition Π of G:
    Π := CollectPartitions(Π'_1, ..., Π'_{|Π|/p})
    Π := BalanceAndRefine(G, Π)
return Π
```

7.2 Implementation: dKaMinPar

The previous section introduced Deep MGP as a generic partitioning scheme for distributed-memory systems, but left its concrete building blocks unspecified. We now instantiate distributed Deep MGP by detailing the distributed graph representation, as well as the components used for coarsening, initial partitioning, and refinement. The resulting implementation is dKaMinPar.

7.2.1 Distributed Graph Representation

We only store a subgraph of the global input graph $G = (V, E)$ on each processing element. To construct these subgraphs, we partition the vertices of G using a greedy one-dimensional approach that aims to balance the number of edges assigned to each processor. For each processor i ($0 \leq i < p$), we compute the threshold values

$$T_i = i\frac{m}{p} \quad \text{and} \quad T_{i+1} = (i+1)\frac{m}{p}.$$

Each processor then performs a binary search on the CSR index array $\mathcal{P}$ (stored on disk) to locate the first vertex v_i such that $\mathcal{P}[v_i] \geq T_i$ and the first vertex v_{i+1} such that $\mathcal{P}[v_{i+1}] \geq T_{i+1}$. The consecutive vertex range $v_i..v_{i+1} - 1$ is assigned to processor i. Consequently, each

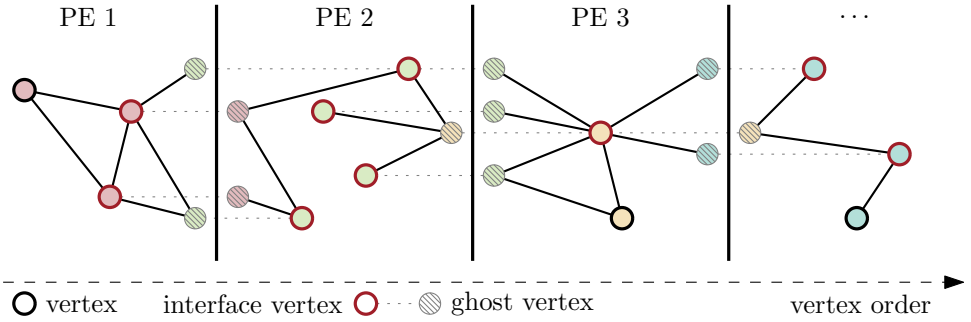

Figure 7.2 Distributed graph representation. Each processor is assigned a subgraph of the input graph. Interface vertices are replicated as ghost vertices on other processors.

processor receives a subgraph with at most $\frac{m}{p} + \Delta$ edges, where Δ denotes the maximum vertex degree in the graph. We represent every undirected edge $\{u, v\}$ by two directed edges, (u, v) and (v, u), which are stored on the processors that own the respective source vertices. Vertices adjacent to vertices owned by other processors are denoted *interface vertices* and are replicated as *ghost vertices* (i.e., they are stored without outgoing edges) on those processors. An illustration of this distributed graph representation is shown in Figure 7.2.

The PE-local subgraphs are stored in CSR representation (see Section 2.1). Let n_l denote the number of vertices assigned to a PE, and let n_g denote the number of ghost vertices replicated on that PE. Each PE relabels its owned vertices to IDs in $[n_l]$, and its ghost vertices to consecutive IDs starting at n_l. We additionally maintain an array of length n_g that maps local IDs of ghost vertices to global vertex IDs, and a hash map for the other direction. Finally, each PE stores an array $\mathcal{N}$ of length $p+1$ containing prefix sums over the numbers of owned vertices, so that PE $0 \leq i < p$ owns $\mathcal{N}[i+1] - \mathcal{N}[i]$ vertices. Hence, given the global ID of a ghost vertex, we can identify its owning PE via a binary search in $\mathcal{N}$ in time $\mathcal{O}(\log(p))$.

With this graph representation, a common communication primitive is to exchange messages between interface vertices and their ghost replications on other PEs (i.e., communication along the edges cut by the 1D partition). Depending on the graph structure, this communication is often sparse and irregular. In the following, we refer to this primitive as *irregular all-to-all communication*.

7.2.2 Coarsening

We start by outlining the coarsening phase of dKaMinPar. To this end, we first describe the distributed parallelization of graph clustering via batched-synchronous size-constrained label propagation. Afterwards, we outline the graph contraction procedure.

Clustering via Batched Size-Constrained Label Propagation. Recall that size-constrained label propagation initializes each vertex in its own cluster. In subsequent iterations (we use 3 iterations), larger clusters are formed by moving vertices to adjacent clusters, so that intra-cluster edge weights are maximized without violating some maximum cluster weight U.

As described in Section 4.3.1, our shared-memory implementation KaMinPar uses *asynchronous* size-constrained label propagation for clustering the graph during the coarsening phase. In this asynchronous approach, vertices processed later in an iteration are influenced by the updated cluster assignments of earlier vertices. This contrasts with synchronous

approaches, where updates occur simultaneously for all vertices. The former is typically preferred for faster convergence while avoiding oscillation [RAK07]. However, implementing asynchronous label propagation in the distributed setting seems challenging, as asynchronous vertex moves between clusters require frequent fine-grained communication (i.e., many small messages), which induces expensive message passing startup overheads and limits performance.

To balance quality and convergence with communication efficiency, we adopt the hybrid batch-synchronous parallelization strategy of ParHIP [MSS17]. This strategy chunks vertices into a fixed number of batches, which are processed synchronously by all PEs, while vertex moves between clusters are communicated collectively between batches. Note that the clustering on a single PE during a single batch is still computed asynchronously. We further adapt the iteration order and randomization strategies from KaMinPar (see Section 4.3.1), and also use the same maximum cluster weight limit $U := \varepsilon \left\lceil \frac{c(V)}{k'} \right\rceil$, where $k' = \min\{k, |V'|/C\}$ denotes the number of blocks that Deep MGP will produce for the (coarse) graph $G' = (V', E')$. Since clusters may span multiple PEs, we achieve *soft* adherence to U by tracking the global weight of each cluster via additional communication steps between batches. Note that we do not strictly enforce U, as discussed below. We now provide more details on these aspects.

Iteration Order and Randomization. As noted in Ref. [ASS20; MSS14], the solution quality of label propagation is improved when iterating over vertices in increased degree order. Since this is not cache efficient and lacks diversification by randomization, we sort the vertices into exponentially space degree buckets, i.e., bucket b contains all vertices with degree $2^b \leq d < 2^{b+1}$, and rearrange the input graph accordingly. This happens locally on each PE. Then, during label propagation, we split buckets into small chunks and randomize traversal on an inter-chunk and intra-chunk level analogous to the randomization of the matching algorithm used by Metis [KK98a].

Batched Communication. To communicate the current cluster assignment of interface vertices, we follow ParHIP and split each iteration into $b = \max\{\alpha, \beta/p\}$ batches, where $\alpha = 8$ and $\beta = 128$ in our experiments. Each batch processes $\lceil |V|/b \rceil$ vertices. After each batch, the updated labels of interface vertices are sent to PEs containing corresponding ghost vertices using an irregular all-to-all operation.

Handling the Maximum Cluster Weight U. Enforcing the maximum cluster weight U in the distributed setting is more challenging than in the shared-memory setting, as clusters can span multiple PEs. ParHIP addresses this issue by relaxing the constraint and enforcing it only locally on each PE (see Section 3.2.2 for more details). However, this relaxation permits global clusters to reach a weight of up to pU in the worst case, and consequently, can lead to coarse vertices with weight up to pU. Such heavy weights can make it impossible for initial partitioning to find balanced solutions. Moreover, if a rebalancing algorithm is used during uncoarsening, coarse vertex weights that scale with p can limit scalability if the workload required for rebalancing increases proportionately. We observed this specific scalability bottleneck during early experiments with dKaMinPar on complex networks such as the Twitter2010 graph.

To mitigate this issue, we implement a global cluster weight tracking mechanism and fix violations in a subsequent rollback step, which we describe below. Each PE maintains a hash table $\mathcal{W}$ storing the global weight $\mathcal{W}[C]$ for every cluster C that contains at least one of its

owned vertices. During batch processing, PEs enforce U locally using the information in $\mathcal{W}$. To be more precise, a vertex v is allowed to move from its current cluster C_s to cluster C_t only if $\mathcal{W}[C_t] + c(v) \leq U$. If the move occurs, the PE records the cluster weight deltas in a separate hash table $\mathcal{W}_\delta$: $\mathcal{W}_\delta[C_s] \mathrel{-}= c(v)$ and $\mathcal{W}_\delta[C_t] \mathrel{+}= c(v)$. After each batch, PEs exchange these deltas. Specifically, PEs send updates to the *owners* of the affected clusters (the PE owning the cluster's initial vertex). Note that the communication volume of this step is small as long as clusters remain mostly local. The owner PEs accumulate the changes and distribute the updated global weights, denoted as $\mathcal{W}'$, back to the sending PEs. Overall, this process involves two all-to-all operations.

If a cluster C violates the constraint (i.e., $\mathcal{W}'[C] > U$), the contributing PEs must revert moves to reduce the weight. To this end, each PE removes vertices from C proportional to its contribution to the cluster's weight increase beyond U, i.e., it removes vertices with total weight at least

$$\frac{\mathcal{W}_\delta[C]}{\mathcal{W}'[C] - \mathcal{W}[C]} \cdot (\mathcal{W}'[C] - U).$$

These vertices are picked arbitrarily and moved back to their prior cluster C_s. They can then be moved to other clusters in subsequent rounds. Note that this rollback may overload C_s if other vertices were moved to C_s during the same batch. However, the weight of C_s is bounded by $I \cdot U$, where I denotes the number of label propagation iterations, as at most weight U is added to C_s during each rollback. Since this bound is independent of p, it is more tolerable than the pU bound of ParHIP. Finally, $\mathcal{W}$ is set to $\mathcal{W}'$ and the algorithm proceeds with the next batch.

Graph Contraction. After clustering the graph, we contract the clusters to build the next graph in the hierarchy. Contracting a clustering consisting of n_C clusters and constructing the corresponding coarse graph works as follows. First, the clustering algorithm described above assigns a cluster ID to each vertex, which corresponds to some vertex ID in the distributed graph. We say that a cluster is *owned* by the PE owning the corresponding vertex. After contracting the local subgraphs (i.e., deduplicating edges between clusters and accumulating vertex- and edge weights), we map clusters to PEs such that each PE gets roughly the same number of coarse vertices. We assign $\leq \delta \cdot n_C/P$ clusters owned by each PE to the same PE (in our experiments, $\delta = 1.1$). If a PE owns more clusters, we redistribute the remaining clusters to PEs that have the smallest number of clusters assigned to them. Afterwards, each PE sends outgoing edges of coarse vertices to the respective PE using an all-to-all operation, then builds the coarse graph by deduplicating edges and accumulating vertex- and edge weights.

7.2.3 Balancing

Recall that balance constraint violations during Deep MGP can occur after initial partitioning or after projecting a coarse graph partition onto a finer level of the graph hierarchy [Got+21b]. Since these balance constraint violations are bounded by the weight of the heaviest vertex, we design the following balancing algorithm based on the assumption that only few vertex moves are necessary to restore balance, and that it is thus feasible to invest a moderate amount of work per vertex move. Nonetheless, we further present a highly parallel fallback mode, which is triggered whenever convergence is too slow. An example illustrating both parts of the algorithm is shown in Figure 7.3.

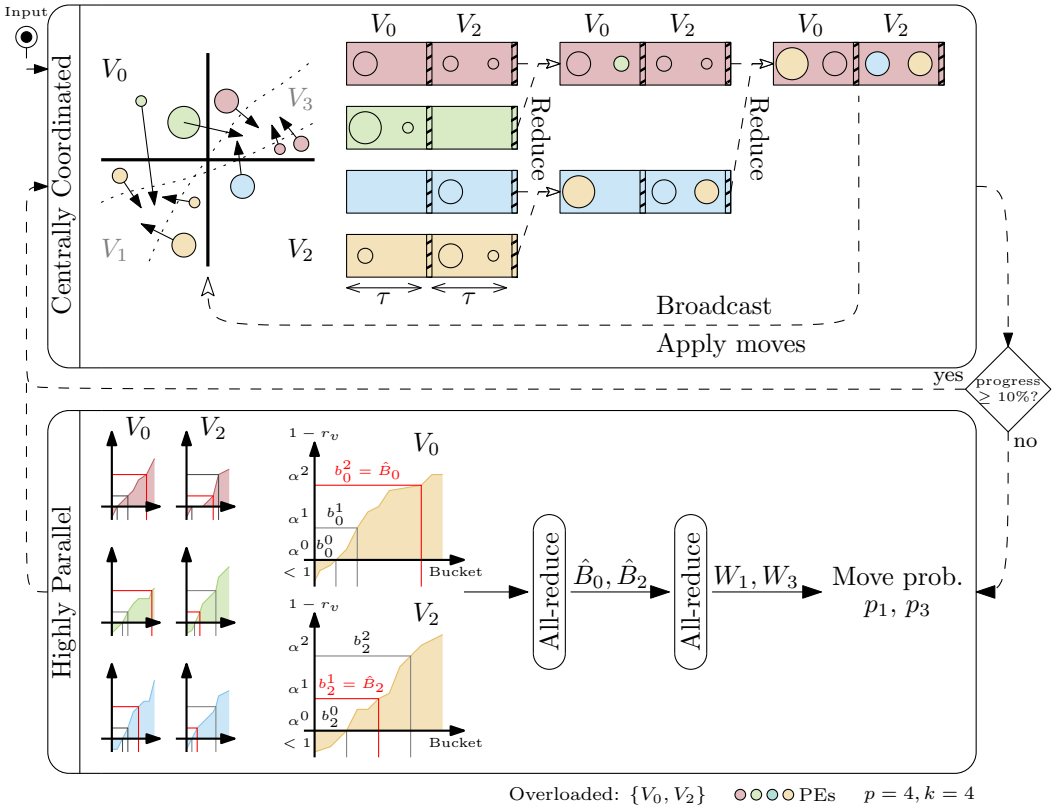

Figure 7.3 Illustration of the distributed rebalancing algorithm: $p = 4$ PEs (colored) rebalance a $k = 4$-way partition with two overloaded blocks, V_0 and V_2. **Top:** Centrally coordinated rebalancing with $\tau = 2$ vertices per block and epoch. Arrows indicate proposed moves, with vertex size proportional to relative gain. PEs collect their best moves in local priority queues, which are subsequently merged via a binary reduction tree. **Bottom:** Highly parallel fallback approach. PEs sort vertices into exponentially spaced buckets based on relative gain. Moves are then performed independently and probabilistically on each PE, such that underloaded blocks do not become overloaded in expectation. The process is also described in Algorithm 11.

Centrally Coordinated Balancing. For each overloaded block V_o, each PE maintains a local priority queue P_o of vertices $v \in V_o$ keyed by their relative gain r_v. Recall that this is $r_v = g \cdot c(v)$ if $g \geq 0$ and $r_v = g/c(v)$ otherwise, where g denotes the highest possible gain when moving v to another block without overloading it. This prioritization prefers moving few heavy vertices over many light vertices. To keep the priority queues small, we maintain the invariant that P_o stores just enough vertices to remove all excess weight $c(V_o) - L_{\max}$ from block V_o. To this end, each PE initializes P_o by iterating over all owned vertices $v \in V_o$, inserting v into P_o if $c(P_o) \leq c(V_o) - L_{\max}$. Otherwise, v is only inserted if it can replace another vertex with worse relative gain.

To choose which vertices to move, we then use a global reduction tree as illustrated in Figure 7.3 (upper half). Using the local PQs, each PE builds a sorted list per overloaded block containing up to τ (a tuning parameter) vertices. At each level of the reduction tree, the sorted lists are then merged and truncated to the prefix that is sufficient to remove all excess weight, but no more than τ vertices. Finally, the root PE selects a subset of the proposed vertices such that no other block becomes overloaded and broadcasts its decision

Algorithm 11 `Rebalance`(G, Π)

Input : $G = (V, E, c, \omega)$, partition Π

// 1. Compute the number of required buckets
$L \coloneqq \lceil \log_\alpha(\text{maximum relative gain}) \rceil + 1$
// 2. Sort vertices into exponentially spaced buckets
forall *overloaded blocks* $V_o \in \Pi$ **do**
  $b_o^\ell \coloneqq \emptyset$ for all $0 \leq \ell \leq L$
  forall $v \in V_o$ **do**
    $r_v \coloneqq \max_{\{V_u \in \Pi \mid c(V_u)+c(v) \leq L_{\max}\}} \texttt{RelGain}(v, V_u)$
    $\ell \coloneqq \begin{cases} 1 + \lceil \log_\alpha(1 - r_v) \rceil & \text{if } r_v < 0 \\ 0 & \text{otherwise} \end{cases}$
    $b_o^\ell \cup= v$
// 3. Compute cut-offs, total weight to each block
forall *overloaded block* $V_o \in \Pi$ **do**
  $c(B_o^\ell) \coloneqq \sum_{i=1}^{p} c(b_o^\ell @ i)$ for all $0 \leq \ell \leq L$ // $b_o^\ell @ i$ denotes b_o^ℓ on PE i
  $\hat{B}_o \coloneqq \min\left\{\ell \mid \sum_{i<\ell} c(B_o^i) \geq c(V_o) - L_{\max}\right\}$
forall *underloaded block* $V_u \in \Pi$ **do**
  $W_u \coloneqq \sum_{V_o \in \Pi} \sum_{i=1}^{p} c(\{v \in b_o^\ell @ i \mid \ell < \hat{B}_o, v \text{ moves to } V_u\})$
  $p_u \coloneqq (L_{\max} - c(V_u))/W_u$
// 4. Move vertices
forall *overloaded block* $V_o \in \Pi$, $\ell < \hat{B}_o$, $v \in b_o^\ell$ **do**
  $V_u \coloneqq \arg\min_{\{V_u \in \Pi \mid c(V_u)+c(v) \leq L_{\max}\}} \texttt{RelGain}(v, V_u)$
  Move v from V_o to V_u with prob. p_u
return Π

to all PEs. Using this information, PEs update the current partition state, remove moved vertices from their PQs and update the relative gains of neighboring vertices. We repeat this process until the partition is balanced.

