Master thesis

Revisiting Deterministic Parallel Hypergraph Partitioning

Robert Krause

Date: 3. Juli 2024

Reviewers:	Prof. Dr. Peter Sanders
	T.T.-Prof. Thomas Bläsius
Advisors:	Dr. Lars Gottesbüren
	M.Sc. Nikolai Maas

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

The *balanced k-way hypergraph partitioning problem* is to partition the vertex set of a hypergraph into k disjoint and non-empty parts of roughly equal size while minimizing an objective function defined on hyperedges. It has many application areas, including VLSI design, simulating quantum circuits, modeling communication volumes in scientific computing, and load balancing in machine learning pipelines. Since hypergraph partitioning is NP-hard, parallel heuristics are used in practice. However, most of the parallel (hyper)graph partitioners are non-deterministic.

The benefits of deterministic parallel algorithms include ease of debugging, reasoning about performance, and, most importantly, reproducibility. Furthermore, the authors of BiPart argue that in VLSI design, deterministic results are even necessary to avoid repeating manual post-processing steps [40]. However, the few existing parallel deterministic partitioners are significantly behind their non-deterministic counterparts regarding solution quality. This has become even more severe since the recent emergence of unconstrained refinement algorithms.

This work fully closes the gap between deterministic and non-deterministic hypergraph partitioning algorithms regarding solution quality. These algorithms are all based on the *multilevel* paradigm, which consists of three phases: In the first phase, the size of the hypergraph is reduced by contracting highly connected vertices. Afterward, an initial partition is computed. In the subsequent refinement phase, the contractions are reversed while improving the solution. Our main contributions include a deterministic version of the Jet refinement algorithm, which is also the first version designed to work on hypergraphs. Additionally, we further improve the state-of-the-art of deterministic coarsening and present all algorithmic components needed to achieve fully deterministic flow-based refinement (due to a bug, our flow refinement implementation is not fully deterministic, yet). We integrate our work into the parallel shared-memory hypergraph partitioning framework Mt-KaHyPar and perform extensive experiments on large hypergraph and graph instances.

Our algorithm matches the quality of Mt-KaHyPar-D on hypergraphs. For graphs, our algorithm even finds the best solution for 70% of the instances, thereby matching or surpassing the quality of FM refinement using the Jet algorithm. However, this improved quality comes at the cost of significantly increased running time. Our deterministic implementation, on average, takes 75% more time than Mt-KaHyPar-D. Compared to other state-of-the-art partitioners, we achieve superior quality. However, our algorithm is slower than all parallel partitioners except for BiPart, which takes roughly three times as long.

Comparing the geometric mean performance of our deterministic flow-based refinement scheme to Mt-KaHyPar-Q, the solutions found by our algorithm are within 3% and 2% for hypergraphs and graphs, respectively. Due to more limited parallelism, our algorithm is roughly 21% slower than its non-deterministic counterpart.

Zusammenfassung

Das *balancierte k-Wege Hypergraph-Partitionierungsproblem* besteht darin, die Knotenmenge eines Hypergraphs in k disjunkte und nicht-leere Teile von ungefähr gleicher Größe zu unterteilen, wobei eine Zielfunktion auf Hyperkanten minimiert wird. Es hat viele Anwendungsgebiete, darunter VLSI-Design, Simulation von Quanten-Schaltungen, Modellierung von Kommunikationsvolumen in der wissenschaftlichen Datenverarbeitung und Lastverteilung in Machine- Lerning-Pipelines. Da die Hypergraph-Partitionierung NP-schwer ist, werden in der Praxis parallele Heuristiken verwendet. Allerdings sind die meisten parallelen (Hyper-)Graph-Partitionierer nicht deterministisch.

Die Vorteile deterministischer paralleler Algorithmen umfassen die Vereinfachung der Fehlersuche und vor allem die Reproduzierbarkeit. Darüber hinaus argumentieren die Autoren von `BiPart`, dass im VLSI-Design deterministische Ergebnisse sogar notwendig sind, um die Wiederholung manueller Nachbearbeitungsschritte zu vermeiden [40]. Allerdings sind die wenigen existierenden parallelen deterministischen Partitionierer den nicht-deterministischen Algorithmen, in Bezug auf die Lösungsqualität, deutlich unterlegen. Dies hat sich seit dem jüngsten Aufkommen von *unconstrained Refinement*-Algorithmen noch verschärft.

Mit dieser Arbeit schließen wir die Lücke zwischen deterministischen und nicht-deterministischen Hypergraph-Partitionierungsalgorithmen hinsichtlich der Lösungsqualität vollständig. Diese Algorithmen basieren alle auf dem *Multilevel*-Paradigma, das aus drei Phasen besteht: In der ersten Phase wird die Größe des Hypergraphs durch Kontraktion stark verbundener Knoten reduziert. Anschließend wird eine initiale Partition berechnet. Im *Refinement* werden die Kontraktionen rückgängig gemacht und die Lösung verbessert. Unsere Hauptbeiträge umfassen eine deterministische Version des Jet-Verfeinerungsalgorithmus, der auch die erste Version ist, die für Hypergraphs entwickelt wurde. Zusätzlich verbessern wir den Stand der Technik der deterministischen *Coarsening*-Algorithmen und präsentieren alle algorithmischen Komponenten, die erforderlich sind, um vollständig deterministisches flussbasiertes *Refinement* zu erreichen (aufgrund eines Fehlers ist unsere Implementierung, ist diese jedoch noch nicht vollständig deterministisch). Wir integrieren unsere Arbeit in das parallele Shared-Memory-Hypergraph-Partitionierungsframework `Mt-KaHyPar` und führen umfangreiche Experimente an großen Hypergraph- und Graphinstanzen durch.

Die Qualität unseres Algorithmus is equivalent ui `Mt-KaHyPar-D` auf Hypergraphen. Für Graphen findet unser Algorithmus sogar die beste Lösung für 70% der Instanzen und erreicht oder übertrifft damit die Qualität des FM-Refinement Algorithmus unter Verwendung des Jet-Algorithmus. Diese verbesserte Qualität geht jedoch auf Kosten einer erheblich verlängerten Laufzeit. Unsere deterministische Implementierung benötigt im Durchschnitt 75% mehr Zeit als `Mt-KaHyPar-D`. Im Vergleich zu anderen modernen Partitionierern finden wir Lösungen mit deutlich besserer Qualität. Unser Algorithmus ist jedoch langsamer als alle parallelen Partitionierer, mit Ausnahme von `BiPart`, welcher ungefähr

dreimal so viel Zeit benötigt.

Vergleicht man die geometrische Mittelwertleistung unseres deterministischen flussbasierten Refinements mit `Mt-KaHyPar-Q`, so liegen die Lösungen unseres Algorithmus innerhalb von 3% und 2% für Hypergraphen bzw. Graphen. Aufgrund der begrenzteren Parallelität ist unser Algorithmus etwa 21% langsamer als sein nicht-deterministisches Gegenstück.

Acknowledgments

I want to thank my supervisors, Lars Gottesbüren and Nikolai Maas, for all their patience and guidance throughout my work on this thesis. Our long and productive discussions were essential to staying motivated and maintaining constant progress toward finishing this work. Without their excellent mentorship and feedback, this work would not have been possible.

Contents

1 Introduction

Hypergraphs are a generalization of graphs, where each hyperedge connects an arbitrary number of vertices. The balanced k-way hypergraph partitioning problem is the corresponding generalization of the graph partitioning problem. The goal is to partition the vertex set of the hypergraph into k disjoint and non-empty parts of roughly equal size while minimizing an objective function defined on hyperedges that connect more than one part. The most commonly used metric is called *connectivity* (or $\lambda - 1$). Here, we aim to minimize the number of blocks each hyperedge connects. The result must satisfy the balance constraint that each part is smaller than $(1 + \epsilon)$ times the average part size.

Hypergraph partitioning is applicable to a variety of different problem domains. In VLSI, hypergraph partitioning is used to partition a circuit into smaller modules such that the length of connecting wires is minimized [35, 6]. It is also used in simulations of distributed quantum circuits [26, 2]. Another application area is scientific computing, where hypergraphs model communication volumes for sparse matrix multiplications. Here, hypergraphs are a more suitable model than graphs due to their ability to represent non-symmetric problems [11]. Further applications involve storage sharding in distributed databases [34, 9], SAT-Solving [14, 13] and load balancing in machine learning pipelines [8].

Since hypergraph partitioning is NP-hard [38, 15] and even hard to approximate within constant factors [5], heuristics are used in practice to keep up with the ever-growing problem sizes of modern applications. The most widespread heuristic is the *multilevel paradigm*. It works in three phases: In the first phase, a sequence of coarser hypergraphs is calculated via a coarsening algorithm by finding a matching or clustering of highly connected vertices and subsequently contracting them. This phase is called the *coarsening phase*. Its purpose is to reduce the problem size while retaining the underlying structure of the hypergraph so that more sophisticated algorithms can be utilized to calculate a first partition in the *initial paritioning* phase. This is followed by the *refinement phase*, where the partition is improved while undoing the first phase's contractions, allowing for both coarse- and fine-grained improvements of the solution.

Although there is much research in the area of parallel hypergraph partitioning, almost all of the algorithms are non-deterministic. Only `BiPart` [40] and `Mt-KaHyPar-SDet` [23] are deterministic hypergraph partitioners. The benefits of deterministic parallel algorithms include ease of debugging, reasoning about performance, and, most importantly, reproducibility. Furthermore, the authors of `BiPart` argue that in VLSI design, deterministic partitioning results are even necessary due to manual post-processing steps optimizing the placement of non-standard cells that should not be repeated [40].

Although `Mt-KaHyPar-SDet` produces significantly better solutions than `BiPart`, it is still noticeably behind in terms of solution quality compared to state-of-the-art hypergraph partitioners, such as the non-deterministic configuration of `Mt-KaHyPar` [23]. In the case of the deterministic configuration of `Mt-KaHyPar`, the loss of solution quality is especially noticeable in the coarsening and the refinement phase of the algorithm. In the case of the refinement phase, it is due to a lack of deterministic implementations of more sophisticated parallel refinement algorithms. The default non-deterministic configuration of `Mt-KaHyPar` uses an unconstrained parallel version of the FM local search [39], the state-of-the-art deterministic partitioner `Mt-KaHyPar-SDet`, on the other hand, uses a more simple label propagation algorithm. Furthermore, the quality configuration of `Mt-KaHyPar` employs the most powerful refinement technique: flow-based refinement, for which there is no deterministic version. Because of this, the gap in partition quality between deterministic and non-deterministic partitioners is quite large. This gap increased even further with the recent introduction of unconstrained refinement algorithms to the non-deterministic configuration of `Mt-KaHyPar` [39]. These algorithms allow the refinement to escape local minima and further increase solution quality, especially on irregular instances. In this work, we aim to close this gap, leveraging the techniques of unconstrained and flow-based refinement.

1.1 Problem Statement

In this thesis, we aim to close the gap between deterministic and non-deterministic hypergraph partitioning in terms of solution quality. To this end, the promising Jet refinement algorithm should be implemented in a fully deterministic fashion. Additionally, the algorithm should be adapted and optimized to allow its application to large hypergraph instances.

In the second step, the differences, in terms of solution quality, between the deterministic and non-deterministic coarsening algorithms of the shared-memory (hyper-)graph partitioning framework `Mt-KaHyPar` should be investigated. The deterministic coarsening algorithm should be improved to produce equally good solutions as the non-deterministic algorithm.

Furthermore, the flow-based refinement of Mt-KaHyPar should be adapted to produce deterministic solutions. This includes a simple deterministic parallel scheduling scheme, changes to the network construction as well as the FlowCutter algorithm.

The work should be integrated into `Mt-KaHyPar` with the goal to match the quality of the non-deterministic configuration `Mt-KaHyPar-D` and `Mt-KaHyPar-Q` for the Jet and flow-based refinement, respectively, without compromising too much on running time.

1.2 Contribution

In this work, we present, to the best of our knowledge, the first fully deterministic version of the Jet algorithm for graph refinement. Additionally, we present several changes and optimizations to make the algorithm viable for hypergraph partitioning. Secondly, we investigate the shortcomings of the deterministic coarsening algorithm of Mt-KaHyPar in comparison to its non-deterministic counterpart. In the course of this thesis, we discuss the improvements and corrections we made to the deterministic algorithm to fully close the gap in solution quality between non-deterministic and deterministic coarsening.

Furthermore, we present a deterministic parallel scheduling scheme, deterministic network construction and tweaks to the FlowCutter algorithm with the aim of deterministic flow-based refinement. Unfortunately, we were not quite able to achieve full determinism because of a bug we are still chasing. To our current understanding the non-determinism is limited to the results of the FlowCutter procedure.

We integrate our algorithms into the shared-memory (hyper-)graph partitioning framework Mt-KaHyPar, which also has a deterministic configuration. We compare our new refinement algorithm and the improvements to the coarsening algorithm to the deterministic configuration Mt-KaHyPar-SDet and the stronger non-deterministic configuration Mt-KaHyPar-D on a large and diverse benchmark set of hypergraphs and graphs. Additionally, we compare our deterministic hypergraph partitioning approach to different state-of-the-art hypergraph and graph partitioners. Furthermore, we compare our flow-based approach to its non-determinisitc counterpart Mt-KaHyPar-Q, as it employs the only other parallel flow based refinement algorithm.

With the algorithms presented in this thesis, the whole repertoire of refinement algorithms of label propagation, FM and flow-based is now also available for deterministic (hyper)graph partitioning. Assuming that the Jet algorithm is able to replace FM.

The results for our configuration using deterministic Jet show that we could not only fully close the gap to the non-deterministic algorithm regarding solution quality, but we even consistently find better solutions on graphs. Our deterministic partitioning approach performs equally on hypergraphs and finds the best solution for 65% and 75% of the irregular and regular graphs, respectively. Compared to the state-of-the-art deterministic hypergraph partitioner Mt-KaHyPar-SDet, our algorithm finds the best solution on almost all of our test instances. On average, the solutions found by our algorithm are better by 12%, 42%, and 11% on hypergraphs, irregular and regular graphs, respectively. The improved solution quality, however, comes at the price of larger running times. Our implementation, on average, takes around 75% more time than Mt-KaHyPar-D to compute the partition and is slower than Mt-KaHyPar-SDet by a factor of roughly 2.

Compared to state-of-the-art graph partitioner `KaMinPar`, we achieve significantly better solution quality, finding the best solution for 95% of the instances. Furthermore, the solutions found by our algorithm are better by 10% or more for more than half of the instances. However, `KaMinPar` is much faster, taking only a fraction of the running time.

Furthermore, we outperform sequential hypergraph partitioner `PaToH-D` both in terms of solution quality and running time. We achieve a geometric mean performance ratio of 1.006 compared to 1.425 of `PaToH-D` while being roughly 6.1 times faster. Additionally, we outperform the only other deterministic hypergraph partitioner `BiPart` in terms of solution quality and running time. We find the better solution for all but two instances, and the quality of our results is better by a factor of 2 for roughly 60% of the instances while only taking less than a third of the running time on average.

Our implementation of the flow-based refinement scheme comes very close to the quality and running time of its non-deterministic counterpart `Mt-KaHyPar-Q`. Comparing the geometric mean performance, partitions found by our algorithm are within 3% and 2% of `Mt-KaHyPar-Q` on hypergraph and regular graph instances, respectively. Due to more limited parallelism, our algorithm is roughly 21% slower than its non-deterministic counterpart.

1.3 Outline

We introduce basic concepts and the necessary notation in Section 2. Section 3 summarizes related work concerning hypergraph partitioning and the Jet algorithm. We present our deterministic version of the Jet refinement algorithm (Mt-KaHyPar-DetJet) with all its optimizations and changes to make it applicable to hypergraphs in Section 4. Afterward, in Section 5, we discuss improvements to the deterministic coarsening algorithm of `Mt-KaHyPar`. We present the experimental evaluation of our algorithms in Section 6 and conclude our work with a summary of the most important findings and possible future directions in Section 7.

2 Preliminaries

In this chapter, we first introduce some basic notation regarding hypergraphs and graphs in Section 2.1 which we use throughout the work. Afterwards we introduce important concepts of hypergraph partitioning in Section 2.2.

2.1 Hypergraphs and Graphs

An *undirected weighted* hypergraph $H = (V, E, c, \omega)$ consists of a set of n *hypernodes* V and m *hyperedges* E, as well as the *vertex weights* $c : V \to \mathbb{R}_{\geq 0}$, and the *edge weights* $\omega : E \to \mathbb{R}_{>0}$. We extend these weights to sets, i.e. for $V' \subseteq V$ and $E' \subseteq E$ we define $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. We denote $W = \omega(E)$ to be the sum of all edge weights. A hyperedge $e \in E$ is defined as a non-empty sub-set of hypernodes. We call a single occurrence of a hypernode $v \in e$ a *pin*. The size of e is its cardinality $|e|$. We define $\Phi(e, v)$ for hyperedge $e \in E$ and node $v \in V$ as the pin count of $v \in e$. We extend this definition to sets of nodes. For the subset $C \subseteq V$ the cumulative pin count of C in e is defined as $\Phi(e, C) = \sum_{v \in C} \Phi(e, v)$.

A vertex $v \in V$ is *incident* to a hyperedge e if $v \in e$. $I(v)$ denotes the set of incident hyperedges of v, which we extend to sets of nodes as the set of unique edges with $\Phi(e, C) > 0$, in other words $I(C) = \{e \in E \mid \Phi(e, C) > 0\}$. A hypergraph $H' = (V', E')$ is called a *subgraph* of $H = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A graph, also called *dyadic* graph, is a special case of a hypergraph, where all edges are exactly of size two. Therefore all definitions from above also apply to them.

2.2 Hypergraph Partitioning

In this section, we first define what a *k-way partition* is and then introduce the *k-way hypergraph partitioning problem* which we try to solve deterministically, in this work. Finally, we provide a concise introduction to vertex moves.

k-way Partition. A *k-way partition* of a hypergraph H is a partition of its vertex set into *k disjoint blocks* $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset \ \forall i \in \{1, \ldots, k\}$. A *k-way* partition is *ϵ-balanced* if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq L_{max} := (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$. We define the *connectivity set* $\Lambda(e) := \{V_i \mid \Phi(e, V_i) > 0\}$

for each edge $e \in E$ and its cardinality $\lambda(e) := |\Lambda(e)|$ is called the *connectivity* of a hyperedge e. A hyperedge e, whose nodes are in more than one block and therefore with connectivity $\lambda(e) > 1$, is called *cut hyperedge* and $E(\Pi) := \{e \in E \mid \lambda(e) > 1\}$ denotes the set of all cut hyperedges. We call two blocks V_i and V_j *adjacent* if they are connected by any cut-hyperedges, i.e. the set of cut hyperedges $E(V_i, V_j) = \{e \in E \mid \{V_i, V_j\} \subseteq \Lambda(e)\}$ is not empty. The *quotient graph* $\mathcal{Q} := (\Pi, E_\Pi := \{(V_i, V_j) \mid E(V_i, V_j) \neq \emptyset\})$ consist of the set of blocks Π as vertices connected by an edge between all adjacent blocks.

k-way Hypergraph Partitioning Problem. The *k-way hypergraph partitioning problem* is to find an ϵ-balanced k- way partition of a hypergraph $H = (V, E, c, \omega)$ that minimizes an objective function for a given $\epsilon \geq 0$. The two most popular objective functions are the *cut-net* metric, which sums the weights of all cut hyperedges: $cut(\Pi) := \sum_{e \in E(\Pi)} \omega(e)$ and the *connectivity*(also km1- or $(\lambda - 1)$-) metric: $(\lambda - 1)(\Pi) := \sum_{e \in E(\Pi)} (\lambda(e) - 1)\omega(e)$. The latter one additionally takes into account the number of different blocks cut by each edge. While both metrics revert to the edge-cut metric on dyadic graphs, we will, in this work, focus on the connectivity metric since it is applicable on both hypergraphs and dyadic graphs.

Vertex Moves. Let $\Pi(v) := V_i$ be the block that contains v. A move m removes a vertex v from its original block $\Pi(v)$ and adds it to a different target block, thereby creating a new partition Π'. We define the gain of a move as gain$(m, \Pi) := (\lambda - 1)(\Pi) - (\lambda - 1)(\Pi')$. Let Π'' be the partition after a set of moves M was applied to partition Π, we then extend the definition of gain to set as gain$(M, \Pi) := (\lambda - 1)(\Pi) - (\lambda - 1)(\Pi'')$.

3 Related Work

In practice, heuristic algorithms are used for hypergraph partitioning since the problem is known to be NP-hard [38, 15]. We will first present the *multilevel paradigm*, which is the most successful approach used in many state-of-the-art-partitioners [35, 46, 22], including the `Mt-KaHyPar` framework which we introduce afterwards.

3.1 The Multilevel Paradigm

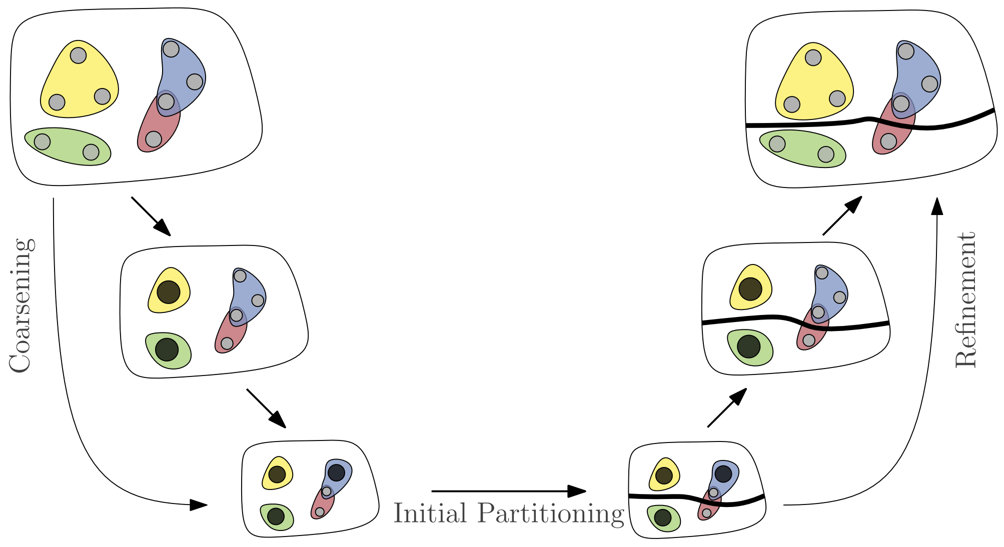

Figure 3.1: Partitioning a Hypergraph via the multilevel paradigm [36].

The multilevel paradigm (see Figure 3.1) consists of three phases: the *coarsening* phase, the *initial partitioning* phase, and the *uncoarsening/refinement* phase. The goal of the first phase is to provide a sequence of successively smaller hypergraphs while maintaining its structural similarity. This is achieved by contracting clusters of vertices of the hypergraph, which is done until the size of the coarsest hypergraph reaches a predefined contraction limit. The second phase, the initial partitioning phase, computes a k-way partition of the smallest hypergraph. This can be done using more sophisticated algorithms, which provide better quality since the hypergraph is small. The final phase, the uncoarsening or refinement phase, successively projects the partition onto the next finer hypergraph in the hierarchy. Moreover, at each uncoarsening step, the solution is refined using local search heuristics.

3.2 The Mt-KaHyPar Partitioning Framework

The `Mt-KaHyPar` framework is a shared-memory multilevel hypergraph partitioning system that provides a variety of different configurations with different quality-runtime trade-offs [24, 23]. Alongside traditional multilevel partitioning, it supports configurations for n-level partitioning. Here, only a single node is contracted on each level, providing a much more fine-grained approach. Additionally, there are special configurations for graph partitioning, which speed up calculations on dyadic graphs utilizing data structure optimizations. Since this work is about deterministic (hyper)graph-partitioning, we will now look closer at the framework's deterministic configuration [23].
`Mt-KaHyPar` uses an additional preprocessing phase, originally proposed in the sequential hypergraph partitioner `KaHyPar`. In this preprocessing phase, community detection is used to identify densely connected regions of the hypergraph, which are then used in the coarsening phase to restrict contractions to vertices within the same community [32]. Both the coarsening and refinement phase of the deterministic configuration utilize so-called local moving algorithms. Given an initial assignment of vertices to clusters, these algorithms iterate over all nodes in random order and make greedy, local decisions to move a vertex to a different cluster. In the deterministic case, these moves can not be performed immediately; otherwise, the results would depend on non-deterministic scheduling decisions. Instead, all local moving algorithms follow the `synchronous local moving` approach by Hamann et al. [29]. Here, vertices are deterministically split into sub-rounds. We calculate the best move for all corresponding vertices in each sub-round based on the unchanged state before the sub-round. In the second step, some of the moves are performed, and others are denied according to the constraints of the algorithm.
In each coarsening pass, the algorithm performs one round of local moving, consisting of three sub-rounds, and then contracts the calculated clusters. Coarsening is performed until the contraction Limit $CL := 160 \cdot k$ is reached. In the case of the coarsening phase, the preferred target clusters are calculated based on the heavy-edge rating function $r(u, v) = \sum_{e \in I(u) \cap I(v)} \frac{\omega(e)}{|e|}$. The aggregated moves of a sub-round are then approved in accordance with the maximum cluster weight $CW_{max} := \min(L_{max}, c(V)/CL)$. If approving all moves would result in a cluster that violates the said constraint, the algorithm approves nodes one by one after sorting them deterministically. This is parallelized by iterating over all moves in parallel. By making the iteration of the first vertex of a sub-range of a cluster responsible for all moves into the cluster, there is only ever one thread handling the approval of moves for each cluster. The initial partitioning is performed via multilevel recursive bipartitioning.
To compute an initial bipartition of the coarsest hypergraph, `Mt-KaHyPar` uses a portfolio of initial bipartitioning algorithms for a detailed overview, we refer to the original papers

[47, 23].
As previously mentioned, the refinement phase is also based on synchronous local moving. The refinement algorithm is a synchronous version of the label propagation refinement [41, 25]. Here, vertex moves for a sub-round are calculated based on the gain of moving the vertex $v \in V$ from its current block to its preferred target block. To maintain a balanced solution, not all moves can be approved. Instead, based on calculated gains, the algorithm performs a sequence of balance-preserving vertex swaps on each block pair.
In addition to label propagation, the non-deterministic configurations of `Mt-KaHyPar` utilize additional refinement algorithms to boost solution quality. The default configuration `Mt-KaHyPar-D` uses an unconstrained parallel version of the FM local search [16]. It improves upon the shared-memory parallel FM algorithm [1, 25] by adding an *approximate penalty* to unconstrained moves that represent the cost of rebalancing afterward. After rebalancing, the algorithm combines unconstrained and rebalancing moves into a single sequence and applies its best prefix to the partition [39].

The quality preset `Mt-KaHyPar-Q` adds flow-based refinement based on the *FlowCutter* algorithm [20]. It works by solving flow problems on block pairs and is able to improve cuts in a more global manner than the previously discussed local search algorithms [18]. To improve a k-way partition multiple block pairs are scheduled in parallel. They use a parallelization of the active block scheduling of Sanders and Schulz [44], where initially, all blocks are active. All block pairs containing at least one active block are scheduled in each round. A block is marked as active for the next round if an improvement for a flow problem containing that block is found. The scheduling policy used by Gottesbüren et al. is highly non-deterministic since it allows the scheduling of block pairs from future rounds. The resulting conflicts are highly dependent on the timing of computations on block pairs of earlier rounds. The additional heuristic of aborting long- running flow computations is also inherently non-deterministic.

3.3 The Jet Refinement Algorithm

In their recent work, Gilbert et al. [17] introduce the refinement algorithm called `Jet`. The algorithm is designed for graph partitioning on graphical processing units and achieves partition quality similar to FM refinement. The Jet algorithm consists of two parts: a modified unconstrained label propagation, which moves all nodes synchronously, followed by a rebalancing algorithm.

