Karlsruhe Institute of Technology

Asynchronous n-Level Hypergraph Partitioning

Master Thesis of

Moritz Laupichler

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers: Prof. Dr. Peter Sanders
PD Dr. Torsten Ueckerdt
Advisors: M.Sc. Lars Gottesbüren
M.Sc. Tobias Heuer

Time Period: April 19th, 2021 – November 17th, 2021

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, November 17, 2021

Abstract

In this thesis, we introduce a novel approach to scalable high-quality parallel hypergraph partitioning. The balanced k-way hypergraph partitioning problem consists of partitioning the vertices of a hypergraph into k blocks of almost equal size s.t. an objective function is optimized (usually the sum of the number of blocks that each hyperedge spans should be minimized). In the n-level variant of the popular multilevel scheme, $\approx n$ vertex contractions are performed on the hypergraph before finding an initial partitioning and then reverting the contractions. In the *uncoarsening* phase, after every uncontraction, localized refinement is applied around the region of the uncontraction, which leads to partitions of very high quality. In this work, we present *asynchronous uncoarsening*, a highly scalable parallelization of the n-level uncoarsening phase in which uncontractions and localized refinement happen concurrently. We introduce a framework that requires no global synchronization during the uncoarsening phase and allows fine-grained control of cross-dependencies between concurrent uncontractions and refinement. We implement asynchronous uncoarsening in the Mt-KaHyPar framework and experimentally compare our approach with Mt-KaHyPar's previous batch synchronous uncoarsening phase. Our extensive experiments on more than 500 real-world hypergraphs with up to 190 million vertices and 2 billion pins (sum of hyperedge sizes) show that our approach scales better on large hypergraphs with only a small loss of quality. We find that Mt-KaHyPar with asynchronous uncoarsening is on average 26% faster than Mt-KaHyPar with batch uncoarsening for large instances using 64 threads. Our uncoarsening phase reaches average self-relative speedups of 39 for 64 threads on difficult instances for which the average speedups of Mt-KaHyPar's batch uncoarsening are limited to less than 24.

Deutsche Zusammenfassung

Wir stellen eine neue Technik für skalierbare, parallele Hypergraph-Partitionierung für Partitionen hoher Qualität vor. Das Problem balancierter k-Wege Hypergraph-Partitionierung besteht aus der Verteilung der Knoten eines Hypergraphen auf k disjunkte Blöcke etwa gleicher Größe, so dass eine Zielfunktion optimiert wird (üblicherweise soll die Summe der Anzahl Blöcke, die jede Kante berührt, minimiert werden). In der n-Level Variante des Multilevel-Schemas werden $\approx n$ Knotenkontraktionen ausgeführt bevor eine initiale Partition gefunden wird und die Kontraktionen rückgängig gemacht werden. In der *Uncoarsening*-Phase, wird dann lokalisiertes Refinement um jede Entkontraktion angewendet, was zu Partitionen sehr hoher Qualität führt. In dieser Arbeit führen wir *Asynchronous Uncoarsening* ein, eine skalierbare Parallelisierung der n-Level Uncoarsening-Phase, in der Entkontraktionen und Refinement gleichzeitig ausgeführt werden. Wir stellen ein Framework für Asynchronous Uncoarsening vor, das keine Synchronisierung benötigt, und eine kleinschrittige Kontrolle von Abhängigkeiten zwischen Entkontraktionen und Refinement ermöglicht. Wir implementieren unseren Algorithmus im Mt-KaHyPar Framework und vergleichen unsere Uncoarsening-Phase mit der vorherigen n-Level Uncoarsening Phase von Mt-KaHyPar, welche auf Batches von Entkontraktionen basiert. Unsere Experimente an mehr als 500 Hypergraphen aus echten Anwendungen zeigen, dass Asynchronous Uncoarsening mit nur geringem Qualitätsverlust die Skalierbarkeit paralleler Hypergraph-Partitionierung für große Hypergraphen erhöhen kann. Mit 64 Threads ist Mt-KaHyPar mit Asynchronous Uncoarsening auf großen Instanzen durchschnittlich 26% schneller als Mt-KaHyPar mit Batches. Asynchronous Uncoarsening erreicht mit 64 Threads durchschnittliche Speedups von 39 auf großen Instanzen, für die Mt-KaHyPar's Batch Uncoarsening durchschnittliche Speedups von weniger als 24 erreicht.

Contents

1. Introduction **1**
1.1. Problem Statement . . . 2
1.2. Contribution . . . 3
1.3. Outline . . . 3

2. Preliminaries **4**

3. Related Work **6**
3.1. The Multilevel Paradigm . . . 6
3.2. k-way Partition Refinement . . . 7
3.2.1. Label Propagation Refinement . . . 7
3.2.2. Fiduccia-Mattheyses Refinement . . . 8
3.3. Sequential Hypergraph Partitioners . . . 9
3.4. Parallel Hypergraph Partitioners . . . 10
3.4.1. Mt-KaHyPar . . . 10
3.4.2. Other Parallel Partitioners . . . 12

4. A Framework for Asynchronous n-Level Hypergraph Partitioning **13**
4.1. Order of Uncontractions . . . 14
4.2. Asynchronous Uncoarsening Scheme . . . 15
4.3. Asynchronous Localized Refinement . . . 17
4.4. Remaining Synchronization Points . . . 18

5. Gain Cache for Asynchronous Uncoarsening **19**
5.1. Uncontraction Gain Cache Update . . . 20
5.2. Node Move Gain Cache Update . . . 21
5.3. Implementation Details . . . 21
5.4. Evaluation of the Gain Cache . . . 22

6. Cross-Dependencies in Asynchronous Refinement **23**
6.1. Intermediate Inconsistent States Caused By Uncontractions . . . 23
6.2. Changes to Connectivity Information Affecting Gain Cache Updates . . . 24
6.3. Changes to Pin Lists Affecting Gain Cache Updates . . . 25
6.4. Effectiveness of Localized FM Refinement in Asynchronous Uncoarsening . 27
6.5. Differences to Mt-KaHyPar . . . 30

7. Experiments **32**
7.1. Instances . . . 32
7.2. Algorithms . . . 33
7.3. System and Methodology . . . 33
7.4. Comparison with Other Algorithms . . . 34
7.5. Scalability . . . 36
7.6. Detailed Discussion . . . 41

8. Conclusion and Future Work **44**

Bibliography **45**

Appendix **49**

A. Parameter Tuning Experiments 49

A.1. Minimum Edge Size for Snapshots (cp_{snap}). 49

A.2. Maximum Region Similarity (cp_{sim}). 50

A.3. Number of Group Picking Retries ($cp_{retries}$). 51

A.4. Minimum Number of Seeds for Loc. Refinement (cp_{seeds}). 52

B. Additional Plots 53

1. Introduction

Graphs are a well-known abstraction used to model objects (vertices) and binary relations between objects (edges). *Hypergraphs* generalize graphs by allowing *hyperedges* or *nets* that connect any number of vertices. A fundamental problem and an important pre-processing step for many high-perfomance computing applications is *hypergraph partitioning (HGP)*. The problem consists of partitioning the vertex set into k disjoint blocks s.t. the number of cut hyperedges is minimized while all blocks have the same size (with a given tolerance). Often, each cut hyperedge is additionally weighted by the number of different blocks the hyperedge connects.
Hypergraphs express object groupings effectively in many areas of computer science and have become increasingly important with the recent big data revolution. High-quality partitions of hypergraphs are required for applications in domains such as VLSI design [4], scientific computing [8], tensor network contraction for quantum circuit simulation [19], hypergraph processing frameworks for network analysis [23] and storage sharding in distributed database management systems [10, 27, 43, 49].
Thus, HGP is an important pre-processing step for many parallel computing applications so it is crucial to parallelize HGP algorithms efficiently.

The HGP problem is known to be NP-hard [30] and hard to approximate [6]. Therefore, heuristic algorithms are used in practice. In particular, *multilevel* algorithms have received a lot of attention in this field in the last decades [1, 8, 45, 25, 26] as they have proven to find good trade-offs between running time and solution quality. Multilevel partitioning is separated into three phases: In the *coarsening* phase, pairs or clusters of vertices are *contracted* to obtain a sequence of hypergraphs with decreasing size and similar structural properties. Once the hypergraph is small enough, it is initially partitioned into k blocks (*initial partitioning*). Finally, in the *refinement* or *uncoarsening* phase, the contractions of the coarsening phase are undone and the current partition is projected onto successively finer hypergraphs. After each uncontraction, local search heuristics are used to improve the quality of the partition.

The number of levels used in the multilevel hierarchy offers a trade-off between running time and solution quality. Traditional multilevel partitioners contract matchings or clusters during the coarsening phase [8, 26, 45], which usually leads to a small number of levels ($\approx \mathcal{O}(\log n)$). The partitioner KaHyPar [21, 16, 40] implements the multilevel scheme in its most extreme version, the *n-level* variant [34]. In n-level partitioning the coarsening phase contracts only a single pair of vertices in each step leading to a hierarchy with almost n levels. That way, local search heuristics are applied after every single uncontraction, i.e.

after every minimal change to the underlying hypergraph. This constitutes a very fine-grained search for improvements which has been shown to currently outperform any other hypergraph partitioners regarding solution quality [16]. The key ingredients for an efficient implementation of the n-level scheme are highly-localized refinement algorithms that only expand around a small region of the uncontracted nodes and an adaptive stopping rule [34] that terminates local searches early if they are unlikely to find further improvements.

Sequential high-quality partitioners are too slow to reasonably partition large hypergraphs. Therefore, various parallel approaches have been proposed. Until recently, however, the work on parallel HGP algorithms had been focusing on distributed-memory systems [44, 12, 24]. The distributed-memory approach makes it difficult to parallelize some of the features of modern partitioners required for partitions of high quality. Therefore, distributed-memory parallel partitioners find partitions of inferior quality compared to sequential partitioners [17, 40, 18].
Mt-KaHyPar [17, 18] is the first shared-memory partitioner that finds partitions that compete with the best sequential partitioners regarding quality while being an order of magnitude faster. The partitioner manages to parallelize core features of high-quality hypergraph partitioners. Mt-KaHyPar offers both a multilevel and an n-level configuration for varying trade-offs between quality and running time. The n-level variant is the slower configuration but concerning quality it achieves solutions that are on par with those found by a comparable configuration of the currently best sequential partitioner KaHyPar [17]. This makes n-level Mt-KaHyPar particularly interesting as it is fast itself and provides high-quality partitions that factor into the efficiency of the subsequent application that uses the partition in its computational model.
However, we believe that the scalability of the uncoarsening phase of n-level Mt-KaHyPar can be improved further. In the uncoarsening phase, the partitioner constructs batches of uncontractions and only works in parallel within each batch. In particular, it performs all uncontractions in a batch in parallel and then applies localized refinement based on those uncontractions in parallel. With a constant number b of uncontractions per batch, this leads to $\mathcal{O}(n/b)$ global synchronization points between uncontractions and refinement that inhibit the parallelism of the uncoarsening phase for large numbers of threads.

1.1. Problem Statement

In this thesis, we aim to increase the scalability of parallel n-level hypergraph partitioning by removing global synchronization points using *asynchronous uncoarsening* in which uncontractions and localized refinement happen concurrently. It is to be analyzed whether n-level partitioning using asynchronous uncoarsening is capable of partitioning a hypergraph faster and more scalably than using a batch synchronous approach to uncoarsening as applied by n-level Mt-KaHyPar. Furthermore, the effect of asynchronous uncoarsening on the solution quality needs to be evaluated.
As all existing hypergraph partitioners separate uncontractions and refinement, several implications of the asynchronous paradigm have to be explored: A framework for asynchronous uncoarsening with minimal global synchronization needs to be established. Moreover, concurrently accessed data structures may need to be adapted and potentially additional synchronization measures may need to be introduced. Also, the efficacy of localized refinement in the presence of concurrent uncontractions must be evaluated.
Asynchronous uncoarsening should be implemented in the shared-memory parallel HGP framework Mt-KaHyPar. The goal of the implementation is a parallel code with better scalability and equivalent solution quality in comparison to the existing n-level Mt-KaHyPar variant. The resulting partitioner should be experimentally compared with existing variants of Mt-KaHyPar and other state-of-the-art parallel and sequential partitioners regarding running times and partition quality.

1.2. Contribution

We present the first n-level hypergraph partitioning algorithm using asynchronous uncoarsening which encompasses concurrent uncontractions and localized refinement. We introduce a framework for asynchronous uncoarsening that reverts a forest of contractions generated in an n-level coarsening phase without requiring any global synchronization. We adapt the gain cache of Mt-KaHyPar for asynchronous uncoarsening to provide correct move gains for refinement heuristics in the presence of concurrent gain cache updates caused by uncontractions and node moves. Additionally, we describe how lock contention for costly gain cache updates can be reduced using snapshots of optimized size. Furthermore, we discuss the negative effects of asynchronous uncoarsening on the effectiveness of localized refinement. To that end, we propose a mechanism to reduce interference with localized refinement by explicitly diversifying the active regions of the worker threads.
We implement our asynchronous uncoarsening phase in the Mt-KaHyPar partitioning framework and demonstrate the viability of our approach in rigorous experiments on more than 500 real world hypergraphs. We find that asynchronous uncoarsening suitably increases the scalability of n-level partitioning for long running instances compared to batch synchronous Mt-KaHyPar. For the longest running instances, we observe an average self-relative speedup of 39.72 using 64 threads for our asynchronous uncoarsening phase. In comparison, the uncoarsening phase of batch synchronous Mt-KaHyPar reaches an average speedup of 23.69 with 64 threads on the same set of instances. We report a small loss of quality compared to batch synchronous uncoarsening and decreasing quality with larger numbers of threads caused by additional interference.

1.3. Outline

The structure of this thesis is as follows: In Chapter 2 we introduce necessary notation. In Chapter 3, we give a more detailed overview of related work with a focus on Mt-KaHyPar which serves as the basis of our approach. Chapter 4 describes our framework for asynchronous uncoarsening. Chapter 5 deals with the intricacies of a gain cache in the asynchronous context while Chapter 6 discusses cross-dependencies and interference patterns that arise with this new paradigm. In Chapter 7 we present our experimental results that confirm the superior scalability of asynchronous uncoarsening at the cost of a small loss in quality. Chapter 8 summarizes our results and presents ideas for future research into asynchronous uncoarsening.

2. Preliminaries

In the following, we introduce some definitions and notation necessary for n-level hypergraph partitioning. These definitions are in large part adopted from Gottesbüren et al. [17] while some definitions are tweaked for the description of our partitioner in later chapters.

Hypergraph. A *weighted hypergraph* $H = (V, E, c, \omega)$ is defined as a set of *vertices/(hyper-)nodes* V with associated *vertex weights* $c : V \to \mathbb{R}_{\geq 0}$ and a set of *hyperedges/nets* E with associated *net weights* $\omega : E \to \mathbb{R}_{\geq 0}$. A net $e \in E$ is a subset of V, where the vertices $v \in e$ are called the *pins* of e. We define the weight of sets of vertices/nets via the sum of weights of their elements, i.e. $c(U) = \sum_{u \in U} c(u)$ and $\omega(F) = \sum_{e \in F} \omega(e)$ for $U \subseteq V, F \subseteq E$. If $v \in e$, we say v and e are *incident* to each other. $I(v)$ denotes the set of all nets incident to a vertex v. Thus, the *degree* of $v \in V$ is $d(v) = |I(v)|$. The *size* of a net e is the number of its pins $|e|$. We call two nets $e, e' \in E$ parallel or identical if they constitute the same set of pins. Two pins $u, v \in e$ for any $e \in E$ are called *neighbors*.

Contraction, Uncontraction. A *contraction* (u, v) consists of contracting a node v *onto* a node u. The *contracted vertex* v is removed from all nets $e \in I(v) \cap I(u)$ and replaced by the *representative* u in all nets $e \in I(v) \setminus I(u)$. After the contraction, u, thus, represents a merged node which also assumes the merged weight $c(u) := c(u) + c(v)$. An *uncontraction* (u, v) is the reverse operation of the contraction (u, v). In the context of reverting contractions, the terms contraction and uncontraction may be used interchangeably when meant to address the partaking vertices u and v. In this case, v may also be called the *uncontracted vertex*.

Contraction Forest. An undirected graph is called a *forest* if it does not contain any cycles and a forest is called a *tree* if it is connected. A rooted forest is a directed graph for which the underlying undirected graph is a forest with a set of root nodes s.t. all maximal directed paths end in a root node. A set C of contractions is *compatible* with a hypergraph $H = (V, E)$ if the directed graph $\mathcal{F} = (V, \{(v, u) \mid (u, v) \in C\})$ with edges pointing from contracted vertex to representative is a rooted forest. $\mathcal{F}$ is called the *contraction forest*. In this work, we only discuss methods that find contraction forests guaranteed to be compatible with the given hypergraph. We describe the contraction forest using the array `rep` where $\texttt{rep}[v] = u$ for a node v contracted onto u and $\texttt{rep}[v] = v$ for a node v that is not contracted. Thus, $\{v \in V \mid \texttt{rep}[v] = v\}$ denotes the roots of $\mathcal{F}$. The *ancestors* of v are the vertices on the unique path from v to the root of the tree. A vertex $w \neq v$ with $\texttt{rep}[w] = v$ is called a *child* of v and all vertices in the subtree rooted at v are called *descendants* of v. Vertices w_1, w_2 are *siblings* if $w_1 \neq \texttt{rep}[w_1] = \texttt{rep}[w_2] \neq w_2$.

Partition. Let $[k] := \{1, \dots, k\}$. A *k-way partition* of a hypergraph $H = (V, E, c, \omega)$ is a function $\Pi : V \to [k]$. The *blocks* $V_i := \Pi^{-1}(i)$ of Π are the inverse images, i.e. sets of vertices assigned the same value $i \in [k]$. We call Π ϵ*-balanced* if each block V_i satisfies the *balance constraint* $c(V_i) \leq L_{\max} := (1 + \epsilon)\frac{c(V)}{k}$ for some parameter $\epsilon \in (0, 1)$. A 2-way partition is also called a *bipartition*.
For each net $e \in E$ the set $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e)$ of a net e is the cardinality of its connectivity set, i.e. $\lambda(e) = |\Lambda(e)|$. A net is called a *cut net* if $\lambda(e) > 1$ or an *internal net* otherwise. A vertex that is incident to a cut net is called a *boundary* or *border* vertex. The number of pins of a net e in block V_i is denoted by $\Phi(e, i) := |V_i \cap e|$.

Connectivity Metric. To measure the quality of a partition we use the *connectivity metric* $(\lambda - 1)(\Pi) := \sum_{e \in E}(\lambda(e) - 1)\omega(e)$. Given a k-way partition Π, moving u from its block V_s with $s := \Pi(u)$ to V_t improves the connectivity metric by the move gain:

$$\begin{aligned} g_{s \to t}(u) &:= b(u, s) - p(u, t) \\ \text{with } b(u, s) &:= \omega(\{e \in I(u) \mid \Phi(e, s) = 1\}), \\ \text{and } p(u, t) &:= \omega(\{e \in I(u) \mid \Phi(e, t) = 0\}) \end{aligned}$$

We call the term $b(u, s)$ the *(move-from) benefit* for moving u out of block V_s and the term $p(u, t)$ the *(move-to) penalty* for moving u to block V_t. We sometimes denote a move m as a triple $m = (u, s, t)$ with the moved vertex u, the origin block V_s and the target block V_t. Then $g(m) := g_{s \to t}(u)$, $b(m) := b(u, s)$ and $p(m) := p(u, t)$ mean the associated gain, benefit and penalty respectively. Note that only moves of boundary vertices can ever have a positive gain.

3. Related Work

In this section, we give an overview of related work in the field of hypergraph partitioning. First, we summarize the multilevel paradigm and established techniques for hypergraph partition refinement that are relevant to our work. Then, we look at related sequential and parallel hypergraph partitioning algorithms with emphasis on the Mt-KaHyPar shared-memory parallel partitioning framework.

3.1. The Multilevel Paradigm

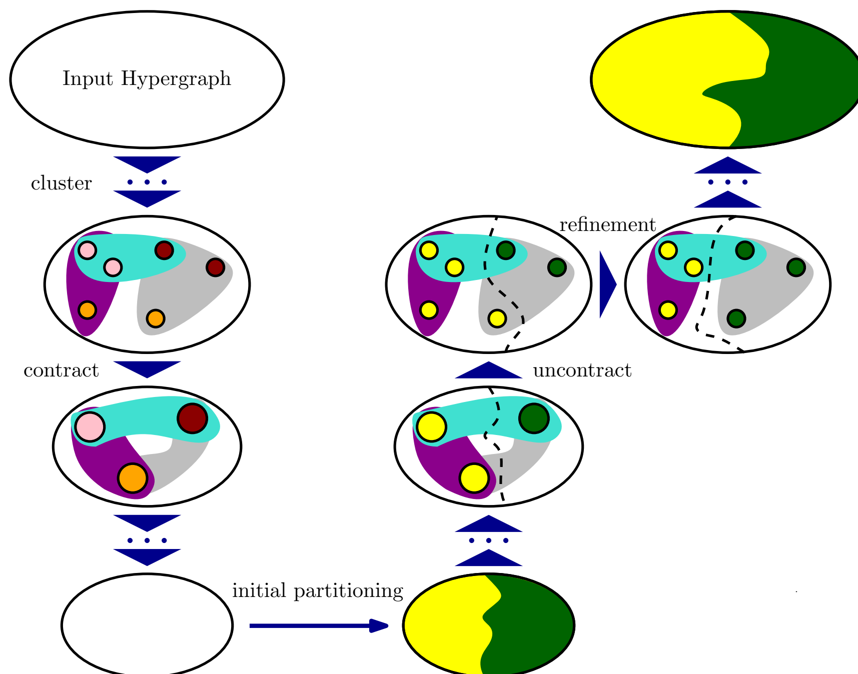

Figure 3.1.: An overview of the multilevel partitioning scheme with the three phases *coarsening* (left), *initial partitioning* (bottom) and *uncoarsening* (right).