Highly-Parallel Balancing. If the partition is difficult to balance, the algorithm described above might introduce a sequential bottleneck due to the global reduction tree and the centralized selection of moves. To make rebalancing scalable even in this case, we propose a second balancing algorithm as fallback that moves many vertices concurrently, while ignoring move dependencies. We illustrated this algorithm in Figure 7.3 (lower half) and formally describe it in Algorithm 11.

As before, we consider vertices $v \in V_o$ in overloaded blocks $V_o \in \Pi$. Each PE assigns its owned vertices $v \in V_o$ to exponentially spaced buckets b_o^ℓ, where

$$\ell \coloneqq \begin{cases} 1 + \lceil \log_\alpha(1 - r_v) \rceil & \text{if } r_v < 0 \\ 0 & \text{otherwise.} \end{cases}$$

The parameter α controls the bucket width. In preliminary experiments, smaller α improved partition quality while having only a minor effect on running time. We therefore set $\alpha = 1.1$.

After constructing the PE-local buckets, we compute global bucket weights $c(B_o^\ell) \coloneqq \sum_{i=1}^{p} c(b_o^\ell @ i)$ using an all-reduce operation, where $b_o^\ell @ i$ refers to the value of variable b_o^ℓ on

PE i. The *cut-off buckets*

$$\hat{B}_o := \min \left\{ \ell \mid \sum_{i<\ell} c(B_o^i) \geq c(V_o) - L_{\max} \right\}$$

subsequently contain the lowest-rated vertices that must be moved to remove all excess weight from overloaded blocks. Moving all vertices in the buckets up to (and including) the cut-off buckets could introduce new overloaded blocks. Thus, we use the following probabilistic approach that maintains balance (of non-overloaded blocks) in expectation instead. Let W_u be the total weight of all vertices that should be moved to the non-overloaded target block V_u (across all PEs). Each PE moves vertices to V_u independently with probability

$$p_u := \frac{L_{\max} - c(V_u)}{W_u}.$$

If $p_u \geq 1$, we always perform the moves.

In the example shown in Figure 7.3 (bottom half), there are two overloaded blocks V_0 and V_2, and two non-overloaded blocks V_1 and V_3. The cut-off buckets for blocks V_0 and V_2 are $\hat{B}_0 = 2$ and $\hat{B}_2 = 1$ (marked in red), which are used to determine the total weights W_1 and W_3 (and subsequently, the move probabilities p_1 and p_3) of vertices that should be moved to blocks V_1 and V_3.

Putting it Together. As indicated in Figure 7.3, we start the balancing process by performing one round of the centrally coordinated balancing algorithm. If this reduces the total overload of the partition (i.e., sum of all excess block weights) by at least 10%, we continue with further rounds of this algorithm until the partition is balanced or the relative overload reduction between rounds falls below the threshold. In that case, we perform one round of the highly-parallel balancing algorithm before returning to the centrally coordinated approach.

7.2.4 Refinement

In this section, we describe two refinement algorithms for distributed graph partitioning. Label propagation refinement is fast, but only offers limited improvements in partition quality. To complement this, we introduce a distributed adaptation of the unconstrained JET refinement algorithm [Gil+24]. Although this algorithm is more computationally expensive, it yields considerable improvements in partition quality, as demonstrated in Section 7.3.

Label Propagation Refinement. We use size-constrained label propagation to improve the current partition. Unlike in the coarsening phase, vertices are initially assigned to clusters representing the current partition blocks, with weight constraints set to the maximum allowed block weight, $L_{\max}$. We adopt the same traversal order (exponentially spaced degree buckets with intra-bucket randomization) and the batched parallelization scheme as in coarsening (see Section 7.2.2). However, we increase the number of iterations from 3 to 5, which we found to slightly improve quality with negligible running time overhead.

Similarly to the clustering phase, enforcing the maximum block weight $L_{\max}$ is more challenging in the distributed setting than in the shared-memory setting, where this can be achieved through atomic compare-and-swap operations. Strict enforcement would require fine-grained synchronization to prevent concurrent moves from multiple PEs overloading a specific block. Fortunately, we can treat $L_{\max}$ as a soft constraint during label propagation and rely on the balancing algorithm introduced in Section 7.2.3 to repair violations subsequently.

Thus, our process works as follows. Each PE stores two arrays $\mathcal{W}$ and $\mathcal{D}$ of size k. We initialize $\mathcal{W}[i] = c(V_i)$ and $\mathcal{D}[i] = 0$ for all $V_i \in \Pi$. The values in $\mathcal{W}$ store the global weights for each block and are only updated after processing a complete batch (see below). The other array $\mathcal{D}$ stores changes to the global block weights due to local vertex moves in the current batch. More precisely, when moving vertex v from block V_s to block V_t, we update $\mathcal{D}[s] \mathrel{-}= c(v)$ and $\mathcal{D}[t] \mathrel{+}= c(v)$. To decide whether V_t is a viable target block for vertex v, the PE evaluates the condition $\mathcal{W}[t] + \mathcal{D}[t] + c(v) \leq L_{\max}$. After processing all vertices of a batch, the PEs perform an all-reduce operation over $\mathcal{D}$ to update the global block weights stored in $\mathcal{W}$, before setting $\mathcal{D}[\cdot] = 0$. Additionally, we perform an irregular all-to-all operation to synchronize the block assignments of interface vertices and their ghost vertex replicas.

Note that this approach is insufficient to strictly enforce the maximum block weight during refinement. For instance, consider a block V_i with weight $c(V_i) < L_{\max}$. During a single batch, all p PEs might move up to $L_{\max} - c(V_i)$ weight to V_i, thus increasing its weight to $L_{\max} + (p-1)(L_{\max} - c(V_i)) > L_{\max}$. As stated above, we ignore these violations and rely on the balancing algorithm introduced in Section 7.2.3 to fix them afterwards.

Distributed Jet Refinement. To achieve higher partition quality, we also implement the Jet [Gil+24] refinement algorithm in the distributed-memory model. This requires only minor modifications to the algorithm, but we restate the entire algorithm here in order to be self-contained. First, recall from Section 3.4.4 that each PE builds a set of local move candidates M by visiting its owned vertices, including a vertex v of block V_i in M if $g(v) \coloneqq \max_{j \neq i} \operatorname{conn}(v, V_j) - \operatorname{conn}(v, V_i) \geq -\lfloor \tau \cdot \operatorname{conn}(v, V_i) \rfloor$. The temperature τ controls the extent of move candidates with negative gain (i.e., that would increase the partition cut) included in M. Note that $\operatorname{conn}(v, \cdot)$ can be computed without communication since it only depends on v's neighborhood. Next, each interface vertex $v \in M$ sends its corresponding $g(v)$ value to its ghost replicates. With this information, PEs independently re-evaluate their move candidates, removing all vertices v that would increase the partition cut assuming that any neighbor u with $g(u) > g(v)$ is moved before v. The remaining vertices are moved to their target blocks, locked for the next round, and the block assignment of ghost replicates is updated accordingly.

The original Jet algorithm runs this algorithm with a fixed temperature τ, depending on the coarsening level. Since we have observed that Jet is very sensitive to the temperature used in the construction of M, we perform multiple rounds ($t = 4$ rounds) of the algorithm to achieve more robust performance. During the i-th round, we set the temperature to $\tau_i = \tau_0 + \frac{i}{t}(\tau_1 - \tau_0)$, where $\tau_0 = 0.75$ and $\tau_1 = 0.25$. Following Ref. [Gil+24], a single round consists of repeating the described Jet refinement and rebalancing steps until 12 consecutive repetitions did not improve the partition.

7.2.5 Graph Compression and Benchmark Graph Generation

In Section 5.3.1, we described a compressed graph representation based on gap- and interval-encodings, which we integrated into KaMinPar to make it more memory-efficient. To process even larger graphs in the distributed-memory setting as well, we have incorporated the same compressed representation into the dKaMinPar partitioner. We denote the configuration of dKaMinPar which uses this representation as xTeraPart.

Recall that standard dKaMinPar stores a local subgraph of the global input graph on each PE using an uncompressed CSR representation. In xTeraPart, we replace this structure with the aforementioned gap- and interval-encoded representation, refer to Section 5.3.1 for more details on the compression scheme. However, auxiliary data structures (in particular,

data structures required to map between local ghost vertex IDs and their global vertex IDs) remain uncompressed. Consequently, despite the use of compression, we expect xTeraPart to be less memory-efficient than its shared-memory counterpart TeraPart.

To evaluate xTeraPart, we will conduct experiments on massive randomly generated graphs containing up to 2^{44} undirected edges. Since storing graphs of this magnitude on disk is impractical, we generate them in-memory using the KaGen [Fun+18] graph generator. However, standard KaGen generates graphs as uncompressed edge lists. On this scale, this representation is infeasible to hold in memory. Therefore, as a secondary contribution, we have extended KaGen with a streaming capability, allowing us to compress the edges during generation on-the-fly without storing the complete uncompressed graph in memory. Although independent of our partitioner, we briefly outline this feature below.

KaGen constructs random geometric graphs by partitioning the geometric space (i.e., the unit square, unit cube or unit disk in case of 2D or 3D random geometric or random hyperbolic graphs) into chunks and assigning them to PEs. Each PE samples vertices at random positions within its assigned chunk and subsequently generates the corresponding edges. To handle edges that connect vertices across chunk boundaries, KaGen relies on deterministic pseudo-random number generators for vertex sampling, allowing communication-free re-computations of vertices near the borders of chunks owned by other PEs.

To encode such graphs using the aforementioned scheme during generation, we partition the geometric space into a considerably larger number of chunks than the number of PEs. Each PE processes its assigned chunks sequentially, generating edges for one chunk at a time. Immediately after a chunk is processed, we compress the edges and discard the uncompressed edge list, ensuring that only the compressed representation remains in memory. Moreover, we discard all sampled vertices before continuing with the next chunk. This increases the running time due to more frequent vertex recomputation, but keeps memory overheads minimal.

7.2.6 Further Implementation Details

Vertex and Edge IDs. To reduce the communication overheads, we distinguish between local- and global vertex- and edge identifiers. This allows us to use 64 bit data types for global and 32 bit data types for local IDs.

Low-latency Irregular All-to-All. As stated before, many steps of dKaMinPar require communication between interface vertices and their ghost replicas, which translates to irregular and often sparse all-to-all communication. We observed that for small messages (e.g., when exchanging updated labels during label propagation), the startup cost for the standard `MPI_Alltoallv` as implemented in OpenMPI (via $\mathcal{O}(p^2)$ point-to-point messages) quickly dominates overall performance. Thus, for small message sizes, we use a two-level approach [SS23a] that organizes PEs in a grid. Then, instead of sending a message from PE (i_s, j_s) to PE (i_t, j_t) directly, it is routed via PE (i_s, j_t) (i.e., first send to PE (i_s, j_t), and then from PE (i_s, j_t) to PE (i_t, j_t). This reduces the number of messages sent through the network from $\mathcal{O}(p^2)$ to $\mathcal{O}(p)$.

7.3 Experiments

We implemented the proposed algorithm dKaMinPar resp. xTeraPart in C++ and compiled it using GCC-13.2.0 with flags `-O3 -march=native`. We use Intel TBB [TBB] and OpenMPI 4.0 as parallelization libraries and growt [MSD19] for hash tables.

Setup. We evaluate the solution quality of our algorithm on a shared-memory machine equipped with 1 TB of main memory and one AMD EPYC 7702P processor with 64 cores (Machine A in Table 2.2). Additionally, we perform scalability experiments on the HoreKa high-performance cluster, where each compute node is equipped with 256 GB of main memory and two Intel Xeon Platinum 8368 processors (Machine C) for a total of 78 cores per compute node. We only use 64 out of the available 78 cores since some of the graph generators that we use for our weak-scaling experiments require the number of cores to be a power of two. The compute nodes are connected by an InfiniBand 4X HDR 200 GBit/s network with approx. 1 μs latency. While our partitioner can use multiple threads per MPI process, we only evaluate the configuration with one thread per MPI process, since this usually gives the best performance.

Competitors and Algorithm Configuration. We compare dKaMinPar against the distributed versions of the algorithms included in Chapter 4, i.e., ParHIP [MSS17] (v3.14) and ParMETIS [KK99] (v4.0.3). ParHIP implements two configuration presets, which we denote as ParHIP-Fast (or simply ParHIP in experiments that only use this preset) and ParHIP-Eco. The presets configure a trade-off between running time and partition quality. More precisely, the Fast preset uses sequential KaHIP [SS11a] for initial partitioning, whereas the Eco preset spends a fixed time budget of $\frac{2^{11}}{p}$ s on an evolutionary algorithm [SS12] to compute an initial partition on p PEs. Moreover, ParHIP uses a varying number of V-cycles [Wal04] to improve the partition: the Fast preset performs 2 V-cycles, while the Eco preset performs 5 V-cycles. For ParMETIS, we use the standard configuration. We do not include the distributed version of PuLP [SMR14] (i.e., XtraPuLP [Slo+17]) in our comparison, since its quality is not competitive with multilevel partitioners, see Table 4.3. We do also not include Pt-Scotch [CP08] since it is outperformed by ParHIP in terms of both running time and solution quality [MSS17].

We evaluate four configurations of our algorithm. All configurations fix $C = 2\,000$ (the same value as used by ParHIP-Fast and by KaMinPar introduced in Chapter 4). First, dKaMinPar refers to the configuration which only uses label propagation for refinement and does not use the compressed graph representation. In contrast, xTeraPart uses the compressed representation. The configuration using a single round of Jet refinement (in addition to label propagation) is denoted as dJet, and the configuration using four rounds with varying temperatures as d4xJet.

Instances. We evaluate our algorithm on the graphs of Benchmark Set B from Ref. [Got+21b] and the graphs used in Ref. [MSS17]. We refer to this benchmark set as Benchmark Set E and list the graphs in Tables 7.1 and 7.2. As before, we roughly classify the graphs as *regular* (mesh-like, with a small maximum degree) or *irregular* (large maximum degree, e.g., complex networks). The benchmark set contains both artificial and real-world graphs with vertex counts ranging from 863.0 K to 214.0 M and edge counts ranging from 2.4 M to 3.3 G. Additionally, we use KaGen [Fun+18] to generate random graphs with controlled parameters to evaluate the weak-scaling capabilities of our algorithms. More precisely, we use 2D and 3D random geometric graphs (denoted rgg2D and rgg3D, respectively) as well as random hyperbolic graphs (denoted rhg, with power-law exponent $\gamma = 3.0$). Random geometric graphs resemble mesh-like graphs, while random hyperbolic graphs feature a power-law degree distribution and therefore resemble complex networks.

Table 7.1 Regular graphs of Benchmark Set E, characterized by the number of vertices (n), the number of undirected edges (m), and the maximum degree (Δ).

Graph	$n/10^3$	$m/10^3$	Δ	Graph	$n/10^3$	$m/10^3$	Δ
Artificial [Bad+12; Fun+18]							
Rgg2e27-2D	134 218	1 193 357	46	Del2e27-3D	134 218	1 042 574	40
Rgg2e27-3D	134 218	787 814	36	Rgg2e26-2D	67 109	574 554	45
Del2e26-3D	67 109	521 273	40	Rgg2e26-3D	67 109	377 952	34
Del2e27-2D	134 218	303 207	14	Del2e26-2D	67 109	201 327	26
Biology [DH11]							
kmerV1r	214 004	232 705	8	kmerA2a	170 372	179 942	40
kmerP1a	138 896	148 465	40	kmerU1a	64 678	66 394	35
Finite Element [Bad+12]							
ChannelB050	4 802	42 681	18	Hugebubbles010	19 458	29 180	3
PackingB050	2 146	17 488	18				
Other [Bad+12]							
Nlpkkt240	27 994	373 239	27	EuropeOSM	50 912	54 055	13

Table 7.2 Irregular graphs of Benchmark Set E, characterized by the number of vertices (n), the number of undirected edges (m), and the maximum degree (Δ).

Graph	$n/10^3$	$m/10^3$	Δ	Graph	$n/10^3$	$m/10^3$	Δ
Social [Les]							
Twitter2010	41 652	1 202 513	2 997 487	Orkut	3 073	117 185	33 313
Youtube	1 135	2 988	28 754	Amazon	401	2 350	2 747
Web [Lab]							
Uk2007	105 897	3 301 877	975 419	Sk2005	50 636	1 810 063	8 563 816
It2004	41 292	1 027 475	1 326 744	Webbase2001	118 142	854 810	816 127
Uk2005	39 460	783 027	1 776 858	Arabic2005	22 744	553 903	575 628
Uk2002	18 520	261 787	194 955	Eu2005	863	16 138	68 963
In2004	1 383	13 591	21 869				
Wiki [Lab]							
Enwiki2018	5 617	117 244	248 444	Enwiki2013	4 207	91 940	432 260

Methodology. We call a combination of a graph and the number of blocks an *instance*. For single-node experiments, we use $k \in \{2, 4, \ldots, 128\}$, $\varepsilon = 3\%$ and perform 5 repetitions for each instance with different seeds for the pseudo-random number generator. For multi-node experiments (i.e., experiments performed on the HoreKa cluster), we only use $k = 16$ and limit repetitions to 3 to reduce computational cost. To aggregate results (edge cuts and running times) for the same instance, we use the arithmetic mean. When aggregating results across multiple instances, we always use the geometric mean.

7.3.1 Solution Quality and Running Time

We evaluate the partition quality and running time of dKaMinPar across the graphs of Benchmark Set E on our 64-core single-node Machine A. For this experiment, we use $k \in \{2, 4, 8, 16, 32, 64, 128\}$ with $\varepsilon = 3\%$ and run each distributed partitioner with 64 MPI processes. The partition quality and running times are shown in Figure 7.4.