Selecting Move Candidates. The algorithm starts by building a set of move candidates M in the following way: Given vertex $v \in V$ and cluster V_i, then $\text{conn}(v, V_i) := \sum_{u \in V_i} \omega(u, v)$ is the summed edge-weight between v and the vertices in V_i. Let $V_d(v) = \text{argmax}_{V_i} \text{conn}(v, V_i)$ denote the cluster with the strongest connection to vertex v. Then the algorithm considers the vertex v a move candidate if $\text{conn}(v, V_d(v)) - \text{conn}(v, V_s) \geq$

$-\lfloor \tau \cdot \mathrm{conn}(v, V_s) \rfloor$. The constant parameter τ regulates the selection of vertices whose possible moves all have negative gain, i.e., for $\tau = 1$, all vertices are considered, and for $\tau = 0$, only vertices with positive gain are considered.

Afterburner. The gain for candidate $v \in M$ is calculated assuming u is moved to $V_d(u)$ if u is a neighbor of v and u precedes v in the ordering. The afterburner then filters out all moves with non-negative gain. Moved vertices are locked for the next round to avoid oscillation. Since the unconstrained label propagation allows balance-violating moves, a rebalancing algorithm fixes the balance constraint afterward.

Rebalancing. The proposed rebalancer of Gilbert et al. consists of two parts itself. The first rebalancer, they call *weak* (*Jetrw*), moves vertices from oversized clusters to clusters outside of a *dead-zone*. In one pass of this algorithm, the best possible move to a valid neighboring cluster is calculated for all vertices in oversized clusters. Afterward, the moves for each oversized cluster are sorted into buckets according to their gain. For each cluster, the first p moves are executed so that the cluster is not overweight afterward. Because of the bucketing, rather than sorting, this can lead to a suboptimal selection of moves when the moves of a bucket are only executed partly. To prevent once over-sized parts from becoming over-sized in the same rebalancer call again, they introduce a *dead-zone*. If the size of a cluster is within the deadzone, it is not considered a valid target for vertex moves when rebalancing.

The second rebalancing algorithm the authors call *strong* (*Jetrs*) since it guarantees that no oversized clusters remain after a single iteration for graphs with unit weight vertices. In this algorithm, the vertices to be removed from their oversized cluster are not based on their best move. Instead, the mean gain to all valid neighboring clusters is used to make decisions. Vertices are then assigned to valid clusters so that the size of each cluster is as close to the deadzone as possible. This *strong* algorithm is significantly worse than *Jetrw* at retaining solution quality. Therefore, the authors suggest a combination of both, consisting of two iterations of *Jetrw* followed by one iteration of *Jetrs*. If necessary, the remaining imbalance is fixed by further iterations of *Jetrw*.

Non-Determinism. The authors also state a few sources of non-determinism, namely in their data structure for managing connectivity between vertices and clusters and when inserting vertices into buckets in the rebalancing algorithm. These race conditions can lead to different results in the label propagation as well as in the rebalancing part of the algorithm [17].

4 Deterministic Refinement

The goal of the refinement phase is to improve an existing k-way partition, which we get from either the previous level or the initial partitioning. We do this by moving vertices to different parts, based on the improvement in the objective function, we call *gain*. Previous appraoches to deterministic refinement are limited to relatively simple refinement algorithms such as label propagation [23]. In this chapter we present deterministic versions of two of the most powerful refinement techniques: In Section 4.1, we adapt the Jet algorithm by Gilbert et al. [17], which has been shown to achieve partition quality similar to FM refinement. Afterward, in Section 4.2, we present all algorithmic components to achieve deterministic flow-based refinement.

4.1 Deterministic Jet

In this section, we first review the Jet algorithm by Gilbert et al. [17] and then adapt it to work on hypergraphs and show how to make it deterministic. It is originally designed as a refinement algorithm for dyadic graphs on graphical processing units. Because of its design as a synchronous algorithm, it naturally lends itself as a deterministic algorithm for CPUs as well. First, we give a high-level overview of the algorithm. Then, we describe each part of the algorithm in more detail in subsequent paragraphs.

The experiments demonstrating the effectiveness of our optimizations are performed on benchmark sets consisting of 94 large hypergraphs and 38 and 34 large irregular and regular graphs, respectively. We run these experiments on a machine consisting of two Intel Xeon E5-2683 16-core processors clocked at 2.1 GHz using $t = 64$ threads.

Algorithm Overview. Algorithm 1 gives a high-level overview of a single pass of the algorithm. The core of the algorithm is made up of the following steps: First, we calculate a set of move candidates, their preferred target part, and the respective isolated gain if this move would be executed. To possibly escape local minima, vertex moves that worsen solution quality, i.e., have negative gain, are also allowed. Which negative gain moves are allowed is controlled by a parameter τ. Increasing τ allows moves with larger negative gains, whereas $\tau = 0$ only allows moves with non-negative gains. Unlike non-deterministic algorithms, we can not immediately move vertices in parallel and use the updated information to decide on moves. These updates are highly dependent on scheduling and, therefore, not deterministic.

Algorithm 1: Jet Pass

```
1 $M \leftarrow$ computeMoveCandidates
2 $M' \leftarrow$ Afterburn($M$)
3 ExecuteMoves($M'$)
4 if *imbalanced* then
5     Rebalance
6 if *Solution improved* then
7     store as best partition
8 else
9     iterations without improvement++
```

Instead, Gilbert et al. suggest a filtering step they call the `afterburner`. The idea behind this step is to try to predict which moves will be executed. Based on an ordering of the moves, we recalculate the gains of moves as if all moves appearing earlier in the ordering were already executed. Using these updated gains moves with negative gain are filtered out.

Third, all remaining moves are executed in parallel, without regard for the solution's balance. If the solution becomes unbalanced, we invoke the `Rebalancer` to return to a balanced solution.

Finally, we evaluate the quality of the calculated partition. If it did not improve or improved too little, we increment the counter for iterations without improvement but keep the partition in its current state. If the quality of the current partition is better than the best partition we have seen yet, we store it as our best partition. We repeat these passes until $I_{wi} = 8$ consecutive iterations did not improve the solution. We then revert to the stored partition with the best solution quality. As suggested by Sanders and Seemaier [45] we repeat this core algorithm multiple times for different values of τ to further improve the quality. We start with a relatively large value of 0.75, allowing most moves, and in the last round, end up with a value of 0 for τ to only allow moves that do not worsen the solution quality.

Calculating the move candidates is straightforward: We perform a round of synchronous local moving [29] by iterating over all vertices of the hypergraph in parallel and calculating its preferred target cluster and isolated gain. We maintain the results as tuples in an array of length $|V|$ indexed by the vertex-ID. A vertex is only a viable move candidate if all three of the following conditions are fulfilled. First, we restrict vertex moves to border nodes to reduce the number of move candidates per iteration. Additionally, vertices that have been moved in a previous round are not allowed to be moved to avoid oscillation. We maintain this information in another sparse array. Finally, a move is only a viable candidate if it does not worsen the solution quality too much, i.e., the move is allowed to worsen solution quality by at most τ times the worst case of moving the vertex to a non-adjacent part. Given vertex $v \in V$ and its current cluster V_s, then $g_i = \omega(\{e \in I(v) \mid \Phi(e, V_s) > 1\})$ is the gain of moving v to a non-adjacent part. Let V_d denote the calculated target cluster for vertex v. Then the algorithm considers the vertex v a move candidate if $g_i - \omega(\{e \in I(v) \mid \Phi(e, V_d) \geq 1\}) \geq -\lfloor \tau \cdot g_i \rfloor$. We collect the vertex IDs of viable move candidates in a dense vector M.

4.1.1 Hypergraph Afterburner

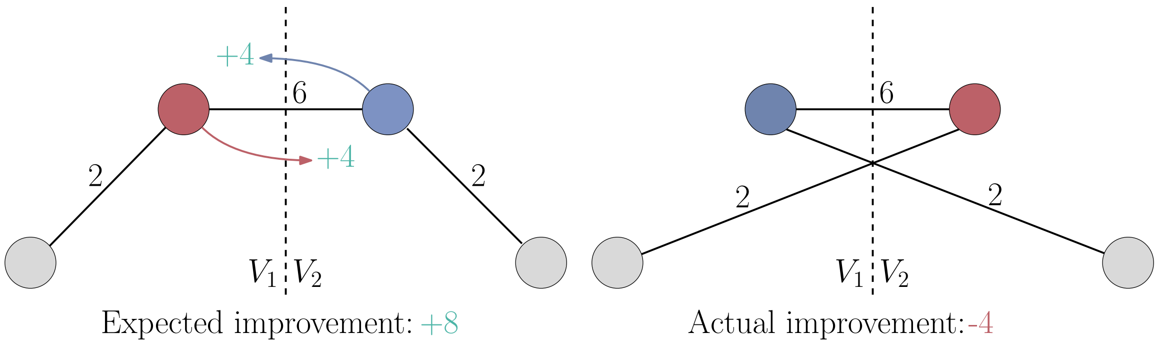

Figure 4.1: Example of two vertex moves influencing each other, resulting in negative improvement.

The Jet algorithm [17] uses the so-called `afterburner` to estimate the sequential move order of FM in the current iteration of the algorithm and prune move candidates accordingly. This is valuable information since different vertex moves influence each other. Moves may look good in isolation but may worsen the solution quality when combined with other moves from the iteration. Figure 4.1 illustrates one such conflict. In isolation, moving the blue and the red vertex each improves the solution quality by 4. However, as we can see on the right, if both moves are applied without accounting for the other, the resulting cut is worsened by 4 since both edges with weight 2 become cut edges while the heavy edge also remains a cut edge. The afterburner is designed to mitigate this effect. To this end, we calculate the gain of each vertex move from the candidate list according to the ordering proposed by the authors. A move candidate u precedes candidate v in the ordering if the gain moving u is larger than the gain moving v. In contrast to dyadic graphs, the naive approach depicted in Algorithm 2 is very inefficient on hypergraphs. For dyadic graphs, the algorithm visits each incident edge to a candidate vertex in M at most two times, resulting in a worst-case complexity of $\mathcal{O}(|E|)$. For hypergraphs, the inner-most loop can become very expensive because we iterate over all $|e|$ pins of an edge e up to $|e|$ times. In this chapter, we present an improved algorithm and techniques to improve the running time in practice. We were able to significantly reduce the work per edge e from the initial $\mathcal{O}(|e|^2)$ to only $\mathcal{O}(|e| \log |e|)$, thereby enabling the algorithm to be applied to large hypergraphs.

Figure 4.2 compares the refinement time with the naive and improved hypergraph afterburner algorithm on the set of large hypergraphs B_{hg}. As we can see, we reduce the running

Algorithm 2: HYPERGRAPH-AFTERBURNER (NAIVE)

```
Input: Hypergraph $H = (V, E, \omega)$, move candidates $M$
Output: Filtered vector of move candidates $M'$
$M' \leftarrow \emptyset$
forall vertices $v \in M$ do in parallel
    forall $e \in I(v)$ do
        $\Phi(e, V_s(v)), \Phi(e, V_d(v)) \leftarrow 0$            // recalculate pin-counts
        forall pins $u \in e, u \neq v$ do                          // according to move order
            if $gain(u, V_s(u)) < gain(v, V_d(v))$ then
                $p \leftarrow V_d(u)$
            else
                $p \leftarrow V_s(u)$
            if $p = V_s(v)$ or $p = V_d(v)$ then
                $\Phi(e, p)$++
        if $attributedGain(\Phi(e, V_s(v)), \Phi(e, V_d(v))) \geq 0$ then
            add $v$ to $M'$
```

time of the refinement phase by a third on average. More importantly, we significantly reduce the running time for the slowest instances, e.g., the refinement phase for the previously slowest instance "sat14_velev-npe-1.0-9dlx-b71.cnf.dual.hgr" now only takes 43 s instead of the previous 8624 s. This is a speedup of roughly 200, illustrating the effectiveness of our optimizations.

Algorithm Overview. To efficiently compute the changes in solution quality for each vertex move in the afterburner (afterburner gain), we want to get rid of the quadratic factor of the edge size. To achieve this, we effectively invert the two outer-most loops from Algorithm 2. Because of this, we regard each hyperedge only once, which is the main reason for the improved running time of our algorithm. The basic idea, illustrated in Algorithm 3, is as follows: We iterate over the edges of the hypergraph and accumulate the afterburner gains for each move candidate of M. Let $e \in E$ be a hyperedge. For each pin $v \in e$, we compute the contribution of e to v's gain based on the presumed vertex move order. We reconstruct the pin-count-in-part values $\Phi(e, V_i)$ for all blocks V_i over the course of the move sequence. We implement this by sorting the pins according to the move order and stepping through them sequentially. To do this, we first materialize the hyperedge by copying each pin into a buffer. To compute the actual afterburner gain, we use attributed gains [25], for which we only need the pin count of the hyperedge in the source and destination part. We keep track of these pin counts of the hyperedge in each part in a buffer of size k. The initial

Algorithm 3: HYPERGRAPH-AFTERBURNER (improved)

```
Input: Hypergraph $H = (V, E, \omega)$, Move candidates $M$
forall $v \in M$ do in parallel
    forall $e \in I(v)$ *once* do
        if $|e| = 2$ then
            return hardcoded-afterburn($e$)
        initialize $\Phi(e, V_i)$ for $i = 0, \dots k - 1$        // thread local memory
        sort pins of $e$ according to the move order
        forall *pin* $u \in e$ do
            $\Phi(e, V_s(u))$- -
            $\Phi(e, V_d(u))$++
            gain $\leftarrow$ attributedGain($\Phi(e, V_s(u))$, $\Phi(e, V_d(u))$) if *gain* $\neq 0$ then
                afterburnerGain[$u$] += gain        // atomic
```

pin counts represent the state where no vertex has been moved. This allows us to calculate the afterburner gain for the vertex with the highest priority, i.e., the vertex that promises the largest improvement. Afterward, we update the pin counts to represent this first vertex move. This has the effect that the gain calculation for the next pin in the order is under the assumption that the first vertex moved to its desired destination part. Consequently, all gain calculations made this way assume that all vertices with higher priority, i.e., lower in the ordering, have already been moved, which is precisely what we wanted. In the improved algorithm, each edge only contributes $\mathcal{O}(|e| \log |e|)$ to the running time, which is a massive improvement from the initial $\mathcal{O}(|e|^2)$.

Parallelization. Parallelizing this approach is straightforward: We iterate over the edges in parallel. To keep computations correct and independent from each other, we have to copy the pins of a hyperedge into a thread-local buffer before sorting. Additionally, the buffers that keep track of pin counts in parts in a hyperedge also need to be thread local. Finally, we use atomics to accumulate the afterburner gains. To avoid allocations, all buffers are implemented using thread-local scratch memory.

Further Optimizations. To further improve the running time of the algorithm in practice, we design several optimizations. Firstly, instead of iterating over all edges of the hypergraph, it is much more efficient to only iterate over incident edges of move candidates. This is still correct since edges that do not contain any move candidates do not change their contribution to the optimization function.

To ensure that we process each edge only once, we use an array of atomic flags to flag already processed edges in parallel.

We reduce the copying overhead and running time in all subsequent steps by not copying pins into the edge buffer if the vertex is not in the candidate set. The parts of these pins are constant throughout the whole computation and, therefore, only need to be added to the pin count buffer once initially. We can efficiently compute this by checking whether their preferred target cluster differs from their current cluster.

To avoid unnecessary computations for edges with only a few move candidates, we use sorting nets for edges with only three remaining pins. For graph edges, we went even further and fully formulated all possible branches to avoid copying, sorting, and iterating over the pins altogether.

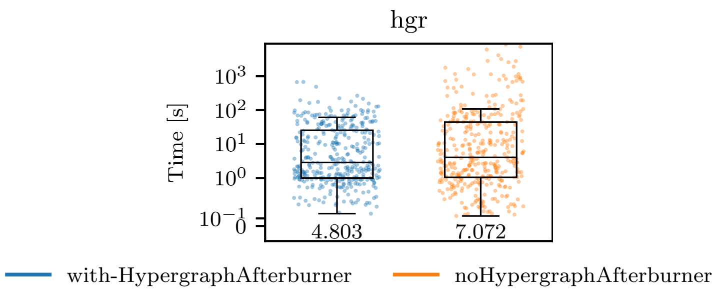

Figure 4.2: Comparing the running time of the refinement phase when utilizing the improved hypergraph afterburner to the naive afterburner.

4.1.2 Deterministic Rebalancer

The rebalancer works as the counterpart to the refinement in the Jet algorithm. The rebalancing algorithm aims to restore the partition's balance while worsening the solution quality as little as possible. To this end, we implemented a deterministic version of the Jetw rebalancing algorithm by Gilbert et al.[17] and introduced some changes inspired by the rebalancing algorithm presented by Maas et al. [39]. While taking running time into account, the main goal of our implementation is to retain as much quality as possible to close the gap in quality to non-deterministic algorithms.

Algorithm Overview. The algorithm works in rounds. As long as there are still imbalanced parts, the algorithm starts a new round. The goal of a round is to move vertices out of all overweight parts so that at the end of the round, none of the initially overweight parts is overweight anymore. Note that we do allow new parts to become overweight to retain as much quality as possible. To prevent these parts from becoming overweight again in later rounds, we define a deadzone. If the weight of a part is within the deadzone, it is no longer a valid target for moving vertices. In more detail, a single round of the rebalancing algorithm works as follows:

First, we iterate all vertices in parallel. For each vertex in an overweight part, we calculate the valid move that worsens the quality the least. We do this by first calculating the change in quality if we moved vertex v to any of the adjacent parts. If none of the adjacent parts is a valid target, i.e., they are in the deadzone or moving v would make them overweight, we also consider non-adjacent parts. A part is within the deadzone if its weight is greater than or equal to L_{dead}. We define this lower bound of the deadzone as $L_{dead} := L_{max} - c_{dead} \cdot (L_{max} - \lceil c(V)/k \rceil)$, where c_{dead} is a parameter we chose as 0.1 based on the parameter tuning experiments in Section 6.2. To prevent heavy vertices from causing oscillations and swaying the weight of parts too much, we additionally exclude vertices with a weight larger than $1.5 \cdot (c(p_s) - \lceil c(V)/k \rceil)$, where p_s is the current part of the vertex.

The goal of the algorithm is to reduce the weight of all blocks below L_{max} while retaining as much of the solution quality as possible. To this end, we sort all moves leaving a part by gain and select the shortest prefix, which reduces the weight of the part below the balance constraint. We implement this by collecting all considered moves based on their source part in k dense vectors. We sort these vectors in parallel according to the priority function. We use $gain(m, \Pi)/c(v)$ if the gain is negative and otherwise $gain(m, \Pi) \cdot c(v)$ as suggested by Maas et al. [39]. If the gain is non-negative, choosing heavier vertices is preferable since we progress faster toward a balanced solution. In the case of negative gain, dividing by the weight lets us evaluate a move as a trade-off between quality loss and progress toward a balanced solution. The latter case makes up the vast majority of cases.

Afterward, we calculate a parallel prefix sum over the vertex weights in the sorted sequence and find the shortest prefix with a combined weight larger than $c(p) - L_{max}$ via binary search. Finally, we can execute the calculated moves in parallel.

After a round, all initially overweight parts are no longer overweight. Due to the synchronous nature of the algorithm, we do not consider other moves executed in parallel. Because of this, other previously not overweight parts may become overweight in the process of a round. We keep performing rebalancing rounds to achieve a balanced solution until no part is overweight. Since, in theory, this could loop indefinitely, we introduce an upper limit of 30 iterations as a safety measure. In our experiments, however, the rebalancing algorithm always resulted in a balanced solution.

Sequential Minimal Move Prefix Calculation. After accumulating the moves per part in a dense vector and sorting it by priority, we want to find the shortest prefix of the sequence to reduce the weight of the part into the deadzone. To this end, we employ a parallel prefix sum that sums up all the weights of moved vertices. Afterward, we use binary search to find the index of the last move we need to execute.

However, these algorithms come with significant overheads, which can dominate their running time, especially for smaller inputs. Most importantly, we calculate the prefix sum

Algorithm 4: DETERMINISTIC REBALANCER

```
Input: Hypergraph $H = (V, E, \omega)$, Partition $P$
num-imbalanced-parts ← countImbalancedParts()
while num-imbalanced-parts > 0 do
    $M[p] \leftarrow \emptyset$, for $p = 0 \ldots k - 1$
    forall $v \in V$ do in parallel
        if $P[v]$ is imbalanced and $v$ not too heavy then
            (gain, target) ← computeGainAndTargetPart($v$)
            add $v$ to $M[P[v]]$
    forall part $p = 0 \ldots k - 1$ do in parallel
        excessWeight ← $\omega(p)$ - maxWeight - 1
        sort $M[p]$ by priority in parallel
        forall $i = 0, \ldots |M[p]| - 1$ do in parallel
            weights$[p][i] \leftarrow c([M[p][i]])$
        parallelPrefixSum(weights)
        lastMoveIdx ← binarySearch for excessWeight
        lastMoveIdx++
        forall $i = 0, \ldots$ lastMoveIdx do in parallel
            changeNodePart($M[p][i]$)
    num-imbalanced-parts ← countImbalancedParts
```

over the whole vector of vertex weights. This means that even if the minimal move prefix is very short, the parallel algorithm calculates a prefix sum over the entire vector of moves. To improve our average running time, we want to use a straightforward linear scan to sum up the weights of the moves and stop when the desired accumulated weight is reached. This algorithm's main advantage is that we can immediately stop summing vertex weights as soon as the desired value is reached. Additionally, it is very cache efficient because of the linear way it accesses memory. We determined the cutoff for the sequential algorithm to be 25600 in preliminary experiments. For larger inputs, the parallel algorithm is quicker on average for our benchmark set. Although this is certainly faster, the impact on overall running time is small since rebalancing itself only takes up a fraction of the running time on the parameter-tuning set.

Unsuccessful Changes. To further reduce the loss of quality, we experimented with introducing sub-rounds to the rebalancing algorithm to leverage more up-to-date information. Each sub-round contains a fraction of the vertices trying to move out of each of the overweight parts. To this end, we calculate all viable moves and then repeat the following three steps for each sub-round. First, we sort the remaining moves by priority. Second, for each part, we execute all moves until the part is either not oversized anymore or there are no more moves out of the part in the current sub-round. Afterward, we recalculate the gain and target part of the remaining vertices of subsequent sub- rounds. We experimented with varying numbers of sub-rounds from 2 to 8 but observed no noticeable improvement in solution quality. Additionally, the adaptation introduces additional work since we recalculate gains.

4.1.3 Decoupling Gain Calculation From Vertex Moves

In early experiments, we observed that applying the calculated moves was a major contributor to the running time of our deterministic Jet implementation. This is because we calculated the attributed gain for each move immediately using attributed gains. To avoid errors, this needs synchronization on the incident edges of the moved vertex, which turned out to be rather expensive. Additionally, further vertex moves may be made in the rebalancing step of the algorithm, which would cause additional calculations. Due to the synchronous structure of the algorithm, all decisions on vertex moves are made before any move is executed, which means there is no need to calculate the actual gains immediately. The actual change in quality is only needed at the end of an iteration when we have to decide whether the iteration improved the solution. To this end, we can omit the gain calculations when executing moves and recalculate the quality delta after all moves are executed. This allows us to avoid synchronization overhead and calculate attributed gains only once per affected edge.

We employ this optimization only for dyadic graphs because, for hypergraphs, we have to update the data structures managing the pin count per part anyway, which makes locking unavoidable.

Figure 4.3 shows the results on the set of large irregular and regular graphs. As we can see, the optimization reduces the running time on the irregular graph instances by roughly 23% on average, while the running time on regular graphs seems largely unaffected.

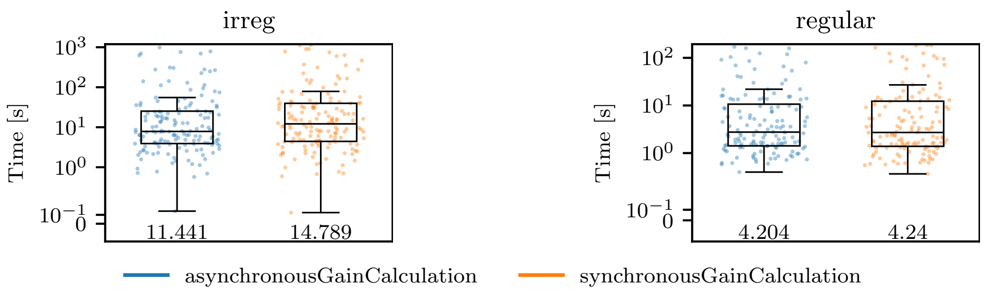

Figure 4.3: Comparing the running time of the refinement algorithm for synchronous vs. asynchronous gain calculation for graph instances.

4.2 Deterministic Flow Refinement

Greedy move-based refinement heuristics such as FM and the Jet algorithm often struggle to find non-trivial improvements. Large hyperedges, in particular, can make it difficult to find moves that improve the solution quality since they often have many pins in the blocks they are connecting, which results in zero-gain moves when only moving a single vertex.

Techniques based on maximum flows are able to overcome these limitations by finding a minimum cut separating two vertices. This leads to a more global view of the partitioning problem. Because of this, flow-based refinement is considered the most potent heuristic for improving partitions of (hyper-)graphs [19].

Algorithm 5: Parallel Flow-Based Refinement

```
Input: Hypergraph $H = (V, E, c, \omega)$, $k$-way partition $\Pi$ of $H$
1  $Q \leftarrow$ buildQuotientGraph($H, \Pi$)
2  while $Q$ has active blocks do
3      $A \leftarrow$ getMatchingOfActiveBlockPairs($Q$)
4      forall $(V_i, V_j) \in A$ do in parallel
5          $B := B_i \cup B_j \leftarrow$ constructRegion($H, V_i, V_j$)
6          $(N, s, t) \leftarrow$ constructFlowNetwork($H, B$)
7          $(M, \Delta) \leftarrow$ FlowCutterRefinement($N, s, t$)
8          if $\Delta \geq 0$ then
9              applyMoves($H, \Pi, M$)
10             if $\Delta > 0$ then
11                 mark $V_i, V_j$ as active
```

In this section, we present our deterministic parallel flow refinement scheme. We were able to reuse large parts of the parallel flow refinement configuration of Mt-KaHyPar [21]. We start by giving an overview of the framework's structure. Afterward, we provide details for the relevant components in the subsequent sections, where we focus on our main contributions to make the algorithm deterministic: The matching-based parallelization of the active block schedule, deterministic flow-network construction, and an adaptation to the refinement routine to make it deterministic.

Algorithm 5 gives a high-level overview of the algorithm. We schedule block pairs for refinement based on parallel scheduling using maximal matchings of edges of the quotient graph Q. For each scheduled block pair, we do the following four main steps: First, we extract a set of vertices B around the boundary nodes of the blocks, which are allowed to be moved. The set B then makes up the set of vertices of the flow problem, based on which we construct the flow hypergraph in the second step. Afterward, the `FlowCutter` algorithm [48, 28] computes a balanced partition of the network, which we convert into a set of vertex moves M with its improvement Δ to the connectivity on the original partition. Finally, if the moves do not worsen the solution quality, we apply them to our partition Π. If the solution quality improves, we additionally mark the corresponding blocks as active for the next round.

4.2.1 Parallel Active Block Scheduling

The active block scheduling algorithm was originally proposed by Sanders and Schulz [44]. The algorithm allows the flow-based refinement algorithm for bipartitions to be applied to k-way partitions. The algorithm schedules blocks in rounds, where a single round contains all pairs of blocks with at least one active block. Initially, all blocks are active. A block is marked as active for the next round if it was part of a search that improved the edge cut in the previous round.

Parallelization. We parallelize this approach by scheduling block pairs that form a maximal matching in the quotient graph in parallel, as suggested by Grötschla [27]. The main reason for doing this is that all scheduled block pairs are fully independent. This means all the following steps can be performed in parallel without interfering with the solutions for the other block pairs. The difference between our deterministic scheduling and the scheduling of Grötschla is that we need an additional synchronization point after all block pairs of a matching finished calculating. Otherwise, the scheduling decision depends on which of the other block pairs finishes processing first, which is non-deterministic. However, this comes at the price of limiting the parallelism to at most $k/2$. This is a significant disadvantage compared to the non-deterministic scheduling approach employed in the flow configuration of Mt-KaHyPar [21], which interleaves blocks scheduled in future rounds and resolves arising conflicts afterward.