As hypergraph partitioning is an NP-complete optimization problem [30], many different heuristics for the problem have been found. Each heuristic offers a different trade-off between partition quality and partitioning time. The most interest has been put into multilevel partitioning algorithms, which consist of three phases that we depict in Figure 3.1:

First, in the *coarsening* phase, nodes are contracted in stages, where, typically, in each stage matchings or clusterings of nodes are each contracted onto one representative *coarse* node. Coarsening terminates when a sufficiently small coarsened hypergraph is reached, usually after $\approx \mathcal{O}(\log n)$ levels of contractions. Second, in the *initial partitioning* phase, an initial partition of the coarsened hypergraph is found, usually using a portfolio of initial partitioners. Third, in the *uncoarsening* (also known as *refinement*) phase, the contractions from the coarsening phase are undone in reverse order with intermediate refinement of the partition at each stage. In *recursive bipartitioning*, the multilevel scheme is applied recursively for $k = 2$ to eventually generate a partition with any number of blocks using only 2-way refinement. In *direct k-way partitioning*, an initial partition with k parts is found that is then refined using k-way refinement algorithms. Traditional multilevel approaches use matchings or clusterings to find suitable contractions resulting in $\mathcal{O}(\log n)$ levels. The n-level partitioning scheme [34, 41, 20] aims to contract single nodes individually onto a representative which leads to $\mathcal{O}(n)$ levels. Thus, during n-level uncoarsening, the partition is refined after every uncontraction, leading to partitions of very high quality.

3.2. k-way Partition Refinement

In multilevel partitioning, refinement is the main factor deciding the solution quality and often comprises a large part of the total running time. Thus, the choice of which refinement heuristics to apply during the uncoarsening phase and their optimization are critical. In the following, we discuss two essential heuristics for the refinement of hypergraph partitions that are often used during the uncoarsening phase of multilevel hypergraph partitioners.

3.2.1. Label Propagation Refinement

Algorithm Outline. In label propagation (LP) [26], the hypernodes of the current hypergraph are traversed in random order. For each node, the gain regarding the partition objective function of any possible move for the node is calculated. If a move with positive gain that does not break the balance constraint is possible, the heuristic acts greedily, performing such a move with the highest possible gain. Label propagation may also be used to rebalance the partition. For that purpose, LP prefers moves that improve the partition balance the most while not worsening the solution quality. As only moves with positive gain are performed, label propagation is not capable of escaping local minima. Therefore, LP refinement may get stuck early on. Thus, label propagation alone cannot be relied upon to find high quality partitions.

Localized LP. In n-level hypergraph partitioning, the hypergraph and its partition change only minimally between two refinement runs caused by a single uncontraction. Therefore, variants of refinement heuristics that are localized around an uncontraction are of particular interest for n-level partitioners. Localized LP variants for n-level partitioning [20] restrict their search to boundary nodes considered active for the search. After an uncontraction, initially, a maximum of two nodes (the uncontracted node and the representative) are active. When the localized LP heuristic moves a node, it activates all neighbors of the moved node. A localized LP search expands in that way until no more eligible moves with positive gain are found in one pass of the active nodes.

Parallelization. Label propagation has a simple parallelization by iterating through the nodes in random order in parallel. This approach also works for parallel localized LP. With that parallelization, threads only interfere with each other because the gains of possible moves change during their examination due to concurrent moves.

Attributed Gains. A possible safeguard against this interference are attributed gains [18]. Attributed gains work by recording the actual changes to the partition caused by a move.

Those changes determine the effect of the move on the partition quality. Therefore, the actual changes serve to find out the actual gain of a move as applied to the partition. Assume that a thread in parallel LP performs a move $m = (u, s, t)$ based on an expected gain of $g(m)$. The thread will perform a synchronized write to the partition when executing the move. This allows us to record if and how the connectivity of hyperedges incident to u changes at the synchronized point in time when m is performed. If the gain increases or decreases the connectivity of an incident hyperedge, we attribute the according gain to m. By summing up the attributed gains for all incident hyperedges, we observe the actual gain of m. Then, if the actual gain is worse than the expected $g(m)$ or m even worsens the solution quality, we can decide to revert the move. Thus, attributed gains work to double-check the merit of a move in parallel LP. Attributed gains are also applicable in other parallel refinement heuristics.

3.2.2. Fiduccia-Mattheyses Refinement

Algorithm Outline. The other important refinement heuristic is the Fiduccia-Mattheyses (FM) heuristic [14]. Much like label propagation, it also works by individually moving nodes between partitions. However, in order to decide which move to perform next, it does not only compare between the possible moves of the same node but between the possible moves of all nodes. To achieve this, the algorithm first examines all possible moves for each node. Then, among all examined moves, the move with the highest gain that does not break the balancing constraint is performed. When a node u is moved, the partition changes and the gains for moving any neighbor of u may change. Consequently, those gains have to be recalculated before the next best move can be chosen. Every node is moved only once and the FM heuristic stops when every node has been moved once. After a complete sequence of moves has been constructed, the prefix of that sequence with the best total gain is found. The prefix is kept while the rest of the moves are reverted. Note that moves with negative gain are explicitly allowed. That way, the best prefix represents a move sequence not restricted by local minima. Moves with negative gain in the prefix may allow the heuristic to climb “hills” of the objective function to find better minima.

Implementation Techniques. The calculation and recalculation of move gains for all possible moves is slow. Therefore, many techniques for efficient FM local search have been established. An important measure is the use of priority queues (PQs) for the next move to perform, ordering moves by their expected gain. The moves in the PQs are reordered if their gains change. Alternatively, the heuristic can lazily verify that a move taken from the PQ has at least positive gain without reordering moves whose expected gain changed. Partitioners may use one PQ per move direction, i.e. for each pair of blocks [37], one PQ per block storing moves *from* the block [28], one PQ per block storing moves *to* the block [34, 40, 1] or simply one global PQ of moves [39]. Gain (re-)calculation can be sped up further by using a gain cache [41, 1] and delta-gain-updates [35] that may exclude nets according to past moves [1].

Localized FM. Different variations of a localized variant of the FM heuristic have been explored for n-level partitioning [34, 41, 1]. As with localized LP, localized FM variants only examine and move active nodes. After a move, neighbors of the moved node are activated and inserted into the PQs of the FM search. However, as FM is allowed to perform negative gain moves, a localized FM search cannot simply run until no more eligible moves are available like localized LP. Instead, stochastic stopping criteria are used to limit the search to the region around the uncontraction. The stopping rules prevent the search from expanding to nodes that are not likely to offer any further improvements [34, 1].

Parallelization. Parallelizing the FM heuristic is non-trivial for several reasons. Firstly, a parallelized FM suffers from the same interferences as parallel label propagation, i.e.

concurrent moves affecting each others' gains. Secondly, an inherently serial global move sequence has to be constructed by threads working in parallel. It cannot be relied upon that moves are committed to the global sequence in the same order that they are examined and chosen in. More precisely, a thread bases its decision for the next best move m on the partition created by the current global move sequence $m_1, \ldots, m_j$ (with $j < n$). In a sequential setting, this would mean that m becomes the next move m_{j+1} in the sequence, improving the partition objective by the gain calculated for m. In the parallel setting, however, any number of concurrent moves $m_{j+1}, \ldots, m_{j+i}$ (with $i \geq 1$) may be executed first. This means m is actually applied to the partition created by the global move sequence $m_1, \ldots, m_{j+i}$ instead. Then, the actual gain of performing m on that partition may vary from the expected gain that was initially calculated for m. Thus, parallel FM implementations are unable to create global sequences of best possible moves with certainty and instead approximate sequences found by sequential FM.

A possible parallelization of k-way FM is to execute parallel localized FM searches around small subsets of active refinement nodes [2, 18]. Each localized FM search, then, constructs a local sequence of moves by performing moves that are initially not visible to concurrent searches. For that purpose, local changes can be stored as deltas to the global partition in a thread-local hash table data structure.
The local sequences can be applied to the global partition in different manners: The authors of Mt-KaHiP [2] propose storing the local sequences until all localized searches have terminated. Then, the local sequences can be written to the global partition one after another, recalculating gains and choosing the best prefix for every sequence.
In Mt-KaHyPar [18] threads instead commit the best prefix of a local sequence to the global partition immediately when a localized search finishes. Thus, using time stamping, a global move sequence develops. However, not all gains in the global move sequence may be as expected due to moves being performed on the global partition concurrently. Therefore, when all localized searches terminate, the gains of all moves in the global sequence are recalculated in parallel [18] and the global sequence is rolled back to the best prefix. This method can be optimized by writing any local sequence with expected positive total gain to the global partition as soon as it is found. In effect, shorter local sequences are applied to the global partition, which limits the divergence of expected and actual gains.
This parallelization of the FM heuristic has been shown to be scalable and is able to produce partitions of quality comparable to those found by sequential partitioners [18, 17].

3.3. Sequential Hypergraph Partitioners

Sequential hypergraph partitioners are well researched and employ a variety of sophisticated techniques to achieve good efficiency or the best possible partition quality. We give a short overview here and refer to Ref. [40] for a comprehensive history of hypergraph partitioning. Due to the complex nature of direct k-way partitioning [9, 15, 4], especially k-way FM refinement [37, 38, 15, 1], recursive bipartition algorithms have seen wide success being able to utilize 2-way FM to find k-way partitions [41, 31, 12, 45, 26]. The partitioner PatoH [8] is notable for using fast 2-way FM refinement on boundary vertices during recursive bipartitioning to achieve reasonable partition quality with great time efficiency. The first direct k-way partitioner to compete with recursive bipartition algorithms in both time and quality was *hMetis* [26] using k-way label propagation refinement. Later, Akhremtsev et al. improved on previous approaches to direct k-way FM refinement [37, 38, 34, 20] in the hypergraph partitioning framework KaHyPar [1], establishing direct k-way partitioning as the state of the art for the highest possible partition quality. The direct k-way, n-level, sequential KaHyPar has subsequently seen further improvement to solution quality using flow-based refinement [22, 16] and to our knowledge now outperforms any other partitioner regarding partition quality [42].

3.4. Parallel Hypergraph Partitioners

The parallelization of hypergraph partitioning is not trivial because hypergraph data does not naturally decompose well (which is part of the reason why hypergraph partitioning is important as pre-processing for other problems in the first place). Refining a hypergraph partition in parallel is particularly difficult as data that different threads work on may be highly interdependent, leading to threads interfering with each others' gains and balance considerations as described in Section 3.2. In this section we give an overview over the parallelization techniques employed by the shared-memory hypergraph partitioner Mt-KaHyPar that this work is based on. Additionally, we give a quick summary of the approaches chosen by other parallel partitioners.

3.4.1. Mt-KaHyPar

Mt-KaHyPar was the first shared-memory hypergraph partitioner. Its initial version [18], now known within the Mt-KaHyPar framework as *Mt-KaHyPar-D*, is a $\mathcal{O}(\log n)$-level, direct k-way partitioner. Later, the authors published an n-level version of Mt-KaHyPar [17], aka *Mt-KaHyPar-Q*, which, while less time efficient, is geared towards the best possible solution quality in a parallel setting. The quality of partitions found by this parallel partitioner has been shown to compete with the quality of the sequential KaHyPar partitioner, only falling behind due to KaHyPar's optimization using flows [17][1]. The approach to asynchronous uncoarsening presented in this paper is integrated into Mt-KaHyPar-Q and bases its refinement on techniques used therein. Moreover, this work utilizes in large parts the same underlying data structures and uses the coarsening and initial partitioning phase of Mt-KaHyPar-Q without change. Therefore, the rest of this section is concerned predominantly with an outline of the Mt-KaHyPar-Q variant. In the rest of this thesis we assume "(batch synchronous) Mt-KaHyPar" to mean the Mt-KaHyPar-Q variant, unless stated otherwise.

Coarsening. The coarsening phase of Mt-KaHyPar proceeds in passes. A pass consists of a parallel iteration over all hypernodes of the current hypergraph. For every node v the algorithm finds the best node to contract v onto using the heavy-edge rating function [8, 21], inspired by the clustering algorithm used for coarsening in Mt-KaHyPar-D [18]. For the sake of using a clustering quality function, Mt-KaHyPar can be understood to treat every node as its own cluster at any point. For that purpose, legal contractions are performed immediately once the contraction partner is found. A contraction (u, v) is considered legal if it does not impair with the correctness of the global contraction forest being constructed. More specifically, (u, v) must not create a cycle in the forest and the contraction of u cannot have started yet. The pass attempts to contract every node onto another but obviously not all nodes can be contracted in one pass. The safety mechanisms in place to prevent illegal contractions establish those nodes that will not be contracted. These nodes only act as representatives of contractions during the pass.
In the later uncoarsening phase, the order of uncontractions will depend on the order of contractions in the coarsening phase. Therefore, during the coarsening phase, Mt-KaHyPar increases an atomic counter at the beginning and at the end of each contraction. Thus, each contraction is assigned a time interval that can be used to order uncontractions later. After each contraction pass, nets with only a single pin are removed as they cannot contribute to the objective. Also, parallel nets, i.e. nets containing the exact same set of pins, are merged, leaving one representative net with the accumulated net weight. The coarsening phase ends when the coarsened hypergraph contains fewer than $160 \cdot k$ nodes after a pass. The contraction phase is truly n-level as the contraction partner is individually chosen for each node. Nevertheless, the coarsening phase contains $\mathcal{O}(\log n)$ synchronization points in which single-pin and parallel nets are removed.

[1] A yet unpublished version of Mt-KaHyPar that uses flows for better refinement is in the works.

Initial Partitioning. Mt-KaHyPar employs n-level recursive bipartitioning to find an initial partition. The coarsening and uncoarsening phases for the bipartitioning are parallelized as in the main partitioner. A 2-way partition of the coarsest hypergraph is found using a portfolio of 9 flat partitioners. At least 5 runs of each flat partitioner are performed. After that, a particular flat algorithm is run again (up to a maximum of 20 times) only if after a run it promises a better solution than the current best according to a stochastic model.

Uncoarsening: Batches. At the beginning of the uncoarsening phase, Mt-KaHyPar creates a sequence of batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$ of contractions comprising a partition of all contractions performed in the coarsening phase. The goal is to be able to safely revert all contractions within a batch in parallel, allowing the uncoarsening phase to proceed through the sequence batch by batch, applying refinement in between batches. The size of any batch is restricted by an input parameter b_{max} that interpolates between scalability and solution quality. Ensuring $\forall B \in \mathcal{B} : |B| \approx b_{max}$ maximizes parallelism within these restraints. To guarantee that contractions within the same batch can be reverted in parallel safely, certain rules apply based on the coarsening phase. These rules require that some contractions must be or cannot be placed in the same batch. Suppose two contractions (u, v) and (u, v') with $v \neq v'$ were performed in the coarsening phase. We say, the contractions have time overlap if their time intervals, as determined in the coarsening phase, intersect. The authors discern three rules:

1. Revert contraction $(\cdot, u)$ before reverting any contraction $(u, \cdot)$
2. If (u, v) and (u, v') were contracted with time overlap, then revert them without intermittent refinement
3. If (u, v) was contracted strictly before (u, v'), then revert (u, v) after (u, v')

Because of rule (1.), any contraction $(u, \cdot)$ has to be placed in a later batch than the contraction $(\cdot, u)$. Rule (2.) dictates that any contractions onto the same representative with time overlap have to be placed in the same batch. Via rule (3.), contractions onto the same representative without overlap cannot be placed in the same batch. Instead, they need to be placed in different batches in the reverse order of their contraction.

Uncoarsening: Localized Refinement. After uncontracting a batch, Mt-KaHyPar performs parallel localized LP refinement followed by parallel localized FM refinement (see Section 3.2.1 and Section 3.2.2). With this approach, the LP refinement serves to find easy improvements quickly so the subsequent FM refinement converges faster. Both refinement heuristics are executed repeatedly until neither finds an improvement to the partition quality in one run. Initially, all border nodes in the batch, i.e. all uncontracted nodes and representatives in the batch that are incident to a net spanning more than one block, are considered seed nodes for the refinement. For gain calculation, Mt-KaHyPar uses a gain cache. For every node this cache stores the benefit to the objective function for moving the node out of its current block as well as the penalty for moving it to any of the other blocks. This gain cache allows examination of the best possible move of a node in $\mathcal{O}(k)$ and is updated when the partition changes due to moves and uncontractions (see also Chapter 5).

Uncoarsening: Remarks on Parallel Localized FM. Mt-KaHyPar uses parallel localized FM searches. Each localized FM search acquires a small number of border nodes in the batch as its refinement seeds (five seeds per search by default). Additionally, when an FM search expands, it acquires exclusive ownership of all activated nodes. That way, the localized FM searches are prevented from overlapping. New localized FM searches are started until all border nodes in the batch have been moved by a search. Whenever a localized FM search terminates or finds a local move sequence with net positive gain, Mt-KaHyPar immediately writes the local move sequence to the global partition and appends it to a global move sequence as described in Section 3.2.2. Additionally, attributed gains (see Section 3.2.1)

track the actual gains of moves applied to the global partition. These actual gains, then, allow Mt-KaHyPar to revert to the best prefix of the local sequence on the global partition.

Uncoarsening: Global Refinement. The single-pin and parallel nets removed in the coarsening phase have to be restored in the uncoarsening phase. All nets that have been removed together at a specific synchronization point of the coarsening phase are restored collectively once a specific batch is uncontracted. Whenever single-pin and parallel nets are restored, an additional global refinement pass is performed before continuing with the next batch. Eventually, when the original hypergraph is restored, a global rebalancer may fix the partition balance. This may become necessary because the global rollback is allowed to accept prefixes that break the balancing constraint by a small constant factor. Usually, such imbalances are sorted out by subsequent label propagation runs as they attempt to execute moves that improve the partition balance if none that improve the partition quality can be found.

Uncoarsening: Data Structures for Gain Calculation. For each hyperedge e, Mt-KaHyPar explicitly stores the connectivity set $\Lambda(e)$ as well as the number of pins $\Phi(e, i)$ in every block $i \in [k]$. As proposed in Mt-KaHyPar-D [18], the memory representations for both Λ and Φ use a format optimizing the number of bits needed per entry. Writes to the data structures are synchronized using a separate spin-lock per hyperedge. Atomic fetch-and-add operations are not feasible due to the optimized memory format. Each thread holds at most one hyperedge lock at the same time. When a thread moves or uncontracts a node u, the thread updates the $\Lambda(e)$ and $\Phi(e, \cdot)$ values of hyperedges $e \in I(u)$ one after another. It acquires the lock for one hyperedge, performs the update, releases the lock and only then acquires the lock for the next hyperedge. Reads from the data structures are not synchronized.

3.4.2. Other Parallel Partitioners

Various approaches to parallel partitioning have been explored. The shared-memory, direct k-way parallel graph partitioner Mt-KaHiP [2] aims to find high quality partitions. Some core aspects of Mt-KaHiP like parallel localized searches inspired features of Mt-KaHyPar. Other parallel partitioners are mostly only tangentially related to this work. Here, we merely give a summary by listing their high-level properties: ParHiP [33] (complex graph networks, distributed-memory, direct k-way), Mt-Metis [29] (graphs, shared-memory, direct k-way), BiPart [31] (hypergraphs, deterministic, shared-memory, recursive bipartitioning), Parkway [44] (hypergraphs, distributed-memory, direct k-way), Zoltan [12] (hypergraphs, 2D memory distribution, recursive bipartitioning). Furthermore, two yet unpublished versions of Mt-KaHyPar exist: One variant uses flow-based refinement to find high-quality partitions and the other variant makes the parallel partitioner deterministic. All parallel partitioners mentioned here follow the multilevel scheme.

4. A Framework for Asynchronous n-Level Hypergraph Partitioning

The uncoarsening phase in n-level hypergraph partitioning consists of reverting almost n contraction operations. The initial partition is projected onto uncontracted nodes and refined by applying local search algorithms on each level of the multilevel hierarchy. Refinement strives to optimize the objective function while adhering to the balance constraint.

Optimally, with each new piece of information the partition should be reevaluated. This would entail localized refinement around the uncontracted node after each uncontraction. For parallel computing that idea is, however, not scalable as it induces one global synchronization point per uncontraction. Yet, only applying localized refinement algorithms after a set number of uncontractions can deliver partitions that have comparable quality to sequential partitioners as batch synchronous Mt-KaHyPar [17] shows (see Section 3.4.1). The idea is to revert b contraction operations in parallel and afterwards apply parallel localized refinement around the uncontracted nodes. This scheme reduces the number of synchronization points by a factor of b. However, that approach still leaves us with $\mathcal{O}(n/b)$ synchronization points for the uncoarsening phase. These synchronization points afflict the possible scalability of the partitioner.