As shown in the performance profile (Figure 7.4, left), ParHIP-Eco finds the best

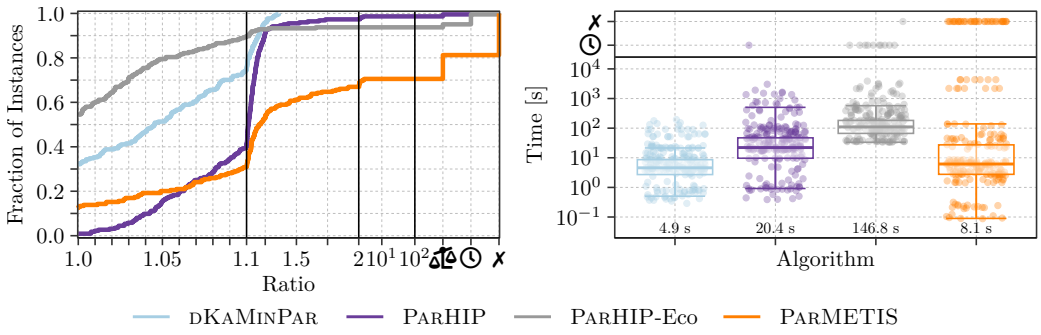

Figure 7.4 Results for $k = \{2, 4, 8, 16, 32, 64, 128\}$ with $\varepsilon = 3\%$ on the single-node machine Machine A. Left: partition quality of distributed-memory dKaMinPar, ParHIP (Fast and Eco presets), and ParMETIS. Right: The corresponding running times; the numbers above the x-axis indicate geometric mean running times over all instances on which none of the partitioners crashed. Timeouts are marked with 🕓, and failed runs are marked with ✗.

partitions for 54.5% of the instances, followed by dKaMinPar (32.1%) and ParMETIS (12.5%). Since ParHIP-Eco often outperforms ParHIP-Fast, the latter only computes the best partitions for 0.9% of the instances. However, we can also see that ParHIP-Eco produces imbalanced solutions on 3 out of 224 instances and exceeds the one-hour time limit on 10 instances. ParMETIS produces imbalanced solutions on over 10% of the instances (24 of 224 instances) and frequently crashes (on 42 instances). All of the instances on which ParMETIS crashes are classified as irregular. The reason for this is most likely ParMETIS's coarsening strategy, which relies on contracting normal matchings (there is no two-hop matching). This is ineffective on complex networks, where a large fraction of the edges are incident to only few vertices. Thus, there are only small maximal matchings.

While ParHIP-Eco offers better solution quality than dKaMinPar, it incurs significant running time overheads. As illustrated in Figure 7.4 (right), dKaMinPar is the fastest algorithm; with an average running time of 4.9 s, it is 30× faster than ParHIP-Eco (146.8 s). Note that these averages are calculated only over instances where all algorithms either computed a partition or exceeded the time limit (counting the limit as the running time). Moreover, dKaMinPar is more than 4 times faster than ParHIP-Fast (20.4 s) on average, and therefore outperforms it in both solution quality *and* running time. The comparison with ParMETIS is more nuanced. Although ParMETIS (8.1 s) is on average 1.7× slower than dKaMinPar, its performance depends heavily on the graph structure. It is faster than dKaMinPar on 112 out of 224 instances, specifically those classified as regular (i.e., listed in Table 7.1). Across these instances, it is 1.8× faster than dKaMinPar. However, on irregular instances (i.e., graphs listed in Table 7.2), ParMETIS is 14.0× slower than dKaMinPar.

Finally, we compare our distributed partitioner against its shared-memory counterpart KaMinPar in Figure 7.5. We run KaMinPar with 64 threads and dKaMinPar with 64 MPI processes. First, we observe that KaMinPar computes slightly better edge cuts. This is expected due to the factors discussed in Section 7.2.2 and Section 7.2.4: in the distributed setting, we have less control over the cluster and block weights during coarsening and refinement, and rely on batched parallelism during label propagation instead of asynchronous label propagation. However, we also note that the quality gap is small. KaMinPar computes better cuts than dKaMinPar for 53.6% of the instances, while dKaMinPar computes better cuts for the remaining 46.4%. On average, edge cuts computed by dKaMinPar

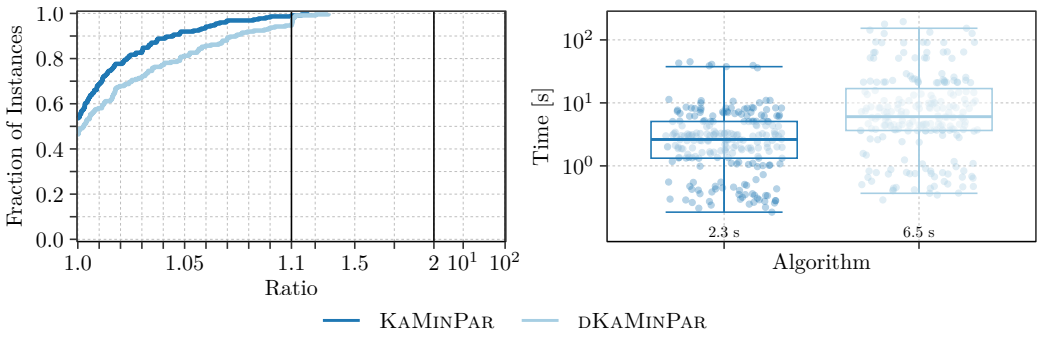

Figure 7.5 Results for $k = \{2, 4, 8, 16, 32, 64, 128\}$ with $\varepsilon = 3\%$ on the single-node machine Machine A. Left: Partition quality of the shared-memory KaMinPar (64 threads) and distributed-memory dKaMinPar (64 MPI processes). Right: The corresponding running times; the numbers above the x-axis indicate geometric mean running times over all instances. Note that these running times differ from the values shown in Figure 7.4 (right), as the latter only aggregates over instances for which all partitioners computed a partition or exceeded the time limit.

are only 0.96% larger than those of KaMinPar. Second, dKaMinPar is on average 2.8× slower than KaMinPar. This overhead is also expected, as message-passing communication inherently incurs higher cost than shared-memory synchronization. Moreover, we require more synchronization due to the batched-parallel label propagation implementation.

7.3.2 Weak Scalability of dKaMinPar

Next, we evaluate the weak scalability of dKaMinPar using families of randomly generated graphs (2D and 3D random geometric and random hyperbolic graphs) with $k = 16$ and $\varepsilon = 3\%$, utilizing 1–128 compute nodes on the HoreKa cluster. Since we use 64 cores per compute node, we subsequently use 64–8 192 cores in total. Since ParHIP-Eco is much slower than the other algorithms, we only compare against ParHIP-Fast, which we simply denote ParHIP from here on. We generate the graphs with 2^{27} vertices per compute node and an average degree $d \in \{16, 32\}$. On 128 compute nodes, the graphs therefore have 2^{34} vertices and up to 2^{39} undirected edges in the denser instances with average degree $d = 32$. The corresponding throughputs are shown in Figure 7.6.

We observe that dKaMinPar maintains weak scalability up to 128 compute nodes across all three graph families: on rgg2D, rgg3D and rhg instances, the average throughputs increase from 55 M, 44 M, and 58 M edges per second on one compute node to 2 884 M, 2 598 M, and 2 956 M edges per second on 128 compute nodes. Averaging over all graphs, throughput increases by a factor of 54.0 when going from one compute node to 128 compute nodes. While ParMETIS achieves slightly higher throughputs on mesh-like geometric graphs (i.e., rgg2D and rgg3D), it scales only to 4 compute nodes on $\mathsf{rhg}_{\gamma=3.0}$ with an average degree of 16, and performs even more poorly on the denser instance. In contrast, ParHIP generally yields considerably lower throughputs than ParMETIS and dKaMinPar on all three families. Moreover, its scalability degrades beyond 32 compute nodes (i.e., 2 048 cores). This is likely due to its extensive communication overhead during coarsening: to contract a clustering, ParHIP redistributes the entire graph across the PEs. However, we note that ParHIP was originally designed to overlap local work and global communication during label propagation through the use of nonblocking MPI operations. The implementation for this relies heavily

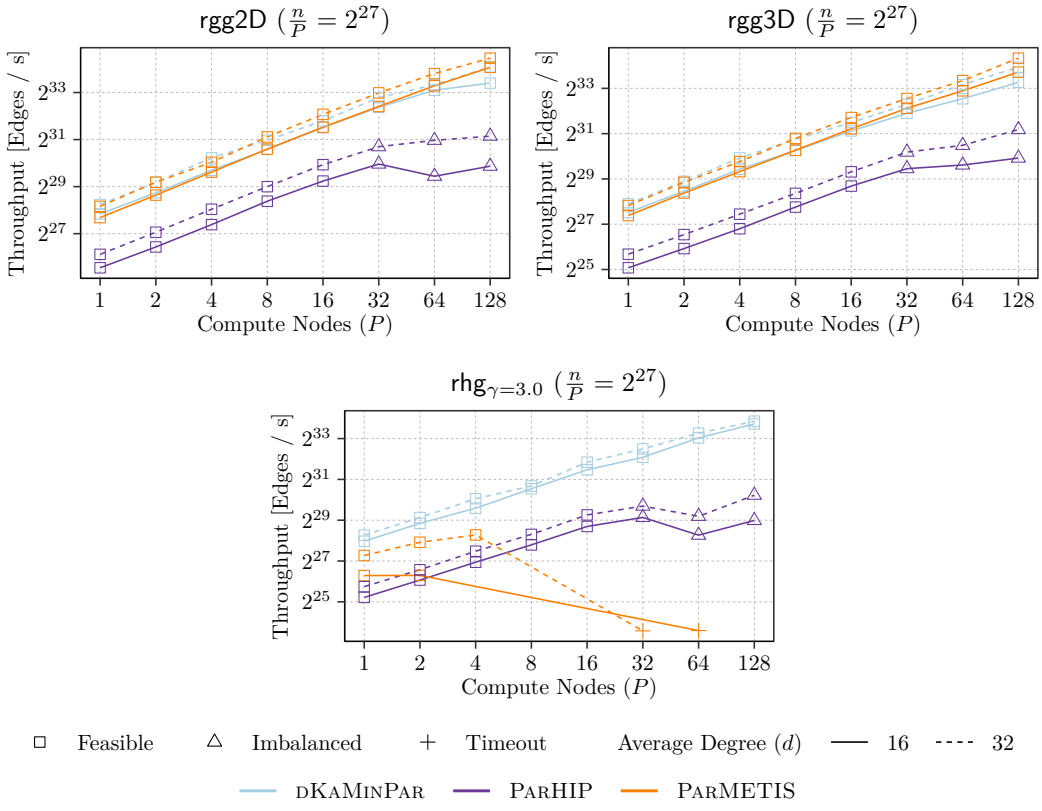

Figure 7.6 Throughput of rgg2D, rgg3D and rhg$_{\gamma=3.0}$ graphs with 2^{27} vertices per compute node, average degree $d \in \{16, 32\}$, $k = 16$, and $\varepsilon = 3\%$ on $P \in \{1, 2, 4, \ldots, 128\}$ compute nodes of the HoreKa cluster, using 64 cores per compute node.

on MPI progression threads, which seem to be unavailable in modern OpenMPI versions. Thus, ParHIP might also perform worse with current OpenMPI versions than with older OpenMPI versions.

The corresponding edge cuts for this experiment are shown in Table 7.4. We find that ParMETIS computes worse cuts than dKaMinPar on the rgg2D graphs (by 11% on average), while computing better cuts on the rgg3D graphs (by 11%). This is likely due to the different coarsening scheme: ParMETIS implements heavy-edge matching (see Section 3.2.1, without two-hop matching), which shrinks the graph considerably slower than the clustering-based coarsening scheme in dKaMinPar. We observed that dKaMinPar shrinks rgg3D very rapidly, which can reduce solution quality when not paired with stronger refinement algorithms. However, for the hyperbolic graph, ParMETIS finds 5.2 times larger cuts than dKaMinPar on average. Such solutions are unsuitable for many applications. ParHIP also computes worse cuts on the rgg2D graphs (by 13%), but computes similar cuts to dKaMinPar for the rgg3D and rhg graphs (2.3% and 0.2% larger cuts than dKaMinPar on average, respectively). However, we note that ParHIP often computes highly imbalanced and thus infeasible partitions on larger core counts (we exclude these instances in the aggregates).

We now evaluate weak scalability in terms of *both* graph size *and* number of blocks k by scaling k proportionally to the number of compute nodes P (i.e., we keep the number of vertices per block fixed). Consequently, the number of blocks grows considerably as the number of compute nodes increases. The throughput of each algorithm in this setting is

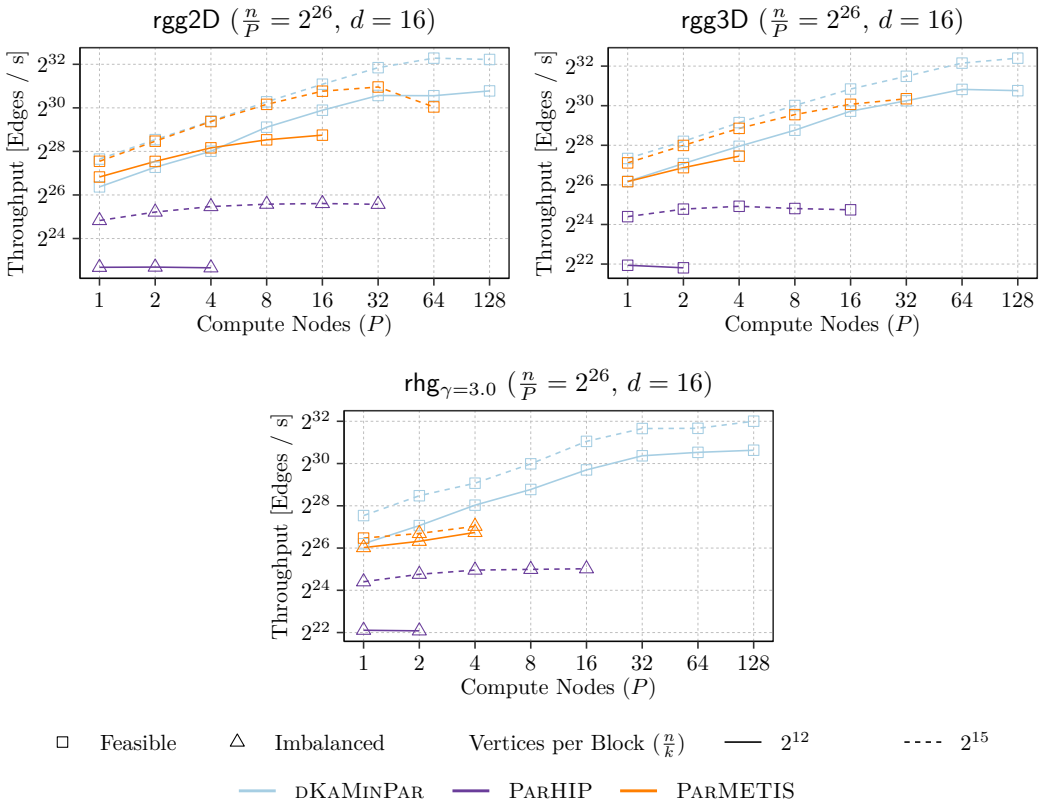

Figure 7.7 Throughput of rgg2D, rgg3D and rhg graphs with 2^{26} vertices per compute node, average degree 16, and $\varepsilon = 3\%$ on 1–128 compute nodes of Machine C. The number of blocks is scaled with the size of the graph such that each block contains 2^{12} or 2^{15} vertices.

summarized in Figure 7.7.

Depending on the graph and the number of blocks k, dKaMinPar scales to 64–128 compute nodes in this setting. As expected, this scaling is more restricted than in the fixed-k weak scaling experiment (compare Figure 7.6), since increasing both n and k results in an increase in workload disproportional to the increase in P.

In contrast, neither competing algorithm performs well in this setting. ParHIP generally achieves much lower throughput across all graph families and fails to compute a partition beyond 2–16 compute nodes, depending on the number of vertices per block. The reason for this limitation is that ParHIP coarsens the graph down to kC vertices (where $C = 5\,000$), before replicating this coarsest graph on every MPI process. For large k, these replicas quickly exceed the available main memory of the compute nodes. Moreover, with such large coarse graphs, initial partitioning dominates the overall running time, preventing throughput gains even at low compute node counts. ParMETIS achieves better throughputs, but similarly fails beyond 4–32 compute nodes. This is likely due to the same bottleneck: like ParHIP, ParMETIS replicates the coarsest graph with kC vertices on every MPI process during initial partitioning, leading to excessive memory consumption and running time overhead. However, ParMETIS performs better since it uses a much smaller constant for C ($C = 20$).

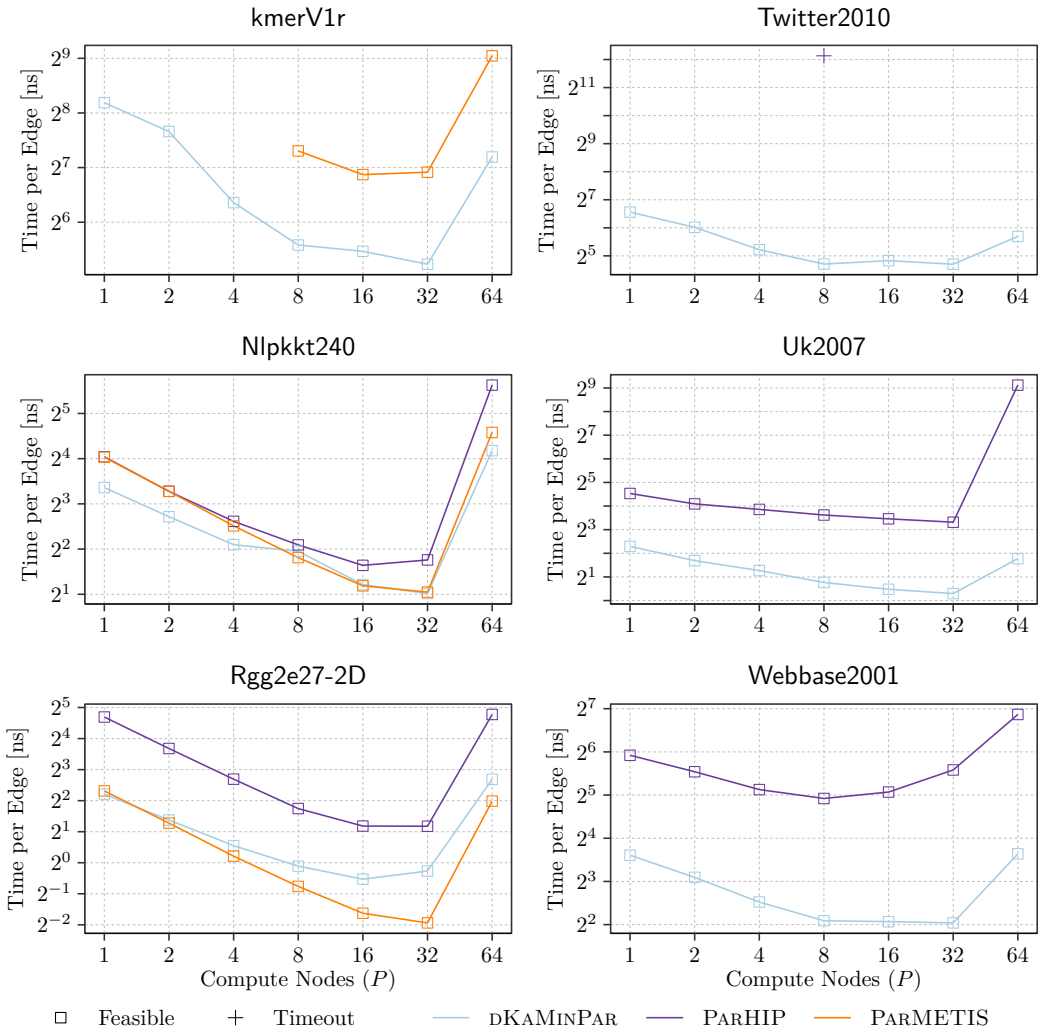

Figure 7.8 Strong scaling running times for the largest low- and high-degree graphs in our benchmark set, with $k = 16$, $\varepsilon = 3\%$ on 1–64 compute nodes of Machine C.