The two goals of the scheduling algorithm are to maximize parallelism on the one hand and maximize improvement on the other hand. Parallelism is hindered when we can not schedule $k/2$ of the remaining active block pairs because too many depend on the same block and can, therefore, not be scheduled simultaneously. In order to combat this, we select the block that is part of the most active block pairs first, as proposed by Gröttschla [27]. This has the effect that, especially in later iterations, the maximum number of participations of any given block is small since we heuristically decrease the participations of the most popular blocks by one for each scheduled matching. Hence, reducing the dependencies between block pairs allows us to schedule more blocks in parallel. To this end, we maintain a list of the blocks sorted by participations, which we update whenever we schedule a new set of block pairs.

Based on the selection of the first block, we select its partner based on the improvement in previous rounds, selecting blocks with larger previous improvements first. We break ties based on the accumulated weight of the corresponding cut edges and finally based on the block number. This heuristic aims to schedule block pairs that promise larger improvements to the solution quality first. To this end, we store a list of eligible block partners sorted by the priority we just described for each block.

Quotient Graph. The quotient graph stores information for all possible $k(k+1)/2$ block pairs, which form the edges of the graph. For each edge of the quotient graph, we explicitly store the cut-hyperedges connecting the two corresponding blocks since this information is needed for network construction. Additionally, we keep track of the accumulated improvement across multiple rounds and whether the block has previously been scheduled or not. We construct the data structure by iterating over the edges $e \in E$ of the hypergraph and adding it to all block pairs (V_i, V_j) where $\Phi(e, V_i) > 0$, $\Phi(e, V_j) > 0$ and $i < j$.

When moving vertices, we have to adjust the cut-hyperedges of the quotient graph accordingly. To this end, we keep track of all hyperedges where the connectivity to the target block V_d increases to one. After processing a matching, we can then iterate over all collected hyperedges in parallel and add them to all block pairs contained in $\{\{V_d, V_k\} \mid V_k \in \Lambda(e) \setminus \{V_d\}\}$. During network construction, cut-hyperedges no longer connecting V_i and V_j are removed lazily. Furthermore, we sort the cut-hyperedges by id to ensure determinism during network construction, but only if the corresponding block pair is scheduled.

Further Heuristics. We employ the pruning rules introduced in `KaHyPar` [31] to improve running time by skipping unpromising flow computations. The first rule skips block pairs, where the accumulated weight of cut weights does not exceed 10. This rule is active on all levels except the top level, where the hypergraph is the finest. The second rule, beginning at the second round, originally prohibited block pairs from being scheduled, which never saw an improvement in previous rounds. We slightly modified this rule to fit our scheduling scheme by only applying this pruning rule when the block pair has been scheduled in a previous round already.

4.2.2 Network Construction

The first step of the network construction is to grow a size-constrained region around the cut hyperedges of a scheduled block pair. Based on this set of vertices B, we then construct the flow hypergraph in the second step. By contracting all vertices $V_i \setminus B$ to the source s and all vertices $V_j \setminus B$ to the sink t, we get a coarser hypergraph H, which then serves as an input into the FlowCutter algorithm. We modify the non-deterministic network construction algorithms from Mt-KaHypar [21] to produce deterministic results.

Region Growing. An implementation of the region-growing algorithm by Sanders and Schulz [31] is already available in the Mt-KaHyPar framework. Since the implementation is sequential and, therefore, deterministic, we were able to reuse it in our deterministic algorithm. We give a quick description of the algorithm for the sake of completeness and refer to the publications of Sanders and Schulz [31] and Gottesbüren et al. [21], respectively, for further details.

The algorithm grows a region $B := B_i \cup B_j$ around the cut hyperedges of the partition via two breadth-first searches (BFS). The first BFS constructing $B_i \subset V_i$ is initialized with all boundary vertices of V_i. The algorithm keeps adding vertices to B_i until $c(B_i) > (1 + \alpha\epsilon)\lceil \frac{c(V)}{2} \rceil - c(V_j)$, where α is an input parameter. The second half of the boundary B_j is computed the same way. The distance of each vertex $v \in B$ is additionally bounded above by a parameter δ.

Parallel Construction Algorithms. We adapt the two parallel non-deterministic construction algorithms of Mt-KaHyPar [21]. To achieve deterministic results, we must ensure that all elements of the hypergraph are identical every time. This includes the order in which pins are stored, as well as the order of incident hyperedges. A further concern is identical hyperedge removal, where we have to ensure that we always detect all parallel hyperedges and remove the same ones independent from any non-deterministic scheduling decisions.

Algorithm 6: Parallel Flow-Hypergraph Construction

```
Input: Hypergraph $H = (V, E, c, \omega), B, E_B$
1 $L_e \leftarrow$ constructListOfHyperedges($H, B, E_B$)
2 invalidateDuplicateHyperedges($L_e$)
3 constructThreadLocalPinList($L_e$)
4 combinePinLists()
5 forall $e \in E_{\mathcal{H}}$ do in parallel
6     sortPins()
7 forall $v \in V_{\mathcal{H}}$ do in parallel
8     sortIncidentEdges()
```

Algorithm 6 gives an overview of the flow-hypergraph construction. The differences between the previously mentioned construction algorithms are limited to the construction of the list of hyperedges in line 1. For both algorithms, we first construct each hyperedge separately while adding it to a hash map for identical hyperedge detection. Afterward, we invalidate duplicate hyperedges and finally calculate a thread local pin list for the remaining ones. We combine pin lists into the final pin list of $\mathcal{H}$ via a prefix sum. Let $E_B = \{e \in E \mid e \cap B \neq \emptyset\}$ the set of hyperedges that contain vertices from the previously constructed region B. Both are sorted afterward to ensure the deterministic order of pins and incident edges.

The first algorithm iterates over the hyperedges of $e \in E_B$ in parallel. While iterating over the pins of a hyperedge, we check whether pin p is contained in B or not. If $p \in B$, we add p to the hyperedge. Otherwise, if $p \in V_i$ or $p \in V_j$, we add the source s or sink t vertex, respectively.

Iterating over all pins of the hyperedge, independent of whether they are contained in the flow hypergraph or not, as in the first algorithm, can be wasteful. This is especially true for large hyperedges, where we expect only a fraction of the pins to remain in the flow hypergraph. Because of this, Gottesbüren et al. [21] propose a second algorithm for sparse hypergraphs or hypergraphs with a large average hyperedge size whose complexity only depends on the remaining pins. This, however, comes at the cost of additionally sorting the pin lists.

The second algorithm iterates over all vertices $v \in B$, and for each incident hyperedge $e \in I(v)$ emits a tuple (e, v) into a collection, which groups entries by e. This is implemented as a hash table where each hash bucket is protected with a spin lock for parallel insertion. The hash function only uses the hyperedge IDs so that all pins of the same hyperedge are inserted into the same bucket. The group-by operation in each bucket is implemented via sorting. Afterward, we create the list of hyperedges by adding the source s or sink t to e, depending on whether $\Phi(e, B_i) < \Phi(e, V_i)$ or $\Phi(e, B_j) < \Phi(e, V_j)$. This way, we reconnect the hyperedge to the source or the sink respectively, since in the case $\Phi(e, B_i) < \Phi(e, V_i)$ we removed pins p from the hyperedge with $p \notin B$ and $p \in V_i$, by only keeping pins $p \in B$. This means the hyperedge should be connected to the source. The same applies to the second case and the sink t.

Furthermore, both algorithms discard single-pin edges as well as edges connected to both the source and the sink, since such hyperedges can not be removed from the cut.

Identical Hyperedge Detection. Since hyperedges of the original hypergraph H are contained in the flow-hypergraph $\mathcal{H}$ to varying degrees, it can happen that hyperedges could be duplicated in $\mathcal{H}$. To reduce the size of the resulting hypergraph and therefore reduce the time of the FlowCutter algorithm, we only keep one representative for each unique hyperedge with the summed weight of all its duplicates. To this end, we use the algorithm of Aykanat et al. [3, 10]. It is important to note that we can not do this on the fly as in the non- deterministic algorithm since the resulting ID of the representative hyperedge depends on which duplicate is scanned first and the possibility of missing duplicate hyperedges due to simultaneous insertions [21]. Instead, we first determine the final set of hyperedges to be able to eliminate duplicates in a deterministic manner.

While constructing the list of hyperedges in line 1 of Algorithm 6, we sort the hyperedges into buckets using the fingerprint $f_e = \sum_{v \in e} v^2$. This allows us to only compare hyperedges with identical fingerprints in the following step since edges with different fingerprints can not be identical. We iterate over all buckets in parallel. To compare the hyperedges within the same bucket, we first sort them by fingerprint, size, and, finally, ID. Afterward, we can do a single sweep over the bucket to identify and invalidate duplicate hyperedges by comparing their pin lists only if both fingerprint and size are identical.

4.2.3 Flow-Based Refinement

The FlowCutter [48, 28] algorithm is used as the flow-based refinement routine on a bipartition in KaHyPar and Mt-KaHyPar. The algorithm is parallelized by employing a parallel push-relabel algorithm [21]. To achieve deterministic results, we only have to add a sorting step to the selection of piercing nodes. Therefore, we give a rough overview of the FlowCutter algorithm and discuss our changes to achieve determinism. For further details, especially to the parallel maximum flow algorithm used, we refer to the paper introducing the non-deterministic flow-refinement configuration by Gottesbüren et al. [21].

Algorithm Overview. FlowCutter is an algorithm designed to find balanced bipartitions in a hypergraph through a sequence of incremental maximum flow problems. The algorithm repeats the following steps: First, the previous flow is augmented to a maximum flow based on the current source set S and sink set T. Then source- and sink-side cuts S_r and T_r are determined using a forward residual BFS from S and a backward residual BFS from T, respectively. This results in the two bipartitions: $(S_r, V \setminus S_r)$ and $(V \setminus T_r, T_r)$. If none of the two is balanced, the vertices found by the BFS are added as a source or sink, depending on whether $c(T_r)$ is larger than $c(S_r)$ or not. Furthermore, an additional *piercing vertex* is added to find a different cut in the next iteration.

Determinism. The most important observation to achieve determinism is that while the maximum-flow algorithm is non-deterministic, the min-cuts found are deterministic. This is because min-cuts are nested, and the algorithm selects the smallest min-cut. This min-cut is unique, therefore making the selection deterministic. Therefore, we only have to ensure that we add *piercing vertices* in a deterministic manner. When adding new vertices to the set of candidates for piercing vertices, we have to make sure that the result candidate list is independent of non-deterministic scheduling decisions. We achieve this by only sorting the newly added candidates afterward. This works since the set of candidates we add is always the same; only the order in which we add the candidates to the vector is non-deterministic.

5 Deterministic Coarsening

In their work presenting the state-of-the-art deterministic hypergraph partitioner Mt-KaHyPar's deterministic mode [23], Gottesbüren and Hamann first present their deterministic coarsening algorithm based on label propagation. Comparing their deterministic algorithms for each phase of the multilevel paradigm, they show, unexpectedly, that the coarsening phase performs the worst compared to its non-deterministic counterpart. In this section, we first describe our approach towards understanding the differences and several observations we made as a result. These observations were the primary motivation for many of the experiments and changes we made to the algorithm. Afterward, we present several corrections and improvements to the algorithm to fully close the gap between deterministic and non-deterministic coarsening regarding partition quality.

All experiments in this chapter are performed on the parameter-tuning benchmark sets consisting of 100 hypergraphs, 38 irregular, and 33 regular graphs. We perform the experiments on a machine consisting of two Intel Xeon E5-2683 16-core processors clocked at 2.1 GHz using $t = 64$ threads.

5.1 Non-Deterministic vs Deterministic Coarsening

Both the non-deterministic and the deterministic coarsening algorithms of Mt-KaHyPar are clustering algorithms and follow the same general structure: Initially, all vertices are in their own singleton cluster. A pass of the coarsening algorithm consists of iteration over all vertices $v \in V$ and calculating the best cluster for v to join based on the heavy-edge rating function. Vertices are then moved to their target cluster. Afterwards, the clustering is contracted to obtain a coarser hypergraph with a similar structure.

Algorithm 7: Asynchronous Local Moving Round

```
shuffle vertices randomly
forall $v \in V$ do in parallel
    determine and perform best move for $v$
    update data structures
```

While both the non-deterministic and the deterministic coarsening algorithms in Mt-KaHyPar are based on clustering and are, therefore, local moving algorithms, there is one major

Algorithm 8: Synchronous Local Moving Round

```
1 shuffle vertices randomly and split into sub-rounds
2 forall sub-rounds do
3     forall $v \in V$ in sub-round do in parallel
4         determine best move for $v$
5     perform or discard moves and update data structures
```

difference between the two, which we can see comparing Algorithm 7 and 8. While the deterministic algorithm moves vertices synchronously, the non-deterministic algorithm performs vertex moves asynchronously. When making decisions, the asynchronous algorithm immediately moves the vertex to its new cluster and updates all corresponding data structures. Because of this, the decision is highly dependent on non-deterministic scheduling decisions and is unsuitable for a deterministic algorithm.

Instead, the deterministic coarsening algorithm works in sub-rounds. The algorithm calculates all clustering decisions for vertices inside a sub-round in parallel based on the information before the sub-round and approves a selection of moves afterward. Because of this, decisions within a sub-round are independent of each other. This has the disadvantage that the information on which decisions are made can be outdated and may result in worse decisions. On the other hand, Gottesbüren and Haman [23] show that the preprocessing and refinement steps, which are both local moving algorithms, do not worsen significantly. Because of this and the counterintuitive fact that more sub-rounds seem to worsen solution quality, we took a closer look at the differences in both algorithms' results.

Hybrid Coarsening Algorithm. First, we implemented a hybrid coarsening algorithm that performs deterministic clustering for the first few levels and then switches to non-deterministic coarsening. The idea behind this experiment is to get a sense of when the bulk of the loss in quality occurs. Figure 5.1 shows how many deterministic passes are necessary to worsen the solution quality enough to be equal to the fully deterministic algorithm. After only two deterministic passes, the hybrid and deterministic algorithms produce equal results for hypergraphs. The results for regular graphs show that three deterministic passes are enough to become roughly equal to the deterministic algorithm. As we can see in the center, the non-deterministic and deterministic coarsening results are very close for irregular graphs. Therefore, it is unsurprising that only one deterministic pass suffices in that case. Overall, the results clearly show that most loss in quality occurs on the first two or three levels.

Differences in the Solutions. Now that we have seen that the first few levels are crucial to closing the gap between the two algorithms, we will take a look at different metrics of the clustering after each pass. To this end, we collected data on the structure

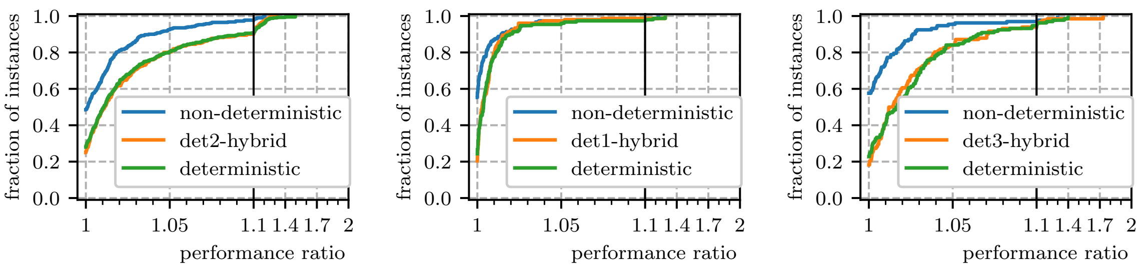

Figure 5.1: Comparing the solution quality when using non-deterministic, deterministic, and hybrid coarsening on hypergraphs (left), irregular graphs (center), and regular graphs (right). det$x$- performs deterministic coarsening for the first x levels and uses non-deterministic coarsening for the remaining levels.

of the clustering at the end of each coarsening pass for both algorithms. With the help of these statistics, we aim to understand how the differences in solution quality manifest in the clustering each algorithm produces and, from there, get an idea of possible solution strategies. We compare the results of both algorithms by calculating the relative ratio of each statistic after each coarsening pass. This means that if we look at a measurement after the first iteration, we divide the value for the deterministic algorithm by the value of the non-deterministic algorithm. Therefore, if the plots show a value larger than one, the value under consideration is larger for the deterministic algorithm than for its non- deterministic counterpart. Because of our previous observation that the loss of quality occurs on the first few levels, we focus on these when analyzing the following Figures 5.2 to 5.6.

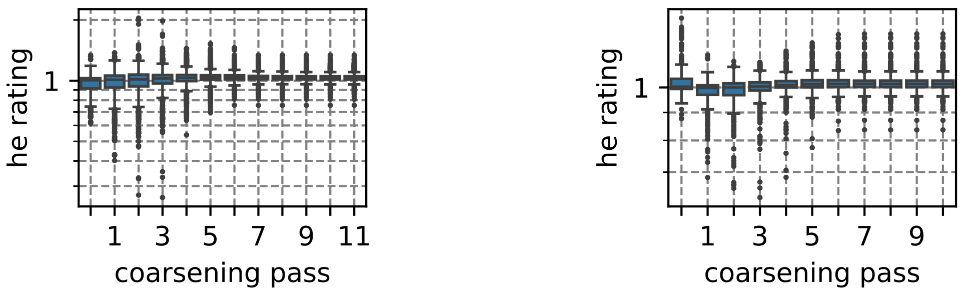

Figure 5.2: Relative ratio of the heavy-edge rating of deterministic compared to non-deterministic on hypergraphs (left) and regular graphs (right).

Figure 5.2 shows that, even though the final rating of the deterministic algorithm tends to be higher than the rating for the non-deterministic algorithm, it starts with a lower heavy-edge rating score for the first few rounds. This makes sense looking at Figure 5.3, in which we can clearly see that, especially for hypergraphs, the deterministic algorithm produces larger clusters on average for the first few rounds. These observations lead us to believe the algorithm is over-merging in the first few levels and then suffers from the consequences in subsequent coarsening passes. This hypothesis is further supported in Figure 5.4. Here, we can see that, especially in the first pass, the deterministic algorithm produces significantly larger clusters than its non-deterministic counterpart. The problem with over-merging in the first few iterations is that these large clusters are more likely to attract additional vertices in subsequent passes, worsening the chances for a more balanced clustering. If too many vertices try to merge into a single cluster, resulting in some of the moves being rejected, there might be vertices with no option left and, therefore, remaining as a singleton cluster. This behavior can be seen in Figures 5.5 and 5.6. The first plot shows that the final result of the deterministic algorithm usually contains a cluster with a minimum size smaller than that of the non-deterministic solution. This can be explained by the second plot, which shows the number of singleton clusters after each coarsening pass. From the very beginning, the deterministic algorithm produces significantly more singleton clusters than the non-deterministic algorithm. Besides the over-merging, one possible explanation is that vertices may decide to join each other, simply swapping cluster IDs instead of joining each other in a single cluster. This is possible because of the synchronous local moving approach, which calculates all target clusters based on information before the sub-round.

Based on these observations, we see two main suspects for the quality difference. First the over- merging in the early coarsening passes and, secondly, vertex swaps, which result in vertices effectively remaining singletons.

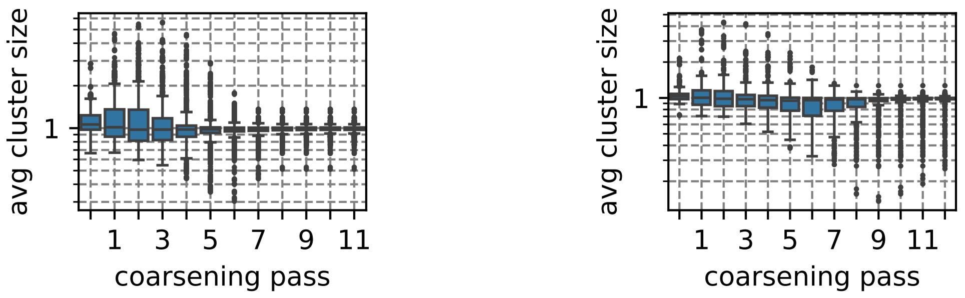

Figure 5.3: Relative ratio of the average cluster size of deterministic compared to non-deterministic on hypergraphs (left) and regular graphs (right).

5.2 Algorithm Overview

To prevent the two prominent symptoms of over-merging and vertex-swaps, we try different approaches promoting a more controlled and slow style of combining vertices, like matchings, splitting the contracted vertices equally between subrounds, and using subrounds exponentially increasing in size. However, as we will see later in this chapter, only the last technique shows any promise in improving the algorithm. Additionally, we check for vertex swaps and prevent them from happening.

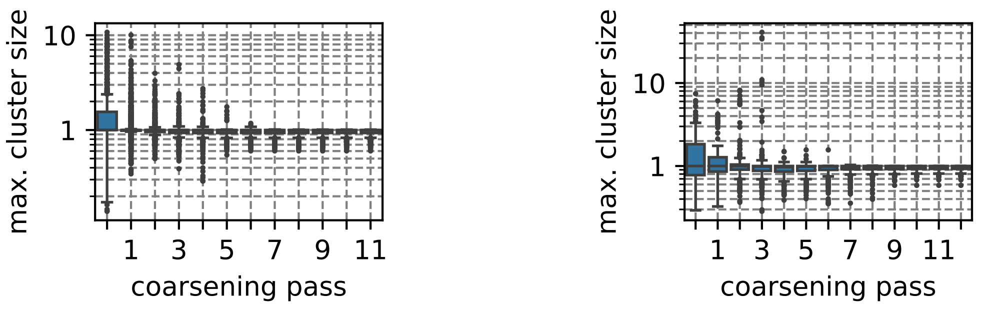

Figure 5.4: Relative ratio of the maximum cluster size of deterministic compared to non-deterministic on hypergraphs (left) and regular graphs (right).

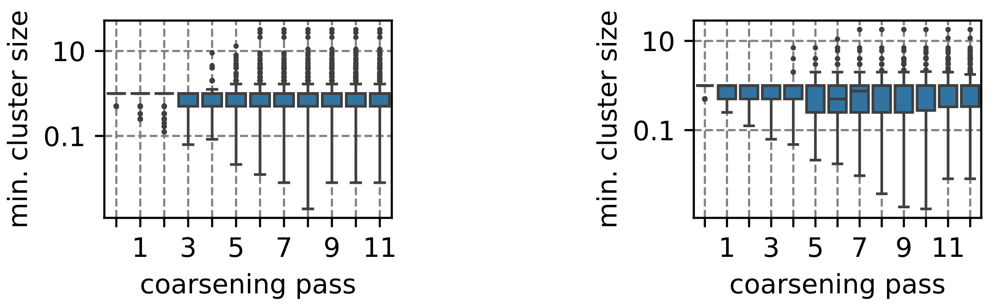

Figure 5.5: Relative ratio of the minimum cluster size of deterministic compared to non-deterministic on hypergraphs (left) and regular graphs (right).

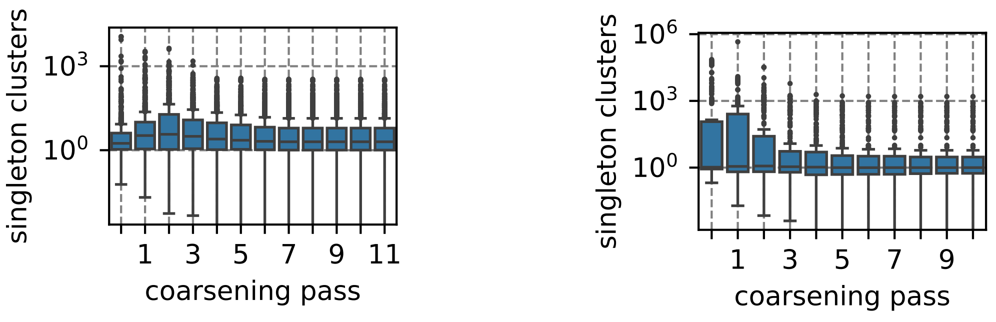

Figure 5.6: Relative ratio of the number of singleton clusters of deterministic compared to non-deterministic on hypergraphs (left) and regular graphs (right).

Algorithm 9 shows pseudocode for a coarsening pass over the vertices. Changes we made are highlighted in blue. Initially, each vertex is in its own cluster. The algorithm works in sub-rounds, which we calculate according to a prefix-doubling approach. First, according to the heavy-edge rating function, we compute the best target for each vertex of a sub-round. We improved the target cluster selection by counting clusters only once per hyperedge. After that, we employ a strategy that merges two vertices that want to join

Algorithm 9: COARSENING PASS

```
1  shuffle vertices randomly and split into sub-rounds according to prefix-doubling
2  $C[v] \leftarrow v, P[v] \leftarrow v : \forall v \in V$
3  opportunistic-weight$[v] \leftarrow c(v) : \forall v \in V$
4  forall *sub-rounds* do
5      forall $v \in V$ *in sub-round* do in parallel
6          $P[v] \leftarrow ComputeHeavyEdgeRating(v)$
7          opportunistic-weight$[P[v]] += c(v)$
8      $handleVertexSwaps()$
9      $M \leftarrow \emptyset$
10     $F \leftarrow \emptyset$
11     forall $v \in V$ *in sub-round* do in parallel
12         if *opportunistic-weight* $[P[v]] \leq CW_{max}$ then
13             $C[v] \leftarrow P[v]$
14             if *opportunistic-weight*$[v] \neq c(v)$ then
15                 add $v$ to $F$
16         else
17             add $v$ to $M$
18             if *opportunistic-weight*$[v] \neq c(v)$ then
19                 add $v$ to $F$
20     sort $M$ lexicographically by $(P[v], c(v), v)$
21     forall $i = 0, \ldots, |M|$ do in parallel
22         if $i = 0$ *or* $P[M[i-1]] \neq P[M[i]]$ then
23             forall $j = i$ *until* $CW_{max}$ *exceeded* do
24                 $C[M[j]] \leftarrow P[M[j]]$
25             set opportunistic-weight of $C[M[i-1]]$
26     forall $v \in V$ *in* $F$ do in parallel
27         if $C[v] \neq v$ then
28             opportunistic-weight$[v] -= c(v)$
29 contract clustering $C$
```

each other's cluster to prevent vertex swaps. Now that we have calculated the moves for each cluster, we have to decide whether to execute moves. To this end, we utilize the *opportunistic weight* of a cluster, which we calculate simultaneously to the target clusters. It contains the combined cluster weight of all vertices that want to join the cluster and thus is an upper limit to the cluster weight after the sub-round. Therefore, we can approve all moves where the opportunistic weight is at most the maximum cluster weight

$CW_{max} := min(L_{max}, C(v)/CL)$, where CL is the contraction limit of $160 \cdot k$. All other moves are aggregated in a dense vector M to be processed later. Meanwhile, the dense vector F collects all vertices that leave clusters that may end up non-empty after the sub-round, for which we have to update the opportunistic weights at the end of the sub-round. Finally, we have to deal with moves into clusters that would exceed CW_{max} if all moves were approved. We sort them lexicographically by cluster, vertex weight, and vertex ID. For each cluster in parallel, we approve moves one by one and reject all remaining moves that would exceed the maximum cluster weight.

Prefix-Doubling. Instead of a fixed number of sub-rounds of equal size, we structure sub-rounds similarly to the concept of prefix-doubling, i.e., we use sub-rounds of exponentially increasing size. We start with 100 sub-rounds of size 1 and then increase the size of each subsequent sub-round by a factor of 1.8 up to the upper limit of one percent of the total number of vertices. Intuitively, more sub-rounds should result in better solution quality since the information is more up-to-date. Additionally, especially on early levels, many target clusters have equal ratings. Since all clusters start out as singletons with unit weights, many ties can occur when calculating the heavy-edge rating. We try to break this uniformity by starting with smaller sub-rounds to allow the formation of some clusters so as not to amplify the equal score issue. As shown in Figure 5.7, this approach to defining sub-rounds performs equally to the old approach of using three sub-rounds of equal size. However, because of its finer-grained structure, which is closer to moving one vertex at a time, we expect the algorithm to be more robust in instances we have not tested. Additionally, as illustrated in Figure 5.8, prefix-doubling produces slightly better initial cuts for graph instances. Especially for irregular graphs, the configuration using prefix-doubling produces the best result for 75% of the instances, and for regular graphs, it slightly outperforms its counterpart on 63% of our parameter-tuning instances. In Figure 5.9, we can see the influence of prefix-doubling on the running time of the coarsening phase. While it increases the geometric mean running times across all instance types, the peaks in running time remain stable. The increased running time is likely due to the initial 100 sequential sub-rounds.