The goal of this thesis is to minimize the number of synchronization points further to improve scalability and potentially reduce running times. For that reason, we strive to achieve fully asynchronous uncoarsening wherein uncontractions of nodes and refinement are executed concurrently. Our asynchronous uncoarsening approximates a parallelization of the original n-level idea: Each thread uncontracts a node and, afterwards, employs localized refinement around the uncontracted node while other threads concurrently also perform uncontractions and localized refinement.

In this chapter, we present an outline of our shared-memory partitioner with asynchronous uncoarsening. Our algorithm is integrated into the shared-memory hypergraph partitioning framework Mt-KaHyPar [17] (see Section 3.4.1). We give a top-level overview of our partitioner in Algorithm 4.1. Our algorithm follows the direct k-way, n-level hypergraph partitioner scheme: First, the input hypergraph is coarsened (line 1) using the coarsening phase of Mt-KaHyPar described in Section 3.4.1. Then, we obtain an initial partition Π (line 2) using the initial partitioning phase of Mt-KaHyPar explained in Section 3.4.1. Our contribution lies in the subsequent asynchronous parallel uncoarsening phase (line 3). Eventually, we apply top-level global refinement and rebalance the partition if necessary (lines 4-5).

Algorithm 4.1: Algorithm Outline

Input: $H = (V, E, c, \omega);\ k : \mathbb{N}_0$
Result: $\Pi : V \to [k]$
1 $(H_{coarse}, \mathcal{F}) \leftarrow coarsen(H, k)$ `// Section 3.4.1`
2 $\Pi \leftarrow initialPartitioning(H_{coarse}, k)$ `// Section 3.4.1`
3 $\Pi \leftarrow uncoarsen(H_{coarse}, \Pi, \mathcal{F})$ `// Chapter 4`
4 *refine* Π *globally using parallel FM*
5 *rebalance* Π *if necessary*

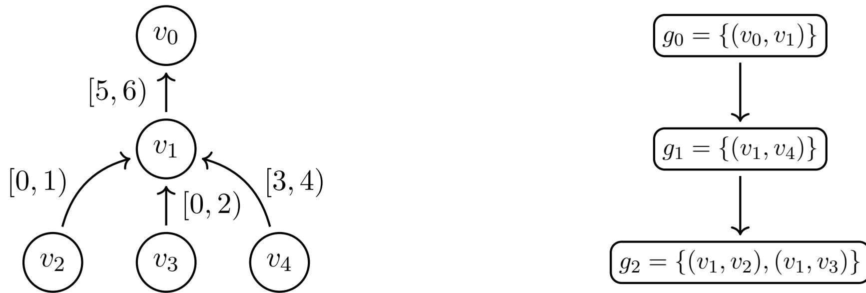

Figure 4.1.: Left: An example contraction tree with time intervals for each contraction. Right: The resulting uncontraction group hierarchy.

In the following, we discuss our main contribution: an asynchronous parallel uncoarsening scheme, that performs uncontractions and refinement concurrently. First, we reiterate and contextualize the rules for ordering uncontractions in parallel uncoarsening found by the authors of synchronous Mt-KaHyPar [17]. Then, we go into detail about how we implement parallel asynchronous uncoarsening using that order of uncontractions. Afterwards, we discuss how localized refinement can be applied effectively in asynchronous uncoarsening. Finally, we explain how some remaining necessary synchronization points comprise a limitation to the asynchronous approach.
This chapter only illustrates our base idea of realizing asynchronous uncoarsening. Chapter 5 deals with the intricacies of gain calculation in an asynchronous setting. Chapter 6 explains challenging cross-dependencies arising with concurrent uncontractions and refinement.

4.1. Order of Uncontractions

Uncontraction Order. Any valid sequence of contractions form a so-called contraction forest. One can revert the contractions by uncontracting the nodes in the contraction forest in a top-down fashion in parallel. Once we uncontract a node u, its children in the contraction forest, i.e. the nodes contracted onto u, become eligible for uncontraction. However, the authors of Mt-KaHyPar [17] showed that there remain some dependencies defining in which order siblings in the contraction forest must be uncontracted (see Section 3.4.1). They found that sibling nodes in the forest must be uncontracted in reverse order of their contraction to ensure correctness. However, since contractions are performed in parallel some sibling contractions may happen concurrently. Therefore, in Mt-KaHyPar an atomic counter is incremented before and after each contraction to obtain a specific time interval for each contraction. Siblings with intersecting time intervals then have to be uncontracted without intermittent refinement.

Synchronous Mt-KaHyPar embeds the uncontractions in a sequence of batches such that the nodes within each batch can be uncontracted independently in parallel. The batches are also constructed in such a manner that the uncontraction of a batch resolves the last

dependencies required to uncontract the next batch. Localized refinement is only applied between the uncontraction of batches.

Uncontraction Groups. In asynchronous uncoarsening, however, our goal is to apply localized refinement immediately after each uncontraction. As stated above, though, siblings in the contraction forest with intersecting time intervals disallow intermediate refinement. Therefore, we group such siblings together, only applying localized refinement after all of their uncontractions are finished. More specifically, let $S := \{((u,v),(u,w)) \in V^2 \times V^2 \mid (u,v) \text{ and } (u,w) \text{ have intersecting time intervals}\}$. We call an equivalence class of the transitive closure of S an *uncontraction group*. Then, we have to uncontract all contractions in an uncontraction group before we can apply localized refinement around all border nodes contained in the group. Note that all uncontractions in the same group have the same representative which we also call the representative of the group.

Order of Uncontraction Groups. The order of uncontractions put forth by the authors of Mt-KaHyPar [17] also defines the order in which uncontraction groups have to be uncontracted. Firstly, groups with the same representative contain sibling uncontractions and, therefore, have to be uncontracted in reverse order of the contraction of the siblings. Secondly, a group with representative u may only be uncontracted once the group containing the contraction of u has been uncontracted.

In Figure 4.1 we depict an example contraction forest with contraction time intervals on the left. The forest of uncontraction groups shown on the right originates as follows. Node v_1 has no siblings in the contraction forest. Therefore, we find an uncontraction group $g_0 = \{(v_0, v_1)\}$. Node v_4 does have siblings but their contractions have no time overlap with the contraction of v_4. Thus, we get an uncontraction group $g_1 = \{(v_1, v_4)\}$. The sibling nodes v_2 and v_3 have been contracted with time overlap which means they comprise a group $g_2 = \{(v_1, v_2), (v_1, v_3)\}$. As v_1 is the parent of nodes v_2, v_3 and v_4 in the contraction forest, the group g_0 containing the uncontraction of v_1 has to be uncontracted first. Furthermore, v_4 has been contracted strictly later than v_2 and v_3 so group g_1 has to be uncontracted next. Finally, g_2 can be uncontracted.

4.2. Asynchronous Uncoarsening Scheme

In the following, we explain our asynchronous uncoarsening scheme based on uncontraction groups. A summary is provided in Algorithm 4.2.

Group Traversal Order. Initially, we have to construct the forest $\mathcal{F}_{group}$ of uncontraction groups (line 1). For that purpose, we iterate through the contraction forest $\mathcal{F}$ that was constructed in the coarsening phase from roots to leaves. We consult the time intervals of sibling contractions to find uncontraction groups and the ordering among sibling groups. We parallelize this by iterating over the different trees of $\mathcal{F}$ in parallel.

Once the group forest is completed, we want to traverse the uncontractions in parallel. Different trees of $\mathcal{F}_{group}$ contain uncontractions in different regions of the hypergraph. Therefore, we want different threads to work in different trees of $\mathcal{F}_{group}$ to reduce the likelihood of concurrent localized refinement searches interfering with each others' gains. To achieve that behavior, we favor performing the uncontractions in the forest in a breadth-first style order. The BFS traversal is facilitated by a relaxed parallel priority queue of groups ordered by their depth in $\mathcal{F}_{group}$[1]. This priority queue represents a pool of currently eligible or *active* uncontraction groups. We insert a group into the priority queue when we mark it as active. We start by marking the roots of $\mathcal{F}_{group}$ as active (line 2).

[1]We find that MultiQueues [48] work well for this approach as they provide fast access and additionally inherently randomize the order of groups among groups with the same depth.

Algorithm 4.2: uncoarsen

```
    Input: H = (V, E, c, ω); Π : V → [k]; F : Contraction Forest
    Result: Π : V → [k]
1   F_group ← constructGroupForest(F)
2   mark roots of F_group as active
3   start parallel worker threads, each performing:
4       seeds ← ∅
5       loop:
6           g ← try to pick active group
7           if no active group found then
8               if all groups in F_group uncontracted already then
9                   join thread
10              else
11                  go to loop
12          uncontract(g, H, Π)
13          seeds ← seeds ∪ border vertices of Π contained in g
14          if |seeds| > minSeeds then
15              localizedRefine(seeds, H, Π)
16              seeds ← ∅
17          mark children of g in F_group as active
18          go to loop
19  wait for all worker threads
```

Uncoarsening Tasks. The asynchronously parallel uncoarsening begins by spawning worker threads (line 3). A worker thread works on an uncoarsening task which consists of the following steps: First, the thread tries to pick an active group from the priority queue (line 6). If no active group is available, the thread retries picking a group until an active group becomes available (lines 7-11). While waiting, the thread intermittently checks whether in the meantime a different thread has uncontracted the last group in $\mathcal{F}_{group}$. In that case the thread terminates (line 8-9).

If an active group has been found, the thread performs all uncontractions in the group (line 12). Also, it collects all border nodes contained in the group which we use as seed nodes for our localized refinement algorithms (line 13).

Subsequently, if a sufficient number of seeds is available, the worker thread applies localized refinement (lines 14-16). Our asynchronous localized refinement is presented in more detail in Section 4.3. A worker thread only applies localized refinement after having collected a small minimum number of refinement seeds, possibly from multiple groups (line 14). This minimum number of seeds is necessary for localized FM refinement. Generally, only few localized FM searches will yield an improvement with most FM moves having to be reverted. With too few refinement seeds, a localized FM search would initially likely be forced to perform a move that increases the connectivity of incident hyperedges. Such a move triggers many costly delta gain cache updates. By using several seeds for localized refinement, we are more likely to be able to move a seed with positive gain. Thus, we reduce the running time as costly unpromising moves are pruned. For our purposes, even a minimum of five seeds is enough for a large improvement in running time compared with always refining after uncontracting a group. Localized FM searches in batch synchronous Mt-KaHyPar [17] also acquire five seeds each. There, this value has proven to have negligible effect on partition

quality compared with one seed per search. We consider the minimum number of seeds as a configuration parameter in our parameter tuning experiments (see Appendix A).

When the localized refinement search concludes, all children of the group in $\mathcal{F}_{group}$ are marked as active and, thus, inserted into the priority queue (line 17). Afterwards, the worker thread continues by attempting to pick a new active group (line 18).
As the number of groups is known in advance, a check for termination does not depend on emptiness of the priority queue. Instead, the number of uncontracted groups is tracked and each thread simply terminates when no more uncoarsening tasks are due (lines 8-9).

4.3. Asynchronous Localized Refinement

In this section, we describe in more detail how localized refinement heuristics can be applied effectively in the asynchronous context, i.e. concurrently to each other and to uncontractions. The method *localizedRefine* in line 15 of Algorithm 4.2 is comprised as follows: Inspired by the localized refinement used in synchronous Mt-KaHyPar [17], a worker thread applies localized LP first as this heuristic inexpensively finds obvious improvements. Then, the subsequent localized FM refinement that uses the same seeds converges faster. The worker thread repeatedly applies an LP search followed by an FM search on the same seeds until neither finds an improvement to the partition objective. However, due to the small number of seeds per call to *localizedRefine*, only a small fraction of calls will find an improvement and perform multiple iterations of LP and FM. It is notable that parallelization is only achieved through asynchronously concurrent refinement on different sets of seeds. Every thread only ever works on the seeds it collected in uncontractions that it performed itself.
We disallow overlapping localized FM searches by requiring FM searches to acquire exclusive ownership over their active nodes as proposed by the authors of synchronous Mt-KaHyPar [18] (see Section 3.4.1). Additionally, a single localized FM search may only move each node at most once. However, when an FM search terminates, the nodes moved by that search may immediately be moved again. Parallel localized FM in synchronous Mt-KaHyPar uses gain recalculation on a global move sequence as a sanity check with global perspective. In asynchronous uncoarsening, we omit global gain recalculation for the sake of avoiding global synchronization. Instead, we can only assume that the move decisions made using only local information collectively result in a good global partition without any global perspective.

For asynchronous localized refinement, it is important that FM searches take priority over LP searches. More specifically, suppose an LP search and an FM search concurrently work in the same hypergraph region, i.e. on overlapping or neighboring node sets. The FM search is able to navigate out of a local minimum of the objective function using negative gain moves. At the same time, the LP search always performs moves steering towards the nearest local minimum. Thus, if we allow the LP and FM search to work on overlapping sets of nodes, the FM search will attempt to climb a hill in the objective function while the LP search pulls it back down. Therefore, we disallow LP searches moving nodes that are considered active by a concurrent FM search. We implement this by utilizing the fact that FM searches already acquire exclusive ownership of every node they consider active in order to keep the FM searches non-overlapping. We simply require LP searches to acquire ownership of a node using the same acquisition mechanism before being allowed to move it. If an LP search is not able to move a node because it is held by an FM search, the LP search reexamines the move at the end of the LP pass. Note that an LP search only acquires ownership of a node when it tries to move it, not when it activates it during expansion. That way, an FM search may still expand into the active region of an LP search.

Among localized LP searches, a different node acquisition mechanism prevents multiple LP searches from working on overlapping sets of active nodes. That way, nodes are not unnecessarily examined by multiple LP searches simultaneously. This is purely a matter of increasing the efficiency of parallel LP searches.

4.4. Remaining Synchronization Points

With the asynchronous uncoarsening algorithm described in Section 4.2, no synchronization points between the worker threads are needed for uncontractions or refinement. However, the coarsening phase that we adopt from synchronous Mt-KaHyPar removes single-pin and parallel nets during its inherent $\mathcal{O}(\log n)$ synchronization points as described in Section 3.4.1. Each contraction is performed between two such synchronization points that define which nets are incident to the contracted node at the time of the contraction. Exactly those incident nets are affected by the contraction but also later by the associated uncontraction. Thereby, all nets that were removed after a contraction during the coarsening phase have to be restored in the uncoarsening phase before the contraction can be reverted.

Thus, even in asynchronous uncoarsening we are left with $\mathcal{O}(\log n)$ synchronization points in which parallel and single-pin nets have to be restored. Consequently, we apply stretches of fully asynchronous uncoarsening only between those synchronization points. We compute an uncontraction group forest for each such stretch instead of one forest for the whole uncoarsening phase. Furthermore, after restoring single-pin and parallel nets at a synchronization point, we utilize the synchronization to apply a pass of global parallel FM refinement.

5. Gain Cache for Asynchronous Uncoarsening

Recall that we define the gain $g_{s \to t}(u)$ of moving a node u from V_s to V_t using the move-from-benefit $b(u, s)$ and the move-to-penalty $p(u, t)$ as follows:

$$\begin{aligned} g_{s \to t}(u) &:= b(u, s) - p(u, t) \\ \text{with } b(u, s) &:= \omega(\{e \in I(u) \mid \Phi(e, s) = 1\}), \\ \text{and } p(u, t) &:= \omega(\{e \in I(u) \mid \Phi(e, t) = 0\}) \end{aligned}$$

Synchronous Mt-KaHyPar uses a gain cache to compute the gain of moving a node to a different block in its refinement algorithms. For each node $u \in V$ the gain cache used in synchronous Mt-KaHyPar stores the benefit $b(u, \Pi(u))$ for moving u out of its current block $\Pi(u)$. Additionally, the gain cache stores the penalty $p(u, i)$ for moving u to block i for all $i \in [k]$. Then, the gain for a move (u, s, t) is calculated as $g_{s \to t}(u) = b(u, \Pi(u)) - p(u, t)$. This gain cache allows constant time access to the gain of any given move. Thus, the LP and FM algorithms use the gain cache to determine the best possible move of a node in $\mathcal{O}(k)$. Whenever a node v is moved or uncontracted, the gain cache entries for any adjacent node u may need to be updated as described in detail in Ref. [18] and [17]. As synchronous Mt-KaHyPar stores only one benefit entry $b(u, \Pi(u))$ per node u, the term can no longer be correctly updated after u is moved without extensive locking overhead. However, the refinement algorithms in synchronous Mt-KaHyPar work in rounds where each node is only moved once per round. That way, the benefits of moved nodes can safely be recalculated after each round [18].

Asynchronous uncoarsening lacks such a concept of refinement rounds. Therefore, we use a slightly modified version of the gain cache where for each node $u \in V$ we store $b(u, i)$ for all $i \in [k]$. This means that a hyperedge $e \in I(v)$ contributes $\omega(e)$ to $b(v, j)$ if $\Phi(e, j) = 1$, i.e. if exactly one pin of e is in block V_j, without regarding whether v is that one pin in $e \cap V_j$. In effect, the gain cache can tell us the benefit for moving v from *any* block at any time, not only for moving it from $\Pi(v)$. That way, we correctly cache the current benefit of a node irrespective of any moves of the node.
However, using k benefit entries per node introduces additional complexity regarding gain cache updates. In the following sections, we address updates to our modified gain cache for uncontractions and node moves.

5.1. Uncontraction Gain Cache Update

An uncontraction demands a different update to the gain cache in two cases per hyperedge incident to the uncontracted node as identified by Gottesbüren et al. [17]: Consider a contraction (u, v) and a hyperedge e that was incident to v before the contraction. At that point in time, either (i) u was not a pin of e which means u replaced v as a pin of e or (ii) u was already a pin of e.

Benefit Update. In case (i) the uncontraction of (u, v) replaces u with v as a pin of e. Thus, the benefits associated with e are transferred from u to v by decreasing $b(u, i)$ and increasing $b(v, i)$ by $\omega(e)$ for all blocks for which e currently contributes a benefit, i.e. for all blocks $i \in \Lambda(e)$ with $\Phi(e, i) = 1$.
In case (ii) both u and v are pins of e after the uncontraction. If u was previously the only pin of e in $\Pi(u)$, then e contributed $\omega(e)$ to the benefits $b(w, \Pi(u))$ of every pin $w \in e$ before the uncontraction. Now, however, there is more than one pin of e in block $\Pi(u)$ which means e no longer contributes to the benefits of $w \in e$ in that way. Consequently, we have to reduce the benefits $b(w, \Pi(u))$ by $\omega(e)$ for all $w \in e$ except v as it was previously inactive. Additionally, to initialize the benefit entries of v, we have to increase $b(v, i)$ by $\omega(e)$ for every block $i \in \Lambda(e)$ with $\Phi(e, i) = 1$.

Penalty Update. In both cases, v becomes a pin of e so we have to add $\omega(e)$ to its penalty $p(v, j)$ for any blocks that are not connected to e, i.e. to blocks $j \in [k] \setminus \Lambda(e)$. If after the uncontraction $u \notin e$ (case (i)), then we also subtract $\omega(e)$ from $p(u, j)$ for those blocks $j \in [k] \setminus \Lambda(e)$ as u is no longer a pin of e.

Summary. To summarize, the uncontraction (u, v) affects the gain cache entries of pins of any hyperedge $e \in I(v)$ as follows (Φ values represent the state after the uncontraction):

$$
\begin{aligned}
&(i.)\ \text{If } v \in e \text{ and } u \notin e \ (v \text{ replaces } u \text{ as pin), then} \\
&\quad (i.a.)\ \forall i \in \Lambda(e):\ \text{If } \Phi(e,i) = 1, \text{ then } \begin{cases} b(u,i) \mathrel{-}= \omega(e), \\ b(v,i) \mathrel{+}= \omega(e) \end{cases} \\
&\quad (i.b.)\ \forall j \in [k] \setminus \Lambda(e): \begin{cases} p(u,j) \mathrel{-}= \omega(e), \\ p(v,j) \mathrel{+}= \omega(e) \end{cases} \\
&(ii.)\ \text{If } u, v \in e \ (v \text{ is added as pin), then} \\
&\quad (ii.a.)\ \forall i \in \Lambda(e):\ \text{If } \Phi(e,i) = 1, \text{ then } b(v,i) \mathrel{+}= \omega(e) \\
&\quad (ii.b.)\ \text{If } \Phi(e, \Pi(u)) = 2, \text{ then } \forall w \in e \setminus \{v\}:\ b(w, \Pi(u)) \mathrel{-}= \omega(e) \\
&\quad (ii.c.)\ \forall j \in [k] \setminus \Lambda(e): p(v,j) \mathrel{+}= \omega(e)
\end{aligned}
\tag{5.1}
$$

Differences to Synchronous Mt-KaHyPar. In synchronous Mt-KaHyPar, the gain cache update for an uncontraction (u, v) is less complex [17]: The benefit update $(i.a.)$ only takes $\mathcal{O}(1)$ instead of $\mathcal{O}(k)$ work as the gain cache stores one benefit entry per node. Similarly, the benefit entries for v do not need to be initialized if v is added as a pin, i.e. $(ii.a.)$ is not necessary. In the synchronous case, u may be replaced as a pin of e by a different node $v' \neq v$ by a concurrent uncontraction (u, v')[1]. Therefore, $(ii.b.)$ still takes *up to* work of $|e|$ to find v' in the pin list of e. However, in practice finding v' is still faster than in the asynchronous context where we update $|e| - 1$ benefit entries.