7.3.3 Strong Scalability of dKaMinPar

To evaluate the strong scalability of dKaMinPar, we select three of the largest regular and three of the largest irregular graphs of Benchmark Set E and partition them into k blocks using an increasing number of compute nodes. As before, we set $\varepsilon = 3\%$. The resulting throughputs are shown in Figure 7.8, while the corresponding edge cuts are listed in Table 7.5.

For dKaMinPar, we observe strong scalability up to 8–32 compute nodes on the irregular instances and up to 16–32 compute nodes on regular instances. ParMETIS is unable to partition the irregular graphs regardless of the number of compute nodes used, but demonstrates similar scalability on the regular graphs. On the random geometric graph Rgg2e27-2D only, ParMETIS achieves considerably better throughputs than dKaMinPar on larger core counts. ParHIP generally shows similar scalability to dKaMinPar, but is consistently slower. For example, ParHIP *always* requires more time to partition any of irregular graphs regardless of the number of compute nodes used than dKaMinPar on just a single compute node. The Twitter2010 graph is especially challenging to coarsen efficiently,

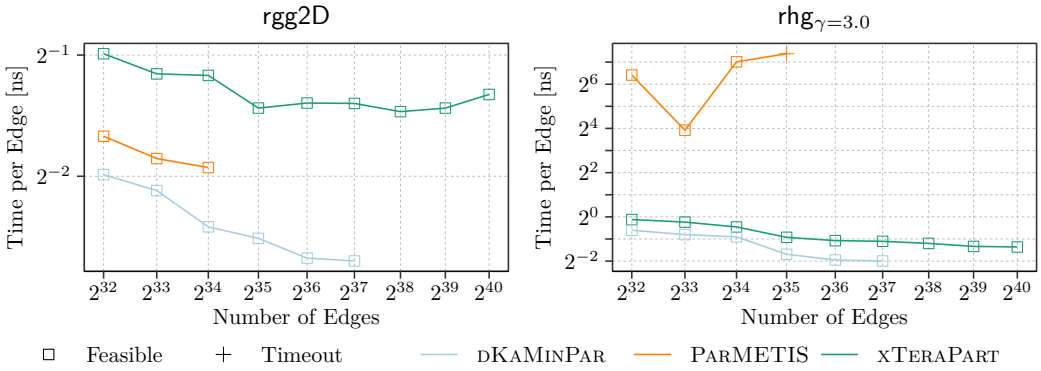

Figure 7.9 Comparison of xTeraPart against ParMETIS on 8 HoreKa compute nodes with increasing rgg2D and rhg graphs. ParMETIS runs out of memory on rgg2D graphs and exceeds our 1 h time limit on rhg graphs beyond 2^{34} and 2^{35} edges, respectively. The corresponding edge cuts are compared in Table 7.3.

causing both competing algorithms to fail on this instance. For ParMETIS, this is likely due to its lack of two-hop matching capabilities, which prevents the algorithm from coarsening the graph efficiently and sufficiently. ParHIP likely fails to compute a balanced clustering during coarsening, which leads to severely imbalanced vertex weights on coarse levels.

Turning to the edge cuts listed in Table 7.5, we find that edge cuts are generally stable as the number of compute nodes increases (for all algorithms). On 64 compute nodes, edge cuts are slightly smaller than on a single compute node (by 2.6% for dKaMinPar). ParHIP produces larger edge cuts on all graphs except Nlpkkt240, where its edge cuts are 1.2% smaller than those of dKaMinPar. ParMETIS produces smaller edge cuts for kmerV1r and Nlpkkt240 (by 9.5% and 7.3%, respectively), but computes 18.6% larger cuts for Rgg2e27-2D and fails to partition any of the irregular graphs. Averaged over all graphs and compute node counts (considering only instances where the respective competitor computes a balanced partition), dKaMinPar computes cuts that are 7.4% and 8.6% smaller than those of ParHIP and ParMETIS, respectively.

From these results, we can conclude that dKaMinPar achieves throughputs comparable to or slightly worse than ParMETIS on regular graphs, depending on the instance, while substantially outperforming it on irregular graphs. Compared to ParHIP, dKaMinPar generally yields considerably higher throughputs across both regular and irregular graphs, while producing better edge cuts on most instances. dKaMinPar is the only distributed partitioner in our evaluation that can successfully partition the Twitter2010 graph, and kmerV1r on fewer than 8 compute nodes.

7.3.4 Evaluation of xTeraPart

We now consider additional variants of dKaMinPar. In this section, we equip dKaMinPar with the same compression techniques as implemented in TeraPart, see Section 5.3.1 and Section 7.2.5 for details. We refer to the resulting algorithm as xTeraPart and evaluate it on randomly generated rgg2D and rhg graphs with $k = 64$ and $\varepsilon = 3\%$. We compare xTeraPart against dKaMinPar and contrast our results against ParMETIS, as it was the strongest competitor in Section 7.3.2 and Section 7.3.3. In Figure 7.9, we compare xTeraPart against dKaMinPar and ParMETIS by partitioning graphs of increasing size while fixing the number of compute nodes to 8 (i.e., use 512 cores and 2 TiB RAM in total).

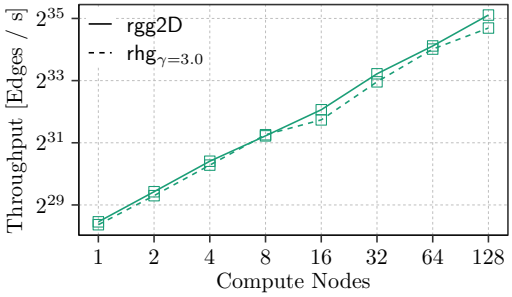

Figure 7.10 Weak scaling results on rgg2D and rhg graphs for xTeraPart with the largest feasible rgg2D and rhg graphs on up to 128 HoreKa compute nodes. The largest graphs have 2^{37} vertices and 2^{44} edges, which xTeraPart can partition in under 10 minutes.

Table 7.3 Edge cuts corresponding to Figure 7.9, reported relative to m (xTeraPart) resp. xTeraPart (ParMETIS). Out-of-memory errors are marked with ✗ and out-of-time with 🕓 (> 1 h).

		Cut rel. to m resp. xTeraPart			
G	Algorithm	$m = 2^{32}$	$m = 2^{33}$	$m = 2^{34}$	$m = 2^{35}$
rgg2D	xTeraPart	0.93%	0.67%	0.48%	0.35%
	ParMETIS	1.00×	1.00×	0.99×	✗
rhg	xTeraPart	0.23%	0.16%	0.11%	0.04%
	ParMETIS	1.12×	1.14×	0.86×	🕓

All graphs are generated with average degree $d = 128$.

We obverse that xTeraPart is able to partition graphs of both families with up to 2^{40} edges, closing the gap to our single-machine tera-scale experiments in Section 5.6.1. There, TeraPart was able to partition the same graphs on a single machine equipped with 1 TiB of RAM. Since distributed graph partitioning requires additional auxiliary data structures, xTeraPart requires slightly more memory to partition these graphs than TeraPart (1.2×–1.3×). In contrast, ParMETIS only manages to partition graphs of either family that are 64 times smaller than that, running out of memory or exceeding our one hour time limit already at 2^{35} edges. Without graph compression, dKaMinPar can partition graphs with up to 2^{37} edges, i.e., 8 times smaller than xTeraPart, requiring 4.8×–4.5× more memory. Compared to the medium-sized graphs of Benchmark Set A on which we evaluted the same compression techniques within TeraPart in Section 5.6.1, we observe that the compression techniques introduce higher running time penalties for these graphs.

In Figure 7.10, we show weak scaling results up to 128 compute nodes (the largest number of nodes that the cluster can allocate to a single job and that is also a power of two). We observe good weak scaling behavior for both graph families. On 128 compute nodes, xTeraPart can partition graphs with 2^{44} edges in slightly less than 10 minutes.

7.3.5 Evaluation of dJet

Finally, we equip dKaMinPar with the stronger Jet refinement algorithm described in Section 7.2.4. As before, we evaluate its partition quality using our 64 core single-node

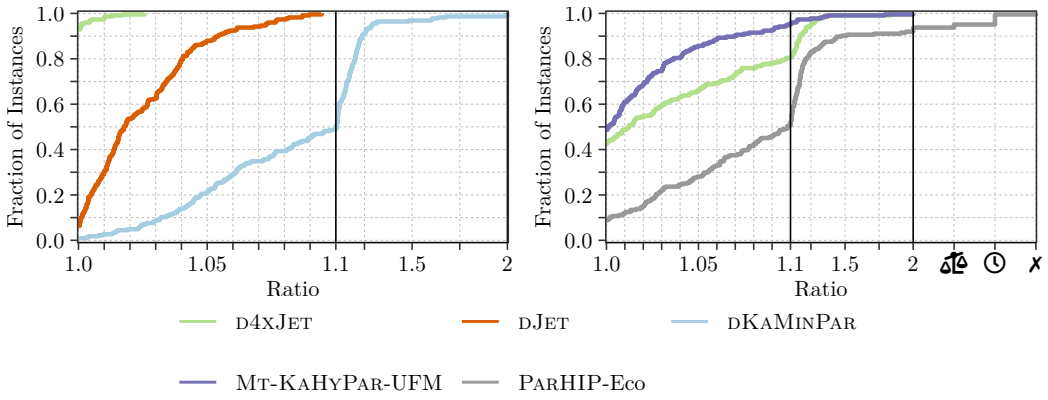

Figure 7.11 Results for $k = \{2, 4, 8, 16, 32, 64, 128\}$ with $\varepsilon = 3\%$ on the 64 core single-node shared-memory machine. Left: partition quality of D4XJET (four rounds of JET with different temperatures), DJET (single-temperature JET) and DKAMINPAR (label propagation refinement only). Right: partition quality of D4XJET, PARHIP-ECO (state-of-the-art distributed partition quality) and MT-KAHYPAR-UFM (state-of-the-art shared-memory partition quality). The corresponding running times are shown in Figure 7.12.

Machine A. Here, we use $k \in \{2, 4, 8, 16, 32, 64, 128\}$ and set $\varepsilon = 3\%$. To evaluate its weak- and strong-scalability, we use 1–128 compute nodes of the HOREKA cluster, using $k = 16$ and $\varepsilon = 3\%$.

To the best of our knowledge, the highest-quality graph partitioner for the distributed setting is PARHIP-Eco, which also computes the best partitions in Section 7.3.1 and Ref. [MSS17]. In the shared-memory setting, MT-KAHYPAR equipped with unconstrained FM refinement [MGS] (denoted MT-KAHYPAR-UFM in the following) achieves the highest partition quality to the best of our knowledge [Got+24a]. Thus, we focus our evaluation in this section on these two competitors. In particular, we do not compare against PARHIP-Fast and PARMETIS, since they compute worse partitions than PARHIP-Eco in Section 7.3.1.

Looking at Figure 7.11 (left), we observe that our strongest configuration with 4 rounds of JET (denoted D4XJET) using different temperatures improves the edge cut by at least 10% on roughly 50% of all benchmark instances compared to plain DKAMINPAR, which only uses label propagation for refinement. This is considered a lot in the context of multilevel graph partitioning, as most multilevel partitioners achieve average edge cuts within a few percentage points of each other (see, for instance, Table 4.3). However, this improvement also comes at a cost: D4XJET is on average 3.42 times slower than DKAMINPAR, see Figure 7.12. Recall that the original JET only uses a single round with one temperature [Gil+24]. We do the same for DJET and observe that using 4 rounds improves edge cuts by 2.3% on average, while increasing running times by a factor of 1.96. Compared to MT-KAHYPAR-UFM [MGS] (run with 64 threads), we find that D4XJET computes partitions with at most 1.5% larger cuts on 50% of all instances, shrinking the quality gap between distributed and shared-memory partitioners considerably (Figure 7.11, right). On average, the cuts found by MT-KAHYPAR-UFM are 2.8% smaller than those of D4XJET (and 5.0% resp. 14.0% smaller than those of DJET resp. DKAMINPAR). However, as a shared-memory algorithm without overheads for message-passing, MT-KAHYPAR-UFM is faster than D4XJET by a factor of 1.28 when run on a single machine. Figure 7.11 (right) also compares D4XJET against PARHIP-Eco. We see that D4XJET finds at least $\approx 8\%$ lower cuts on roughly 50% of all instances. On average,

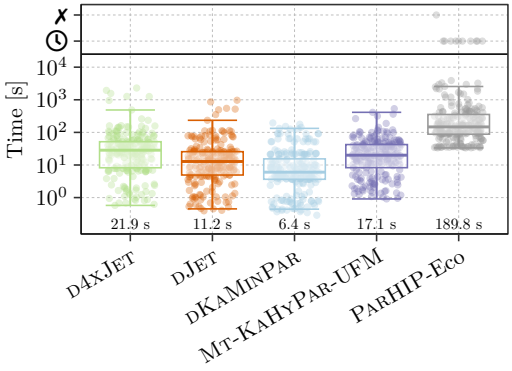

Figure 7.12 Running times for $k = \{2, 4, 8, 16, 32, 64, 128\}$ with $\varepsilon = 3\%$ on the 64 core single-node shared-memory machine. Values above the x-axis state the geometric mean running time across all instances for which all algorithms produced a result. Timeouts (> 1 h) are marked with 🕓, while crashes are marked with ✗. The corresponding partition quality is shown in Figure 7.11.

D4XJET computes 7.0% lower edge cuts than PARHIP-Eco. More, PARHIP-Eco is 8.67 times slower than D4XJET due to its expensive evolutionary initial partitioning algorithm. We can therefore conclude the following. While D4XJET does not quite reach the partition quality possible in the shared-memory setting, it considerably outperforms the previous state-of-the-art in the distributed setting in *both* running time *and* solution quality.

To evaluate the scalability of D4XJET, we perform additional weak- and strong-scaling experiments. Results are shown in Figure 7.13, where we observe weak scalability up to (at least) 128 compute nodes (i.e., 8 192 cores) on randomly generated graphs. Looking at strong scaling results (Figure 7.14), we generally observe scalability up to 16–32 compute nodes on medium-sized real-world regular graphs, and 8–32 compute nodes on the irregular ones. This is roughly in line with competing distributed partitioners (e.g., compare Figure 7.8) and DKAMINPAR.

7.4 Conclusion

Our distributed-memory graph partitioner DKAMINPAR successfully partitions a wide range of input graphs using many thousands of cores, yielding high speed and good quality. Further improvements of the implementation might be possible, for example making better use of shared-memory on each compute node. For instance, initial partitioning is already delegated to KAMINPAR once each core obtained a full copy of the current graph. Instead, KAMINPAR could be invoked using all cores of a compute node once the coarsest graph fits on a single compute node.

We have presented two extensions to our base implementation: XTERAPART incorporates a compressed graph representation to extend scalability up to 2^{44} edges on 128 compute nodes. By adopting the JET [Gil+24] algorithm to the distributed setting and using multiple temperatures, D4XJET achieves considerably improved solution quality over previous distributed partitioners.