Computing the Heavy-Edge Rating. We changed the rating calculation for hypergraphs, which is used to determine the target cluster of a vertex $v \in V$. In the original implementation by Gottesbüren and Hamann [23], the rating for a cluster is calculated by iterating over each incident edge of v and adding the heavy-edge rating for each pin to its corresponding cluster. Therefore, edges with n pins in the same cluster are counted n times towards the rating of the same cluster, which results in a heavy bias towards larger clusters. Instead, we want to count a hyperedge only once for each cluster. Rather than tracking exactly which clusters we have already counted for a hyperedge, we employ a simple bloom-filter-like technique to reduce the overhead. We allocate an array of 16-bit integers of size s, where $s := min\{10 \cdot |e_{max}|, n\}$ is the minimum of ten times the largest

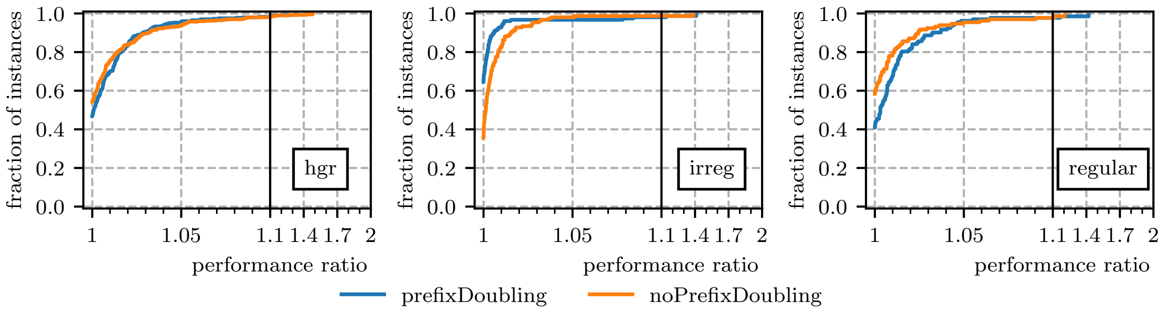

Figure 5.7: Comparing the overall quality with and without prefix-doubling to generate sub-rounds in the coarsening phase.

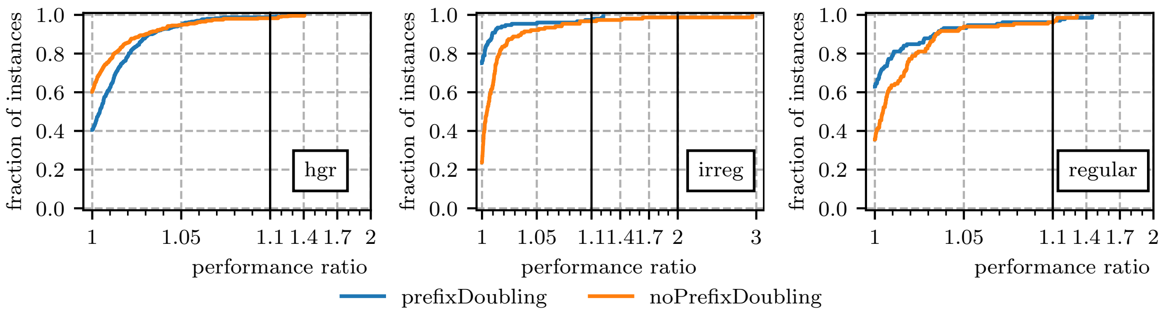

Figure 5.8: Comparing the quality after **initial partitioning** with and without prefix-doubling to generate sub-rounds in the coarsening phase.

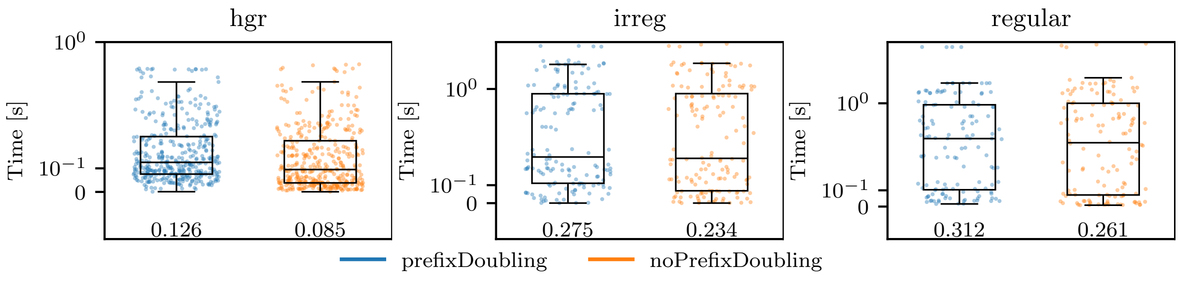

Figure 5.9: Comparing the coarsening time with and without prefix-doubling.

occurring edge size $|e_{max}|$ and the number of vertices n. The choice for s results in a false positive rate of less than 1% [4]. Let b be the number of bits needed to represent s. We use the b least significant bits of the cluster label as an index into the array to check if a cluster has already been processed for a hyperedge. If not, we write the current 16-bit flag to mark the cluster as processed. To reset the array for a new hyperedge, we only have to increment the flag used to mark clusters as processed.

Preventing Vertex Swaps. The synchronous local moving algorithm employed in the coarsening phase first calculates the target clusters for each node of the sub-round before executing moves synchronously. Because of this, all calculated moves are computed independently so that calculations do not influence each other. This can result in two separate vertices selecting each other as their respective target cluster. When executing these moves synchronously, both vertices are moved, effectively only swapping cluster IDs instead of merging into a combined cluster. We want to avoid this for multiple reasons: Firstly, since we do not combine them into a single cluster, we do not reduce the size of the graph and, therefore, make no progress towards the actual goal of coarsening the graph. Additionally, vertices that wanted to join v in the current sub-round now join u and vice versa. Therefore, the joining vertices may not even be connected to the cluster they are joining since the vertex they originally wanted to merge with is no longer present. Finally, as mentioned before, vertex swaps effectively lead to swapping the labels of vertices, which may reduce the effect of locality that is inherently encoded in the vertex ids.

To avoid this, we perform an additional sweep over all vertices of a given sub-round, checking for said vertex swaps. If we find two vertices with each other as their target cluster, we resolve the issue by setting the target of both vertices to the cluster with a larger weight and using the cluster IDs as a fallback. This does not always solve the second problem since it may still occur when the original target vertex moves to a different cluster, just not swapping with another vertex. To this end, we also tried merging all vertices connected via their preferred target cluster into one single combined cluster. This, although, did not result in any improvement over the previous resolution approach.

Updating Cluster Weights. Finally, we corrected the calculation of the cluster weights between sub-rounds. In this case, the vertex initially representing a cluster is moved out of its cluster, the calculated cluster weight still included the weight of the vertex since in the non-deterministic algorithm the cluster is empty after the move, and the ID is no longer used. In the deterministic algorithm, other vertices can join this cluster, which is not necessarily desirable. These wrong cluster weights were then used in subsequent sub-rounds. This may cause clusters to not be eligible targets because of the additional weight. If an affected cluster becomes overcrowded, i.e., the combined weight of all vertices that want to move into the cluster is larger than the limit for the cluster weight, fewer vertices end up in the cluster than the weight capacity would allow. Hence resulting in a worse clustering. To correct this, we collect all vertices that move out of their cluster while other vertices try to join their cluster in a dense vector. After approving and rejecting all moves for the current sub-round, we iterate over the collected vertices in parallel. We subtract the weight $c(v)$ from the original cluster of vertex v if the vertex was moved into another cluster. This, however, did not impact solution quality or running time in a significant way (see Figures 5.10 and 5.11).

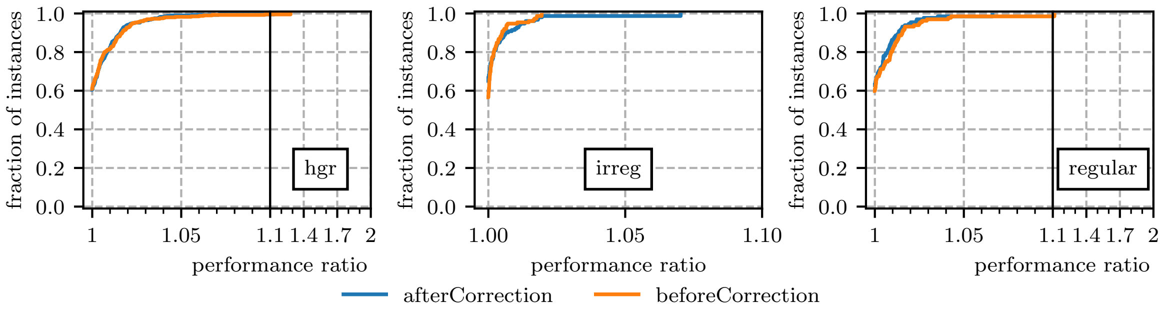

Figure 5.10: Impact on the overall solution quality when correcting the cluster weights in the coarsening phase.

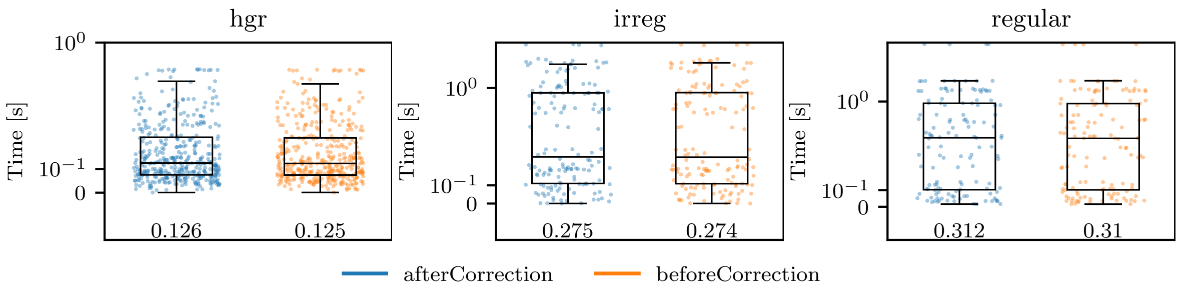

Figure 5.11: Comparing the coarsening time with and without correcting the cluster weights.

5.3 Unsuccessful Changes

In this section, we describe different approaches motivated by the observations in Section 5.1, that did not make it into the final algorithm. While the first two approaches try to prevent the observed over-merging of the deterministic coarsening algorithm, the third approach tries to reduce the effect of outdated information caused by the synchronous local moving approach. While these approaches seemed promising in theory, they could not improve the solution quality and, with the exception of the adaptive edge sizes, even worsened the score of the final partition.

Matching on the First Few Levels. Instead of using the clustering coarsening approach, we implemented a simple matching algorithm to perform the coarsening on the first few levels. The goal behind this approach is to prevent the over-merging we observed in Section 5.1 by using a more structured approach and strictly following the contraction limit of the coarsening pass. We implemented this by sorting all edges according to an edge rating function. Then, we iterate over all edges and combine the edge's vertices into the same cluster. For hypergraphs, we try to contract the whole hyperedge, combining all its pins into the same cluster. This succeeds if none of the edge's vertices have been matched and if the resulting cluster does not exceed the weight limit. We experimented with different rating functions [43] as can be seen in Figure 5.12. Unfortunately, even just performing the algorithm on the first coarsening pass worsens the solution quality, especially for hypergraphs. Further passes with the matching algorithm only worsened the result. Because of this, the matching approach is unsuitable for our deterministic coarsening algorithm.

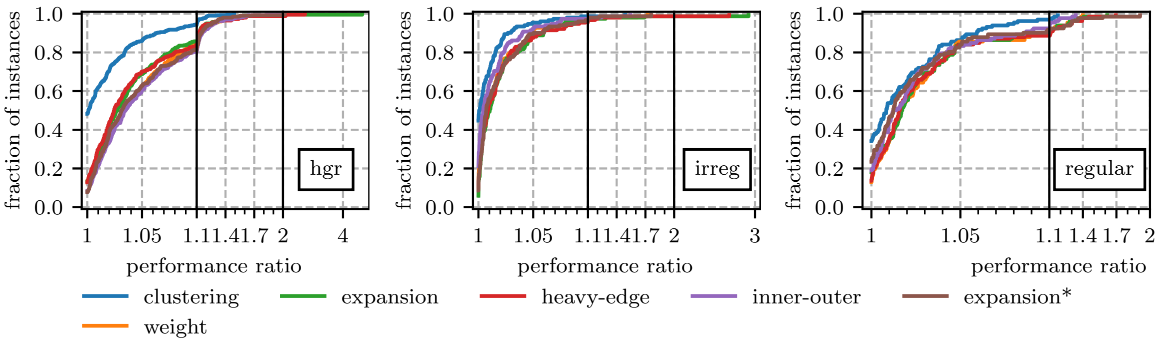

Figure 5.12: Performance profiles comparing the quality of configurations using a matching approach with different edge rating functions on the first level compared to the clustering approach.

Splitting Contracted Nodes Equally Between Sub-Rounds. The goal of a pass of the coarsening algorithm is to reduce the size of the hypergraph by a constant factor. Because of this, in a single pass, only a certain number of vertices should be contracted to avoid overmerging. The current algorithm, however, performs all calculated moves within a sub-round and checks afterward if this contraction limit is reached. This can have multiple effects: Firstly, the number of contractions is not actually limited by the contraction limit. In some cases, this can lead to the contraction limit being overshot significantly. Therefore, the hypergraph may shrink significantly faster than intended, reducing the advantages of the multilevel structure of the algorithm. Secondly, vertices in earlier sub-rounds are significantly more likely to join other clusters than vertices in later sub-rounds. This is due to the limitations on cluster sizes. This means that vertices whose preferred target cluster is already overweight from previous sub-rounds have to settle for worse targets or do not find a suitable cluster at all. Furthermore, if the hypergraph is small enough, the algorithm may terminate after any sub-round. Because of this, vertices might not get the chance to join any cluster if the algorithm terminates after an earlier sub-round.

To mitigate these effects, we split the number of allowed contractions equally between all sub-rounds. To do this, we collect all vertices that want to move to a different cluster. If the number of moves is larger than the limit ℓ for the sub-round, then we select ℓ moves deterministically at random and try to approve them. We repeat this until either there are no moves left to approve or the limit is reached. By design, this has the effect that the number of contracted vertices in a coarsening pass never exceeds the contraction limit. Therefore, the algorithm never terminates before the last sub-round. Additionally, we reduce the bias towards vertices of earlier sub-rounds by reducing the number of moved vertices in earlier sub-rounds. While we could observe the described effects on moved vertices per sub-round and per coarsening pass, these changes negatively impact overall results (see Figure 5.13). This change barely impacts the solution quality for hypergraphs. For regular graphs, on the other hand, splitting the contracted nodes equally between sub-rounds worsened the final partition for roughly 70% of the instances.

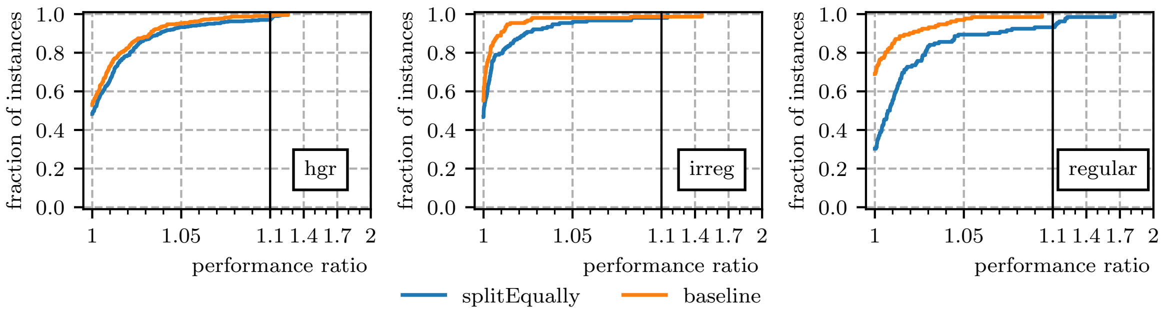

Figure 5.13: Comparing the overall quality when strictly splitting contracted vertices equally between sub-rounds to the baseline of checking the contraction limit after the sub-round.

Adaptive Edge Sizes. The synchronous local moving approach of the coarsening algorithm contracts vertices only when all move calculations are finished. Because of this, sizes of hyperedges in later sub-rounds are not accurate anymore, i.e., when two pins that were previously in different clusters moved to the same cluster, the edge size should be reduced by one. These outdated edge sizes are then used to calculate the heavy-edge rating to decide target clusters. The idea, mentioned briefly by Gottesbüren and Hamann [23], is to update the edge sizes after each sub-round to reflect the previously performed vertex moves accurately. We expected this to improve solution quality since the algorithm can make decisions based on more up-to-date information. As we can see in Figure 5.14, this did not improve the solution quality. A possible explanation for this is that while the approach provides more up-to-date information on edge sizes for the algorithm, moves performed in the same sub-round are still not taken into account. Additionally, we can see on the right that coarsening time is increased by roughly 20% in the geometric mean. Because of this, we did not include this adaptation in our final configuration.

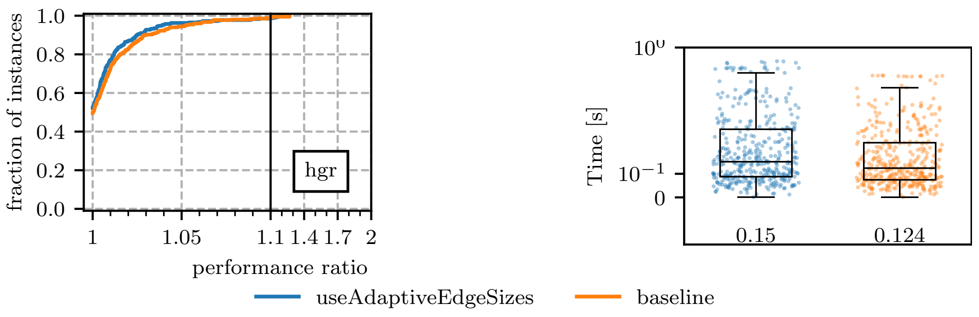

Figure 5.14: Impact of using adaptive edge sizes in coarsening on the final solution quality.

6 Experimental Evaluation

First, we present the experimental Setup in Section 6.1, which we used to conduct our experiments. After that, we evaluate different configurations and optimizations regarding our deterministic Jet implementation in Section 6.2. In Section 6.3, we present the results of several improvements and parameter-tuning experiments for the deterministic-coarsening phase. Finally, we evaluate our final configuration in Section 6.4, including scalability experiments and runtime shares. We compare our deterministic partitioning algorithm with improved coarsening and deterministic-Jet refinement (`Mt-KaHyPar-DJet`) to the current implementation of `Mt-KaHyPar-SDet`, the non-deterministic configuration `Mt-KaHyPar-D` and a non deterministic Jet implementation `Mt-KaHyPar-Jet` in Section 6.4.

6.1 Setup and Methodology

We implemented the algorithms described in Section 4 and 5 in the shared memory partitioning framework Mt-KaHyPar. It was originally designed for partitioning hypergraphs but contains optimized data structures for dyadic graphs, as well. The code is written in C++ and compiled using g++-11.4 with the flags -O3 -mtune=native -march=native. Parallelization is implemented with work-stealing using the TBB library [42].

Methodology. Due to time constraints, we use different parameters for the parameter tuning (Section 6.2 and final experiments Section 6.4). For the parameter tuning experiments we use parameters $k \in \{2, 8, 16, 64\}$ and perform 3 repetitions for each instance (graph combined with the number of blocks k) using different random seeds. We do more extensive experiments for the final configuration, using $k \in \{2, 8, 11, 16, 27, 64, 128\}$ and performing 5 repetitions per instance using random seeds. When comparing different configurations, we first calculate the arithmetic mean over all seeds to summarize the results for the objective function ($\lambda - 1$ and edge cut for hypergraphs and graphs, respectively) and running time. We aggregate the running times over multiple instances using the geometric mean.

Instances. We use two sets for hypergraph experiments: Set A_{hg} consists of 100 hypergraphs with unit hyperedge and vertex weight. It was originally chosen by Schlag [46] as a subset of a much larger set of 488 hypergraphs [32] to produce the same qualitative results. For a detailed overview, we refer to the original publications. We use this set to conduct our parameter-tuning experiments.

We run the experiments involving the evaluation of our final configuration and the scalability of our implementation on a different benchmark set consisting of 94 large hypergraphs, which we call set B_{hg} in the following. We refer to the original paper by Gottesbüren et al. [25] for a more detailed description.

The parameter tuning set A_{g} consists of 38 irregular and 33 regular graphs. It is made up from graphs generated by a compression tool using the *recompression* technique [33] on texts from the pizza&chilli corpus[1], from the 10th DIMACS implementation challenge, as well as a subset of the medium benchmark set from the dissertation of Heuer [30].

Set B_{g} consists of the 38 highly irregular graphs and the 33 fairly regular graphs from the sets I and R originally compiled by Maas et al. [39]. We use this set for our scalability experiments and the evaluation of the final configuration in Sections 6.4.1 and 6.4.

System. Due to the limited availability of machines, we perform preliminary and parameter-tuning experiments on a different machine consisting of two Intel Xeon E5-2683 16-core processors clocked at 2.1 GHz. It has 512 GB of main memory, 40 MB of Cache, and runs Ubuntu 20.04.6 LTS.
We run the experiments on sets B_{hg} and B_{g} on a machine using an AMD EPYC Rome 7702P processor with 64 cores, running at 2.0-3.35 GHz with 1024GB RAM and 256MB L3 cache. It runs Ubuntu 20.04.6 LTS Linux as an operating system.

Performance Profiles. To visualize the performance of different algorithms, we use a measure called *performance profiles*, which was first introduced by Dolan and Moré [12]. We compare the quality of the results produced by the different algorithms using the $(\lambda - 1)$-metric. In the plots, each colored line represents a different algorithm or configuration. The x-axis displays the quality τ relative to the best result of all algorithms in the graph, which we call *performance ratio*. The y-axis displays the fraction of instances for which the quality, compared to the best partition, is worse by a factor of τ or less. For example, an algorithm with a value of 0.75 on the y-axis for a value of 2 on the x- axis, i.e., the performance ratio $\tau = 2$. This means that the quality of the resulting partitions of the algorithm are worse than the best by a factor of up to 2 on 75% of the instances. The y-axis for $\tau = 1$ shows the fraction of instances the algorithm produced the best results out of all the algorithms in the plot. Therefore, when comparing two algorithms, if the fraction of instances is 0.5 for $\tau = 1$, both algorithms produce the best partition on an equal amount of instances. Generally speaking, an algorithm performs better than others if its line in the plot is above the others. If needed, we divide the x-axis into three parts. The first part goes up to 1.1, showing the fraction of instances where the algorithm is within 10% of

[1]`http://pizzachili.dcc.uchile.cl/`

the best algorithm. The second one goes up to 2, showing instances of medium quality in comparison. The last section, displaying the worst instances, uses a logarithmic scale for all remaining instances.

6.2 Parameter Tuning: Deterministic Jet

We conducted parameter tuning experiments for the various hyper-parameters of our deterministic Jet implementation. First, evaluate different configurations for the deterministic-Jet refinement algorithm and explain our choices for the final configuration. Afterwards, we compare different choices for the hyper-parameters of the deterministic rebalancing algorithm in Section 6.2.1.

In this section, we take a look at the various hyper-parameters of our deterministic-Jet implementation and evaluate them based on overall solution quality and their impact on running time. To this end, we conducted extensive experiments on the parameter-tuning benchmark sets A_{hg} for hypergraphs and A_{g} for graphs. First, we evaluate different values for τ_c and τ_f revolving around negative gain moves. We call these values *temperatures*. Afterward, we investigate how using multiple different temperatures r_τ can improve the solution quality and improve our implementation's performance by reducing the number of iterations without improvements I_{wi}. Finally, we take a look at an unsuccessful idea revolving around repeating the afterburner calculation I_{ab} times.

Allowing Moves With Negative Gain. As mentioned in Section 4.1, allowing moves that worsen solution quality, i.e., with negative gain, is an important part of the strategy to escape local minima and to improve upon standard label propagation. We use the parameter τ_c for the coarse levels and τ_f for the fine level of the refinement, to control which moves are suitable candidates. Remember that for $\tau = 1$, all vertex moves are viable candidates, independent of their calculated gain. For $\tau = 0$, on the other hand, only vertices with non-negative gain are considered. The authors of the original Jet paper suggest static values of $\tau_c = 0.75$ and $\tau_f = 0.25$ for their algorithm exclusively for dyadic graphs. When retuning these values for our implementation on graphs as well as hypergraphs, we noticed that these values are not optimal choices for hypergraphs. Note that in our final configuration, we use multiple values for both parameters to improve solution quality further. However, we want to highlight the different behavior on hypergraphs with the following experiments. In the following, each run uses only one value.

We performed a grid search with values for $\tau_c = \{0.8, 0.6, 0.4, 0.2, 0.0\}$ and $\tau_f = \{0.0, \ldots, \tau_c\}$. Figures 6.1 and 6.2 show the most relevant results. In the first figure, we can see the results for $\tau_c = 0.8$, which shows that for graphs, the choice of τ_f barely influences solution quality. The only exception is $\tau_f = 0.8$ on regular graphs, which performs significantly worse, producing solutions worse than 10% of the best solution for more than 15% of the instances. More importantly, we see an apparent improvement for the hypergraph instances the smaller τ_f becomes. The configuration with $\tau_f = 0.0$ is at most 6% worse than the best solution found by all the configurations.

We can also see different behaviors for graphs and hypergraphs when varying the parameter for coarse levels. Figure 6.2 illustrates how increasing τ_c worsens solution quality for hypergraphs while simultaneously improving solution quality for regular graphs. For hypergraphs, the configuration with $\tau_c = 0.2$ barely improves upon τ_c, while for both irregular and regular graphs, $\tau_c = 0.8$ is clearly the best configuration.

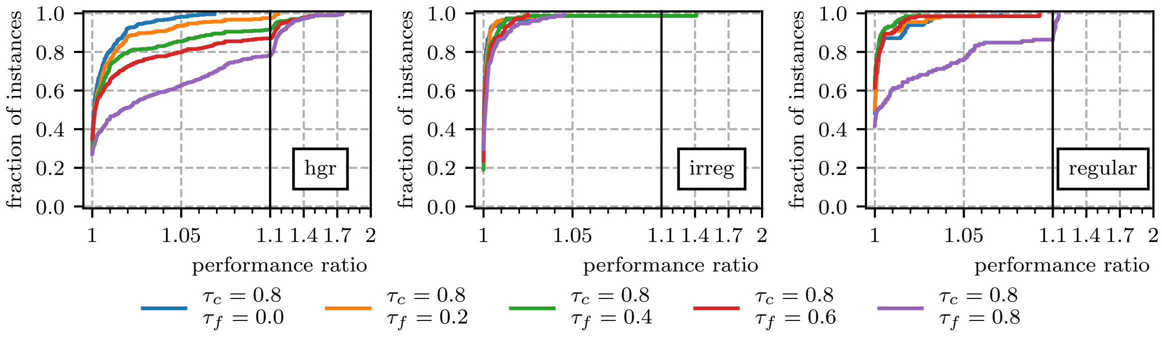

Figure 6.1: Comparing the overall quality for $\tau_c = 0.8$ and varying values of τ_f.