[1] In asynchronous uncoarsening, a lock is held on the representative u during an uncontraction (see also Section 6.1), which means no two concurrent uncontractions from u are possible and this issue does not arise.

5.2. Node Move Gain Cache Update

When a node is moved from one block of the partition to another during refinement, the move-to-penalties and move-from-benefits of all pins of every hyperedge incident to the moved node may be affected. An uncontraction always triggers a gain cache update at least to initialize the gain cache entries for the uncontracted node. A node move, however, only requires a gain cache update if the pin count of a hyperedge within a block decreases to zero or one or increases to one or two.

Benefit Update. A hyperedge $e \in I(u)$ contributes $\omega(e)$ to $b(u, \Pi(u))$ exactly if u is the only pin of e in the block $\Pi(u)$. Therefore, moving a node u' from V_s to V_t with $e \in I(u)$ affects the benefits of pins of e in four cases as follows: First, if the move decreases the pin count of e in block V_s to zero, all pins of e lose the benefit for being moved out of V_s. Second, if the move decreases the pin count of e in block V_s to one, all pins of e gain a benefit for being moved out of V_s. Third, similarly, if the move increases the pin count of e in block V_t to one, all pins of e gain a benefit for being moved out of V_t. Fourth, if the move increases the pin count of e in block V_t to two, all pins of e lose the benefit for being moved out of V_t. It may seem counter-intuitive to talk about *all* pins of a hyperedge e gaining or losing a benefit for being moved out of V_s or V_t when not all pins are in those blocks. However, updating the benefits of all pins in that way is important to correctly track the benefits of all pins in the presence of concurrent moves as any pin may be moved to V_s or V_t concurrently.

Penalty Update. A hyperedge $e \in I(u)$ contributes $\omega(e)$ to $p(u, i)$ exactly if $\Phi(e, i) = 0$. Consider moving a node u' from V_s to V_t. If the move decreases the pin count of e in block V_s to zero, all pins of e gain a penalty for being moved to V_s. Conversely, if the move increases the pin count of e in block V_t to one, all pins of e lose the penalty for being moved to V_t.

Summary. To summarize, a move (u, s, t) affects the gain cache entries in the gain cache of pins of any hyperedge $e \in I(u)$ as follows (Φ values after the move):

$$\begin{aligned}
&(i.) \quad \text{If } \Phi(e,s) = 0, \text{ then } \forall v \in e : \begin{cases} b(v,s) \mathrel{-}= \omega(e), \\ p(v,s) \mathrel{+}= \omega(e) \end{cases} \\
&(ii.) \quad \text{If } \Phi(e,s) = 1, \text{ then } \forall v \in e : b(v,s) \mathrel{+}= \omega(e) \\
&(iii.) \text{ If } \Phi(e,t) = 1, \text{ then } \forall v \in e : \begin{cases} b(v,t) \mathrel{+}= \omega(e), \\ p(v,t) \mathrel{-}= \omega(e) \end{cases} \\
&(iv.) \quad \text{If } \Phi(e,t) = 2, \text{ then } \forall v \in e : b(v,t) \mathrel{-}= \omega(e)
\end{aligned} \tag{5.2}$$

Differences to Synchronous Mt-KaHyPar. In synchronous Mt-KaHyPar, the gain cache update for a move (u, s, t) is less complex [18]: In case *(ii.)* and *(iv.)* only the benefit entries for the pins of e in block V_s and V_t, respectively, need to be updated as only one benefit entry per node is stored. Also, benefit entries of moved nodes are allowed to be invalidated so in cases *(i.)* and *(iii.)* no benefits need to be updated. Consequently, in each of the four cases, the gain cache update takes $\Theta(|e|)$ less work than in asynchronous uncoarsening.

5.3. Implementation Details

As in synchronous Mt-KaHyPar [18, 17], all updates to gain cache entries use atomic fetch-and-add operations as changes to any entry can be caused by concurrent uncontractions and refinement at any time. Also, we store the underlying Λ and Φ values in the same data structures used in synchronous Mt-KaHyPar (see Section 3.4.1). Writes to the Λ and Φ

values for a gain cache update are synchronized via one spin lock per hyperedge. Reads are not synchronized.

It is notable that, as in synchronous Mt-KaHyPar, our implementation does not explicitly store $[k] \setminus \Lambda$ but only Λ. Therefore, we utilize $p(u, j) = \omega(I(u)) - p'(u, j)$ with $p'(u, j) = \omega(\{e \in I(u) \mid \Phi(e, j) > 0\})$. With this, our gain cache actually stores $\omega(I(u))$ and $p'(u, j)$ instead of $p(u, j)$ for every $j \in [k]$. Then, we can update the penalty of a node by changing the stored $p'(u, \cdot)$ and $\omega(I(u))$ values using just Λ. In practice $|\Lambda(e)| < |[k] \setminus \Lambda(e)|$ for each $e \in E$, so we accelerate updates to penalty entries for uncontractions by storing p' instead of p.

5.4. Evaluation of the Gain Cache

The gain cache in asynchronous uncoarsening is less efficient than in batch synchronous Mt-KaHyPar, requiring additional memory for $(k - 1) \cdot |V|$ benefit entries and more expensive updates per move and uncontraction. As in other versions of Mt-KaHyPar [18, 17] and sequential KaHyPar [1], the advantage of finding the best possible move for a node in $\mathcal{O}(k)$ still outweighs the disadvantages of extra memory and time for gain cache updates. In preliminary experiments, attempts to compute the gains on-the-fly in asynchronous uncoarsening have consistently been outperformed by using the gain cache. This may be explained by refinement algorithms like Fiduccia-Mattheyses evaluating and re-evaluating best possible moves often and for all nodes in the scope of the refinement, not only those that end up being moved. This requires fast access to move gains when using such refinement algorithms.

6. Cross-Dependencies in Asynchronous Refinement

The notion of asynchronous uncoarsening introduces cross-dependencies between nodes being concurrently uncontracted and moved from one block to another during refinement. In this section, we will examine these dependencies and our measures against interference in more detail. First, we consider how to ensure correct refinement by prohibiting moves of nodes that are in an intermediate inconsistent state due to concurrent uncontractions. Then, we describe how changes to the connectivity information Λ and Φ can lead to incorrect concurrent gain cache updates. Afterwards, we explain how the correctness of gain cache updates is, additionally, afflicted by concurrent changes to hyperedge pin lists. Further, we argue how the effectiveness of localized FM searches may be reduced in the asynchronous uncoarsening context. Finally, we discuss why these dependencies do not occur in batch synchronous Mt-KaHyPar.

6.1. Intermediate Inconsistent States Caused By Uncontractions

In asynchronous uncoarsening, we would like to allow any worker thread to move any node at any time during localized refinement. However, as explained in Section 4.1, we have to disallow moving nodes contained in an uncontraction group while that uncontraction group is being uncontracted. In this section, we will quickly describe how we enforce this in asynchronous uncoarsening.

Consider an uncontraction group $g = \{(u, v), (u, w)\}$. Assume a thread t_1 is currently uncontracting g and has already performed the uncontraction (u, v) but not (u, w). Then, a different thread $t_2 \neq t_1$ may not move the nodes v or u as v and w were contracted onto u at the same time. As stated in Section 4.1, Gottesbüren et al. [17] explain that u and v are in an inconsistent intermediate state. Refinement algorithms on these nodes would then yield incorrect results. Thus, we must prohibit moving the representative and the already uncontracted nodes in this case which we accomplish as follows.
Firstly, we prevent moving representatives during uncontractions using one spin lock per node. We require threads to acquire the lock on the representative for each uncontraction and to acquire the lock on the moved node for each move. In our example, t_1 acquires the lock on u before the uncontraction of g and releases the lock once all uncontractions in g are finished. Additionally, we require t_2 to acquire the lock when trying to move u which

Algorithm 6.1:
Uncontraction w/o Snapshots

Input: $u \in V, v \in V, e \in I(u)$
1 *acquire lock on e*
2 *update pin list of* e *and* $\Phi(e, \Pi(u))$
3 *update gain cache using* $\Lambda(e)$, $\Phi(e, \cdot)$ *and* e
4 *release lock on e*

Algorithm 6.2:
Uncontraction w/ Snapshots

Input: $u \in V, v \in V, e \in I(u)$
1 *acquire lock on e*
2 *update pin list of* e *and* $\Phi(e, \Pi(u))$
3 $\Lambda_{snap} \leftarrow \Lambda(e)$
4 $\forall i \in [k] : \Phi_{snap}(i) \leftarrow \Phi(e, i)$
5 $e_{snap} \leftarrow e$
6 *release lock on e*
7 *update gain cache using* Λ_{snap}, Φ_{snap} *and* e_{snap}

fails as t_1 is holding on to the lock.
Secondly, we need to prevent moving uncontracted nodes until all uncontractions in their group are finished. In the above case, we do not want to let t_2 move v until t_1 completes the uncontraction (u, w). For that purpose, recall that a node is considered active if it is not currently contracted onto another node. Only active nodes are allowed to be moved. We delay the activation of sibling nodes in the same group until after all uncontractions in the group are finished. In the given example, t_1 activates both v and w when it is done uncontracting g, allowing t_2 to move the uncontracted nodes afterwards.

6.2. Changes to Connectivity Information Affecting Gain Cache Updates

Assume a thread t_1 is currently performing an uncontraction (u, v). For each hyperedge $e \in I(u)$, t_1 needs to update the pin list of e as well as $\Phi(e, \Pi(u))$ accordingly and perform an associated gain cache update as described in Section 5.1. Consider now a specific hyperedge $e \in I(u)$. As described in Section 5.1, t_1 may require the values of $\Lambda(e)$ and $\Phi(e, i)$ with $i \in \Lambda(e)$ for the gain cache update. For an accurate gain cache update, these values need to represent the partition at the moment when t_1 performs the synchronized update to the pin list of e. In synchronous Mt-KaHyPar [18], t_1 achieves this by performing the gain cache update synchronously with the changes to the pin list of e, thus preventing intermittent changes to $\Lambda(e)$ and $\Phi(e, \cdot)$. Synchronous Mt-KaHyPar synchronizes these steps using one lock per hyperedge, as explained in Section 3.4.1. We illustrate this approach in Algorithm 6.1: First, t_1 acquires the lock for e (line 1) and then updates the pin list of e and $\Phi(e, \Pi(u))$ if necessary (line 2). Then, t_1 performs the gain cache update (line 3) before releasing the lock on e (line 4). That way, t_1 can safely use the global values of $\Lambda(e)$ and $\Phi(e, \cdot)$ because the values could not have been changed since t_1 had updated the pin list of e.

In asynchronous uncoarsening, the gain cache update for an uncontraction is more expensive than in synchronous Mt-KaHyPar (see Section 5.1). Consequently, performing the gain cache update while holding the lock on e possibly introduces lock contention, especially for large hyperedges. Therefore, in our asynchronous approach, t_1 performs the gain cache update after releasing the lock on e to reduce lock contention.
Then, any concurrent thread t_2 may perform a move (w, s, t) after t_1 releases the lock on e but before t_1 updates the gain cache. With the move of w, t_2 may change the globally stored values of $\Lambda(e')$, $\Phi(e', s)$ and $\Phi(e', t)$ for $e' \in I(w)$. Then, the global values of Λ and Φ represent the partition after the move of w. If $I(v) \cap I(w) \neq \emptyset$, t_2 may also have changed the values of $\Lambda(e)$ and $\Phi(e, i)$ with $i \in \Lambda(e)$ for a hyperedge $e \in I(v)$ which is relevant for the gain cache update performed by t_1. In that case, the global values of $\Lambda(e)$ and $\Phi(e, i)$

now differ from the partition state at the moment when t_1 updated the pin list of e for the uncontraction (u, v). Thus, if t_1 uses the global values for the gain cache update, it may update the gain cache incorrectly.

Therefore, we require t_1 to explicitly capture the state of $\Lambda(e)$ and $\Phi(e, i)$ for $i \in [k]$ as snapshots while still holding the lock on e. We present the approach using snapshots in Algorithm 6.2: As before, first t_1 acquires the lock on hyperedge e (line 1) and updates the pin list of e and $\Phi(e, \Pi(u))$ if necessary (line 2). Then, t_1 takes a snapshot Λ_{snap} by copying $\Lambda(e)$ (line 3) and a snapshot $\Phi_{snap}(i)$ of $\Phi(e, i)$ for each $i \in [k]$ (line 4). Subsequently, t_1 releases the lock on e (line 6) and uses the snapshots Λ_{snap} and Φ_{snap} for a safe gain cache update afterwards (line 7). We also need to take a snapshot of the pin list of e in certain cases (line 5). This is explained in Section 6.3.

6.3. Changes to Pin Lists Affecting Gain Cache Updates

In Section 6.2 we explained how moves can change the partition and, thus, affect the gain cache updates for concurrent uncontractions. In this section we consider a similar dependency: Consider a thread t_1 performing a move $m := (u, s, t)$ and a hyperedge $e \in I(u)$. The thread synchronously updates the values of $\Phi(e, s)$ and $\Phi(e, t)$ for m and reads the resulting values for a possible gain cache update (see (5.2)). The access to $\Phi(e, \cdot)$ is synchronized with the same hyperedge locks used for the updates of pin lists and Φ in uncontractions (see Algorithm 6.1 and Algorithm 6.2). Much like the gain cache update for an uncontraction, we aim to have t_1 perform the gain cache update for m after releasing the lock on e. For this gain cache update caused by the changes to $\Phi(e, \cdot)$, t_1 needs to iterate through the pin list of e (see (5.2)).
However, a different thread t_2 may perform an uncontraction (u, v) of a pin $u \in e$ after t_1 released the lock on e but before t_1 performs the gain cache update. In the course of this uncontraction, t_2 changes the stored pin list of e, either adding v as a pin or replacing u by v as a pin. Then, the stored pin list of e represents e after the uncontraction. The pin list notably now differs from its state at the moment when t_1 synchronously updated and read the values of $\Phi(e, s)$ and $\Phi(e, t)$. Therefore, if t_1 uses the stored pin list for e for the gain cache update caused by m, it may update the gain cache incorrectly. More precisely, t_1 may change the gain cache entry of the node v which was not a pin of e at the time when t_1 applied m. Additionally, if u is no longer in the pin list of e, t_1 may miss a necessary update to the gain cache entry of u as it was a pin at the time the pin counts were read.

Again, we solve this issue with snapshots by requiring t_1 to explicitly capture the pin list of e while holding the lock on e. Then, t_1 can safely use the snapshot of the pin list of e taken at the time of updating $\Phi(e, s)$ and $\Phi(e, t)$ to update the gain cache after releasing the lock.

Stable and Volatile Pins. Whereas the size of snapshots of Λ and Φ described in Section 6.1 is in $\mathcal{O}(k)$, the size of pin list snapshots is only bounded by $\max_{e \in E} |e|$. This means taking a pin snapshot is asymptotically as slow as performing the gain cache update entirely synchronized. Thus, taking snapshots of pin lists may cause lock contention nevertheless. To decrease the number of pins that need to be copied for a pin list snapshot, we separate the pin lists of all hyperedges into stable and volatile pins:

Definition 6.1. *A pin $u \in e$ is stable if it can safely be assumed that u will not be replaced as a pin of e by another node due to an uncontraction.*

Thus, every node u for which there is no uncontraction $(u, \cdot)$ remaining is a stable pin in all its incident hyperedges. The uncontraction group forest defines which uncontraction (u, v)

is the last uncontraction from u. When that uncontraction (u, v) is finished, u becomes stable in all of its incident hyperedges. Leaves of the contraction forest are initially stable, i.e. stable in all incident hyperedges without any uncontractions.

We never have to take snapshots of the stable pins of any hyperedge for a move or uncontraction. We achieve this by keeping the stable pins at the beginning of the pin list of every hyperedge. Then, for a gain cache update associated with a hyperedge e, it suffices to store the number of stable pins c while holding the lock for the pin count update of e caused by the move or uncontraction. Those c pins of e will no longer change due to uncontractions. Therefore, at the time of the gain cache update, the first c pins of the pin list of e can be safely queried directly from the current stored pin list of e without synchronization or a snapshot.
We maintain the range of stable pins at the beginning of the pin list of each hyperedge as follows: Before the asynchronous uncoarsening phase, we iterate over each hyperedge e in parallel, identifying the root nodes that are initially stable pins for e. We, then, sort the initially stable pins for e that are also root nodes to the beginning of the pin list of e. When a pin becomes stable or an initially stable pin is activated when it is uncontracted, it is swapped to the end of the range of stable pins and the number of stable pins is atomically increased by one.

A pin u of e is called *volatile* if it may still be replaced by another node v due to an uncontraction (u, v) within the remaining uncontractions. We situate volatile active pins after the stable pins in the pin list of each hyperedge to restrict changes of the pin list to its tail. Volatile pins may still change due to uncontractions activating and replacing pins. Therefore, we still have to take snapshots of volatile active pins while holding the hyperedge lock.

Pin Snapshots for Uncontractions. The problem of changing pin lists can also occur for gain cache updates caused by uncontractions as we explain in the following. Consider a thread t_1 performing an uncontraction (u, v) and a hyperedge $e \in I(u)$. If the uncontraction (u, v) causes v to be added as a pin of e, t_1 may require the pin list of e for the associated gain cache update (see case (*ii.b.*) in (5.1)). As stated in Section 6.2, t_1 updates the gain cache after releasing the lock on e. Therefore, a different thread t_2 can perform a concurrent uncontraction that changes the stored pin list of e after t_1 releases the lock on e but before t_1 can execute the gain cache update for (u, v). That way, concurrent uncontractions can also impede with each others' gain cache updates. We solve this variant of the problem using pin list snapshots, too: When t_1 performs the changes to e caused by the uncontraction (u, v), the thread takes a snapshot of the volatile pins of e (Algorithm 6.2, line 5) while holding the lock. Subsequently, t_1 can use the pin list snapshot for a safe gain cache update after releasing the lock on e (Algorithm 6.2, line 7).

Pin Snapshot Remarks. Taking snapshots of pin lists reduces the work performed while holding the spin lock for a hyperedge in practice but it still demands iterating over the pins of a hyperedge twice: once to take the snapshot and once to perform the gain cache update after releasing the lock on the hyperedge. Moreover, the management of stable and volatile pins introduces some additional work during uncontractions. Due to these overheads, we observe that it is beneficial to perform gain cache updates while still holding the spin lock for small hyperedges instead of taking pin list snapshots and managing stable/volatile pins for those hyperedges. The minimum hyperedge size required for snapshots is a configuration parameter of our implementation. Our parameter tuning experiments (see Appendix A) showed that no snapshots should be taken for hyperedges smaller than 1000 pins to optimize running time.

Also, for a hyperedge e incident to a moved or uncontracted node, the $\Phi(e, \cdot)$ values decide whether the gain cache update will even use the pin list of e (see (5.1) and (5.2)). Importantly,

these $\Phi(e, \cdot)$ values are known before the pin list snapshot is taken. Consequently, we only take a pin list snapshot if it is necessary for the gain cache update. Furthermore, in the case of an uncontraction (u, v), we only need to take a pin list snapshot of $e \in I(u)$ if v is added as a pin to e. Otherwise, if v replaces u as a pin of e, the gain cache update does not need to iterate through the pin list of e at all (see case (i.) in (5.1)).

6.4. Effectiveness of Localized FM Refinement in Asynchronous Uncoarsening

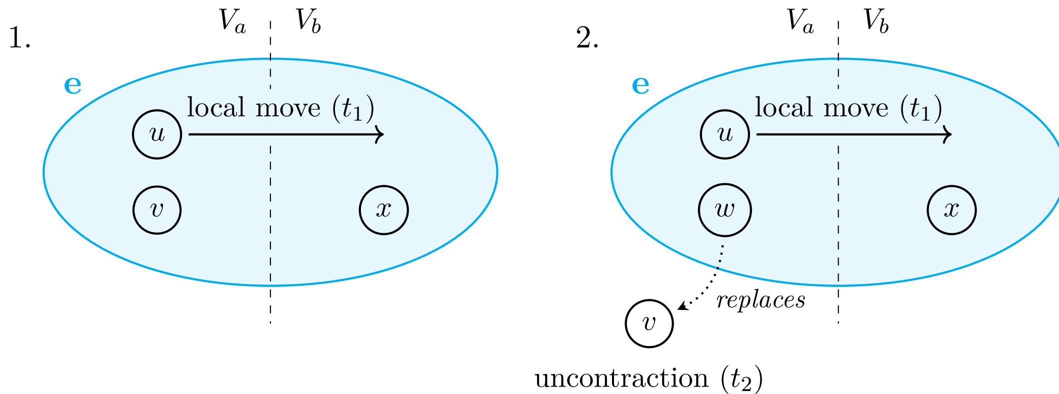

Figure 6.1.: An uncontraction invalidates gain deltas of a concurrent localized FM search: Left: t_1 locally moves pin u to V_b so v gains a benefit delta of $\omega(e)$. Right: t_2 uncontracts (v, w); w replaces v as pin of e. Deltas of v, w incorrect.

