Parallel and Flow-Based High Quality Hypergraph Partitioning

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

Dissertation

von

Lars Gottesbüren
aus Aachen

Tag der mündlichen Prüfung:	17.11.2022
Erste Referentin:	Prof. Dr. Dorothea Wagner
Zweiter Referent:	Prof. Dr. Henning Meyerhenke

Abstract

Balanced hypergraph partitioning is a classic NP-hard optimization problem that is a fundamental tool in such diverse disciplines as VLSI circuit design, route planning, sharding distributed databases, optimizing communication volume in parallel computing, and accelerating the simulation of quantum circuits. Given a hypergraph and an integer k, the task is to divide the vertices into k disjoint blocks with bounded size, while minimizing an objective function on the hyperedges that span multiple blocks. In this dissertation we consider the most commonly used objective, the connectivity metric, where we aim to minimize the number of different blocks connected by each hyperedge.

The most successful heuristic for balanced partitioning is the multilevel approach, which consists of three phases. In the coarsening phase, vertex clusters are contracted to obtain a sequence of structurally similar but successively smaller hypergraphs. Once sufficiently small, an initial partition is computed. Lastly, the contractions are successively undone in reverse order, and an iterative improvement algorithm is employed to refine the projected partition on each level.

An important aspect in designing practical heuristics for optimization problems is the trade-off between solution quality and running time. The appropriate trade-off depends on the specific application, the size of the data sets, and the computational resources available to solve the problem. Existing algorithms are either slow, sequential and offer high solution quality, or are simple, fast, easy to parallelize, and offer low quality. While this trade-off cannot be avoided entirely, our goal is to close the gaps as much as possible. We achieve this by improving the state of the art in all non-trivial areas of the trade-off landscape with only a few techniques, but employed in two different ways. Furthermore, most research on parallelization has focused on distributed memory, which neglects the greater flexibility of shared-memory algorithms and the wide availability of commodity multi-core machines.

In this thesis, we therefore design and revisit fundamental techniques for each phase of the multilevel approach, and develop highly efficient shared-memory parallel implementations thereof. We consider two iterative improvement algorithms, one based on the Fiduccia-Mattheyses (FM) heuristic, and one based on label propagation. For these, we propose a variety of techniques to improve the accuracy of gains when moving vertices in parallel, as well as low-level algorithmic improvements. For coarsening, we present a parallel variant of greedy agglomerative clustering with a novel method to resolve cluster join conflicts on-the-fly. Combined with a preprocessing phase for coarsening based on community detection, a portfolio of from-scratch partitioning algorithms, as well as recursive partitioning with work-stealing, we obtain our first parallel multilevel framework. It is the fastest partitioner known, and achieves medium-high quality, beating all parallel partitioners, and is close to the highest quality sequential partitioner.

Our second contribution is a parallelization of an n-level approach, where only one vertex is contracted and uncontracted on each level. This extreme approach aims at high solution quality via very fine-grained, localized refinement, but seems inherently sequential. We devise an asynchronous n-level coarsening scheme based on a hierarchical decomposition of the contractions, as well as a batch-synchronous uncoarsening, and later fully asynchronous uncoarsening. In addition, we adapt our refinement algorithms, and also use the preprocessing and portfolio. This scheme is highly scalable, and achieves the same quality as the highest quality sequential partitioner (which is based on the same components), but is of course slower than our first framework due to fine-grained uncoarsening.

The last ingredient for high quality is an iterative improvement algorithm based on maximum flows. In the sequential setting, we first improve an existing idea by solving incremental maximum flow problems, which leads to smaller cuts and is faster due to engineering efforts. Subsequently, we parallelize the maximum flow algorithm and schedule refinements in parallel.

Beyond the strive for highest quality, we present a deterministically parallel partitioning framework. We develop deterministic versions of the preprocessing, coarsening, and label propagation refinement. Experimentally, we demonstrate that the penalties for determinism in terms of partition quality and running time are very small.

All of our claims are validated through extensive experiments, comparing our algorithms with state-of-the-art solvers on large and diverse benchmark sets. To foster further research, we make our contributions available in our open-source framework Mt-KaHyPar.

While it seems inevitable, that with ever increasing problem sizes, we must transition to distributed memory algorithms, the study of shared-memory techniques is not in vain. With the multilevel approach, even the inherently slow techniques have a role to play in fast systems, as they can be employed to boost quality on coarse levels at little expense. Similarly, techniques for shared-memory parallelism are important, both as soon as a coarse graph fits into memory, and as local building blocks in the distributed algorithm.

Acknowledgements

I am deeply grateful to my advisor Dorothea Wagner for granting me the opportunity to work towards my PhD in her group. The doctoral students here enjoy a great deal of freedom when pursuing their own research directions, which cannot be appreciated enough.

A special thank you to my office mate Michael Hamann for being a great mentor, teaching me resilience, being extremely dilligent when proof-reading, and always being intent on chasing pragmatic solutions to algorithmic problems. Thank you to Tobias Heuer for pushing me to finish my parts of the code, being a great collaborator, again teaching me the importance of pragmatism, and of course the jokes. Thank you to Sebastian Schlag for always having an open door. Your experience and advice were invaluable.

I would also like to extend my sincere gratitude to all of the additional collaborators and co-authors I worked with during my time here, in no particular order: Peter Sanders, Daniel Seemaier, Christian Schulz, Noah Wahl, Niklas Uhl, Thomas Bläsius, Philipp Fischbeck, Jonas Spinner, Christopher Weyand and Marcus Wilhelm. And finally thank you to Tim Zeitz, Noah Wahl, and Daniel Seemaier for proof-reading parts of this dissertation.

Contents

1 Introduction

1.1 Motivation

Balanced graph partitioning is a classic NP-hard optimization problem with a rich and beautiful research history. The goal is to partition the vertices of a graph into a fixed number of disjoint blocks with bounded size, with as few edges running between blocks as possible. Practical studies are often motivated by an application. In fact not many optimization problems have such a diverse set of applications as balanced partitioning. Perhaps the most archetypical application is data distribution while minimizing communication between parallel processors.

The Role of Hypergraphs. Yet there are applications where the restriction to pair-wise relationships is inaccurate at modeling the more complex relationships prevalent [SK72, CA99, KQDK14, GK21]. Hypergraphs are a generalization of graphs where each hyperedge can connect an arbitrary subset of the vertices. The corresponding hypergraph partitioning problem rose to popularity in the context of VLSI design [SK72, FM82, Kri84, San89, Len90, CS93, AK95, YW96, Alp98, KAKS99]. The goal is to divide circuit elements while producing as few wires running between components as possible. In the graph model, a wire connecting more than two elements is modeled as a clique, however multiple cut clique edges translate to just one external wire in the chip design. Modeling wires as hyperedges instead of cliques yields the correct cost model [SK72].

The objective function used in this application is the number of cut hyperedges (*cut-net metric*). Since a hyperedge can connect arbitrary many vertices, it can also span more than two blocks. To account for this, the *connectivity metric* counts the number of different blocks

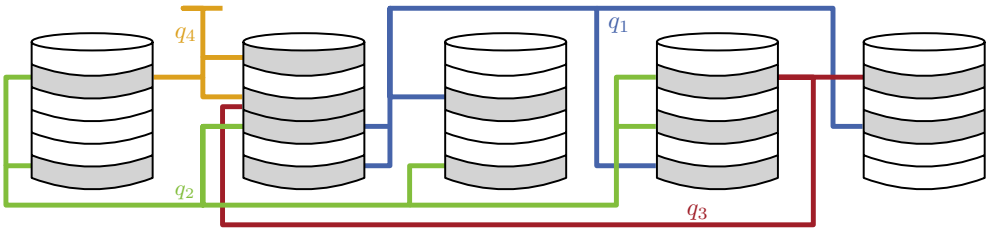

Figure 1.1: A distributed database with each query accessing multiple machines.

spanned by each hyperedge (minus one necessary block), with the goal to span as few blocks as possible over all hyperedges.

Hypergraph partitioning enjoys wide-spread use in many more application areas where hypergraphs are the appropriate model to capture the cost associated with a partition. In particular the connectivity metric is relevant in practice, thus it is the main focus in this work. Some prominent examples are parallel sparse matrix-vector multiplication [CA99], SAT solving [MP14, MP17], quantum computing [AH19, GK21, Hua+20, PZ21] and sharding distributed databases [CZJM10, KQDK14, Ser+16, YWCC18, Kab+17a]. In the following we outline the database application further, as it has become extremely important in recent years.

Data Distribution Applications. Tech companies operate enormous data centers to host and query "their" data. Optimizing data placement not only accelerates queries or increases throughput, but also has a significant impact on reducing operation costs. Considering the absolute scale of these operations[1], even small relative savings are worthwhile. Shalita et al. [Sha+16] report an impressive 50% decrease in both CPU utilization and query latency in a data center with thousands of storage servers, using only a very simple optimization technique [Kab+17a].

In this application, vertices correspond to data records, hyperedges represent queries that access multiple data records, and the blocks are machines. The goal is to place data records that are frequently queried together on the same machine in order to improve query locality, while adhering to memory constraints of the machines and not overloading them with work. More precisely, minimizing connectivity corresponds to minimizing the average number of machines involved in a query, which optimizes the latency and query processing time. The queries are collected from logs, working under the assumption that optimizing for the historic query loads is representative for future queries. Figure 1.1 shows an example hypergraph and partition representing a sharded database. While there are previous works that apply graph partitioning to this problem [CZJM10], hypergraphs are clearly the more natural choice [KQDK14]. Whether machines cooperate in solving a query, or whether one machine

[1] `https://dgtlinfra.com/facebook-18-data-centers-20bn-investment/`

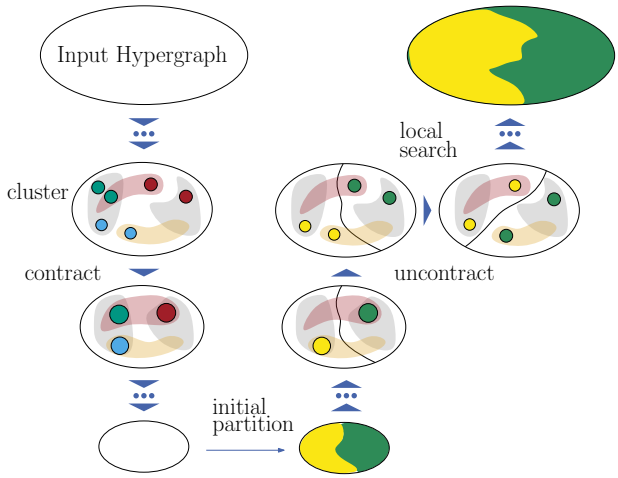

Figure 1.2: The multilevel paradigm illustrated.

gathers the necessary records and then performs the query, the optimization objective is the same: reduce the number of involved machines to reduce the incurred communication and latency.

Hardness and Heuristics. Unfortunately, solving the balanced partitioning problem optimally is NP-hard [Len90], and it is even NP-hard to compute approximate solutions within a constant factor [BJ92]. In practice, heuristics are used to compute adequate close-to-optimal solutions, without any theoretical approximation guarantees. Broadly speaking there are two categories: from-scratch algorithms and iterative improvement algorithms which take a given partition and try to improve it by moving vertices between the blocks. Clearly the two can be combined. The most successful heuristic is the *multilevel* paradigm [HL93], which adds a third phase beforehand. Figure 1.2 shows an illustration of the three phases.

In the *coarsening* phase, vertex clusters are contracted to obtain a sequence of structurally similar but successively smaller hypergraphs, which constitute the levels. Once the last hypergraph is sufficiently small, a from-scratch algorithm computes an *initial partition* on it. Finally, the contractions are successively undone in reverse order in the *uncoarsening* or *refinement* phase. At each level, the vertices are assigned to the same block as their coarse representative on the previous level, which results in an equivalent partition on the finer hypergraph in terms of balance and objective function. Iterative improvement algorithms then refine the projected partition, in order to improve its objective function, before continuing with the next level.

The Role of Parallelism. In recent years, problem sizes have grown beyond what full-fledged multilevel single-threaded codes can handle in reasonable time. Hence, there is an increased need for performing partitioning tasks in parallel. This led to very fast but overly simple approaches that are easy to parallelize, some in distributed memory, or even approaches that abandoned the multilevel paradigm altogether. As a consequence these approaches achieve abysmal solution quality. While there are some situations where this is appropriate [Jia+19], for most applications the solution quality directly impacts the application scalability and thus this is a bad trade-off. Therefore our focus is on techniques for high solution quality and the multilevel paradigm. As Moore's law continues to stall, chip developers keep increasing the number of cores per chip. This makes research on shared-memory algorithms attractive, additionally considering the fact that there is more flexibility for algorithm design than in distributed memory environments.

1.2 Summary of Main Contributions

As such, this dissertation presents two separate shared-memory parallel frameworks for multilevel partitioning. The first is a classic multilevel algorithm focused on speed and medium-high quality with approximately $\log(n)$ levels, where n is the number of vertices. We call this algorithm `Mt-KaHyPar-D` for multi-threaded Karlsruhe Hypergraph Partitioning framework, where D stands for default configuration. The second is an n-level algorithm focused purely on high quality, where on each level only one vertex is removed and later restored [OS10, AHSS17]. This achieves high quality, because the instances are as similar as possible between refinement steps, offering the finest possible granularity for refinement. We call this approach `Mt-KaHyPar-Q` for quality. Furthermore, we equip both frameworks with a parallel version of flow-based refinement [GHW19a, GHSW20], the currently strongest iterative improvement algorithm in terms of partition quality. And finally we propose a deterministic version of the $\log(n)$-level partitioner.

To derive meaningful conclusions from experiments, our benchmark sets contain a large number of instances (488 + 94 + 53), which are compiled from well-established benchmark sets [HS17, ASS18a] as well as applications such as VLSI design, SAT solving, scientific computing, social networks and web graph matrices. The instances have strongly varying characteristics, and the largest hypergraph instance has around 2 billion pins. We outline the selection process and characteristics of our benchmark instances as well as the competing state-of-the-art solvers in Section 2.4.

As we will show in this dissertation, our algorithms are the new state of the art in the high speed *and* high quality regimes, as they currently occupy all non-trivial fronts of the Pareto trade-off curve. Depending on the application and data set size, either Mt-KaHyPar-D or one of our variants with flow-based refinement is the method of choice. To illustrate this further, in Figure 1.3 we draw a simplified sketch of the trade-offs between solution quality (y-axis) and speed (x-axis) achieved by relevant hypergraph partitioners from the literature. The partitioners are color-coded by whether they are sequential, shared-memory parallel or

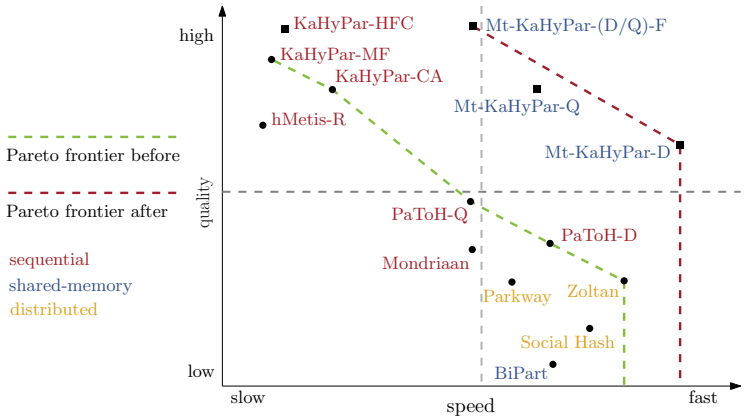

Figure 1.3: A hand-wavy sketch of the Pareto trade-offs between speed and quality achieved by several hypergraph partitioners in our evaluations. Our contributions are marked with a square, prior works are marked with a dot. The green dashed line shows the Pareto front prior to this work, the red dashed line the front with our work included.

distributed. Our contributions are marked with a square, whereas prior works are marked with a dot. We draw the Pareto frontiers before this work (green) and after (red), showing that we improved in all places along the frontier.

Fast Parallel Multilevel Partitioning. For the coarsening phase of Mt-KaHyPar-D, we choose a greedy agglomerative clustering algorithm [CA99]. We present a parallelization of this algorithm with on-the-fly resolution of cluster join conflicts, as well as a parallel contraction algorithm based on prefix sums, fingerprinting and hashing. Additionally we use a preprocessing technique based on community detection [HS17] to boost solution quality, which we parallelize using the parallel Louvain algorithm [SM16] where we contribute some implementation-level improvements.

For the refinement phase we follow a previous work [ASS18a] on parallelizing localized refinement for graph partitioning, but improve upon it in several ways. We put vertex moves in a better ordering, that more accurately reflects the gains (objective function difference) at the time they are computed, and parallelize a previously sequential step to recalculate exact gains if moves were performed in that sequence. Furthermore, we employ a gain table to facilitate faster updates in the neighborhoods of moved vertices, as well as a means to infuse up-to-date global partition information into the local move decisions. While the concept of gain tables is not new, we are the first to prove that they can be updated efficiently in parallel, and the first to use them as a communication tool. Furthermore, a naive adaptation of [ASS18a] to hypergraphs would lead to an infeasible amount of extra memory used for thread-local hash tables. We propose two very simple methods to keep memory consumption

low. Besides parallel gain recalculation and parallel gain tables, we propose a third technique to enhance the accuracy of gains, named attributed gains, which we use to double-check calculated gains and trace overall improvements. We additionally use label propagation refinement [KK00] because it is easy to parallelize.

Compared to the conference publication, we analyze our algorithms in the work-depth model and provide extensive experimental evaluations of the components and optimizations, showcasing their impact on partition quality and running time. Furthermore, we describe extensive details of the algorithms and engineering to facilitate better understanding for engineers wishing to re-implement our approaches, an area that has been lacking in the past and for which the partitioning community has received criticism [CKM00a].

In a comparison with existing hypergraph partitioning algorithms, we demonstrate that our algorithm is the fastest partitioner, achieves excellent speedups (around 22 on 64 cores), and achieves medium to high partition quality in the pool of compared algorithms, which is close to competitive with hMetis [KAKS99] or KaHyPar [Sch20]. These are the highest quality sequential partitioners, which are however much slower. Out of the fast or parallel algorithms it achieves the highest solution quality by far.

Furthermore, even in a comparison with dedicated parallel graph partitioners [ASS18a, LK13, Got+21] our algorithm achieves the highest quality, demonstrating again the effectiveness of our improvements. Furthermore it is the second fastest algorithm in this comparison, the only faster code is one of our own, which uses only label propagation [Got+21].

Deterministic Parallel Partitioning. Subsequently, we develop a deterministic $\log(n)$-level partitioner, which still reaps the performance benefits of randomized scheduling. Determinism is an important feature for certain logic synthesis applications [MABP21], and at the very least a highly convenient property [Ste90, BFGS]. For agglomerative clustering in coarsening, community detection preprocessing and label propagation refinement, we apply the synchronous local moving approach [HSWZ18]. Vertex moves are calculated and performed in synchronous steps, such that concurrent move decisions do not influence each other. In the presence of weight constraints on the clusters, some of the moves are approved and performed, whereas some are denied. The algorithmic novelty lies in the details of the approval steps, and for community detection in subsequent updates. For example the refinement approval uses a merge-style parallelization. Furthermore, we show how to incorporate determinism in the remaining parts of the framework, such as contraction and initial partitioning.

We demonstrate that the deterministic version of our algorithm achieves similar quality as the non-deterministic version, though small penalties are incurred, which are justifiable if determinism is desired in the application. Additionally, it is a bit slower, but achieves better speedups (around 29 on 64 cores). Furthermore, it completely outperforms a recent deterministic multilevel hypergraph partitioner [MABP21] in terms of partition quality, running time and speedups. Finally, we analyze likely sources for the quality loss, and suggest directions for improving this in future work.

Parallel n-Level Partitioning. In this work, we parallelize the core component of the highest quality sequential partitioner KaHyPar [AHSS17, Sch20], which is its n-level coarsening and uncoarsening. At first glance this seems inherently sequential as on each level only one vertex is removed, however an efficient parallelization is possible. We propose an asynchronous n-level coarsening routine based on a representation of contractions as a forest, which imposes partial ordering constraints, from which we derive a parallel execution schedule. The contractions are performed on a novel highly intrusive, semi-dynamic hypergraph data structure with fine-grained locking. For uncoarsening we partition the contractions into independent batches that can be uncontracted in parallel, around which we subsequently perform parallel localized refinement, before proceeding with the next batch. The size of the batches interpolates between parallelism and refinement granularity. The uncoarsening schedule follows the reverse of the partial constraints from the coarsening phase as well as another technical condition. An intricacy in n-level uncoarsening is that values in the gain table must be updated or initialized as vertices are uncontracted.

The resulting algorithm achieves the same quality as sequential KaHyPar but in a parallel code. On medium size instances, we achieve around a factor 9 speedup over sequential KaHyPar with 10 cores, and on large instances around factor 25 self-relative speedup on 64 cores. In a second step, we propose a completely asynchronous uncoarsening with even better speedups (around 29) at the cost of worse partition quality due to interference between parallel localized searches.

Advanced Flow-Based Refinement and Parallelization. Our last contribution is an advanced refinement algorithm based on maximum flows and subsequently its parallelization. This is currently the strongest iterative improvement algorithm in terms of pushing the envelope on partition quality, but it is rather slow due to having to solve large flow problems. The basic version is restricted to partitions into two blocks, but can be applied to k-way partitions by scheduling refinement on block pairs. There is prior work on flow-based refinement [SS11, HSS19] which solves one flow problem on a reduced instance to potentially obtain a smaller cut.

We improve this by solving incremental maximum flow problems to trade off cut size for better balance in the same way as the from-scratch algorithms FlowCutter and FBB [HS18a, YW96]. Using only small increments, we find partitions with smaller cut than the previous approach. As solving many incremental flow problems is more expensive, we invest substantial engineering effort into the main bottleneck, the flow algorithms. We implement them directly on the hypergraph, which is faster than the previous approach of computing flows on an auxiliary graph. So much in fact that our approach becomes faster than the old one, *and* yields better partition quality. In the faster out of the two proposed configurations, flow-based refinement takes up substantially less time than local search, whereas in previous work it accounts for the largest share by far. Yet, both configurations find higher quality partitions than the previous flow-based refinement.

In a follow-up work, we parallelize our approach by plugging in a parallel push-relabel

algorithm [BBS15] for computing maximum flows, as well as parallelizing the scheduling. For the flow algorithm we propose two optimizations which speed up the execution by three orders of magnitude if combined, though this improvement is specific to our instance types. Furthermore, we describe and fix a bug in the existing parallel flow algorithm, and discuss intricacies when using push-relabel algorithms with incremental flow problems.

We provide three separate experimental evaluations, as these results are combined from three separate publications [GHW19a, GHSW20, GHS22]. In the first, we demonstrate the effectiveness of flow-based refinement as a postprocessing step to a simple but fast sequential multilevel partitioner [CA99]. In the second, we integrate our approach into sequential KaHyPar and improve the state of the art in terms of partition quality further, while also reducing the running time compared to the previous flow-based refinement. And finally, we demonstrate that our parallel version integrated in Mt-KaHyPar achieves the same partition quality as sequential KaHyPar with flow-based refinement, but is a factor 9.7 faster on 10 cores. Unfortunately, the speedups on larger core numbers are not as good as for our previous works. This is however expected, as flow algorithms are notoriously difficult to parallelize efficiently, and previous studies on flow algorithms [BBS15] show even worse speedups than ours.

Highly Engineered Implementation in a Unified Framework. Our two frameworks and their algorithmic components are bundled together in one framework `Mt-KaHyPar`. We make the C++ source code available online at `https://github.com/kahypar/mt-kahypar`. Additionally, flow-based refinement is available in a separate repository at `https://github.com/larsgottesbueren/WHFC`. The source codes are published under permissive open source licenses to welcome the use in future research projects on applications or algorithms. All codes are highly optimized and designed with extensibility in mind.

1.3 Thesis Outline

This dissertation is organized as follows. In Chapter 2 we introduce notation and concepts used throughout this thesis, as well as the methodology employed in the experimental evaluations. In Chapter 3 we discuss the existing literature on partitioning with a particular focus on parallel techniques and challenges these have to face. Subsequently we present our traditional parallel multilevel algorithm Mt-KaHyPar-D and its components in Chapter 4. We extend this to a deterministic version in Chapter 5. In Chapter 6 we then present our parallel n-level algorithm. Finally, in Chapter 7 we present our flow-based refinement, first the sequential then the parallel version. Each algorithms chapter contains an extensive experimental comparison with the existing state of the art, in both the sequential and parallel setting, as well as a thorough evaluation of the algorithmic components we employ. We conclude the dissertation in Chapter 8 with a summary of our results and outline directions for future work.

2 Preliminaries

This chapter introduces the fundamental concepts used in this thesis. We start with basic definitions and notation in Section 2.1. In Section 2.2 we introduce the parallel computation model we use in running time analyses and discuss algorithmic primitives employed throughout our work. In Section 2.3 we discuss the fundamentals of maximum flows. We conclude with the experimental setup in Section 2.4, as this will be reused throughout this thesis, with minor adaptations where necessary.

2.1 Notation and Definitions

By $[m]$ we denote the set $\{0, 1, \ldots, m-1\}$ for a positive integer m.

Hypergraphs. A *weighted hypergraph* $H = (V, E, c, \omega)$ is defined as a set of vertices V and a set of hyperedges/nets $E \subseteq 2^V$ with vertex weights $c : V \to \mathbb{R}_{>0}$ and net weights $\omega : E \to \mathbb{R}_{>0}$. The vertices of a net are called its *pins*. We extend c and ω to sets in the natural way, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e if $v \in e$. $\mathrm{I}(v)$ denotes the set of all incident nets of v. The set $\Gamma(v) := \{u \mid \exists e \in E : \{v, u\} \subseteq e\}$ denotes the neighbors of v. The *degree* of a vertex v is $\deg(v) := |\mathrm{I}(v)|$. The *size* $|e|$ of a net e is the number of its pins. Nets of size one are called *single-pin* nets. We call two nets e_i and e_j with different identifiers $i \neq j$ *identical* if they have the same pins. We use $p := \sum_{e \in E} |e|$ as the total number of pins, $n := |V|$ as the number of vertices and $m := |E|$ as the number of nets.

An *undirected* graph is a hypergraph where each net has size 2. For graphs, we use the terminology nodes and edges instead of vertices and nets/hyperedges, to distinguish more easily when hypergraphs and graphs are used jointly. A *directed* graph is a graph where

edges are ordered pairs instead of unordered pairs. To distinguish from undirected graphs we use the term arcs for directed edges. The *transpose* of a directed graph $G = (V, E)$ is the graph $\overline{G} = (V, \{(v, u) \mid (u, v) \in E\})$.

A hypergraph can be represented as an undirected bipartite graph $G^* = (V \cup E, \{(v, e) \mid e \in E, v \in e\})$ with the vertices and nets as nodes and an edge for each pin. This is called the *star expansion* or *bipartite graph representation*. A different representation is the *clique* expansion, where each net is replaced by all pair-wise edges between its pins.

Partitioning and Cuts. A *k-way partition* of a hypergraph H is a function $\Pi : V \to \{1, \ldots, k\}$. The blocks $V_i := \Pi^{-1}(i)$ of Π are the inverse images. We call Π *ε-balanced* if each block V_i satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1 + \varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some parameter $\varepsilon \in (0, 1)$. When ε is clear from the context we often just say balanced or feasible.

A 2-way partition is also called a *bipartition*. At times we represent a partition as the set of blocks, particularly for bipartitions (A, B) or (V_1, V_2).

For each net e, $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e) := |\Lambda(e)|$ is the number of different blocks connected by the net. Given parameters ε and k, and a hypergraph H, the *balanced hypergraph partitioning problem* is to find an ε-balanced k-way partition Π that minimizes the *connectivity metric* $(\lambda - 1)(\Pi) := \sum_{e \in E} (\lambda(e) - 1)\, \omega(e)$.

A net is called a *cut net* if $\lambda(e) > 1$ and the *cut* of a partition is set of all cut nets. A *boundary vertex* is a vertex that is incident to at least one cut net. The number of pins of a net e in block V_i is denoted by $\Phi(e, V_i) := |e \cap V_i|$. Given a k-way partition Π, the *quotient graph* $\mathcal{Q} := (\Pi, E_\Pi := \{(V_i, V_j) \mid \exists e \in E : \{V_i, V_j\} \subseteq \Lambda(e)\})$ is a graph with the blocks as the nodes and it contains an edge for each block pair with non-empty cut.

Throughout this dissertation the terms *partition quality* and *solution quality* refer to the value of the objective function of a partition, assuming the objective function is clear from the context. Another common objective function is the *cut-net metric* $\sum_{e \in E : \lambda(e) > 1} \omega(e)$. On graphs or bipartitions the connectivity metric is equivalent to the cut-net metric.

A *clustering* is a partition without a restriction on the number of blocks or their weight. In this context a block is also called *cluster* and sometimes *community*. A *move* (u, s, t) is the operation of assigning a new block t to a vertex u, moving it out of its old block s and into the new block t. The *gain* of a move is the absolute reduction in the objective function we currently consider, for example connectivity (reduction for minimization, increase for maximization). A move is called *feasible* if the resulting partition is balanced.

Contraction. *Contracting* a vertex pair (u, v) removes v from all nets $e \in I(u) \cap I(v)$ and replaces v with u in all nets $e \in I(v) \setminus I(u)$. The weight of u becomes $c(u) := c(u) + c(v)$. We refer to u as the *representative* and v as the *contracted vertex*, i.e., the pair is ordered. We also say v is contracted onto u, or just v is contracted when the description does not need the representative. Vertices that are not yet contracted are called *active*. *Uncontraction* is the reverse operation of contraction.

Contracting a clustering Π means contracting all vertices assigned to the same cluster. For correctness the order can be arbitrary, though an implementation will not contract them one by one.

Balance Constraint. The balance constraint we use follows the classic definition used throughout the literature. However, it has some flaws with weighted vertices. Finding any balanced partition is NP-hard since the problem can be reduced to scheduling variable length jobs on identical parallel machines with bounded capacity [GJ79]. Furthermore, if there is even a single vertex v with $c(v) > L_{\max}$ there is no solution. For this particular case, a reasonable strategy proposed by Caldwell et al. [CKM00b] is to place too heavy vertices in a separate block each, remove them, and partition the remaining hypergraph with a reduced k.

A different approach is to use $\max(L_{\max}, \frac{c(V)}{k} + \max_{v \in V} c(v))$ as the upper bound on the block weights. With this, a feasible partition can always be found through a simple greedy algorithm, but the imbalance parameter ε is potentially no longer meaningful. Even with $\varepsilon = 0$ the block weights are allowed to deviate by $\max_{v \in V} c(v)$.

Heuer, Maas and Schlag [HMS21] propose to use $(1 + \varepsilon)$ times the result of a 4/3-approximation algorithm for job scheduling/load balancing [Gra69] as the balance constraint. This approach is fairly unorthodox and inelegant, but why should a partitioning algorithm do better than a load balancing algorithm.

So far there is no commonly accepted way to formulate a balance constraint that gracefully deals with this problem. The benchmark instances we consider in this dissertation are unweighted, and only become weighted through coarsening where reasonable measures to favor uniform weights are in place, such that the original simple definition is sufficient.

Hypergraph Representation. We consider two common graph representations, namely adjacency arrays (also called compressed sparse row or CSR) and adjacency lists. The adjacency lists data structure has an array of size n storing a vector of neighbors and associated arc weights for each node. In the CSR representation the neighbor vectors are stored in one common array, and a second offsets array points at the first entry of each node. The next node's entry denotes the end of the range of the former. The CSR representation can be constructed in parallel from the adjacency lists representation by performing a prefix sum over the degrees, and then writing the neighbors of each node to the destination starting at the prefix sum value. We can represent a hypergraph with two graph representations mapping vertices to incident nets and mapping nets to their pins.

Paths, Cycles and Forests. A *(directed) path* is a sequence $\langle v_1, v_2, \ldots, v_{r-1}, v_r \rangle$ of nodes such that two consecutive nodes v_i, v_{i+1} are connected by an edge $\{v_i, v_{i+1}\}$ (or an arc (v_i, v_{i+1})). A path is a *cycle* if additionally the edge or arc (v_r, v_1) exists. A *(strongly) connected component* $C \subset V$ is an inclusion-maximal set of nodes such that for each $u, v \in C$ there is a (directed) path from u to v and from v to u.

An undirected graph is called a *forest* if it is does not contain a cycle, and a forest is called a *tree* if it is connected. A *rooted forest* is a directed graph whose underlying undirected graph is a forest, with a designated set of root vertices such that all arcs point towards the root of their tree.

A natural way to represent a rooted forest is an array of *parent pointers* $\text{rep}[0..n]$, where each node stores the endpoint of its outgoing arc, and roots (with no outgoing arc) point at themselves $\text{rep}[v] = v$. The *ancestors* of v are the nodes on the unique path towards a root. The *children* of v are all nodes $w \in V \setminus v$ with $\text{rep}[w] = v$. Two nodes are *siblings* if they have the same parent.

2.2 Parallelism

Our algorithms are designed for shared-memory multi-core machines. In this section we discuss the particular machine model we use for algorithm analysis, as well as a variety of parallel programming and algorithmic patterns we use in the implementations.