Beyond that, one can further explore the quality versus time trade off. Through distributed implementations of more powerful refinement algorithms like flow-based techniques one could achieve better quality at the price of higher execution time. It then also makes sense to look

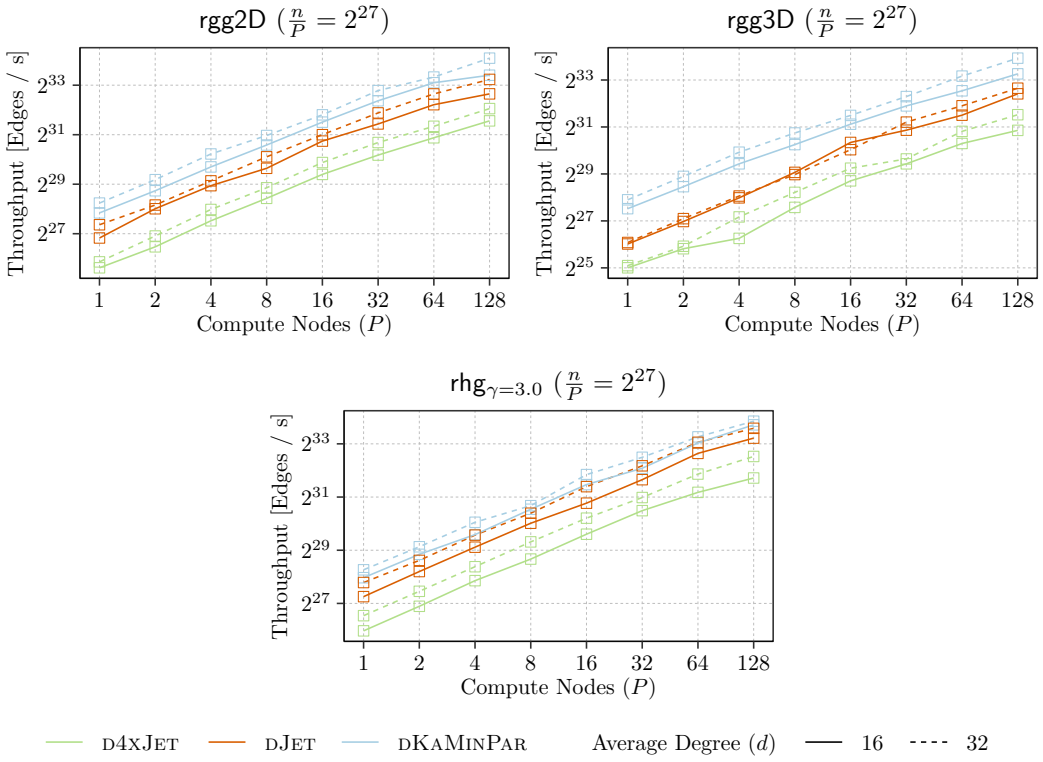

Figure 7.13 Throughputs achieved on randomly generated rgg2D, rgg3D and rhg$_{\gamma=3.0}$ graphs with 2^{27} vertices per compute node, average degree $d \in \{16, 32\}$, $k = 16$ and $\varepsilon = 3\%$ on 1–128 HOREKA compute nodes (with 64 cores per compute node).

at a portfolio of different partitioner variants that can be run in parallel, achieving good quality for subsets of inputs. For example, matching-based coarsening as in PARMETIS might help for mesh-like networks. On the other hand, more aggressive methods for handling high-degree nodes might help with some social networks. For instance, graph sparsification techniques such as those described in Chapter 6 could be used to reduce their degrees. It might also be useful to distribute the neighbors of high-degree vertices across multiple PEs to improve work balance. Lastly, it might help to prepartition graphs to improve graph locality, e.g., by using a streaming partitioner during graph I/O.

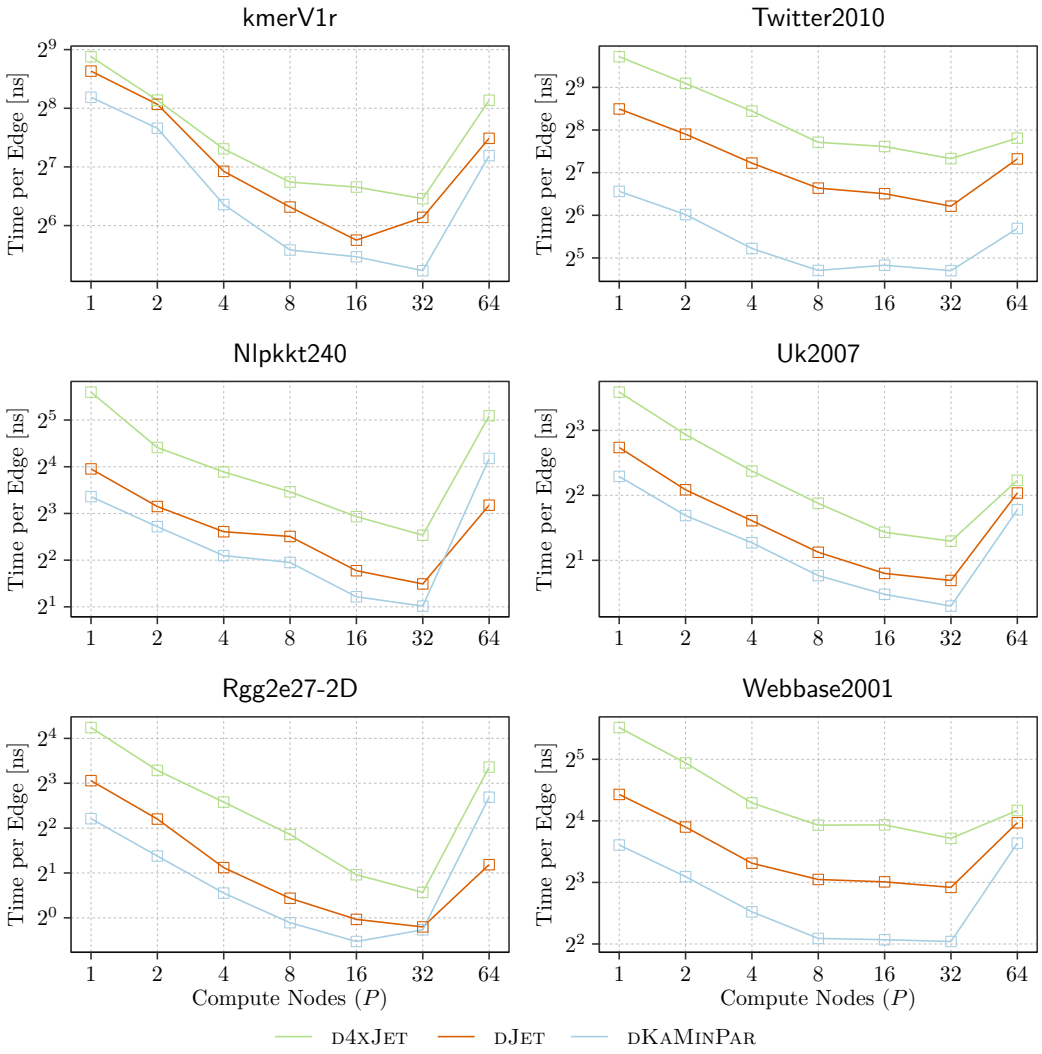

Figure 7.14 Strong scaling running times for large regular and irregular graphs in our benchmark set, with $k = 16$, $\varepsilon = 3\%$ on 1–64 HoreKa compute nodes.

Table 7.4 Edge cut results for our weak scaling experiments on randomly generated graphs using 1–128 compute nodes. Some columns are not shown due to size constraints. Edge cuts are reported as thousands for dKaMinPar, and relative to dKaMinPar for all other algorithms (i.e., a value of < 1 indicates a better partition). Imbalanced partitions are marked with ⚖, and crashes are marked with ✗. The gmean aggregates over all computes node counts and both graphs (average degrees $d = 16$ and $d = 32$).

		Edge Cuts on Number of Compute Nodes / 1 000								
G	Algorithm	2	8	32	128	2	8	32	128	Gmean
rgg2D	dKaMinPar	394	811	1 674	3 451	1 716	3 514	7 250	14 888	2 028
	dJet	1.00	0.98	0.96	0.96	0.97	0.97	0.95	0.93	0.97
	d4xJet	0.96	0.95	0.93	0.92	0.93	0.92	0.90	0.89	0.93
	ParMETIS	1.21	1.21	1.21	1.20	1.01	1.02	1.01	1.00	1.11
	ParHIP	1.07	1.10	⚖	⚖	1.15	1.18	⚖	⚖	1.13
rgg3D	dKaMinPar	7 520	19 204	49 650	128 608	23 726	60 526	156 638	406 198	43 341
	dJet	0.85	0.83	0.82	0.79	0.85	0.82	0.81	0.78	0.83
	d4xJet	0.82	0.79	0.78	0.75	0.82	0.78	0.77	0.73	0.79
	ParMETIS	0.94	0.94	0.95	0.92	0.88	0.88	0.86	0.85	0.90
	ParHIP	1.01	1.01	⚖	⚖	1.03	1.03	⚖	⚖	1.02
rhg	dKaMinPar	3	2	5	2	22	65	36	20	8
	dJet	1.00	1.00	1.00	1.00	0.95	0.94	0.94	1.00	0.98
	d4xJet	1.00	1.00	1.00	1.00	0.95	0.92	0.94	1.00	0.98
	ParMETIS	4.33	✗	✗	✗	4.82	✗	✗	✗	5.19
	ParHIP	0.67	2.50	⚖	⚖	0.73	0.51	⚖	⚖	1.00
		$d = 16$				$d = 32$				

Table 7.5 Edge cuts corresponding to the strong scaling experiments shown in Figure 7.8 and Figure 7.14, using 1–64 compute nodes. Edge cuts are reported as thousands for dKaMinPar, and relative to dKaMinPar for all other algorithms (i.e., a value of < 1 indicates a better partition). Crashes are marked with ✗. The last column titled *Gmean* aggregates across all compute node counts, while the last 5 rows aggregate across all graphs. The last column of the last 5 rows subsequently aggregate across all graphs and all node counts.

		Edge Cut on Number of Compute Nodes / 1 000							
G	Algorithm	1	2	4	8	16	32	64	Gmean
					Irregular Graphs				
Twitter2010	dKaMinPar	616 943	610 419	616 499	597 415	600 907	597 780	578 828	603 242
	dJet	0.78	0.78	0.77	0.79	0.79	0.79	0.81	0.79
	d4xJet	0.77	0.78	0.77	0.79	0.79	0.79	0.81	0.78
	ParMETIS	✗	✗	✗	✗	✗	✗	✗	–
	ParHIP	✗	✗	✗	✗	✗	✗	✗	–
Uk2007	dKaMinPar	4 084	4 021	4 111	4 019	4 038	4 057	3 984	4 032
	dJet	0.90	0.91	0.86	0.88	0.88	0.87	0.88	0.88
	d4xJet	0.87	0.90	0.85	0.86	0.86	0.86	0.87	0.87
	ParMETIS	✗	✗	✗	✗	✗	✗	✗	–
	ParHIP	1.10	1.07	1.06	1.10	1.02	1.04	1.06	1.07
Webbase2001	dKaMinPar	9 573	9 356	9 420	9 380	9 414	9 410	9 388	9 413
	dJet	0.93	0.94	0.93	0.94	0.94	0.94	0.94	0.94
	d4xJet	0.90	0.92	0.92	0.92	0.92	0.93	0.92	0.92
	ParMETIS	✗	✗	✗	✗	✗	✗	✗	–
	ParHIP	1.10	1.12	1.06	1.08	1.12	1.15	1.16	1.12
					Regular Graphs				
kmerV1r	dKaMinPar	10 128	10 260	10 140	10 117	10 133	10 088	10 022	10 110
	dJet	0.94	0.92	0.93	0.91	0.94	0.91	0.90	0.92
	d4xJet	0.92	0.91	0.93	0.92	0.91	0.91	0.90	0.91
	ParMETIS	✗	✗	✗	0.90	0.90	0.90	0.91	0.91
	ParHIP	✗	✗	✗	✗	✗	✗	✗	–
Nlpkkt240	dKaMinPar	5 831	5 743	5 740	5 730	5 711	5 629	5 600	5 727
	dJet	0.89	0.90	0.90	0.90	0.90	0.90	0.90	0.90
	d4xJet	0.83	0.85	0.85	0.85	0.84	0.86	0.85	0.84
	ParMETIS	0.91	0.92	0.92	0.92	0.93	0.94	0.95	0.93
	ParHIP	0.98	0.98	0.99	0.98	0.99	1.00	1.01	0.99
Rgg2e27-2D	dKaMinPar	346	346	344	346	346	345	346	345
	dJet	1.00	1.00	1.00	1.00	0.99	1.00	0.99	1.00
	d4xJet	0.96	0.96	0.97	0.96	0.96	0.96	0.96	0.96
	ParMETIS	1.20	1.20	1.19	1.19	1.18	1.20	1.17	1.19
	ParHIP	1.17	1.16	1.17	1.16	1.14	1.14	1.17	1.16
					Gmean				
	dKaMinPar	8 887	8 812	8 842	8 758	8 779	8 744	8 654	**8 780**
	dJet	0.90	0.91	0.90	0.90	0.90	0.90	0.90	**0.90**
	d4xJet	0.87	0.88	0.88	0.88	0.88	0.88	0.88	**0.88**
	ParMETIS	1.04	1.05	1.05	1.00	1.00	1.01	1.00	**1.01**
	ParHIP	1.09	1.08	1.07	1.08	1.07	1.08	1.10	**1.08**

8 Conclusion

In this work, we looked at regimes in which current state-of-the-art multilevel graph partitioners become brittle: (extremely) large number of blocks k, highly irregular and non-local graphs, and graph instances whose size exceeds the available memory of even high-end single-node servers. The traditional multilevel scheme remains a strong foundation, but must be re-engineered along three axes to remain robust at modern problem scale. First, coarsening must be robust, even for large k and irregular graphs with highly skewed degree distributions. Second, the memory footprint of traditional building blocks must be reduced. Third, scalability must be improved across compute models.

Concretely, we established that multilevel partitioning can be made robust and high-throughput even for very large block counts by extending multilevel thinking deep into the initial partitioning phase. The resulting Deep MGP scheme, presented in Chapter 4, only performs a single (un)coarsening cycle, in which bipartitioning is applied to graphs of controlled size during uncoarsening. This approach combines the merits of direct k-way partitioning – performing a single multilevel cycle with k-way refinement across the hierarchy – and those of recursive bipartitioning, applying potentially costly bipartitioning only to small graphs. Depending on the concrete building blocks for coarsening, uncoarsening, and bipartitioning, Deep MGP can be implemented with linear work and polylogarithmic span unless k is very large.

We further instantiated Deep MGP with parallel building blocks for coarsening, balancing, k-way refinement, and sequential bipartitioning on small graphs to obtain KaMinPar. Notably, we propose a clustering-based coarsening scheme that achieves provable geometric vertex-reduction rates on arbitrary input graphs and arbitrary values of k. The resulting shared-memory implementation, KaMinPar, is engineered for scalability and high-throughput, thereby improving the state-of-the-art in graph partitioning empirically. Across a large and diverse set of benchmark instances, it is more than 2× faster than the previously fastest partitioner Mt-METIS, even for small values of k, while achieving better solution quality on average. KaMinPar is, on average, even faster than the single-level partitioner PuLP. For large values of k, it achieves much better robustness, consistently computing balanced partitions within seconds to minutes on large graphs. In particular, it is at least 3.8× faster than Mt-METIS, which on average produces 3–5% larger edge cuts (and often violates the balance constraint, in particular for large k). All other competitors included in our evaluation are at least 25× slower than KaMinPar in this setting, with Mt-KaHyPar being the only partitioner that finds better edge cuts on average (by 12%, while being two orders of magnitude slower).

During the development of KaMinPar, we identified its memory footprint as the primary obstacle for scaling to (much) larger input graphs. Depending on the graph instance, it would either exceed the available memory of our high-end machines, or finish within minutes. In Chapter 5, we therefore addressed this bottleneck and showed that multilevel partitioning can be made substantially more memory-efficient without sacrificing throughput or solution quality. The resulting partitioner, TeraPart, combines two-phase label propagation to reduce auxiliary memory in clustering without sacrificing efficiency, a compressed graph

representation, an improved coarse-graph construction algorithm, and space-efficient gain tables that make parallel FM refinement feasible at scale. Together, these techniques reduce peak memory considerably while preserving competitive running times and solution quality, and they unlock in-memory partitioning for graphs that are otherwise infeasible. At the extreme end, our evaluation showed that huge web graphs with up to one hundred billion edges as well as tera-scale synthetic graphs can be partitioned within minutes on a single shared-memory machine.

In Chapter 6, we subsequently addressed the worst-case behaviour of multilevel graph partitioning. Multilevel schemes can incur superlinear work on instance families where coarsening fails to achieve geometric shrinkage of the edges. We showed that by integrating edge sparsification into coarsening, one can enforce constant-factor reductions between successive levels. This yields a multilevel partitioner with a $\mathcal{O}(n + m)$ expected total-work guarantee without any assumptions on the input graph. We integrated the resulting linear-time multilevel algorithm into KaMinPar. Experiments demonstrated substantial speedups across a benchmark set of near-worst-case instances at only minor average quality loss, and performance that exceeds that of competitive single-level and streaming-based partitioners.

In Chapter 7, we showed that Deep MGP can be realized efficiently in the distributed-memory model, and that many of the improvements observed in the shared-memory setting carry over. The resulting system, dKaMinPar, retains label propagation as the core primitive for both coarsening and refinement, and achieves scalable enforcement of balance constraints during both coarsening and uncoarsening despite limited synchronization and delayed visibility of remote moves. Beyond the basic system, we adapted the Jet algorithm to the distributed-memory model, thereby narrowing the long-standing quality gap to high-quality shared-memory partitioners considerably. We further combined our distributed partitioner with the compression techniques of TeraPart to obtain xTeraPart. Our evaluation demonstrated scalability to thousands of cores and graphs far beyond single-node memory limits, including synthetic instances with up to 16 trillion edges.

Across these chapters, we substantially broaden the practical operating range of multilevel graph partitioning. We show that multilevel methods can remain fast and robust at very large block counts, that their memory footprint can be reduced to make previously infeasible instances tractable in main memory, that near-worst-case coarsening behaviour can be controlled with linear expected work, and that these ideas transfer to distributed memory to scale beyond single-node limits. Taken together, these results push multilevel graph partitioning to a new scale in throughput, robustness, and memory efficiency, while preserving competitive solution quality. This is in contrast to alternative approaches such as single-level partitioning or streaming algorithms, that historically only reached these scales at a notable quality cost.

We highlight possible directions for future work in the respective conclusion sections of the individual chapters.