So far, we have considered the effects of asynchronous uncoarsening on uncontractions and global moves. A localized FM search, though, relies not only on correctness of the global partition but also on correct tracking of the changes that the FM search has made only locally. In particular, the changes to the gains of neighbors of locally moved nodes are stored as deltas to the global partition in hash tables kept in thread-local storage (see Section 3.2.2). Although we disallow global moves and uncontractions of locally moved nodes, concurrent threads may still change the global partition *around* the nodes that an FM search has already moved locally. These changes may interfere with the localized FM search. The authors of Mt-KaHyPar already identify this as a problem for synchronous Mt-KaHyPar where concurrent localized FM searches can interfere with each other in that way [17, sec. 5.4]. However, in asynchronous uncoarsening, this issue is exacerbated by concurrent uncontractions and LP searches that change the global partition more frequently than other localized FM searches. In this section, we explore the effects of changes of the global partition in the vicinity of a localized FM search on that FM search. Furthermore, we explain how synchronous Mt-KaHyPar deals with the problem and how we counteract the interference in asynchronous uncoarsening.

Global Changes Affecting Local FM Information. To start with, we describe how a localized FM search is affected by concurrent global changes in its proximity. Concurrent changes to the global partition may cause two issues for the FM search:
Firstly, the overall gain of the current local move sequence may change. This could diminish the efforts of the FM search and result in a move sequence that worsens the solution quality if applied to the global partition. However, using attributed gains, at least bad global moves can be prevented.
The second, more important problem is that concurrent changes to the surrounding partition irreparably invalidate the gain cache deltas of a localized FM search. Thus, the FM search may choose to perform local moves that are actually bad for the global partition. More

pressingly, though, the FM search may miss moves that would make a good move sequence because of invalidated gain deltas.

As an example for invalidated gain cache deltas, consider the situation depicted in Figure 6.1. The left picture represents the initial situation: Pins $u, v \in e$ are both in block V_a and thread t_1 has locally moved u to V_b as part of a localized FM search. The local move of u causes a delta gain cache update in which v gains a benefit delta of $\omega(e)$ as it is now the only pin of e in block V_a in the local partition state. The right picture depicts the situation after a concurrent thread t_2 has performed an uncontraction (v, w) that has replaced v with w as a pin of e. Then, the benefit deltas for v and w are wrong: v has an additional benefit delta of $\omega(e)$ that it should not have and w is missing that delta. Similar situations occur if t_2 performs a concurrent node move or an uncontraction in which w is added as a pin to e instead of replacing v.

Consequently, with many close concurrent changes of the partition, the gain deltas of a localized FM search may diverge from the deltas that would be correct for the changed global partition. In that case, the FM search uses its incorrect deltas to compute gains that do not represent the gains of the moves when applied to the global partition.

Effects on Synchronous Mt-KaHyPar. We will now quickly explain how synchronous Mt-KaHyPar handles this interference pattern. Synchronous Mt-KaHyPar uses attributed gains when writing FM moves to the global partition. That way, the partitioner ensures not to globally apply moves that worsen the solution quality. Additionally, *all* localized FM searches in synchronous Mt-KaHyPar globally move each node at most once until the next synchronization point. Therefore, the synchronous partitioner is not affected by invalidated gain cache deltas for nodes that have already been moved. Most importantly, in synchronous Mt-KaHyPar, only a localized FM search can cause global changes that are concurrent with another localized FM search. When FM refinement is applied in synchronous Mt-KaHyPar, preliminary LP refinement has already performed simple improvements, though. Thus, we do not expect a lot of FM searches to find further, more difficult improvements. This means that only few global changes occur during FM refinement. In conclusion, the effect of the interference in synchronous Mt-KaHyPar is small and no further countermeasures are needed.

Locality Sensitive Asynchronous Uncoarsening. In asynchronous uncoarsening, we execute LP refinement and uncontractions concurrently to localized FM searches. Thus, we observe that changes to the global partition are more frequent than in synchronous Mt-KaHyPar. Therefore, in the asynchronous case, interference with localized FM searches is a serious problem for the effectiveness of parallel FM refinement. We cannot feasibly solve the issue by keeping the deltas of all concurrent FM searches up to date for every global change. To achieve this, when working with t threads, we would have to update up to $t - 1$ delta gain caches in addition to the global gain cache for every global move or uncontraction.
Our solution instead revolves around reducing the number of changes of the global partition in the proximity of localized FM searches. Our goal is to have different threads perform uncontractions (and subsequent refinement based on those uncontractions) in different regions of the hypergraph.

For that purpose, we give a more precise definition of the region that a thread is working in. As we treat uncontraction groups as atomic units of work, a thread can be working on one or more uncontraction groups at a time. The groups that a thread is currently working on then define the active region of the thread. Therefore, first, we define which groups a thread is working on. Assume there are p worker threads partaking in asynchronous uncoarsening, each defined by a unique identifier in $T = \{1, \ldots, p\}$.

Definition 6.2. *We denote the set of groups a thread $t \in T$ is (currently) working on with $A(t)$. Then, t is working on a group g, i.e. $g \in A(t)$, if one of the following conditions applies:*

1. *t is currently uncontracting g.*
2. *t has already uncontracted g and has extracted at least one border node from g as a refinement seed but t has not yet started a localized refinement search using that seed (because the minimum number of refinement seeds has not been reached).*
3. *t is currently performing localized refinement based on a set of seeds that includes at least one seed extracted from g.*

We then define the region of a group and the active region of a thread as follows:

Definition 6.3. *The region $I(g)$ of an uncontraction group g is the set of hyperedges $I(u)$ incident to the representative u of g. The active region $I(t)$ of a thread $t \in T$ is the union of the regions of all groups that t is currently working on, i.e. $I(t) = \bigcup_{g \in A(t)} I(g)$.*

Now, consider a thread $t \in T$ that needs to pick a new uncontraction group to work on from the PQ of eligible groups. We strive to achieve the following behavior: t always picks an eligible group g s.t. the similarity between the region of g and the active regions of all other threads $T \setminus \{t\}$ is minimized. As a coefficient for the similarity, we use the Jaccard-Index J between the region of g and the union of the active regions of other threads:

Definition 6.4. *Let $\tilde{I}(t) := \bigcup_{t' \in T \setminus \{t\}} I(t')$ for $t \in T$. We define the similarity $sim(t, g)$ between the region of a group g and the active regions of all threads except $t \in T$ as:*

$$sim(t, g) := J(I(g), \tilde{I}(t)) = \frac{|I(g) \cap \tilde{I}(t)|}{|I(g) \cup \tilde{I}(t)|}$$

The thread t proceeds as follows to pick a new group to work on: First, t picks a group g from the PQ of eligible groups. Afterwards, t computes $sim(t, g)$. Then, the similarity is compared against a maximum permissible similarity s_{max}. If $sim(t, g) \leq s_{max}$, t will begin working on g by uncontracting it. If $sim(t, g) > s_{max}$, t picks a new group and checks the similarity for that one. After retrying and failing for a set number of groups, t starts uncontracting the group with the smallest similarity among the ones examined. When t starts working on a group, all other examined groups are reinserted into the PQ of eligible groups. The number of retries and the maximum similarity s_{max} are configuration parameters.

Computing Region Similarities. We compute $sim(t, g)$ for a thread t and a group g in one pass over $I(g)$. For that purpose, we need a data structure that represents the active region of every worker thread. More specifically, for each hyperedge $e \in E$ and each thread $t' \in T$ we need to be able to query if at the moment $e \in I(t')$. We accomplish this using a bit set containing one bit per worker thread for each hyperedge. The j-th bit in the bit set of hyperedge e serves as a flag indicating whether currently $e \in I(j)$ for a thread $j \in T$. We store the bit set for each hyperedge as one or more integer values. If p is small enough to store the entire bit set in one integer value, we can use atomic bit-logic operations to synchronize reads and writes to the flags. Otherwise, we use one spin lock per hyperedge to synchronize access. We store the bit sets of all hyperedges in a contiguous array to optimize cache efficiency.

With that data structure, t can easily compute $|I(g) \cap \tilde{I}(t)|$ for a group g in one pass over $I(g)$: For each hyperedge $e \in I(g)$, the thread t collectively checks whether $e \in I(t')$ for any $t' \in T \setminus \{t\}$ using synchronized bit-logic operations.

However, computing $|I(g) \cup \tilde{I}(t)|$ is more difficult. We aim to use

$$|I(g) \cup \tilde{I}(t)| = |I(g)| + |\tilde{I}(t)| - |I(g) \cap \tilde{I}(t)|$$

but we cannot easily track $|\tilde{I}(t)|$ for each $t \in T$. Therefore, we need to calculate $|\tilde{I}(t)|$ without explicitly knowing $\tilde{I}(t)$. We accomplish this as follows: Let $I_{all} := \bigcup_{t' \in T} I(t')$. Then:

$$\begin{aligned}
|\tilde{I}(t)| &= |(I_{all} \setminus I(t)) \cup (I(t) \cap \tilde{I}(t))| \\
&= |I_{all} \setminus I(t)| + |I(t) \cap \tilde{I}(t)| - |(I_{all} \setminus I(t)) \cap I(t) \cap \tilde{I}(t)| \\
&= |I_{all}| - |\underbrace{I_{all} \cap I(t)}_{=I(t)}| + |I(t) \cap \tilde{I}(t)| - |\underbrace{(I_{all} \setminus I(t)) \cap I(t) \cap \tilde{I}(t)}_{=\emptyset}| \\
&= |I_{all}| - |I(t)| + |I(t) \cap \tilde{I}(t)|
\end{aligned} \tag{6.1}$$

Thereby, to accurately determine $|\tilde{I}(t)|$, we keep track of $|I_{all}|$ as well as $|I(t)|$. Additionally, we explicitly compute $|I(t) \cap \tilde{I}(t)|$ by iterating over $I(t)$ much like we do for $I(g)$. When t picks a new group to work on, it has to calculate $|\tilde{I}(t)|$ in this way only once for all groups it examines.

Limitations. Our approach of locality sensitive asynchronous uncoarsening works reasonably well in improving the quality of local move sequences generated by asynchronous localized FM searches. We attribute this to the fact that our method diminishes the likelihood of different threads working on neighboring groups. However, our definition of the active region of a thread only approximates the area of the hypergraph that the thread affects. For refinement searches, only the hyperedges incident to the seeds are considered to be part of the active region of the thread that performs the search. Localized searches can expand, though. Thus, threads may still end up working on neighboring nodes even if the similarity according to our definition of active regions disallows it. This issue could be solved by dynamically expanding the active region of a thread as its local refinement search expands. However, refinement searches would then cause a lot of changes to the data structure that represents the active region of each thread. With our data structure described above, the time overhead for these changes is too large. Future approaches to locality sensitive asynchronous uncoarsening may use different definitions of an active region of a thread or a different mechanism for separating threads entirely. That way, these approaches may be able to better deal with expanding refinement searches.

In addition, our method of determining a group with permissible similarity to the active regions of other threads is arguably quite crude. By examining a set number of groups and only accepting one, we calculate a large number of similarity values that end up being simply discarded. It would be beneficial to have prior indication of how likely each eligible group is to be a good choice for a thread. Optimally, for each thread we would want a priority queue of eligible groups that orders the groups by increasing similarity between the region of the group and the active regions of other threads. As the active regions of all threads rapidly change, though, such an ordering by similarity would hardly ever be accurate for a long time.

6.5. Differences to Mt-KaHyPar

The dependencies described in Section 6.1 and Section 6.2 are inherently caused by moves and uncontractions being performed concurrently. As such, those problems can never occur in synchronous parallel partitioners like Mt-KaHyPar where phases of parallel uncontractions and phases of parallel refinement are isolated from one another.

Yet, Section 6.3 also deals with concurrent uncontractions interfering with each other without the presence of concurrent moves. However, the gain cache update for (u, v) is only performed after releasing the lock on e because gain cache updates in asynchronous uncoarsening are more expensive than in the synchronous partitioner. As stated in Section 6.2, synchronous Mt-KaHyPar simply performs the gain cache update while holding the lock on e. Thus, concurrent changes to pin lists do not pose a problem for synchronous Mt-KaHyPar and no mechanism like pin snapshots is necessary for correct gain cache updates.

As discussed in Section 6.4, global changes invalidating gain cache deltas in localized FM searches are not a central issue in synchronous Mt-KaHyPar. The authors of the synchronous partitioner observe that even without any methods to reduce the amount of interference itself the solution quality decreases only slightly with greater numbers of threads [17]. To further reduce the interference in synchronous Mt-KaHyPar, it may be interesting to construct batches with respect to node regions and perform locality sensitive parallel FM. Managing locality sensitive uncoarsening does come with a time overhead for region comparisons, though.

7. Experiments

We integrate our asynchronous uncoarsening in the hypergraph partitioning framework Mt-KaHyPar[1]. The framework is implemented in C++17, parallelized using the TBB library [36] and compiled using `g++` version 9.2 with the flags `-O3`, `-mtune=native` and `-march=native`. We refer to the synchronous $\mathcal{O}(log(n))$-level version of Mt-KaHyPar [18] as Mt-KaHyPar-D and the synchronous n-level version of Mt-KaHyPar [17] as Mt-KaHyPar-Q. Additionally, we refer to our asynchronous n-level version as Mt-KaHyPar-Async. For parallel partitioners we add a suffix to their name to indicate the number of threads used, e.g. Mt-KaHyPar-Async 64 for 64 threads. We omit the suffix for sequential partitioners. In the following, we give the values for the configuration parameters related to asynchronous uncoarsening that we used in our experiments. These values are based on our parameter tuning experiments which we describe in more detail in Appendix A. We take snapshots only for gain cache updates affecting hyperedges of at least 1000 pins (see Section 6.3). Furthermore, when choosing a group to uncontract, a thread only accepts a group with a region similarity of 0 to the active regions of other threads (see Section 6.4). A thread examines a maximum of 10 eligible groups to find a group with similarity 0 before resorting to accepting the group with the lowest similarity (see Section 6.4). Lastly, a thread collects a minimum of 5 seed nodes before applying localized refinement (see Section 4.2).

7.1. Instances

We run our experiments on the extensive collection of benchmark instances that Gottesbüren et al. [17] use to evaluate Mt-KaHyPar-Q and Mt-KaHyPar-D. The instances are derived from four sources encompassing three application domains: the ISPD98 VLSI Circuit Benchmark Suite [3], the DAC 2012 Routability-Driven Placement Contest [46], the SuiteSparse Matrix Collection [11] and the 2014 SAT Competition [5]. VLSI instances are converted into hypergraphs by transforming the netlist of each instance into a set of hyperedges. Sparse matrices are translated to hypergraphs using the row-net model [8]. SAT instances are transformed to three different hypergraph representations: *literal*, *primal* and *dual* (see Ref. [22] for details). All hypergraphs have unit node and hyperedge weights. The benchmark instances are separated into two sets: Firstly, set A, originally used by Heuer and Schlag [21], contains 488 hypergraphs. We use set A to compare our algorithm with other versions of Mt-KaHyPar and sequential partitioners. Secondly, set B, originally used in Ref. [18], contains 94 large hypergraph instances. We use set B to examine the scalability

[1]The Mt-KaHyPar framework is available at `https://github.com/kahypar/mt-kahypar`

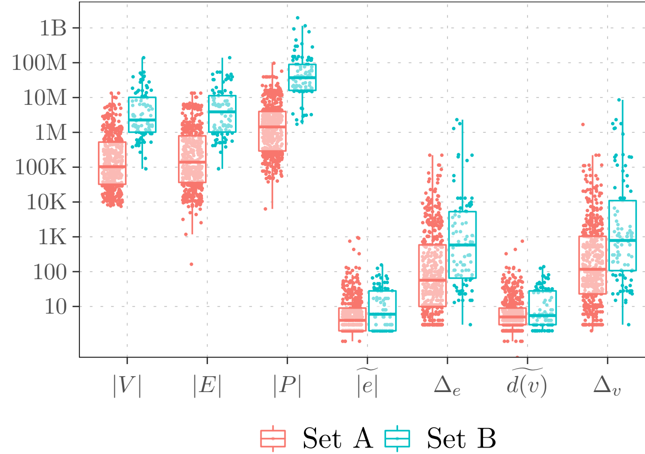

Figure 7.1.: Statistics for the hypergraph benchmark instances in set A and set B. The plot shows a dot for each hypergraph for the number of vertices $|V|$, hyperedges $|E|$ and pins $|P|$ as well as the median and maximum hyperedge size ($\widetilde{|e|}$ and Δ_e) and vertex degree ($\widetilde{d(v)}$ and Δ_v).

of our algorithm and to compare our algorithm against other parallel partitioners. The hypergraphs in set B are an order of magnitude larger on average than those in set A. We give an overview of important hypergraph metrics for set A and set B in Figure 7.1.

7.2. Algorithms

On set A, we compare Mt-KaHyPar-Async with Mt-KaHyPar-Q and Mt-KaHyPar-D as well as the sequential partitioners KaHyPar-CA [21], KaHyPar-HFC [16], PaToH-D and PaToH-Q [8]. We do not consider the sequential partitioner hMetis [26] as KaHyPar-CA outperforms hMetis regarding solution quality while also being faster on average [21].

On set B we compare Mt-KaHyPar-Async with Mt-KaHyPar-Q and Mt-KaHyPar-D. We do not consider the deterministic shared-memory parallel partitioner BiPart [31], the distributed-memory parallel partitioner Zoltan [12] or the sequential partitioners PaToH-D [8] and HYPE [32] as they have been shown to be dominated by Mt-KaHyPar-D on set B [17, 18]. Furthermore, we do not consider any other sequential partitioners (KaHyPar-CA, KaHyPar-HFC, PaToH-Q) on set B as they cannot partition the large hypergraphs in a reasonable time frame.

7.3. System and Methodology

We adapt our experimental setup from Gottesbüren et al. [17]. We run our experiments on two different types of machines.
Machines of type A are nodes of a cluster with Intel Xeon Gold 6230 processors (2 sockets with 20 cores each) running at 2.1 GHz with 96 GB RAM. We use machines of type A for the comparison with sequential partitioners on benchmark set A. For these experiments, we use $k \in \{2, 4, 8, 16, 32, 64, 128\}$, $\epsilon = 0.03$, ten different seeds and a time limit of eight hours. As in Ref. [17], we run Mt-KaHyPar-Async on set A with ten threads for the comparison with sequential partitioners. Ten threads are chosen by Gottesbüren et al. to resemble workload scenarios on commodity multi-core machines.
Machine B is an AMD EPYC Rome 7702P (1 socket with 64 cores) running at 2.0-3.35 GHz with 1024 GB RAM. For the comparison with other parallel partitioners, we run Mt-KaHyPar-Async 64 on benchmark set B using machine B with $k \in \{2, 8, 16, 64\}$,

$\epsilon = 0.03$, five seeds and a time limit of two hours. Additionally, we conduct experiments for the self-relative speedup of Mt-KaHyPar-Async on a subset of benchmark set B (82 out of 94 hypergraphs[2]) using machine B with $k \in \{2, 8, 16, 64\}$, $\epsilon = 0.03$, three seeds and $p \in \{1, 4, 16, 64\}$ threads.
All partitioners optimize the connectivity metric $(\lambda - 1)$ which we also refer to as the quality of the partition. We consider a partition Π to have better quality than a partition Π' of the same hypergraph if $(\lambda - 1)(\Pi) < (\lambda - 1)(\Pi')$. For each instance, i.e. a hypergraph with a desired number of blocks k, we aggregate running times using the arithmetic mean over all seeds. To further aggregate over multiple instances, we use the harmonic mean for relative speedups and the geometric mean for absolute running times. Runs with imbalanced partitions are not excluded from aggregated running times. For runs that exceed the time limit, we use the time limit itself as the running time in aggregates. In plots, we mark these instances with ⏲ if *all* runs of that algorithm timed out. Similarly, we use ✗ to mark instances for which all runs produced imbalanced partitions.
We compare the solution quality of different algorithms with *performance profiles* [13]: Let $\mathcal{A}$ be the set of all algorithms we want to compare, $\mathcal{I}$ the set of instances and $q_A(I)$ the average quality of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm $A \in \mathcal{A}$, we plot a line depicting the fraction of instances (y-axis) for which $q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)$ where τ is on the x-axis. Achieving higher fractions at lower τ-values is considered better. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm performs best. Note that these plots relate the quality of an algorithm to the best solution. Thus, they do not permit a full ranking of more than two algorithms.

Additionally, we perform Wilcoxon signed rank tests [47] to determine whether or not the differences between two partitioners with similar quality or running time are statistically significant. At a 1% significance level ($p \leq 0.01$), a Z-score of $|Z| > 2.576$ is considered significant [7, p. 180]. Higher p-values admit higher thresholds of $|Z|$.