2.2.1 Computational Model

The machine model we analyze our algorithms in consists of multiple RAM machines (cores/threads) working asynchronously on a shared memory. Read access always takes constant time. Concurrent write accesses to the same memory location exhibit contention, thus taking $\mathcal{O}(t)$ time in the worst case with t participating cores. As opposed to the classic PRAM model which operates in synchronous lockstep, we deem it important to include asynchronicity as found in modern CPUs in the model. Additionally, it emphasizes the cost of synchronization, e.g., via locking. We assume a basic binary fork-join model (or spawn-sync), where a thread can spawn another thread, or wait for the completion of its children with a join. This is done to highlight that starting a parallel computation comes with overhead. This simple model suffices to implement common parallel programming primitives such as parallel for-loops or task parallelism by recursively forking. It can be implemented efficiently with a thread pool and a work-stealing scheduler on modern multi-core machines. For the sake of simplicity we do not consider cache effects in the model.

We use the work-depth model [CLRS01] to analyze running times of parallel algorithms. In the literature depth is also often called span. The work of an algorithm is the running time if run on a single core, and the depth is the time if run on an infinite number of cores. More formally, work is the total number of instructions performed and depth is the longest sequential dependency chain in the computation. With this model we can analyze parallel algorithms independent of the specific number of threads used, but still understand how well the algorithm is parallelized. The goals to achieve are work efficiency (same bound as the best sequential algorithm) and poly-logarithmic depth. For example, a parallel loop with n iterations has a $\log(n)$ additive term in the depth, to account for loop control, regardless of the work performed in an iteration.

Finally, we augment the model with thread-local storage. Unfortunately initialization costs that are done once per thread (but not on each task) do not fit well into the work-depth model. Attributing initialization costs to each task is unrealistic in terms of behavior in practice. Therefore, we often omit these costs in the analysis, but address the issue with an additional comment.

2.2.2 Concurrent Writes and Atomic Instructions

Multiple threads writing to the same memory location concurrently constitutes a data race. However, the race is *benign* if they all write the same value.

Modern CPUs support *atomic instructions* as a fast way to correctly manipulate the same memory location in parallel without resorting to locks. With little contention atomic instructions are almost as fast as non-atomic instructions.

We use the atomic `CompareAndSwap`, `FetchAndAdd` and `TestAndSet` functions. The function `FetchAndAdd(&x, Δ)` takes a memory address x, atomically adds Δ to the value at x and returns the value immediately prior to its execution. In pseudocode we use the notation $y \underset{\text{atomic}}{\texttt{+=}} \Delta$ for this function, where y is the data at address x. Similarly `AddAndFetch` performs the update atomically, but returns the value after the update. There are more such incremental update functions for other arithmetic operations, e.g., bitwise logic manipulation such as OR, XOR, AND, NAND, etc., but interestingly not for multiplication. To the best of our knowledge, we only use XOR next to addition and subtraction.

`CompareAndSwap(&x, expected, new)` takes a memory address x, an old expected value, and a new value. If x contains the expected value, then `CompareAndSwap` replaces it with the new value and returns true; otherwise it does not perform the store and returns false. `TestAndSet` works similarly to `CompareAndSwap` but the data must be a boolean and can only be raised from false to true.

One frequently used application of `TestAndSet` is spinlocks. Furthermore, it can be used to assign responsibility for a computation task or object to a unique thread. If multiple threads test and set on a variable, only one succeeds. To reduce write contention with `TestAndSet` and `CompareAndSwap`, we first check whether the memory location already has the desired value and if so do not perform the atomic instruction.

Modern CPUs execute instructions out of order to hide memory latencies and improve bus throughput. To achieve *correct* atomic consistency, the programmer must specify a memory ordering parameter to these atomic functions. Looser constraints yield better performance. For `FetchAndAdd` we use *relaxed* consistency, which means arbitrary reordering is possible. This is the loosest ordering one can specify. For acquiring a lock and resource assignment we need *acquire* consistency, which means instructions after the acquire cannot be hoisted before it. Releasing a lock only needs relaxed consistency.

2.2.3 Parallelization Libraries

There is a variety of parallel programming libraries, with Intel TBB, OpenMP and Cilk being the most common. We use TBB for our implementations since it has a very simple programming interface. Its parallel primitives are functions that take C++ lambdas to perform the work. Furthermore, it has a cache-aligned and a scalable memory allocator, thread-local storage, and contains implementations of common algorithms such as parallel prefix sums, reductions, and sorting. TBB also has implementations of common data structures such as a concurrent queue or a vector, but these are too inefficient because they have to be more general than we need in most cases.

2.2.4 Parallel Algorithmic Primitives

Reduction and Prefix Sum. The reduce and prefix sum operations both take a sequence of elements A of size n, and a binary associative operator such as + as input. Reduce computes the sum $\sum_{i=0}^{n-1} A[i]$. Prefix sum comes in two flavors: exclusive and inclusive. Exclusive prefix sum outputs a sequence S of size $n+1$ where $S[j] = \sum_{i=0}^{j-1} A[i]$ is the partial sum up to but excluding j. Both operations take $\mathcal{O}(n)$ work and $\mathcal{O}(\log(n))$ depth, assuming the operator takes constant time. Reduce can be implemented in a single pass over the data, whereas prefix sum requires two (upward then downward sweep). Prefix sums are a classic tool in parallel algorithms, particularly for data arrangement problems such as constructing a CSR from an edge list, filtering and packing a sequence, or counting sort.

Integer Sorting. The input is a sequence A of integers of bounded size $[K]$, and the output is the sorted stable permutation of A and an array of offsets pointing to the first element of each key. The sequential counting sort algorithm takes $\mathcal{O}(n+K)$ time and works as follows. Compute a histogram $h : [K] \to \mathbb{N}_0$ mapping a key to its frequency in A. An exclusive prefix sum over h yields an array of offsets $d : [K+1] \to [n]$ (the destinations) such that $d[i]$ is the position of the first element in the sorted sequence with key i and $d[i+1]$ is one past the last. In a second pass over A we assign the elements to the output range, incrementing $d[i]$ to obtain the position for the next element with key i. Finally, rotating d by one position to the left restores the offsets required for the output. We avoid the rotation by counting key i at $h[i+1]$ instead of $h[i]$ and thus also use $d[i+1]$ for the sorted output.

Computing the histogram can be parallelized by splitting A into t chunks of size n/t (parameter), computing a histogram for each sub-range, and then summing up the counts for each key. The histogram of one chunk takes $\mathcal{O}(n/t)$ time. The loop over the chunks takes an additive $\mathcal{O}(\log(t))$ depth for the loop control, and a combined $\mathcal{O}(n)$ work for all histograms.

Summing up the counts of one key takes $\mathcal{O}(\log(t))$ depth with a prefix sum (iterating over chunks). This yields chunk-specific offsets that are combined with d to determine the output positions. The loop control over the keys takes $\mathcal{O}(\log(K))$ depth, and summing the counts of all keys takes $\mathcal{O}(tK)$ work.

The prefix sum for d takes $\mathcal{O}(\log(K))$ depth and $\mathcal{O}(K)$ work and is thus dominated by the previous step. Writing the output takes the same work and depth as the histogram step. Overall this results in $\mathcal{O}(tK + n)$ work and $\mathcal{O}(n/t + \log(t) + \log(K))$ depth. Choosing t offers a trade-off between work and depth. A good choice is $t = n/K$ to maintain work-efficiency, which then results in $\mathcal{O}(K + \log(n/K))$ depth; though in practice we simply use the number of threads available.

GroupBy and Aggregate. A common algorithmic pattern is to group a set of elements by an associated key such that they are consecutive in memory and then aggregate over elements of the same key. We use this in several places, for example to contract a graph based on a clustering [HSWZ18], to remove identical nets [DKÇ13], or when a computation on elements of the same key has to be sequential but we want to leverage parallelism over the keys [GH21].

There is a vast amount of literature from the database community who identify sorting and hashing as the two main approaches [Mül+15]. Since the order between keys is not important, a *semi-sort* which only brings elements with the same key together suffices [GSSB15]. Which approach gives better performance depends on the context. For graph contraction the keys are the cluster IDs which are small, so counting sort works best. A great benefit of counting sort is that it already gives the start and end points of the ranges. For deterministic update aggregation (Algorithm 5.3 and Section 5.1) we need to establish a deterministic order between elements with the same key, so plain parallel sorting works well. For detecting identical nets (Algorithm 4.3) a hybrid approach with hashing to obtain parallelism and then sequentially sorting elements in the same hash bucket to resolve collisions works better than sorting.

Duplicate Detection. Another commonly used pattern is de-duplicating elements (with an associated integer ID) by using a bitvector to check if an element has been generated before. For example, Breadth-First-Search (BFS) scans the neighbors of nodes in the current layer and inserts them into a container for the next layer, ensuring that each node is inserted at most once and only in one layer. In parallel, this check can be implemented in atomically consistent fashion with a `TestAndSet` instruction. To avoid resetting each entry in the bitset, we employ a well-known time-stamping trick. We have an array of timestamps and a current time, such that all elements with a timestamp older than the current time are considered as not seen. In parallel, this can be done with an atomic `CompareAndSwap`. For repeated searches we can implement the timestamping trick directly on the distance labels, starting the next BFS at one past the highest distance of the previous BFS.

Applications of this pattern are parallel BFS, parallel gain recalculation (Algorithm 4.9), localized FM (Algorithm 4.11), n-level batch uncontractions (Section 6.2.3), and maintaining active nodes in parallel push-relabel (Algorithm 7.6).

In the various clustering algorithms considered as sub-routines in this thesis (for example Algorithms 4.2, 4.4), we use a similar trick to enumerate neighbor clusters of a node. To find

a good cluster to join, we construct cluster ratings in a map/array, using a rating of zero to indicate that a cluster has not been seen before and we can add it to the list. To implement this in parallel, we can check the result of `FetchAndAdd` when aggregating ratings.

2.3 Maximum Flows

A flow network is a symmetric, directed graph with two disjoint non-empty *terminal* node sets $S, T \subsetneq V$, the source and sink node set, as well as a capacity function $\text{cap}\colon E \to \mathbb{R}_{>0}$. For a non-symmetric graph, we assume zero capacity on the missing reverse arcs. A flow is a function $f\colon E \to \mathbb{R}$ subject to the *capacity constraint* $f(e) \leq \text{cap}(e)$ for all arcs e, *flow conservation* $\sum_{(u,v)\in E} f(u,v) = 0$ for all non-terminal nodes v, and *skew symmetry* $f(u,v) = -f(v,u)$ for all arcs (u,v). The *value* of a flow $|f| \coloneqq \sum_{s\in S,(s,u)\in E} f(s,u)$ is the amount of flow leaving S. The *residual capacity* $r_f(e) \coloneqq c(e) - f(e)$ is the additional amount of flow that can pass through e without violating the capacity constraint. If $r_f(e) = 0$ the arc is *saturated*. The residual network with respect to f is the directed graph $\mathcal{N}_f = (V, E_f)$, where $E_f \coloneqq \{e \in E | r_f(e) > 0\}$ contains all non-saturated arcs. A node v is *source-reachable* if there is a path from S to v in $\mathcal{N}_f$, it is *sink-reachable* if there is a path from v to T in $\mathcal{N}_f$. We denote the source-reachable and sink-reachable nodes by S_r and T_r, respectively. An *augmenting path* is an S-T path in $\mathcal{N}_f$. The flow f is a *maximum flow* if $|f|$ is maximal of all possible flows. This is the case if and only if there is no augmenting path in $\mathcal{N}_f$. An S-T edge cut is a set of edges whose removal disconnects S and T. The value of a maximum flow equals the weight of a minimum-weight S-T edge cut [FF56]. The *source-side cut* consists of the arcs from S_r to $V \setminus S_r$ and the *sink-side cut* consists of the arcs from T_r to $V \setminus T_r$. The bipartition $(S_r, V \setminus S_r)$ is induced by the source-side cut and $(V \setminus T_r, T_r)$ is induced by the sink-side cut. Note that $V \setminus S_r \setminus T_r$ is not necessarily empty. We also call S_r and T_r the *cutsides* of a maximum flow. The source-side cut can be computed by exploring the nodes reachable from the source via residual arcs (for example via BFS), and analogously the sink-side cut from the sink in the transpose of $\mathcal{N}_f$.

Augmenting Path Algorithms. Ford and Fulkerson [FF56] developed the first known maximum flow algorithm. It is based on repeatedly finding augmenting paths and pushing the smallest residual capacity on the path (bottleneck capacity), until there is no augmenting path. Edmonds and Karp [EK72] achieve polynomial running time $\mathcal{O}(nm^2)$ by using shortest augmenting paths, which are computed via BFS in the residual network. Dinitz [Din70] achieves $\mathcal{O}(n^2m)$ time by augmenting multiple shortest paths of the same length. The *layered network* is the sub-graph of $\mathcal{N}_f$ with all arcs that are on a shortest S-T path. Then flow is augmented along shortest paths until there is no augmenting path in the layered network, which is implemented by DFS with backtracking. It suffices to compute just the distance labels via BFS instead of constructing the layered network, and then follow residual arcs that lead to one higher distance label.

Preflow Algorithms. The *push-relabel* [GT88] algorithm by Goldberg and Tarjan stores a distance label $d(u)$ and an excess value $\text{exc}(u) := \sum_{v \in V} f(v,u)$ for each node. It maintains a *preflow* [Kar74] which is a flow where the conservation constraint is replaced by $\text{exc}(u) \geq 0$. A node $u \in V$ is *active* if $\text{exc}(u) > 0$. An arc $(u,v) \in E$ is *admissible* if $r_f(u,v) > 0$ and $d(u) = d(v) + 1$. A *push*(u,v) operation sends $\Delta = \min(\text{exc}(u), r_f(u,v))$ flow over (u,v). It is applicable if u is active and (u,v) is admissible. A *relabel*(u) operation updates the distance label of u to $\min(\{d(v) + 1 \mid r_f(u,v) > 0\})$, which is applicable if u is active and has no admissible arcs.

The distance labels are initialized to $\forall u \in V \setminus S\colon d(u) = 0$ and $\forall s \in S\colon d(s) = n$. Furthermore all arcs (s,v) are saturated by pushing all residual capacity from the source. Performing applicable push and relabel operations (in arbitrary order) gives an $\mathcal{O}(n^2 m)$ algorithm.

Efficient variants of push-relabel use the *discharge* routine, which repeatedly scans the arcs of an active node until its excess is zero. During a scan, all admissible arcs of the active node are pushed. If there is excess left after the scan, there are no admissible arcs left, so the node is relabeled and another scan is started. The appropriate label is tracked during the scan. Discharging active nodes in FIFO order results in an $\mathcal{O}(n^3)$ time algorithm. Another processing order that works well is to discharge the node with highest label, which achieves $\mathcal{O}(n^2\sqrt{m})$. Dynamic trees [GT88] reduce the theoretical complexity to $\mathcal{O}(nm \log(\frac{n^2}{m}))$ by pushing flow along a path instead of a single arc, but are not useful in practice, due to the following optimizations.

The *global relabeling* heuristic [CG97] frequently assigns exact distance labels by performing a reverse BFS from the sink, to reduce the amount of necessary relabel work in practice. The *gap* heuristic [CG97] works well with the highest-label order, but has little impact when used with the FIFO order. If a label l has no nodes with $d(u) = l$, then any node with $l < d(u) < n$ can be relabeled to $n + 1$.

Note that preflows already induce minimum sink-side cuts, so if only a minimum cut is required, the algorithm can already stop once no active nodes with distance label $< n$ exist. The preflow can be converted into an actual flow by continuing until no active node remains (pushing flow back to the source), or by performing flow decomposition [CG97].

Maximum Flows On Hypergraphs. Lawler [Law73] proposes a transformation to compute a minimum S-T cut on a hypergraph via maximum flow on a graph. It combines the star expansion with a standard technique to compute node separators (on the nets) via edge cuts by splitting a node into two nodes connected by an arc with the desired capacity. The *Lawler network* of a hypergraph $H = (V, E, c, \omega)$ is a directed graph consisting of V and two nodes $e_{\text{in}}, e_{\text{out}}$ for $e \in E$ as the nodes, and arcs $\forall u \in V, e \in \text{I}(u) : (u, e_{\text{in}}), (e_{\text{out}}, u)$ with infinite capacity, and bridge arcs $\forall e \in E : (e_{\text{in}}, e_{\text{out}})$ with capacity $\omega(e)$. The reverse arcs all have capacity zero. A minimum S-T cut in the Lawler network only consists of bridge arcs and thus directly corresponds to one in H. Via the Lawler network, all notions of maximum flow on graphs translate naturally to hypergraphs.

Table 2.1: Machines used in this dissertation and their properties. The cores column displays sockets x cores on each socket, so 2x20 means 40 cores overall spread over two sockets.

	frequency	boost	RAM	L3	cores	model	arch
A	2.1 GHz	3.9 GHz	96 GB	27.5 MB	2x20	Xeon Gold 6230	Cascade Lake
B	2.25 GHz	3.4 GHz	1 TB	256 MB	2x64	EPYC 7742	Zen 2
C	2.0 GHz	3.35 GHz	1 TB	256 MB	1x64	EPYC 7702P	Zen 2
D	2.6 GHz	3.3 GHz	64 GB	20 MB	2x8	Xeon E5-2670	Sandy Bridge
E	2.7 GHz	3.5 GHz	256 GB	20 MB	2x8	Xeon E5-2680	Sandy Bridge

2.4 Experimental Methodology

In this dissertation we conduct a thorough experimental analysis of the proposed algorithms. We consider two main types of evaluations: a horse-race comparison with other algorithms, and analysis of our proposed components. Horse-race comparisons are good at exploring the landscape of different trade-offs in terms of connectivity and running time, and identifying which partitioner works best depending on which criterion is valued most. For most cases, the horse race comparison is a great choice, and it works even for testing the impact of a component or parameter. However in some cases we might want to dig deeper, to gain a better understanding, particularly when there is interaction between different components, as is the case in the refinement stage. Therefore, we additionally analyze which refinement algorithm contributed how much of the improvement from the initial partition, and how well our techniques for accurate gains work. Finally, we analyze the scaling behavior of our codes with increasing numbers of cores.

In the following, we describe the experimental setup, from machines and benchmark sets, to the various plots we use to evaluate the data.

2.4.1 Machines

Table 2.1 shows the different machines used throughout this dissertation. Machines A and D are part of a cluster at KIT, which we use for running slow and sequential partitioners. Machine D is used for the sequential versions of flow-based refinement. This cluster was decommissioned in 2020, thus machine A is used for parallel flow-based refinement, Mt-KaHyPar-D and Mt-KaHyPar-Q. Faster parallel partitioners are also run on machines B and C. Machine B is the main machine where all comparisons are done as well as the speedup experiments for Mt-KaHyPar-D. Due to excessive running time (6 weeks) we decided not to repeat the speedup experiments for Mt-KaHyPar-Q and parallel flow-based refinement, but instead take the existing measurements taken for the conference publication on machine C. Furthermore, all data for the asynchronous n-level version were collected on machine C. Finally, machine E is used to conduct parameter tuning experiments, to keep machine B free for the main experiments. The parameter tuning instances are too small to profit from more

than 16 cores anyways. The component comparisons are also done on machine B.

To ensure fairness, comparisons between different algorithms are always done on the same machine; and number of threads if both are parallel.

2.4.2 Benchmark Sets

We use five benchmark sets in our experiments, two main benchmark sets (A and B) of real-world hypergraphs compiled from a variety of application sources, two parameter tuning sets (C, D) and a set of graphs (E) to compare Mt-KaHyPar with dedicated graph partitioners. All experiments use a default imbalance parameter of $\varepsilon = 0.03$. We use the term *instance* to refer to a combination of hypergraph and number of blocks k.

Sources. All hypergraphs used in the experimental evaluation are derived from four sources encompassing three application domains: the ISPD98 VLSI Circuit Benchmark Suite [Alp98], the DAC 2012 Routability-Driven Placement Contest [Vis+12] (also VLSI), the SuiteSparse Matrix Collection [DH11], and the 2014 SAT Competition [BDHJ14]. VLSI circuits are transformed into hypergraphs by converting the net-list of each circuit element into a set of hyperedges. Sparse matrices are translated to hypergraphs using the row-net model [CA99]. SAT inputs are converted to three different hypergraph representations: *literal*, *primal*, and *dual* [MP14, PM07]. In the primal model each clause is a net, and each variable is a vertex, with variable-in-clause occurrences as the pins. The literal model further distinguishes negated and non-negated variables, and thus has twice the number of vertices as the primal version but the same number of nets and pins. On the other hand, in the dual model variables are nets and clauses are vertices. All hypergraphs have unit vertex and net weights. Figure 2.1 shows basic properties of the hypergraphs in set A and B such as number of vertices, nets, pins, as well as median/maximum net size and degree.

Set A: Medium Size Hypergraphs. Set A is the main benchmark set compiled by Sebastian Schlag for his works on sequential hypergraph partitioning [Sch20]. It contains 488 medium size hypergraphs and is used for comparison with sequential and slow partitioners. We use $k \in \{2, 4, 8, 16, 32, 64, 128\}$, which yields a total of 3416 instances, and we use 10 seeds which makes 34160 runs per algorithm. Each run has a time limit of 8 hours. The parallel algorithms are run with 1, 10, 20, and 40 cores on machine A, whereas the experiments on sequential flow-based refinement were still done on machine D.

Set B: Large Hypergraphs for Fast Algorithms. Set B is our main benchmark set for comparison with fast partitioning algorithms and for analyzing speedup behavior. It consists of 94 large hypergraphs comprised from the same sources as set A, but around an order of magnitude larger as shown by Figure 2.1. We included the 24 largest SAT instances (8x3 to keep all three representations) from set A as well as 18 new, even larger ones. From the SuiteSparse Matrix collection we sampled 42 random entries with at least 15 million

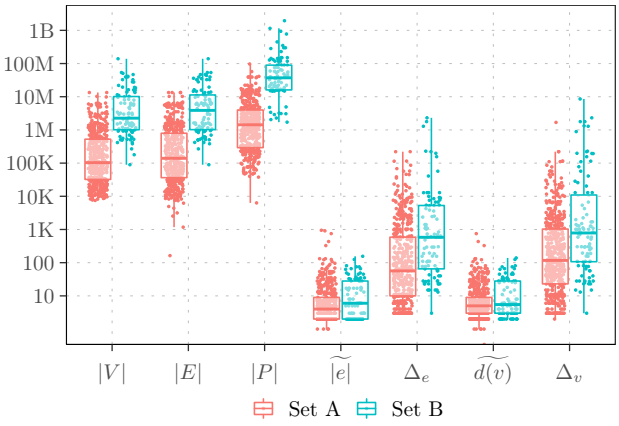

Figure 2.1: Basic properties of benchmark sets A and B: number of vertices $|V|$, nets $|E|$, pins $|P|$ as well as median/maximum net size $\widetilde{|e|}, \Delta_e$ and degree $\widetilde{d(v)}, \Delta_v$.

non-zeroes. Further, none of the ISPD98 hypergraphs are included since they are too small, but we included all 10 DAC hypergraphs (even though they are the smallest in the set) in order to have some representation for VLSI inputs. We use $k \in \{2, 4, 8, 16, 32, 64\}$, which yields a total of 564 instances, and we use 5 seeds per instance which makes 2820 runs per algorithm. Each run has a time limit of 2 hours. For Zoltan and Mt-KaHyPar-D we also perform a comparison with large values of $k \in \{512, 1024, 2048\}$, since this is an area these algorithms were not designed for, but we exclude others since they are too slow. The scalability experiments for Mt-KaHyPar-D and Mt-KaHyPar-SDet (deterministic) are done on machine B with 1,2,4,8,16,32,64 cores (and 128 for deterministic). All horse-race comparisons are done on machine B with 64 cores. The scalability experiments for Mt-KaHyPar-Q (n-level) and Mt-KaHyPar-D-F (parallel flow-based refinement) were done on machine C with a reduced parameter set of $k \in \{2, 8, 16, 64\}$ and 3 seeds in order to save running time. All experiments on the asynchronous n-level version were done with this setup.

Set C and D: Parameter Tuning. We use two parameter tuning sets. The first, set C is a subset of set A that was assembled to evaluate the memetic algorithm in KaHyPar [ASS18b], and is also called set C in [Sch20]. It contains 4 DAC hypergraphs, 10 ISPD98 hypergraphs, 18x3 SAT hypergraphs and 32 SPM hypergraphs. Since these were chosen to adequately represent the full set A but to reduce running time, this set works well as a parameter tuning set. Moreover, it is almost disjoint from set B (except DAC), to ensure the parameters are not tuned specifically to these instances when evaluating Mt-KaHyPar on set B. We use this set to tune all parameters for Mt-KaHyPar-D, performing the experiments on machine E with

16 threads, with $k \in \{2, 4, 8, 16, 32, 64\}$ and five seeds.

The second set D is a subset of set B, containing the 10 smallest sparse matrics and 15 smallest SAT instances. It is used to tune the batch size parameter for the batch-synchronous n-level variant. To achieve decent parallelism, we need larger instances than set C, but not too large so that we can still fit them in the 96 GB memory of machine A.

Set E: Graphs. Finally, we evaluate Mt-KaHyPar-D against dedicated graph partitioners on 53 plain graph instances. The benchmark set is combined from two sets, taking 42 graphs from Akhremtsev [Akh19] (all but the four largest graphs due to memory constraints) and 11 additional graphs from Seemaier [Got+21]. These experiments are run on machine B with 64 cores, $k \in \{2, 4, 8, 16, 32, 64\}$ and five seeds per instance.

2.4.3 Source Code

The C++ implementations of our algorithms are available as open-source code, either under `https://github.com/kahypar/mt-kahypar/` for the main multilevel partitioning framework, under `https://github.com/larsgottesbueren/WHFC/` for the flow-based refinement with FlowCutter, and under `https://github.com/kahypar/kahypar` for sequential KaHyPar equipped with our flow-based refinement. The code is well documented and easy to adapt for future use in research, and has already been used by several students. Each configuration parameter can be set via a command-line argument or a configuration file. For each algorithm/publication we include preset files for the specific configurations used in that publication.

All of our parallel algorithms are part of the Mt-KaHyPar framework, and thus start with this prefix. The default configuration with approximately $\log(n)$ levels is called Mt-KaHyPar-D, and the quality configuration with approximately n levels is called Mt-KaHyPar-Q. The deterministic version is called Mt-KaHyPar-SDet for speed and determinism, as it uses only label propagation refinement, and we additionally consider a non-deterministic speed configuration Mt-KaHyPar-S. Finally, we add a suffix -F if flow-based refinement is enabled, e.g., Mt-KaHyPar-D-F for Mt-KaHyPar-D with flow-based refinement.

2.4.4 Competing Codes

As we perform horse-race comparisons, we must determine a set of state-of-the-art codes to compare against. More detailed algorithmic choices of these are described in Chapter 3, whereas in this section, we focus on implementations and their issues.

On benchmark set B, we compare our algorithms with BiPart[1] [MABP21], which is a recent shared-memory parallel multilevel partitioner focused on determinism; with Zoltan[2] [Dev+06],

[1] `https://github.com/IntelligentSoftwareSystems/Galois/tree/master/lonestar/analytics/cpu/bipart`
[2] `https://github.com/sandialabs/Zoltan`

a distributed memory partitioner developed at Sandia labs; and PaToH-D, which is the default configuration of the sequential partitioner PaToH [CA99] version 3.3[3]. It is the only sequential partitioner fast enough to be included on set B.

We could not include Parkway [TK04] as the publicly available implementation[4] crashes on all of the inputs we tested. We invested a day's work to fix numerous obvious bugs, but could not reach a state that works.

Similarly we were unable to include the Social Hash Partitioner (SHP) [Kab+17a] from Facebook. The code is only available as a pull-request/issue[5] for the Giraph graph processing framework, which must be combined with additional patches, according to personal correspondence with one of the authors. Even then the compilation fails, as it triggers package downloads which simply time out. Since we cannot run their code on our instances, we perform a comparison on the limited set of publicly available instances from their paper, using their reported numbers.

On the graph benchmark set E, we compare Mt-KaHyPar-D with Mt-KaHiP[6] [ASS18a] in the fast-social configuration, a shared-memory parallel graph partitioning algorithm that inspired our refinement algorithms; Mt-Metis[7] [LK13, LK16], a shared-memory parallel version of the well-known Metis framework using version 0.72 with hill-scanning enabled; as well as KaMinPar[8] [Got+21], a recent shared-memory parallel algorithm that implements the novel deep multilevel approach and uses only label propagation for refinement.

On benchmark set A, we have three competitors that are always included: PaToH-D and PaToH-Q (quality configuration, uses 3 full and 3 shorter V-cycles as well as higher filter thresholds), as well as hMetis-R [KAKS99] version 2.0, which is the recursive bipartitioning variant of hMetis[9].

The first two evaluations in Chapter 7 still consider KaHyPar-MF [HSS19], the version of KaHyPar with the previous flow-based refinement as it constituted the state of the art at that time. This is replaced with our sequential version KaHyPar-HFC [GHSW20] in all later comparisons (the other chapters).

Furthermore, we evaluated several algorithms that we then deemed as outperformed, such that they could be removed in later comparisons, which we list here. The first is hMetis-K [KK00], the direct k-way variant of hMetis, as its quality is inferior to hMetis-R and it often produces imbalanced partitions. Mondriaan[10] [VB05] is a multilevel recursive bipartitioning algorithm that is always outperformed by PaToH-D. Zoltan-AlgD[11] [SCS19a] enhances the Zoltan partitioner with coarsening based on algebraic distances (a vertex-similarity metric),

[3] https://faculty.cc.gatech.edu/~umit/software.html
[4] https://github.com/parkway-partitioner/parkway
[5] https://issues.apache.org/jira/browse/GIRAPH-1131
[6] https://github.com/KaHIP/mt-KaHIP
[7] http://glaros.dtc.umn.edu/gkhome/metis/metis/download
[8] https://github.com/KaHIP/KaMinPar
[9] http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download
[10] https://webspace.science.uu.nl/~bisse101/Mondriaan/
[11] https://github.com/rsln-s/aggregative-coarsening-for-multilevel-hypergraph-partitioning

which is always outperformed by PaToH-Q. And finally HYPE[12] [May+18], a flat algorithm based on neighborhood expansion, which is faster than PaToH-D by a negligible margin and has abysmal solution quality.

In chapters 4, 6 and 7 (last evaluation only), we additionally consider a version of sequential KaHyPar without flow-based refinement, called KaHyPar-CA [HS17] for community-aware coarsening. It has been previously shown that this version outperforms hMetis-R, yet we keep hMetis around, as it provides a level of quality between KaHyPar and PaToH.

2.4.5 Aggregates

While we make sure to present all data points using any of the following plots, we do aggregate data to give a quick overview. For running times and connectivity values on the same instance, we use the arithmetic mean to aggregate over different seeds. To aggregate running times across different instances we use the geometric mean since it is not as sensitive to outliers and is a better choice for skewed data. Again, we stress that any aggregate is insufficient to give a full picture of the data but it is often helpful to establish an initial picture. For example, the algorithm ranking provided by geometric mean running times is usually accurate, but we of course cannot reasonably estimate the running time on a particular instance from it. We refrain from making summarizing quantitative statements regarding partition quality (e.g., has $x\%$ lower connectivity), and instead make qualitative statements using attributes such as *barely, slightly, moderate, significant, substantial* or *enormous*, listed in increasing order of intensity.

2.4.6 Performance Profiles

To compare the solution quality of different algorithms, we use *performance profiles* [DM02]. Let $\mathcal{A}$ be the set of algorithms we want to compare, $\mathcal{I}$ the set of instances, and $q_A(I)$ the objective function value of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$; in our case the arithmetic mean connectivity of the repeated runs with different seeds. The performance ratio

$$r(A, I) = \frac{q_A(I)}{\min\{q_{A'}(I) \mid A' \in \mathcal{A}\}}$$

indicates by what factor A deviates from the best solution on instance I. In particular, algorithm A found the best solution on instance I if $r(A, I) = 1$. The performance profile

$$\rho_A \colon [1, \infty) \to [0, 1], \tau \mapsto \frac{|\{i \in \mathcal{I} \mid r(A, I) \leq \tau\}|}{|\mathcal{I}|}$$

of A is the fraction of instances for which it is within a factor of τ from the best solution computed by the pool of algorithms $\mathcal{A}$. Therefore, the goal is to achieve higher fractions at lower τ-values than the competitors. Runs that did not finish within the time limit, resulted

[12] https://github.com/mayerrn/HYPE

in a balance violation or crashed are excluded. If this concerns all runs of an algorithm on an instance, we report the corresponding fractions as the steps at special symbols (❋ for timeouts and crashes, ✘ for balance violations) on the x-axis. If only some are affected, all proper runs are included in the aggregate, and the faulty runs are ignored.

The plot is split into three segments with different axis scalings for τ ($[1.0, 1.1)$, $[1.1, 1.2)$, $[1.2, \infty)$) to give more detail to the most important range $[1.0, 1, 1]$.

The $\rho_A(1)$ value is the fraction of instances on which algorithm A found the best partition. We could be tempted to use these values to rank the algorithms directly. However, this is not possible. A direct ranking is only possible if one curve is clearly above the other. For example, two algorithms A,B may be very good on similar instances (e.g., because they have similar components), but one (A) could use an additional, more powerful component (e.g., flow-based refinement) and thus dominate the other (B). Assume, some third algorithm C outperforms both A and B on a small set of the instances but is otherwise largely outperformed by both. Then C may seem superior to B if that assessment is made only based on the $\rho_B(1), \rho_C(1)$ values, as $\rho_B(1)$ is close to zero because it is dominated by A. In a direct comparison between B and C, B could be completely superior to C, but in the comparison with A it is overshadowed. We can see this with the curve of B converging to 1 much faster than that of C. When this happens, we show additional performance profiles with restricted competitor sets.