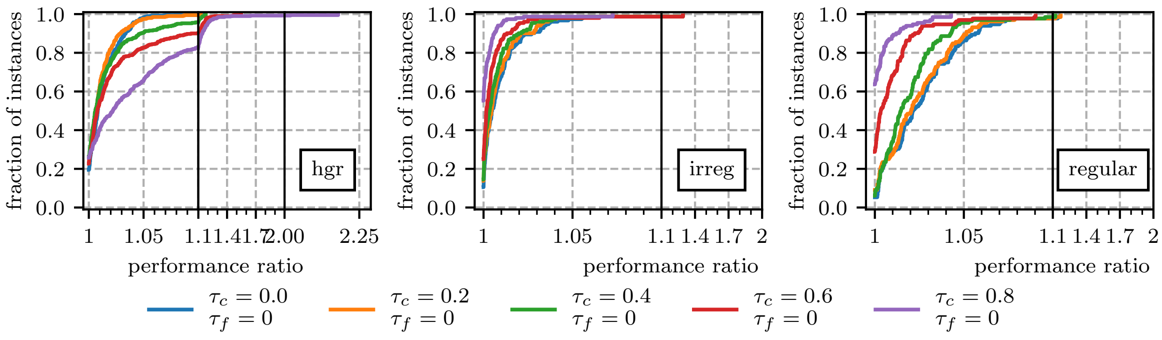

Figure 6.2: Comparing the overall quality for $\tau_f = 0.0$ and varying values of τ_c.

Variable Temperatures. We saw in the previous experiment that there is no single configuration for τ_c and τ_f that works well for both hypergraphs and graphs. Furthermore, preliminary experiments showed that the best value for both parameters varies from instance to instance. Because of this, we instead use a more variable approach to hopefully combine the advantages of different parameter configurations. One important thing we learned from the previous experiments is that $\tau_f = 0.0$ seems to be essential to include for hypergraphs. Instead of doing passes of the deterministic-Jet algorithm for only one value of τ, we repeat the algorithm for multiple.

Starting with $\tau_s = 0.75$, we decrease τ by a constant amount so that the final repetition uses $\tau_t = 0.0$. We compare configurations with increasing numbers of temperatures to the static configuration using $\tau_c = 0.75$ and $\tau_f = 0.0$ in Figure 6.3. As we can see, increasing the number of rounds improves the quality of final partitions for hypergraphs(left) as well as regular graphs(right). The results for irregular graphs are not affected by this change. Interestingly, adding only one round already significantly improves the solution quality on hypergraphs. Further rounds improve the quality, but we can clearly see diminishing returns. The added rounds also significantly affect running times. As we can see in Figure 6.4, adding a single round roughly increases the time for the refinement phase by 50%. We choose to do three rounds as a compromise between solution quality and running time. Additionally, we expect an additional round compared to just two to be more robust for instances we have not experimented on.

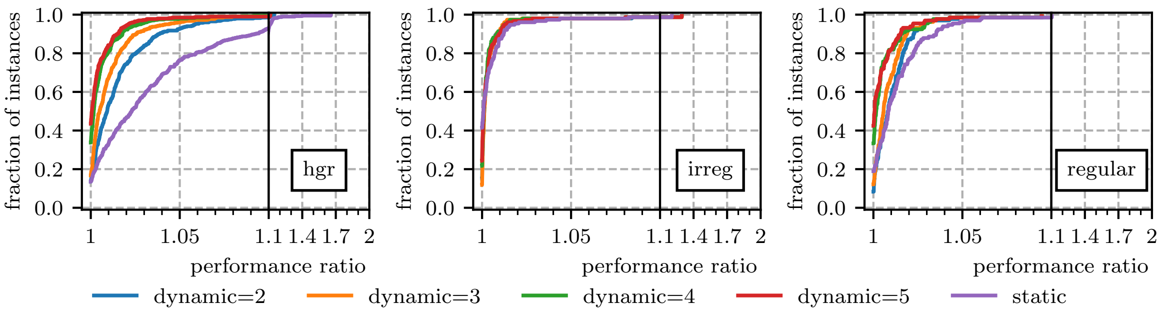

Figure 6.3: Comparing the overall quality for a varying number of temperatures $r_\tau = \{2, 3, 4, 5\}$ to the static configuration.

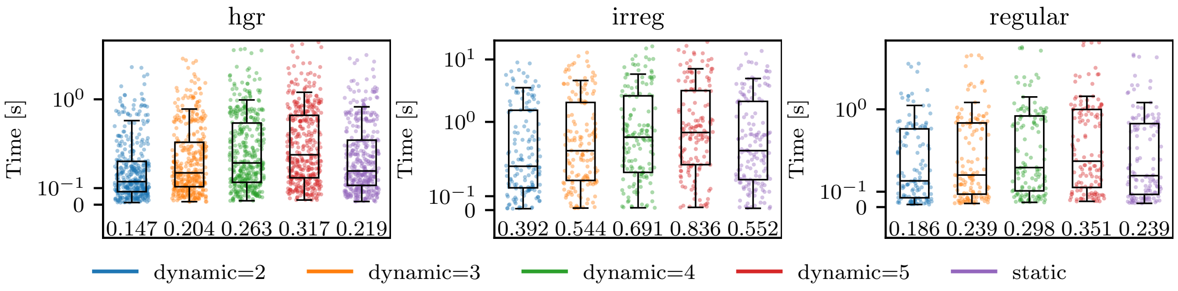

Figure 6.4: Comparing the refinement time for varying numbers of temperatures $r_\tau = \{2, 3, 4, 5\}$ to the static configuration.

Iterations Without Improvement. The deterministic-Jet algorithm repeats the refinement algorithm until I_{wi} consecutive iterations did not improve the solution quality by more than 0.1%. To compensate for the increased running time of three rounds with different temperatures ($r_\tau = 3$), we try to reduce the iterations per round I_{wi} from the initial value of 12. The impact on quality and refinement time can be seen in Figure 6.5. While only doing 6 iterations would cut the running time almost in half, we chose to do 8 iterations and, therefore, only worsen the quality by less than 5% for all instances. This still reduces the running time by roughly two thirds, so the running time of our algorithm using variable temperatures should be at most twice as slow as its static counterpart using 12 iterations.

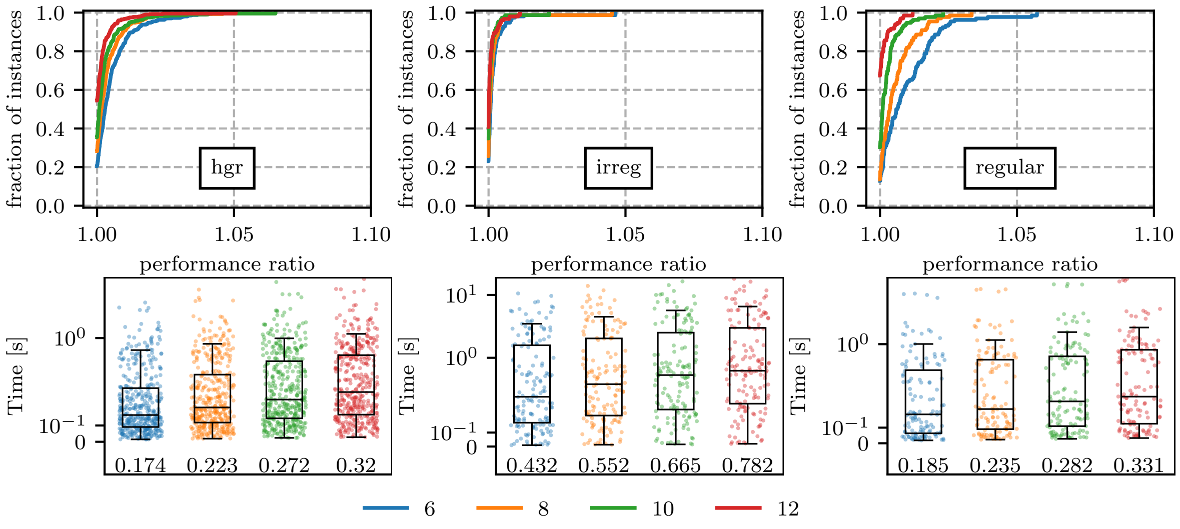

Figure 6.5: Comparing the overall quality (top) and refinement time (bottom) for different limits on the number of iterations without improvement $I_{wi} = \{6, 8, 10, 12\}$.

Iterating the Afterburner. The idea of the afterburner is to improve the move selection by predicting which vertices will be moved and recalculating the gains of vertex moves based on these predictions. It acts as a filter to remove vertex moves from the set of candidates, presumably resulting in negative gains. We observed that, especially for early levels on hypergraphs, the real gain of the calculated set of moves is often still negative. Because of this, we repeat the afterburner multiple times to filter out more moves with negative gain and hopefully further improve solution quality. To this end, we experimented with two different versions.

In the first version, the gains of the move candidates M are recalculated in each iteration of the afterburner, and all moves, no matter the gain, remain in M. We filter out moves with negative gain only after the final iteration. The results can be seen in Figure 6.6. As we can see, repeating the filter does not affect the solution quality for hypergraphs. For dyadic graphs, however, adding only a single additional iteration of the afterburner significantly reduces the solution quality for both irregular and regular graph instances.

On the other hand, the second approach immediately filters out all moves with negative gains after each iteration of the afterburner, drastically reducing the set of move candidates every iteration. The results, illustrated in Figure 6.7, are similar to the first variant. This time, increasing the repetitions slightly worsens the solution quality for hypergraphs. As we can see, the solution quality, again, gets worse the more iterations the afterburner does on dyadic graphs. Interestingly, the results for two and three iterations are exactly the same, which means that too many vertices are filtered out, resulting in the third iteration of the afterburner having no effect at all.

Overall, this approach did not improve the algorithm. We assume that the prediction of the afterburner is too inaccurate to allow iterating to be beneficial. Instead, each subsequent iteration worsens the solution quality by basing its predictions on already inaccurately predicted gain values. However, further research is needed to properly understand the effects of iterating the afterburner.

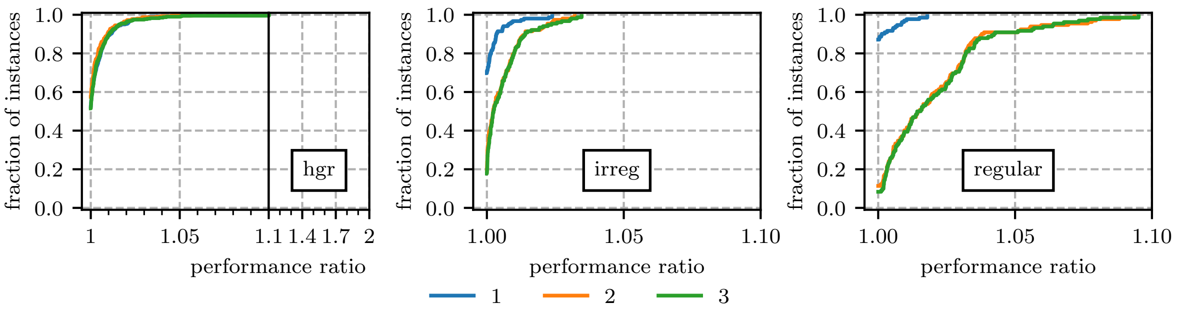

Figure 6.6: Comparing the overall quality for $I_{ab} = \{1, 2, 3\}$ while retaining all moves until the final iteration of the afterburner.

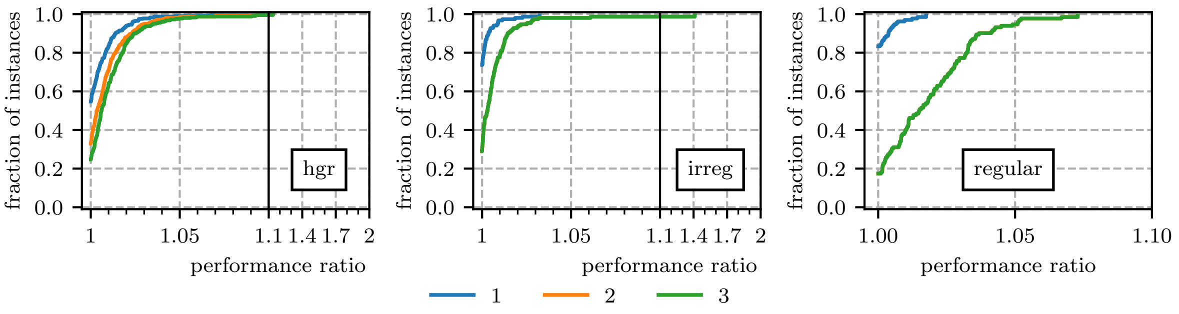

Figure 6.7: Comparing the overall quality for $I_{ab} = \{1, 2, 3\}$, while pruning all negative gain moves after each iteration.

6.2.1 Deterministic Rebalancer

Tuning Deadzone Size. As explained in Section 4.1.2 the rebalancer uses the hyperparameter c_{dead} to determine whether a part is a valid target to move vertices to. Setting the value to $c_{dead} = 1$ would exclude all parts with a weight above a perfectly balanced weight $\lceil c(V)/k \rceil$ from being a valid target, while a value of $c_{dead} = 0$ would make any non-overweight part a valid target. Choosing this parameter is a trade-off between quality and running time. Decreasing c_{dead} leads to more parts being valid targets, which makes it possible to make better moves. On the other hand, if c_{dead} is too low, non-overweight parts may frequently become overweight again, resulting in additional iterations of the rebalancing algorithm. To guarantee the termination of the rebalancer, we impose an absolute limit on the maximum number of iterations of 30.

The results are illustrated in Figure 6.8. Decreasing the value of c_{dead} improves the solution quality especially compared to $c_{dead} = 1$. Although, for graphs, all results stay within 10% of the best solution, independent of c_{dead}. For hypergraphs, there are a few outliers that result in a quality loss of more than 10% for $c_{dead} = 1$. As expected, we can see in the figure's bottom row that decreasing the deadzone's size also increases running time. This is especially true for irregular graphs, where decreasing c_{dead} to 0.1 increases the running time of the refinement phase by roughly 10% compared to $c_{dead} = 1$. This makes sense since we expect the rebalancer to do more work for irregular graphs than for other instances.

Figure 6.9 shows a direct quality comparison between the values 0.1 and 1. As we can see, there is a clear improvement in solution quality for the smaller value of c_{dead}, even for the relatively small instances in our parameter-tuning set. With the smaller dead-zone size, the algorithm computes the best solution for more than 80% of the instances across all three graph types. We expect the impact of this hyper-parameter to be even more prominent on the large instances of set B_{hg} and B_g since there is more rebalancing to do. We use $c_{dead} = 0.1$ to achieve the best quality possible in our final configuration.

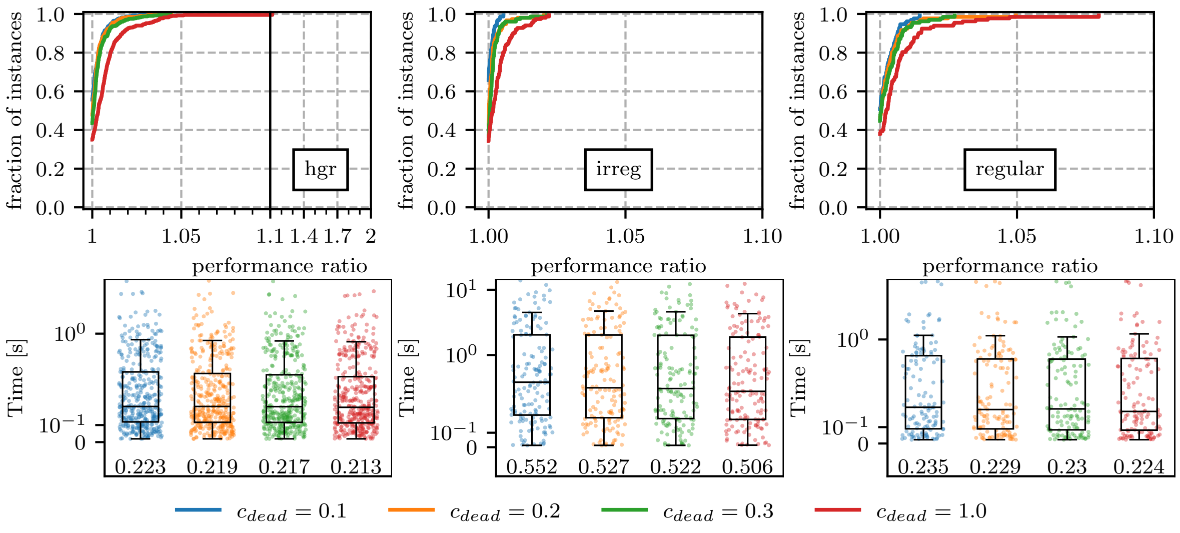

Figure 6.8: Comparing the overall quality (top) and refinement time (bottom) for $c_{dead} = \{0.1, 0.2, 0.3, 1\}$.

Including Vertex Weight in Rebalancing Priority. As mentioned in Section 4.1.2, we utilize the priority function proposed by Maas et al. [39]. It incorporates the weight of the moving vertex into the priority for rebalancing instead of using just the gain as suggested by Gilbert et al. [17].

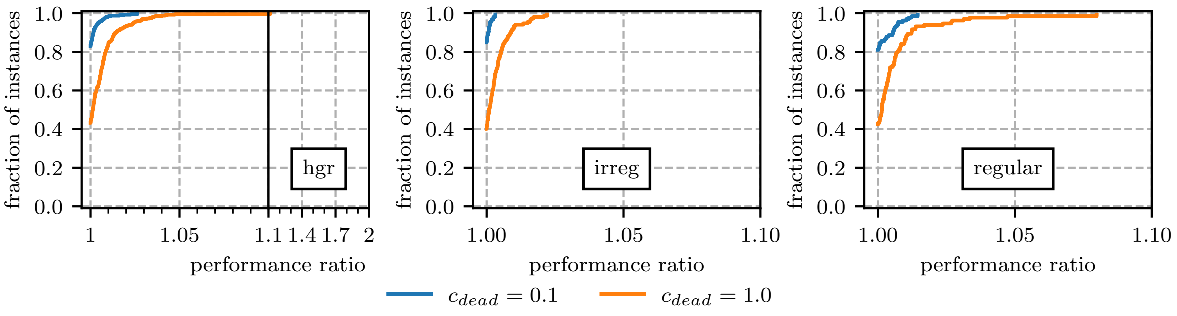

Figure 6.9: Comparing the overall quality of using a dead-zone with $c_{dead} = 0.1$ and only only excluding overweight parts ($c_{dead} = 1$).

We illustrate its impact on solution quality and running time in Figure 6.10. Including the weights improves the solution quality significantly. For all graph instances, the configuration using weights is never worse than the best solution by more than 2%. For hypergraphs, the configuration always stays within 5% of the best solution. This simultaneously helps illustrate the impact of the rebalancing algorithm since, especially for irregular graphs, the quality can worsen by up to 70% when excluding the weights from the priority function. While running times are generally close, we can see a slight improvement in the geometric mean running time of the refinement phase on the irregular graph instances (middle plot).

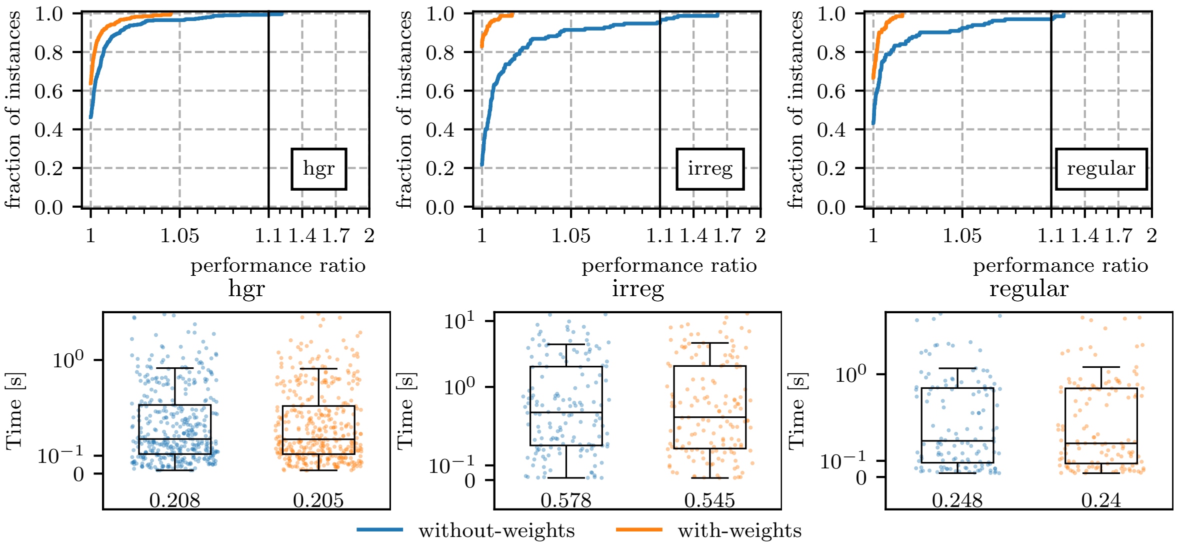

Figure 6.10: Performance profiles comparing the overall solution quality for different priority functions for rebalancing.

6.3 Parameter Tuning: Coarsening

In this section, we configure our deterministic coarsening implementation by running extensive experiments on the parameter-tuning benchmark sets A_{hg} and A_{g}. We start by evaluating different strategies to handle vertex swaps and then compare different ways of evaluating hyperedges with multiple pins in the same cluster. Here, we also revisit the prefix-doubling approach for sub-rounds on large instances to highlight its effects on the initial partition. Afterward, we present results for parameters where we were unable to improve on the original choice made for `Mt-KaHyPar-SDet` in Section 6.3.1. These involve handling moves into over-crowded clusters and tie-breaking strategies in the cluster target selection. Finally, we revisit the number of sub-rounds for the coarsening algorithm to see whether our changes to the algorithm had any effect on the original choice.

Handling Vertex Swaps. When two vertices try to join each others cluster in the same sub-round of the coarsening algorithm, the resulting moves do not improve the solution. Instead, the vertices just swap labels, which may reduce the effects of locality, which is inherently encoded in the vertex IDs. To prevent these vertex swaps, we evaluate different strategies: When applying the strategy *stay*, the swapping vertices each stay in their own cluster. This results in no progress in the clustering but at least retains the original vertex IDs. *Ignore* denotes the default of ignoring vertex swaps and just treating them as any other vertex move. The strategy *to-larger* and *to-smaller* resolve vertex swaps by moving both vertices to the heavier or lighter cluster, respectively. With the final strategy *connected-components*, we merge all vertices, forming a connected component via their calculated target clusters. Here, we do not only prevent vertex swaps, but we additionally resolve vertex moves creating a cycle or a path. These are potentially problematic since a vertex joins a cluster which is simultaneously left by its original vertex.

The results are illustrated in Figure 6.11. As we can see, there is no clear winning strategy. This does not mean, however, that none of the approaches is worth implementing. We can also see that the results vary by up to 40% across all different instance types. Because of this, we take a closer look at the strategy *to-larger*, which we use in the final configuration in Figure 6.12. While the results are very close to ignoring vertex swaps, the strategy, at least for irregular graphs, reduces the tail for a few instances and produces slightly better results for regular instances. Because of the aforementioned reason to preserve vertex IDs and the minimal impact on running time (see Figure 6.12 bottom), we chose to employ this strategy over *ignore*.

Edge-Deduplication. When calculating the heavy-edge rating for hyperedges to form the decision to which cluster to move a vertex to, the original algorithm used in `Mt-KaHyPar-SDet` counts pins belonging to the same cluster multiple times, which results in a bias toward clusters that already contain many vertices. To reduce this bias, we want to count each cluster only once. To this end, we evaluate the following strategies: As the name

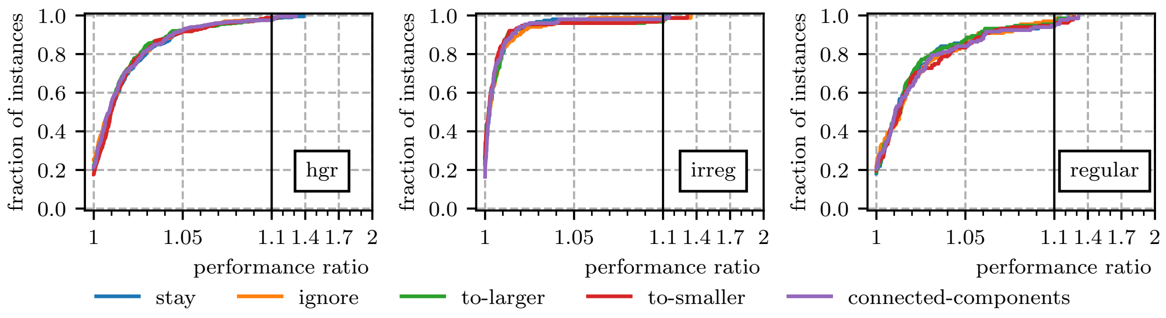

Figure 6.11: Evaluation of different strategies to resolve vertex-swaps in terms of overall solution quality.

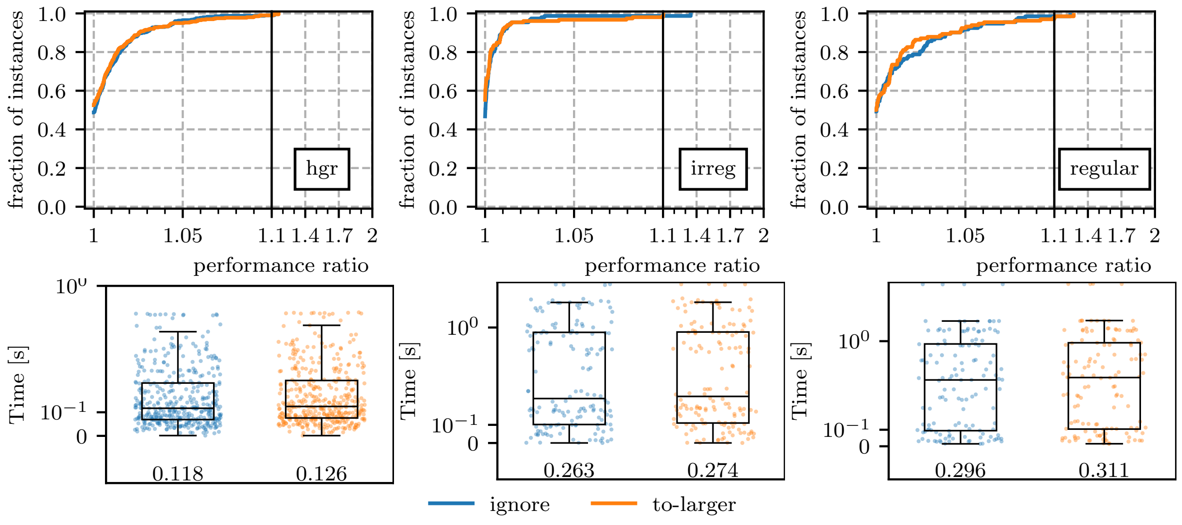

Figure 6.12: Direct comparison of ignoring vertex swaps and combining them into the larger cluster in terms of solution quality (top) and coarsening time (bottom).

suggests, the *exact* strategy keeps exact track of all clusters within a hyperedge and ensures the rating of an edge is added to each incident cluster only once. The *bloom* strategy uses a simple bloom filter to reduce the memory overhead and associated cache misses of *exact*, as explained in Section 5.2. We call the original version *count-duplicates*. The final strategy *exponential-decay* tries to compromise between *exact* and *count-duplicates*. The idea behind this strategy is to reflect some of the pin-count information in the cluster scores. To this end, additional pins from the same cluster should further improve the cluster score but with diminishing returns. Hence, each subsequent pin of the same cluster C contributes half of the score of its predecessor to the cluster rating, resulting in $\sum_{i=0}^{\Phi(e,C)-1} \text{heavy-edge}(e)/2^i$.

As we can see in Figure 6.13, the original strategy *count-duplicates* performs the worst overall. It is, on average, 5% worse than the best strategy. The strategies *exact* and *bloom* perform equally and produce the best solutions overall. Our final strategy, using exponential decay, could not improve the quality of the solution any further. Therefore, we conclude that it is clearly better to only count each cluster once per hyperedge, since any additional bias as in *exponential-decay* or *count-duplicates* results in a degradation of the solution quality. We employ the bloom-filter variant in the final configuration since it is equal in quality to the exact strategy and additionally is slightly faster.