7.4. Comparison with Other Algorithms

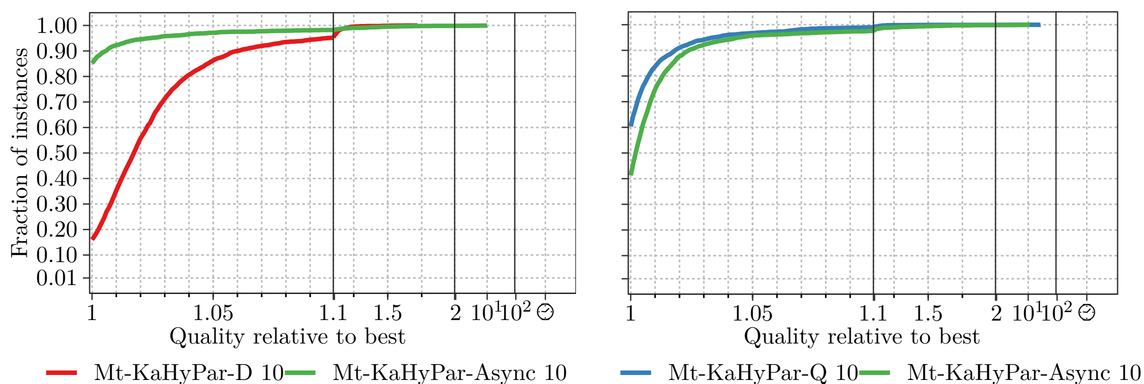

Figure 7.2.: Performance profiles comparing the solution quality of Mt-KaHyPar-Async 10 with Mt-KaHyPar-D 10 (left) and with Mt-KaHyPar-Q 10 (right) on set A.

Quality Comparison on Set A. Figure 7.2 shows two performance profile plots that directly compare the quality of Mt-KaHyPar-Async 10 with Mt-KaHyPar-D 10 as well as Mt-KaHyPar-Q 10 on set A. Mt-KaHyPar-Q finds better solutions than Mt-KaHyPar-Async but Mt-KaHyPar-Async finds better solutions than Mt-KaHyPar-D. A Wilcoxon signed rank test demonstrates that the difference in quality between Mt-KaHyPar-Async 10 and

[2] The subset contains all hypergraphs on which Mt-KaHyPar-Async 64 was able to finish in under 800 seconds for all $k \in \{2, 8, 16, 64\}$. This experiment still took more than 5 weeks on machine B.

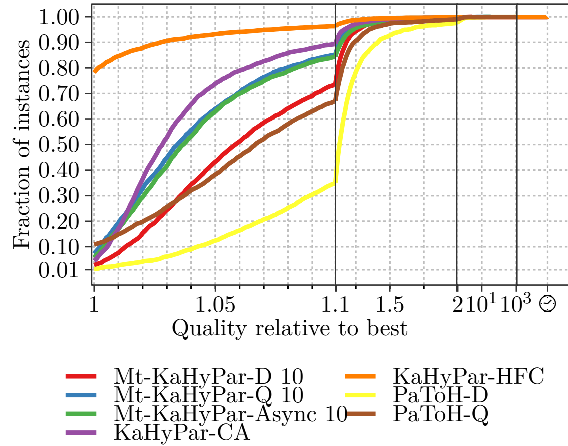

Figure 7.3.: Performance profile comparing the solution quality of Mt-KaHyPar-Async with the other evaluated sequential and parallel partitioners on set A.

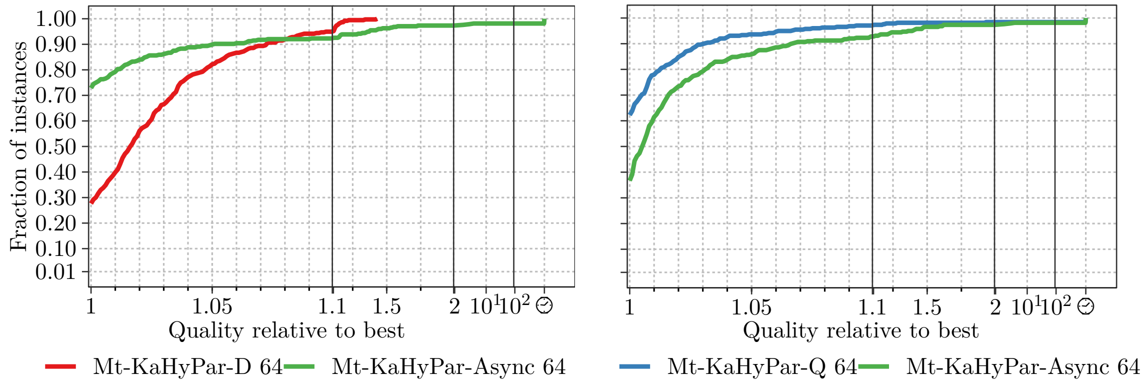

Figure 7.4.: Performance profiles comparing the solution quality of Mt-KaHyPar-Async 64 with Mt-KaHyPar-D 64 (left) and with Mt-KaHyPar-Q 64 (right) on set B.

Mt-KaHyPar-Q 10 is statistically significant ($Z = 13.42$ with $p < 2.2e - 16$). The plot in Figure 7.3 depicts a performance profile comparing the quality of Mt-KaHyPar-Async 10 with both other Mt-KaHyPar versions as well as the sequential partitioners on set A. Due to the additional flow-based refinement, KaHyPar-HFC greatly outperforms KaHyPar-CA, Mt-KaHyPar-Q 10 and Mt-KaHyPar-Async 10 in terms of solution quality. In turn, those three partitioners find better solutions than Mt-KaHyPar-D 10 and both PaToH variants. In an individual comparison, Mt-KaHyPar-Async 10 finds better partitions than PaToH-D, Mt-KaHyPar-D 10, PaToH-Q, Mt-KaHyPar-Q 10, KaHyPar-CA and KaHyPar-HFC on 86.39%, 83.96%, 68.79%, 39.67%, 35.74% and 11.01% of instances in set A, respectively.

Quality Comparison on Set B. In Figure 7.4, we depict two performance profile plots comparing Mt-KaHyPar-D 64 and Mt-KaHyPar-Q 64 with Mt-KaHyPar-Async 64 on set B. Similarly to set A, Mt-KaHyPar-Q 64 produces partitions with better quality than Mt-KaHyPar-Async 64 on set B while Mt-KaHyPar-Async 64, in turn, finds better solutions than Mt-KaHyPar-D 64. According to Wilcoxon signed rank tests, these differences are statistically significant ($Z = 5.83$ with $p = 5.59e - 9$ and $Z = -7.23$ with $p = 4.69e - 13$, respectively). In an individual comparison, Mt-KaHyPar-Async 64 finds better partitions than Mt-KaHyPar-D 64 and Mt-KaHyPar-Q 64 on 72.34% and 36.76% of instances in set B, respectively[3].

[3]These percentages exclude any instances for which both partitioners timed out in all runs.

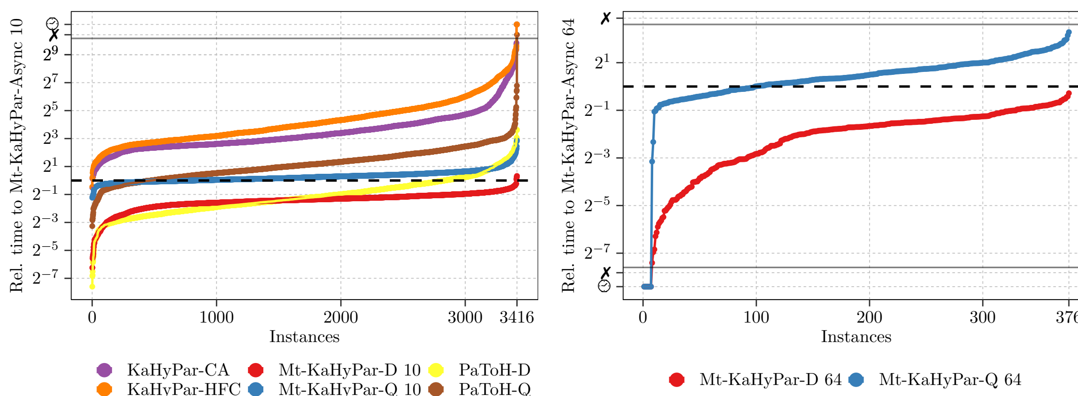

Figure 7.5.: Running times of all evaluated algorithms relative to Mt-KaHyPar-Async 10 on set A (left) and to Mt-KaHyPar-Async 64 on set B (right). The ✗ and ⏲ symbols mark instances for which Mt-KaHyPar-Async (bottom) or a different partitioner (top) produced imbalanced partitions or timed out.

Set A		Set B	
Partitioner	t[s]	Partitioner	t[s]
Mt-KaHyPar-D 10	0.95	Mt-KaHyPar-D 64	4.89
PaToH-D	1.17	Zoltan 64	12.63
Mt-KaHyPar-Async 10	2.66	Mt-KaHyPar-Async 64	22.60
Mt-KaHyPar-Q 10	3.19	HYPE	25.56
PaToH-Q	5.86	BiPart 64	29.19
KaHyPar-CA	28.14	Mt-KaHyPar-Q 64	30.70
KaHyPar-HFC	48.98	PaToH-D	51.20

Table 7.1.: Geometric mean running times on set A (left) and set B (right).

Running Times. In Figure 7.5, we give two plots comparing the running times of other algorithms relative to Mt-KaHyPar-Async 10 on set A (left) and to Mt-KaHyPar-Async 64 on set B (right).
On set A, Mt-KaHyPar-Async 10 is faster than the sequential partitioners KaHyPar-HFC, KaHyPar-CA and PaToH-Q, and slightly faster than Mt-KaHyPar-Q 10. Furthermore, Mt-KaHyPar-Async 10 is slower than Mt-KaHyPar-D 10 and PaToH-D on set A.
On set B, Mt-KaHyPar-Async 64 is faster than Mt-KaHyPar-Q 64 and slower than Mt-KaHyPar-D 64.
In Table 7.1, we additionally report the geometric mean running times over all instances for each algorithm on set A (left) and set B (right).

7.5. Scalability

Self-Relative Speedups. In Figure 7.6, we present plots describing the self-relative speedups of Mt-KaHyPar-Async on the 82 hypergraph subset of set B (see Section 7.3) with $p \in \{4, 16, 64\}$ threads. We separately consider total running time, the initial partitioning phase and the refinement phase. For an instance with a sequential running time x, the plot contains a dot at point (x, y) with the speedup y for each evaluated number of threads p. The line represents the cumulative harmonic mean speedup over all instances with a single-threaded running time $\geq x$.
Additionally, Table 7.2 shows the harmonic mean speed ups for each of the considered phases. The overall harmonic mean speedup for Mt-KaHyPar-Async is 3.69 for $p = 4$,

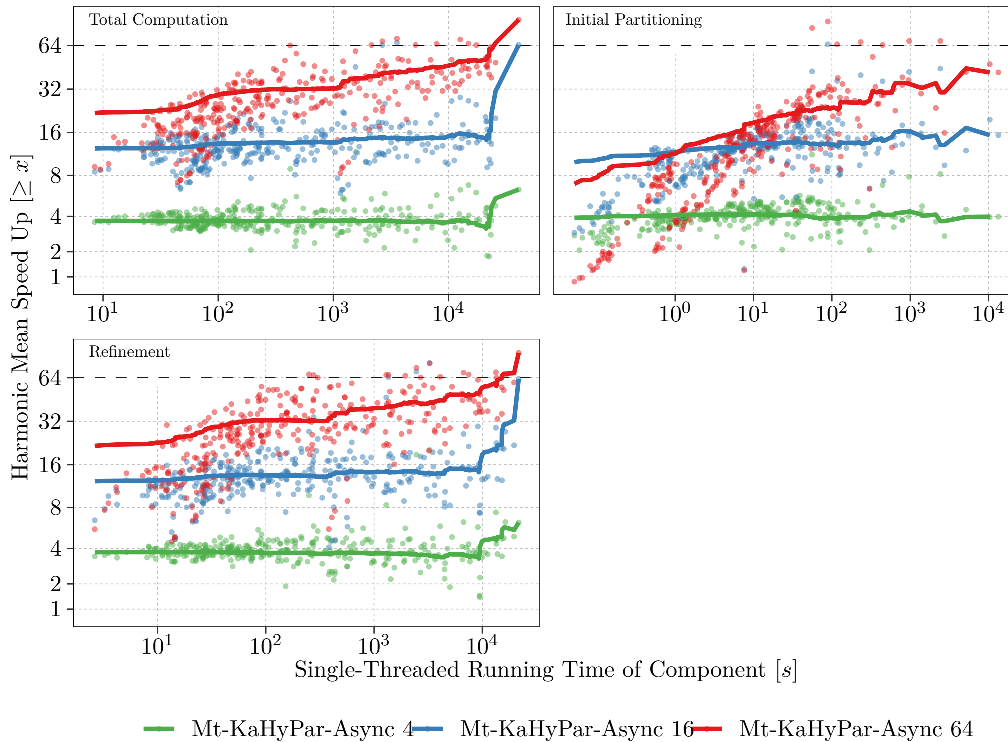

Figure 7.6.: Self-relative speedups for the components of Mt-KaHyPar-Async affected by asynchronous uncoarsening on the 82 hypergraph subset of set B.

Num. Threads		T	IP	R
	4	3.69	3.90	3.75
All	16	12.38	9.92	12.28
	64	21.81	6.96	21.60
	4	3.69	3.87	3.71
$\geq 100s$	16	13.37	13.18	13.48
	64	29.14	23.77	32.51
	4	3.69	4.15	3.66
$\geq 1000s$	16	13.63	15.24	14.23
	64	32.32	32.24	39.46
Inst. $\geq 100s$ [%]		62.20	12.80	50.30
Inst. $\geq 1000s$ [%]		28.66	2.44	23.17

Table 7.2.: Harmonic mean speedups of Mt-KaHyPar-Async on our 82 hypergraph subset of set B for different components: Total Computation (T), Initial Partitioning (IP) and Refinement (R). The row marked $\geq 100s$ ($\geq 1000s$) shows the speedups for instances on which the component has a single-threaded running time greater than 100 (1000) seconds. The last two rows show the percentage of instances for which each component took longer than those thresholds using one thread.

12.38 for $p = 16$ and 21.81 for $p = 64$. If we only consider instances with a single-threaded running time $\geq 100s$ or $\geq 1000s$, we observe harmonic mean speedups of 29.14 and 32.32, respectively, for $p = 64$.

Our main goal with asynchronous uncoarsening is to improve the scalability of the refinement phase. The refinement phase makes up a significant part of the total running time: With one thread, refinement accounts for more than a third of the running time in 91.46%

of the instances and for more than half of the running time in 62.80% of the instances. Asynchronous refinement scales slightly worse than the total computation on all instances (speedup of 21.60 compared to 21.81 for $p = 64$). However, on instances with single-threaded running times $\geq 100s$ and $\geq 1000s$, refinement scales better than the total computation (speedups of 32.51 and 39.46 compared to 29.14 and 32.32, respectively, for $p = 64$).

The initial partitioning phase scales worse than the refinement phase but mostly does not contribute a large part of the total running time. The single-threaded running time for this component exceeds 100 seconds in only 12.80% of instances. Similar values have been observed for Mt-KaHyPar-Q [17].

As in Mt-KaHyPar-Q, we achieve some super-linear speedups. This can be attributed to the non-determinism of our partitioner. In the coarsening and refinement phases, the non-deterministic choice and order of contractions and uncontractions contribute to extreme outliers in speedups.

		M-Async		M-Q	
	Num. Threads	T	R	T	R
All	4	3.71	3.80	3.71	3.86
	16	12.28	12.24	11.73	11.79
	64	21.03	20.87	22.60	23.08
$\geq 100s$	4	3.73	3.80	3.74	3.84
	16	13.31	13.59	12.37	12.29
	64	28.06	31.62	25.30	24.85
$\geq 1000s$	4	3.78	3.90	3.78	3.86
	16	13.54	14.99	12.24	11.85
	64	30.13	39.72	25.07	23.69
	Inst. $\geq 100s$ [%]	59.21	46.38		
	Inst. $\geq 1000s$ [%]	23.02	17.11		

Table 7.3.: Harmonic mean speedups of Mt-KaHyPar-Async ("M-Async") and Mt-KaHyPar-Q [17] ("M-Q") on the shared 76 hypergraph subset of set B for the total computation (T) and the refinement phase (R). The row marked $\geq 100s$ ($\geq 1000s$) shows the speedups for instances on which the component of Mt-KaHyPar-Async 1 takes at least 100 (1000) seconds. The last two rows show the percentage of instances for which each component of Mt-KaHyPar-Async 1 took longer than those thresholds.

Comparison with Speedups of Mt-KaHyPar-Q. For a comparison with the speedups of Mt-KaHyPar-Q we use the results of the authors' original experiments [17]. We used the same method as the authors of Mt-KaHyPar-Q to derive a suitable subset of set B for our scalability experiments but ended up with a different subset (82 vs. 77 hypergraphs with 76 hypergraphs contained in both subsets). Therefore, for a fairer comparison, in Table 7.3 we report the speedups for Mt-KaHyPar-Async and Mt-KaHyPar-Q on only the shared 76 hypergraph subset[4]. We omit the initial partitioning phase here.

Mt-KaHyPar-Q scales slightly better overall with a harmonic mean speedup for all instances of 22.60 with $p = 64$ compared to 21.03 for Mt-KaHyPar-Async. However, considering only instances with a large single-threaded running time, Mt-KaHyPar-Q scales worse than Mt-KaHyPar-Async: Mt-KaHyPar-Q achieves a harmonic mean speedup of 25.30 (25.07) with $p = 64$ compared to 28.06 (30.13) for Mt-KaHyPar-Async for instances on which Mt-KaHyPar-Async takes ≥ 100 (1000) seconds with one thread.

[4]Note that the numbers given in Tables 7.2 and 7.3 differ as they are based on different subsets.

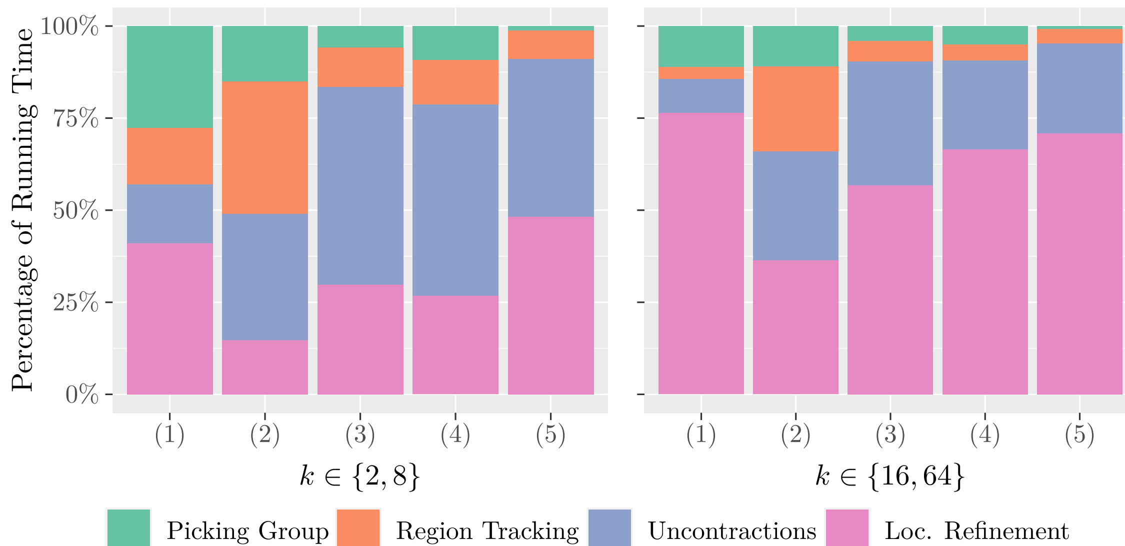

Figure 7.7.: Fraction of running time that each component of asynchronous uncoarsening takes for Mt-KaHyPar-Async 64 on five different hypergraphs (see Table 7.4 for hypergraph metrics). Arithmetic mean for 5 seeds and $k \in \{2, 8\}$ (left) and for 5 seeds and $k \in \{16, 64\}$ (right).

These differences are exacerbated if we consider only the refinement phase: The harmonic mean speedup of the refinement phase of Mt-KaHyPar-Q decreases for longer running instances from 24.85 for instances on which Mt-KaHyPar-Async 1 takes ≥ 100 to 23.69 for instances on which Mt-KaHyPar-Async 1 takes ≥ 1000 seconds. Conversely, the mean speedup of the refinement phase of Mt-KaHyPar-Async increases with larger instances from 31.62 for instances on which Mt-KaHyPar-Async 1 takes ≥ 100 to 39.72 for instances on which Mt-KaHyPar-Async 1 takes ≥ 1000 seconds.

Speedups for Short Running Instances. Mt-KaHyPar-Async scales better than Mt-KaHyPar-Q on instances with long single-threaded running times but on all instances the average speedups of Mt-KaHyPar-Async are worse than those of Mt-KaHyPar-Q. In particular, considering only instances with a single-threaded running time of *less* than 100 seconds, we observe that Mt-KaHyPar-Async only reaches an average speedup of 16.13 with 64 threads compared to 21.73 for Mt-KaHyPar-Q. We believe that these bad speedups for short running instances are related to time overheads caused by our approach to locality sensitive uncoarsening (see Section 6.4). Therefore, in the following, we analyze these time overheads in more detail.

In Figure 7.7, we show the running times of each individual component of asynchronous uncoarsening in Mt-KaHyPar-Async 64 for five example hypergraphs (for hypergraph metrics see Table 7.4)[5]. The “Picking Group” component entails examining uncontraction groups with regard to the similarity of their region to the active regions of other threads (see Section 6.4). The component “Region Tracking” describes the work that a thread performs to update its active region in the shared bit set data structure described in Section 6.4.