Even with only two algorithms in the ring, the $\rho_A(1)$ values cannot always be used for a ranking directly. An example is a pool of two algorithms A and B, where A computes the best partition on 40% of the instances but is within $\tau = 1.01$ on 100% of the instances, whereas algorithm B computes the best partition on 60% of the instances but only 65% have $\tau \geq 2$. In this case A is certainly preferable to B. While such extreme cases do not happen in our data, we want to emphasize that it is important to always look at the full profile.

2.4.7 Effectiveness Tests with Virtual Instances

Often we have algorithms with different time-quality trade-offs. Some algorithm may be slow and has high quality, and another may be fast and has low quality. Ahkremtsev et al. [ASS20] propose a method named *effectiveness tests* to compare two algorithms when they are given a similar running time. The idea is to give the faster algorithm the same time as the slower one, to perform additional repetitions of which the best is taken. Instead of performing more runs, we sample from the existing runs with different seeds.

Consider two algorithms A and B, and an instance I. We first sample one run of both algorithms on I. Let t_A^1, t_B^1 be their running times and assume that $t_A^1 \geq t_B^1$. We sample additional runs without replacement for B until their accumulated time exceeds t_A^1 or all runs have been sampled. Let $t_B^2, \ldots, t_B^l$ denote their running times. We accept the last run with probability $(t_A^1 - \sum_{i=1}^{l-1} t_B^i)/t_B^l$ so that the expected time for the sampled runs of B equals t_A^1. The solution quality is the minimum out of the sampled runs.

We generate *virtual instances* which we compare using performance profiles. For each instance (hypergraph, k), we generate 20 virtual instances. If more runs than the number

of seeds are required to reach t_A^1, this favors the slower algorithm A. This approach can be favorable for algorithms that have high variance in solution quality.

We will present effectiveness tests with algorithms run on different numbers of cores if one algorithm is sequential. An argument against this approach is that the sequential algorithm could simply be run independently on the additional cores. However, this argument requires that the machine has sufficient memory to run independent copies, which is not the case for machine A for example. Moreover, in some cases the parallel algorithm is faster by a factor larger than the number of available runs *and* the maximum speedup.

2.4.8 Speedup Plots

To compare the scaling behavior of our algorithms, we plot self-relative speedups for an increasing number of threads. For each instance, we plot the speedup on the y-axis, and the sequential running time on the x-axis. This is done to distinguish the instances where parallelism really matters. On instances that are solved very quickly we cannot expect good speedups and we do not need them.

We found that any one of the common metrics (such as number of pins) is not well correlated with speedups (or running times for that matter), since the running time depends on a variety of different factors (metrics and events that trigger repetitions). Fitting suitable parameters for a combination of the metrics seems much more complicated than plotting against sequential running time, which is often nicely correlated with speedups.

In addition to the per-instance scatter plot, we show rolling geometric means with a fixed window size. For each phase of the multilevel framework (preprocessing, coarsening, initial partitioning, refinement, total) or component (FM, LP, flows) we produce a separate plot to show which parts scale well.

2.4.9 Relative Running Time Plots

Finally, to compare running time between different algorithms we use a *relative slowdown plot*. One algorithm is designated as the baseline and for each other considered algorithm its slowdown over the baseline (running time of the other divided by baseline on the y-axis) is plotted per instance (x-axis). Each algorithm's curve is sorted ascendingly to quickly identify on how many instances it is slower by a certain factor. Similar to performance profiles, we use special symbols (✻, ✘) to denote timeouts and balance violations, this time on the y-axis. The symbols plotted below 1 are for the baseline (where the baseline is slower), symbols above 1 are for the compared algorithm. The y-axis is scaled logarithmically to cope with skewed data, but beware the scaling is not symmetric around 1. With base 10 logarithms, the first minor tick above 1 indicates a factor 2x slowdown, whereas the first minor tick below 1 indicates a factor 0.9x slowdown. The corresponding factor 2x speedup is at $5 \cdot 10^{-1}$, the 5th minor tick below 1.

3 Literature Overview

Research on graph and hypergraph partitioning has started more than 50 years ago, with the seminal work of Kernighan and Lin [KL70] inspiring a long line of research that continues to this day. In this time frame a huge variety of works have been published, far too vast to survey all. Therefore, we focus on the basics and classics, early seminal works, techniques that we use in our work, and recent achievements in parallel algorithms. The dissertation of Schlag [Sch20] contains a historical overview focused on hypergraph partitioning with detailed attributions of different techniques, highlighting in particular the achievements of the VLSI design community. Surveys of recent advances for graph partitioning are given by Buluc et al. [Bul+16] and Bader et al. [BMSW13]. To illustrate the history of partitioning, we start with early local search algorithms such as Kernighan-Lin [KL70] and Fiduccia-Mattheyses [FM82] and then the multilevel framework, before covering recent parallel algorithms and more advanced iterative improvement such as flow-based refinement. We continue with an overview of commonly used software tools for graph and hypergraph partitioning, which predominantly follow the multilevel framework. We finish the discussion of related work with methods that are outside the scope of this work, namely exact and non-constant factor approximation algorithms, spectral and streaming partitioning.

3.1 Iterative Improvement

An *iterative improvement* algorithm takes a given initial partition and tries to improve the objective function by perturbing the partition. The most common approach to iterative improvement are *local search* algorithms, which move one vertex at a time (or a few) to a different block, while maintaining feasibility of the partition.

Algorithm 3.1: Kernighan-Lin

```
1  repeat
2      swaps ← ⟨⟩                                          // empty swap list
3      initialize per-node gains g(u)
4      for i = 1 to min(|V1|, |V2|) do
5          find u ∈ V1, v ∈ V2 with highest swap gain       // O(n^2)
6          swap u and v
7          mark u and v as not movable
8          append (u, v) to swaps and record their gain as g_i
9          update gains of neighbors                       // O(deg(u) + deg(v))
10     j ← arg max Σ_{i=1}^{j} g_i                         // find best prefix
11     g_max ← Σ_{i=1}^{j} g_i
12     for i = j + 1 to min(|V1|, |V2|) do
13         revert swaps[i]
14     reset movable markers
15 until g_max = 0
```

An important concept for this is the gain of a vertex move, which is the absolute reduction in the objective function. Given a move (u, s, t), moving vertex u from block s to t, the gain in the connectivity metric is

$$g(u, s, t) = \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, s) = 1\}) - \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, t) = 0\})$$

where the first term (*benefit*) corresponds to the weight of nets from which s is removed from the blocks $\Lambda(e)$ it has pins in, and the second term (*penalty*) is the weight of nets where t is newly introduced to $\Lambda(e)$.

3.1.1 Kernighan-Lin

The algorithm of Kernighan and Lin [KL70] (KL) is considered the first local search algorithm for graph partitioning. It works only on undirected graphs with unit node weights and bipartitions. Due to the restriction to bipartitions, each node only has one associated gain $g(u)$, that for the opposite block. The central idea is to maintain balance by swapping two nodes $u \in V_1, v \in V_2$ in each step. The gain of swapping u and v is the sum of the individual gains minus a correctional term to account for the potential edge (u, v) between them $g(u) + g(v) - 2\omega(\{(u, v) \in E\})$. In both individual gains we consider the edge (u, v) as removed from the cut, but after the swap it is still cut.

Algorithm Outline. Algorithm 3.1 outlines the KL algorithm in pseudocode. One iteration of the repeat-until loop in line 1 is called a *pass* (also round) and an iteration of the for-loop

in line 4 is called a *step*. The KL algorithm operates in passes in which each node is allowed to move once. In a pass it aims to construct a sequence of node swaps that improves the current partition as much as possible. In each step, it searches for the node pair that offers the highest swap gain, out of those not yet moved nodes. The pair is then swapped, marked as no longer movable in this pass, and the gains of their neighbors are updated to account for newly cut or uncut edges. The swaps and their gains are recorded, so they can potentially be reverted later. Negative gain swaps are allowed temporarily to potentially escape local minima later on. After all possible swaps have been performed, the swap prefix j that yields the highest gain sum $g_{\max} = \sum_{i=1}^{j} g_i$ is selected, and the swaps made after it are reverted (line 12,13). The improved partition is used as input for the next pass, presuming $g_{\max} > 0$, i.e., the current pass found an improvement.

Analysis. Initializing and updating the gains $g(u)$ can be done in $\mathcal{O}(m)$ time. However, selecting the best node pair takes $\mathcal{O}(n^2)$ time per step, leading to an overall $\mathcal{O}(n^3)$ time bound for one pass. In each step, Kernighan and Lin sort the nodes by decreasing gain and the two nested loops for selection visit nodes in this order. They claim an $\mathcal{O}(n^2 \log(n))$ bound because in practice much less than $\Theta(n^2)$ node pairs have to be checked to find the one with maximum gain, using $\Theta(n)$ pair checks in their analysis based on empirical observations. Since the instances of that time were tiny, applying such observations to current instances would be extremely dubious.

Yet there is merit to this claim, in fact Dutt [Dut93] showed that only $\mathcal{O}(\Delta^2)$ pairs have to be checked, with $\Delta = \max_{v \in V} \deg(v)$. The main observation is that the correctional term is non-zero for at most Δ nodes in the inner loop of the selection. After this, a swap gain that is the sum of two individual gains is encountered (non-neighbors). Because the correctional term is subtracted and nodes are visited in decreasing order of their individual gains, the inner loop can stop as soon as a non-neighbor of the node in the outer loop is found. Applying symmetry shows that the outer loop can be restricted in the same way. Dutt claims an $\mathcal{O}(m(\Delta + \log(n)))$ time bound per pass by additionally maintaining the sorted order between steps with a balanced search tree.

There is no good bound on the number of passes; a trivial one is m since in each pass the cut must contain one less edge. In practice, a small constant number of passes around 4-6 suffice [KL70, SK72, Dut93].

Hypergraphs. Schweikert and Kernighan [SK72] implement the KL algorithm on hypergraphs, noting that the clique expansion does not accurately model cuts of electrical circuits. With multiple nets between two vertices, the swap gain becomes the following term.

$$g(u) + g(v) - \omega(\{e \in \mathrm{I}(u) \cap \mathrm{I}(v) \mid \Phi(e, \Pi(u)) = 1\}) - \omega(\{e \in \mathrm{I}(u) \cap \mathrm{I}(v) \mid \Phi(e, \Pi(v)) = 1\})$$

This makes it more time-consuming to compute the correctional term. Schweikert and Kernighan note that the hypergraph KL implementation is a factor 2-4 slower than the graph

KL implementation running on the clique expansion. Again, due to tiny instances at the time, the effects should be much larger today.

Limitations. The KL algorithm has several limitations. It is unclear how to convert it to a heuristic for optimizing k-way partitions directly, since the elementary primitive is a node pair swap. The authors therefore suggest running it on block pairs. Non-unit node weights can be handled by replacing each node v with a clique of size $c(v)$, with edge weights set sufficiently high that they will not be cut, but beware of overflow issues and huge running time overheads from the expansion. However, the biggest limiting factor for applying KL in practice is the time complexity of even the basic version.

3.1.2 Fiduccia-Mattheyses

Therefore Fiduccia and Mattheyses [FM82] developed a linear time variation of KL. This should be deemed the most important improvement of local search algorithms, and it is to this day the basis of refinement engines in modern partitioning frameworks. In addition to the smaller time complexity, an upside of this method is that it works with non-unit vertex weights. The idea of the FM algorithm is to drop the quadratic time selection of node pairs in favor of moving one node in each step. Furthermore, the algorithm is designed directly for hypergraphs not graphs. It retains the same notion of passes as KL. After each pass it reverts back to the best observed prefix, in order to escape local minima by enabling intermediate steps with negative gain moves.

In order to determine the highest gain move quickly, for each block of the bipartition a priority queue stores the vertices in that block with the gain of moving it to the opposite block as the key. Since balance is no longer maintained by swapping vertex pairs, a strategy to select which block to move from is necessary; for example in alternating fashion if perfect balance is desired. Instead, Fiduccia and Mattheyses select the block with higher gain if both moves are feasible, breaking ties for the move resulting in a more balanced partition. If only one is feasible that move is performed. The case that neither move is feasible is not specified (because they use a different balance definition where this case does not occur). One way to deal with this is to repeatedly discard both moves (and mark the vertices as moved), until either one move is feasible or no unmarked vertices remain.

Analysis. Fiduccia and Mattheyses describe an implementation with $\mathcal{O}(p)$ time per pass for unit weight nets. After each move the gains of neighbors must be updated. A naive implementation that recomputes each neighbor's gain from scratch would take $\mathcal{O}(p^2)$. Instead, an increment or decrement of $\omega(e)$ is applied to the pins of a net e depending on $\Phi(e, i)$ values; this is discussed in further detail in Section 4.3.5. To make this efficient, Fiduccia and Mattheyses show that each net e is involved in a constant number of updates, each of which takes $\Theta(|e|)$ time. Overall, this yields time $\mathcal{O}(p)$ for maintaining gains, however the gains must also be updated in the priority queue.

Therefore, a bucket priority queue is employed, which stores all vertices with the same gain in a linked list (or vector), encompassed in an array indexed by gain value. For unit weight nets the maximum/minimum gain is $\pm \max_{v \in V} \deg(v)$. Using an additional array mapping vertices to their entry in the PQ, vertices can be moved between gain buckets in constant time.

To retrieve the highest gain move a pointer to the highest non-empty bucket is used. The pointer is raised if an insertion happens at a higher gain bucket. If the highest gain bucket becomes empty, a linear search to lower non-empty buckets is necessary. Let v_i, v_{i+1} be two consecutive vertices in the move order produced by the FM algorithm. Then the distance between their gain buckets is at most $\deg(v_i) + \deg(v_{i+1})$ (in the case $g(v_i) = \deg(v_i)$ and $g(v_{i+1}) = -\deg(v_{i+1})$). Therefore the total work for maintaining the highest gain bucket pointer is at most $\sum_{v \in V} 2 \deg(v) \in \mathcal{O}(p)$.

Implementation Intricacies. For non-unit net weights the bucket PQ implementation with an array of the size of possible gain values is infeasible. Papa and Markov [PM07] suggest storing non-empty gain buckets in a hash table. A balanced search tree stores pointers to non-empty gain buckets to efficiently retrieve the second highest gain bucket, if the highest becomes empty. However, this increases the update complexity to expected logarithmic from previously worst-case constant. Schlag found that in practice a simple implementation with heaps is faster [Sch20].

A large number of implementation improvements were found in the 1990s by the VLSI design community, which are particularly important for flat FM refinement. Hagen et al. [HHK97] noted that the order in which vertices are extracted from the highest gain bucket has a significant impact on partition quality. Last-In-First-Out (LIFO) order performs significantly better than random order which is still better than FIFO order. Intuitively, LIFO order encourages a more localized search because neighbors of the moved vertex are considered next. This also favors moving clusters of vertices across the cut, which is often the obstacle for escaping a local minimum. Caldwell et al. [CKM99] show that moving neighbors to the front of the gain bucket even if they did not receive a gain update actually worsens partition quality (zero-gain deltas).

Caldwell et al. [CKM99] note additional implicit implementation decisions such as tie-breaking when the highest gain in both blocks is equal (move in the same direction as the previous works best due to the intuition of moving clusters across the cut), and tie-breaking when selecting the best prefix (by best balance). Finally, Caldwell et al. point out that the aforementioned implicit implementation choices barely exhibit their effects when employed within the multilevel framework. Intuitively this makes sense because there are fewer ties and the framework is better at moving clusters. While this is framed as bad because the multilevel framework hides bad implementation choices, we see it as a justification. The LIFO order only works with bucket queues not heaps. Yet, we want the faster speed of heaps, so using them in the multilevel setting is not disadvantageous regarding partition quality compared to the slower bucket queues.

Extensions. Another gain tie-breaking mechanism is due to Krishnamurthy [Kri84], who proposes to look a fixed number of steps l ahead via *higher level gains*. The *binding number* $\beta_A(e)$ of a net e in a block A is the number of pins $\Phi(e, A)$ still in A or ∞ if a pin of e can not be removed from A in this pass. This is a measure how hard it is to remove A from $\Lambda(e)$. Given a bipartition (A, B), the i-th level gain of vertex $u \in A$ is defined as the following term.

$$\omega(\{e \in \mathrm{I}(u) \mid \beta_A(e) = i \text{ and } \beta_B(e) > 0\}) - \omega(\{e \in \mathrm{I}(u) \mid \beta_B(e) = i - 1\})$$

The first term accounts for the nets e for which A can be removed from $\Lambda(e)$ in i moves, whereas the second term accounts for the nets for which removing B would require an additional move. The higher level gains are used as tie-breaking by comparing gains lexicographically. This comes at the expense of a factor l in running time. Krishnamurthy claims an overall $\mathcal{O}(lp)$ time bound per pass, however this bound only holds for updating gain values. Sanchis [San89] shows that an additional factor of the gain span (maximum degree for unweighted) is necessary to update the highest gain pointers.

The CLIP (Cluster-Oriented Iterative-improvement Partitioner) approach of Dutt and Deng [DD96] inserts all vertices into the zero-gain bucket, but sorted by the initial gains (at the beginning of the pass), placing the highest gain as the first to extract. The algorithm then proceeds like regular FM, applying gain delta updates to this initial setup. This approach again favors neighbors of just moved vertices and moving clusters across the cut, because the other vertices are still in the zero-gain bucket.

Karypis and Kumar [KK98] improve the running time of FM in practice by only keeping boundary nodes in the priority queues, as only these can have positive gain. Furthermore, the search is terminated if no improved cut was found after a fixed number of steps, instead of until all nodes were moved. In recent works, this was called the fruitless moves stopping rule [Sch20].

3.1.3 k-way FM Local Search

A severe limitation of the FM algorithm so far is its restriction to bipartitions. Sanchis [San89] was the first to propose a version of FM that optimizes k-way partitions directly. The main difference is that we first have to decide a move direction and then select the best of that direction. She uses $k(k-1)$ PQs, where each PQ represents one move direction, as well as one additional PQ to select the best direction. Her variant achieves $\mathcal{O}(kp)$ time for updating gain values, or $\mathcal{O}(lkp)$ if used with Krishnamurthy's higher level gains. The additional factor k is because now each block is involved in a constant number of updates per net. This translates to $\mathcal{O}(lkp(\log(k) + l\Delta))$ overall, where Δ is the maximum degree. The $\log(k)$ term stems from updates to the PQ for selecting a direction (a heap is used).

All following variants implement basic FM without higher level gains.

Träff [Lar06] uses k PQs: one for each block i storing the highest gain (feasible) move of vertices in block i, plus an additional PQ to select the best block to move from. We call this approach *from-PQs* and it is the one we use, because we can prefer heavier blocks to

move from as a tie-breaker to gain. One can even explicitly rebalance by keeping only heavy blocks in the block-PQ. Additionally, one PQ per vertex tracks the highest gain move, so that the target block t can be found in constant time. The bottleneck is the time for gain updates, which dominates the time for PQ management. For graphs with unit edge weights this variant achieves $\mathcal{O}(m)$ time using bucket PQs, even though $k > 2$ because with graphs only a constant number of blocks are involved per edge. Without lazy initialization a further $\mathcal{O}(nk)$ additive term is necessary to initialize gain values, which is preferable in practice. For non-unit weights an extra $\log(n)$ factor is needed for an implementation with heaps. In Lemma 4.2 we give a more sensitive analysis for gain updates on hypergraphs to reduce the $\mathcal{O}(kp)$ bound, since a reasonable assumption is that many nets are small.

Osipov and Sanders [OS10] as well as Schlag [Sch20] use k PQs to implement what we call *to-PQs*. Each PQ is associated with one block and contains all possible moves into that block leading to a $\mathcal{O}(k \cdot n)$ memory requirement. This enables looking at only moves into non-overloaded blocks, though in the presence of vertex weights there is still no guarantee that the extracted move is feasible.

Karypis and Kumar [KK98], as well as Sanders and Schulz [SS11] use a single PQ to store the highest gain move for each vertex. This is similar to from-PQs but lacks the rebalancing capabilities. After extracting the highest gain node u, Sanders and Schulz recompute gains from scratch and select the best target block, whereas Karypis and Kumar select the best target block using a gain table but only check blocks of neighbors. The downside of this approach is that after a gain update a different target block may be better for the neighbor, which thus requires a scan of the blocks.

3.2 k-way Partitions

There are two main approaches for computing a k-way partition: recursive bipartitioning (RB) and direct. With recursive bipartitioning the hypergraph is first split into two blocks, which are then both split into two each to obtain four, etc., recursively subdividing the blocks until the desired k is reached. This approach is popular [CA99, Kab+17a, Dev+06, KAKS99] due to its simplicity because the refinement only needs 2-way local search algorithms. Simon and Teng [ST97] showed that the quality of a k-way partition obtained via recursive bipartitioning can be arbitrarily far from optimal. While the same can be said for direct k-way with heuristics, in practice the solutions are better than for recursive bipartitioning [Sch20]. This is because objective functions such as the connectivity metric cannot be optimized directly. Kernighan and Lin [KL70] already noted that good bipartitions early in the recursion tree can eliminate optimization potential in later recursion steps. It is doubtful whether accepting partitions that are not locally optimal in early recursion stages will ultimately translate to better partitions later.

We believe that direct k-way is the better approach since it allows to optimize the connectivity metric directly. In his dissertation Schlag [Sch20] compared recursive bipartitioning and direct k-way within the same framework, showing that direct k-way finds better par-

titions *and* is faster. Inefficiencies from the more complicated refinement routines can be compensated by thorough engineering. Another postulated downside of k-way local search is that there are too many potential moves, making the algorithms susceptible to local minima [Sch20, p.96]. Here, localized refinement is the remedy since it restricts the move candidates without blocking the not considered moves at later stages.

A hybrid between the two approaches is recursive partitioning into more than two blocks at each recursion level. This is particularly useful for process mapping, a closely related problem to partitioning. The network topology of a compute cluster is often organized hierarchically (islands, racks, nodes, sockets, cores) with slower communication if higher links in the hierarchy must be used. Recursively partitioning along the topological hierarchy, into the number of next-lower entities, is shown to produce very good process mappings [ST17].

3.3 The Multilevel Paradigm

The second big revolution after the FM algorithm is the multilevel paradigm. Its current form is due to Hendrickson and Leland [HL93], but its origins can be traced back to algebraic multigrid solvers for linear equations [Sou35]. The fundamental idea is to contract vertices that are likely in the same block, such that they can be moved together. There are three phases: the *coarsening* phase, *initial partitioning* and *refinement*. Together they make up one cycle. In the coarsening phase a sequence of successively smaller hypergraphs that approximately preserve the cut structure is constructed. Each hypergraph in this sequence constitutes a level. Typical implementations contract vertex clusterings or vertex pairs. Coarsening stops once the latest hypergraph is sufficiently small to find an initial partition using a potentially expensive algorithm, or such that even simple algorithms are able to find good solutions since the search space is small. In the refinement phase (or uncoarsening), the contractions are reverted level by level in reverse order. The current partition is projected to the next finer hypergraph by assigning vertices that were contracted together to the same block as their coarse vertex. This yields a partition of the finer hypergraph with the same balance and cut properties as that of the coarser one, because the vertex and net weights of the coarse hypergraph were set appropriately through the contraction. Subsequently iterative improvement algorithms are used to further optimize the objective function.

The multilevel approach has two tremendous benefits. It offers refinement at different levels of granularity, moving multiple vertices on coarser levels to make progress more quickly. Through local techniques, the partition is optimized at both a local and a global scale and various granularities in between. The number of levels offers a trade-off between partition quality (more levels, more granularity, more refinement) and running time. We mentioned already that some improvements to FM emphasize moving entire clusters across the cut, which is most naturally captured in the multilevel scheme. Secondly, iterative improvement algorithms converge much faster if they are fed an already good initial solution. Since the coarser hypergraphs are small, substantially less running time is necessary to find a good macro-scale partition compared to running iterative improvement directly on the

input. After projection, only a small amount of work is necessary to repair assignments of vertices that were clustered together but should not belong to the same block, given the new information available. We discuss components used in each of the phases in more detail in the following sections.

3.4 Coarsening Components

The purpose of the coarsening phase is to derive structurally similar but smaller versions of the input hypergraph. This enables faster improvements in the refinement phase and serves as a filter [Wal03] for irrelevant information at different optimization scales. As mentioned, the number of levels offers a trade-off between solution quality and running time. Therefore the number of vertices should be reduced quickly, but not too quickly. Typically the input is shrunk geometrically on each level, resulting in a hierarchy with $\Theta(\log(n))$ levels, such that the overall algorithm takes near-linear time if each component takes near-linear time.

Hypothetically assume we had access to an optimal partition. Then the best way to coarsen the input is to only contract vertices that are in the same block. This preserves the optimal partition down to the coarsest hypergraph.

3.4.1 Hypergraphs

In practice, coarsening algorithms identify clusters of densely connected vertices via local similarity measures that are optimized with local greedy methods. Contracting the clustering then constitutes the next level in the multilevel hierarchy. The most common measure is the heavy-edge rating function [CA99, KK00, AHSS17, GHSS21], which prefers many nets with large weight and small size between a vertex and a cluster.

$$r(u, C) := \sum_{e \in I(u) \cap I(C)} \frac{\omega(e)}{|e| - 1},$$

Catalyürek and Aykanat [CA99] use a simple greedy agglomerative algorithm to optimize this function, which has since been adopted by the majority of partitioning frameworks. It is similar in style to other clustering algorithms for community detection [BGLL08, RAK07]. Initially each vertex is in a cluster of its own (singletons) and vertices are visited in random order. If not part of a multi-vertex cluster yet, the visited vertex joins the cluster in its neighborhood that would yield the highest increase in the rating function. To prevent the formation of too heavy vertices, heavy clusters are penalized by additionally dividing the rating function with the cluster weight. Akhremtsev et al. [AHSS17] instead use a weight constraint to prevent the formation of too heavy vertices.

3.4.2 Graphs

While for hypergraph partitioning, contracting clusterings was the go-to method from the start (due to skewed degrees and net sizes), this has been long overlooked for graph partitioning. Here, contracting maximal matchings has been the standard [KK98, SS11] because it works well on uniform degree mesh-type graphs. Karypis and Kumar [KK95] visit nodes in random order and match them to the unmatched neighbor with highest edge weight. This has no approximation guarantee on the weight of the matching. A simple $\frac{1}{2}$-approximation algorithms visits edges by decreasing weight, and greedily includes them if possible. Holtgrewe et al. [HSS10] investigate different edge rating functions.

On modern complex networks with power-law degree distributions, contracting a matching cannot reduce the graph size quickly enough. In such, many edges connect few hub nodes (high degree) with many less important nodes (low degree). However, at most one edge per node is contracted in a matching, thus leaving many unimportant nodes uncontracted and stalling coarsening progress. To address this, non-adjacent nodes are allowed to be matched if they share a common neighbor and if they are still unmatched after a pass of the regular matching algorithm. This is called two-hop matching. In consecutive passes, LaSalle et al. [LaS+15] first match nodes of degree one, then nodes with identical neighborhoods of any size, then nodes prioritized by similarity of their neighborhoods; starting the next pass only if insufficient coarsening progress was made. Davis et al. [DHKY20] extend this by matching nodes that had the same preferred matching partner during the regular matching pass, but were denied.

Meyerhenke et al. [MSS14] use size-constrained label propagation [RAK07] to obtain a graph clustering for contraction, in order to get rid of the matching restrictions. The rating function is equivalent to the heavy-edge rating function on graphs. Furthermore, the optimization algorithm is equivalent to that of Catalyürek and Aykanat, but it is non-agglomerative, i.e., nodes can back out of a cluster with multiple nodes in it. In our experiments we show that not having this property is actually harmful. Gottesbüren et al. [Got+21] additionally use the two-hop matching technique of Davis et al. for still singleton nodes with the same preferred target cluster. These still exist when using a clustering algorithm due to the cluster size constraint.

3.4.3 Shared Memory Parallelization

Catalyürek et al. [ÇDKU12] propose parallel schemes for agglomerative clustering and greedy matching that use locking, in addition to a lock-free version of greedy matching that resolves conflicts in a second pass. Vertices are visited in parallel. The lock-based algorithms first try to lock the visited vertex, calculate ratings, and then iterate through the candidates. When a better candidate is found, its lock is tested. If successful, the old candidate is replaced and unlocked.

For the lock-free resolution scheme, the matches are stored in a global array M without protecting write access. After one pass, vertices u with $M[M[u]] \neq u$ are matched with

themselves $M[u] \leftarrow u$. These are vertices whose match was visited concurrently and matched to a different vertex. The resolution-based matching scheme is also used by LaSalle and Karypis [LK13].

Size-constrained label propagation coarsening [ASS18a, Got+21] is parallelized by visiting nodes in parallel, but implemented without locking, such that it is possible that nodes cyclically join each other's cluster.

3.4.4 Distributed Memory Parallelization

Karypis and Kumar [KK99] use a 1D distribution of the nodes: all edges of a node are stored on the processor that owns the node. Each node determines its most desirable matching partner. If the matching partner is on the same processor, both are matched and the partner is skipped later in the loop. Otherwise a matching request to the processor owning the partner is sent. In a second step, the nodes who received multiple requests select the most desirable one and approve it. The remaining ones are rejected. The authors also consider a parallelization via node coloring, where nodes of the same color are processed in parallel. However, the matching request scheme is still necessary because two independent nodes can match with the same third node of a different color. The downside of coloring is that it induces a lot of synchronization (after each color class). Meyerhenke et al. [MSS17] implement size-constrained label propagation coarsening for distributed memory with a 1D distribution.

Devine et al. [Dev+06] show how to parallelize heavy edge rating matching (called inner product in their paper) with a 2D distribution of the hypergraph. Through alternating phases of horizontal and vertical communication, exact scores between all unmatched vertices and a small set of randomly sampled matching candidates are calculated. The candidates and their highest rated match are marked as matched. This process is repeated until sufficient coarsening progress is made.

3.4.5 Enhancements

Shaydulin et al. [SCS19b] use algebraic distances (a non-local similarity measure) to modify the net weights used in the coarsening routine, thus encouraging contraction of close vertices and penalizing contraction of far vertices. A similar approach is taken by Sybrandt et al. [SSS20] with embeddings (such as node2vec [GL16]), who incorporate spatial similarity as an extra factor into the heavy-edge rating function. While these approaches are too slow at the moment, they pose an interesting new direction for coarsening, where the development of new ideas has stalled significantly over the years.

Heuer and Schlag [HS17] add a preprocessing phase based on community detection to the coarsening phase. Community detection aims to partition the nodes of a graph into an unspecified number of blocks such that connections are *internally dense and externally sparse*. The idea is to compute a community structure first and then restrict contractions to vertices in the same community. In its essence, this is equivalent to performing a V-cycle [Wal04]

except the partition is obtained through community detection instead of a multilevel cycle. This prevents potentially harmful contractions (good cuts) through guidance from a more global view, where local similarity measures are misguided.

Graph-based Representation. Since the notion of community structure on hypergraphs and corresponding research is not as mature as on graphs, Heuer and Schlag instead perform community detection on the star expansion $G = (V_G, E_G)$ of the input hypergraph H. Thus, this approach partitions both the vertices and nets. As we are only interested in the assignment of vertices, the communities of the nets are discarded.

There is one peculiarity with this approach. Let $\delta := \frac{m}{n}$ denote the *density* of H. If $\delta \ll 1$, the hypergraph has few but large nets, and the corresponding bipartite graph nodes have high degrees. These would predominantly shape the community structure, acting as central nodes that attract a lot of low-degree nodes. As the nodes representing nets are subsequently ignored, the resulting communities would be poorly connected. In order to reduce the impact of large nets on the community structure, Heuer and Schlag propose to incorporate density information into the edge weights of the bipartite graph. For low density instances the edge weight $w'(v, e) := \frac{\omega(e)|\mathrm{I}(v)|}{|e|}$ is used, which penalizes large nets and rewards high vertex degrees. Otherwise, for both uniform and high density, the original net weight $w'(v, e) := \omega(e)$ is used since it performed better in the experiments [HS17]. The authors chose $\delta < 0.75$ as threshold for $\delta \ll 1$.

On this bipartite graph with modified edge weights w', the authors optimize the well-known modularity objective function [NG04] using the Louvain method [BGLL08]. While the argument of providing a more global view is rather hand-wavy in the sense that the used community detection algorithm also performs only local optimization, the quality improvements are impressive (in particular for initial partitions), which motivated us to include this component in our work and parallelize it.

Modularity. Let $\mathcal{C} = \{C_1, \ldots, C_r\}$ denote a community assignment and let $w'(u, C) := \sum_{v \in \Gamma(u) \cap C} w'(u, v)$ denote the weight of edges from node u to community C. The *coverage*

$$\mathrm{cov}(\mathcal{C}) := \sum_{C \in \mathcal{C}} \sum_{u \in C} w'(u, C) / \mathrm{vol}(V_G)$$

is the fraction of edge weights inside communities. The *volume* $\mathrm{vol}(u) := \sum_{v \in \Gamma(u)} w'(u, v)$ is the sum of incident edge weights of a node (counting self-loops twice), which is extended to node-sets $\mathrm{vol}(X) := \sum_{u \in X} \mathrm{vol}(u)$. The modularity of a community assignment $\mathcal{C}$ is:

$$\mathcal{Q}(\mathcal{C}) := \mathrm{cov}(\mathcal{C}) - \sum_{C \in \mathcal{C}} \mathrm{vol}(C)^2 / \mathrm{vol}(V_G)^2$$

where the second term is the *expected coverage* of a randomly rewired graph with the same degree sequence, which alleviates the problem that placing all vertices in the same community would yield the maximum coverage value of 1. Optimizing modularity is NP-hard [Bra+08], which is why heuristics are used in practice.