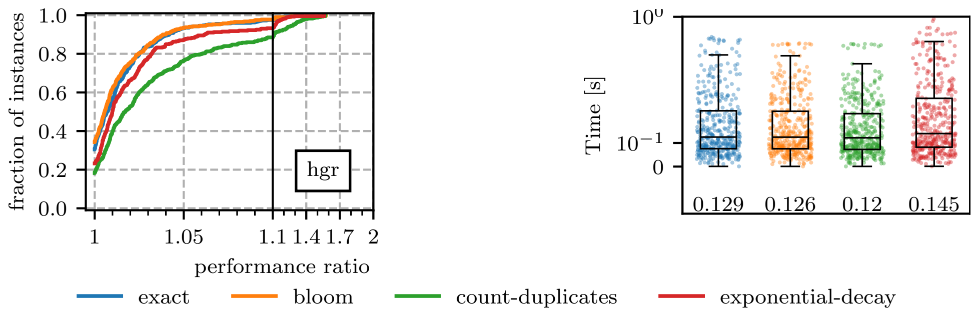

Figure 6.13: Comparing the impact on overall solution quality of different strategies to resolve duplicate clusters in edges.

Prefix-Doubling on Large Instances. Since the previous experiments on the parameter-tuning benchmark set are inconclusive, we repeat the experiment for the large benchmark sets B_{hg} and B_{g}. Figure 6.14 shows that the overall quality of the partitioning results is still very similar with prefix-doubling and without prefix-doubling. For irregular graphs, using prefix-doubling in coarsening is slightly better, whereas for regular graphs, it is slightly worse. On hypergraphs, the results are equal in quality, except for one outlier instance "circuit5M.mtx.hgr" for $k = 2$ where prefix-doubling has a worse result. However, the results for this hypergraph are very volatile between different runs for both configurations.

We examine the solution quality directly after initial partitioning since the effect of the coarsening phase may be overshadowed by our strong refinement algorithm. If we consider the cut of the partition directly after initial partitioning, as presented in Figure 6.15, we observe the same behavior on hypergraphs. However, the results are more interesting on graphs. For irregular graphs, prefix-doubling finds the best solution for two thirds of the instances. The difference between the two configurations is the most extreme on regular graphs. Here, prefix doubling produces initial partitions with significantly better solution quality than its counterpart. It finds the best initial partition for about 78% of the instances. Additionally, the results are better by at least 10% for roughly 24% of the regular graph instances.

Figure 6.16 illustrates the impact on the running time of the coarsening algorithm. As we can see, adding sub-rounds of exponentially increasing size has a considerable impact on the running time. For hypergraphs, the geometric mean time increases by roughly 24%. However, the approach is not strictly slower, e.g., the maximum coarsening time across all instances is decreased from 152s to 129s. On the graph instances, the relative impact on the running time is slightly smaller, increasing the average running time by roughly 7% and 3% for irregular and regular graphs, respectively.

As we can see, the prefix-doubling approach is able to improve the solution quality for the initial partitions significantly. While this does not translate to any substantial improvements in the final solutions in our case, it may be beneficial to use for other refinement techniques. Since the initial solution is better, the refinement phase may have to do less work, or we may even find better overall solutions. This may open the door to using slightly weaker refinement algorithms while still maintaining solution quality, which could benefit the running time of the partitioner significantly.

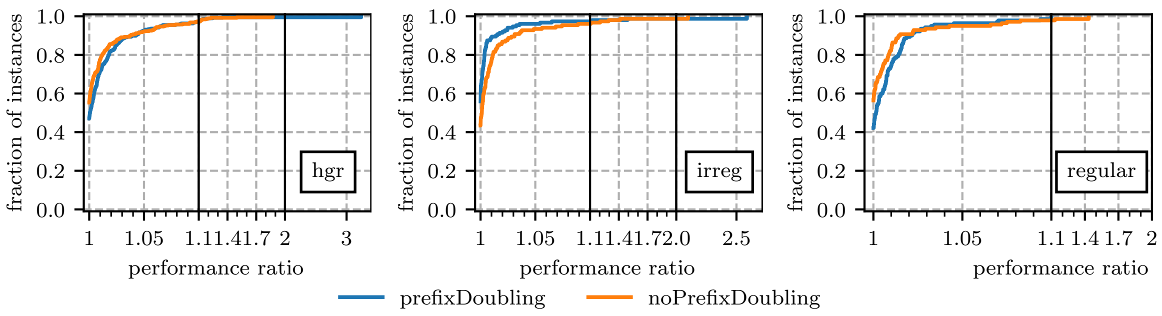

Figure 6.14: Comparing the overall quality with and without prefix-doubling to generate sub-rounds in the coarsening phase.

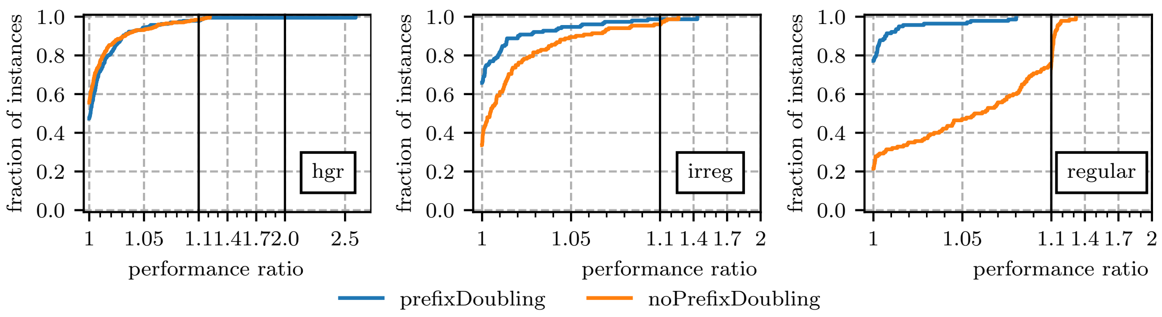

Figure 6.15: Comparing the quality after **initial partitioning** with and without prefix-doubling to generate sub-rounds in the coarsening phase.

6.3.1 Unsuccessful Changes

In this section, we explore some changes to the coarsening algorithm that did not result in an improvement of the final solution. We start by evaluating different approaches to handling moves into overcrowded clusters. Afterward, we explore different tie-breaking policies when selecting the target cluster for a move. Finally, we revisit the number of

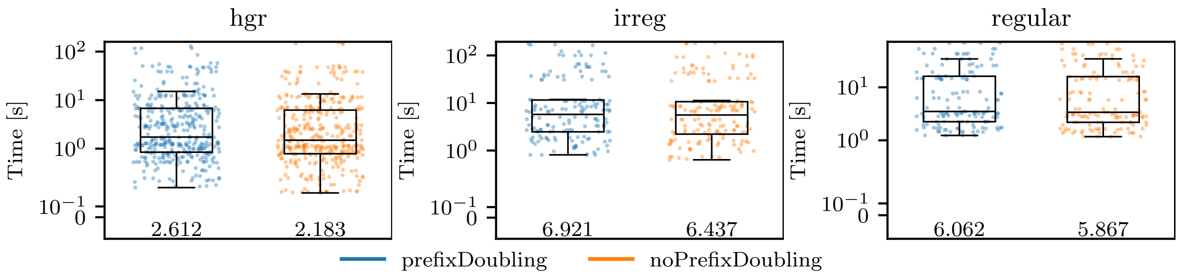

Figure 6.16: Comparing the coarsening time with and without prefix-doubling.

sub-rounds to investigate whether our changes to the algorithm influenced the previously optimal selection of 3.

Handling Moves Into Over-Crowded Clusters. When too many vertices try to join the same cluster, which would result in the cluster exceeding the maximum cluster size, we have to decide which moves to approve and which to reject. We evaluate four different strategies to handle these overcrowded clusters: The *fill* strategy is the default strategy used in the deterministic coarsening algorithm (see Section 9). We sort the vertices in ascending order according to their weight and then fill up the cluster, prioritizing lighter vertices. The next strategy, *pass-on*, does not move the vertices. Instead, they are added to the next sub-round. Another idea, similar to the previous strategy, is to *recalculate* the preferred target clusters for all vertices that want to join overcrowded clusters. This may result in different preferred target clusters because other vertices of the same sub-round have already joined their target cluster. It differs from the *pass-on* strategy since the vertices do not have to compete with the vertices of the next sub-round. The final strategy is not moving the affected vertices at all and *reset* their preferred target clusters.

The results are illustrated in Figure 6.18. All strategies produce equally good results, with the sole exception of *pass-on*, which performs slightly worse on average and calculates solutions up 60% worse on the regular graph instance “cnr-2000.graph” for $k = 2$. Because of this, we stick with the default strategy *fill* since there are no obvious advantages to any of the other strategies, and this is the most straightforward approach.

Cluster Tie-Breaking Policy. We investigate different policies to break ties if they occur when calculating the preferred target cluster of a vertex using the heavy-edge rating function. As the names suggest, *geometric* and *uniform* respectively utilize a geometric and uniform distribution to select a cluster. Policies *first* and *last* simply select the first and last cluster from the list of equal clusters, respectively.

The results in Figure 6.18 show that for hypergraphs, all policies produce results of equal solution quality. On graphs however, *uniform* produces the best results, especially for regular graphs, where it stays within 11% of the best result for all instances. All other approaches perform significantly worse on some of the instances, *last* performs worst and

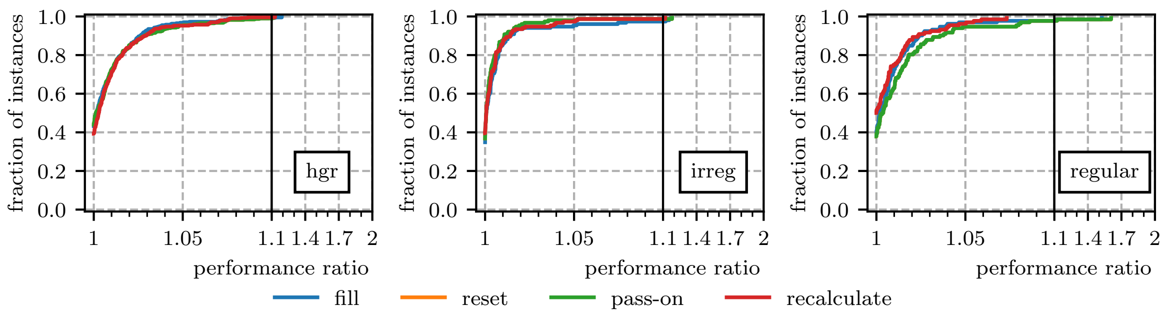

Figure 6.17: Performance profiles comparing the overall solution quality for different strategies on how to approve moves into over-crowded clusters.

is up to 63% worse than the best solution. *first* and *geometric* stay within 43% and 38% percent of the best solution on regular graphs. On irregular graphs, the different approaches behave similarly as on the regular graphs but are closer in quality overall.

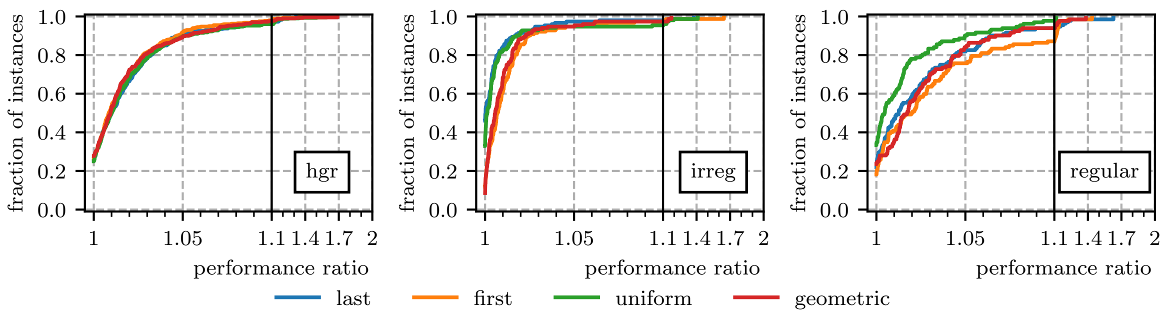

Figure 6.18: Evaluating different tie-breaking policies in the target cluster computation.

Number of Sub-Rounds. Gottesbüren and Hamann presented their rather counterintuitive result of 3 sub-rounds of synchronous local moving for the coarsening algorithm. Although we use a prefix-doubling approach for the final configuration, it is nonetheless interesting to revisit the number of sub-rounds for the coarsening algorithm and investigate whether our changes to the algorithm had any influence on the optimal number of sub-rounds.

Figure 6.19 shows the performance profiles for sub-rounds from 1 through 5. As we can see, with the exception of a single sub-round, the results are very close in solution quality. When drastically increasing the number of sub-rounds, illustrated in Figure 6.20, we see the solution quality decreasing for regular graph instances. However, increasing the number of sub-rounds barely affects the final solution quality for hypergraphs and irregular graphs.

Even with our changes, the original choice of Gottesbüren and Hamann of 3 sub-rounds still seems to be optimal when considering a static number of sub-rounds of equal size.

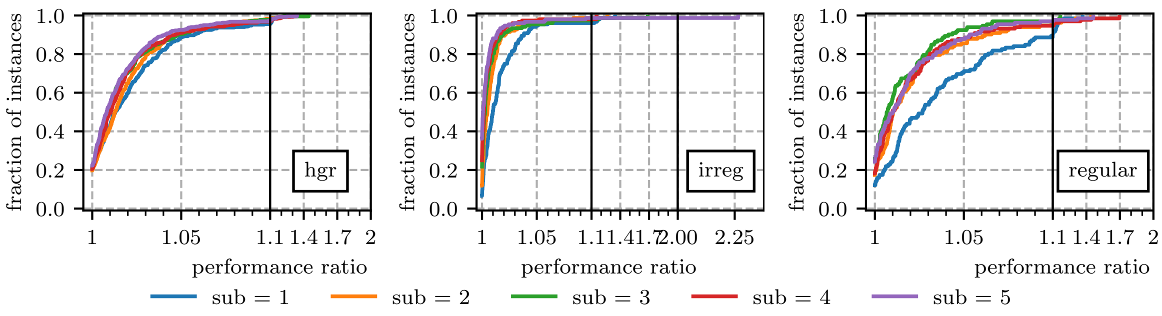

Figure 6.19: Evaluating different tie-breaking policies in the target cluster computation.

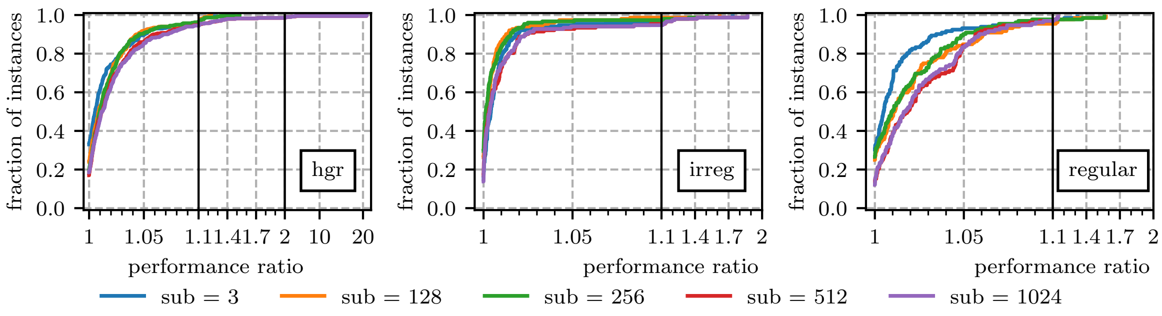

Figure 6.20: Evaluating different tie-breaking policies in the target cluster computation.

6.4 Evaluation of Final Configuration

In this section, we evaluate our final configuration in terms of both running time and quality. The experiments are run on machine B and using the large benchmark sets B_{hg} and B_{g} for hypergraphs and graphs, respectively. We conduct scalability experiments reporting good speedups, break down the contributions to the running time before finally comparing our algorithm to other state-of-the-art hypergraph and graph partitioners.

6.4.1 Scalability: Refinement

To demonstrate the scalability of our deterministic-Jet implementation, we repeated the execution of the algorithm on the top level for $p \in \{1, 4, 16, 64\}$ on sets B_{hg} and B_{g}. Due to time constraints, we had to reduce the number of random seeds per instance to 3. We illustrate the speedups of the refinement phase on the top level in Figure 6.21. Points in the plot represent the speedup for a single instance, and lines depict an aggregate rolling geometric mean with a window size of 50. We sort the instances by sequential running time to show we achieve better speedups on longer running instances.

The overall geometric mean speedups for hypergraphs are 3.74, 12.47, 24.1 for 4, 16, and 64 threads, respectively, while the maximum speedups are 6.34, 34.05, and 55.01. For irregular graphs, the speedups are worse overall, with 3.62, 11.32, and 20.77 for 4,16, and 64 threads, respectively, and corresponding maximum speedups of 4.81, 18.50, and 43.34. Finally, for regular graphs, we observe overall speedups of 3.83, 11.62, and 19.7 and maximum speedups of 5.1, 11.62, and 43.04.

We observe some super-linear speedups, especially on regular graphs with $p = 4$. Here, 24.9% of the runs exhibit super-linear speedups. For hypergraphs and irregular graphs, we observe super-linear speedups for 6.4% and 9% of the runs. Additionally, for $p = 16$, 0.9%, 0.8%, and 1.6% of the runs result in super-linear speedups for hypergraphs, irregular, and regular graphs, respectively. A possible explanation is that the order in which moves are collected in the dense vector is non-deterministic, and so is the order in which moves are executed. Therefore, different execution orders may result in varying amounts of locking. Additionally, when rebalancing, we sort the dense vectors containing vertex moves. These are again filled non-deterministically. We use library implementations for sorting, which are able to leverage already sorted runs. This can lead to variable amounts of work and may cause some of the fluctuations.

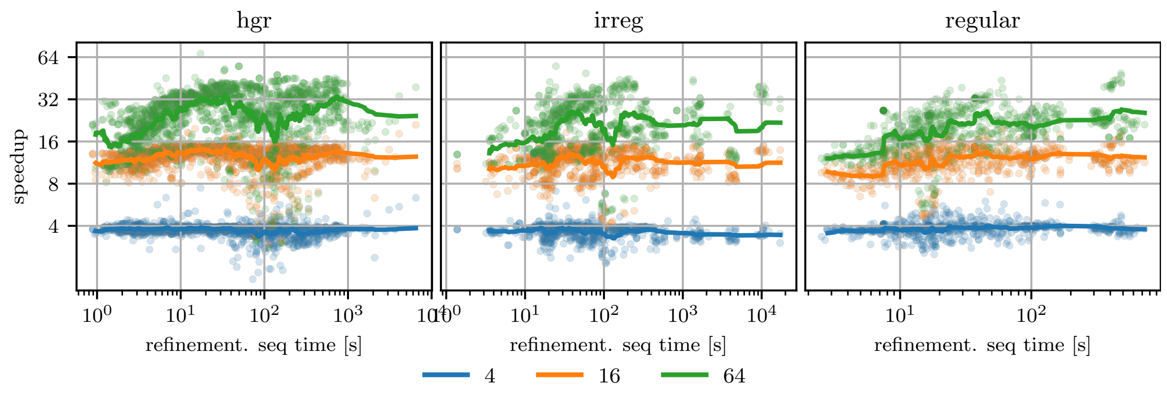

Figure 6.21: Speedup per instance (points) and aggregate rolling geometric mean with window size 50 (lines) of the deterministic-Jet refinement algorithm on the top level with $p \in \{4, 16, 64\}$ for each instance of set B_{hg} and B_g respectively.

6.4.2 Running Time Share

In this section, we look more closely at the running time. To this end, Figure 6.22 shows the running time breakdown into preprocessing, coarsening, initial partitioning, and refinement per instance. The instances are sorted by the fraction of refinement, which is generally the slowest phase. As we can see, the breakdown for hypergraphs and irregular graphs looks very similar. For both instances, the majority of the running time is spent in the refinement phase. In the geometric mean, it makes up 36% and 42% of the running time for hypergraphs and irregular graphs, respectively. This is followed by coarsening and preprocessing. On regular graphs, however, with 45%, the coarsening takes up a much larger portion of the running time, even surpassing the 40% of the refinement in the geometric mean.

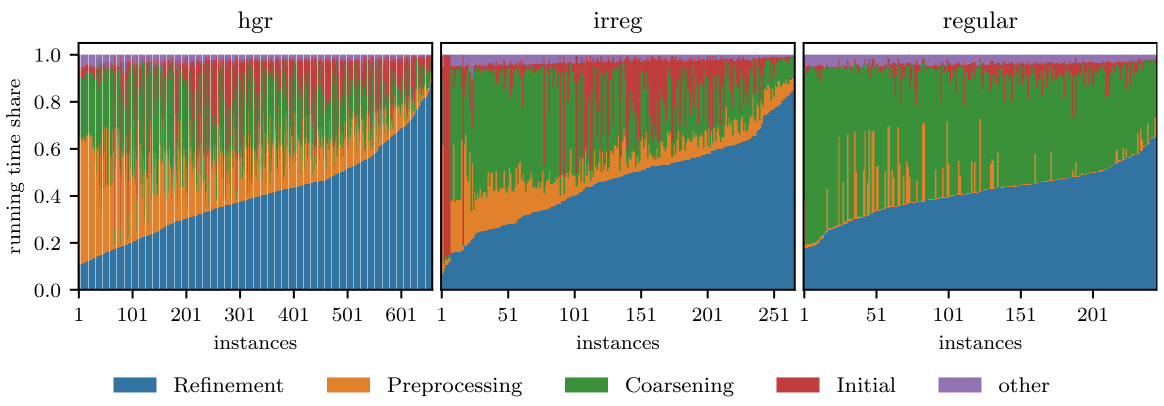

Figure 6.22: Shares of running times spent in each phase of Mt-KaHyPar-DetJet.

In Figure 6.23, we break down the running time of our deterministic Jet implementation. There are two main contributors to the running time for all three instance types: The computation of the move candidates and the rebalancing algorithm. While the running time of the move candidate computation is expected, the running time of the rebalancing algorithm is relatively long in comparison to our experiments on smaller benchmark instances. We mainly attribute this to the many iterations the rebalancing algorithm is allowed to perform. An interesting avenue for future work may be to find a way to terminate the rebalancing algorithm earlier while retaining its solution quality.

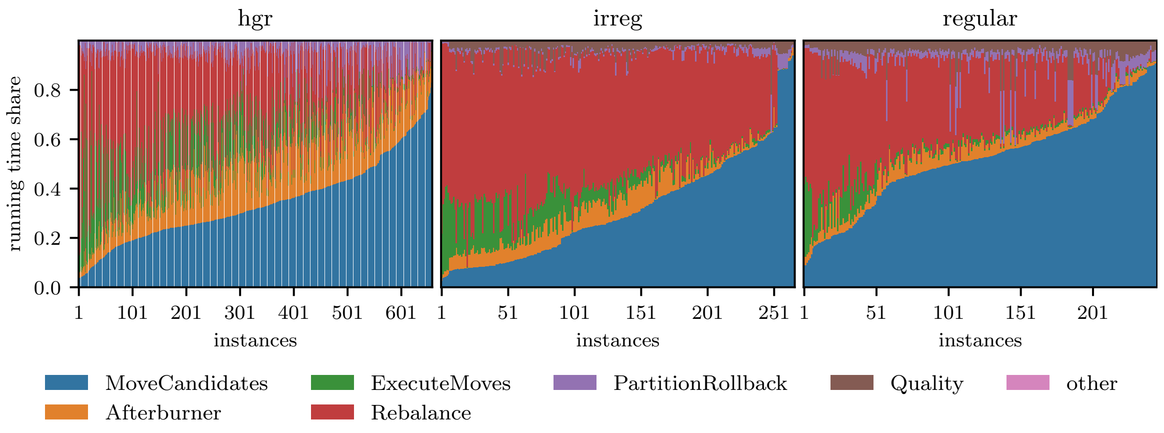

Figure 6.23: Shares of running time spent in each component of the deterministic Jet refinement algorithm.

In Figure 6.24, we can see the running time breakdown of our rebalancing algorithm. Since multiple parts are rebalanced in parallel, all measurements, except for *CalculateMoves*, are only measured for the most imbalanced part. The instances are sorted by the fraction of

CalculateMoves. As we can see, the vast majority of the running time is spent calculating the vertex moves out of the overcrowded parts. The second slowest part of the algorithm is sorting the vertex moves by their priority. In the geometric mean, this makes up 13%, 13%, and 25% of the running time for hypergraphs, irregular, and regular graphs, respectively. It is worth noting that the move execution is always negligible for regular graphs. This shows that, in this case, the rebalancer is not doing a lot of actual work. Therefore, it might be worth adding a special case treatment for almost balanced partitions to avoid the overhead of the full algorithm. For some hypergraphs and irregular graphs, the move execution can make up a significant portion of the running time. However, in the geometric mean, this contributes only 7%, 7%, and 1% of the running time on hypergraphs, irregular and regular graphs, respectively.

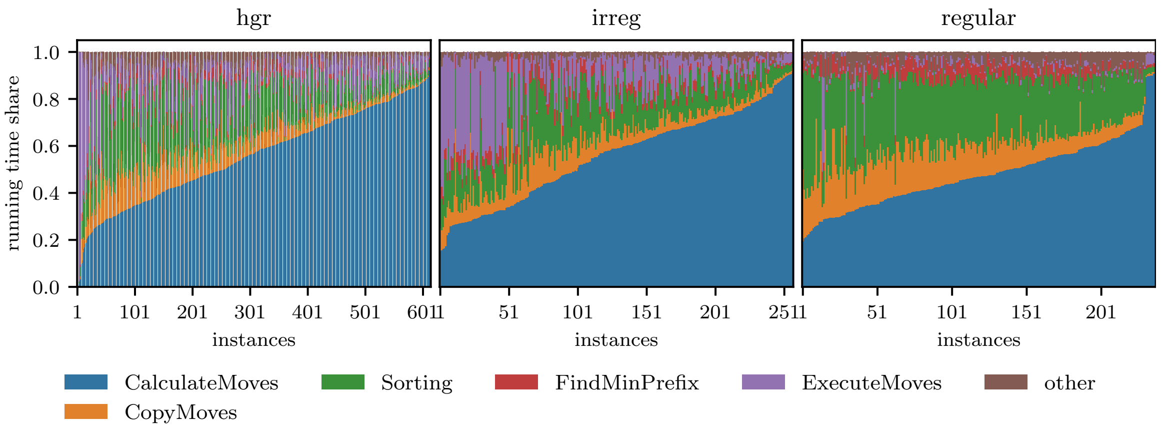

Figure 6.24: Shares of running time spent in each component of the deterministic rebalancing algorithm.

Although the running time for the rebalancing could be improved by swapping out sorting for bucketing, the most important reason for the long running time of our refinement algorithm is the many iterations it does. For each of the $r_\tau = 3$ rounds the algorithm performs at least $I_{wi} = 8$ iterations, resulting in a minimum of 24 iterations of the deterministic Jet algorithm per level of the multilevel algorithm. This is significantly more than comparable algorithms such as FM. Figure 6.25 shows the number of iterations on the top-level for the FM algorithm of `Mt-KaHyPar-D` and our deterministic Jet implementation. Note that the FM implementation has an upper limit of 10 on the number of iterations. Our algorithm performs roughly 5.5, 3.7, and 5 times as many iterations as FM on hypergraphs, irregular, and regular graphs, respectively. Notably, both algorithms perform significantly more iterations on the graph instances than on the hypergraph instances. Especially for the regular graph instances, our algorithm performs 10 additional iterations compared to irregular graphs and hypergraphs.

Additionally, we can observe that the FM algorithm hits its upper limit of 10 iterations significantly more often on the graph instances than on the hypergraph instances. Namely

87% and 83% on irregular and regular graphs compared to 56% on hypergraphs. Therefore, FM may lose some of its solution quality on graph instances by limiting the iterations too much.