Appendix

List of Algorithms

List of Figures

5 Tera-Scale Multilevel Graph Partitioning

List of Tables

Publications and Supervised Theses

In Conference Proceedings

- L. Gottesbüren, N. Maas, D. Rosch, P. Sanders, and D. Seemaier. "Linear-Time Multilevel Graph Partitioning via Edge Sparsification". In: *33rd Annual European Symposium on Algorithms, ESA 2025, September 15-17, 2025, Warsaw, Poland*. Volume 351. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 32:1–32:20. DOI: 10.4230/LIPICS.ESA.2025.32
- A. Chhabra, F. Kurpicz, C. Schulz, D. Schweisgut, and D. Seemaier. "Partitioning Trillion Edge Graphs on Edge Devices". In: *2025 Proceedings of the Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 74–89. DOI: 10.1137/1.9781611978759.6
- D. Salwasser, D. Seemaier, L. Gottesbüren, and P. Sanders. "Tera-Scale Multilevel Graph Partitioning". In: *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2025, Milano, Italy, June 3-7, 2025*. IEEE, 2025, pages 285–296. DOI: 10.1109/IPDPS64566.2025.00033
- T. N. Uhl, M. Schimek, L. Hübner, D. Hespe, F. Kurpicz, D. Seemaier, C. Stelz, and P. Sanders. "KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI". in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2024, Atlanta, GA, USA, November 17-22, 2024*. IEEE, 2024, page 44. DOI: 10.1109/SC41406.2024.00050
- L. Gottesbüren, N. Maas, P. Sanders, and D. Seemaier. "Modern Software Libraries for Graph Partitioning (Abstract)". In: *Proceedings of the 2024 ACM Workshop on Highlights of Parallel Computing, HOPC 2024, Nantes, France, 17 June 2024*. ACM, 2024. DOI: 10.1145/3670684.3673417
- D. Hespe, L. Hübner, F. Kurpicz, P. Sanders, M. Schimek, D. Seemaier, and T. N. Uhl. "Brief Announcement: (Near) Zero-Overhead C++ Bindings for MPI". in: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2024, Nantes, France, June 17-21, 2024*. ACM, 2024, pages 289–291. DOI: 10.1145/3626183.3660260
- P. Sanders and D. Seemaier. "Brief Announcement: Distributed Unconstrained Local Search for Multilevel Graph Partitioning". In: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2024, Nantes, France, June 17-21, 2024*. ACM, 2024, pages 443–445. DOI: 10.1145/3626183.3660257
- A. Chhabra, M. F. Faraj, C. Schulz, and D. Seemaier. "Buffered Streaming Edge Partitioning". In: *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*. Volume 301. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 5:1–5:21. DOI: 10.4230/LIPICS.SEA.2024.5
- N. Maas, L. Gottesbüren, and D. Seemaier. "Parallel Unconstrained Local Search for Partitioning Irregular Graphs". In: *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 32–45. DOI: 10.1137/1.9781611977929.3. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611977929.3

- P. Sanders and D. Seemaier. "Distributed Deep Multilevel Graph Partitioning". In: *Euro-Par 2023: Parallel Processing - 29th International Conference on Parallel and Distributed Computing, Limassol, Cyprus, August 28 - September 1, 2023, Proceedings*. Volume 14100. Lecture Notes in Computer Science. Springer, 2023, pages 443–457. DOI: 10.1007/978-3-031-39698-4_30

- L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier. "Deep Multilevel Graph Partitioning". In: *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*. Volume 204. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 48:1–48:17. DOI: 10.4230/LIPICS.ESA.2021.48

- M. Popp, S. Schlag, C. Schulz, and D. Seemaier. "Multilevel Acyclic Hypergraph Partitioning". In: *Proceedings of the 23rd Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*. SIAM, 2021, pages 1–15. DOI: 10.1137/1.9781611976472.1

- S. Schlag, C. Schulz, D. Seemaier, and D. Strash. "Scalable Edge Partitioning". In: *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*. SIAM, 2019, pages 211–225. DOI: 10.1137/1.9781611975499.17

Journal Articles

- Ü. V. Çatalyürek, K. D. Devine, M. F. Faraj, L. Gottesbüren, T. Heuer, H. Meyerhenke, P. Sanders, S. Schlag, C. Schulz, D. Seemaier, and D. Wagner. "More Recent Advances in (Hyper)Graph Partitioning". In: *ACM Comput. Surv.* 55.12 (2023), 253:1–253:38. DOI: 10.1145/3571808

Theses

- D. Seemaier. "Acyclic n-Level Hypergraph Partitioning". Master's thesis. Karlsruhe Institute of Technology, 2020

- D. Seemaier. "Engineering Graph Partitioning Algorithms to Minimize Communication Volume". Bachelor's thesis. Karlsruhe Institute of Technology, 2017

Supervised Theses

- F. Kellenbenz. “Buffered Streaming Hypergraph Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2026
- J. Hayes. “Shared-Memory Parallelization of Memetic Hypergraph Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2026
- T. Kempf. “Engineering Dynamic Hypergraph Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2025
- S. H. Schrape. “Improving Coarsening for Multilevel Graph Partitioning via Machine Learning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2025
- M. Hilgers. “Evaluation Embedding-Based Coarsening for Multilevel Graph Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2024
- D. Rosch. “Sparsification for Linear Time Multilevel Graph Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2024
- K. von Pückler. “Engineering Multi-Constraint Graph Partitioning Algorithms with Unconstrained Refinement”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2024
- E. Yilmaz. “Evaluation Coarsening Algorithms for Multilevel Hypergraph Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2024
- D. Salwasser. “Optimizing a Parallel Graph Partitioner for Memory Efficiency”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2024
- S. Gil. “Engineering Asynchronous Label Propagation for Multilevel Graph Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2024
- T. Fritsch. “Multilevel Hypergraph Partitioning with Unrestricted Coarsening and Rebalancing Techniques”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2023
- C. Knoesel. “Evaluation of Local Search Heuristics for Graph Partitioning with Large k”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2022
- M. Haag. “Engineering of Algorithms for Very Large k Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2021
- T. Fuchs. “Machine-Learning based Hypergraph Pruning for Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2020

Usage of Generative Models

In accordance with the Deutsche Forschungsgemeinschaft statement[1] on the use of generative large language models, we disclose that such models were used for the following purposes:

- We used GitHub Copilot for line-level code completions while implementing the described algorithms in C++. All suggestions made by the model were reviewed by the author and adopted only where appropriate.
- We used ChatGPT-{4o, 5, 5.1, 5.2} during the writing of this thesis for sentence-level refinement (improve sentence flow, select more precise verbs, and shorten formulations). All suggestions made by the model were reviewed by the author and adopted only where appropriate.
- We used ChatGPT-{4o, 5, 5.1, 5.2} to identify grammatical, spelling, and typographical errors.

No generative models were used for the design of the algorithms or experimental studies. No images in this thesis were generated using generative models.

[1] https://www.dfg.de/de/aktuelles/neuigkeiten-themen/info-wissenschaft/2023/info-wissenschaft-23-72

Bibliography

[AB12] B. F. Auer and R. H. Bisseling. "Graph coarsening and clustering on the GPU". In: *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*. Volume 588. Contemporary Mathematics. American Mathematical Society, 2012, page 223. URL: http://www.ams.org/books/conm/588/11706.

[Abb+18] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov. "Streaming Graph Partitioning: An Experimental Study". In: *PVLDB* 11.11 (2018), pages 1590–1603. DOI: 10.14778/3236187.3236208.

[ABR20] S. Acer, E. G. Boman, and S. Rajamanickam. "SPHYNX: Spectral Partitioning for HYbrid aNd aXelerator-enabled systems". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*. IEEE, 2020, pages 440–449. DOI: 10.1109/IPDPSW50202.2020.00082.

[Ace+21] S. Acer, E. G. Boman, C. A. Glusa, and S. Rajamanickam. "Sphynx: A parallel multi-GPU graph partitioner for distributed-memory systems". In: *Parallel Comput.* 106 (2021), page 102769. DOI: 10.1016/J.PARCO.2021.102769.

[ACU08] C. Aykanat, B. B. Cambazoglu, and B. Uçar. "Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices". In: *J. Parallel Distributed Comput.* 68.5 (2008), pages 609–625. DOI: 10.1016/J.JPDC.2007.09.006.

[AK06] A. Abou-Rjeili and G. Karypis. "Multilevel algorithms for partitioning power-law graphs". In: *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006. DOI: 10.1109/IPDPS.2006.1639360.

[Akh+17] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. "Engineering a direct k-way Hypergraph Partitioning Algorithm". In: *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017*. SIAM, 2017, pages 28–42. DOI: 10.1137/1.9781611974768.3.

[Akh19] Y. Akhremtsev. "Parallel and External High Quality Graph Partitioning". PhD thesis. Karlsruhe Institute of Technology, Germany, 2019. URL: https://nbn-resolving.org/urn:nbn:de:101:1-201910300359299618 5903.

[Ang+22] E. Angriman, A. van der Grinten, M. Hamann, H. Meyerhenke, and M. Penschuck. "Algorithms for Large-Scale Network Analysis and the NetworKit Toolkit". In: *Algorithms for Big Data - DFG Priority Program 1736*. Volume 13201. Lecture Notes in Computer Science. Springer, 2022, pages 3–20. DOI: 10.1007/978-3-031-21534-6_1.

[ANK13] N. K. Ahmed, J. Neville, and R. R. Kompella. "Network Sampling: From Static to Streaming Graphs". In: *ACM Trans. Knowl. Discov. Data* 8.2 (2013), 7:1–7:56. DOI: 10.1145/2601438.

[AR06] K. Andreev and H. Räcke. "Balanced Graph Partitioning". In: *Theory Comput. Syst.* 39.6 (2006), pages 929–939. DOI: 10.1007/S00224-006-1350-7.

[ASS15] Y. Akhremtsev, P. Sanders, and C. Schulz. "(Semi-)External Algorithms for Graph Partitioning and Clustering". In: *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*. SIAM, 2015, pages 33–43. DOI: 10.1137/1.9781611973754.4.

[ASS20] Y. Akhremtsev, P. Sanders, and C. Schulz. "High-Quality Shared-Memory Graph Partitioning". In: *IEEE Transactions on Parallel and Distributed Systems* 31.11 (2020), pages 2710–2722. DOI: 10.1109/TPDS.2020.3001645.

[Avi83] D. Avis. "A survey of heuristics for the weighted matching problem". In: *Networks* 13.4 (1983), pages 475–493. DOI: 10.1002/NET.3230130404.

[AY95] C. J. Alpert and S. Yao. "Spectral Partitioning: The More Eigenvectors, The Better". In: *Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995*. ACM Press, 1995, pages 195–200. DOI: 10.1145/217474.217529.

[Ayk+07] C. Aykanat, B. B. Cambazoglu, F. Findik, and T. M. Kurç. "Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids". In: *J. Parallel Distributed Comput.* 67.1 (2007), pages 77–99. DOI: 10.1016/J.JPDC.2006.05.005.

[Bad+12] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering*. 2012.

[BCP18] R. Barat, C. Chevalier, and F. Pellegrini. "Multi-criteria Graph Partitioning with Scotch". In: *Proceedings of the Eighth SIAM Workshop on Combinatorial Scientific Computing, CSC 2018, Bergen, Norway, June 6-8, 2018*. SIAM, 2018, pages 66–75. DOI: 10.1137/1.9781611975215.7.

[BDR13] E. G. Boman, K. D. Devine, and S. 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'13, Denver, CO, USA - November 17 - 21, 2013*. ACM, 2013, 50:1–50:12. DOI: 10.1145/2503210.2503293.

[BH11] U. Benlic and J. Hao. "An effective multilevel tabu search approach for balanced graph partitioning". In: *Comput. Oper. Res.* 38.7 (2011), pages 1066–1075. DOI: 10.1016/J.COR.2010.10.007.

[Bir+13] M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava. "Efficient Parallel and External Matching". In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Volume 8097. Lecture Notes in Computer Science. Springer, 2013, pages 659–670. DOI: 10.1007/978-3-642-40047-6_66.

[BK96] A. A. Benczúr and D. R. Karger. "Approximating s-t Minimum Cuts in $\tilde{O}(n^2)$ Time". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. ACM, 1996, pages 47–55. DOI: 10.1145/237814.237827.

[Blo+08] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008. DOI: 10.1088/1742-5468/2008/10/P10008.

[BLV14] F. Bourse, M. Lelarge, and M. Vojnovic. "Balanced graph edge partition". In: *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. ACM, 2014, pages 1456–1465. DOI: 10.1145/2623330.2623660.

[BMR83] A. Brandt, S. McCormick, and J. Ruge. "Algebraic Multigrid (AMG) for Automatic Multigrid Solution with Application to Geodetic Computations". In: (Jan. 1983).

[Bol+11] P. Boldi, M. Rosa, M. Santini, and S. Vigna. "Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks". In: *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*. ACM, 2011, pages 587–596. DOI: 10.1145/1963405.1963488.

[Bol+18] P. Boldi, A. Marino, M. Santini, and S. Vigna. "BUbiNG: Massive Crawling for the Masses". In: *ACM Trans. Web* 12.2 (2018), 12:1–12:26. DOI: 10.1145/3160017.

[Bop87] R. B. Boppana. "Eigenvalues and Graph Bisection: An Average-Case Analysis (Extended Abstract)". In: *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*. IEEE Computer Society, 1987, pages 280–285. DOI: 10.1109/SFCS.1987.22.

[BS13] C. Bichot and P. Siarry. *Graph Partitioning*. ISTE. Wiley, 2013. ISBN: 9781118601259.

[BS93] S. T. Barnard and H. D. Simon. "A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems". In: *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, PP 1993, Norfolk, Virginia, USA, March 22-24, 1993*. SIAM, 1993, pages 711–718.

[Bul+16] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. "Recent Advances in Graph Partitioning". In: *Algorithm Engineering - Selected Results and Surveys*. Volume 9220. Lecture Notes in Computer Science. 2016, pages 117–158. DOI: 10.1007/978-3-319-49487-6_4.

[BV04] P. Boldi and S. Vigna. "The webgraph framework I: compression techniques". In: *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*. ACM, 2004, pages 595–602. DOI: 10.1145/988672.988752.

[ÇA99] Ü. V. Çatalyürek and C. Aykanat. "Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication". In: *IEEE Trans. Parallel Distributed Syst.* 10.7 (1999), pages 673–693. DOI: 10.1109/71.780863.

[Çat+12] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. "Multithreaded Clustering for Multi-level Hypergraph Partitioning". In: *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, pages 848–859. DOI: 10.1109/IPDPS.2012.81.

[Çat+23] Ü. V. Çatalyürek, K. D. Devine, M. F. Faraj, L. Gottesbüren, T. Heuer, H. Meyerhenke, P. Sanders, S. Schlag, C. Schulz, D. Seemaier, and D. Wagner. "More Recent Advances in (Hyper)Graph Partitioning". In: *ACM Comput. Surv.* 55.12 (2023), 253:1–253:38. DOI: 10.1145/3571808.

[Che+23] Y. Chen, H. Ye, S. Vedula, A. M. Bronstein, R. G. Dreslinski, T. N. Mudge, and N. Talati. "Demystifying Graph Sparsification Algorithms in Graph Properties Preservation". In: *Proceedings of the VLDB Endowment* 17.3 (2023), pages 427–440. DOI: 10.14778/3632093.3632106.

[Chh+] A. Chhabra, F. Kurpicz, C. Schulz, D. Schweisgut, and D. Seemaier. "Partitioning Trillion Edge Graphs on Edge Devices". In: *2025 Proceedings of the Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 74–89. DOI: 10.1137/1.9781611978759.6.

[Chh+24] A. Chhabra, M. F. Faraj, C. Schulz, and D. Seemaier. "Buffered Streaming Edge Partitioning". In: *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*. Volume 301. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 5:1–5:21. DOI: 10.4230/LIPICS.SEA.2024.5.

[CKM00] A. E. Caldwell, A. B. Kahng, and I. L. Markov. "Iterative Partitioning with Varying Node Weights". In: *VLSI Design* 11.3 (2000), pages 249–258. DOI: 10.1155/2000/15862.

[CL98] J. Cong and S. K. Lim. "Multiway partitioning with pairwise movement". In: *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1998, San Jose, CA, USA, November 8-12, 1998*. ACM / IEEE Computer Society, 1998, pages 512–516. DOI: 10.1145/288548.289079.

[CP08] C. Chevalier and F. Pellegrini. "PT-Scotch: A tool for efficient parallel graph ordering". In: *Parallel Comput.* 34.6-8 (2008), pages 318–331. DOI: 10.1016/J.PARCO.2007.12.001.

[CZF04] D. Chakrabarti, Y. Zhan, and C. Faloutsos. "R-MAT: A Recursive Model for Graph Mining". In: *4th International Conference on Data Mining (ICDM)*. SIAM, 2004, pages 442–446. DOI: 10.1137/1.9781611972740.43.

[Dav+20] T. A. Davis, W. W. Hager, S. P. Kolodziej, and S. N. Yeralan. "Algorithm 1003: Mongoose, a Graph Coarsening and Partitioning Library". In: *ACM Trans. Math. Softw.* 46.1 (2020), 7:1–7:18. DOI: 10.1145/3337792.

[Dev+06] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek. "Parallel hypergraph partitioning for scientific computing". In: *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006. DOI: 10.1109/IPDPS.2006.1639359.

[Dev+16] M. Deveci, S. Rajamanickam, K. D. Devine, and Ü. V. Çatalyürek. "Multi-Jagged: A Scalable Parallel Spatial Partitioning Algorithm". In: *IEEE Trans. Parallel Distributed Syst.* 27.3 (2016), pages 803–817. DOI: 10.1109/TPDS.2015.2412545.

[DH03a] D. E. Drake and S. Hougardy. "A simple approximation algorithm for the weighted matching problem". In: *Inf. Process. Lett.* 85.4 (2003), pages 211–213. DOI: 10.1016/S0020-0190(02)00393-9.

[DH03b] D. E. Drake and S. Hougardy. "Linear Time Local Improvements for Weighted Matchings in Graphs". In: *Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Ascona, Switzerland, May 26-28, 2003, Proceedings*. Volume 2647. Lecture Notes in Computer Science. Springer, 2003, pages 107–119. DOI: 10.1007/3-540-44867-5_9.

[DH11] T. A. Davis and Y. Hu. "The university of Florida sparse matrix collection". In: *ACM Trans. Math. Softw.* 38.1 (2011), 1:1–1:25. DOI: 10.1145/2049662.2049663.

[DH72] W. Donath and A. Hoffman. "Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices". In: *IBM Technical Disclosure Bulletin* 15.3 (1972), pages 938–944.

[DH73] W. E. Donath and A. J. Hoffman. “Lower bounds for the partitioning of graphs”. In: *IBM J. Res. Dev.* 17.5 (Sept. 1973), pages 420–425. ISSN: 0018-8646. DOI: 10.1147/rd.175.0420.