Louvain Method. The Louvain algorithm [BGLL08] is a well-known method to heuristically optimize modularity with good practical performance. It starts with each node in its own community. In a round, it visits each node in a random order and greedily maximizes modularity by possibly moving the node to the community of a neighbor. After a fixed number of rounds or if no node has been moved in the last round, the communities are contracted and the algorithm is applied recursively to the contracted graph. This continues until no node has been moved on the current level, at which point the community assignment is projected back to the input graph.

The following term shows the gain (modularity difference) of moving a node from its current community C to a neighboring community D.

$$\text{gain}(u, C, D) = \frac{w'(u, D) - w'(u, C)}{\text{vol}(V_G)} + (\text{vol}(C) - \text{vol}(u) - \text{vol}(D))\frac{\text{vol}(u)}{\text{vol}(V_G)^2}$$

It can be computed just from the weight of incident edges to the target D and current community C, as well as their volumes. The volumes for the communities are stored in an array and updated when a vertex is moved. To select the highest gain neighbor community of a node, the weights to all of its neighboring communities are aggregated.

Parallel Louvain Method. Staudt and Meyerhenke [SM16] parallelize the local moving of the Louvain method by visiting nodes in parallel. Cluster volumes are updated with atomic instructions, or more precisely a compare-and-swap loop since fetch-and-add is not supported for floating point numbers. The complexity of a parallel local moving round is $O(|E_G|)$ work and $O(\max_{v \in V_G}(\deg(v)) + \log(|V_G|))$ depth.

3.5 Initial Partitioning

The coarsening phase usually stops when only $t \cdot k$ vertices remain, where t is a constant input parameter, such that initial partitioning can place around t vertices in each block and thus has some optimization potential. Typical values for t are in the range of $100 - 1000$. Preen and Smith [PS19] showed that the best point to stop coarsening for partition quality is instance-dependent, and propose an adaptive stopping rule based on the progress made on the last level.

Regarding algorithms, any partitioning technique can be used, such as evolutionary [SS12, ASS18b], spectral partitioning, or even exact methods. Though the latter is often too expensive and it does not offer better quality on the final partition. During development it is not uncommon to first use an existing partitioning framework by other authors for initial partitioning. Maturing frameworks eventually incorporate their own set of initial partitioning routines.

There is a variety of simple graph growing heuristics and BFS-partitioning that work well [KK98, CA99, KAKS99, Sch20, SS11]. These should always be complimented with subsequent iterative improvement. With such simple algorithms it is important to perform

many randomized repetitions, as there are often multiple choices with the same gain. Battiti et al. [BBR97] propose two advanced gain definitions that incorporate the progress to keeping a hyperedge uncut, as opposed to the classic versions that only consider whether the assignment fixes the hyperedge in the cut or not. See Section 4.2 for a more detailed description of the ones we use in our portfolio.

While for the multilevel framework direct k-way works better than recursive bipartitioning, flat direct k-way partitioning is worse than using recursive multilevel bipartitioning [KK98, KK00, Heu15]. Then only algorithms for flat bipartitions are necessary. An additional benefit is that recursive calls are independent and can thus be carried out in parallel.

3.6 More Iterative Improvement

We now turn to more iterative improvement algorithms that are used in the refinement phase, with a focus on parallel algorithms. KL and FM are P-complete via NC-reduction from the circuit value problem [SW91]. This is strong evidence that the existence of poly-log depth parallel algorithms for KL and FM is unlikely, though it is not conclusive proof. Therefore, all of the presented parallel schemes differ from classic FM and KL by relaxing some aspect.

First we start with label propagation as an example of a very simple, inherently parallel algorithm that is widely used. Then we discuss challenges regarding parallelization and some techniques used to address these. We also introduce two algorithms that started out as sequential (localized FM and flow-based refinement).

3.6.1 Label Propagation

Eight years before the label propagation community detection algorithm [RAK07], Karypis and Kumar [KK99, KK00] proposed a simple refinement algorithm that is equivalent, and has later been called size-constrained label propagation [MSS14] or balanced label propagation [UB13]. It is designed with parallelism in mind. Instead of the singleton assignment for community detection [RAK07] or coarsening [MSS14], label propagation refinement starts with a given k-way input partition. Vertices are visited in random order. For each vertex the gains of moving it to any of the blocks in its neighborhood is calculated. Out of the feasible moves the one with the highest gain is selected if the gain is positive. Ties are broken in favor of the more balanced partition. Since only positive gain moves are performed, this scheme lacks the ability to escape local minima. The benefit is its simplicity and it is straightforward to parallelize: simply visit vertices in parallel.

The gains can be incorrect due to concurrent moves in the neighborhood. If moves are performed asynchronously and become visible to other processors, this does not affect optimization a lot in practice. With synchronous implementations some quality loss is observed. Meyerhenke et al. [MSS17] use a hybrid between synchronous and asynchronous, where communication and computation are overlapped. The updates from the previous round come in gradually while the next round is under way. Even if high partition quality

is desired and thus escaping local minima is a must-have feature, label propagation is a valuable component since it finds the easy improvements with much smaller work investment compared to heavier, less scalable algorithms [ASS18a].

3.6.2 Incorrect Gains

Regardless of the specific optimization routine, in order to achieve non-trivial parallel speedup, vertex moves must be performed in parallel. In this case the gain calculated on one thread can be incorrect if neighbors of the vertex in question are moved concurrently. Consider adjacent vertices u, v with $\Pi(u) \neq \Pi(v)$, where u is moved to $\Pi(v)$ and v is moved to $\Pi(u)$. For both u and v the gain contains an $\omega((u, v))$ term since it appears that the edge is removed from the cut. If both moves are performed however, the edge is still cut. This way the overall cut may even get worse, if $\omega(e)$ is sufficiently large to compensate for cutting new edges.

Karypis and Kumar [KK99] employ node coloring to fix this issue. Moving nodes with the same color in parallel does not result in incorrect gains since each color class is an independent set. However, this incurs a large amount of synchronization (one barrier per color class), and is not well applicable to hypergraphs because all vertices of one net need different colors.

A more common technique is to split the refinement into an upstream and a downstream pass [KK99, LK13, Dev+06], where the move directions are restricted to blocks with higher/lower ID, respectively. This prevents the example given above, but restricts optimization capabilities heavily since the target blocks become overloaded very quickly. For all moves with the same source and target block the actual total gain is at least the sum of the calculated gains. However, cut degradation is still possible if more than two blocks are involved. Yet, for refining 2-way partitions and thus recursive bipartitioning this works.

Given moves in a fixed order, Akhremtsev et al. [ASS18a] recompute the exact gains and keep the highest gain prefix subject to balance. Unfortunately, their algorithm is sequential, but in Chapter 4 we present a parallelization.

3.6.3 Balance Constraint

With concurrent vertex moves the balance constraint cannot be ensured by checking whether each individual move preserves it. Either a synchronization mechanism or explicit rebalancing is necessary.

Maintaining Balance. With shared memory it is possible to use atomic fetch-and-add instructions to update block weights. If the weight of the target block is raised above the maximum weight the move is reverted. It is important that the weight to the target is added first, so that the weight of the source block has not been reduced if the move is reverted, accidentally inviting concurrent moves into the source block that may exceed the balance

constraint after the revert. This technique is used by Akhremtsev et al. [ASS18a] and Gottesbüren et al. [Got+21], as well as in the works of this dissertation.

In distributed memory, maintaining the balance constraint is more difficult. Karypis and Kumar [KK99] and Meyerhenke et al. [MSS17] only update block weights with processor-local moves and use these for balance checks. Block weight updates are synchronized at the end of a pass. Meyerhenke et al. split each pass into multiple subrounds to reduce the chance that balance violations occur, but cannot provably prevent it.

Trifunovic and Knottenbelt [TK04] gather the calculated positive gain moves on the root processor, which then approves a subset. As the root processor is clearly a bottleneck, this approach cannot be expected to scale to a large number of machines. The approval algorithm used is described but not in sufficient detail. The following description is taken verbatim: "a greedy scheme favouring taking back moves of sets with large weight and small gain".

Ugander and Backstrom [UB13] gather all calculated moves and solve a linear program to select the subset of moves that maximizes the gain sum while maintaining balance. For each move direction the moves are sorted by gain and a prefix is selected by the linear program. The size of the linear program is independent of the graph size but takes $\Theta(k^2)$ variables and $\mathcal{O}(k^2 n_g)$ constraints, where $n_g = 1 + 2 \max_{v \in V} \deg(v)$ is the number of different gain values. This makes it feasible to solve the LP on a single machine for moderate k. The technique is restricted to unit weight nodes and does not consider that gains are incorrect from simultaneous moves.

Kabiljo et al. [Kab+17a] simplify this by pairing up moves in opposite directions, sorting by gain, and swapping all vertices in the shorter sequence with the prefix of the longer sequence with the same length. Again, this is restricted to unweighted nodes, however we propose a parallel version with weights in Chapter 5. This technique only allows pairwise movement, whereas the version of Ugander and Backstrom allows arbitrary cycles of exchanged nodes. Two ideas are proposed to implement the approach in distributed memory. First, instead of sorting, the gains are grouped into exponentially spaced buckets to which the two-way prefix swap is applied. Secondly, a probabilistic version is run on the leftover moves that preserves balance in expectation. Each processor counts the number of locally computed moves in each direction, which are aggregated at a root processor. Let $S_{i,j}$ denote the number of moves from block i to j. For each direction, the root processor calculates $\frac{\min(S_{i,j}, S_{j,i})}{S_{i,j}}$ and broadcasts it to all processors. Each move in a direction is then approved independently with this probability.

Rebalancing. Finally, we describe rebalancing algorithms. Lasalle and Karypis [LK13] track the performed moves in buffers: each thread maintains the moves it performed. To rebalance, the buffers are traversed in reverse order. Moves into overloaded blocks are reverted. Each thread is responsible for restoring excess weight proportional to the amount it moved into that block, such that combined across threads, the balance is restored.

Slota et al. [SMR16] employ label propagation with modified gains. The original gain is multiplied by $L_{\max}/|c(V_i)|$ to favor moving into lighter blocks. This is similar to an approach

used in streaming partitioning [MLLS17]. Maleki et al. [MABP21] use label propagation but in synchronous fashion. Only a subset of the moves prioritized by calculated gain is performed. This is repeated, with recalculated gains, until the partition is balanced.

For large k, Gottesbüren et al. [Got+21] store the moves out of overloaded blocks in one priority queue per block. Since k is large, the overloaded blocks are processed in parallel. The key for the priority queues is the ratio of highest gain (to a non-overloaded block) and node weight. Each priority queue is initialized with just enough nodes to remove the excess weight. In case designated target blocks become close to overloaded, neighbors of moved nodes in the same former block are inserted.

3.6.4 Incomplete Information

Another challenge to computing accurate gains is incomplete information. Some processors may not know that vertices in the neighborhood were moved, even if this happened long before the current vertex is visited. This is particularly severe for distributed memory environments, where the fine-grained communication necessary to exchange this information is too expensive. Therefore distributed algorithms often follow a synchronous approach [Kab+17a, MSS17, UB13, HSWZ18], where gains are calculated based on only locally available, potentially outdated information. After a pass the updated block IDs are sent to the processors of the neighbors in batched communication, and then the next pass is started.

An even more severe case is the framework Zoltan [Dev+06] which uses a 2D partition of the hypergraph's incidence matrix. The rows are the nets and the columns are the vertices. Each processor is responsible for a rectangular submatrix and therefore has neither access to all incident nets of a vertex nor to all pins a of net. Calculating initial gains is done cooperatively. Gain updates after a move are done with only local information. The actual moving is done by one processor per processor-column (all processors in a column share the same vertices), the one with the most non-zeroes since it has the most information.

3.6.5 Localized FM

Osipov and Sanders [OS10] propose a *localized* version of FM. Instead of considering all nodes at once, the search starts with one or a few seed nodes in the priority queues. After a node is moved, all of its neighbors are added to the search, such that it gradually expands in directions where progress is made.

The intent for this scheme is to have a dynamic algorithm that can react to small changes. It was designed for n-level uncoarsening where only one node is uncontracted per level. During uncoarsening, localized FM is initialized with just the uncontracted node and its former representative.

Adaptive Stopping Rule. Modelling the objective function behavior over time as a random walk, an adaptive stopping rule is proposed, since the fruitless moves stopping rule is too insensitive and thus causes too long running times (order of magnitude factor). If it is deemed

unlikely that the search will find an overall improvement given the recent history of gains, the search is terminated.

Gain values are assumed as identically distributed, independent random variables with expected value μ and variance σ^2. These are estimated from the previous gain values since the best objective function value seen so far. Let r denote the number of moves since the best cut value, α, β a tuning parameter, where β is a minimum number of steps typically set to $\log(n)$. If $r\mu^2 > \alpha\sigma^2 + \beta$ it is deemed unlikely that the local search will find a better partition. This is of course heuristic, but works well in practice.

Static Version. Sanders and Schulz [SS11] devise a static version of localized FM that is organized in passes like classic FM. In a pass each node is allowed to move once. Each node that is a boundary node at the beginning of the pass serves as a seed for localized FM. If the next potential seed was already moved it is skipped.

The intuition why this is better at escaping local minima than classic FM is that if only negative gain moves are left, classic FM performs a large number of moves with similar negative gain in arbitrary areas. However, only a few of these moves will enable good positive gain moves in the future for an overall improvement. At the point these are performed, it is likely that the overall partition has deteriorated significantly, such that no overall improvement is possible anymore. Localized FM attempts to isolate a few negative gain moves in the hope of quickly enabling positive gain moves for an overall improvement. If this fails, the adaptive stopping rule triggers quickly and stops the search, keeping the untouched nodes available for moving in later searches.

Parallelization. Another benefit of localized FM is that it can be parallelized by starting parallel searches with different seeds [ASS18a]. The searches are kept disjoint with atomic test-and-set to claim exclusive ownership of nodes. There are a variety of details to make this efficient and practical on hypergraphs, which are one of the main contributions of this dissertation. Further details including details of Akhremtsev's version [ASS18a] are discussed in Chapter 4, in order to highlight the differences between their version and ours.

3.6.6 Greedy and Hill-Scanning

LaSalle and Karypis [LK13] parallelize Metis' refinement [KK98] on shared memory architectures. The sequential algorithm performs FM on boundary nodes but performs only positive gain moves, i.e., terminates when only negative gain moves are left. This eases parallelization since no moves must be reverted, and thus there is no serial move order to observe. However, it does eliminate the ability to escape local minima. Supposedly, this is still better than label propagation, as more promising nodes are moved first.

Nodes are statically assigned to threads, where the sequential algorithm is run on the local nodes. Moves are communicated to the threads of neighboring nodes via message buffers. The threads frequently check their buffers to update local gains.

Balance checks are done optimistically with locally updated block weights. Potential balance violations are repaired at synchronization points using the approach with move-buffers outlined above.

In a second paper [LK16] an extension that is able to escape local minima to some extent is proposed. This is a simplified variant of localized FM. When the extracted node u from the priority queue has negative gain, a small group of up to 16 nodes around u is incrementally constructed in the same way as localized FM. A difference is that all nodes must be moved to the same block. Furthermore, the expansion stops as soon as an improvement is possible, even if there may be further improvements within the 16 node expansion limit. As opposed to the greedy algorithm, the selected nodes are not restricted to the locally owned nodes.

3.6.7 Interface Optimization

Walshaw and Cross [WC00] propose a parallel refinement algorithm that combines label propagation with an arbitrary 2-way refinement routine. First, the vertices with positive gain moves are grouped by their highest gain move directions. More precisely, vertices with preferred move direction from block p to q or q to p are grouped together. Then sequential 2-way refinement is applied to each such group. This is parallelized over the groups but each block participates in only one refinement at a time due to their scheduling algorithm. As such, this approach is restricted to the setting where the number of blocks equals the number of processors.

3.6.8 Flow-Based Refinement

As opposed to balanced minimum cuts, unbalanced minimum cuts can be computed in polynomial time via maximum flows, and can thus be a valuable subroutine.

Sanders and Schulz [SS11] propose a refinement algorithm for bipartitions, which is then scheduled on different block pairs for refining k-way partitions. Given a bipartition (V_0, V_1), the idea is to select sets $B_0 \subset V_0, B_1 \subset V_1$ of nodes that are allowed to be moved. A flow network is constructed on the subgraph induced by $B_0 \cup B_1$, and creating an artificial source and sink. For each node in B_0 with edges to $V_0 \setminus B_0$ an arc with infinite capacity from the source is created; and analogously from each node in B_1 with edges to $V_1 \setminus B_1$ an arc to the sink. A minimum cut on this flow network yields a bipartition with smaller or equal cut size as the input bipartition, though it may be imbalanced. Therefore, the sizes of V_0, V_1 are restricted. We revisit this approach in Chapter 7 with an improved method (incremental max flow instances) and parallelization, so refer to this chapter for more details.

The MQI algorithm of Lang and Rao [LR04] optimizes a quotient-cut metric such as expansion (ratio of cut to size of the smaller side) or conductance (ratio of cut to smaller volume). It is an iterative algorithm that solves a sequence of nested maximum flow problems. Starting with an initial bipartition, the edge cut becomes smaller and the partition more unbalanced with each iteration. The cuts are nested in the sense that the smaller side of the cut is a subset of the previous iteration's smaller side. Once no improvement is found, the cut

is guaranteed to be optimal regarding the quotient metric, out of all cuts where the smaller side is a subset of the initial bipartition's smaller side. The flow network contains the leftover nodes of the smaller side and uses modified edge weights to achieve this convergence criterion. A particularly elegant feature of this approach is that it can be phrased as a parametric flow problem, such that the sequence can be solved in the same asymptotic time as a single maximum flow instance. In a follow-up work [AL08] Andersen and Rao extend this approach to enable growing the cut. The new algorithm is provably at least as strong as MQI but will work better in situations where the initial bipartition is less balanced.

3.7 Deep Multilevel Partitioning

A recent approach that combines recursive partitioning and direct k-way is called *deep multilevel* partitioning [Got+21]. This approach is important for large k where both other approaches fail. Recursive multilevel bipartitioning performs $\log_2(k)$ *cycles* of coarsening and uncoarsening (full graph, two graphs half the size, four graphs a quarter the size, etc.). This overhead factor is too high if k is very large. With direct k-way the coarsest graph should have $\Omega(k)$ nodes so that a constant number of nodes can be placed in each block. Therefore coarsening stops once $t \cdot k$ nodes are left, where t is a constant input parameter. If recursive bipartitioning is used for initial partitioning, the direct k-way scheme degenerates to recursive bipartitioning since coarsening is stopped very early. Otherwise, an expensive flat direct k-way algorithm is used, which is too slow because the assumption that the coarsest graph is small no longer holds. Deep multilevel performs only one cycle of coarsening and uncoarsening as direct k-way. But it starts with less than k blocks at the coarsest level and gradually expands the number of blocks during uncoarsening.

More formally, the idea is to select the appropriate number of blocks for each of the graphs in the multilevel hierarchy depending on their size. The input graph is coarsened until only $2t$ vertices are left. During uncoarsening, the blocks from the previous level are further subdivided using bipartitioning, to achieve the appropriate number of blocks k' for this level, before a k'-way refinement algorithm improves the current partition. The value for k' is chosen such that each bipartitioning call works on approximately $2t$ vertices. Combined with the fact that only one cycle is performed, this leads to overall near-linear time if each algorithmic component has near-linear time. The appropriate choice is $k' := \min(k, \text{ceil}_2(\frac{n'}{C}))$ for an n'-vertex graph, where $\text{ceil}_2(x)$ is x rounded up to the next power of two. If the top-level partition does not have the desired k yet, it is further subdivided to achieve k blocks. This happens if $n < k \cdot t$, and in this case an additional $\mathcal{O}(\log(\frac{kt}{n}))$ factor is incurred for bipartitioning.

3.8 n-level Partitioning

Inspired by a speedup technique for shortest path computations in road networks [GSSV12], Osipov and Sanders [OS10] study an extreme version of the multilevel paradigm where only one node is contracted on each level. This leads to a hierarchy of almost n levels, thus the name n-level partitioning, which instantiates the multilevel scheme for maximum refinement granularity at the cost of slow running times. To avoid quadratic memory usage, a dynamic graph data structure is used to implement contractions. On each level, localized FM local search with the adaptive stopping rule is performed, initialized with the just uncontracted node and its partner as seeds. Schlag et al. [Sch+16, AHSS17] extend this approach to hypergraphs, investing significant engineering efforts to make the approach feasible on both the data structure and algorithmic side. For example, two orders of magnitude in running time were saved by visiting vertices randomly and contracting greedily (as in agglomerative heavy-edge clustering) as opposed to maintaining the highest rated contractions in a priority queue like Osipov and Sanders [OS10]. Further improvements are based on known techniques such as gain caching, the adaptive stopping rule of Osipov and Sanders, and avoiding gain update cases. In Chapter 6 we propose a parallel version of the n-level scheme.

3.9 Partitioning Frameworks

These various techniques are implemented in different frameworks. In Table 3.1, we therefore list the most relevant partitioning frameworks and the components they use. These frameworks are all considered in our evaluations, with the exception of Parkway, as already mentioned.

Mt-KaHiP and (Mt-)KaHyPar are the only partitioners with a k-way FM implementation. Surprisingly a lot of frameworks still use random initial partitioning. Most frameworks use similar coarsening approaches based on either heavy-edge clustering or matching. However, BiPart uses a different coarsening algorithm, where each vertex is matched to its smallest incident net. Subsequently all vertices matched to the same net are merged.

While Zoltan has a 2-way FM implementation, it uses a 2D distribution of the hypergraph's incidence matrix. The actual FM routine is performed by one processor per column using only locally available information, namely the one with the most non-zeroes in its rectangular submatrix. Therefore, we can expect quality penalties if many processors are used. Furthermore, it uses the upstream/downstream pass approach where only moves in one direction are allowed, such that only few moves are feasible in each pass before the balance constraint is reached.

3.10 Complexity and Approximation

Wagner and Wagner [WW93] study the complexity of cut minimization problems ($k = 2$) with different balance requirements between minimum unbalanced cut and minimum bisection.

Table 3.1: Partitioning frameworks and their components. HEC = agglomerative heavy-edge clustering, HEM = heavy-edge matching, LP = label propagation, K = direct k-way, RB = recursive bipartitioning, GHG = greedy hypergraph growing. The last three frameworks are restricted to graphs. Frameworks marked with † are shared-memory parallel and those marked with ᵞ are distributed.

Framework	Reference	Coarsening	Initial	Refinement	Mode
PaToH	[CA99]	HEC	portfolio	FM	RB
Mondriaan	[VB05]	HEM	random	FM	RB
hMetis-R	[KAKS99]	HEC	random, BFS	FM	RB
hMetis-K	[KK00]	HEC	hMetis-R	LP	K
KaHyPar	[Sch20]	n-level HEM	RB, portfolio	localized FM, FlowCutter	K
Zoltan-AlgD	[SCS19a]	HEM + AlgD	GHG	2-phase FM	RB
HYPE	[May+18]	–	GHG	–	K
BiPart†	[MABP21]	smallest net	GHG	LP, pairwise prefixes	RB
Mt-KaHyPar†		HEC or n-level HEM	RB, portfolio	localized FM, LP, FlowCutter	K, (RB, deep)
Zoltanᵞ	[Dev+06]	2D HEM	GHG	2-phase 2D FM	RB
SHPᵞ	[Kab+17a]	–	random	LP, pairwise prefixes	RB, K
Parkwayᵞ	[TK04]	HEM	hMetis-K	LP, gather	K
Mt-Metis†	[LK16]	HEM + 2-hop	RB, Metis	2-phase Greedy, Hill-Scanning	K
Mt-KaHiP†	[ASS18a]	LP	KaHiP	localized FM, LP	K
KaMinPar†	[Got+21]	LP + 2-hop	portfolio, deep	LP	deep

Their results indicate that the problem becomes harder with tighter constraints. As soon as the smaller side must contain at least αn^{δ} nodes the problem is NP-hard, for arbitrary $\alpha > 0, \delta > 0$. Balanced 2-way partitioning is a special case with $\delta = 1$. If the smaller side must only contain a constant number of nodes, the problem can be solved in polynomial time, and the case $\geq \alpha \log(n)$ is still open.

For $k \geq 3, \varepsilon = 0$, there is no finite factor approximation algorithm subject to P ≠ NP, via reduction from 3-partition [AR06]. With $\varepsilon \in (0, 1]$ the best known approximation factor is $\mathcal{O}(\log(n)$ (however with exponential running time in $1/\varepsilon$ [FF15]. For $\varepsilon = 1$ a better approximation ratio of $\mathcal{O}(\sqrt{\log(n)\log(k)})$ can be achieved [KNS09]. For $k = 2, \varepsilon = 0$, an $\mathcal{O}(\log(n))$ approximation is also possible [Räc08].

3.11 Exact Solvers

There are a lot of exact solver implementations for graphs, but not hypergraphs. Most are based on branch-and-bound and focus on bipartitioning. Bounds are computed using various

methods such as semi-definite programming [KRC00, AFHM08], linear programming [BCR97, Fer+98, AFHM08], multi-commodity flows [Sen01, SST03], or combinatorial bounds based on max flow and packing paths [DGRW12, DW12]. The combinatorial bounds can be computed much faster but are less effective at pruning the search. This is a common trade-off in branch-and-bound algorithms. Usually it is settled on the side of the stronger bounds, but further recent works in other areas prefer faster bounds as well [Got+20].

For the most part, these solvers are not practical on instances with more than 100 nodes, yet the solver by Delling et al. [DGRW12] is appropriate as long as the cut is very small.

3.12 Spectral Partitioning

Since the success of combinatorial iterative improvement algorithms and the multilevel paradigm, spectral partitioning has been deemed outdated. It is slower and produces worse partitions. Yet, due to the difficulty of parallelizing the combinatorial algorithms and the advent of highly optimized linear algebra libraries on GPUs, spectral partitioning has recently experienced a resurgence [ABR20]. At the moment spectral partitioning is not competitive with the parallel combinatorial algorithms, but the research area has become active again.

The fundamental idea [Fie75] is to bipartition the nodes based on their entries in the eigenvector x_2 of the second smallest eigenvalue λ_2 (the Fiedler vector) of the graph's Laplacian matrix L. The smallest eigenvalue λ_1 of L is 0 with eigenvector $(1, \dots, 1)^T$, which holds no information. Nodes with an entry smaller than the median, are assigned to one block, and those greater to the other. To allow imbalances $\varepsilon > 0$ other quantiles can be used as the splitter. The Laplacian is defined as $L = D - A$ where D is the matrix with node degrees on the diagonal and A is the adjacency matrix.

The following description is based on one in the the dissertation of Christian Schulz [Sch13]. First note that $x^T L x = \sum_{(u,v)\in E}(x[u] - x[v])^2$. Furthermore, assume we have a bipartition (V_1, V_2) with cut edges C. Let x be the vector with $x[u] = -1$ if $u \in V_1$ and $x[u] = 1$ if $u \in V_2$. Then $x^T L x = \sum_{(u,v)\in C}(x[u] - x[v])^2 = 4|C|$ as $x[u] - x[v]$ is zero if u and v are in the same block. This leads to a formulation of graph bisection as the optimization problem of minimizing $x^T L x$, subject to $x[u] \in \{-1, 1\}$ (partition), as well as $(1, \dots, 1)x = 0$ (balance). If we relax the integrality constraint on x to real values but restrict the values via $x^T x = n$, the optimal solution is the Fiedler vector by Lagrange's optimality conditions.

This approach is restricted to bipartitions and thus recursive bipartitioning is used for k-way partitions. Hendrickson and Leland [HL95] show how to use multiple eigenvectors to extend this approach to direct k-way partitions.

3.13 Streaming Partitioning

In the streaming setting the graph is too large to be stored in memory. Instead, it is streamed in via the network or file system. In the classic model the algorithm receives one node and

its incident edges at a time. The node must be assigned right away, and the decision cannot be changed later. The buffered model allows keeping a limited set of nodes and their edges in memory.

All known algorithms in the classic model perform some kind of one-pass label propagation. A modification of the rating function encourages balanced partitions, as only partial node assignments are known.

Stanton and Kliot [SK12] use $|V_i \cap \Gamma(v)| \cdot (1 - \frac{|V_i|}{L_{\max}})$ as the rating function to maximize, in their *linear deterministic greedy* algorithm. The first term is simply the label propagation objective, whereas the second term is a penalty for heavy blocks.

In their *Fennel* algorithm, Tsourakakis et al. [TGRV14] use $|V_i \cap \Gamma(v)| - \alpha\gamma|V_i|^{\gamma-1}$, where $\gamma = \frac{3}{2}$ is a parameter and $\alpha = m\frac{k^{\gamma-1}}{n^\gamma}$. The parameter γ interpolates between pure label propagation aka optimizing for the number of neighbors in the block ($\gamma = 1$) and optimizing for the number of non-neighbors ($\gamma = 2$). The authors claim that the corresponding objective function is essentially a form of modularity, but it seems like they used the wrong null model (Erdos-Renyi instead of configuration model).

Faraj and Schulz [FS21] propose a buffered/batched streaming algorithm that incorporates a multilevel partitioner. The algorithm loads a batch of nodes, builds a model graph, and then partitions the model with a multilevel algorithm in-memory, before proceeding with the next batch. The model consists of the already streamed/assigned nodes contracted into one node per block (fixed to that block) as well as the nodes in the current batch and the thus induced subgraph. Hence, the optimizer takes the already built partition assignment into account, without using memory proportional to the number of already streamed nodes. Jafari et al. [JSA21] follow a similar approach, but omit the model graph. The already built partition can still be taken into account, however the running time does not benefit from a compact graph representation (smaller IDs, more cache-friendly) and all partition assignments must be stored in memory.

Taşyaran et al. [TDKU21] use streaming label propagation without a penalty term for partitioning hypergraphs. Furthermore, they propose methods to reduce the memory used for storing connectivity sets, such as (1) storing only a fixed number of entries and using a random eviction policy, (2) using a Bloom filter. This comes at the cost of storing incomplete (1) or partially wrong (2) connectivity sets.

4 Parallel Multilevel Hypergraph Partitioning

Recent work on hypergraph partitioning has focused on improving the solution quality further and further. This was for example done by investing substantial amounts of running time for complex refinement routines such as flow-based refinement. With our works on FlowCutter refinement integrated in KaHyPar [GHSW20] and recent evolutionary algorithms [ASS18b] this research direction appears saturated for now. So far, the majority of partitioning algorithms are still sequential. The time was ripe to pivot towards improving the speed of these algorithms, in order to make them more applicable in practice. Motivated by recent advances in shared-memory graph partitioning [ASS18a], this led us to parallelize the techniques employed in KaHyPar. The fast and parallel multilevel hypergraph partitioning algorithm presented in this chapter is the first step in a series of works on this topic. In this chapter we focus on a classic multilevel algorithm with approximately $\log(n)$ levels, which uses k-way refinement. The missing components from KaHyPar are n-level (un)coarsening and flow-based refinement, which are parallelized in Chapter 6 and Chapter 7, respectively.

Attributions. This chapter is based on a joint publication [GHSS21] with Tobias Heuer, Peter Sanders and Sebastian Schlag. The paper was written by Tobias Heuer and me, with editing by Peter Sanders and Sebastian Schlag. However, due to the space constraints of a conference paper, the contents were almost completely reworked for this dissertation, adding a substantial amount of details. The source code was written by Tobias Heuer and me. Tobias focused on coarsening, label propagation refinement and initial partitioning (which had already been the subject of his bachelor thesis [Heu15]), whereas I worked on parallel localized FM refinement and community detection. As both contributed to the entirety of the code, both dissertations describe the full system. We are grateful to Michael Hamann for his ideas on a first version of the parallel gain recalculation. While we describe a more memory-

efficient version here, his idea is still published in the conference paper. Additionally, we include some parts that were not yet in the code/algorithm at the time of writing the paper, but do benefit it. These came about during performance engineering in the context of the follow-up paper on parallel n-level partitioning [GHSS22]. Due to this, the experiments are completely redone for this dissertation. Finally, some algorithmic descriptions are based on a more recent publication with Michael Hamann on deterministic parallel partitioning [GH21] (written predominantly by me), because some low-level components were replaced with newer versions developed for said paper.

Chapter Overview. The structure of this chapter follows the multilevel paradigm. We start with coarsening in Section 4.1, where we extend an existing parallelization for agglomerative heavy-edge clustering by showing how to use less locking, and resolve cyclic cluster join conflicts on-the-fly. Additionally, we propose a parallel hypergraph contraction algorithm, and describe engineering and parallelization details for the community detection preprocessing. This is followed by the initial partitioning phase in Section 4.2, where we discuss shortcomings in the parallelization of existing approaches and propose using work-stealing as a remedy to scalability issues. Our main novel contributions are in the refinement phase described in Section 4.3, where we describe several techniques to improve the accuracy of calculated gains in scenarios where parallel vertex moves are performed: attributed gains to double-check and trace actual gains, a parallel gain table, and a parallel gain recalculation. These are integrated into our direct k-way label propagation and localized FM refinement algorithms, for which we provide implementation and engineering details; with specific focus on hypergraph implementations. With parallel gain recalculation, this is the first partitioner to implement a fully parallel k-way FM refinement.

Compared to the conference publication, we add running time analyses in terms of work and depth, as well as substantial algorithmic details. This chapter not only serves as a description of novel techniques, but also as a description of the overall framework Mt-KaHyPar. Therefore, this chapter contains discussions of existing approaches that we use in more detail than common for short conference publications. We conclude with a thorough experimental evaluation, in which we assess the effectiveness of the algorithmic components employed in our framework, perform extensive comparisons with state-of-the-art partitioning systems, and provide configuration experiments.