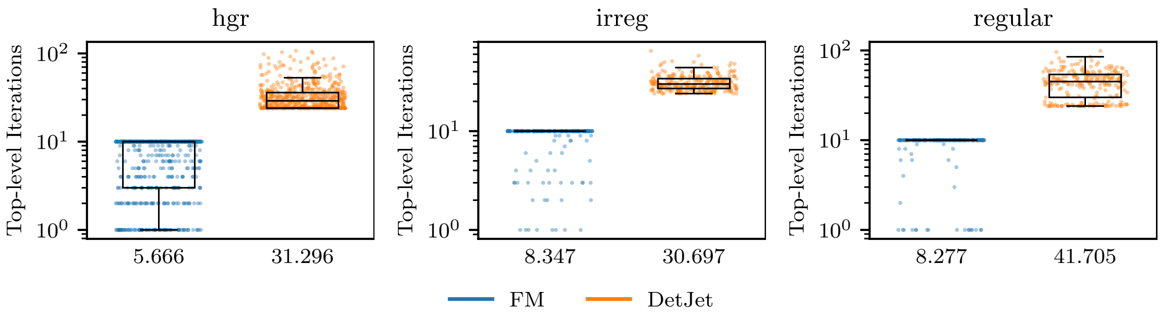

Figure 6.25: Comparing the number of iterations on the top level for FM (left) and our deterministic Jet implementation (right).

6.4.3 Comparison to Other Partitioners

In this section, we compare the final configuration of our algorithm to other state-of-the-art partitioners, in extensive experiments on the large benchmark sets B_{g} and B_{hg}. We start by comparing our new deterministic algorithm to existing deterministic and non-deterministic configurations of Mt-KaHyPar. Afterward, we compare our partitioner to other state-of-the-art sequential and parallel hypergraph and graph partitioners. Finally, we report the results for our deterministic flow-based refinement scheme.

Comparison to Relevant Configurations of Mt-Kahypar. Figure 6.26 shows performance profiles, comparing our new deterministic configuration with its non-deterministic predecessor (`Mt-KaHyPar-D`), its deterministic predecessor (`Mt-KaHyPar-SDet`) and a non-deterministic Jet implementation [39] (`Jet-w`), which we modified to utilize vertex weights when rebalancing in the same way as the rebalancers of the unconstrained FM refinement and deterministic Jet algorithms. This was done to ensure a fair comparison.

As we can see, our algorithm finds the highest quality solution out of the compared algorithms. However, the non-deterministic configuration `Mt-KaHyPar-D` performs equally well on hypergraphs. On hypergraphs, both our algorithm and the non-deterministic configuration find the best solution on about 50% of the instances. On irregular graphs, `Mt-KaHyPar-DetJet` is slightly ahead of `Mt-KaHyPar-D`, finding the best solution for 65% of the instances. Finally, the gap between our implementation and the other algorithms is the largest on regular graphs. Our algorithm finds the best solution on roughly 75% of the instances and always stays within 10% of the best solution found by all algorithms. Unlike on irregular graphs, both non- deterministic algorithms `Jet-w` and `Mt-KaHyPar-D` perform similarly on regular graphs. Due to the lack of unconstrained refinement and negative gain moves, the previous deterministic configuration Mt-KaHyPar-SDet is significantly worse on all three instance types. Its quality is off by more than 10% on more than half of the instances across all instance types.

The total running times of the algorithms are reported in Figure 6.26 (bottom), with the number depicting the geometric mean running time for each of the algorithms. We can see that our algorithm is noticeably slower than its competition. Mt-KaHyPar-DetJet is roughly 75% slower than the non-deterministic configuration and takes roughly twice as long as the previous deterministic configuration.

For plots directly comparing Mt-KaHyPar-DetJet to the other configurations of Mt-KaHyPar, see Appendix B. The direct comparison of Mt-KaHyPar-D and Mt-KaHyPar-DetJet shows that the Jet algorithm is not only able to match the solution quality of FM on hypergraphs but even find better solutions on graph instances. Especially on regular graphs, we find the best solution for 185 of the 245 instances. Therefore, we achieve our goal of closing the gap in solution quality between non-deterministic and deterministic (hyper)graph partitioning, although at the cost of increased running time.

The direct comparison to the previous deterministic configuration Mt-KaHyPar-SDet shows that our algorithm completely outperforms its predecessor regarding solution quality. We find the best solution quality on all but 23 and 4 instances on hypergraphs and irregular graphs, respectively. On regular graphs, our algorithm finds the best solution for all instances with a geometric mean performance ratio of 1.0 compared to 1.11 of Mt-KaHyPar-SDet.

Finally, comparing to the non-deterministic Jet implementation Jet-w, we were able to significantly improve upon the solution quality of Jet. For both graph benchmark sets, we find the best solution for more than 80% of the instances. Additionally, on the graph "recomp_proteins1GB_7.graph" for $k = 2$ we find a solution better by a factor of 21.77. We mainly attribute this to the significant increase in iterations, due to the multiple values of τ and to the stronger, although significantly slower, rebalancing algorithm we implemented.

Comparison to Prior Work. In Figure 6.27, we compare the quality of our algorithm to the sequential hypergraph partitioner PaToH[2] [7], as well as the parallel graph partitioners KaMinPar [22] and Mt-Metis [37].

On hypergraphs, our algorithm, as well as the non-deterministic configuration of Mt-KaHyPar, find the best solution for roughly 45% of the instances each. The sequential partitioner performs notably worse, finding the best solution for the remaining 10% of the instances. Additionally, roughly half of the solutions found are worse by 10% or more compared to the best solution found. Furthermore, PaToH-D could not finish its computations within the time limit of eight hours for four instances with $k = 128$.

[2]Experiments involving $k = \{2, 8, 16, 64\}$ are from Gottesbüren and Hamann [23] and were run on a different machine consisting of an AMD EPYC Zen 2 7742 CPU clocked at 2.25GHz (3.4GHz turbo boost) with 1TB DDR4 RAM, and 256MB L3 cache.

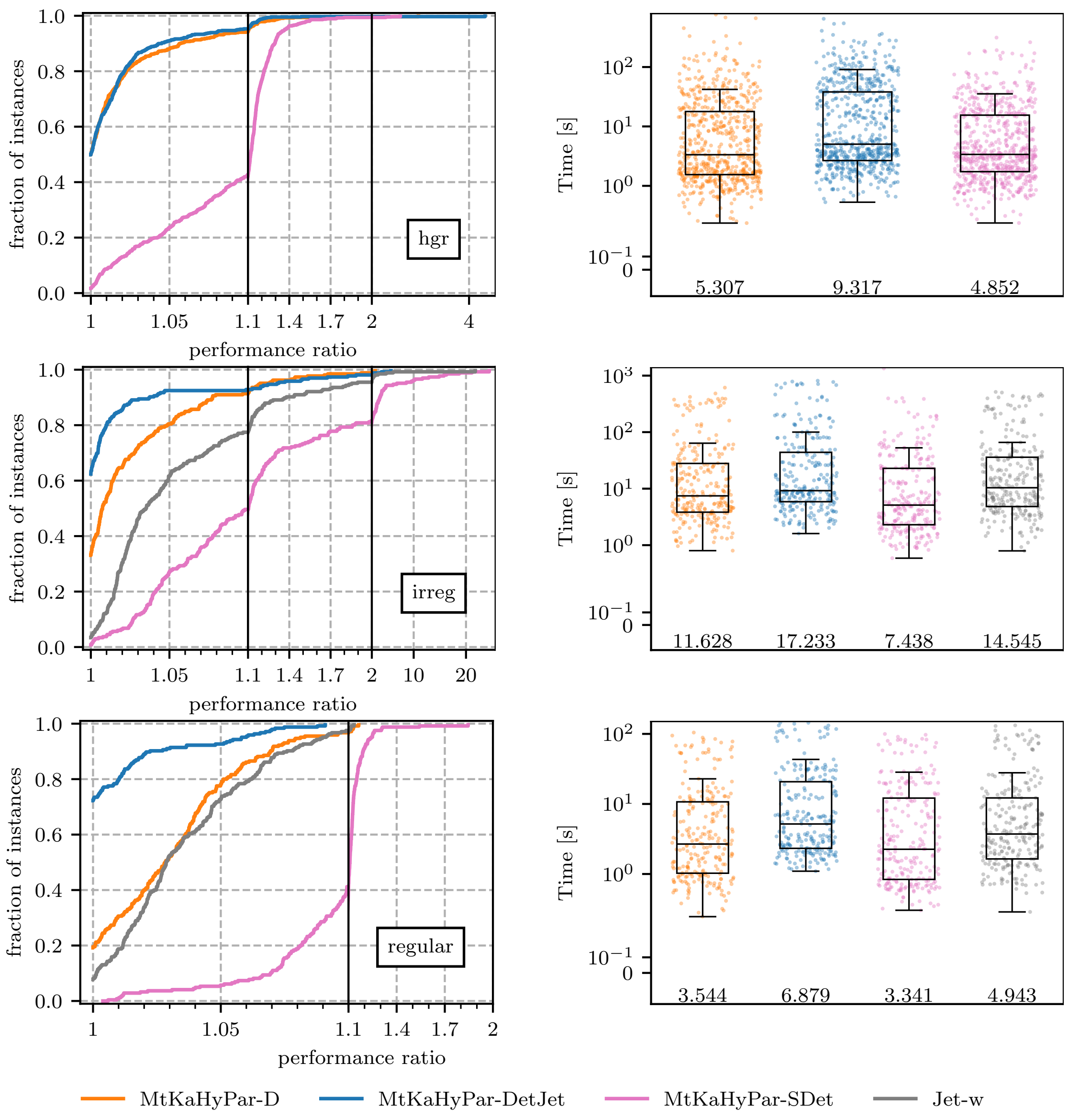

Figure 6.26: Comparing the overall quality (left) and running times (right) of different configurations of Mt-KaHyPar

On irregular graphs, our algorithm performs the best out of the considered partitioners. `KaMinPar` produces notably worse solutions with a gmean performance ratio of 1.550 compared to 1.041 and 1.039 of `Mt-KaHyPar-D` and `Mt-KaHyPar-DetJet` respectively. `Mt-Metis` struggles to find balanced solutions. Out of 1330 experiments (graph, k, seed), the algorithm only finds a balanced solution for 231. Additionally, it fails to find any solution for 223 of the runs.

On the benchmark set for regular graphs, our algorithm performs the best overall. We find

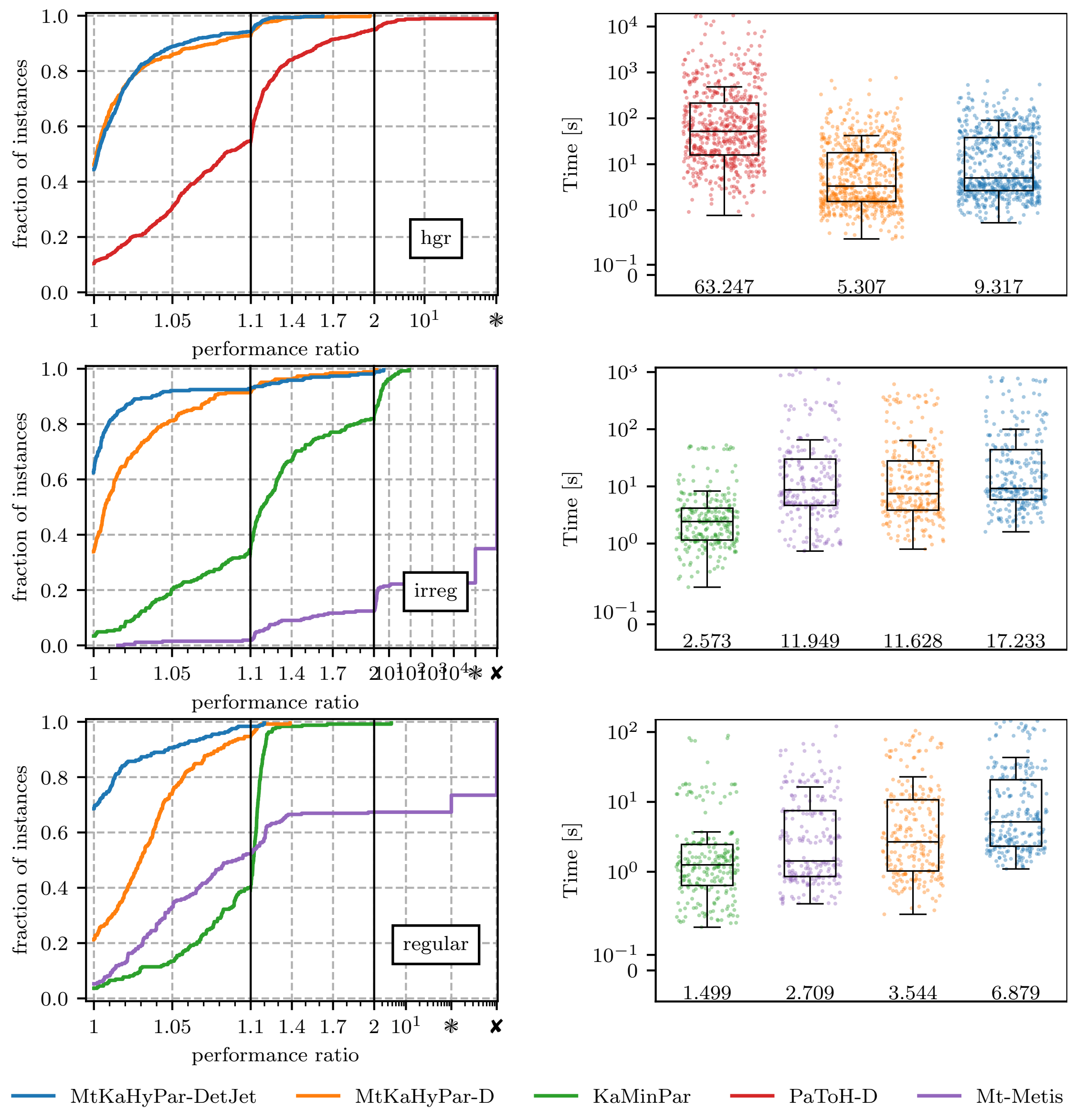

Figure 6.27: Comparing the overall quality(left) and running time (right) of the state-of-the-art non-deterministic and deterministic hypergraph partitioners.

the best solution for roughly 69% of the instances (graph, k), followed by `Mt-KaHyPar-D`. The solutions found by `KaMinPar` and `Mt-Metis` are significantly worse. Compared to our algorithm, `KaMinPar` produces edge cuts that are 11.6% worse in the geometric mean. Again, `MtMetis` does not find a solution for all of our test instances. The algorithm fails to produce any solution for almost 10% of the experiments and only finds a balanced solution in 661 of the 1225 experiments we ran.

Regarding running time, the parallel configurations outperform the sequential partitioner

PaToH. Our algorithm is roughly 6.1 times faster with 64 threads. Most notably on the graphs instances, KaMinPar is significantly faster than all competitors. With its geometric mean running time of 2.573 seconds, it is roughly 6.7 times faster than our algorithm and still roughly 4.5 faster than both other non-deterministic partitioners. This behavior is very similar on regular instances where KaMinPar is, again, significantly faster than all competitors. However, the gap in running time is considerably smaller. With a geometric mean running time of 6.879, our algorithm is the slowest out of the competition, which is expected since we do many iterations to ensure our superior solution quality. However, comparing the slowest run of our algorithm to the slowest run of KaMinPar, our algorithm only takes roughly 69% longer than the 90 seconds of KaMinPar.

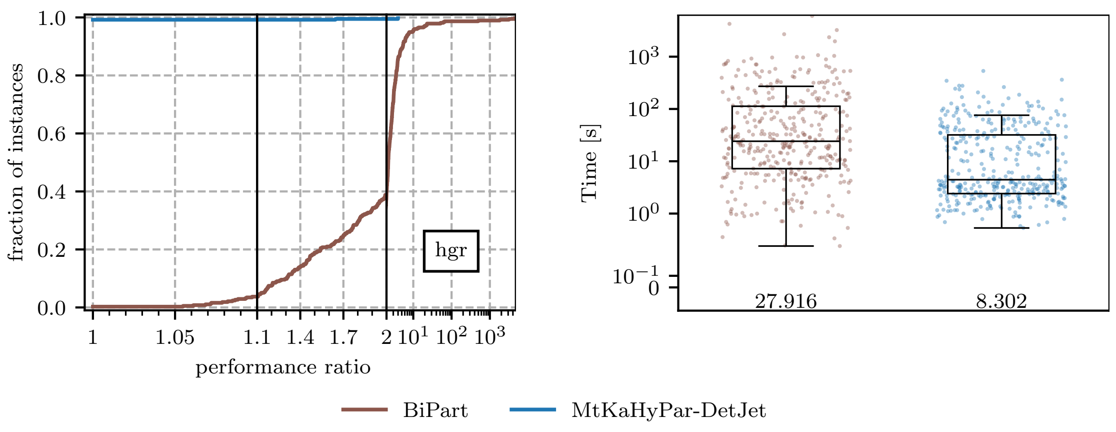

Figure 6.28: Comparing the overall quality (left) and running time (right) to deterministic partitioner BiPart on Hypergraphs with $k = \{2, 8, 16, 64\}$.

We compare our deterministic hypergraph partitioner to the only other deterministic parallel hypergraph partitioner BiPart in Figure 6.28. Unfortunately, we were unable to get the algorithm running on our benchmark machines since it would always produce segmentation faults. Because of this, we fall back to results from Gottesbüren and Hamann [25], which were performed on a similar machine. The machine consists of an AMD EPYC Zen 2 7742 CPU clocked at 2.25GHz (3.4GHz turbo boost) with 1TB DDR4 RAM and 256MB L3 cache. Because of this, the comparison in terms of running times is not perfect but still reflects the general difference. Furthermore, the experiments only include benchmark set B_{hg} and $k = \{2, 8, 16, 64\}$, since those are the only values for k for which we have results for both algorithms. As expected, BiPart is outperformed by our algorithm in terms of solution quality and running time. We produce the best partition for all but two instances while being roughly 3.36 times faster. Furthermore, we find significantly better solutions, resulting in a gmean performance ratio of 1.005 compared to 2.632 of the competition.

Comparing Flow-Based Approaches. Due to time limitations we evaluate our configuration using deterministic flow-based refinement (Mt-KaHyPar-Det-F) with fewer parameters than the previous experiments. We only use three instead of 5 seeds. Additionally, we removed the three slowest hypergraph instances from B_{hg}, which are "sk-2005.mtx.hgr", "uk-2005.mtx.hgr" and "it-2004.mtx.hgr". Furthermore, we impose a time limit of two hours per run. Finally, we ran out of time running the experiments for the irregular graph instances of B_{g}, which is why we only present the results on hypergraph and regular graphs.

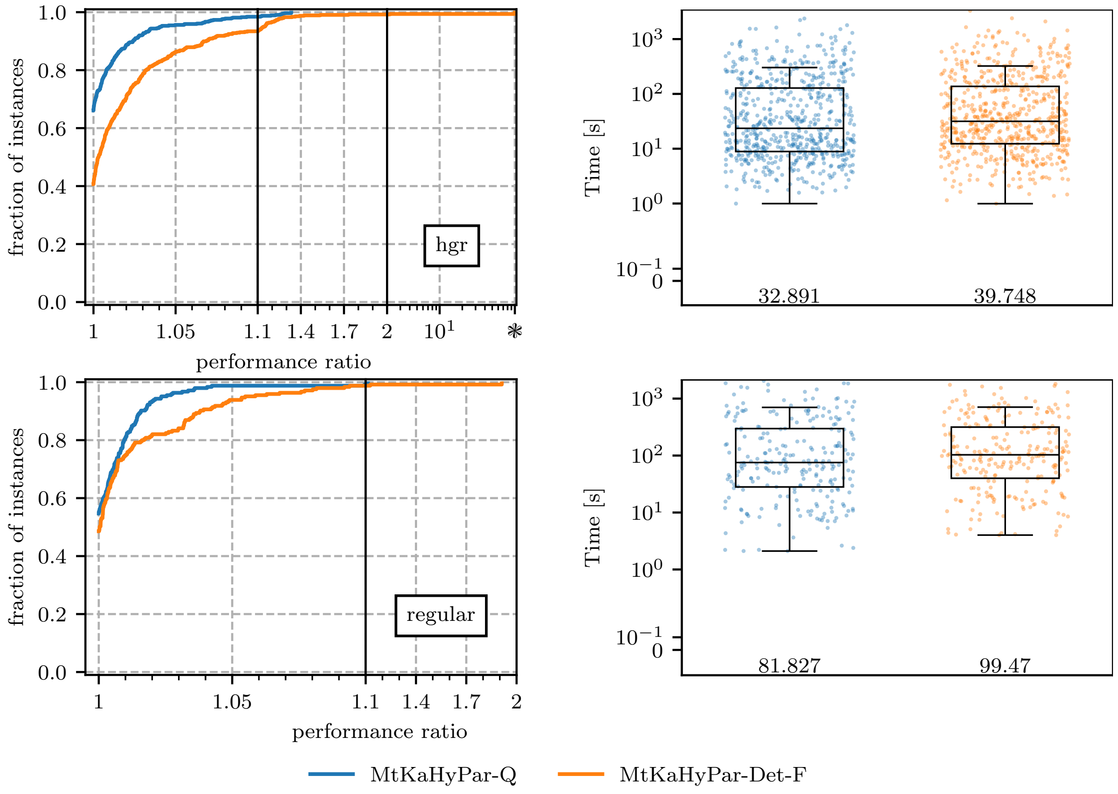

Figure 6.29: Comparing the overall quality(left) and running time (right) of all relevant configurations of Mt-KaHyPar.

Figure 6.29 illustrates the results regarding quality on the left and running time on the right. The quality of our deterministic approach is slightly worse than of the non-deterministic configuration `Mt-KaHyPar-Q` using flow-based refinement. However, the geometric mean performances of 1.009 and 1.025 for `Mt-KaHyPar-Q` and our algorithm, respectively, reveal that the quality is very close. Additionally, we find the best solution for 260 out of the 637 instances. Unfortunately, our algorithm was unable to find a solution within the given time limit for the instances "sat14_9vliw_m_9stages_iq3_C1_bug8.cnf.dual.hgr" for $k = \{64, 128\}$ and "sat14_velev-npe-1.0-9dlx-b71.cnf.dual.hgr" with $k = 128$ which we can see by the long tail in the top left.

The results on regular graph instances draw a similar picture. Our algorithm is slightly worse than its non-deterministic counterpart. We find the best solution for roughly 51% of the instances and are worse by more than 10% for only 2 instances with $k = 2$.

Overall, the quality of the results are very good and the slight drop in quality compared to the non-deterministic configuration is expected due to its more complex scheduling. We provide additional plots, which put the flow-based results in the context of previous configurations in Appendix C.

Looking at the running times the results are very promising since our algorithm, despite its limit of only scheduling $k/2$ block pairs simultaneously, is not much slower than its non-deterministic counterpart. Our algorithm is only roughly 21% slower in the geometric mean on both hypergraphs and regular graphs. However, keep in mind, that our algorithm did run into a few timeouts, whereas the non- deterministic configuration was able to find a solution within the time limit for all instances. This however looks more like an error in our code than a limitation of deterministic refinement, since the affected instances are generally fast.

7 Conclusion

In this work, we fully closed the gap in terms of partition quality between non-deterministic and deterministic hypergraph partitioning algorithms. To this end, we presented the first fully deterministic version of the parallel Jet refinement algorithm [17]. Additionally, we made essential optimizations to the *afterburner* to make the algorithm viable for large-scale hypergraph partitioning. We were able to reduce the complexity of the afterburner from its initial $\mathcal{O}(|e|^2)$ to only $\mathcal{O}(|e| \log |e|)$ by effectively processing each hyperedge only once. In combination with further optimizations to the hypergraph afterburner, we were able to reduce the running time of the Jet algorithm on hypergraphs by a factor of up to 200. Furthermore, we improved the already good quality of the algorithm by employing a technique first presented by Sanders and Seemaier [45]. We repeat the core algorithm multiple times with multiple different temperatures τ to allow the algorithm to explore moves with varying amounts of negative gain. Our rebalancing algorithm leverages techniques from both the *weak* rebalancing algorithm by Gilbert et al. [17] and the rebalancing algorithm of the non-deterministic configuration of Mt-KaHyPar [39] to minimize the loss in quality.

Secondly, we improved upon the existing deterministic coarsening algorithm of `Mt-KaHyPar-SDet`. We enhanced the clustering algorithm based on synchronous local moving [29] by employing sub-rounds of exponentially increasing size, preventing vertex swaps, and changing the rating calculation to count each hyperedge only once per cluster. With these changes, the deterministic-clustering algorithm is able to match the quality of its non- deterministic counterpart. Additionally, thanks to the prefix-doubling approach to the sub-rounds, we significantly enhanced the quality of initial cuts, especially on regular graph instances.

Our flow-based refinement approach employs a deterministic parallelization of the active block scheduling by Sanders and Schulz [44], based on maximal matchings. To improve parallelism we prioritize scheduling blocks which are part of many block pairs, as propsed by Grötschla [27]. Our network construction algorithm produces deterministic results by sorting pin and incident hyperedge lists as well as deterministically removing duplicate hyperedges. Furthermore, we presented a tweak to make the selection of piercing vertices deterministic.

In our extensive experimental results, we show that our deterministic algorithm, utilizing Jet refinement, is able to match the solution quality of the default configuration of state-of-the-art parallel shared memory partitioner `Mt-KaHyPar`. Furthermore, our algorithm finds better solutions than all considered state-of-the-art graph partitioners (KaMinPar [22], Mt-Metis [37], Mt-KaHyPar-D [39], BiPart [37]) on both irregular and regular graph instances. On irregular graphs, our algorithm finds the best solution for 65% of the instances. The gap between solutions found by our deterministic algorithm and the other algorithm is the largest on regular graphs, where we find the best solution for 69% of the instances. Compared to the non-deterministic configuration of Mt-KaHyPar, we improve the solution quality by at least 5% for 20% of the instances. However, the improved solution quality comes at the cost of increased running time. Our algorithm is slower than all of the parallel partitioners, except for BiPart, taking roughly 75% more time than the non-deterministic configuration of Mt-KaHyPar. Because of this, future research should focus on reducing the running time overheads while maintaining the excellent solution quality.

The experimental results for our flow-based refinement show that a deterministic approach can reproduce the success of the quality configuration `Mt-KaHyPar-Q`. The solutions found by our algorithm are within 3% and 2%, on average, on hypergraph and regular graph instances, respectively. Even though our algorithm's parallelism is more limited than its non- deterministic counterpart, due to the matching-based scheduling, it is only roughly 21% slower on average.

7.1 Future Work

As seen in Section 6.4.2, our deterministic Jet algorithm performs significantly more iterations than the FM algorithm. This primarily stems from the number of variable temperatures τ and the number of iterations without improvement I_{wi}. In combination, we do a minimum of 24 iterations of the Jet algorithm and perform several iterations of the rebalancing algorithm for each iteration of the Jet algorithm.

The most significant improvement to the running time would be to find better ways to vary the temperatures. The design space is large: One idea could be to reduce the temperature for each consecutive round without improvement, reducing the set of move candidates in each iteration. Other ideas could include dynamic approaches to changing the temperature based on, e.g., level, improvement of the previous iterations, or the difference in quality to the best partition. This could also be an interesting area to employ techniques from machine learning, such as adaptive learning rate scheduling.

A dynamic approach to the number of iterations without improvement I_{wi} may also be beneficial to the algorithm's running time. E.g., we could prematurely terminate the algorithm if the quality strayed too far from the currently best solution and is not expected to recover the quality based on the gains of previous iterations.

Furthermore, a lot of the running time is spent rebalancing. Improving this part of the algorithm could also greatly benefit the running time of the algorithm. Again, reducing the number of iterations is crucial to reduce its running time. Therefore, different approaches to stopping the algorithm early while guaranteeing the balance of the partition should be investigated. Another idea could be to allow a small amount of imbalance after the call to the rebalancer to save some iterations and only fully balance the solution in the final iteration of the refinement algorithm. Although not the largest portion of the rebalancing time, the time spent sorting could be reduced by bucketing the move candidates instead. Only the last bucket would have to be sorted to ensure determinism and use of the best moves available.