The bar charts show that these measures needed for locality sensitive uncoarsening contribute a considerable fraction of the total running time with 64 threads for most hypergraphs, particularly ones with a small total partitioning time. We observe that the

[5] We collected the times per component on each individual worker thread and then accumulated the times of all worker threads for an instance.

	Type	$\|V\|$	$\|E\|$	$\|P\|$	$\widetilde{\|e\|}$	Δ_e	$\widetilde{d(v)}$	Δ_v	t_1	t_2
(1)	DAC [46]	1.3M	1.3M	4.8M	2	8.5K	4	1.0K	4	8
(2)	SPM [11]	9.8M	6.9M	57.2M	5	3.8K	2	25.7K	25	33
(3)	Primal [5]	9.6M	46.6M	142.5M	3	1.1K	6	197.8K	308	361
(4)	Literal [5]	191.6M	46.6M	142.5M	3	1.1K	3	193.8K	376	530
(5)	Dual [5]	46.6M	9.6M	142.5M	6	197.8K	3	1.1K	673	1119

Table 7.4.: Table with metrics of the hypergraphs used for tracking running times for each component of asynchronous uncoarsening (see Figure 7.7). We state the type of the hypergraph, the number of vertices $|V|$, hyperedges $|E|$ and pins $|P|$ as well as the median and maximum hyperedge size ($\widetilde{|e|}$ and Δ_e) and vertex degree ($\widetilde{d(v)}$ and Δ_v). The last two columns give the average total partitioning time for $k \in \{2, 8\}$ (t_1) and for $k \in \{16, 64\}$ (t_2) across 5 seeds in seconds.

fraction of the total running time that is put into picking groups and tracking active regions decreases with a longer total running time caused by both larger hypergraphs and larger k values on the same hypergraph (for the partitioning times see Table 7.4). This implies that locality sensitive uncoarsening constitutes a major time overhead but also that this overhead does not grow proportionally to the total running time for long running instances. Thus, we deem this time overhead a reason for the discrepancy between the scalability of Mt-KaHyPar-Async on instances with small single-threaded running times and on instances with large single-threaded running times.

Based on these results, we reconsidered the definition of the uncontraction groups that a thread is working on (see Section 6.4). We found that a lot of time for tracking active regions is spent on groups that contain no boundary vertices, i.e. no refinement seeds. For these groups, a thread marks the hyperedges in the region of the group as active in the shared bit set data structure before uncontracting the group and then as inactive again right after uncontracting the group. At the same time, considering a group g active for a thread t while t is uncontracting g does not serve the purpose of preventing interference with refinement searches. Consequently, t should not mark the hyperedges of g as active before the uncontraction. Instead, t can mark the hyperedges as active after the uncontraction if g contains at least one refinement seed.

In experiments on our five example graphs, we observed that this measure significantly reduces the time overhead for locality sensitive uncoarsening. In Table 7.5, we show the reduction in running time for the "Picking Group" (PG) and "Region Tracking" (RT) components for our five example hypergraphs with $k \in \{2, 8\}$ and $k \in \{16, 64\}$[6]. On average across all five hypergraphs, the times for these components decreased by 32.41% and 45.65%, respectively, for $k \in \{2.8\}$ and by 20.42% and 28.93%, respectively, for $k \in \{16, 64\}$. Particularly for instances with a small overall running time, we reduced the fraction of the running time required for the relevant components by up to 83%. Furthermore, instances with smaller k values are affected stronger. We believe this is due to the smaller number of boundary vertices and consequently larger

	$k \in \{2, 8\}$		$k \in \{16, 64\}$	
	PG	RT	PG	RT
(1)	43.99	83.74	10.12	40.29
(2)	55.02	82.02	46.36	77.11
(3)	14.97	26.20	4.29	5.73
(4)	17.21	22.31	6.76	15.16
(5)	30.89	14.00	34.55	6.35
All	32.41	45.65	20.42	28.93

Table 7.5.: Improvements to component times in percent.

[6]For a visual representation of the improvements, we show the bar plots of the component times after the optimization in Figure B.1 in the appendix.

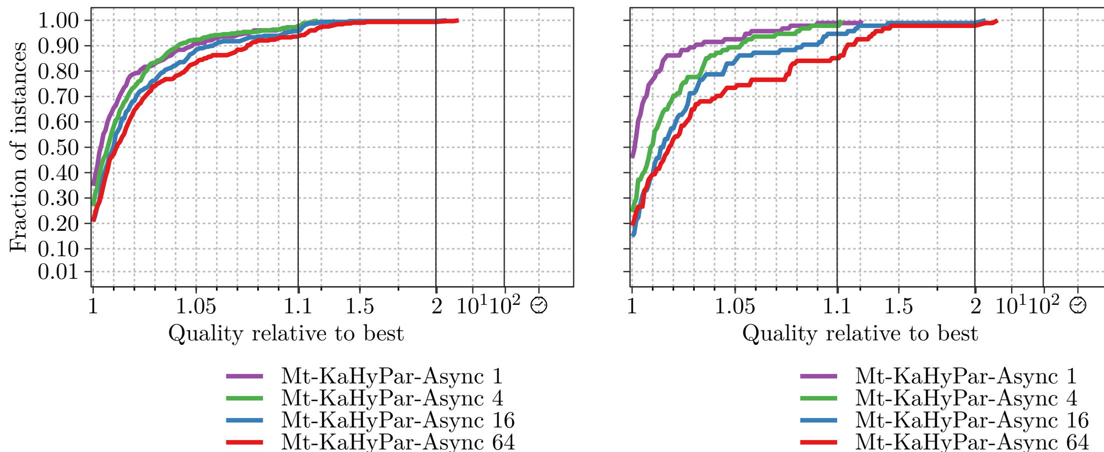

Figure 7.8.: Performance profile comparing the quality of Mt-KaHyPar-Async with 1, 4, 16 and 64 threads on the 82 hypergraph subset of set B for all instances (left) and only instances with a single-threaded running time of ≥ 1000 seconds (right).

number of groups that contain no boundary vertices. Then, more time is saved by no longer unnecessarily activating and deactivating hyperedges for those groups.
These improvements suggest that threads treating groups as active during their uncontraction hindered the scalability of asynchronous uncoarsening for instances with a small running time in our previous experiments.

With our optimization, we believe that we will see better speedups for small instances in future scalability experiments.

Quality with Increasing Number of Threads. The left plot in Figure 7.8 shows a performance profile that compares the quality of Mt-KaHyPar-Async with $p \in \{1, 4, 16, 64\}$ threads on the entire 82 hypergraph subset of set B. The right plot compares the solution quality only on those instances with a single-threaded running time of at least 1000 seconds. We can see that, overall, the quality of Mt-KaHyPar-Async decreases with increasing numbers of threads. The difference in quality between 1 and 64 threads on all instances is statistically significant according to a Wilcoxon signed rank test ($Z = -5.3912$ with $p = 6.997e-08$). This behavior intensifies for instances with a single-threaded running time of ≥ 1000 seconds.

7.6. Detailed Discussion

In this section, we evaluate and interpret the results of our experiments with respect to the original goal of asynchronous uncoarsening: increasing the scalability of Mt-KaHyPar-Q whilst not affecting the solution quality. Additionally, we give some conclusive remarks and ideas for possible future experiments based on the lessons we have learned.

Running Time and Scalability of Mt-KaHyPar-Async. Generally, we can say that we reached the goal of making Mt-KaHyPar-Q more scalable with asynchronous uncoarsening. Given the scalability data of Mt-KaHyPar-Async in Section 7.5 and the comparison with Mt-KaHyPar-Q in Section 7.4, we can see that Mt-KaHyPar-Async is generally faster than Mt-KaHyPar-Q and scales better for long-running instances. We believe that the cause for these improvements is the optimized parallelism that we achieved by reducing the number of global synchronization points using asynchronous uncoarsening.

However, the improvements are, perhaps, not as large as we had expected. Moreover, considering all benchmark instances, Mt-KaHyPar-Async does not scale any better than

Mt-KaHyPar-Q. We attribute this to several challenges we have encountered during our work on asynchronous uncoarsening that we had not fully been aware of previously.

Firstly, managing the gain cache as described in Chapter 5 and controlling the cross-dependencies mentioned in Chapter 6 introduces memory and time overheads. Our gain cache requires additional memory for an extra $k - 1$ benefit entries per node compared to the cache used in Mt-KaHyPar-Q. Furthermore, gain cache updates for moves and uncontractions are more expensive as stated in Section 5.1 and Section 5.2. This leads to an increased likelihood of lock contention, especially for large hyperedges, which can, in turn, reduce parallelism. In order to reduce this lock contention, we use snapshots to perform gain cache updates that concern every pin of a large hyperedge as described in Section 6.2 and Section 6.3. We found that on average fewer than 10% of all gain cache updates are performed with snapshots if we only perform snapshots for hyperedges with at least 1000 pins. For instances that required any gain cache updates with snapshots, keeping track of stable pins (see Section 6.3) reduced the number of pins that needed to be copied by 57% on set A and by 60% on set B. In our parameter tuning experiments (see Appendix A), we tested taking snapshots for all hyperedges and for hyperedges of at least 1000 pins and found that the latter has a slight running time advantage. However, the large percentages of stable pins suggest that in the future we should try values between 0 and 1000 as the minimum hyperedge size for snapshots, too.

Secondly, our locality sensitive uncoarsening used to control the interference with localized FM searches (see Section 6.4) causes a significant time overhead. As discussed in Section 7.5, we believe that inefficient locality asynchronous uncoarsening is a cause for the bad speedups on instances with a small running time in our scalability experiments. In that section, we give an optimization for locality asynchronous uncoarsening that may help with this issue in the future. However, even with our optimization, we observe that for short running instances each worker thread spends up to 30% of its time on picking suitable groups and updating the active region of the thread. Therefore, in the future, we want to reassess the methods used to store and compare the active regions of threads.

Loss of Quality Compared to Mt-KaHyPar-Q. We have not managed to achieve the same solution quality with asynchronous uncoarsening that synchronous Mt-KaHyPar-Q provides. We observe that Mt-KaHyPar-Async is outperformed by Mt-KaHyPar-Q in terms of quality especially with larger numbers of threads (compare Figure 7.2 and Figure 7.4). We see two causes for the decreased quality with asynchronous uncoarsening.

Firstly, we believe that the principal reason is the inherent increased interference in asynchronous uncoarsening. Generally, refinement heuristics in the asynchronous setting have to deal with a more volatile partition state caused by more frequent changes to the global partition. This can affect parallel gain calculation using the gain cache because of race conditions between reads and writes to gain cache entries. As explained in Section 6.4, localized FM searches are additionally afflicted by invalidated gain deltas due to concurrent changes of the global partition. Our approach to locality sensitive asynchronous uncoarsening cannot reliably prevent interference with refinement algorithms entirely. The loss in quality with an increasing number of threads (see Figure 7.8) suggests that there is room for improvement in future work. Perhaps, more effective diversification of search regions could improve the quality of asynchronous parallel partitioners in the future.

Secondly, some mechanisms used by the synchronous refinement phase of Mt-KaHyPar-Q are either not applicable in asynchronous uncoarsening at all or only work less effectively. We do not expect these missing features to greatly affect the solution quality of Mt-KaHyPar-Async. However, our experimental results do not allow us to gauge the impact of these features precisely so we mention them here, nonetheless.
To begin with, the parallel FM implementation of Mt-KaHyPar-Q constructs a global

move sequence and applies a global rollback on all moves at the end of the parallel FM phase (see Section 3.4.1). In asynchronous uncoarsening, we could similarly apply global rollbacks but this would require additional global synchronization points which we are trying to avoid. Thus, we opted not to include global rollbacks. Therefore, refinement in Mt-KaHyPar-Async lacks the global perspective which Mt-KaHyPar-Q obtains that way. In the future, we may attempt to introduce additional occasional synchronization points for global rollbacks in Mt-KaHyPar-Async to measure the effect on the quality of the partition.
Moreover, each call to localized refinement in Mt-KaHyPar-Q is handed a much larger set of refinement seeds than in Mt-KaHyPar-Async. Both partitioners repeat localized LP and FM on a set of refinement seeds iteratively until no more improvement is found. Therefore, the number of refinement iterations in Mt-KaHyPar-Q is greater on average simply because more possible moves are examined for a potential improvement. Then, Mt-KaHyPar-Q has more chances to find further improvements using the larger number of additional refinement iterations.

We suspect that interference is the paramount reason for the loss of quality while the difference in refinement mechanisms plays a minor role. We could gather further evidence for this hypothesis in a future experiment in which we would compare Mt-KaHyPar-Async with a variant of Mt-KaHyPar-Q that minimizes differences to Mt-KaHyPar-Async in the refinement phase. If the only difference between the partitioners were whether uncontractions are performed concurrently to refinement or separately, we could obtain insight into the effect on the quality of increased interference alone. According to our hypothesis, we would then expect the quality of Mt-KaHyPar-Async to still be worse than the variant of Mt-KaHyPar-Q and the scalability to still be better.

Final Remarks on Experiments. Ultimately, our experimental results presented here lead us to believe that asynchronous uncoarsening is a viable avenue of research for massive scale hypergraph partitioning. The increased scalability of asynchronous uncoarsening may outweigh a potential loss of quality for processing very large hypergraphs with large numbers of threads. For that purpose, we are confident that the quality gap can be reduced in the future using better approaches to asynchronous localized refinement.

8. Conclusion and Future Work

In this thesis, we demonstrate that asynchronous uncoarsening is viable for shared-memory n-level hypergraph partitioning and that it exhibits increased scalability compared with the existing n-level variant of the Mt-KaHyPar framework. We discuss core challenges arising with asynchronous uncoarsening and present effective solutions. These issues include cross-dependencies and interference caused by concurrent uncontractions and node moves as well as the management of a gain cache in the asynchronous context. Furthermore, we introduce the idea of explicit locality sensitive uncoarsening to decrease the amount of interference with refinement searches. We implement our approach in the Mt-KaHyPar hypergraph partitioning framework. Our experiments on more than 500 real-world hypergraphs show that asynchronous uncoarsening can improve the running times and scalability of an n-level parallel partitioner compared to separate phases of uncontractions and refinement. On average, we manage to find partitions of large hypergraphs 26% faster than the previous n-level version of Mt-KaHyPar. Using 64 threads, our asynchronous uncoarsening phase achieves average self-relative speedups of almost 40 for instances with a large single-threaded running time. We find that high-quality, asynchronous partitioners may in the future be able to process large hypergraphs for which other high-quality partitioners exceed prohibitive time limits.

Future work on the asynchronous paradigm needs to focus on bringing the quality of asynchronous refinement up to par with the previous, synchronous n-level Mt-KaHyPar variant. We see a large potential in the notion of locality sensitive uncoarsening and suggest additional work on more efficient and effective approaches to explicit diversification of search regions in the future. Here, the central issues for the future should be faster, more scalable region comparison and a more comprehensive representation of the active region of each thread. The diversification of search regions needs to be revised for short running instances where it currently makes up large parts of the total running time while not providing a clear quality advantage. With future insight, the basic idea of locality sensitive uncoarsening may also be applicable to batch construction and parallel localized FM searches for synchronous parallel partitioners. Lastly, the limited time of a master thesis did not allow us to engineer our algorithms to the standard of the Mt-KaHyPar framework. Future optimizations could be undertaken in that regard to settle asynchronous uncoarsening into the role of a more scalable variant of n-level uncoarsening in Mt-KaHyPar.

Bibliography

[1] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a Direct k-way Hypergraph Partitioning Algorithm. In *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 28–42. SIAM, 01 2017. doi: 10.1137/1.9781611974768.3.

[2] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-Quality Shared-Memory Graph Partitioning. In *European Conference on Parallel Processing (Euro-Par)*, pages 659–671. Springer, 8 2017. doi: 10.1007/978-3-319-96983-1_47.

[3] Charles J. Alpert. The ISPD98 Circuit Benchmark Suite. In *International Symposium on Physical Design (ISPD)*, pages 80–85, 4 1998. doi: 10.1145/274535.274546.

[4] Charles J. Alpert and Andrew B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration*, 19(1-2):1–81, 1995. doi: 10.1016/0167-9260(95)00008-4.

[5] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. The SAT Competition 2014. `http://www.satcompetition.org/2014/`, 2014.

[6] Thang N. Bui and Curt Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In *6th SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*, pages 445–452, 1993. URL `https://www.osti.gov/biblio/54439`.

[7] Michael J. Campbell and Thomas D.V. Swinscow. *Statistics at Square One*. BMJ Publishing Group, 2009.

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

[9] P.K. Chan, M.D.F. Schlag, and J.Y. Zien. Spectral K-way ratio-cut partitioning and clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems*, 13(9):1088–1096, 1994. doi: 10.1109/43.310898.

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

[11] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 11 2011. doi: 10.1145/2049662.2049663.

[12] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Catalyurek. Parallel Hypergraph Partitioning for Scientific Computing. In *IEEE Transactions on Parallel and Distributed Systems*, pages 10–pp. IEEE, 2006. doi: 10.1109/IPDPS.2006.1639359.

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

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

[15] J. Gong and Sung Kyu Lim. Multiway partitioning with pairwise movement. In *1998 International Conference on Computer-Aided Design (ICCAD)*, pages 512–516, 1998. doi: 10.1109/ICCAD.1998.144316.

[16] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced Flow-Based Multilevel Hypergraph Partitioning. *18th International Symposium on Experimental Algorithms (SEA)*, 2020. doi: 10.4230/LIPIcs.SEA.2020.11.

[17] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Shared-Memory n-level Hypergraph Partitioning. *CoRR*, abs/2104.08107, 2021. URL `https://arxiv.org/abs/2104.08107`.

[18] Gottesbüren, Lars and Heuer, Tobias and Sanders, Peter and Schlag, Sebastian. Scalable Shared-Memory Hypergraph Partitioning. In *23rd Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 01 2021. doi: 10.1137/1.9781611976472.2.

[19] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, March 2021. ISSN 2521-327X. doi: 10.22331/q-2021-03-15-410. URL `https://doi.org/10.22331/q-2021-03-15-410`.

[20] Vitali Henne, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, and Christian Schulz. n-Level Hypergraph Partitioning. *CoRR*, abs/1505.00693, 2015. URL `http://arxiv.org/abs/1505.00693`.

[21] Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *16th International Symposium on Experimental Algorithms (SEA)*, pages 21:1–21:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 06 2017. doi: 10.4230/LIPIcs.SEA.2017.21.

[22] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. *ACM Journal of Experimental Algorithmics (JEA)*, 24(1):2.3:1–2.3:36, 09 2019. doi: 10.1145/3329872.

[23] Wenkai Jiang, Jianzhong Qi, Jeffrey Xu Yu, Jin Huang, and Rui Zhang. HyperX: A Scalable Hypergraph Framework. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):909–922, 2019. doi: 10.1109/TKDE.2018.2848257.

[24] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. volume 10, pages 1418–1429, 2017. doi: 10.14778/3137628.3137650.

[25] George Karypis and Vipin Kumar. Multilevel k-way Hypergraph Partitioning. *VLSI Design*, 2000(3):285–300, 2000. doi: 10.1155/2000/19436. URL `https://doi.org/10.1155/2000/19436`.

[26] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999. doi: 10.1109/92.748202.

[27] K Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. SWORD: Workload-Aware Data Placement and Replica Selection for Cloud Data Management Systems. *The VLDB Journal*, 23(6):845–870, 2014. doi: 10.1007/s00778-014-0362-1.

[28] Jesper Larsson Träff. Direct graph k-partitioning with a Kernighan–Lin like heuristic. *Operations Research Letters*, 34(6):621–629, November 2006. doi: 10.1016/j.orl.2005.10.003.

[29] Dominique LaSalle and George Karypis. Multi-Threaded Graph Partitioning. In *IEEE Transactions on Parallel and Distributed Systems*, pages 225–236. IEEE, 2013.

[30] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990. doi: 10.1017/S0263574700015691.

[31] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. BiPart: A Parallel and Deterministic Multilevel Hypergraph Partitioner. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–174, 2021. URL `https://doi.org/10.1145/3437801.3441611`.

[32] Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. HYPE: Massive Hypergraph Partitioning With Neighborhood Expansion. In *IEEE International Conference on Big Data*, pages 458–467. IEEE Computer Society, 2018. doi: 10.1109/BigData.2018.8621968.

[33] 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.

[34] Vitaly Osipov and Peter Sanders. n-Level Graph Partitioning. In *18th European Symposium on Algorithms (ESA)*, pages 278–289. Springer, 2010.

[35] David A. Papa and Igor L. Markov. Hypergraph Partitioning and Clustering. In *Handbook of Approximation Algorithms and Metaheuristics*. 2007. doi: 10.1201/9781420010749.ch61.

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

[37] Sanchis, Laura A. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989. doi: 10.1109/12.8730.

[38] Sanchis, Laura A. Multiple-way network partitioning with different cost functions. *IEEE Transactions on Computers*, 42(12):1500–1504, 1993. doi: 10.1109/12.260640.

[39] Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *19th European Symposium on Algorithms (ESA)*, pages 469–480. Springer, 2011.

[40] Sebastian Schlag. High-Quality Hypergraph Partitioning. 2020.