4.1 Coarsening

The purpose of the coarsening phase is to provide a sequence of structurally similar and successively smaller (coarser) hypergraphs $\langle H_0 = H, H_1, \ldots, H_r \rangle$, to enable faster improvements and convergence in the refinement phase. We obtain a coarser hypergraph H_{i+1} from the finer H_i by contracting a vertex clustering of H_i.

Algorithm 4.1: Coarsening Phase

```
1  V_0 ← V, i ← 0
2  while |V_i| > t · k do
3      rep[u] ← u, weight[u] ← c(u) ∀u ∈ V_i
4      for u ∈ V_i in random order do in parallel
5          C ← HeavyEdgeRating(u)
6          if ClusterJoinProtocol(u, C) succeeds
7              if (weight[C] +=_atomic c(u)) ≤ W_max
8                  rep[u] ← C
9              else
10                 weight[C] −=_atomic c(u)
11     H_{i+1} = (V_{i+1}, E_{i+1}) ← Contract(rep)
12     i++
```

4.1.1 Agglomerative Clustering

Algorithm 4.1 shows pseudocode for our approach. Initially, each vertex is in its own cluster (line 3, $\text{rep}[u] = u$) and we visit the vertices in random order in parallel (line 4). For each vertex u we select a cluster C to join (line 5) that maximizes the *heavy-edge* rating function [CA99, HS17, KAKS99].

$$r(u, C) := \sum_{e \in I(u) \cap I(C)} \frac{\omega(e)}{|e| - 1}$$

This rating prefers clusters that share a large number of heavy nets of small size with u. We then run a locking protocol (described in the next paragraph) to actually perform the move into the desired cluster, which performs on-the-fly conflict resolution.

A clustering is *agglomerative* if no vertex leaves a cluster that already consists of multiple vertices. We achieve this by skipping non-singleton vertices in the parallel loop and via the locking protocol. This property ensures that each cluster contains a vertex with the same ID (its leader). Furthermore, the actual rating of a cluster is at least as good as the calculated one; it may be better if some other vertex joined concurrently, but it will never be worse.

Additionally, we enforce a weight constraint $\sum_{v \in C} c(v) \leq W_{\max} := \lceil \frac{c(V)}{t \cdot k} \rceil$ on the clusters. Here t is a parameter defined shortly. This constraint ensures that a balanced k-way partition of the coarsest hypergraph exists, since we prevent overly heavy vertices.

We use atomic fetch-and-add instructions to update cluster weights concurrently. We first check if an update would violate the constraint, and only perform it if not. This still has no atomic consistency, so we perform the atomic fetch-and-add instruction and check its return value (line 7). If the new cluster weight exceeds the maximum weight, we subtract the vertex weight again and keep the vertex in its singleton cluster, so that it may try to move again

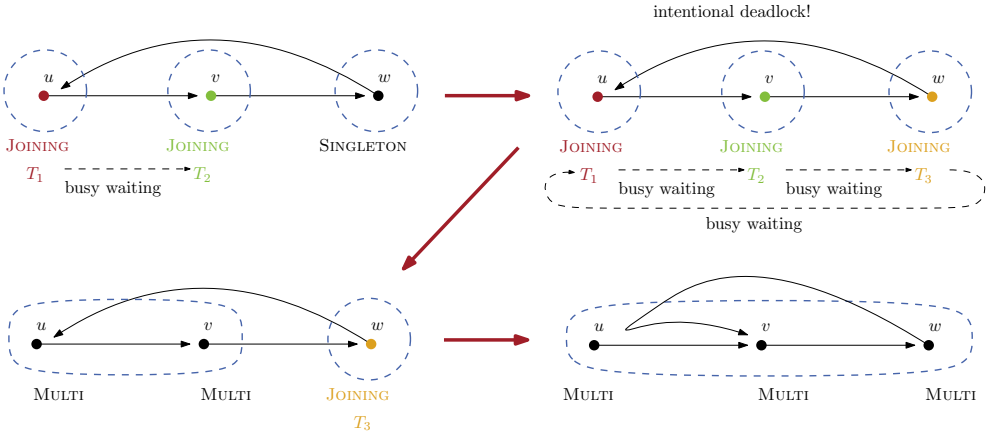

Figure 4.1: Cluster locking protocol. Blue dashed lines show the current cluster assignment, whereas the arrows show intentions to join a specific cluster.

in the next iteration. Note that we do not update the weight of u's old cluster, since it is now empty and no longer needed. We remark that in this phase, strictly enforcing weight constraints is not particularly important; the non-atomic check would probably work fine due to the $1/t$ factor in $W_{\max}$.

After one pass over the vertices, we contract the clustering to construct the next hypergraph H_{i+1} (line 11). We keep performing passes until the latest hypergraph is small enough for initial partitioning ($|V_i| \leq t \cdot k$, line 2). The parameter t controls the size of the coarsest hypergraph, such that initial partitioning has good potential for optimization decisions, placing roughly t vertices in each block. A typical value for us is $t = 160$, even though a fixed choice is not optimal [PS19].

Cluster Join Protocol. A side effect of the parallelization is that two vertices may concurrently join each other's cluster and are subsequently still left as singletons. More generally, there may be a cycle of vertices trying to join each other. This type of race condition is often left unaddressed, (1) because maybe multiple rounds are performed before contracting (offering a second chance), (2) because clustering quality is not severely affected, or (3) because it does not happen very often. Yet, we argue that such behavior is harmful for the algorithm's convergence, in particular once the hypergraphs are small.

We employ a vertex-locking protocol with three states per lock that resolves such cyclic join conflicts on-the-fly. Each vertex can be in one of three states: *singleton*, currently *joining* a cluster, or part of a *multi-vertex cluster*. We maintain vertex states with compare-and-swap instructions. The semantics of the singleton and multi-vertex state are *unlocked*, whereas the joining state has *locked* semantic. We distinguish between singleton and multi-vertex due to

the agglomerative property. We need to protect singletons whose vertex is about to leave from taking on new vertices, whereas multi-vertex clusters can be joined right away, as their vertices are not considered for moving.

If vertex u is in a singleton cluster, we find a cluster C for u to join, and atomically set the state of u to joining. If C is already a multi-vertex cluster, we directly set the state of u to multi-vertex cluster, letting the join operation succeed.

Otherwise, if C is a singleton, we try to atomically change the state of the vertex v in C to joining. Because the clustering is agglomerative v's identifier is just C. If that succeeds, u joins C and both u and v are marked as part of a multi-vertex cluster.

In the case that v is currently trying to join some other cluster C', we spin in a busy-waiting loop until the state of v is updated to multi-vertex cluster by the thread currently working on v, and then join its new cluster. In that busy-waiting loop, we check if u is part of a cycle of vertices trying to join each other, as this may result in a deadlock. This is done by storing intended joins in an array, which we can use as parent pointers to chase until we get back to u (cycle detected) or a vertex pointing to itself (tree root). The deadlock case is illustrated in Figure 4.1. Notice that provoking this deadlock is intentional, as it synchronizes all join operations of a cyclic dependency, allowing us to fix it. If a cycle is detected, the vertex with the smallest ID in the cycle gets to break it by joining its intended cluster. This then triggers a cascade of join approvals in both ways around the cycle, effectively merging all associated vertices. The joins are still subject to the weight constraint, which is checked before each approval.

Heavy-Edge Rating Calculation. Algorithm 4.2 shows pseudocode for calculating the heavy-edge rating function for the neighbor clusters of a vertex u. First, we aggregate the ratings in a sparse array indexed by cluster ID and store the potential candidates in a dense vector. Line 4 incorporates the information from community detection preprocessing, restricting candidates to clusters of neighbors in the same community. In a second step we select the highest-rated candidate (ties broken uniformly at random) and reset the ratings. We already check the cluster weight constraint to make it less likely that the join is denied in the subsequent atomic check.

Rating calculation is the most time-consuming part of the clustering routine by far. For a vertex u it takes $\mathcal{O}(\sum_{e \in \mathrm{I}(u)} |e|)$ work if implemented naively as in Algorithm 4.2. With this, the overall work of one full coarsening pass would be $\mathcal{O}(\sum_{e \in E} |e|^2)$ which is infeasible in practice. To accelerate this, we use a parameter η to skip nets with $|e| > \eta$. Their contribution to the rating is small due to the $1/|e|$ term. In practice we choose $\eta = 1000$ based on previous experience [Sch20]. This reduces the work for one vertex down to $\mathcal{O}(\deg(u))$, and the overall work of a pass down to $\mathcal{O}(\sum_{u \in V} \deg(u)) = \mathcal{O}(p)$, though the constant term is quite large. The depth of one coarsening pass is thus $\mathcal{O}(\max_{u \in V} \deg(u) + \log(n))$.

We could further reduce the depth to $\mathcal{O}(\log(\max_{u \in V} \deg(u)) + \log(n))$ by parallelizing the loop over incident nets in the per-vertex calculation, using atomic fetch-and-add instructions for the ratings. The check rating[rep[v]] = 0 can be faithfully implemented because

Algorithm 4.2: Compute Heavy-Edge Rating

```
   Input: vertex u ∈ V
 1 candidates ← ∅
 2 for e ∈ I(u) do
 3 |   for v ∈ e do
 4 |   |   if community[u] = community[v]
 5 |   |   |   if rating[rep[v]] = 0
 6 |   |   |   |   add rep[v] to candidates
 7 |   |   |   rating[rep[v]] += ω(e)/|e|
 8 for C ∈ candidates do
 9 |   if weight[C] + c(u) ≤ W_max and rating[C] > best rating
10 |   |   store C as best candidate
11 |   rating[C] ← 0
12 return best candidate
```

the atomic instruction returns the value immediately prior to its execution. However, in practice the outer level of parallelism over the vertices is clearly sufficient and the per-vertex parallelization would only be worthwhile for very large degrees.

Coarsening Progress. As already stated, the number of levels in the multilevel hierarchy offers a trade-off between speed and quality. Therefore it is important to find the right pace: not too quickly where information gets lost, but not too slow either. During a coarsening pass we track the number of clusters, which corresponds to the number of vertices on the next level. If enough progress is made, we stop the coarsening pass and create the next level. We use the condition $\frac{|V_i|}{|V_{i+1}|} > \beta$ on level i, where β is a user parameter we call the *shrink factor*. Intuitively, good values for β are in the range $[1.5, 10]$. For example matching-based coarsening is hard-capped at 2. With clustering-based coarsening, the hypergraph can be shrunk substantially faster, which is usually not desirable, hence we need to control it. In the parameter study, we determine that $\beta = 2.5$ works best for us.

Similarly, if too little progress $\frac{|V_i|}{|V_{i+1}|} < \alpha$ is made, we terminate coarsening altogether and proceed with initial partitioning. Here α is a second input parameter with reasonable values around 1.01. When this condition is triggered, it is usually due to a lack of light clusters in the neighborhood of singleton vertices. Oscillation issues arising from parallelism are handled by the locking protocol; in fact the locking protocol encourages more progress.

Differences to PaToH's Agglomerative Clustering. Catalyürek et al. [ÇDKU12] consider a similar parallelization of PaToH's agglomerative clustering [CA99]. The major difference to our approach is the way cluster join conflicts are handled. They try to lock the visited vertex u before the rating calculation and skip u if this fails. When a higher rated candidate

is found in the selection loop, the lock on the cluster's representative vertex (same ID due to agglomerative property) is tried. In case of success, the candidate is stored and the previously best candidate is released. If the lock fails the candidate is ignored. Once again, this means the lock for u is held during the entirety of the rating process.

The intention is to avoid cyclic joins, as the candidate that u is trying to join must be successfully locked. This is not possible if the candidate currently tries to join a cluster itself. Note that deferring the try-lock until the candidate selection loop would give the same parallel consistency guarantees, but the rating calculation would be wasted if some other vertex joined u in the meantime. One upside of this approach is that only try-lock operations are used. However, the convergence issue we mentioned still persists here. If no suitable candidate can be locked, the vertex stays a singleton, whereas in our approach this only happens if the cluster weight constraint fails. This is remedied to a certain extent by trying multiple candidates prioritized by rating.

During the writing of this dissertation, we noticed that we did not conduct any comparison between our approach and theirs for the conference paper. It was still unclear whether either of the locking approaches have a noticeable impact on partition quality or running time. Additionally, there is some design space left to explore. Therefore, we perform this comparison in Section 4.4 and add two variants of their approach (try all candidates, lock only best candidate) as well as a variant using no locks at all (which corresponds to label propagation coarsening). We believe our approach is better since we hold locks for a shorter time, perform fewer lock operations overall, and do not ignore the best choice if a lock fails. This is confirmed in the experiments.

Additionally, our heavy-edge rating calculation algorithm differs from the one in PaToH. We aggregate ratings directly at the clusters as we iterate over the neighborhood, whereas their algorithm fully constructs the neighborhood in the rating map. The ratings are then aggregated at the same time as selecting a candidate, which means the second loop (line 8) is not over the candidates but the neighbor vertices. This aggravates the heavier use of locking further.

4.1.2 Contraction

We now recapitulate the semantics of contracting a clustering, before describing our parallel algorithm. Vertices of the same cluster C are merged into a *coarse* vertex with weight $\sum_{v \in C} c(v)$. The incident nets of the coarse vertex are obtained from the union of the nets of its constituents. For each net e of H_i, its pins are replaced by their corresponding coarse vertex in H_{i+1}. Nets consisting of a single pin are discarded. From a set of identical nets (containing the same pins in H_{i+1}) we keep one representative and aggregate their weights.

We describe a different algorithm than in the conference paper, since this one performed better when writing the deterministic partitioning code (Chapter 5). The process consists of the following four steps, the first three of which are also shown in Algorithm 4.3.

1. Remap cluster IDs to a compact interval.

2. Generate pin lists of coarse nets.
3. Detect and remove identical nets.
4. Assemble the CSR data structures.

Remap Cluster IDs. To remap cluster IDs, we compute a prefix sum over an array with a 1 at position i if cluster ID i is used, and a 0 otherwise. This array then contains the remapped compact ID for cluster i at position i, which we assign to the vertices in a parallel loop, replacing their old cluster ID with the remapped one. The prefix sum also yields the number of coarse vertices $n' = |V_{i+1}|$. This step has $\mathcal{O}(n)$ work and $\mathcal{O}(\log(n)$ depth.

Generate Pin Lists. Generating pin lists for H_{i+1} is done independently in parallel over the nets of H_i (line 6 - 18). For each $e \in E_i$, we remap vertex IDs in the pin list to the compact IDs. We use a thread-local bitvector of size n' to avoid duplicate entries. Deduplication could also be done without extra memory by sorting and using a filter to remove consecutive duplicates (`std::unique`), but the bitvector is faster in practice. If only one pin is left, we clear the vector to mark the single-pin net as removed; this is considered later in the CSR assembly step. This step takes $\mathcal{O}(p)$ work and $\mathcal{O}(\max_{e \in E_i} |e| + \log(m))$ depth. While the depth may be reduced by parallelizing the deduplication, we do not implement this due to the sufficient outer level of parallelism. The new pin list vector is stored in a global array coarse-pins at position e. This effectively creates an adjacency list for H_{i+1} but still uses net IDs for H_i, since we will only know the number of nets in H_{i+1} after the next step: identical net detection.

Identical Net Detection. A trivial approach to detecting identical nets is to compare all pin lists pair-wise. Since this is not feasible, the InrSrt algorithm of Aykanat et al. [ACU08, DKÇ13] uses a fingerprint $\sum_{v \in e} v^2$ to eliminate unnecessary comparisons. Nets with different fingerprints (or different size) need not be compared. This is achieved by sorting nets by fingerprint and size, such that nets with the same fingerprint are consecutive in memory. On each range with the same fingerprint the pairwise comparisons are performed (see line 22-25). The net in the outer loop (line 22) is the representative at which duplicate's weights are aggregated, if detected in the inner loop (line 24). Duplicates are marked as removed by clearing their coarse-pins$[e]$ vector. The outer loop thus skips nets with an empty pin list. The equality-check is implemented with a bitvector of size n' (re-use from previous step) with bits for the pins of the outer loop's net set. Surprisingly, this was faster than sorting the pin-lists (once in the previous step) for direct comparison, even if parallel sorting was used for very large nets.

While InrSrt can be parallelized by simply plugging in a parallel sorting algorithm, a hashing-based parallelization was faster. This can be interpreted as a form of a map-reduce computation, where we group items on processors by some key, and then treat all items with the same key sequentially. For this, we perform the fingerprint computation during

Algorithm 4.3: Contract Clustering: Steps 1 - 3

```
    // remap cluster IDs
 1  for u ∈ V do in parallel mapping[u] ← 0
 2  for u ∈ V do in parallel mapping[rep[u]] ← 1
 3  n' ← PrefixSum(mapping)                                  // n' = # coarse vertices
 4  for u ∈ V do in parallel rep[u] ← mapping[rep[u]]
    // generate pin lists
 5  buckets ← ∅                          // hash map with a vector + spinlock per slot
 6  for e ∈ E do in parallel
 7      pins ← ∅                                                        // empty vector
 8      contained[0..n] ← false                    // thread-local bitvector of size n
 9      for v ∈ e do                                                // remap and deduplicate
10          if not contained[rep[v]]
11              contained[rep[v]] ← true
12              add rep[v] to pins
13      if |pins| > 1
14          fingerprint ← ∑_{v∈pins} v^2
15          add (e, fingerprint) to buckets, hashing by fingerprint
16      else
17          clear pins                                                  // mark e as removed
18      coarse-pins[e] ← pins                                          // global adjacency list
    // identical net detection
19  m' ← 0, p' ← 0
20  for B ∈ buckets do in parallel
21      sort B by fingerprint
22      for i = 0 to |B| with coarse-pins[B[i].e] ≠ ∅ do              // pairwise comparisons
23          w ← 0
24          for j = i + 1 to |B| and B[i].fingerprint = B[j].fingerprint do
25              if coarse-pins[B[i].e] = coarse-pins[B[j].e]
26                  clear coarse-pins[B[j].e]                  // mark B[j].e as removed
27                  w += ω(B[j].e)
28          coarse-net-weight[B[i].e] ← w
29          m' +=_reduce 1
30          p' +=_reduce |coarse-pins[B[i].e]|
```

the pin-list generation (line 14), and then insert the net into a parallel hash map (line 15), with the fingerprint as key. The hash map's slots are vectors (called buckets), such that all nets with the same fingerprint are placed in the same bucket. We then run the sequential

algorithm on each bucket independently in parallel (line 20).

Analyzing the complexity of this step is more difficult since it depends on the false-positive rate of the fingerprint function and the distribution of the hash keys, which is why we refrain from it. One might expect that this is the most time-consuming step, but during performance engineering it was always faster than generating the pin lists, showing the effectiveness of the simple $\sum_{v \in e} v^2$ fingerprint function in practice. This fingerprint function performed best in experiments by Deveci et al. [DKÇ13] as well as Akhremtsev et al. [AHSS17] who compared false-positive rates. We should also mention that with more sophisticated fingerprint functions, one can actually obtain provable bounds on the false-positive rates [HS18b].

Assemble CSR Data Structures. The last step is to assemble the two CSR data structures for the pins of nets and the incident nets of vertices. At this point we have generated the pin lists of H_{i+1} but they are still stored in an adjacency list that is indexed by the net IDs of H_i.

To obtain the CSR representation, we can apply the standard conversion from the pin list format (adjacency lists for the nets), with some ID remapping. We first compute two prefix sums over the nets of E_i with non-empty coarse pin lists to obtain a fine-to-coarse net ID mapping and their sizes to obtain offsets into the CSR for the pins. In a second parallel loop over E_i we write the pin lists to the CSR. We compute the degrees of the coarse vertices by atomically incrementing a counter for each coarse vertex in the copy loop. With the degrees, we can again use a prefix sum to compute offsets for the incident nets CSR. In a last parallel loop, this time over the nets of E_{i+1}, we iterate over the (now constructed) pin list of H_{i+1} and write the net ID to the position obtained from atomically incrementing the offset of the pin. The offsets are now rotated by one position, which we fix by employing a trick. We actually construct the degrees at one position to the right, and atomically decrement during the net ID writing, such that the final offsets are at the correct position.

In the introduction, we mentioned that this contraction algorithm was developed for the deterministic partitioning paper (Chapter 5). To avoid repeating this algorithm later, we show how to make it deterministic here. These steps are omitted in the non-deterministic default version. There are only two sources of non-determinism in this algorithm. The first is what we just described: the incident net IDs may be in an arbitrary order, which we fix by sorting each range. The second is which one from a set of identical nets becomes the representative we keep. Different choices here will cause different coarse net IDs, since we assign them based on a prefix sum, which then causes differences in the CSR. We fix this by including the net ID as a tertiary comparison criterion for the sorting during identical net detection, such that the order is deterministic.

4.1.3 Community Detection Enhances Coarsening

The last piece of the coarsening phase is what is actually run first: the enhancement of coarsening decisions based on community detection, as proposed by Heuer and Schlag [HS17]

for sequential KaHyPar. We follow their approach and optimize modularity on the star expansion with hand-crafted edge weights w' as described in Section 3.4.5, using the parallel version of the Louvain algorithm by Meyerhenke and Staudt [SM16].

Our Contribution. In our implementation, we parallelize local moving in the same way, but employ a randomized visit order. Furthermore, we remove a term from the gain computation that only depends on the origin cluster, reducing arithmetic operations by a factor of about 2.

Recall that the modularity gain of moving a node from its current community C to a neighboring community D is the following term.

$$\frac{w'(u,D) - w'(u,C)}{\mathrm{vol}(V_G)} + (\mathrm{vol}(C) - \mathrm{vol}(u) - \mathrm{vol}(D))\frac{\mathrm{vol}(u)}{\mathrm{vol}(V_G)^2}$$

First, we remove the redundant $\mathrm{vol}(V_G)$ factor in both denominators, already eliminating a division. For comparing potential target clusters, we use $w'(u,D) - \mathrm{vol}(D)\frac{\mathrm{vol}(u)}{\mathrm{vol}(V_G)}$, since the term $-w'(u,C) + (\mathrm{vol}(C) - \mathrm{vol}(u))\frac{\mathrm{vol}(u)}{\mathrm{vol}(V_G)}$ is the same for all potential targets. We incorporate the option of not moving the node by considering this as the initially best choice with gain $w'(u,C) - (\mathrm{vol}(C) - \mathrm{vol}(u))\frac{\mathrm{vol}(u)}{\mathrm{vol}(V_G)}$, due to the adjusted gain definition for target clusters $D \neq C$. Additionally, we precompute $\frac{1}{\mathrm{vol}(V_G)}$ once for the graph and $\frac{\mathrm{vol}(u)}{\mathrm{vol}(V_G)}$ once per node so that the gain computation does not perform any divisions at all, which is one of the more expensive arithmetic operations.

Our contraction algorithm works differently than that of Staudt and Meyerhenke, and is a shared-memory version of a map-reduce contraction algorithm by Zeitz [HSWZ18]. We remap the community IDs to a compact interval as for the hypergraph contraction. Subsequently, we sort the nodes by community ID via parallel counting sort, which also gives us an array of offsets, pointing to the ranges where communities begin. We parallelize the generation of the coarse edges on a per-community level. For each community, one thread generates all of its outgoing coarse edges by iterating sequentially over the neighbors of the vertices in the community, and aggregating the edge weights in a sparse vector indexed by community ID of the neighbor. The work is $O(|E_G|)$, and the depth is the largest degree sum over vertices in a community. Again, the depth can be reduced by parallelizing the per-community generation analogously to the generation of coarse pin lists (Algorithm 4.3).

Community Detection on Hypergraphs. We have since supervised a bachelor thesis [Kra21] that faithfully transfers two community structure measures (modularity and map equation) to hypergraphs, and implements a parallel Louvain method directly on the hypergraph for these measures. Other researchers [Kam+19, CVB21] have worked on transferring modularity to hypergraphs, coming up with different measures depending on how hyperedges are counted as covered: all pins in the same community, the majority of pins in the same community, etc. Our modularity measure is based on $\lambda - 1$, i.e., it accounts for the number of different blocks cut by a hyperedge, whereas other measures are based on cut or

model what the clique expansion would do. The goal for this was to both establish the notion of community structure on hypergraphs more deeply, and get rid of the hand-crafted edge weights. While this did improve partition quality by a small margin, it was substantially slower (not just due to implementation issues), which is why we ultimately did not include it in the framework.

4.2 Initial Partitioning

Based on previous research [KK00] and our own experience [Heu15], recursive bipartitioning works better than flat direct k-way algorithms for initial partitioning. Therefore, we perform recursive bipartitioning with the parallel coarsening and uncoarsening code. More precisely, we have a main coarsening cycle for direct k-way that coarsens until $t \cdot k$ vertices are left. This is then followed by recursive bipartitioning which coarsens further down to $t \cdot 2$ vertices and only then runs flat initial partitioning. For recursive bipartitioning, we use the adaptive imbalance formulation of Schlag et al. [Sch+16] to guarantee ε-balance of the final k-way partition.

4.2.1 Parallel Recursive Bipartitioning

To gain further parallelism, we perform recursive partitioning tasks in parallel. A previous approach [LK13] proposes to split the threads for recursive sub-tasks, statically assigning them. This will cause bad load imbalance issues if some sub-tasks deal with substantially larger (or more dense) sub-hypergraphs than others. This is particularly severe for hypergraphs (as opposed to graphs), since the number of hyperedges and pins may still be large, even if only few vertices are left. For this observation, note that the number of distinct edges in a graph is bounded by $\binom{n}{2}$, thus guaranteeing well-collapsing instances, whereas for a hypergraph it is only bounded by 2^n. This is further aggravated if coarsening converges long before reaching the $t \cdot k$ limit. To alleviate this load balance issue, we implement the entire multilevel code with a framework that uses a work-stealing task scheduler, such that threads will join the work in other branches of the recursive partition tree.

4.2.2 Portfolio-Based Flat Bipartitioning

We use the same portfolio of sequential flat bipartitioning heuristics as sequential KaHyPar [Sch+16], but parallelize independent runs. In total, there are 9 different sequential algorithms, including six different versions of greedy hypergraph growing [CA99]. These are repeated several times with different random seeds to diversify the solution space. Each 2-way partition is refined with sequential 2-way FM to navigate even the partitions of potentially weaker algorithms into a local minimum. Since the optimization landscape is diverse, even seemingly bad algorithms can contribute the best initial partition if refined with FM, as

shown by previous results [Heu15]. Out of the computed solutions that are balanced, we pick the one with smallest cut with ties broken by balance and a coin toss.

Random. A very simple algorithm is to iterate through the vertices and *toss a coin* for the block assignment. In case an assignment would exceed the maximum block weight, the vertex is assigned to the other block. If that would also exceed the other block's weight we take the result of the coin toss and accept that the partition is imbalanced.

Pseudo-Peripheral Seeds. The following algorithms start with one vertex (seed) in each block and gradually grow the blocks. The quality of seeds influences the partition quality. One choice that works well are vertices that are *far away* from each other, resulting in the name pseudo-peripheral. Starting from a random vertex, we perform a BFS. The last visited vertex is the seed for V_0. It also serves as the start point for a second BFS, whose last visited vertex is the seed for V_1.

BFS. The BFS-based flat algorithm performs two alternating BFSs from pseudo-peripheral seeds, adding one vertex at a time to each block. In each iteration, the next unassigned vertex from the BFS queue is added to the associated block if it fits, and its neighbors are added to the BFS queue. If a queue becomes empty before all vertices are assigned, a random unassigned vertex is added to it.

Label Propagation. For label propagation initial partitioning, the vertices are initially unassigned. Two pseudo-peripheral vertices and a fixed number ($\tau = 5$ [Heu15]) of their neighbors are initially assigned to the respective blocks V_0, V_1. Then label propagation rounds are performed until no vertex was moved in the last round or a maximum number of rounds is exceeded. In each round, *all* vertices are visited in random order and moved to the block that has the highest connectivity gain, subject to the balance constraint. Note that the gain computation formula distinguishes between unassigned and assigned vertices. The gain for unassigned vertices is still well-defined since it is forced to be assigned at the time it is first visited. Note also that after the first round all vertices are assigned.

Greedy Hypergraph Growing. Finally, we employ six different variants of greedy hypergraph growing [CA99], combining three priority queue (PQ) selection strategies and two scoring functions (PQ keys). Again, each block is initialized with a pseudo-peripheral seed. For each block, we maintain a PQ with unassigned neighbors of the vertices currently in the block. The two different scoring functions are (1) the direct connectivity gain of adding the vertex u to the associated block and (2) the *max-net* gain, which picks the block with the highest weight $\arg\max_{i \in [k]} \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, V_i) > 0\})$ of already incident nets. The latter does not suffer from zero-gain moves as the former, where a net only contributes positively if $\Phi(e, V_i) = |e| - 1$. We have three PQ selection strategies: alternating (round-robin), global (pick the PQ with higher gain), and sequential (which first grows V_0 and then V_1).

Prefer Promising Algorithms. In Ref. [GHSS22] we propose a method to reduce the many repetitions of less promising flat bipartitioning algorithms. Each algorithm in the portfolio is run at least 5 times. After this, we only run it again if it is likely to find a better partition than the best partition Π^* observed so far. We estimate this based on the arithmetic mean μ and standard deviation σ of connectivity values the flat algorithm achieved so far, using the 95% rule[1]. Assuming the connectivity values follow a normal distribution, roughly 95% of the runs will fall between $\mu - 2\sigma$ and $\mu + 2\sigma$. If $\mu - 2\sigma > (\lambda - 1)(\Pi^*)$, we do not run this flat algorithm again. While this is somewhat ad hoc and hand-wavy, it cuts down on initial partitioning time and only affects partition quality a little, as shown in our experiments.

4.3 Refinement

In this section, we discuss our two parallel k-way refinement algorithms label propagation (Section 4.3.4) and localized FM (Section 4.3.7); with a focus on techniques to compute *more accurate* gains. For algorithms that greedily move vertices concurrently it is inherently difficult to compute exact gain values, as moves in the neighborhood affect the gain of a vertex. An example are two concurrent moves worsening the partition quality, even though the individual gains suggested an improvement.

Consider adjacent vertices u, v with $\Pi(u) \neq \Pi(v)$, where u is moved to $\Pi(v)$ and v is moved to $\Pi(u)$ concurrently. For both vertices, $\omega(e)$ is added to the gain since the thread thinks the edge (u, v) will be removed from the cut, yet after both moves are executed the edge is still cut. If $\omega(e)$ compensated for new edges being cut, the overall cut increases. Since connectivity reduces to cut on graphs, this problem is just as relevant for hypergraphs.

This poses a significant challenge in the parallelization of existing sequential refinement algorithms, which is why we propose three techniques for more accurate gains in parallel vertex moving heuristics. For detecting and reverting such quality-degrading moves we propose a technique named *attributed gains* that is based on atomically consistent updates of certain data structures, which are described in Section 4.3.1.

To make the reader familiar with these concepts, we describe how to use them to calculate gains from scratch in Section 4.3.2, before showing how attributed gains work in Section 4.3.3. Subsequently, we show how to maintain a *globally shared* gain table in parallel, based on similar ideas (Section 4.3.5). In Section 4.3.6 we parallelize the so far sequential gain recalculation phase [ASS18a] at the end of an FM round.

4.3.1 Partition Data Structures

We store and maintain the pin counts $\Phi(e, i)$ and connectivity sets $\Lambda(e)$ for each net e and block V_i. When a vertex u is moved from block s to t, we iterate over its incident nets $I(u)$ and for each $e \in I(u)$ we increment $\Phi(e, t)$ += 1 and decrement $\Phi(e, s)$ −= 1. If $\Phi(e, t)$ is now 1, then we add t to $\Lambda(e)$, and if $\Phi(e, s)$ is now 0, then we remove s from $\Lambda(e)$.

[1] https://online.stat.psu.edu/stat200/lesson/2/2.2/2.2.7

For the $\Phi(e, V_i)$-values we use a packed representation with $\lceil \log(\max_{e \in E} |e|) \rceil$ bits per entry to save memory. In the presence of very large nets, this can be further reduced by grouping similar-sized nets (e.g., exponentially spaced), such that the many small nets profit from the smaller representation, though for simplicity this is not done in our implementation. Unfortunately, this packed representation prevents the use of atomic instructions to update the $\Phi(e, i)$ values. Instead we must use one spin lock-per net. Each thread holds at most one spin-lock at a time for an $\mathcal{O}(1)$ operation. Note that only the writes are synchronized, not the reads.

For the connectivity sets we use a bit-set of size k per net, with the entry at position i indicating that $i \in \Lambda(e)$. To iterate through $\Lambda(e)$, we use *count-leading-zeroes* instructions. Similarly, we use *pop-count* instructions to calculate the connectivity of a net. Adding and removing a block is implemented by toggling the bit using an atomic fetch-XOR instruction.

Finally, the last piece of data associated with a partition that we maintain is the block weights. Whenever a vertex is moved, the block weights of the source and target block are updated with atomic fetch-and-add instructions. One downside this bears is that maintaining block weights globally results in false sharing and contention on these memory locations. However, compared to the amount of other updates this is negligible. In early versions we experimented with different means of reducing contention (e.g., storing thread-local deltas that are applied frequently). The simple global array was faster and we can leverage the result from the atomic update to actually guarantee balance. An option to reduce false sharing we did not explore is to align entries with cache line boundaries; for example TBB offers an allocator that implements this. The wasted space is not an issue since k cache lines is small compared to the rest of the data structures.