While we have presented all algorithmic components to achieve fully deterministic flow-based refinement, our code unfortunately still contains some errors, making it non-deterministic. Finding and eliminating these bugs is of highest priority. Afterward, future research should focus on improving the deterministic scheduling, since our approach is limited to scheduling $k/2$ block pairs at a time. An approach scheduling all block pairs of the same round similar to the scheduling of the non-deterministic configuration Mt-KaHyPar-Q [21]. The difficulty with this approach is that multiple threads may operate on the same block resulting in conflicts. Resolving these conflicts deterministically is of great importance to improve the scalability of the deterministic flow-based refinement scheme.

A Results by k

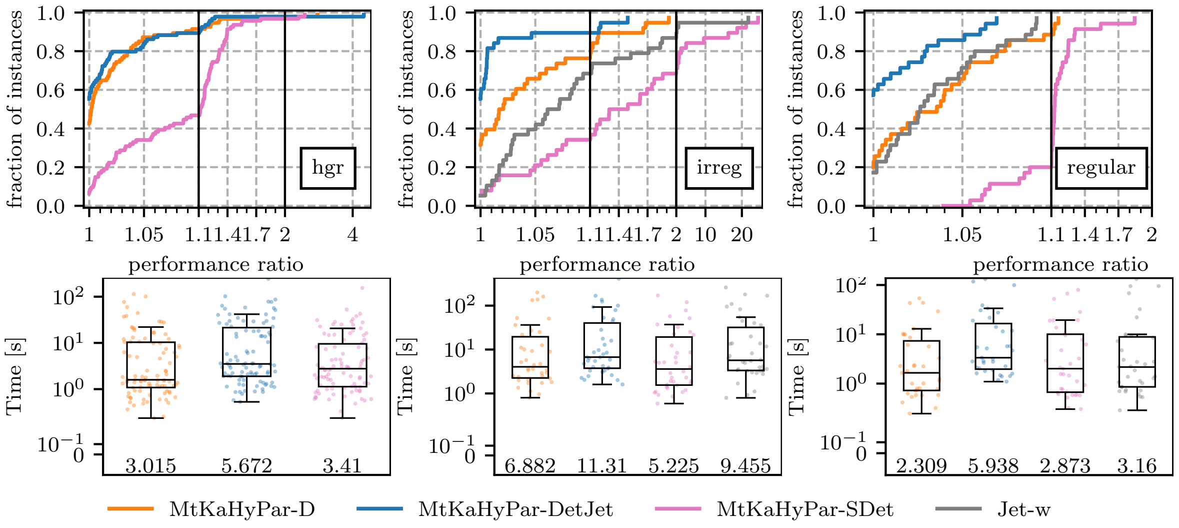

Figure A.1: Comparing the overall quality(top) and running time of the state-of-the-art non-deterministic and deterministic hypergraph partitioners for $k = 2$.

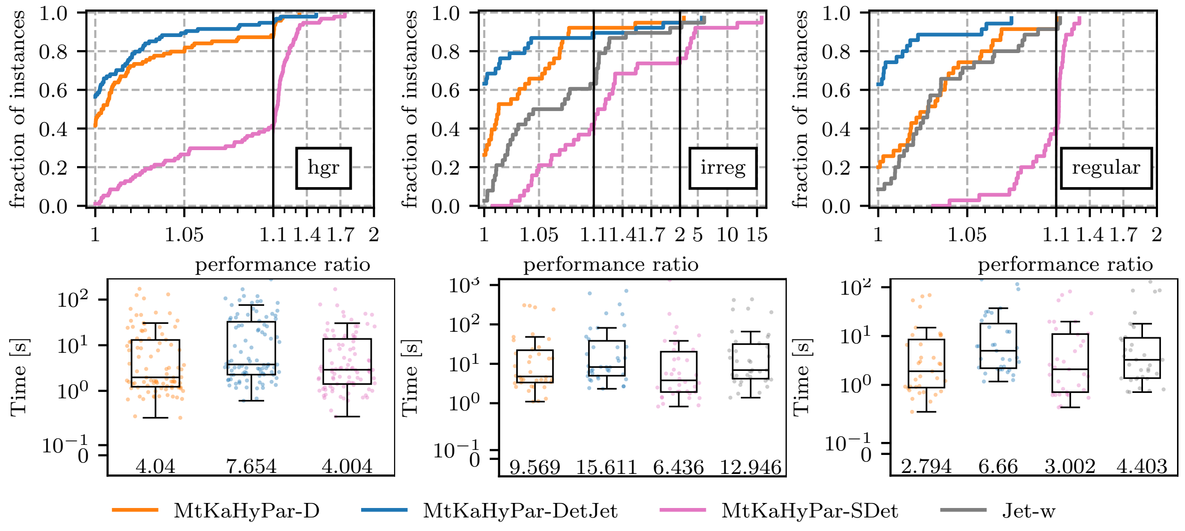

Figure A.2: Comparing the overall quality(top) and running time of the state-of-the-art non-deterministic and deterministic hypergraph partitioners for $k = 8$.

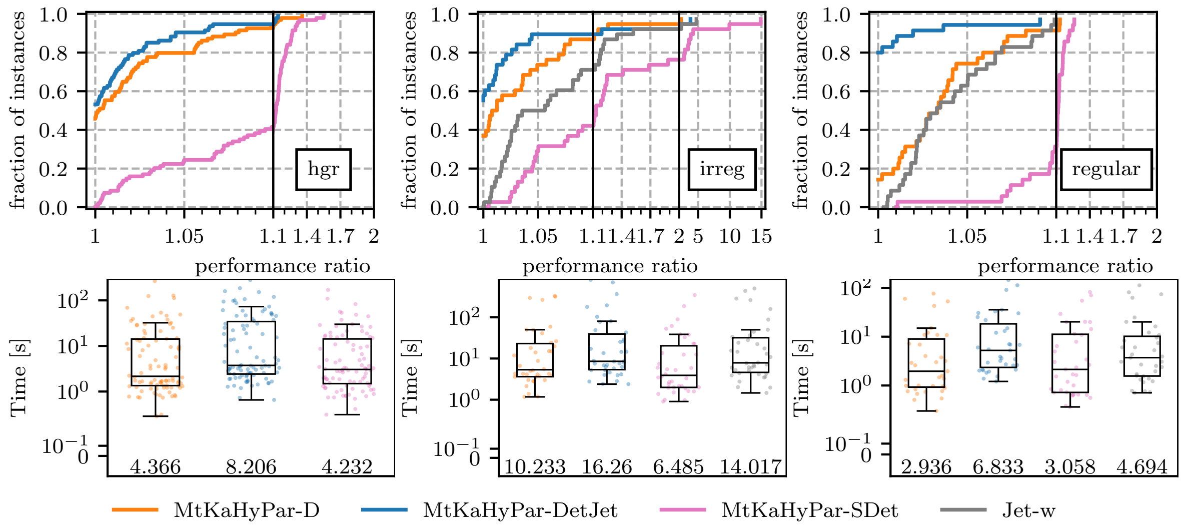

Figure A.3: Comparing the overall quality(top) and running time of the state-of-the-art non-deterministic and deterministic hypergraph partitioners for $k = 11$.

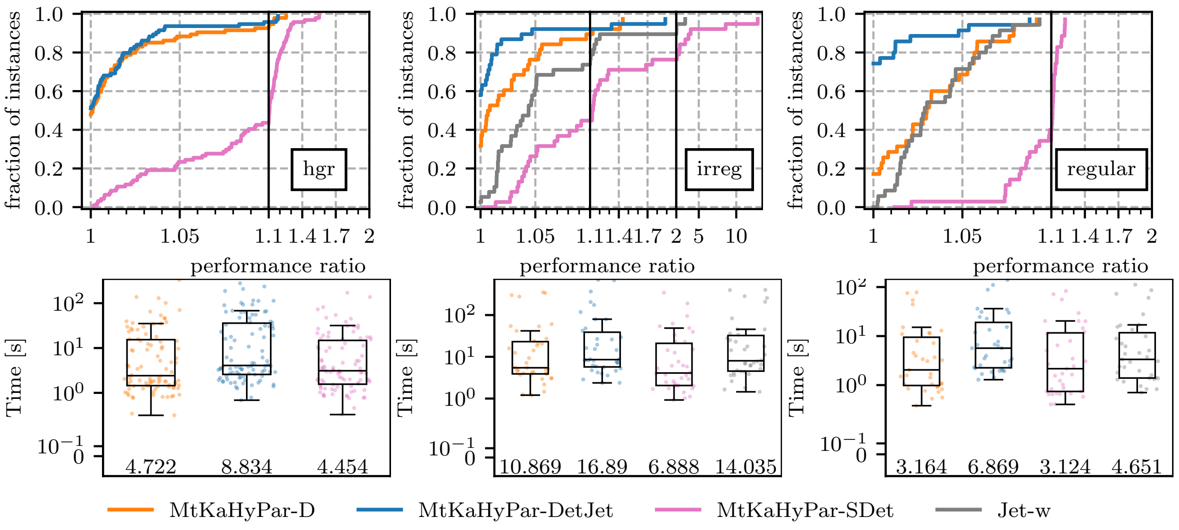

Figure A.4: Comparing the overall quality(top) and running time of the state-of-the-art non-deterministic and deterministic hypergraph partitioners for $k = 16$.

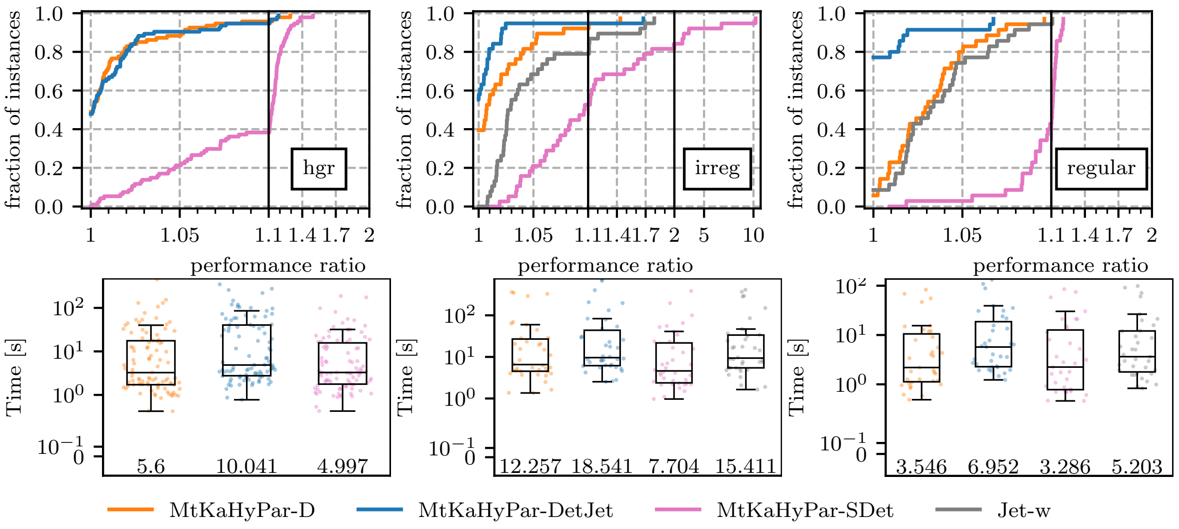

Figure A.5: Comparing the overall quality(top) and running time of the state-of-the-art non-deterministic and deterministic hypergraph partitioners for $k = 27$.

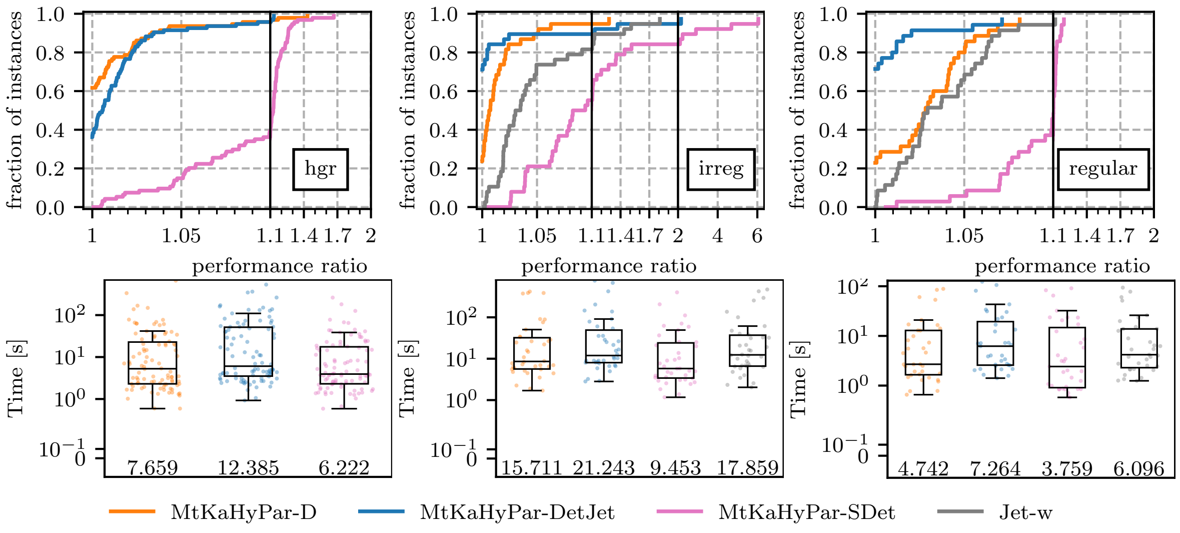

Figure A.6: Comparing the overall quality(top) and running time of the state-of-the-art non-deterministic and deterministic hypergraph partitioners for $k = 64$.

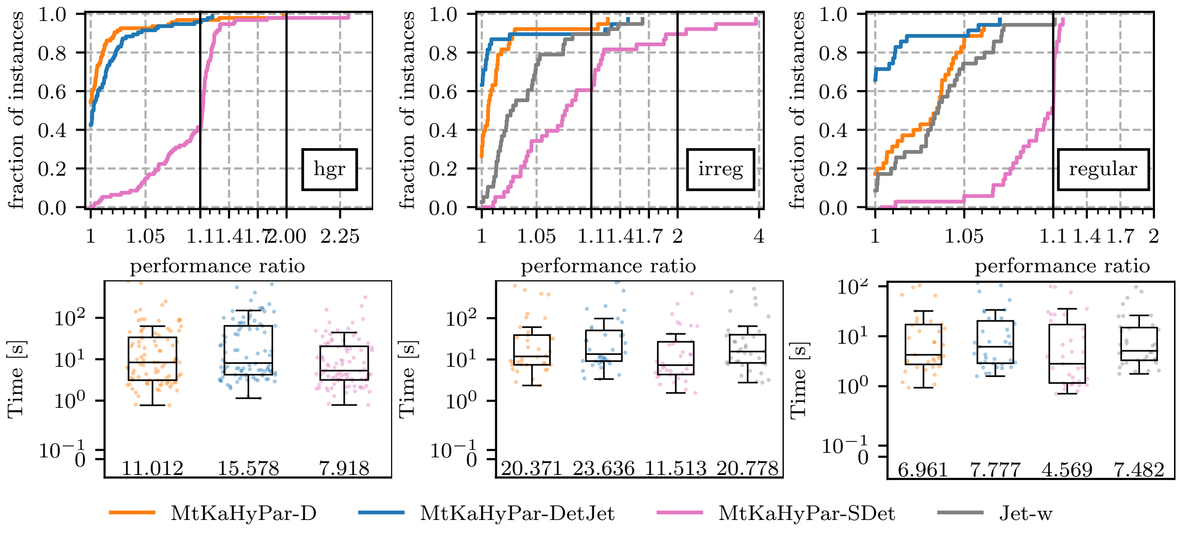

Figure A.7: Comparing the overall quality(top) and running time of the state-of-the-art non-deterministic and deterministic hypergraph partitioners for $k = 128$.

B Direct Comparison Between Mt-KaHyPar-DetJet and Other Configurations

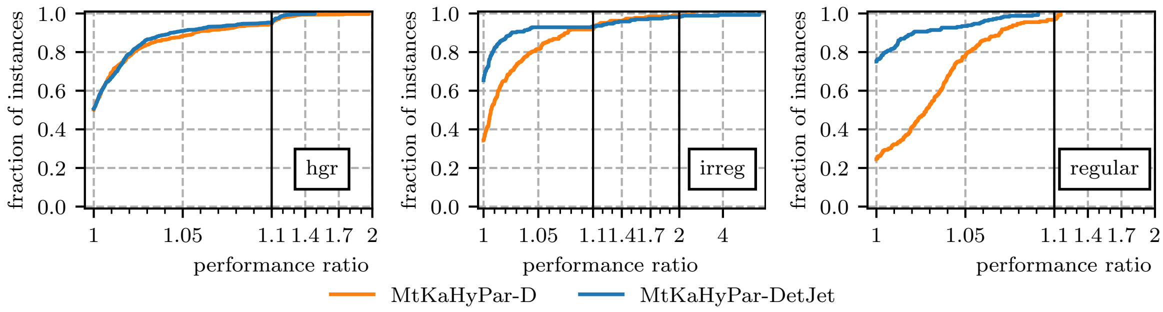

Figure B.1: Directly comparing the solution quality of our algorithm `Mt-KaHyPar-DetJet` to its non-deterministic counterpart `Mt-KaHyPar-D`.

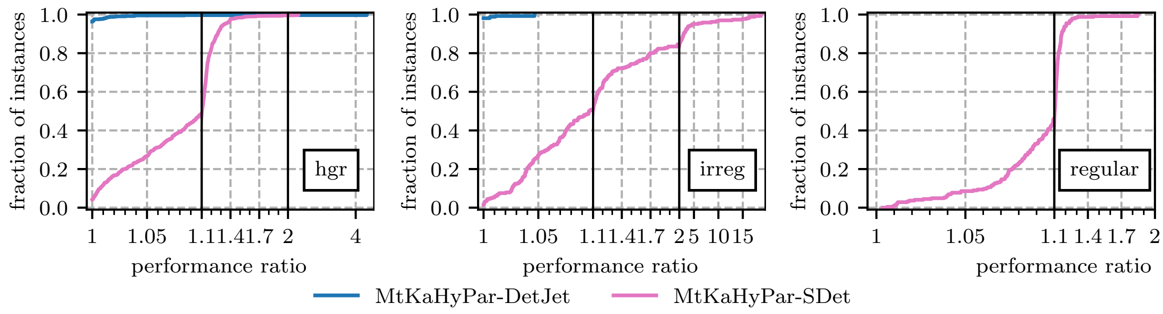

Figure B.2: Directly comparing the solution quality of our algorithm `Mt-KaHyPar-DetJet` to the previous deterministic configuration `Mt-KaHyPar-SDet`.

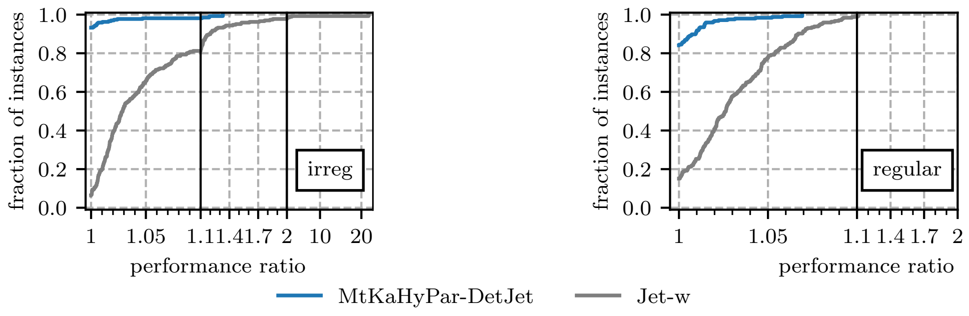

Figure B.3: Directly comparing the solution quality of our algorithm `Mt-KaHyPar-DetJet` to a non-deterministic implementation of the Jet algorithm (using weights in rebalancing)

C Comparing Flow-Based and non-Flow-Based Configurations of Mt-KaHyPar

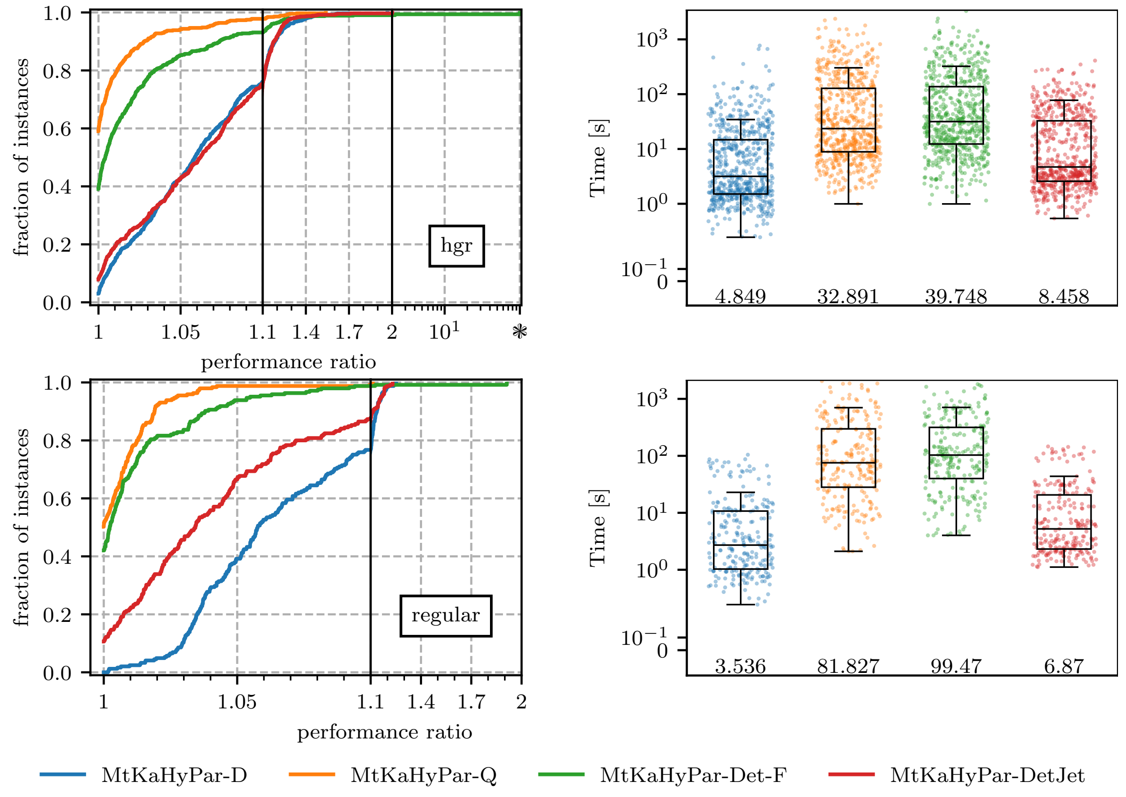

Figure C.1: Comparing the overall quality(left) and running time (right) of our configuration using flow-based refinement to `Mt-KaHyPar-Q`.

Bibliography

[1] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-quality shared-memory graph partitioning. *IEEE Transactions on Parallel and Distributed Systems*, PP, 10 2017.

[2] Pablo Andrés-Martínez and Chris Heunen. Automated distribution of quantum circuits via hypergraph partitioning. *Physical Review A*, 100(3), September 2019.

[3] Cevdet Aykanat, B. Barla Cambazoglu, and Bora Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008.

[4] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In Yossi Azar and Thomas Erlebach, editors, *Algorithms – ESA 2006*, pages 684–695, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[6] Ismail Bustany, Grigor Gasparyan, Andrew B. Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. An open-source constraints-driven general partitioning multi-tool for vlsi physical design. *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[7] U.V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[8] Min-Chi Chiang, Lu Zhang, Yu-Min Chou, and Jerry Chi-Yuan Chou. Dynamic resource management for machine learning pipeline workloads. *SN Computer Science*, 4:1–19, 2023.

[9] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1–2):48–57, sep 2010.

[10] Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. Hypergraph sparsification and its application to partitioning. In *2013 42nd International Conference on Parallel Processing*, pages 200–209, 2013.

[11] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10 pp.–, 2006.

[12] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, January 2002.

[13] V. Duraira and P. Kalla. Exploiting hypergraph partitioning for efficient boolean satisfiability. In *Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No.04EX940)*, pages 141–146, 2004.

[14] V. Durairaj and P. Kalla. Guiding CNF-SAT search via efficient constraint partitioning. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 498–501, San Jose, CA, USA, 2004. IEEE.

[15] Andreas Emil Feldmann. Fast balanced partitioning is hard even on grids and trees. *Theoretical Computer Science*, 485:61–68, 2013.

[16] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181, 1982.

[17] Michael S. Gilbert, Kamesh Madduri, Erik G. Boman, and Sivasankaran Rajamanickam. Jet: Multilevel Graph Partitioning on Graphics Processing Units. 2023. Publisher: [object Object] Version Number: 2.

[18] Lars Gottesbüren. *Parallel and Flow-Based High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2023.

[19] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *ACM Trans. Algorithms*, 20(1), jan 2024.

[20] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel Flow-Based Hypergraph Partitioning. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:21, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[21] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel flow-based hypergraph partitioning. *CoRR*, abs/2201.01556, 2022.

[22] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep Multilevel Graph Partitioning. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[23] Lars Gottesbüren and Michael Hamann. Deterministic Parallel Hypergraph Partitioning. 2021. Publisher: [object Object] Version Number: 1.

[24] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable High-Quality Hypergraph Partitioning. 2023. Publisher: [object Object] Version Number: 1.

[25] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. *Scalable Shared-Memory Hypergraph Partitioning*, pages 16–30.

[26] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, March 2021.

[27] Florian Grötschla. Parallel flowcutter refinement for hypergraph partitioning. Master's thesis, Karlsruher Institut für Technologie (KIT), 2020.

[28] Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM J. Exp. Algorithmics*, 23, feb 2018.

[29] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. Distributed graph clustering using modularity and map equation. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27 - 31, 2018, Proceedings*, page 688–702, Berlin, Heidelberg, 2018. Springer-Verlag.

[30] Tobias Heuer. *Scalable High-Quality Graph and Hypergraph Partitioning*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2022.

[31] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network flow-based refinement for multilevel hypergraph partitioning. *ACM J. Exp. Algorithmics*, 24, sep 2019.

[32] Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISSN: 1868-8969.

[33] Artur Jeż. Faster fully compressed pattern matching by recompression, 2013.

[34] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, and Alon Shalita. Social hash partitioner: a scalable distributed hypergraph partitioner. *Proc. VLDB Endow.*, 10(11):1418–1429, aug 2017.

[35] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, March 1999.

[36] Robert Krause. Community detection in hypergraphs with application to partitioning, 2021.

[37] Dominique Lasalle and George Karypis. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 225–236, 2013.

[38] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Vieweg+Teubner Verlag, Wiesbaden, 1990.

[39] Nikolai Maas, Lars Gottesbüren, and Daniel Seemaier. *Parallel Unconstrained Local Search for Partitioning Irregular Graphs*, pages 32–45.

[40] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. Bipart: a parallel and deterministic hypergraph partitioner. In *Proceedings of the 26th ACM SIG-*

PLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '21, page 161–174, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2625–2638, 2017.

[42] Chuck Pheatt. Intel® Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298, April 2008. Place: Evansville, IN, USA Publisher: Consortium for Computing Sciences in Colleges.

[43] Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. *ACM J. Exp. Algorithmics*, 19, jan 2015.

[44] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms – ESA 2011*, pages 469–480, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[45] Peter Sanders and Daniel 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 '24, page 443–445, New York, NY, USA, 2024. Association for Computing Machinery.

[46] Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD Thesis, Karlsruher Institut für Technologie (KIT), 2020.

[47] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way Hypergraph Partitioning via n-Level Recursive Bisection. *arXiv:1511.03137 [cs]*, November 2015. arXiv: 1511.03137.

[48] Honghua Hannah Yang and D. F. Wong. *Efficient Network Flow Based Min-Cut Balanced Partitioning*, pages 521–534. Springer US, Boston, MA, 2003.