[DM02] E. D. Dolan and J. J. Moré. “Benchmarking optimization software with performance profiles”. In: *Math. Program.* 91.2 (2002), pages 201–213. DOI: 10.1007/S101070100263.

[DMP94] R. Diekmann, B. Monien, and R. Preis. “Using helpful sets to improve graph bisections”. In: *Workshop on Interconnection Networks and Mapping and Scheduling Parallel Computations, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, February 7-9, 1994*. Volume 21. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1994, pages 57–73. DOI: 10.1090/DIMACS/021/06.

[DP14] R. Duan and S. Pettie. “Linear-Time Approximation for Maximum Weight Matching”. In: *J. ACM* 61.1 (2014), 1:1–1:23. DOI: 10.1145/2529989.

[DPS18] R. Duan, S. Pettie, and H. Su. “Scaling Algorithms for Weighted Matching in General Graphs”. In: *ACM Trans. Algorithms* 14.1 (2018), 8:1–8:35. DOI: 10.1145/3155301.

[DT97] S. Dutt and H. Theny. “Partitioning around roadblocks: tackling constraints with intermediate relaxations”. In: *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1997, San Jose, CA, USA, November 9-13, 1997*. IEEE Computer Society / ACM, 1997, pages 350–355. DOI: 10.1109/ICCAD.1997.643546.

[ETS14] H. C. Edwards, C. R. Trott, and D. Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pages 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003.

[FdV23] S. Forster and T. de Vos. “Faster Cut Sparsification of Weighted Graphs”. In: *Algorithmica* 85.4 (2023), pages 929–964. DOI: 10.1007/S00453-022-01053-4.

[Fie75] M. Fiedler. “A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory”. In: *Czechoslovak Mathematical Journal* 25 (1975), pages 619–633. URL: https://api.semanticscholar.org/CorpusID:123006817.

[FM82] C. M. Fiduccia and R. M. Mattheyses. “A linear-time heuristic for improving network partitions”. In: *Proceedings of the 19th Design Automation Conference, DAC ’82, Las Vegas, Nevada, USA, June 14-16, 1982*. ACM/IEEE, 1982, pages 175–181. DOI: 10.1145/800263.809204.

[FS22] M. F. Faraj and C. Schulz. “Buffered Streaming Graph Partitioning”. In: *ACM J. Exp. Algorithmics* 27 (2022), 1.10:1–1.10:26. DOI: 10.1145/3546911.

[Fun+18] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. “Communication-Free Massively Distributed Graph Generation”. In: *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE Computer Society, 2018, pages 336–347. DOI: 10.1109/IPDPS.2018.00043.

[Gab+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. “Open MPI: Goals, Concept, and Design

of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, Sept. 2004, pages 97–104.

[Gab18] H. N. Gabow. "Data Structures for Weighted Matching and Extensions to b-matching and f-factors". In: *ACM Trans. Algorithms* 14.3 (2018), 39:1–39:80. DOI: 10.1145/3183369.

[GBG16] B. Goodarzi, M. Burtscher, and D. Goswami. "Parallel Graph Partitioning on a CPU-GPU Architecture". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pages 58–66. DOI: 10.1109/IPDPSW.2016.16.

[GH22] L. Gottesbüren and M. Hamann. "Deterministic Parallel Hypergraph Partitioning". In: *Euro-Par 2022: Parallel Processing - 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22-26, 2022, Proceedings*. Volume 13440. Lecture Notes in Computer Science. Springer, 2022, pages 301–316. DOI: 10.1007/978-3-031-12597-3_19.

[GHS22] L. Gottesbüren, T. Heuer, and P. Sanders. "Parallel Flow-Based Hypergraph Partitioning". In: *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*. Volume 233. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 5:1–5:21. DOI: 10.4230/LIPICS.SEA.2022.5.

[GHW19] L. Gottesbüren, M. Hamann, and D. Wagner. "Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm". In: *27th Annual European Symposium on Algorithms, ESA 2019, Munich/Garching, Germany, September 9-11, 2019*. Volume 144. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 52:1–52:17. DOI: 10.4230/LIPICS.ESA.2019.52.

[Gil+21] M. S. Gilbert, S. Acer, E. G. Boman, K. Madduri, and S. Rajamanickam. "Performance-Portable Graph Coarsening for Efficient Multilevel Graph Analysis". In: *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 2021, pages 213–222. DOI: 10.1109/IPDPS49936.2021.00030.

[Gil+24] M. S. Gilbert, K. Madduri, E. G. Boman, and S. Rajamanickam. "Jet: Multilevel Graph Partitioning on Graphics Processing Units". In: *SIAM J. Sci. Comput.* 46.5 (2024), page 700. DOI: 10.1137/23M1559129.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[Goo+19] B. Goodarzi, F. Khorasani, V. Sarkar, and D. Goswami. "High Performance Multilevel Graph Partitioning on GPU". In: *17th International Conference on High Performance Computing & Simulation, HPCS 2019, Dublin, Ireland, July 15-19, 2019*. IEEE, 2019, pages 769–778. DOI: 10.1109/HPCS48598.2019.9188120.

[Got+20] L. Gottesbüren, M. Hamann, S. Schlag, and D. Wagner. "Advanced Flow-Based Multilevel Hypergraph Partitioning". In: *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*. Volume 160. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 11:1–11:15. DOI: 10.4230/LIPICS.SEA.2020.11.

[Got+21a] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag. "Scalable Shared-Memory Hypergraph Partitioning". In: *Proceedings of the Symposium on Algorithm*

Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021. SIAM, 2021, pages 16–30. DOI: 10.1137/1.9781611976472.2.

[Got+21b] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier. "Deep Multilevel Graph Partitioning". In: *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*. Volume 204. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 48:1–48:17. DOI: 10.4230/LIPICS.ESA.2021.48.

[Got+22] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag. "Shared-Memory n-level Hypergraph Partitioning". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*. SIAM, 2022, pages 131–144. DOI: 10.1137/1.9781611977042.11.

[Got+24a] L. Gottesbüren, T. Heuer, N. Maas, P. Sanders, and S. Schlag. "Scalable High-Quality Hypergraph Partitioning". In: *ACM Trans. Algorithms* 20.1 (2024), 9:1–9:54. DOI: 10.1145/3626527.

[Got+24b] L. Gottesbüren, N. Maas, P. Sanders, and D. Seemaier. "Modern Software Libraries for Graph Partitioning (Abstract)". In: *Proceedings of the 2024 ACM Workshop on Highlights of Parallel Computing, HOPC 2024, Nantes, France, 17 June 2024*. ACM, 2024. DOI: 10.1145/3670684.3673417.

[Got+25] L. Gottesbüren, N. Maas, D. Rosch, P. Sanders, and D. Seemaier. "Linear-Time Multilevel Graph Partitioning via Edge Sparsification". In: *33rd Annual European Symposium on Algorithms, ESA 2025, September 15-17, 2025, Warsaw, Poland*. Volume 351. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 32:1–32:20. DOI: 10.4230/LIPICS.ESA.2025.32.

[Got23] L. Gottesbüren. "Parallel and Flow-Based High Quality Hypergraph Partitioning". PhD thesis. Karlsruhe Institute of Technology, Germany, 2023. URL: https://nbn-resolving.org/urn:nbn:de:101:1-202304260459210760238 4.

[Gra69] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies". In: *SIAM Journal of Applied Mathematics* 17.2 (1969), pages 416–429.

[Haa21] M. Haag. "Engineering of Algorithms for Very Large k Partitioning". Bachelor's thesis. Karlsruhe Institute of Technology, 2021.

[Ham] W. L. Hamilton. "Graph Representation Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.3 (), pages 1–159.

[Han+19] M. Hanai, T. Suzumura, W. J. Tan, E. S. Liu, G. Theodoropoulos, and W. Cai. "Distributed Edge Partitioning for Trillion-edge Graphs". In: *Proc. VLDB Endow.* 12.13 (2019), pages 2379–2392. DOI: 10.14778/3358701.3358706.

[Her+17] J. Herrmann, J. Kho, B. Uçar, K. Kaya, and Ü. V. Çatalyürek. "Acyclic Partitioning of Large Directed Acyclic Graphs". In: *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. IEEE Computer Society / ACM, 2017, pages 371–380. DOI: 10.1109/CCGRID.2017.101.

[Her+19] J. Herrmann, M. Y. Özkaya, B. Uçar, K. Kaya, and Ü. V. Çatalyürek. "Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs". In: *SIAM J. Sci. Comput.* 41.4 (2019), A2117–A2145. DOI: 10.1137/18M1176865.

[Hes+24] D. Hespe, L. Hübner, F. Kurpicz, P. Sanders, M. Schimek, D. Seemaier, and T. N. Uhl. "Brief Announcement: (Near) Zero-Overhead C++ Bindings for MPI". In: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2024, Nantes, France, June 17-21, 2024*. ACM, 2024, pages 289–291. DOI: 10.1145/3626183.3660260.

[Heu22] T. Heuer. “Scalable High-Quality Graph and Hypergraph Partitioning”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2022. URL: https://nbn-resolving.org/urn:nbn:de:101:1-20221207035935604963 25.

[HL95a] B. Hendrickson and R. Leland. “A Multi-Level Algorithm For Partitioning Graphs”. In: *Supercomputing ’95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. 1995, pages 28–28.

[HL95b] B. Hendrickson and R. W. Leland. “An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations”. In: *SIAM J. Sci. Comput.* 16.2 (1995), pages 452–469. DOI: 10.1137/0916028.

[HMS21] T. Heuer, N. Maas, and S. Schlag. “Multilevel Hypergraph Partitioning with Vertex Weights Revisited”. In: *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*. Volume 190. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 8:1–8:20. DOI: 10.4230/LIPICS.SEA.2021.8.

[HNS18] A. Henzinger, A. Noe, and C. Schulz. “ILP-based Local Search for Graph Partitioning”. In: *17th International Symposium on Experimental Algorithms, SEA 2018*. Volume 103. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 4:1–4:15. DOI: 10.4230/LIPIcs.SEA.2018.4.

[HR73] L. Hyafil and R. L. Rivest. *Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems*. Technical report Rapport de Recherche no. 33. IRIA – Laboratoire de Recherche en Informatique et Automatique, Oct. 1973.

[HS17] T. Heuer and S. Schlag. “Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure”. In: *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Volume 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 21:1–21:19. DOI: 10.4230/LIPICS.SEA.2017.21.

[HS18] M. Hamann and B. Strasser. “Graph Bisection with Pareto Optimization”. In: *ACM J. Exp. Algorithmics* 23 (2018). DOI: 10.1145/3173045.

[HS20] L. Hübschle-Schneider and P. Sanders. “Linear work generation of R-MAT graphs”. In: *Netw. Sci.* 8.4 (2020), pages 543–550. DOI: 10.1017/NWS.2020.21.

[HSS10] M. Holtgrewe, P. Sanders, and C. Schulz. “Engineering a scalable high quality graph partitioner”. In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010, pages 1–12. DOI: 10.1109/IPDPS.2010.5470485.

[HSS19] T. Heuer, P. Sanders, and S. Schlag. “Network Flow-Based Refinement for Multilevel Hypergraph Partitioning”. In: *ACM J. Exp. Algorithmics* 24.1 (2019), 2.3:1–2.3:36. DOI: 10.1145/3329872.

[HSS24] M. R. Hajidehi, S. Sridhar, and M. I. Seltzer. “CUTTANA: Scalable Graph Partitioning for Faster Distributed Graph Databases and Analytics”. In: *Proc. VLDB Endow.* 18.1 (2024), pages 14–27. URL: https://www.vldb.org/pvldb/vol18/p14-hajidehi.pdf.

[Int24] Intel Corporation. *Intelő 64 and IA-32 Architectures Software Developer’s Manual, Volume 2 (2A, 2B, 2C, & 2D): Instruction Set Reference, A-Z*. Technical report 325383-081US. Intel Corporation, June 2024. URL: https://cdrdv2.intel.com/v1/dl/getContent/671110.

[IWW93] E. Ihler, D. Wagner, and F. Wagner. “Modeling Hypergraphs by Graphs with the Same Mincut Properties”. In: *Inf. Process. Lett.* 45.4 (1993), pages 171–175. DOI: 10.1016/0020-0190(93)90115-P.

[Jez15] A. Jez. “Faster Fully Compressed Pattern Matching by Recompression”. In: 11.3 (2015), 20:1–20:43. DOI: 10.1145/2631920.

[Kab+17] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, A. Shalita, Y. Akhremtsev, and A. Presta. “Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner”. In: *Proc. VLDB Endow.* 10.11 (2017), pages 1418–1429. DOI: 10.14778/3137628.3137650.

[Kar+99] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. “Multilevel hypergraph partitioning: applications in VLSI domain”. In: *IEEE Trans. Very Large Scale Integr. Syst.* 7.1 (1999), pages 69–79. DOI: 10.1109/92.748202.

[Kar13] G. Karypis. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* Version 5.1.0. Department of Computer Science & Engineering, University of Minnesota. Mar. 2013. URL: https://karypis.github.io/glaros/files/sw/metis/manual.pdf.

[KGB15] F. Khorasani, R. Gupta, and L. N. Bhuyan. “Scalable SIMD-Efficient Graph Processing on GPUs”. In: *2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*. IEEE Computer Society, 2015, pages 39–50. DOI: 10.1109/PACT.2015.15.

[KGK10] K. Kourtis, G. I. Goumas, and N. Koziris. “Exploiting compression opportunities to improve SpMxV performance on shared memory systems”. In: *ACM Trans. Archit. Code Optim.* 7.3 (2010), 16:1–16:31. DOI: 10.1145/1880037.1880041.

[KGM] R. Krause, L. Gottesbüren, and N. Maas. “Deterministic Parallel High-Quality Hypergraph Partitioning”. In: *2025 Proceedings of the Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 222–236. DOI: 10.1137/1.9781611978759.17. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611978759.17.

[KK95a] G. Karypis and V. Kumar. “Analysis of Multilevel Graph Partitioning”. In: *Proceedings Supercomputing ’95, San Diego, CA, USA, December 4-8, 1995*. ACM, 1995, page 29. DOI: 10.1145/224170.224229.

[KK95b] G. Karypis and V. Kumar. “Multilevel Graph Partitioning Schemes”. In: *Proceedings of the 1995 International Conference on Parallel Processing, Urbana-Champain, Illinois, USA, August 14-18, 1995. Volume III: Algorithms & Applications*. CRC Press, 1995, pages 113–122.

[KK96] G. Karypis and V. Kumar. “Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, November 17-22, 1996, Pittsburgh, PA, USA*. IEEE Computer Society, 1996, page 35. DOI: 10.1109/SC.1996.32.

[KK97] G. Karypis and V. Kumar. “A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm”. In: *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PP 1997, Hyatt Regency Minneapolis on Nicollet Mall Hotel, Minneapolis, Minnesota, USA, March 14-17, 1997*. SIAM, 1997.

[KK98a] G. Karypis and V. Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM J. Sci. Comput.* 20.1 (1998), pages 359–392. DOI: 10.1137/S1064827595287997.

[KK98b] G. Karypis and V. Kumar. “Multilevel Algorithms for Multi-Constraint Graph Partitioning”. In: *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA*. IEEE Computer Society, 1998, page 28. DOI: 10.1109/SC.1998.10018.

[KK98c] G. Karypis and V. Kumar. “Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *J. Parallel Distributed Comput.* 48.1 (1998), pages 96–129. DOI: 10.1006/JPDC.1997.1404.

[KK99] G. Karypis and V. Kumar. “Parallel Multilevel series k-Way Partitioning Scheme for Irregular Graphs”. In: *SIAM Rev.* 41.2 (1999), pages 278–300. DOI: 10.1137/S0036144598334138.

[KL70] B. W. Kernighan and S. Lin. “An efficient heuristic procedure for partitioning graphs”. In: *Bell Syst. Tech. J.* 49.2 (1970), pages 291–307. DOI: 10.1002/J.1538-7305.1970.TB01770.X.

[Kny01] A. V. Knyazev. “Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method”. In: *SIAM J. Sci. Comput.* 23.2 (2001), pages 517–541. DOI: 10.1137/S1064827500366124.

[Lab] U. o. M. Laboratory of Web Algorithms. *Datasets*. URL: http://law.di.unimi.it/datasets.php.

[LaS+15] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis. “Improving Graph Partitioning for Modern Graphs and Architectures”. In: *Proceedings of the 5th Workshop on Irregular Applications - Architectures and Algorithms, IA3 2015, Austin, Texas, USA, November 15, 2015*. ACM, 2015, 14:1–14:4. DOI: 10.1145/2833179.2833188.

[Les] J. Leskovec. *Stanford Network Analysis Package (SNAP)*.

[LF06] J. Leskovec and C. Faloutsos. “Sampling from large graphs”. In: *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*. ACM, 2006, pages 631–636. DOI: 10.1145/1150402.1150479.

[Li+17] L. Li, R. Geda, A. B. Hayes, Y. Chen, P. Chaudhari, E. Z. Zhang, and M. Szegedy. “A Simple Yet Effective Balanced Edge Partition Model for Parallel Computing”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 1.1 (2017), 14:1–14:21. DOI: 10.1145/3084451.

[Lin+15] G. Lindner, C. L. Staudt, M. Hamann, H. Meyerhenke, and D. Wagner. “Structure-Preserving Sparsification of Social Networks”. In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2015, Paris, France, August 25 - 28, 2015*. ACM, 2015, pages 448–454. DOI: 10.1145/2808797.2809313.

[LK13] D. Lasalle and G. Karypis. “Multi-threaded Graph Partitioning”. In: *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*. IEEE Computer Society, 2013, pages 225–236. DOI: 10.1109/IPDPS.2013.50.

[LK16] D. LaSalle and G. Karypis. “A Parallel Hill-Climbing Refinement Algorithm for Graph Partitioning”. In: *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*. IEEE Computer Society, 2016, pages 236–241. DOI: 10.1109/ICPP.2016.34.

[LRT79] R. J. Lipton, D. J. Rose, and R. E. Tarjan. “Generalized Nested Dissection”. In: *SIAM Journal on Numerical Analysis* 16.2 (1979), pages 346–358. DOI: 10.1137/0716027.