4.3.2 Calculating Gains

Recall that the gain $g_i(u)$ can be written as $g_i(u) = b(u) - p_i(u)$, where $b(u) := \omega(\{e \in I(u) \mid \Phi(e, \Pi(u)) = 1\})$ and $p_i(u) := \omega(\{e \in I(u) \mid \Phi(e, V_i) = 0\})$. We think of the term $b(u)$ as the *benefit* of moving u out of its current block, and we consider $p_i(u)$ the *penalty* for moving u into V_i.

Calculating $g_i(u)$ is straight-forward: iterate over all incident nets $e \in E$, increase the gain by $\omega(e)$ if $\Phi(e, \Pi(u)) = 1$, and decrease it by $\omega(e)$ if $\Phi(e, i) = 0$. Often, we are interested in calculating the gains to all blocks at the same time, in order to pick the highest gain target block. Algorithm 4.4 shows how to achieve this without an $O(k)$ loop over the blocks for each incident net, performing a loop over the connectivity set $\Lambda(e)$ instead, which is expected to be much smaller. To see why this works, we rewrite

$$b(u) = \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi(u)) = 1\}) = \omega(\mathrm{I}(u)) - \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, \Pi(u)) > 1\})$$

and

$$p_i(u) = \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, i) = 0\}) = \omega(\mathrm{I}(u)) - \omega(\{e \in \mathrm{I}(u) \mid \Phi(e, i) > 0\})$$

Algorithm 4.4: Compute Max Gain Move

```
   Input: vertex u ∈ V
1  gains[0..k] ← 0
2  internal ← 0
3  for e ∈ I(u) do
4  |   if Φ(e, Π[u]) > 1
5  |   |   internal += ω(e)
6  |   for block i ∈ Λ(e) do
7  |   |   gains[i] += ω(e)
8  j ← arg max_{i∈[k]}(gains[i])
9  return j, gains[j] − internal
```

such that $g_i(u)$ can be written as

$$g_i(u) = b(u) - p_i(u) = \omega(\{e \in \mathrm{I}(u) \mid \Phi(e,i) > 0\}) - \omega(\{e \in \mathrm{I}(u) \mid \Phi(e,\Pi(u)) > 1\})$$

where the $\omega(\mathrm{I}(u))$ terms cancel each other out. Note that $\{e \in \mathrm{I}(u) \mid \Phi(e,\Pi(u)) = 0\} = \emptyset$ since clearly $u \in V_{\Pi(u)}$.

Due to our bitset implementation of connectivity sets, the running time for Algorithm 4.4 is still $\mathcal{O}(\deg(u) \cdot k)$. However in practice it is much faster due to skipping up to 64 not contained blocks in a single count-leading-zeroes instruction.

Note that this gain definition does not yield the correct value 0 for the current block of a vertex $g_{\Pi(u)}(u) \neq 0$. Therefore, we skip $\Pi(u)$ in the selection (line 8). While $p_{\Pi(u)}(u) = 0$ is guaranteed, $b(u) = 0$ is not. The intuition here is that after moving u out of its block, the benefit term would shift to the penalty, which then cancels out the original benefit, but this is of course not reflected.

4.3.3 Attributed Gains

Since we cannot rely on the correctness of computed gains when moving vertices in parallel, we additionally compute an *attributed gain* based on the synchronized writes to $\Phi(e,i)$. We attribute a connectivity reduction by $\omega(e)$ to the move that reduces $\Phi(e,i)$ to zero and an increase by $\omega(e)$ for increasing it to one. See lines 5-7 and 8-10 in Algorithm 4.5 Since we only lock one incident net at a time, this scheme may distribute the contributions to different threads. Hence, there is still no guarantee on the correctness of one attributed gain. However, the sum of the attributed gains equals the overall connectivity reduction. We use attributed gains to correctly track the connectivity metric through concurrent vertex moves, and as a secondary check to boost confidence. If both calculated and attributed gain agree that a move is good, we should perform it.

Algorithm 4.5: Move Vertex with Attributed Gains

```
   Input: vertex u ∈ V, source block s, target block t
1  Π[u] ← t
2  attributed ← 0
3  for e ∈ I(u) do
4      lock(e)
5      if (Φ(e, s) −= 1) = 0
6          attributed += ω(e)
7          remove s from Λ(e)
8      if (Φ(e, t) += 1) = 1
9          attributed −= ω(e)
10         add t to Λ(e)
11     unlock(e)
12 return attributed
```

4.3.4 Parallel Label Propagation

With these ingredients set up, we are ready to describe the parallel label propagation implementation. We employ a commonly used optimization named *active set*, which is an array A that contains all neighbors of vertices moved in the previous round. Initially the active set contains all boundary vertices.

In each round we iterate over the active set in parallel in random order and greedily move visited vertices. We determine the highest gain target block t using Algorithm 4.4, while already filtering target blocks that would exceed $L_{\max}$ if moved into and preferring lighter target blocks as a secondary criterion to gain. This is particularly useful for zero-gain moves. If the gain is positive, or zero and improves balance between the two blocks (block-weight$[\Pi(u)] >$ block-weight$[t] + c(u)$) we perform the move using Algorithm 4.5. We check again whether the result from block-weight$[t] \underset{\text{atomic}}{\mathrel{+}=} c(u)$ actually stays below $L_{\max}$ to guarantee a balanced partition. If $L_{\max}$ is exceeded or the attributed gain is less than zero, we revert the move.

If we keep the move, we insert u's neighbors into the active set A. This is implemented as a global vector that contains the vertices, filled from thread-local buffers. To avoid duplicate insertions, we use the atomic time-stamping from Section 2.2.4. We need to construct A explicitly in order to shuffle it, but we note that it is possible to instead iterate through all vertices in parallel and skip those not incident to nets of vertices moved in the previous round.

Equivalently to Algorithm 4.2 we employ a size limit on nets we scan, so that the work for activating neighbors in one round is $\mathcal{O}(p)$ and the depth is $\mathcal{O}(\max_{u \in A} \deg(u))$. Tracing attributed gains has the same work bound, but an additional $\mathcal{O}(\max_{e \in E} |e \cap A|)$ term in the depth for contention when updating $\Phi(e, i)$. This is negligible in practice because the

updates are well interleaved. The dominating term is the gain calculation. Due to the bitset implementation of $\Lambda(e)$, the work is $\mathcal{O}(\sum_{u \in A} \deg(u) \cdot k) \subseteq \mathcal{O}(pk)$ and the depth is the maximum work of one gain calculation $\mathcal{O}(\max_{u \in A}(\deg(u) \cdot k) + \log(|A|))$. As already mentioned this is much faster in practice; we observe behavior closer to $\mathcal{O}(p)$.

4.3.5 Parallel Gain Tables

For our FM algorithm, we propose to use a gain table as we do not move vertices immediately after exploring them for the first time. This enables repeatedly looking up gains in $O(1)$ and is a globalized way of updating the gains of vertices owned by other threads (more on this in the FM section). The idea of using a gain table is not new [KK98, Lar06, AHSS17], though it has gone out of fashion in recent years [SS11, ASS18a] due to the memory requirements. To the best of our knowledge, this idea has not yet been employed in a parallel setting and thus showing how to perform gain updates in parallel is novel.

We use atomic fetch-and-add instructions to update the gains as vertices are moved. Updates *trickle in* over time, at the same time as the gain values are read on other cores. Therefore, some observed gains may be not up to date. This guarantee can only be given if no updates are pending anymore. Still, this is the most accurate we can be in a parallel setting.

Our Approach. Instead of storing $g_i(u)$, we store $b(u)$ and $p_i(u)$ separately for each vertex u, so that changes to $b(u)$ only require one update, instead of updates to k gain values as is the case for sequential KaHyPar [AHSS17]. This approach uses $(k+1)n$ memory words (integers) in total. Now, let vertex u be moved from block s to t. For each net $e \in I(u)$, we update $b(u)$ and $p_i(u)$ using atomic fetch-and-add instructions as follows.

1. If $\Phi(e, s) = 0$, then $\forall v \in e : p_s(v)$ += $\omega(e)$
2. If $\Phi(e, s) = 1$, then $\forall v \in e \cap V_s : b(v)$ += $\omega(e)$
3. If $\Phi(e, t) = 1$, then $\forall v \in e : p_t(v)$ −= $\omega(e)$
4. If $\Phi(e, t) = 2$, then $\forall v \in e \cap V_t : b(v)$ −= $\omega(e)$

The update conditions are checked via the synchronized writes to $\Phi(e, s)$ and $\Phi(e, t)$ and refer to the values after the update (post-increment/decrement). In case 2 and 4 we only update one or two pins, respectively. In case 4 we already know one pin (u) but not the second, in case 2 we do not know the remaining pin, so we still need to iterate over the pins.

Maintaining Benefits. This approach has one down-side, which is however easy to fix. The benefit term $b(u)$ of a vertex cannot be correctly updated after it is moved. There is a race condition on $\Pi(v)$ in the check $\Pi(v) = s$ (case 2) or $\Pi(v) = t$ (case 4). When $\Pi(v)$

changes, we may perform a benefit update on v that was intended for a different vertex. The penalty values are not affected since they are independent of the pin's current block.

Our FM algorithm is organized in rounds in which each vertex can be moved at most once. Therefore, once u gets moved, we do not read $b(u)$ for the rest of the round. Due to the race condition it may still be written, which is why we recalculate $b(u)$ after the round is finished, instead of recalculating $b(u)$ for the new block *immediately* after the move.

We note that it is possible to correctly maintain benefit values throughout a round by using k benefit values $b_i(u) = \omega(\{e \in \mathrm{I}(u) \mid \Phi(e,i) = 1\})$ instead; one for each block even if the vertex is not in that block. The gain $g_i(u) = b_{\Pi(u)} - p_i(u)$ is calculated on demand, taking the benefit term associated with u's current block. The differences in how updates are performed are highlighted in the following (green for added, red for removed).

1. If $\Phi(e,s) = 0$, then $\forall v \in e : p_s(v)$ += $\omega(e)$, $b_s(v)$ −= $\omega(e)$
2. If $\Phi(e,s) = 1$, then $\forall v \in e$ ~~$\cap V_s$~~ : $b_s(v)$ += $\omega(e)$
3. If $\Phi(e,t) = 1$, then $\forall v \in e : p_t(v)$ −= $\omega(e)$, $b_t(v)$ += $\omega(e)$
4. If $\Phi(e,t) = 2$, then $\forall v \in e$ ~~$\cap V_t$~~ : $b_t(v)$ −= $\omega(e)$

However, this comes at the cost of an additional $(k-1) \cdot n$ memory words and two additional updates (case 1 and 3), which is why we use the first approach. Note that the additions in case 1 and 3 would only affect the just moved vertex in our first approach, which is why they can be safely omitted.

Correctness Proof. In the following we prove that our approach produces correct gain values. We have to specify and formalize what that means. We can perform updates for multiple moves in parallel, but we cannot observe gains at any possible time because updates trickle in asynchronously. Instead, we prove correctness in a scenario where we first finish all updates, then freeze execution before looking at the gains to see if they are correct.

Theorem 4.1. *Let $M = \{(u,s,t) \mid u \in V, s = \Pi(u), t \in [k] - \Pi(u)\}$ be a set of moves with pair-wise disjoint vertices $MV = \{u \mid \exists (u,s,t) \in M\}$. After performing the gain updates associated with M, each vertex $v \in V \setminus MV$ has correct $b(v)$, and each $v \in V$ has correct $p_i(v)$ terms.*

Proof. First, we note that the updates are correct in the sequential setting [San89]. Due to the atomic consistency of pin-count and gain updates, it suffices to prove correctness for arbitrary linearized (sequential) orders of updates. The remaining difficulty is that different orders may yield different intermediate values. However, due to commutativity we arrive at the same final $\Phi(e,i)$ values. Thus, it suffices to argue that gain updates triggered by $\Phi(e,i)$ += 1 cancel out those triggered by $\Phi(e,i)$ −= 1. This statement holds, as case 1 and 3 are complimentary, as well as case 2 and 4. Therefore, the final $p_i(v)$ and $b(v)$ values only depend on the final $\Phi(e,i)$ values. Furthermore, for $v \notin MV$ there is no race condition on $\Pi(v)$ such that updates to $b(v)$ are applied to the correct vertex. □

Algorithm 4.6: Gain Updates after Move

```
   Input: vertex u ∈ V, source block s, target block t
1  for e ∈ I(u) do
2      lock(e)                          // fused with updates for attributed gains if used together
3      Φ_s ← (Φ(e, s) −= 1)
4      Φ_t ← (Φ(e, t) += 1)
5      unlock(e)
6      if Φ_s = 0
7          for v ∈ e do
8              p_s(v) +=_atomic ω(e)
9      if Φ_s = 1
10         for v ∈ e do
11             if Π(v) = s
12                 b(v) +=_atomic ω(e)
13     if Φ_t = 1
14         for v ∈ e do
15             p_t(v) −=_atomic ω(e)
16     if Φ_t = 2
17         for v ∈ e do
18             if Π(v) = t and v ≠ u
19                 b(v) −=_atomic ω(e)
```

Update Complexity. We now analyze the complexity of performing gain updates in terms of work and depth, using the pseudocode in Algorithm 4.6 as a guide. To simplify the analysis, we assume that all vertices are moved exactly once, as moving only a subset incurs less work. The core to the argument is that each vertex is moved at most once. As a disclaimer, this assumption holds for traditional FM versions, but does not for localized FM in the way we use it (with local rollbacks). We discuss the differences and implications in the paragraph on local rollbacks in Section 4.3.7.

Lemma 4.2. *The work of gain updates for moving all vertices once is* $\mathcal{O}(\sum_{e \in E} |e| \cdot \min(k, |e|)) \subseteq \mathcal{O}(kp)$.

Proof. For each vertex we perform a loop over its incident nets, which results in $\mathcal{O}(p)$ iterations overall. Now we show that for each net, each of the conditions in line 6, 9, 13, 16 is true at most twice per block. This results in up to $\mathcal{O}(\min(k, |e|))$ updates per net that take $\mathcal{O}(|e|)$ work each. This leads to $\mathcal{O}(\sum_{e \in E} |e| \min(k, |e|)) \subset \mathcal{O}(kp)$ work for the nets, which is the dominating term of the overall work. The argument hinges on the requirement that each vertex is moved at most once. Only vertices that are in the block at the beginning of

the round can be removed from it; vertices that were moved into a block are *locked* in it.

The condition $\Phi(e, s) = 0$ is true at most once, since it requires $\Phi(e, s) = 1$ before the move. If $\Phi(e, s)$ is incremented to 1 again at any point, the responsible vertex is locked in s. Therefore $\Phi(e, s)$ cannot be decremented to 0. The same argument holds for $\Phi(e, t) = 1$, which requires $\Phi(e, t) = 0$ before the move. After the increment, the responsible vertex is locked in t, which prevents the prerequisite $\Phi(e, t) = 0$. Therefore the conditions in line 6 and 13 are each triggered at most once. Note that for this argument the roles of s (moved out of) and t (moved into) matter.

The condition $\Phi(e, s) = 1$ can be triggered at most twice: one move out of s at $\Phi(e, s) = 2$, followed by one into s, and finally the last move out. The round may start with $\Phi(e, s) > 2$ but only the last two remaining matter here. Again it is important that we trigger the update only when we decrement. Similarly $\Phi(e, t) = 2$ can be true at most twice: one move in at $\Phi(e, t) = 1$, one out, and one in.

□

The proof uses the same argument as Fiduccia and Mattheyses for 2-way partitions [FM82], except they prune the second invocation in both benefit updates at the expense of additional memory. This bound is unfortunately tight in the worst case. Think of a net with $k/2$ pins in distinct blocks that are each moved to a distinct one of the remaining $k/2$ blocks. Each of the $k/2$ moves triggers two penalty updates. Creating disjoint copies of this instance yields arbitrary size instances. Note also that due to the $\min(k, |e|)$ term, this bound matches the $\mathcal{O}(m)$ bound for k-way gain updates on plain graphs, i.e., we did not mess up the algorithm, updating gains with larger hyperedges is simply more expensive.

On real-word instances we observed work much closer to $\mathcal{O}(p)$ since most nets have constant size. With the same proof we can also show a (situationally) more strict $\mathcal{O}(\sum_{e \in E}(\lambda_{\text{pre}}(e) + \lambda_{\text{post}}(e)) \cdot |e|)$ bound, where $\lambda_{\text{pre}}(e)$ is the size of $\Lambda(e)$ before all moves and $\lambda_{\text{post}}(e)$ the size after. This bound resembles the behavior in practice.

Lemma 4.3. *The worst-case work for one vertex is* $\mathcal{O}(\max_{u \in V} \sum_{e \in \mathrm{I}(u)} |e|)$. *The depth of all gain updates is* $\mathcal{O}(\max_{u \in V} \sum_{e \in \mathrm{I}(u)} |e| + \log(n))$.

Proof. Each incident net of one vertex may require a gain update, which shows the worst-case work for one vertex. The depth for all gain updates is the per-vertex work plus a $\log(n)$ term for the parallel loop control overhead. □

We can perform all loops in Algorithm 4.6 in parallel. Let $\sigma := \max_{e \in E} |e|$ and let $\Delta := \max_{v \in V} \deg(v)$. Then the new depth bound would be $\mathcal{O}(\sigma + \Delta)$, not the desired $\mathcal{O}(\log(\sigma) + \log(\Delta))$ because we have to update $\Phi(e, i)$ with a lock $|e|$ times (same lock for all blocks), and there are up to $2 \deg(u)$ atomic updates per benefit/penalty value per vertex. If not all vertices are moved, these bounds become better. Our implementation does not parallelize these loops, since extremely large nets often have more than 2 pins in each block so that they rarely trigger updates, and due to the hidden moves variant of FM with thread-local partition data that we describe in Section 4.3.7, which would require parallel access to that data.

Algorithm 4.7: Gain Table Initialization

```
1  for u ∈ V do in parallel
2      b(u) ← 0
3      p_i(u) ← 0 ∀i ∈ [k]
4      w ← 0
5      for e ∈ I(u) do in parallel
6          w +=_reduce ω(e)
7          if Φ(e, Π(u)) = 1
8              b(u) +=_reduce ω(e)
9          for i ∈ Λ(e) do
10             p_i(u) −=_reduce ω(e)
11     for i ∈ [k] do
12         p_i(u) ← w − p_i(u)
```

Initialization. So far we have shown how to update the gain table; now we show how to initialize it on each level using Algorithm 4.7. The approach is similar to the gain calculation in Algorithm 4.4. We employ the same trick to avoid an $\mathcal{O}(k)$ loop over the blocks for each incident net by leveraging connectivity sets. Again, due to the bitset implementation this does not eliminate the $\mathcal{O}(k)$ term in theory, it is just substantially faster in practice. The difference to Algorithm 4.4 is that we have to extract benefits and penalties separately. For the inner loop we use a reduce operation since atomic instructions would cause too much contention. This leads to $\mathcal{O}(kp)$ work and $\mathcal{O}(\log(n) + \log(\Delta) + k)$ depth. The work bound matches the one for the updates, whereas the initialization depth contains an additive $\mathcal{O}(k)$ that is missing in the updates depth, because each move only touches two blocks.

Algorithm 4.8: Distribute Gains of Net

```
   Input: net e, move sequence M, vertex positions pos in M
1  first-in[0..k] ← ∞, last-out[0..k] ← −∞, rem-pins[0..k] ← 0
   // first pass: set up data
2  for v ∈ e do
3      if pos[v] ≠ ⊥ // vertex was moved
4          m ← M[pos[v]]
5          first-in[m.t] ← min(first-in[m.t], pos[v])
6          last-out[m.s] ← max(last-out[m.s], pos[v])
7      else
8          rem-pins[Π(v)] += 1
   // second pass: distribute gains
9  for v ∈ e do
10     if pos[v] ≠ ⊥ // vertex was moved
11         s ← M[pos[v]].s, t ← M[pos[v]].t
12         if last-out[s] = pos[v] and first-in[s] > pos[v] and rem-pins[s] = 0
13             gain[pos[v]] +=_atomic ω(e)
14         if first-in[t] = pos[v] and last-out[t] < pos[v] and rem-pins[t] = 0
15             gain[pos[v]] −=_atomic ω(e)
```

4.3.6 Parallel Gain Recalculation

Finally, we propose a parallel algorithm to recompute exact gains of vertex moves if they are supposed to be performed in a given order, as is the case for FM. This approach is of independent interest to graph partitioning, as Mt-KaHiP [ASS18a] does this step sequentially. The parallel gain recalculation is even easier to implement for plain graphs. But this is one scenario (of certainly many) where transferring the problem to hypergraphs revealed more insight into the problem. Considering edges as *first-class entities* enables us to change perspective, which leads to the idea of distributing gains from nets to vertices.

Given a sequence of vertex moves $M = \langle m_0, \ldots, m_{r-1} \rangle$, we want to compute the exact gain of each move, as though they were performed sequentially in this order. A move m_j is represented as a tuple (u, s, t) meaning that vertex u is moved from block s to t. We use an array pos$[0..n]$ to denote the position of a vertex in M. Note here that this requires that each vertex is moved at most once per round, as in the previous section.

Our idea is to identify at which position block i would have $\Phi(e, i) = 0$ or $\Phi(e, i) = 1$ for each $e \in E, i \in [k]$ and accordingly attribute the associated gain to the move in that position. This is shown in Algorithm 4.8. For a given net e, we determine the positions of the first pin moved into i and the last pin moved out of i for each $i \in [k]$. If no pin of e was moved out of i, this position is $-\infty$, and similarly ∞ for the first moved in. Additionally, we compute

Algorithm 4.9: Parallel Gain Recalculation

```
   Input: Move sequence M = ⟨m_0, ..., m_{r-1}⟩, FM round ID: round
1  for i = 0 to r with m_i = (u, s, t) do in parallel
2  |   pos[u] ← i
3  |   gain[i] ← 0
4  for i = 0 to r with m_i = (u, s, t) do in parallel
5  |   for e ∈ I(u) do in parallel
6  |   |   if CAS(&last-recalc-round[e], round)
7  |   |   |   DistributeGainsOfNet(e, pos) // Algorithm 4.8
8  for i = 0 to r with m_i = (u, s, t) do in parallel
9  |   pos[u] ← ⊥
```

rem-pins$[i]$, the number of pins in block i at the beginning of the round, minus the ones that were moved out.

If rem-pins$[i] = 0$, all pins of e that were initially in V_i were removed over the course of the round, or there were none in the beginning. In the former case (last-out$[i] > -\infty$) there is a chance that i was removed from $\Lambda(e)$ since there may be a point at which $\Phi(e, i) = 0$. This happens exactly if last-out$[i] <$ first-in$[i]$, i.e., no pin moves into i before the last initial one leaves. Therefore the pin $v \in e$ with last-out$[i] =$ pos$[v]$ and pos$[v] <$ first-in$[i]$ is awarded an $\omega(e)$ reduction, see line 12 and 13. Note that first-in$[i]$ may be ∞, in which case no pin was moved into i.

Similarly, if there are moves into i (indicated by first-in$[i] < \infty$) and rem-pins$[i] = 0$, there may be a point at which i is *newly* added to $\Lambda(e)$. It may have been removed at some point (in which case the previous case awarded gain to the earlier move) or not been part of $\Lambda(e)$ at the beginning of the round (last-out$[i] = -\infty$). Block i is added newly, exactly if last-out$[i] <$ first-in$[i]$ and rem-pins$[i] = 0$, i.e., the first pin newly moved into i comes after the last initial one moved out (if any). The $\omega(e)$ connectivity increase is attributed to the vertex v with first-in$[i] =$ pos$[v]$, see line 14 and 15. It is rather unintuitive that this condition last-out$[i] <$ first-in$[i]$ looks exactly like the condition for the other case (line 12 and 13). The difference is in the role of i as the target block of v.

We now know how to assign the gain contributions of one net e. We could simply run Algorithm 4.8 for each $e \in E$ in parallel to obtain an $\mathcal{O}(p)$ work algorithm. However, we would like to run it only on nets incident to moved vertices, since that set may be substantially smaller. Let $E_m := \bigcup_{j \in [r]} \mathrm{I}(m_j.u)$ denote this set. The second parallel for loop in Algorithm 4.9 shows how we can construct E_m on-the-fly in parallel and call Algorithm 4.8 on each contained net. We iterate in parallel over the moved vertices and their incident nets. To avoid duplicate calls on the same net we use the atomic time-stamping from Section 2.2.4.

Theorem 4.4. *Algorithm 4.9 assigns correct gain values to each move in a given sequence $M = \langle m_0, \ldots, m_{r-1} \rangle$ as if they were applied in this order, assuming the vertices in M are unique.*

Algorithm 4.10: Parallel Localized FM

```
1  while improvement found and less than maximum rounds performed do
2      work-queue ← {v ∈ V | ∃e ∈ I(v) : λ(e) > 1}
3      do on each thread
4          while seeds ← work-queue.tryPopMultiple(num seeds) do
5              insert seeds into thread-local PQs
6              while not done do
7                  find best move in PQs
8                  perform best move
9                  claim and insert/update neighbors into PQs
10             rollback to best local prefix
11     rollback to best global prefix
```

Let MV denote the set of moved vertices.
It takes $\mathcal{O}(\sum_{u \in MV} \deg(u) + \sum_{e \in E_m} |e|) \subseteq \mathcal{O}(p)$ work and the depth is $\mathcal{O}(\max_{e \in E_m} |e| + \log(r) + \max_{u \in MV} \log(\deg(u)))$.

Proof. Correctness follows directly from the previous paragraph. The first work term stems from iteration over incident nets of moved vertices, the second term stems from iterating over pins of nets incident to moved vertices. This also constitutes the first term in the depth, whereas the other two stem from loop control. □

The data structures in line 1 of Algorithm 4.8 would incur $\mathcal{O}(k)$ overhead for initialization. We use thread-local objects that are reset to a clean state by performing a third pass over the pins if $|e| < k$. Since thread-local objects do not fit the work-depth model we omitted this term. The $\mathcal{O}(k)$ term for initialization could be eliminated by using a folklore array with constant time initialization [Meh84], though this is not necessary in practice. Algorithm 4.8 could be further parallelized by using priority-writes for the min/max operations (early-exit compare-and-swap loops) and a fetch-and-add instruction for rem-pins.

4.3.7 Parallel Localized FM

We have now gathered all ingredients to describe our version of parallel localized FM. Recall that sequential FM consists of two phases: *finding a sequence of moves* by repeatedly performing a feasible move with highest gain, followed by *reverting moves back to the prefix with the highest cumulative gain* in that sequence.

We implement a relaxed version of the first phase that performs non-overlapping *localized* FM searches on different threads to obtain a global sequence of moves M. In the second phase we use our new parallel gain recalculation (Section 4.3.6) to assign exact gains to moves in M. We perform a prefix-sum (gains) and reduce operation (for selection) to identify

the highest gain prefix of M. Since we have no guarantee that each move maintains balance, we additionally filter out prefixes that do not yield a balanced partition. We do this by incorporating the block weights into the prefix sum operation. In the following, we describe how to implement the first phase.

Algorithm 4.10 shows high-level pseudocode for parallel localized FM that also fits the version of Akhremtsev et al. [ASS18a] in Mt-KaHiP. We perform multiple global rounds in which each vertex is allowed to be moved once (line 1). Thread-local data structures are highlighted in red, globally shared data in blue. In each round we collect all boundary vertices in a global work queue and shuffle it (line 2). Each thread polls up to a fixed number of seed vertices from the work queue (line 4).

Starting with the seeds and expanding to neighbors of moved vertices, we perform k-way FM, i.e., repeatedly perform the highest gain move (line 7-9) among the feasible ones. A search ends once there are no vertices left (empty PQs) or the adaptive stopping rule by Osipov and Sanders [OS10] is triggered. This stopping rule is also used by Mt-KaHiP and sequential KaHyPar.

Since we may perform negative gain moves to escape local minima, we revert back to the best prefix of our thread-local sequence and save it (line 10). At this point, we try to poll new seeds to start a new search, or terminate the thread's task if the work queue is empty (line 4). Once all threads are finished, the global rollback (second phase) is performed (line 11).

Mt-KaHiP's Version and Two Issues. For this, the local move sequences have to be combined into a global sequence. Akhremtsev et al. [ASS18a] just concatenate the sequences ordered by thread IDs, first all moves of thread 1, then thread 2, etc. We argue that this is bad since it will more likely revert moves by threads with higher ID in the global step. This disincentives using many cores. There is no mechanism to assign promising seeds to threads with lower ID; in fact the work queue is randomized. Note that each thread performed multiple searches and thus has multiple sequences ordered by time.

This problem is further aggravated by the fact that threads do not communicate their moves with each other. The moves are kept private until the end of the round. Each search only knows the moves of searches previously run on the same thread. Therefore, the searches can make arbitrarily bad move decisions, as only outdated information on the partition is available. The differences to the global partition are stored in thread-local data: a hash table for the block IDs and an array for the block weights.

Our Version. Note that hiding *some* moves is actually a good idea. Towards the end of a local move sequence we have only negative gain moves, which we need to revert. These are performed in the hope of enabling positive gain moves later on, to steer out of a local minimum. Applying them to the global partition and later reverting would *confuse* the searches on other threads, as they can influence the other searches' move decisions.

We want to incorporate more up-to-date global information (gains and block weights) and get rid of the arbitrary move ordering, while retaining the benefits of hiding bad moves.

Therefore, we apply the best local prefix to the global partition at the end of a search. Moves are inserted into the global move sequence in the same order as they are applied to the global partition. We use the gain table from Section 4.3.5 to provide up-to-date gain values for all vertices based on all globally available moves so far. In our experiments, we show that hiding moves is very important for high partition quality. While we expected it to be slightly slower than applying directly to the global partition, and this was the case in an early version, the local-first version is now faster thanks to several implementation-level optimizations.

First we discuss issues related to our local partition data structures and the local rollback. Subsequently we show how to infuse state from the central gain table into the thread-local FM state. This happens in two places: when finding the next move and when updating the neighbors' gains after the move. To thoroughly describe this, we outline low-level details of our k-way FM implementation.

Local Partition Data. Like Mt-KaHiP we keep block IDs in a hash table and block weight deltas in an array. However, as opposed to Mt-KaHiP we use a gain table and $\Phi(e, i)$ to get good performance (no $\Lambda(e)$ needed for our version). Unfortunately deltas for these have to be stored in hash tables as well. We add the benefit and penalty and values in the central gain table to the deltas in the thread-local table to obtain the actual gain. To reduce their memory overhead, we actually apply local move sequences to the global partition *as soon as we find an improvement*, i.e., the sum of the local gains so far is positive. This allows us to clear all hash tables, while still hiding all bad moves.

In rare cases, we try to move vertices with extremely high degree and extremely negative gain, as the stopping rule only considers such gains once the move is made and there is a minimum number of moves to be made. These bloat the hash tables with useless updates, since the stopping rule will trigger in a few steps and we will revert the move. Therefore, we skip high degree moves if they do not yield an overall improvement (sum of local gains so far > 0). We omit the updates for the last move, since the local deltas are cleared right after. This ensures that we can move high degree vertices (if they net an improvement) without bloating the hash tables.

Local Rollback. When applying a local move sequence to the global partition, we use attributed gains to double-check whether a shorter prefix may be even better and if so revert back to it. Since these moves are visible only shortly and it happens very rarely, this still keeps bad moves sufficiently hidden from searches on other threads. To determine its position in the global sequence, we atomically increment an offset by the length of the local sequence.

Note that performing local rollbacks breaks the $\mathcal{O}(kp)$ work bound we can prove for gain updates on the global partition. Think of a very large net e and a block i with $\Phi(e, i) = 0$ (this combination occurs rarely). Moving a pin of e into i removes the $\omega(e)$ penalty for further pins to move into i, causing a $\Theta(|e|)$ gain update on the local partition. If the move is locally reverted, the next search can move a second pin of e into i, seemingly for the first time; thus requiring the same gain update. This can happen up to $|e|$ times, leading to $\Theta(|e|^2)$ work for

Algorithm 4.11: Acquire or Update Neighbors

```
   Input: move (u, s, t), search ID z
1  E_g ← {e ∈ I(u) | Φ(e,s) ≤ 1 or Φ(e,t) ≤ 2}          // observe values at move
2  seen[0..n] ← false
3  for e ∈ E_g do
4      for v ∈ e do
5          if seen[v] = false
6              seen[v] ← true
7              if search[v] = z
8                  update gains of v to values in gain table          // already own v
9              else if search[v] = ⊥ and CompareAndSwap(&search[v], ⊥, z)
10                 insert v into PQs with values in gain table        // claimed v
```

gain updates.

We emphasize that this is not an artifact of the parallelization or hiding local moves from other searches. It equally affects any localized FM implementation with local rollbacks, such as the one in sequential KaHyPar. This stems purely from not adhering to the rule of moving at most once per round. In practice, this bad behavior does not appear to occur frequently, otherwise it would be highly unlikely that the run finishes.

Vertex Ownership. The vertices are exclusively owned by threads. Each localized search is assigned an ID when it starts. We keep an array search$[0..n]$ that maps a vertex to the ID of the search that owns it, or to a special value $\perp$ if it is not owned. To acquire a vertex, we first check if it is owned and if not, perform a compare-and-swap with the search ID to acquire it (see line 9 in Algorithm 4.11). If a search ends because the adaptive stopping rule is triggered, we still own vertices, which we release (set search$[v] \leftarrow \perp$) so that other searches may acquire them.

Acquire or Update Neighbors. In our case, vertices are acquired at the beginning of a search (seeds) and after a move (neighbors). While it is possible to acquire a vertex only once we move it, doing so at the time it is added to the search allows us to use just one globally shared array for the priority queue handles (positions in the heaps). Algorithm 4.11 shows pseudocode for acquiring neighbors after a move. If they are already owned, we update their gain in the PQs.