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

[42] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-Quality Hypergraph Partitioning. *CoRR*, abs/2106.08696, 2021. URL `https://arxiv.org/abs/2106.08696`.

[43] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016. doi: 10.14778/3025111.3025125.

[44] Aleksandar Trifunovic and William J. Knottenbelt. Parkway 2.0: A Parallel Multilevel Hypergraph Partitioning Tool. In *International Symposium on Computer and Information Sciences*, pages 789–800. Springer, 2004. doi: 10.1007/978-3-540-30182-0_79.

[45] B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005. doi: 10.1137/S0036144502409019.

[46] Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite. In *49th Conference on Design Automation (DAC)*, pages 774–782. ACM, 6 2012. doi: 10.1145/2228360.2228500.

[47] Frank Wilcoxon. Individual Comparisons by Ranking Methods. In *Breakthroughs in Statistics*, pages 196–202. Springer, 1992.

[48] Marvin Williams, Peter Sanders, and Roman Dementiev. Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues. *CoRR*, abs/2107.01350, 2021. URL `https://arxiv.org/abs/2107.01350`.

[49] Wenyin Yang, Guojun Wang, Kim-Kwang Raymond Choo, and Shuhong Chen. HEPart: A Balanced Hypergraph Partitioning Algorithm for Big Data Applications. *Future Generation Computer Systems*, 83:250–268, 2018. doi: 10.1016/j.future.2018.01.009.

Appendix

A. Parameter Tuning Experiments

In this section, we will describe our parameter tuning experiments in more detail. We used a subset of set A containing 19 comparatively small hypergraphs. We ran the experiment with $p = 16$ threads, $k \in \{2, 8, 16, 64\}$ and 10 seeds. In the following, for each of our four configuration parameters, we describe the set of values we examined and their impact on the running time and the solution quality of Mt-KaHyPar-Async.

A.1. Minimum Edge Size for Snapshots (cp_{snap}).

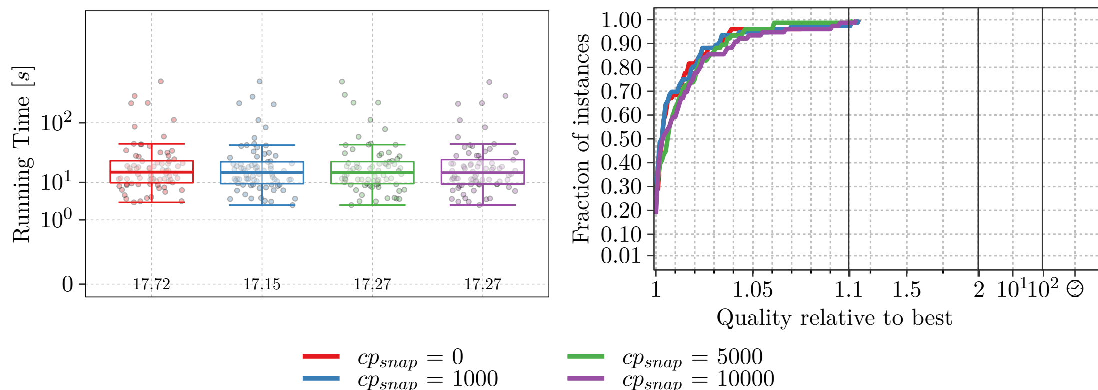

Figure A.1.: Running time box plot and performance profile for the tested values of cp_{snap}.

In Section 6.3, we state that we take snapshots of the pin list and connectivity information of hyperedges to perform gain cache updates outside of a hyperedge lock. For large hyperedges this can reduce lock contention caused by a large number of concurrent gain cache updates on the hyperedge. However, the overhead for taking snapshots may outweigh this benefit for smaller hyperedges. Therefore, we interpolate between these arguments by taking snapshots only for hyperedges of a minimum size cp_{snap}.

We tested the values $cp_{snap} \in \{0, 1000, 5000, 10000\}$ for the minimum number of pins cp_{snap} and found that neither the running time nor the quality differed too much between those values. We show a running time box plot and a performance profile plot for the different values of cp_{snap} in Figure A.1. We decided to use a threshold of $cp_{snap} = 1000$ pins as it

provides the shortest running time. In the future, we will also test values between 0 and 1000 with a finer resolution. This may improve the running time as we found that stable pins managed to reduce the size of pin list snapshots significantly (see Section 7.6).

A.2. Maximum Region Similarity (cp_{sim}).

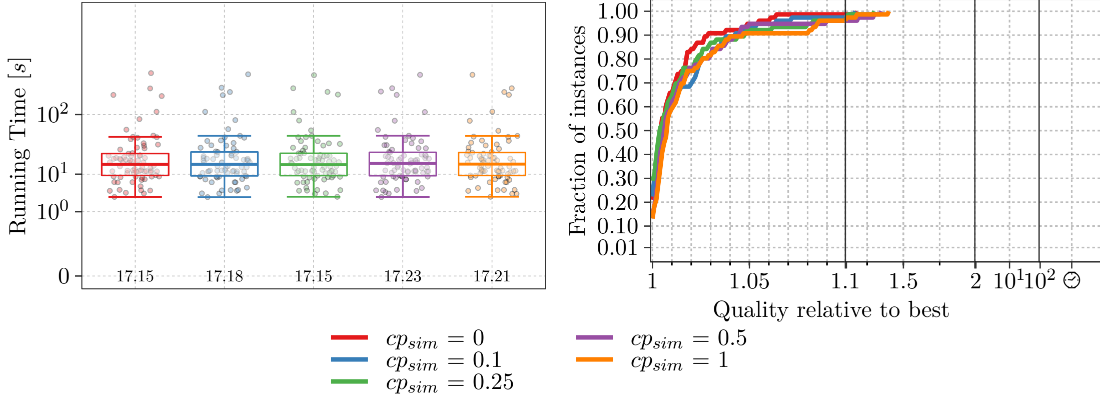

Figure A.2.: Running time box plot and performance profile for the tested values of cp_{sim} on the original 19 small hypergraphs.

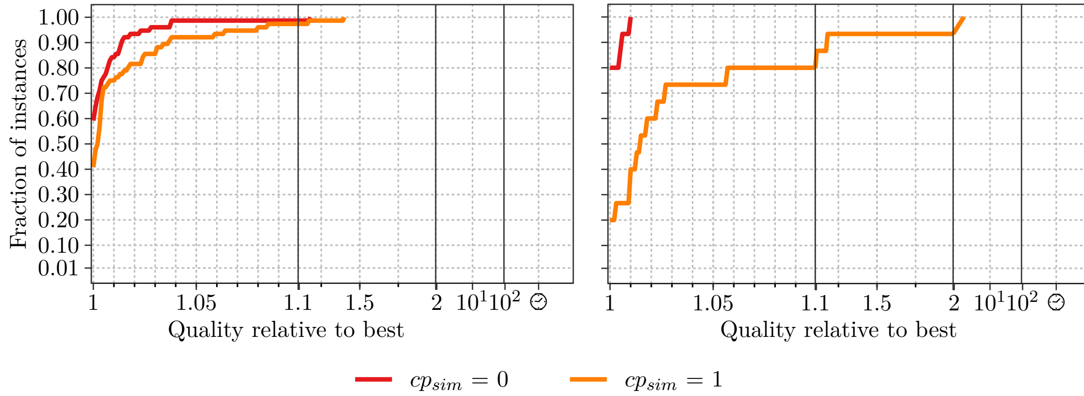

Figure A.3.: Performance profiles comparing $cp_{sim} = 0$ and $cp_{sim} = 1$ on the original 19 hypergraphs with $p = 16$ threads (left) and on five larger hypergraphs with $p = 64$ (right).

In Section 6.4, we explain two configuration parameters related to a thread t picking a group g with regards to locality sensitive uncoarsening. The first parameter is the maximum permissible similarity between the region of g and the active regions of threads other than t. In the context of parameter tuning, we call this configuration parameter cp_{sim}. With larger values of cp_{sim}, t may find a group with a region similarity smaller than cp_{sim} with fewer tries. However, larger permissible similarities may also lead to more interference with localized FM searches. Therefore, we expect this parameter to decide a trade-off between running time and quality.

We tested the values $cp_{sim} \in \{0, 0.1, 0.25, 0.5, 1\}$ and found that this value only has a small effect on running time and quality for our parameter tuning instances as the plots in Figure A.2 show.

This result suggests that locality sensitive uncoarsening generally has little effect on the partition quality. However, we attribute this only to an ineffective experiment on our part. In order to accommodate thousands of runs for the parameter tuning experiments, we chose

small instances with a short total partitioning time. An evaluation of partition quality on small instances cannot reliably predict the effects of different parameter values on the solution quality for larger hypergraphs, though. In fact, locality sensitive uncoarsening was originally motivated by bad partitions of larger hypergraphs. Therefore, the parameter tuning experiment for cp_{sim} needs to be repeated on larger instances.
At this point, we will only provide a small experiment on larger hypergraphs as a kind of sanity check for the effectiveness of locality sensitive uncoarsening: We tested the values $cp_{sim} = 0$ (refinement seeds in different threads cannot be adjacent) and $cp_{sim} = 1$ (no explicit diversification of search regions) on our five larger example hypergraphs from Section 7.5 (see Table 7.4 for hypergraph metrics). For this small experiment, we used $p = 64$ threads, $k \in \{2, 16, 64\}$ and five seeds.
We show performance profiles comparing the two values on the original 19 hypergraphs and on the five larger hypergraphs in Figure A.3. We observe that the quality difference between $cp_{sim} = 0$ and $cp_{sim} = 1$ is much greater for the larger instances on the right. For the five larger hypergraphs, Mt-KaHyPar-Async was, on average, less than 5% slower with $cp_{sim} = 0$ than with $cp_{sim} = 1$. This small experiment does not replace thorough parameter tuning on larger instances but gives some indication that the quality results for small instances do not necessarily hold for larger instances with more threads.

Due to time constraints, we based our decision for a value of cp_{sim} only on these preliminary results and chose $cp_{sim} = 0$ to prioritize solution quality.
In the future, we will consider further large instances with more values for cp_{sim} to gain an understanding of the trade-off between quality and running time that the parameter defines. Nonetheless, the results from the experiments presented here also imply that locality sensitive uncoarsening may not be of much value for small instances. Therefore, in the future, we have to reconsider for which instances the diversification of search regions is actually worth the time overhead.

A.3. Number of Group Picking Retries ($cp_{retries}$).

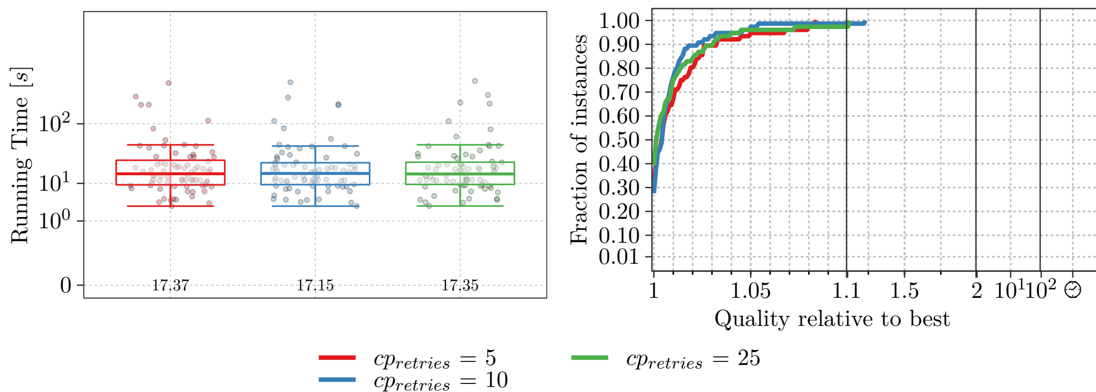

Figure A.4.: Running time box plot and performance profile for the tested values of $cp_{retries}$.

The second configuration parameter related to locality sensitive uncoarsening is the number of groups $cp_{retries}$ that a thread t examines to find a group with a permissible region similarity (see Section 6.4). If t has examined $cp_{retries}$ groups and not found one with a region similarity of cp_{sim} or less (see Appendix A.2), t will resort to accepting the group with the smallest region similarity out of the examined ones. Therefore, with higher values, the chance of finding a group that is not too similar to the active regions of other threads increases, which can lead to less interference and improved partition quality. However, examining more groups also takes longer.

We tried the values $cp_{retries} \in \{5, 10, 25\}$. We observed that we achieve the best running time with $cp_{retries} = 10$ while the quality is not significantly affected by differing values as the plots in Figure A.4 demonstrate. Keep in mind, though, that the evaluation of locality sensitive uncoarsening on our small benchmark instances may not be indicative of the effects on larger instances as we describe in Appendix A.2.

A.4. Minimum Number of Seeds for Loc. Refinement (cp_{seeds}).

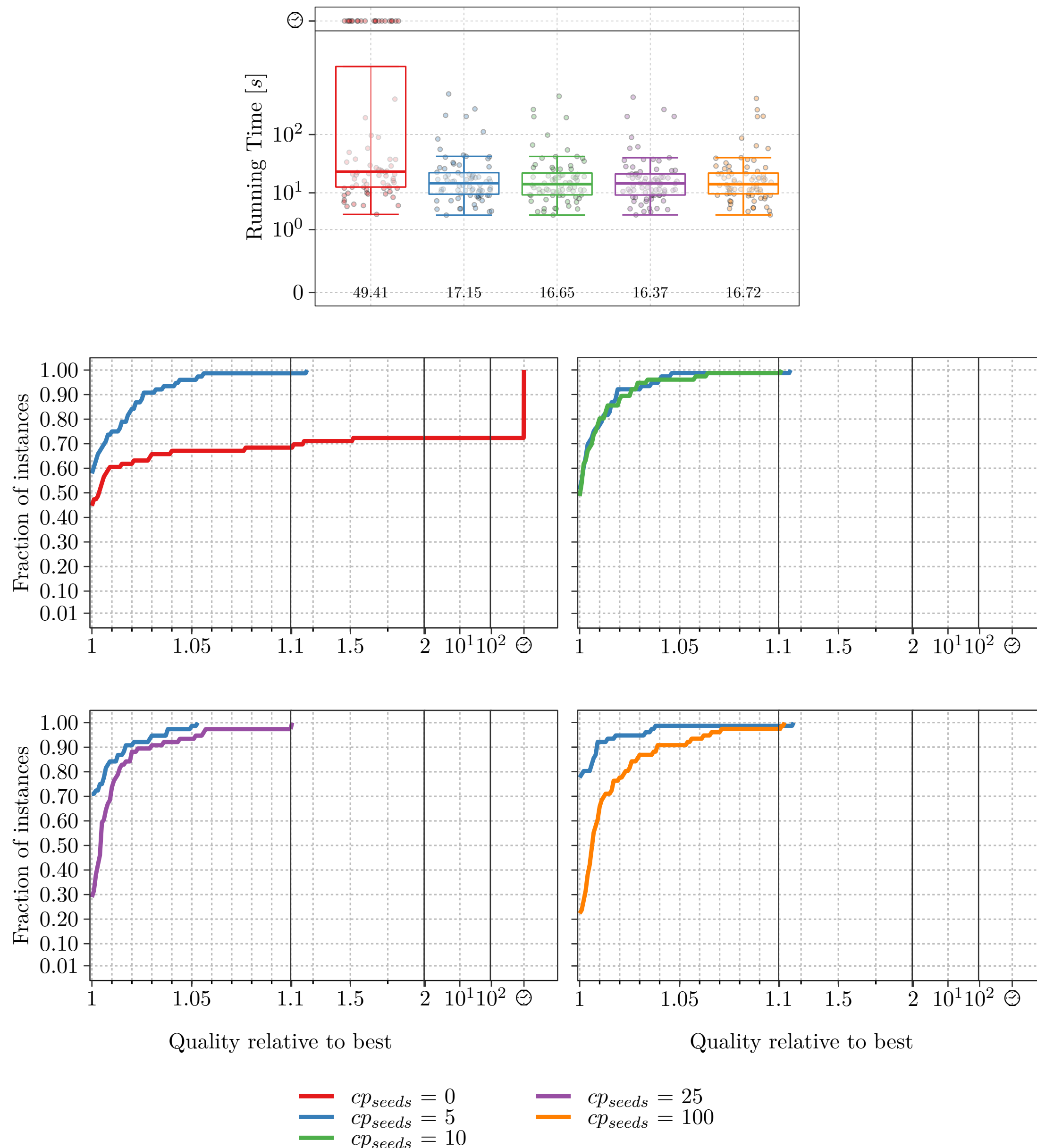

Figure A.5.: Running time box plot and performance profiles for the tested values of cp_{seeds}.

In Section 4.2, we describe that localized refinement searches are the most efficient with a small minimum number of seed nodes that we call cp_{seeds} here. Therefore, each thread accumulates at least cp_{seeds} seed nodes, possibly from multiple groups, before starting a localized refinement search. With larger values of cp_{seeds}, localized refinement is applied less frequently which potentially reduces the solution quality but decreases the running time. Also, as explained in Section 4.2, with too few refinement seeds, localized FM searches are forced to attempt time-intensive local moves with negative gain that are unlikely to lead to an improvement. Therefore, in this parameter tuning experiment, we try to find a

value that is small enough to provide good quality solutions but also large enough to allow efficient localized refinement.

We tested the values $cp_{seeds} \in \{0, 5, 10, 25, 100\}$. We found that cp_{seeds} has the largest effect on the running time among our configuration parameters. The running time plot in Figure A.5 shows that applying localized refinement whenever any seeds are found ($cp_{seeds} = 0$) has considerable running time implications. Many runs with this value reached our time limit of 600 seconds. Considering the rest of the values, we see that the running time with $cp_{seeds} = 25$ is the smallest.
Interestingly, with a minimum of 25 seeds, our algorithm is also faster than with a minimum of 100 seeds. We believe that this may be caused by the relation of the minimum number of seeds to locality sensitive uncoarsening: A thread t is considered to be working on a group g if t has already uncontracted g and has extracted seeds from g (see Section 6.4). Only after t finishes the localized refinement search based on the seeds extracted from g, we no longer consider t to be working on g. Therefore, if t needs to collect more seed nodes for a run of localized refinement, the number of groups that t is working on also grows larger. In turn, this may make it harder for other threads to find groups that are not in the active region of t which increases the total running time.
Regarding the solution quality, we can see in the performance profile plots in Figure A.5 that the quality does not differ a lot between a minimum number of seeds of 5 and 10. However, $cp_{seeds} = 5$ does outperform the other values regarding quality, though. We decided to prioritize quality and used $cp_{seeds} = 5$ in our experiments. This decision was additionally motivated by the similarity to Mt-KaHyPar-Q in which parallel localized FM searches (see Section 3.4.1) use 5 seeds per search. In the future, we would like to consider the implications of using $cp_{seeds} = 10$ and $cp_{seeds} = 25$ for the running time and quality of Mt-KaHyPar-Async on larger instances.

B. Additional Plots

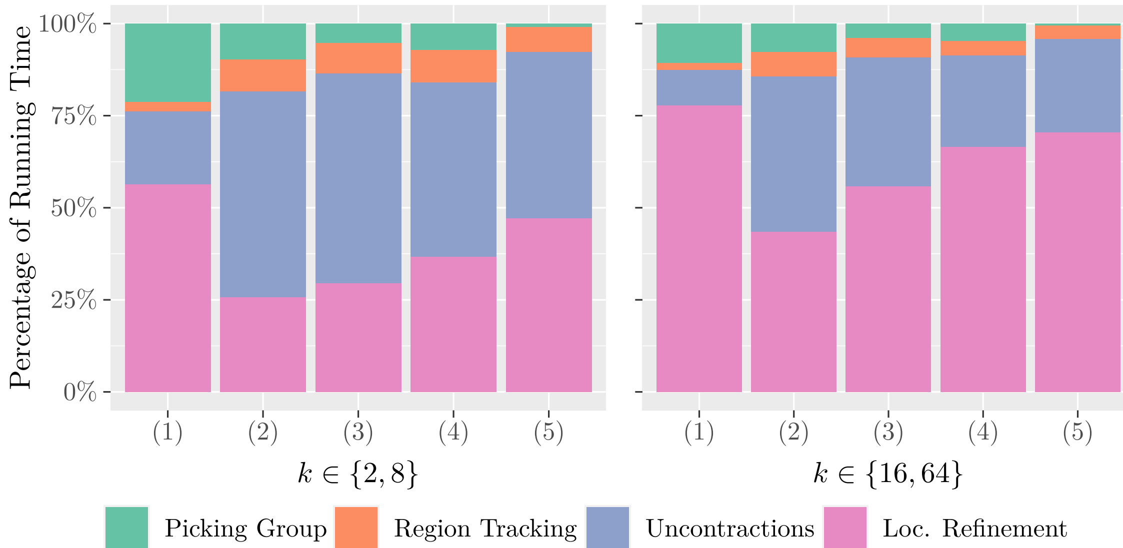

Figure B.1.: Fraction of running time that each component of asynchronous uncoarsening takes for Mt-KaHyPar-Async 64 with the opimization described in Section 6.4 on five different hypergraphs (see Table 7.4 for hypergraph metrics). Arithmetic mean for 5 seeds and $k \in \{2, 8\}$ (left) and for 5 seeds and $k \in \{16, 64\}$ (right).