[Mal+10] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. “Pregel: a system for large-scale graph processing”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 2010, pages 135–146. DOI: 10.1145/1807167.1807184.

[Mar+17] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. “Spinner: Scalable Graph Partitioning in the Cloud”. In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 2017, pages 1083–1094. DOI: 10.1109/ICDE.2017.153.

[May+18] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, L. Laich, L. Rieger, and K. Rothermel. “ADWISE: Adaptive Window-Based Streaming Edge Partitioning for High-Speed Graph Processing”. In: *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 2018, pages 685–695. DOI: 10.1109/ICDCS.2018.00072.

[MB07] F. Manne and R. H. Bisseling. “A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem”. In: *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007, Revised Selected Papers*. Volume 4967. Lecture Notes in Computer Science. Springer, 2007, pages 708–717. DOI: 10.1007/978-3-540-68111-3_74.

[Mer+25] N. Merkel, D. Stoll, R. Mayer, and H.-A. Jacobsen. *An Experimental Comparison of Partitioning Strategies for Distributed Graph Neural Network Training*. 2025. DOI: 10.48786/EDBT.2025.14.

[Mes23] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[Meu+15] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. “The Graph Structure in the Web - Analyzed on Different Aggregation Levels”. In: *J. Web Sci.* 1.1 (2015), pages 33–47. DOI: 10.1561/106.00000003.

[MGS] N. Maas, L. Gottesbüren, and D. Seemaier. “Parallel Unconstrained Local Search for Partitioning Irregular Graphs”. In: *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 32–45. DOI: 10.1137/1.9781611977929.3. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611977929.3.

[MH11] J. C. Miller and A. A. Hagberg. “Efficient Generation of Networks with Given Expected Degrees”. In: *8th International Workshop on Algorithms and Models for the Web Graph (WAW)*. Springer, 2011, pages 115–126. DOI: 10.1007/978-3-642-21286-4_10.

[MMS09a] H. Meyerhenke, B. Monien, and T. Sauerwald. “A new diffusion-based multilevel algorithm for computing graph partitions”. In: *J. Parallel Distributed Comput.* 69.9 (2009), pages 750–761. DOI: 10.1016/J.JPDC.2009.04.005. URL: https://doi.org/10.1016/j.jpdc.2009.04.005.

[MMS09b] H. Meyerhenke, B. Monien, and S. Schamberger. “Graph partitioning and disturbed diffusion”. In: *Parallel Comput.* 35.10-11 (2009), pages 544–569. DOI: 10.1016/J.PARCO.2009.09.006. URL: https://doi.org/10.1016/j.parco.2009.09.006.

[MPD00] B. Monien, R. Preis, and R. Diekmann. “Quality matching and local improvement for multilevel graph-partitioning”. In: *Parallel Comput.* 26.12 (2000), pages 1609–1634. DOI: 10.1016/S0167-8191(00)00049-1.

[MPS17] O. Moreira, M. Popp, and C. Schulz. “Graph Partitioning with Acyclicity Constraints”. In: *16th International Symposium on Experimental Algorithms, SEA 2017, London, UK, June 21-23, 2017*. Volume 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 30:1–30:15. DOI: 10.4230/LIPICS.SEA.2017.30.

[MPS20] O. Moreira, M. Popp, and C. Schulz. “Evolutionary multi-level acyclic graph partitioning”. In: *J. Heuristics* 26.5 (2020), pages 771–799. DOI: 10.1007/S10732-020-09448-8.

[MS04] B. Monien and S. Schamberger. “Graph Partitioning with the Party Library: Helpful-Sets in Practice”. In: *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2004), 27-29 October 2004, Foz do Iguacu, Brazil*. IEEE Computer Society, 2004, pages 198–205. DOI: 10.1109/SBAC-PAD.2004.18.

[MS07] J. Maue and P. Sanders. “Engineering Algorithms for Approximate Weighted Matching”. In: *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*. Volume 4525. Lecture Notes in Computer Science. Springer, 2007, pages 242–255. DOI: 10.1007/978-3-540-72845-0_19.

[MSD19] T. Maier, P. Sanders, and R. Dementiev. “Concurrent Hash Tables: Fast and General(?)!” In: *ACM Trans. Parallel Comput.* 5.4 (2019), 16:1–16:32. DOI: 10.1145/3309206.

[MSS14] H. Meyerhenke, P. Sanders, and C. Schulz. “Partitioning Complex Networks via Size-Constrained Clustering”. In: *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*. Volume 8504. Lecture Notes in Computer Science. Springer, 2014, pages 351–363. DOI: 10.1007/978-3-319-07959-2_30.

[MSS17] H. Meyerhenke, P. Sanders, and C. Schulz. “Parallel Graph Partitioning for Complex Networks”. In: *IEEE Trans. Parallel Distributed Syst.* 28.9 (2017), pages 2625–2638. DOI: 10.1109/TPDS.2017.2671868.

[NF98] V. Nr and P.-O. Fjallstrom. “Algorithms for Graph Partitioning: A Survey”. In: (Oct. 1998).

[NM16] M. Naumov and T. Moon. “Parallel Spectral Graph Partitioning”. In: 2016. URL: https://api.semanticscholar.org/CorpusID:51920131.

[NU13] J. Nishimura and J. Ugander. “Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing”. In: *19th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining (KDD)*. 2013, pages 1106–1114.

[OS10] V. Osipov and P. Sanders. “n-Level Graph Partitioning”. In: *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*. Volume 6346. Lecture Notes in Computer Science. Springer, 2010, pages 278–289. DOI: 10.1007/978-3-642-15775-2_24.

[Pel07a] F. Pellegrini. “A Parallelisable Multi-level Banded Diffusion Scheme for Computing Balanced Partitions with Smooth Boundaries”. In: *Euro-Par 2007 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 195–204. ISBN: 978-3-540-74466-5.

[Pel07b] F. Pellegrini. “A Parallelisable Multi-level Banded Diffusion Scheme for Computing Balanced Partitions with Smooth Boundaries”. In: *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*. Volume 4641. Lecture Notes in Computer Science. Springer, 2007, pages 195–204. DOI: 10.1007/978-3-540-74466-5_22.

[Pel09] F. Pellegrini. *Contributions au partitionnement de graphes parallèle multi-niveaux (Contributions to parallel multilevel graph partitioning)*. 2009. URL: https://tel.archives-ouvertes.fr/tel-00540581.

[Pop+21] M. Popp, S. Schlag, C. Schulz, and D. Seemaier. “Multilevel Acyclic Hypergraph Partitioning”. In: *Proceedings of the 23rd Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*. SIAM, 2021, pages 1–15. DOI: 10.1137/1.9781611976472.1.

[PR96] F. Pellegrini and J. Roman. “SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs”. In: *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1996, Brussels, Belgium, April 15-19, 1996, Proceedings*. Volume 1067. Lecture Notes in Computer Science. Springer, 1996, pages 493–498. DOI: 10.1007/3-540-61142-8_588.

[Pre99] R. Preis. “Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs”. In: *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*. Volume 1563. Lecture Notes in Computer Science. Springer, 1999, pages 259–269. DOI: 10.1007/3-540-49116-3_24.

[PSL90] A. Pothen, H. D. Simon, and K.-P. Liou. “Partitioning Sparse Matrices with Eigenvectors of Graphs”. In: *SIAM Journal on Matrix Analysis and Applications* 11.3 (1990), pages 430–452. DOI: 10.1137/0611030. eprint: https://doi.org/10.1137/0611030.

[RA15] R. A. Rossi and N. K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *AAAI*. 2015. URL: https://networkrepository.com.

[Rah+14] F. Rahimian, A. H. Payberah, S. Girdzijauskas, and S. Haridi. “Distributed Vertex-Cut Partitioning”. In: *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*. Volume 8460. Lecture Notes in Computer Science. Springer, 2014, pages 186–200. DOI: 10.1007/978-3-662-43352-2_15.

[RAK07] U. N. Raghavan, R. Albert, and S. Kumara. “Near linear time algorithm to detect community structures in large-scale networks”. In: *Phys. Rev. E* 76 (3 Sept. 2007), page 036106. DOI: 10.1103/PhysRevE.76.036106.

[Ros24] D. Rosch. “Sparsification for Linear Time Multilevel Graph Partitioning”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2024.

[RPG96] E. Rolland, H. Pirkul, and F. W. Glover. “Tabu search for graph partitioning”. In: *Ann. Oper. Res.* 63.2 (1996), pages 209–232. DOI: 10.1007/BF02125455.

[Sal+25] D. Salwasser, D. Seemaier, L. Gottesbüren, and P. Sanders. “Tera-Scale Multilevel Graph Partitioning”. In: *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2025, Milano, Italy, June 3-7, 2025*. IEEE, 2025, pages 285–296. DOI: 10.1109/IPDPS64566.2025.00033.

[Sal24] D. Salwasser. “Optimizing a Parallel Graph Partitioner for Memory Efficiency”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2024.

[San+19] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0.

[Sch+16] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. “k-way Hypergraph Partitioning via n-Level Recursive Bisection”. In: *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*. SIAM, 2016, pages 53–67. DOI: 10.1137/1.9781611974317.5.

[Sch+19] S. Schlag, C. Schulz, D. Seemaier, and D. Strash. “Scalable Edge Partitioning”. In: *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*. SIAM, 2019, pages 211–225. DOI: 10.1137/1.9781611975499.17.

[Sch+22] S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz, and P. Sanders. “High-Quality Hypergraph Partitioning”. In: *ACM J. Exp. Algorithmics* 27 (2022), 1.9:1–1.9:39. DOI: 10.1145/3529090.

[Sch13] C. Schulz. “High Quality Graph Partitioning”. PhD thesis. Karlsruhe Institute of Technology, 2013. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000035713.

[Sch20] S. Schlag. “High-Quality Hypergraph Partitioning”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2020. URL: https://nbn-resolving.org/urn:nbn:de:101:1-202003040358162016576 5.

[SDB15] J. Shun, L. Dhulipala, and G. E. Blelloch. “Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+”. In: *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*. IEEE, 2015, pages 403–412. DOI: 10.1109/DCC.2015.8.

[Shu20] J. Shun. “Practical parallel hypergraph algorithms”. In: *PPoPP ’20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*. ACM, 2020, pages 232–249. DOI: 10.1145/3332466.3374527.

[Sim91] H. D. Simon. “Partitioning of unstructured problems for parallel processing”. In: *Computing Systems in Engineering* 2 (1991), pages 135–148. URL: https://api.semanticscholar.org/CorpusID:16197093.

[SK12] I. Stanton and G. Kliot. “Streaming Graph Partitioning for Large Distributed Graphs”. In: *18th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining, (KDD)*. 2012, pages 1222–1230.

[SKK00] K. Schloegel, G. Karypis, and V. Kumar. “Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning (Distinguished Paper)”. In: *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings*. Volume 1900. Lecture Notes in Computer Science. Springer, 2000, pages 296–310. DOI: 10.1007/3-540-44520-X_39.

[Slo+17] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri. “Partitioning Trillion-Edge Graphs in Minutes”. In: *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 2017, pages 646–655. DOI: 10.1109/IPDPS.2017.95.

[SM13] C. Staudt and H. Meyerhenke. “Engineering High-Performance Community Detection Heuristics for Massive Graphs”. In: *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*. IEEE Computer Society, 2013, pages 180–189. DOI: 10.1109/ICPP.2013.27.

[SM16] C. L. Staudt and H. Meyerhenke. “Engineering Parallel Algorithms for Community Detection in Massive Networks”. In: *IEEE Trans. Parallel Distributed Syst.* 27.1 (2016), pages 171–184. DOI: 10.1109/TPDS.2015.2390633.

[SMR14] G. M. Slota, K. Madduri, and S. Rajamanickam. “PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks”. In: *2014 IEEE International Conference on Big Data (IEEE BigData 2014), Washington, DC, USA, October 27-30, 2014*. IEEE Computer Society, 2014, pages 481–490. DOI: 10.1109/BIGDATA.2014.7004265.

[SMR16] G. M. Slota, K. Madduri, and S. Rajamanickam. “Complex Network Partitioning Using Label Propagation”. In: *SIAM J. Scientific Computing* 38.5 (2016). DOI: 10.1137/15M1026183.

[SS11a] P. Sanders and C. Schulz. “Engineering Multilevel Graph Partitioning Algorithms”. In: *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*. Volume 6942. Lecture Notes in Computer Science. Springer, 2011, pages 469–480. DOI: 10.1007/978-3-642-23719-5_40.

[SS11b] D. A. Spielman and N. Srivastava. “Graph Sparsification by Effective Resistances”. In: 40.6 (2011), pages 1913–1926. DOI: 10.1137/080734029.

[SS12] P. Sanders and C. Schulz. “Distributed Evolutionary Graph Partitioning”. In: *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*. SIAM / Omnipress, 2012, pages 16–29. DOI: 10.1137/1.9781611972924.2.

[SS13] P. Sanders and C. Schulz. “Think Locally, Act Globally: Highly Balanced Graph Partitioning”. In: *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. Volume 7933. Springer, 2013, pages 164–175.

[SS16a] P. Sanders and C. Schulz. “Advanced Multilevel Node Separator Algorithms”. In: *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*. Volume 9685. Lecture Notes in Computer Science. Springer, 2016, pages 294–309. DOI: 10.1007/978-3-319-38851-9_20.

[SS16b] P. Sanders and C. Schulz. “Scalable generation of scale-free graphs”. In: *Inf. Process. Lett.* 116.7 (2016), pages 489–491. DOI: 10.1016/J.IPL.2016.02.004.

[SS23a] P. Sanders and M. Schimek. “Engineering Massively Parallel MST Algorithms”. In: *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*. IEEE, 2023, pages 691–701. DOI: 10.1109/IPDPS54959.2023.00075.

[SS23b] P. Sanders and D. Seemaier. “Distributed Deep Multilevel Graph Partitioning”. In: *Euro-Par 2023: Parallel Processing - 29th International Conference on Parallel and Distributed Computing, Limassol, Cyprus, August 28 - September 1, 2023, Proceedings*. Volume 14100. Lecture Notes in Computer Science. Springer, 2023, pages 443–457. DOI: 10.1007/978-3-031-39698-4_30.

[SS24] P. Sanders and D. Seemaier. “Brief Announcement: Distributed Unconstrained Local Search for Multilevel Graph Partitioning”. In: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2024, Nantes, France, June 17-21, 2024*. ACM, 2024, pages 443–445. DOI: 10.1145/3626183.3660257.

[ST97] H. D. Simon and S. Teng. “How Good is Recursive Bisection?” In: *SIAM J. Sci. Comput.* 18.5 (1997), pages 1436–1445. DOI: 10.1137/S1064827593255135.

[SW13] S. Salihoglu and J. Widom. “GPS: a graph processing system”. In: *Conference on Scientific and Statistical Database Management, SSDBM ’13, Baltimore, MD, USA, July 29 - 31, 2013*. ACM, 2013, 22:1–22:12. DOI: 10.1145/2484838.2484843.

[SWT16] V. Sadhanala, Y. Wang, and R. J. Tibshirani. “Graph Sparsification Approaches for Laplacian Smoothing”. In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*. Volume 51. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pages 1250–1259. URL: http://proceedings.mlr.press/v51/sadhanala16.html.

[TBB] TBB. *Intel Threading Building Blocks*. https://www.threadingbuildingblocks.org.

[Trä06] J. L. Träff. “Direct graph k-partitioning with a Kernighan-Lin like heuristic”. In: *Oper. Res. Lett.* 34.6 (2006), pages 621–629. DOI: 10.1016/J.ORL.2005.10.003.

[Tro+22] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke. “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pages 805–817. DOI: 10.1109/TPDS.2021.3097283.

[Tso+14] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. “FENNEL: streaming graph partitioning for massive scale graphs”. In: *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*. ACM, 2014, pages 333–342. DOI: 10.1145/2556195.2556213.

[Uhl+24] T. N. Uhl, M. Schimek, L. Hübner, D. Hespe, F. Kurpicz, D. Seemaier, C. Stelz, and P. Sanders. “KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2024, Atlanta, GA, USA, November 17-22, 2024*. IEEE, 2024, page 44. DOI: 10.1109/SC41406.2024.00050.

[Wal04] C. Walshaw. “Multilevel Refinement for Combinatorial Optimisation Problems”. In: *Ann. Oper. Res.* 131.1-4 (2004), pages 325–372. DOI: 10.1023/B:ANOR.0000039525.80601.15.

[Wal23] C. Walshaw. *The Graph Partitioning Archive*. https://chriswalshaw.co.uk/partition/. Last updated Mar 27, 2023. Accessed: 2026-01-02. 2023.

[Wan+14] L. Wang, Y. Xiao, B. Shao, and H. Wang. “How to partition a billion-node graph”. In: *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. IEEE Computer Society, 2014, pages 568–579. DOI: 10.1109/ICDE.2014.6816682.

[WC00] C. Walshaw and M. Cross. “Parallel optimisation algorithms for multilevel mesh partitioning”. In: *Parallel Comput.* 26.12 (2000), pages 1635–1660. DOI: 10.1016/S0167-8191(00)00046-6.

[WC07] C. Walshaw and M. Cross. “JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview”. In: *Mesh Partitioning Techniques and Domain Decomposition Techniques*. 2007, pages 27–58. ISBN: 978-1-874672-29-6.

[WCE97] C. Walshaw, M. Cross, and M. G. Everett. “Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes”. In: *J. Parallel Distributed Comput.* 47.2 (1997), pages 102–108. DOI: 10.1006/JPDC.1997.1407.

[WMS09] J. Wassenberg, W. Middelmann, and P. Sanders. “An Efficient Parallel Algorithm for Graph-Based Image Segmentation”. In: *Computer Analysis of Images and Patterns, 13th International Conference, CAIP 2009, Münster, Germany, September 2-4, 2009. Proceedings*. Volume 5702. Lecture Notes in Computer Science. Springer, 2009, pages 1003–1010. DOI: 10.1007/978-3-642-03767-2_122.

[Zha+17] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. “Graph Edge Partitioning via Neighborhood Heuristic”. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 2017, pages 605–614. DOI: 10.1145/3097983.3098033.