During the move we update the $\Phi(e,s), \Phi(e,t)$ values for $e \in I(u)$. We use these to track the nets E_g whose pins have received a gain update ($\Phi(e,s) \in \{0,1\}, \Phi(e,t) \in \{1,2\}$), see line 1. Since the update is so far only reflected in the gain table, we now propagate it to the priority queues of the search. If a neighbor is already owned by our search, we update its gains based on the gain table, otherwise we try to acquire it. This way, we gradually infuse

Algorithm 4.12: Find Best Feasible Move

```
1  if block-PQ.empty()
2      return ⊥
3  while true do
4      s ← block-PQ.top()
5      u ← vertex-PQs[s].top()
6      PQ gain ← vertex-PQs[s].topKey()
7      t ← arg max_{i∈[k]} p_i(u)                    // balance tie break, filter infeasible
8      if b(u) − p_t(u) ≥ PQ gain                    // success. return move
9          vertex-PQs[s].deleteTop()
10         if vertex-PQs[s].empty()
11             block-PQ.remove(s)
12         return u, t, b(u) − p_t(u)
13     else                                          // retry. refresh PQ entries
14         vertex-PQs[s].adjustKey(u, b(u) − p_t(u))
15         if vertex-PQs[s].topKey() ≠ block-PQ.keyOf(s)
16             block-PQ.adjustKey(s, vertex-PQs[s].topKey())
```

global partition state into the thread-local search state without resorting to message passing. In particular, this keeps information fresh in the direction the search is currently expanding.

Note that with E_g we may not acquire all neighbors of u, but we do encounter all that have become boundary vertices through the move. Only boundary vertices can have positive gain, so this is a perfectly acceptable choice. This can still miss vertices that were on the boundary prior to any move, but these are in the global work queue and will thus be seeds of their own search. Similar to Algorithm 4.2 and sequential KaHyPar, we filter out nets with $|e| > \eta$ to accelerate updates in the presence of large nets, as it is unlikely that vertices that are only incident to large nets admit a positive gain move [Sch20]. Vertices that share small nets with the just moved vertex receive their updates. We use a bitset (line 5 and 6) to avoid performing duplicate updates, implemented with timestamping to avoid manual resets. The bitset is initialized once per thread, not once per call to Algorithm 4.11.

Priority Queue Layout. A vital part of an efficient k-way FM implementation is the way priority queues are used to extract the next move. There are several variants which offer different capabilities and tradeoffs, which we discussed in Chapter 3. We chose to implement from-PQs by Träff [Lar06], due to their tie-breaking capability when selecting blocks to move from. Since each vertex is in only one block, only one array is needed to track positions in the k heaps, even in our parallel setting due to exclusive vertex ownership.

Algorithm 4.12 shows how we extract the next move using from-PQs [Lar06]. The block-PQ is the PQ to select the best block s to move from (line 4) and vertex-PQs$[s]$ is the PQ to

select the highest gain vertex $u \in V_s$ to move (line 5). We omit the per-vertex PQs to select the highest gain target block t, due to the excessive memory requirement. Instead, we select t from scratch (line 7) using the gain table.

With this, we can react to changes on the global partition that are not yet reflected in our local PQs. We may have missed some other move that is now better due to outdated local state. We consider entries from the gain table as more trustworthy than the local PQ entries. Additionally, we filter infeasible moves at this stage, as hidden local moves of other threads may have made the designated move infeasible. If the gain stored in the PQ is better than the table entry (line 8), we adjust the gain of u (line 14) and retry with a different vertex (line 3). Otherwise, we return the found move (line 12) after updating vertex-PQs$[s]$ and the block-PQ (line 9-11).

4.3.8 Rebalancing

While our refinement algorithms are guaranteed to produce balanced partitions, intermediate balance violations can improve solution quality. The intuition behind this is that localized FM searches may each find good improvements but their combination is barely infeasible. Hence, we relax the balance constraint for the global rollback step by a small amount, multiplying the imbalance parameter ε by a small factor. Based on preliminary experiments we use 1.25, which turns $\varepsilon = 0.03$ into $\varepsilon' = 0.0375$. Often, label propagation refinement is able to rebalance the partition with zero gain moves.

If the partition is still imbalanced on the finest level, we rebalance it using an approach that is similar to label propagation. Since we allow only a small amount of imbalance, the amount of work to rebalance the partition is small as well. In this case, we prefer worsening the partition by as little as possible, even if the algorithm is not as efficient as it could be.

We iterate over the vertices in overloaded blocks in parallel and compute their highest gain feasible move, but ignore staying in the current block. Non-negative gain moves are performed immediately. Negative-gain moves are collected for a second pass, disregarding that their gains may no longer be correct then.

If the partition is still imbalanced after the pass with non-negative gain moves, we perform a second pass where we perform the negative gain moves. One way to prioritize higher gains is to sort, and apply them sequentially until balance, but this lacks gain accuracy. Instead, we distribute the negative gain moves to thread-local PQs. Each thread polls moves from its PQ, and recomputes the best target block and according gain. If the current block is not overloaded anymore, the move is discarded. If the fresh gain is not worse than the one in the PQ, the move is applied, otherwise the vertex is reinserted into the PQ with the new gain. To emulate the prioritization of gains, a thread spins idle if its top gain is worse than the global top gain by more than a small threshold (we use 5). Under normal circumstances this is unacceptable in terms of performance, but here it works because the imbalance is very small and thus only few moves are considered.

Structure of Experiments

We have concluded the algorithmic description of our multilevel algorithm, and now turn to evaluating it experimentally. The experiments are structured as follows. First, we evaluate the different components and design choices made in our algorithms (Sections 4.4 and 4.5). In Section 4.5 we also consider techniques from a competing refinement algorithm and demonstrate that our choices are superior. Following this, we assess how scalable each component and the overall algorithm are in Section 4.6. Subsequently, we perform a horse-race comparison with state-of-the-art partitioning algorithms in Section 4.7, first with fast and parallel algorithms, then with slower sequential algorithms. This is the centerpiece of the evaluation. Finally, we perform a parameter study to explain configuration choices for a subset of the parameters. The volume of experiments has been substantially expanded from the conference publication, which contained only the horse-race comparisons on set A and B.

We call our algorithm Mt-KaHyPar-D. The Mt stands for multi-threaded, and the D stands for default configuration, as we present further configurations in the following chapters. We use the experimental setup as described in Section 2.4.

4.4 Evaluating the Algorithmic Components

The component evaluation experiments are conducted on set B and machine B with 64 threads, not the parameter tuning benchmark set. The order is as in the multilevel framework: first preprocessing, then coarsening, and ultimately refinement, which makes up the largest part. Except for the variable part, the algorithm uses the default configuration discussed in Section 4.8.

Community Detection. In Figure 4.2 we show performance profiles for the final and initial partition quality, with and without community detection preprocessing. The preprocessing significantly improves both metrics substantially, though the effect is larger on initial partitions as should be expected. The quality difference warrants enabling this component in the main configuration, particularly considering the contribution to the overall running time (which we show later). This approach's effectiveness was already shown in prior works [Sch20] with sequential codes, but we reconfirm this result for our algorithm.

Coarsening Locking Schemes. Next we investigate the impact of the locking scheme employed for clustering in the coarsening phase. We proposed a two-stage locking approach called `ResolveByMerge` in the following. We compare this with a version that performs no locking at all (which thus corresponds to label propagation coarsening as for example employed in Mt-KaHiP [ASS20]), and two versions of the locking scheme by Catalyürek et al. [ÇDKU12] for the parallelization of PaToH's coarsening. `LockByRating` tries to lock all candidates in sorted order of their rating, stopping as soon as it succeeds, whereas

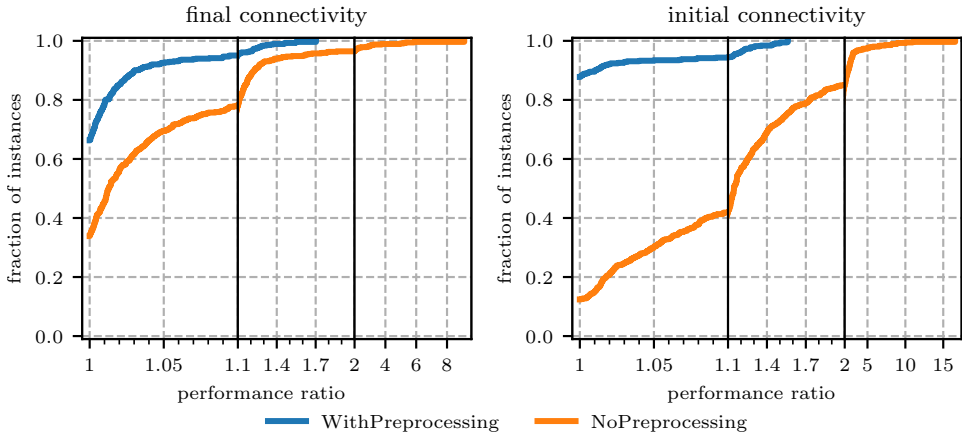

Figure 4.2: Impact of community detection preprocessing on partition quality: for final partition and initial partition.

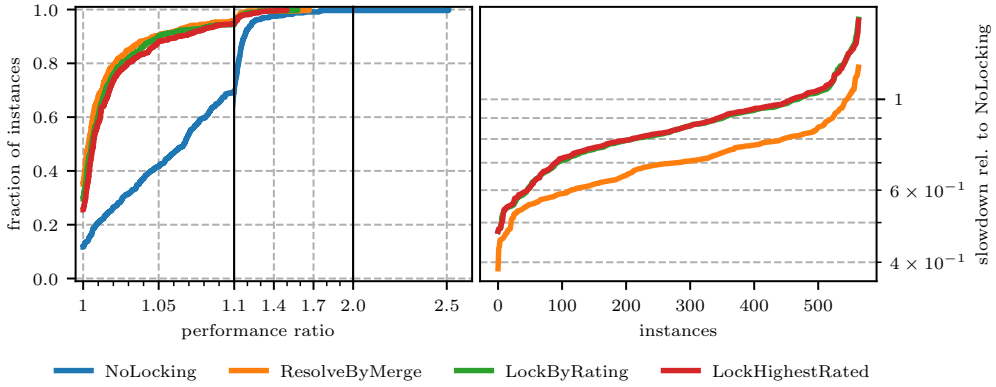

Figure 4.3: Partition quality and running time of different locking schemes during coarsening. The running time plot (right) considers coarsening time, not total partitioning time.

`LockHighestRated` only tries the highest rated candidate and then gives up. Both versions lock the current vertex before calculating ratings as this was faster.

Figure 4.3 shows the quality and running time results. In terms of quality, our approach `ResolveByMerge` comes out barely ahead of the two PaToH variants and far ahead of `NoLocking`. `LockByRating` and `LockHighestRated` barely differ, which is a strong indication that locking the first candidate usually succeeds. In terms of running time ours is the fastest because it uses less locking than PaToH (joins to multi-vertex clusters are immediately

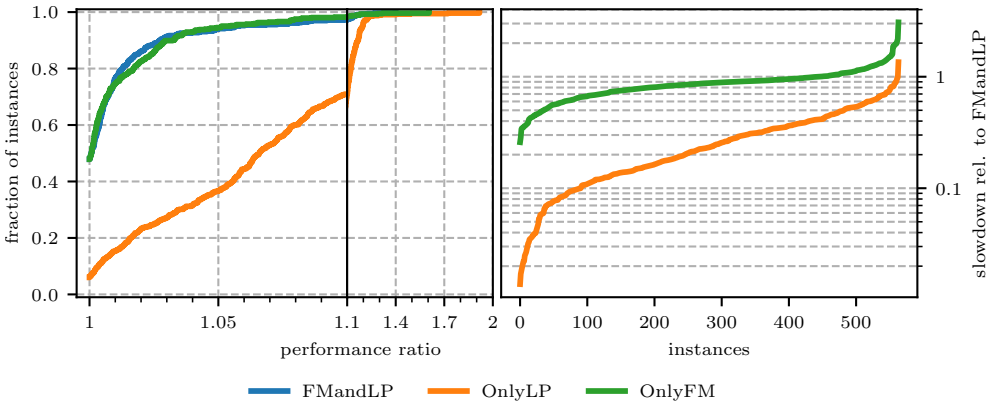

Figure 4.4: Partition quality and running time when using just label propagation, just FM or the combination of LP and FM. The running time plot (right) considers the refinement time (LP + FM).

approved), and because it encourages more progress when the instances are small, due to merging more vertices rather than rejecting joins (`LockByRating` and `LockHighestRated`) or performing cyclic joins which lead to oscillation (`NoLocking`).

We are not definitively certain where the severe quality penalty for `NoLocking` stems from. Due to oscillation, we do not reach the contraction limit as often, and smaller contraction limits lead to better partition quality, as shown later in the parameter study. However, our main idea is that we do not properly control the coarsening progress. We create level $i+1$ as soon as $\frac{|V_i|}{|V_{i+1}|} > \beta$, where β is the shrinking parameter. During a coarsening pass, we track $|V_{i+1}|$ by incrementing a counter for each approved cluster join (thread-local deltas, zero-initialized, synchronized from time to time), which is subtracted from $|V_i|$. Due to the agglomerative property of the algorithms with locking this method is accurate; over the course of the pass $|V_{i+1}|$ only becomes smaller. However, it fails without locking as two vertices joining each other at the same time cause two increments, while subsequently both are still singletons. Here, we could accurately track $|V_{i+1}|$ via the events ”cluster weight becomes 0” and ”cluster weight becomes > 0” observed with atomic instructions as triggers for decrementing/incrementing the number of clusters, but we did not investigate further.

Combining LP and FM Refinement. In the Mt-KaHiP paper, Akhremtsev et al. [ASS20] claim that combining LP and FM refinement leads to better running time than using only FM, because LP finds all of the easy improvements in shorter time, such that FM needs to perform less work to find the non-trivial improvements. We dispel this claim in Figure 4.4, at least for our benchmark instances. Using only FM is a bit faster than the combination FM + LP on the majority of the instances. Even the tail at slowdowns < 0.5 is larger than the tail

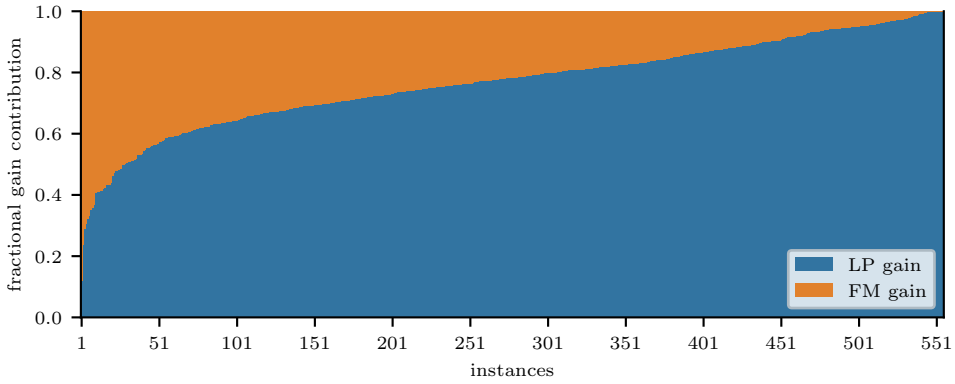

Figure 4.5: Contribution to the overall gain made by LP and FM refinement plotted per instance.

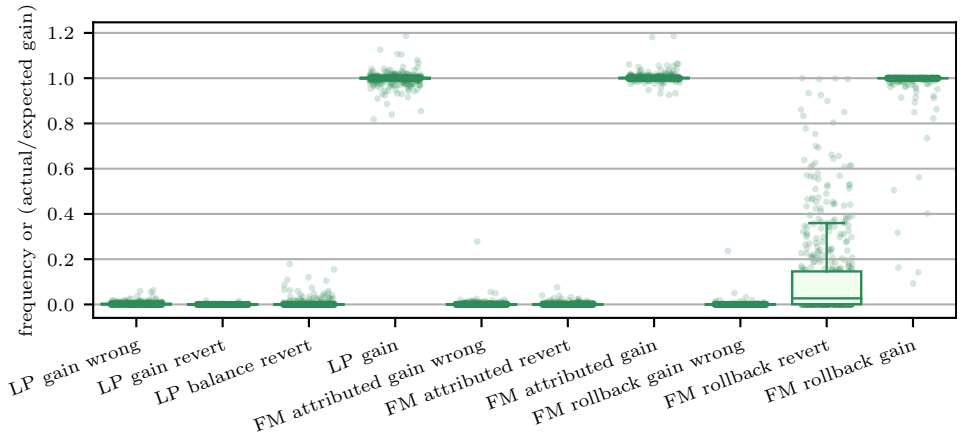

Figure 4.6: Intervention frequency of gain accuracy techniques, and overall gain fluctuation.

at > 2. There is almost no quality difference between only FM and the combination, though the combination looks barely ahead. Using only LP is not viable if good partition quality is desired.

In Figure 4.5 we plot the fraction each of the two refinement algorithms makes to the overall gain, that is the improvement made from the initial to the final partition. LP makes the larger contribution since it is run first and thus able to find the easy improvements. Considering this, the contribution of FM is quite impressive.

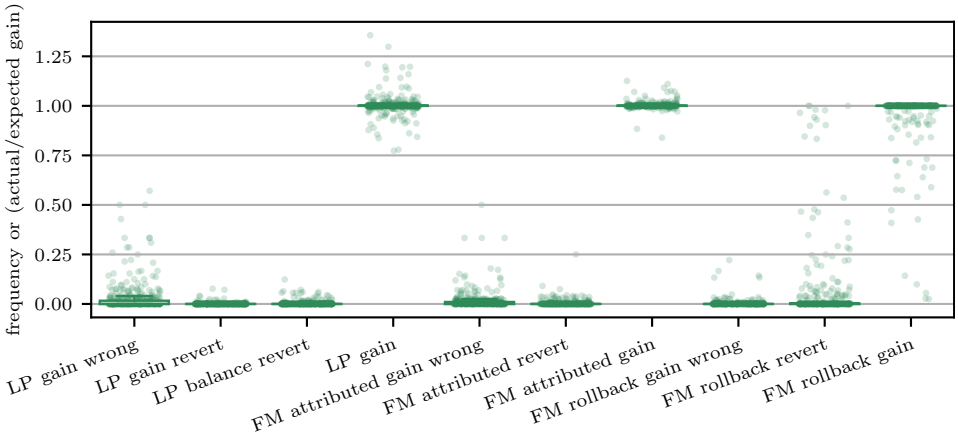

Figure 4.7: Intervention frequency of gain accuracy techniques, and overall gain fluctuation, but restricted to the coarsest level.

Gain Accuracy. Up next, we investigate how often our techniques for accurate gains have to intervene because the calculations were wrong. Figure 4.6 shows combined box-and-scatter plots for the frequency of various events (relative to the total number of moves in that part, all below 1), as well as how this affected the overall gain (LP gain, FM attributed gain, FM rollback gain; fluctuate around 1). We plot how often the gains are wrong, whether this lead to a revert, and whether moves had to be reverted due to accidental balance violation. The plots are separated for LP, FM at local rollbacks (attributed), where we compare the calculated gain from the gain table with the attributed gain, and FM at global rollbacks (rollback), where we compare the attributed gain with the recalculated gain within the move sequence. For FM we cannot distinguish whether the revert is due to negative gain or balance violation, since a combination of moves, and only non-violating prefixes are considered. To clarify, we plot the number of reverted moves and total moves calculated, not the number of local rollbacks that perform reverts.

It is astonishing how rare incorrect gains are and how small the effect on the overall gain is, even for attributed gains with FM, where some time passes between moving the vertex locally and applying the move to the global partition. Consequently the effect on the actual gain is rather small, though in some cases the overall gain is even better. Out of the three contexts, the gain difference is most pronounced at the global rollback, making the parallel gain recalculation the most important of our accuracy techniques.

Reverts are much more often triggered by balance violations, up to 20% of all moves for LP on an outlier instance. For FM global rollbacks this is expected because move sequences that were not coordinated with one another are combined; for LP not so much since a check is performed before the atomic block weight update incurred by the move.

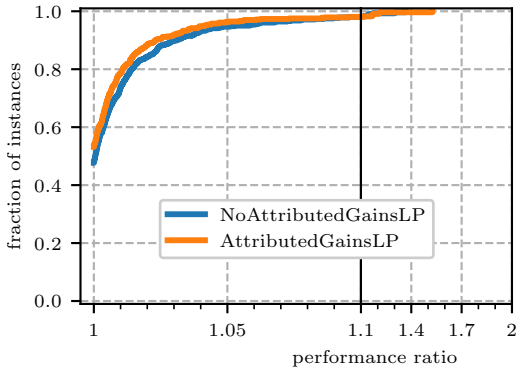

Figure 4.8: Impact of attributed gains on partition quality with LP refinement. For these measurements FM was turned off.

The numbers are tracked for the main uncoarsening (actual k, no recursive bipartitioning) and aggregated across all levels. Interference between threads is more prominent on coarser than on finer levels, and more moves are possible on finer levels. Therefore Figure 4.7 shows the same statistics measured on just the coarsest level. As can be expected, we see that the calculated gains in LP and FM are wrong more often, but interestingly balance violations occur less frequently.

Attributed Gains. Considering the results in the previous paragraph, we cannot expect a significant quality improvement from using attributed gains. In Figure 4.8 we compare the quality of just label propagation refinement (FM turned off) with and without attributed gains. The difference is tiny. However, the running time impact is tiny as well, so the technique can be used without considering a trade-off. Furthermore, we have a follow-up work with substantially more interference between threads (Chapter 6), where attributed gains have a bigger impact.

Gain Tables. Next we evaluate the parallel gain table used during FM and two alternative methods that were preferred in recent years: recomputing gains from scratch [ASS20] and using to-PQs [Sch20] with delta-gain updates. Additionally, we consider a variant where the gain table entries of a vertex are only initialized once it is acquired, instead of initializing all entries before FM starts. We call this `GainTableOnDemand`.

Figure 4.9 shows the results. Recomputing gains from scratch is not viable. With the gain table, FM takes 80 seconds on our largest instance `sk-2005` for $k = 64$, whereas the version that recomputes gains takes 20700 seconds. To clarify the timeout symbol, these runs had a time limit of 8 hours, instead of the usual 2 hours. `DeltaGainsInToPQs` exhibits similar issues but is slightly faster than recomputing gains. The overhead compared to the gain table

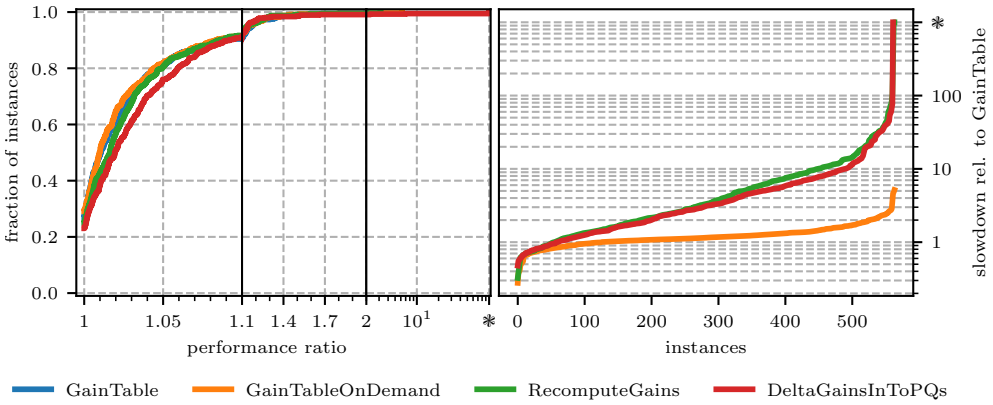

Figure 4.9: Partition quality and running time when using different techniques to calculate gains: gain tables, initializing the gain table only on demand, recalculating from scratch for each move, and updating gains directly in the priority queues. The running time plot (right) considers the time spent in FM.

stems from one heap operation for each gain update on a net between the moved vertex and neighbor, instead of one heap operation overall per neighbor.

Additionally, `DeltaGainsInToPQs` exhibits slightly worse partition quality than `GainTable`. `GainTableOnDemand` exhibits the same partition quality as `GainTable` but is overall slower. The hope was that by running LP before FM, only a fraction of the vertices would be considered in FM, which would make this approach attractive. However, this did not pan out as expected.

Hidden vs Global Moves. Now we assess the impact of hiding moves from other threads during FM until a net improvement is found. Recall that this comes at the cost of local partition data structures that must be stored in thread-local hash tables. In terms of quality, hiding moves (`ApplyToLocalPartition`) is clearly superior, contributing the best partitions on roughly 70% of the instances, and reaching the 90% fraction at 1.02 performance ratio, whereas `ApplyToGlobalPartition` reaches this marker only at around 1.06.

Interestingly `ApplyToLocalPartition` is even faster than `ApplyToGlobalPartition`, often by a factor 2-3. This was not the case in the conference publication, where the latter was substantially faster. We have since invested engineering effort for the local-first variant: clearing hash tables early, avoiding updates, and engineering the hash tables, such that we made it the sole default.

Running the experiment again with 16 threads reveals that the running time difference becomes less pronounced. The explanation is that FM performs a lot of moves, many of which will be reverted. This causes contention on the locks for updating the connectivity

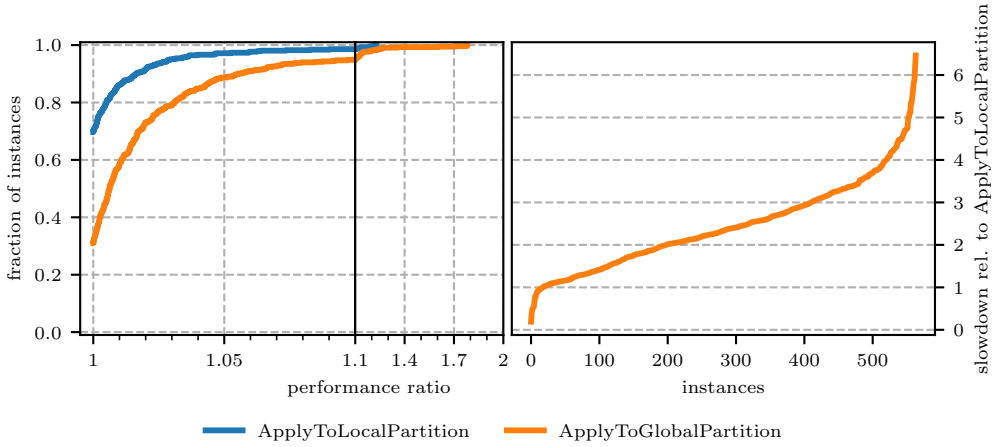

Figure 4.10: Impact on quality and running time of applying moves directly to the global partition or to a thread-local partition first during FM. Interestingly the latter is faster. The running time plot considers time spent in FM.

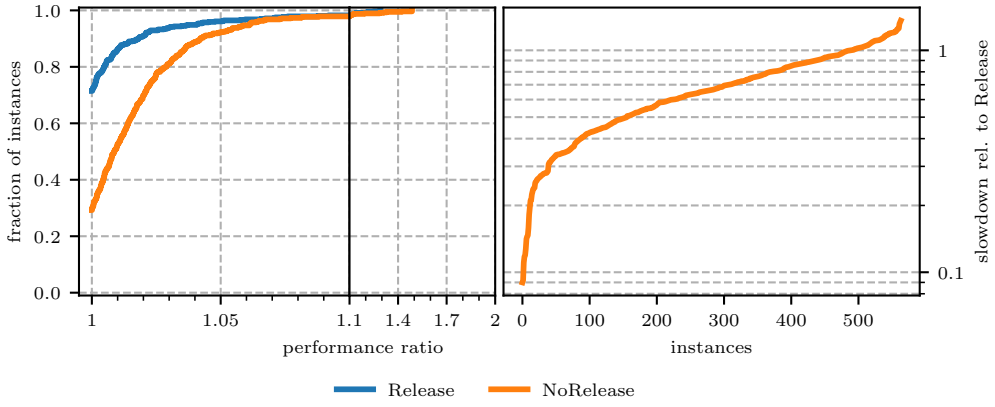

Figure 4.11: Impact on quality and running time of releasing acquired vertices at the end of a localized FM search. The running time plot considers time spent in FM.

sets $\Lambda(e)$ and packed pin-count data structures. Hiding these moves reduces lock contention. We previously argued that there should be little contention on these locks, and for LP this still holds because LP performs only positive gain moves.

Release Vertices. Now, we consider the option of releasing vertices at the end of a localized FM search (such that other searches during the current round can acquire them), versus

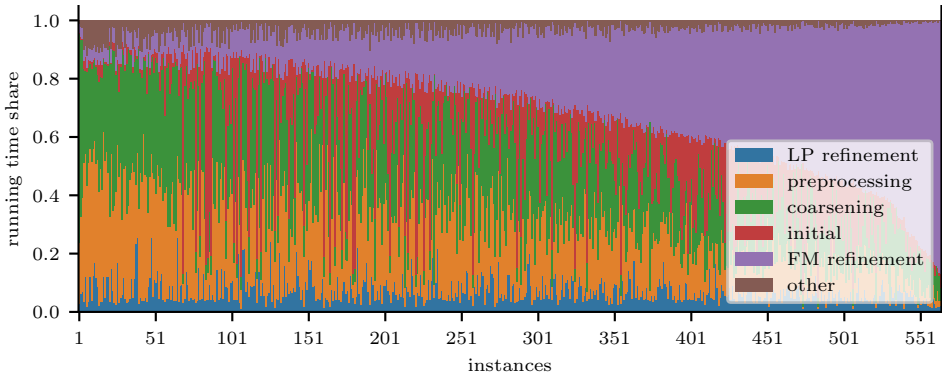

Figure 4.12: Fractions of time spent in each of the components of Mt-KaHyPar-D.

keeping them locked. The results are in Figure 4.11. The quality impact is similar to hiding moves, however this time the higher quality variant `Release` is slower than the lower quality variant, with slowdown ranging between a factor 1-10. On almost 200 instances `Release` is slower by a factor 2 or more.

Running Time Shares. Finally, we show how much running time each phase takes up in the overall multilevel algorithm. In Figure 4.12 this is plotted per instance. The instances are sorted by the share FM refinement has. To make the plot look cleaner, we show only the first seed, as we do not want to aggregate across different runs here. Overall, FM refinement makes up the biggest part of the work on most of the instances, as expected; though the share varies a lot ranging from below 5% up to 80%. Only on about 80 instances it takes more than 50% of the total running time, thus it is rarely the sole bottleneck.

Coarsening and preprocessing usually take between 15-30% each, with preprocessing taking up a bit more. Initial partitioning is often negligible, but there are a few instances where it takes very long. This is often due to coarsening terminating before the contraction limit is reached or uneven work loads in the recursive partitioning calls (sequential 2-way FM), which is why work-stealing is so important. LP usually stays below 5-10%. While LP is a similar algorithm as the coarsening and community detection clustering, it is substantially faster because the number of clusters is smaller, and thus rating maps fit in cache.

The results are close to what we expected beforehand. We thought the shares of FM would be even higher and those of coarsening and preprocessing would be lower. The category ”other” displays the difference between the total time and the sums of the measured phases (so that the bars reach to 1). These are parts of the code that were not timed, e.g., calculating metrics or aggregating statistics.

Table 4.1: Design aspects of parallel FM.

design avenue	Mt-KaHyPar	Mt-Metis
vertex assignment	localized expansion (LocExp)	static to threads (Stat)
gains	negative + local rollback (NegGain)	only positive (PosGain)
apply moves	local partition (Loc)	global partition (Glob)
communicate moves	gain table	message queues (MQ)
unused vertices	release (Rel)	keep locked (NoRel)

4.5 Parallel FM Design Choices

In this section we explore further design choices for implementing a parallel FM type algorithm. Often, researchers compare entire multilevel systems, such that implementation differences and different algorithms in the other two phases make a comparison of just the refinement algorithms difficult. Therefore we conduct our comparison within the same framework, to assess the impact of just the differences in the specific refinement components. The experimental results presented are based on the bachelor thesis of Noah Wahl [Wah21], which was supervised by me. We omit running times, because the implementation is not as optimized as possible, in order to accommodate integrating all components in one implementation.

Besides Mt-KaHiP the only other shared-memory parallel FM type algorithm is from the Mt-Metis graph partitioning framework by Lasalle and Karypis [LK13]. We identified six aspects in which their approach differs from ours, which we summarize in Table 4.1. First, boundary vertices are statically assigned to threads, and each thread performs sequential FM on the boundary vertices it owns; without localized expansion. Only positive gain moves are allowed, such that no sequential move order is necessary. Moves are applied directly to the global partition. Pair-wise message queues are used to communicate moves to threads owning neighbors of moved vertices, such that they can update their internal gain structures. No shared gain table is used. Finally, since there is no expansion, unused vertices cannot be moved by other threads, and are therefore kept locked.

In our evaluation, all variants use the shared gain table to simplify the implementation. In the table we listed abbreviations to identify the different components. For example the configuration of FM we use in Mt-KaHyPar-D would be called `LocExp-NegGain-Loc-NoMQ-Rel`, because it uses localized expansion, allows negative gains, applies moves to a local partition first, does not use message queues, and releases unused vertices. The experiments were run on benchmark set B and machine E with 16 threads.

We turn off the ”good” components step by step and examine the impact on partition quality. The order is the same as the legend in the plot. Starting with the FM configuration as used in Mt-KaHyPar-D (blue), we first switch from applying moves to a local partition to a global partition (Glob, orange), then from releasing unused vertices to keeping them locked (NoRel, green); two components whose impact we have already observed. This already

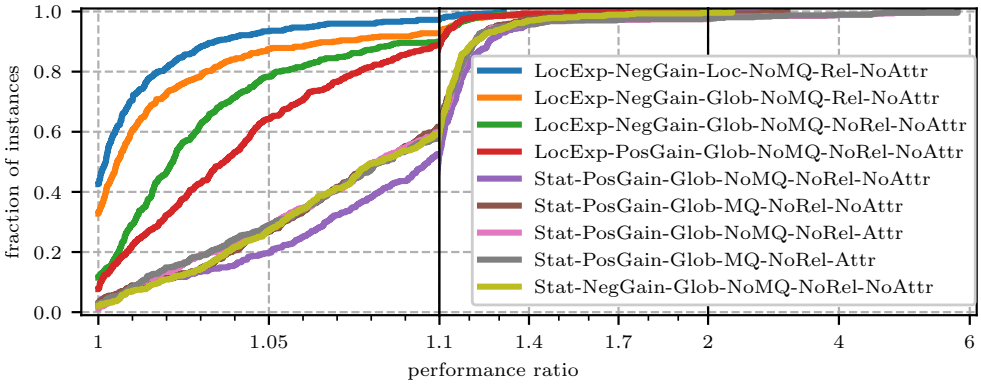

Figure 4.13: Partition quality when turning off FM components one by one.

shows a big decline in partition quality. Next we only allow positive gain moves (PosGain, red), which shows a further decline but not as strong. The biggest dip in partition quality is caused by switching from localized expansion to static assignment of boundary vertices to threads (Stat, purple). We also tried a version that is allowed to expand to not yet assigned vertices (non-boundary vertices at the beginning) but this did not exhibit any difference (not shown in the plot). Some of the quality can be recovered by using message queues (MQ, brown), or attributed gains (Attr, pink). Using message queues and attributed gains in combination (gray) does not give any further benefits. Allowing negative gain moves again (yellow, NegGain) also manages to recover some quality. While message queues can be used in conjunction with the strongest version presented, this did not make an impact (not shown here). Their ineffectiveness is caused by the use of the shared gain table.

4.6 Scalability

After evaluating the components in our framework in terms of quality and running time, it is now time to assess their scalability. In Figure 4.14 we plot self-relative speedups for each component as well as the overall framework (top left), with 4,16,64 threads. We plot one data point (scatter) for each instance, as well as an aggregate rolling geometric mean with window size 50. The instances are sorted by sequential running time, to show that we achieve better speedups on longer running instances. For readability purposes, we did not plot our measurements with 2,8,32 threads, but the overall trends are the same, and we filtered 3 outliers for initial partitioning above 128. Additionally, Table 4.2 shows geometric mean speedups for all instances and instances where the particular phase took at least 100 seconds.

With 4 threads we achieve near-perfect speedups in coarsening, initial partitioning and FM refinement. LP struggles on short-running instances and it is overall the fastest phase

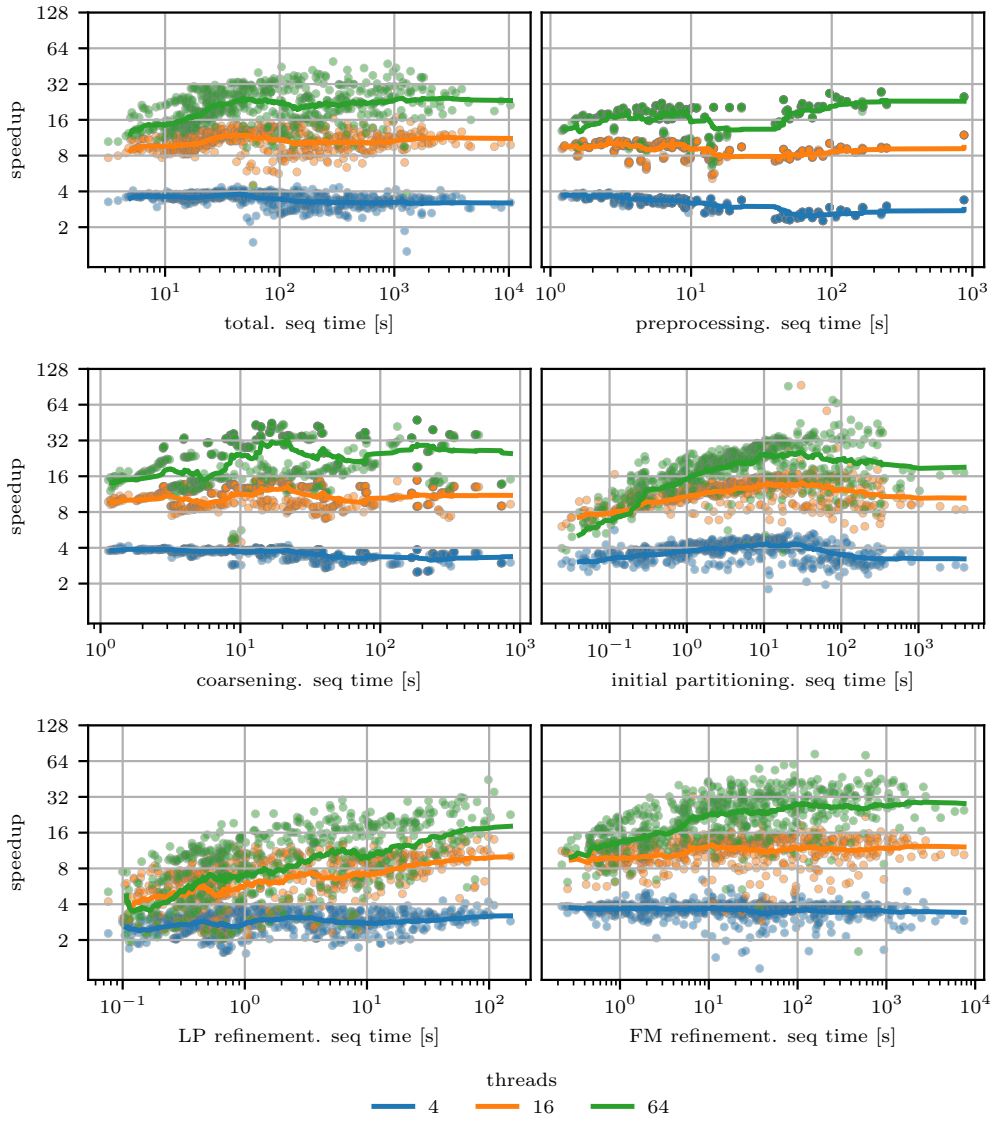

Figure 4.14: Speedups for Mt-KaHyPar-D in total as well as its components separately. The x-axis shows the sequential time in seconds, the y-axis the speedup. The lines are rolling geometric means (window size 50) of the per-instance speedups (scatter).

Table 4.2: Geometric mean speedups for each phase, 16 and 64 threads, all instances or instances that took at least 100 seconds sequentially. The last row shows the percentage of instances where this applies.

instances	threads	total	coarsen	initial	community	LP	FM
all	16	10.62	10.91	11.57	9.15	6.37	11.18
all	64	20.54	22.88	18.16	17.4	8.47	21.29
$\geq 100s$	16	10.53	11.05	10.5	9.34	11.36	11.83
$\geq 100s$	64	22.17	26.8	18.97	23.08	25.82	27.19
% $\geq 100s$		40.6	14	9.8	9.6	0.5	23

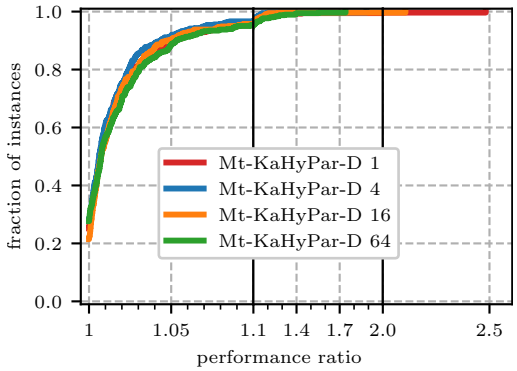

Figure 4.15: Partition quality with increasing number of threads. We can see that there is no quality penalty incurred by using more cores.

by far, so this is expected. On the other hand, community detection preprocessing achieves near-perfect speedup on the fast instances, but struggles on the slower ones (more than 10 seconds), where its speedup is around 3.

With 16 threads the overall geometric mean speedup is 10.62, and with 64 threads it is 20.54. This improves to 22.17 if we consider only instances that take longer than 100 seconds. FM (21.29, 27.19) and coarsening (22.88, 26.8) exhibit the best speedups.

Overall, the speedups are fairly robust, which was not yet the case in the conference publication. With subsequent performance engineering we were able to eliminate startup overheads in FM refinement and shared resource usage in flat initial partitioning. These contributed to poor speedups on instances that were solved quickly. Now we see good speedups even on the small instances, though it is still a general trend that we achieve better speedups on longer running instances.

Finally, Figure 4.15 shows that increasing the number of threads does not affect the partition

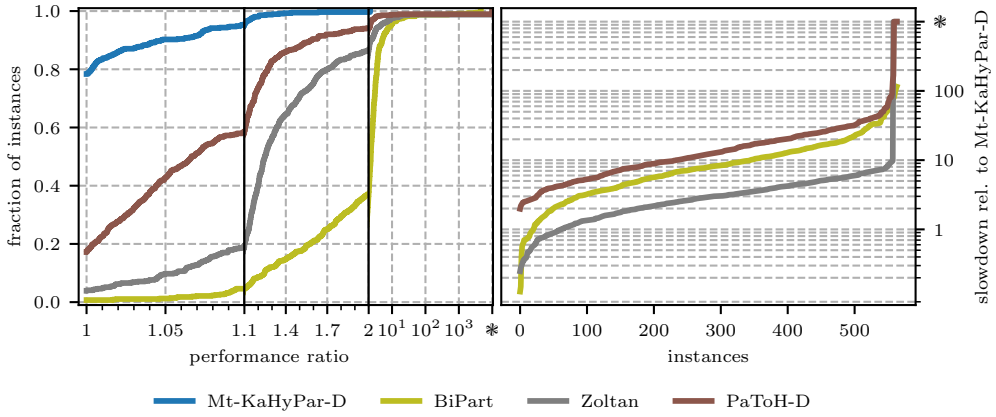

Figure 4.16: Partition quality and running time on benchmark set B. All algorithms are run on 64 cores, except PaToH.

quality either negatively or positively. This is usually an issue for parallel partitioning algorithms.

4.7 Horse-Race Comparisons

Now we turn our attention to comparisons with existing state-of-the-art hypergraph partitioning algorithms. We start with fast and parallel algorithms on set B, and then turn to sequential algorithms on set A.

4.7.1 Comparison with Parallel Algorithms

Figure 4.16 summarizes the results for Mt-KaHyPar-D compared with PaToH-D, Zoltan, and BiPart on benchmark set B. Mt-KaHyPar-D clearly outperforms the competitors by a huge margin, both in terms of partition quality and running time. It contributes 78.5% of the best partitions (443) to the pool, is within a factor of 1.1 of the best solution on over 95% of the instances, and never worse by more than 2.2. PaToH-D contributes around 16.4% of the best partitions (98) and reaches the 1.1 ratio at around 60% of the instances. BiPart is far off, contributing only 5 of the best partitions, reaching 1.1 at below 10% and reaching factor 2 below 40%. Zoltan is situated between BiPart and PaToH. It converges towards 1 much faster than BiPart. This indicates that Zoltan is more robust, as its worst solutions fare better than the worst of BiPart. The worst partition of BiPart is a factor 4551 worse than the best on that instance.

The geometric mean running times are 3.9s for Mt-KaHyPar-D, 47.63s for PaToH-D, 29.15s for BiPart, and 10.64s for Zoltan. The ranking induced by the aggregates fits with the overall

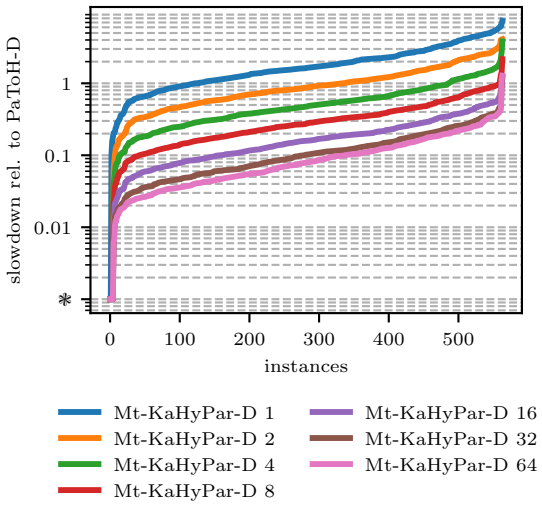

Figure 4.17: Running time comparison with increasing number of threads versus PaToH-D.

picture in the running time plot of Figure 4.16 (right), where Mt-KaHyPar-D is usually the fastest. It is faster than Zoltan by a factor between 1-10, and between 2-200 for BiPart and PaToH-D. For PaToH-D this is expected since it is sequential, for BiPart not so much since it uses only simple components that are easy to parallelize. In Figure 5.5 of Chapter 5 (our deterministic algorithm) we show speedups for BiPart, which are mostly below 2, and the largest speedup is around 7.

In Figure 4.17 we additionally plot the slowdown of our algorithm relative to PaToH, with different numbers of threads. With 1 thread Mt-KaHyPar is slower, with 2 threads the running time is very similar, and starting at 4 threads it is noticeably faster. PaToH is seen as the fastest sequential multilevel partitioner [Sch20] that is extremely well engineered and configured for speed. For example, PaToH uses only 2 rounds of 2-way boundary FM per level (recursive bipartitioning). Considering we use 10 rounds of k-way FM per level, the similar running time at 2 threads is impressive for our implementation.

4.7.2 Comparison with Sequential Algorithms

In this section we compare Mt-KaHyPar with the state-of-the-art sequential hypergraph partitioning codes KaHyPar, PaToH, and hMetis on benchmark set A. Mt-KaHyPar is run on 10 cores. For KaHyPar we consider two configurations: KaHyPar-CA (n-level with community-aware coarsening [HS17]), and additionally using flow-based refinement with FlowCutter [GHSW20]. For hMetis, we include only the recursive bipartitioning configura-

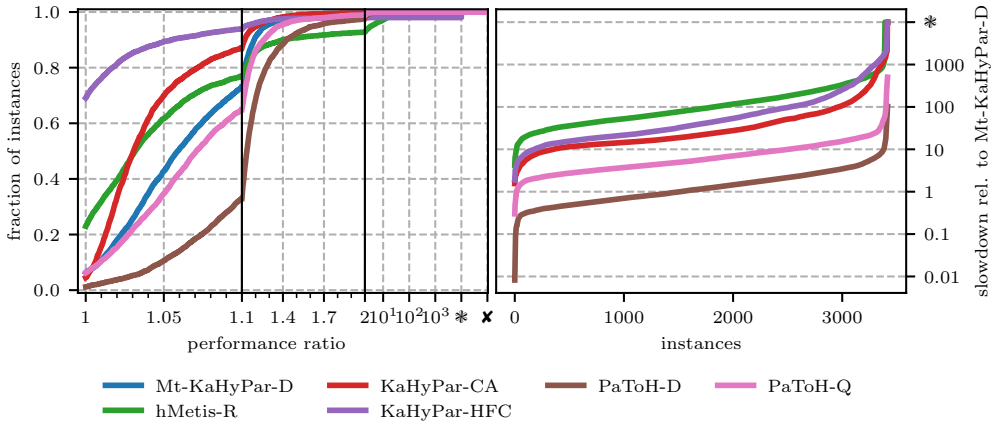

Figure 4.18: Partition quality and running time on benchmark set A, using 10 cores for Mt-KaHyPar.

tion hMetis-R, as the direct k-way version produces mostly imbalanced partitions, which are further of inferior quality to those of hMetis-R. Figure 4.18 summarizes the results.

Mt-KaHyPar is between one and three orders of magnitudes faster than hMetis-R, KaHyPar-CA and KaHyPar-HFC, with the majority of their slowdowns in the range 10-100. hMetis-R is the slowest algorithm in this evaluation, followed by KaHyPar-HFC. On around 2500 instances, Mt-KaHyPar is between a factor 2 to 10 faster than PaToH-Q, and on around 900 instances it is between one to two orders of magnitude faster. PaToH-D has a similar running time as Mt-KaHyPar: it is faster on around half of the instances and on the other half it is slower. In the previous section, we showed that Mt-KaHyPar with just 2 threads has a similar running time as PaToH-D on set B. The difference here is that the instances of set A are substantially smaller, which makes it more difficult to obtain good parallel speedups, and emphasizes the parallelization overheads more. The geometric mean running times are 0.96s for Mt-KaHyPar-D, 1.17s for PaToH-D, 5.86s for PaToH-Q, 28.14s for KaHyPar-CA, 48.95s for KaHyPar-HFC and 93.2s for hMetis-R, which match the ranking suggested by the plot.

KaHyPar-HFC contributes around 70% of the best partitions, hMetis-R around 20%, while PaToH-Q, Mt-KaHyPar-D, and KaHyPar-CA contribute around 8% each. KaHyPar-CA and Mt-KaHyPar-D converge faster towards 1 than hMetis-R, crossing its curve at 1.03 and 1.1, respectively. PaToH-Q has similar quality as Mt-KaHyPar-D though slightly worse overall, and PaToH-D comes in last. Because KaHyPar-HFC works well on similar instances as KaHyPar-CA and Mt-KaHyPar-D and has superior components, we also present two plots without it in Figure 4.19 for a more individualized comparison. In the left one, we compare Mt-KaHyPar-D with PaToH and hMetis-R, and in the right one directly with KaHyPar-CA. Here we see that the gap between Mt-KaHyPar-D and PaToH-Q is larger than suggested by the first plot. Furthermore, hMetis-R and KaHyPar-CA are noticeably better than Mt-KaHyPar-D.

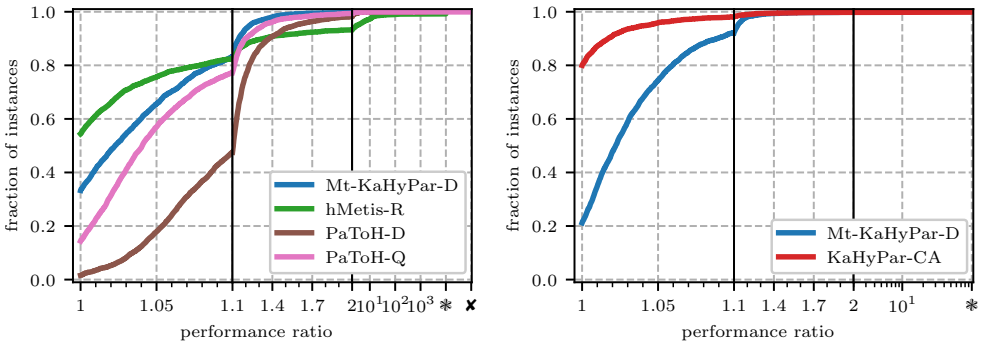

Figure 4.19: More individualized quality comparisons on set A.

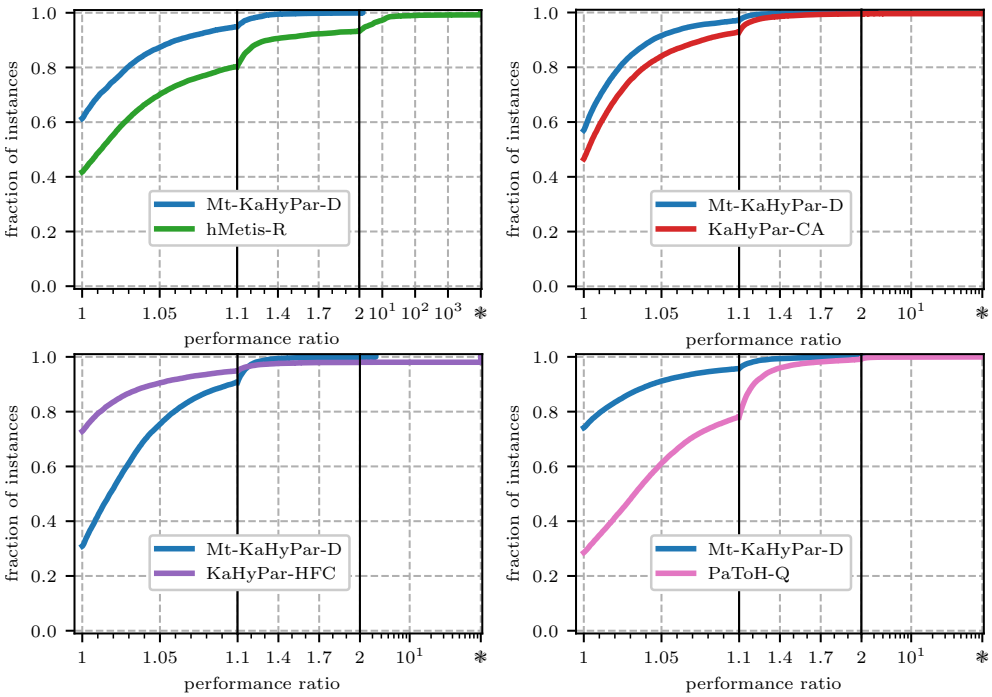

Figure 4.20: Effectiveness tests with virtual instances on benchmark set A, comparing Mt-KaHyPar-D (on 10 cores) with KaHyPar-HFC, KaHyPar-CA, hMetis-R and PaToH-Q.

However, they are also substantially slower, not just due to parallelism but due to implementation and design choices. Therefore, we perform effectiveness tests with virtual instances in Figure 4.20 to compare partition quality when the algorithms are given a similar amount of running time. Given the large running time discrepancy, it is very likely that Mt-KaHyPar will not exhaust the given time, even with all 10 available seeds. We see that in the effectiveness tests, Mt-KaHyPar-D performs noticeably better than hMetis-R, and slightly better than KaHyPar-CA, whereas it cannot beat flow-based refinement purely with repetitions. Mt-KaHyPar is highly non-deterministic and exhibits large fluctuations in partition quality, which is a strength in these effectiveness tests. Lastly, we run an effectiveness test versus PaToH-Q, where the gap simply becomes larger than before.

4.7.3 Comparison with Social Hash

Since we could not include SHP (Social Hash Partitioner) [Kab+17a] in the regular comparison, we ran Mt-KaHyPar on the available real-world instances from their paper. We compare results computed on our machine with those reported in their technical report [Kab+17b]. Note that SHP optimizes the *fanout* objective, which is equivalent to connectivity on unweighted instances, since $(\lambda - 1)(\Pi) = (\text{fanout}(\Pi) \cdot m) - m$. We report connectivity values (converted from fanout for SHP) for $k \in \{2, 8, 32, 128, 512\}$ for SHP-K (direct k-way), SHP-R (recursive bipartitioning), and Mt-KaHyPar-D in Table 4.3. The instances are ordered from smallest to largest. On the two largest instances we additionally consider the configuration Mt-KaHyPar-S (speed, FM disabled).

On most of the instances, Mt-KaHyPar computes substantially better partitions. The difference is most pronounced for the web instances. SHP is only competitive on the two smallest instances email–Enron and soc–Epinions.

In Table 4.4 we report absolute running times for $k \in \{32, 512, 8192\}$. Unfortunately the authors of SHP only included running times for the two largest instances Pokec and LiveJournal, and used different values of k than for the quality measurements. Since different machines were used (4 x 16 threads), which are furthermore older than ours, running times cannot be compared directly. For large k, SHP-K is slower than SHP-R because its label propagation implementation calculates gains to each block separately instead of to all blocks in one simultaneous pass over incident nets. Mt-KaHyPar-S is the fastest out of the considered variants. The running time of Mt-KaHyPar-D is similar to SHP-R for $k = 32$, but slower for $k = 512, 8192$. Compared with SHP-K the running is similar for all three values of k. This is astonishing as Mt-KaHyPar uses k-way FM and multilevel, whereas SHP is flat and uses only label propagation. Recall from Figure 4.12 and Figure 4.14 that label propagation is substantially faster than FM.

4.7.4 Larger Number of Blocks

In this section, we perform experiments with Mt-KaHyPar and Zoltan on benchmark set B but for larger values of $k \in \{512, 1024, 2048\}$ than our previous experiments. PaToH and BiPart

Table 4.3: Quality Comparison with SHP. Best solutions in bold.

		$\lambda - 1$				
H	Partitioner	$k = 2$	$k = 8$	$k = 32$	$k = 128$	$k = 512$
Enron	SHP-R	**3313**	19875	39241	65997	**102434**
	SHP-K	3822	17837	**33635**	60900	112371
	Mt-KaHyPar-D	5398	**16876**	33649	**55330**	110575
Epinions	SHP-R	1246	21804	53888	95627	**151073**
	SHP-K	1869	**20247**	**47658**	**90332**	164467
	Mt-KaHyPar-D	**1082**	22638	56318	102401	168315
Stanford	SHP-R	22779	60743	101239	149327	189823
	SHP-K	20248	43026	75929	116425	197416
	Mt-KaHyPar-D	**42**	**9641**	**34808**	**67186**	**100431**
BerkStan	SHP-R	67048	170668	274287	420574	542479
	SHP-K	60953	146286	195049	304764	597336
	Mt-KaHyPar-D	**109**	**18866**	**90299**	**180352**	**266095**
Pokec	SHP-R	625731	2273064	4175797	6142380	8773004
	SHP-K	549111	2094283	3920396	5772049	8274973
	Mt-KaHyPar-D	**498795**	**1881114**	**3530491**	**5478685**	**7875689**
	Mt-KaHyPar-S	538508	1929862	3693454	5777868	8548565
LiveJournal	SHP-R	1051618	3969011	7768406	11805263	16181352
	SHP-K	1255157	3595856	6275786	9566334	14315578
	Mt-KaHyPar-D	**777664**	**2686922**	**5165858**	**8073151**	**11417656**
	Mt-KaHyPar-S	858789	2978709	5707737	9323059	12569971

are too slow to be included in these experiments. Again, we add the speed configuration Mt-KaHyPar-S without FM to showcase the impact of the multiplicative $\mathcal{O}(k)$ term in the running time of FM. The results are shown in Figure 4.21. The default version with FM yields the best quality; the difference to Zoltan looks even bigger than in Figure 4.16. However, using FM comes at a huge cost in terms of running time for large k. The geometric mean running times are 42.32s for Mt-KaHyPar-D, 15.15s for Zoltan, and 9.59s for Mt-KaHyPar-S. On 65% of the instances Mt-KaHyPar-S is faster than Mt-KaHyPar-D by more than a factor of 3. Even though it only uses label propagation, Mt-KaHyPar-S still finds better partitions than Zoltan and is faster. Zoltan has 31 instances with only imbalanced partitions and 3 crashes. Mt-KaHyPar-D times out on one instance and runs out of memory on one.

Table 4.4: Running time comparison with SHP. Fastest algorithms in bold.

		$t[s]$		
H	Partitioner	$k = 32$	$k = 512$	$k = 8192$
Enron	SHP-R	–	–	–
	SHP-K	–	–	–
	Mt-KaHyPar-D	0.46	1.03	5.23
Epinions	SHP-R	–	–	–
	SHP-K	–	–	–
	Mt-KaHyPar-D	0.77	1.72	7.44
Stanford	SHP-R	–	–	–
	SHP-K	–	–	–
	Mt-KaHyPar-D	0.56	1.79	5.57
BerkStan	SHP-R	–	–	–
	SHP-K	–	–	–
	Mt-KaHyPar-D	1.00	3.36	9.60
Pokec	SHP-R	108	138	270
	SHP-K	156	528	2076
	Mt-KaHyPar-D	70.42	173.76	760.71
	Mt-KaHyPar-S	**45.71**	**55.67**	**87.31**
LiveJournal	SHP-R	144	222	396
	SHP-K	276	1260	2868
	Mt-KaHyPar-D	125.39	332.15	4466.63
	Mt-KaHyPar-S	**80.83**	**91.36**	**154.11**

4.7.5 Comparison with Graph Partitioning Algorithms

Since hypergraphs are a generalization of graphs, Mt-KaHyPar can also be used for partitioning plain graphs. We included a comparison with dedicated graph partitioners in the conference paper. The solution quality was competitive with Mt-KaHiP, but the running time was higher by a factor of about 2. Unsurprisingly, our code was at a sizable disadvantage because the graph is stored in a sub-optimal format and the algorithm implementations are more complex than necessary for this particular case. To alleviate the first issue, we plug in a graph data structure in place of the hypergraph and emulate the $\Phi(e, i)$, $\Lambda(e)$ data structures, as well as the hypergraph iteration methods. Thanks to template meta-programming this switch is easily possible at compile time. Nikolai Maas carried out the implementation and integration of this approach during his employment as a student researcher with Tobias Heuer.

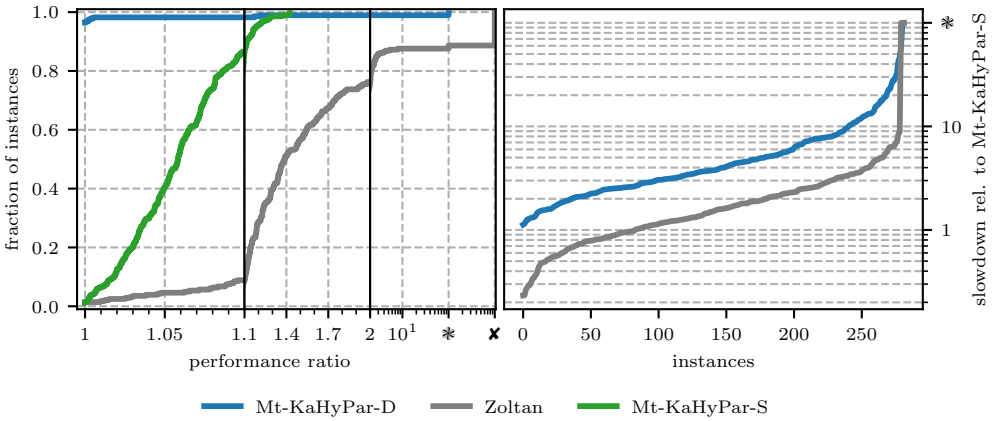

Figure 4.21: Partition quality and running time on benchmark set B for large $k \in \{512, 1024, 2048\}$.

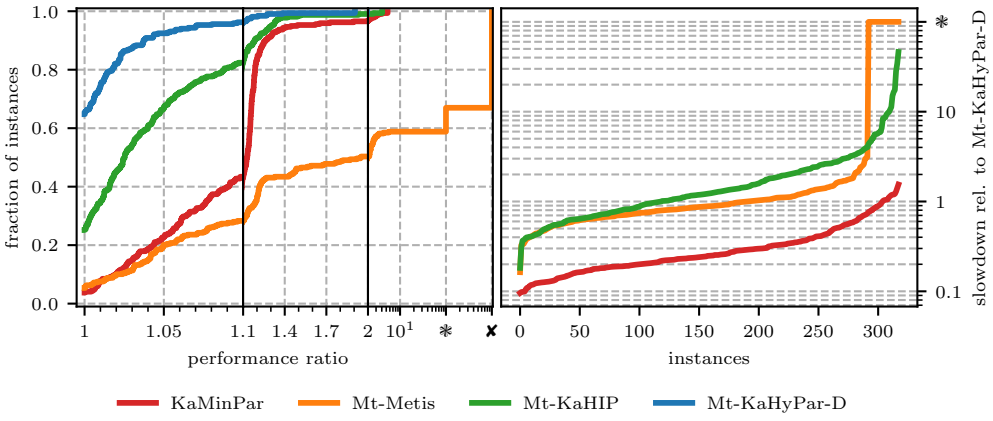

Figure 4.22: Partition quality and running time comparison with dedicated graph partitioners on benchmark set E.

In Figure 4.22 we compare this version with the shared-memory parallel graph partitioning frameworks Mt-KaHiP [ASS20], Mt-Metis [LK16] (with hill-scanning) and KaMinPar [Got+21]. We use benchmark set E (53 graphs, $k \in \{2, 4, 8, 16, 32, 64\}$ makes 318 instances) and machine B with 64 cores for each algorithm. In terms of quality, Mt-KaHyPar performs best, as it contributes around 65% of the best partitions, reaches 1.1 performance ratio around 95% of the instances and is never worse by a factor of more than 2. Mt-KaHiP contributes around 25% of the best partitions and reaches 1.1 performance ratio around 80%. The qual-

ity difference between Mt-KaHyPar and Mt-KaHiP can be attributed to the differences in localized FM we outlined in Section 4.3.7 and the community-aware coarsening. Mt-Metis crashes on 12% of the instances (no timeouts). Out of 1590 runs, only 815 produce balanced partitions, which results in about 35% of the instances where all computed partitions were imbalanced. At some point Mt-Metis used a rebalancing algorithm in the refinement, but it was removed in more recent versions. KaMinPar is far off in terms of quality since it uses only label propagation, but this means it is much faster. The geometric mean running times are 2.55s for KaMinPar, 9.2s for Mt-KaHyPar-D, 11.93s for Mt-Metis, and 12.77s for Mt-KaHiP. The ranking induced by the aggregates almost matches the behavior of the per-instance running time plot. KaMinPar is almost always faster than Mt-KaHyPar, by a factor of 5-10 on 100 instances, and by a factor of 3-10 on 200 instances. On 200/318 instances, Mt-Metis is faster than Mt-KaHyPar, however the crashes are counted as the 2 hour time limit in the aggregate, which explains why the aggregate is higher. Mt-KaHiP is slower than Mt-KaHyPar on around 200/318 instances.

4.8 Algorithm Configuration

Our algorithm possesses a huge number of parameters. We summarized the most important ones and their default configuration value in Table 4.5. In the following, we perform parameter studies for a subset of these where we see interesting results, structured by the multilevel phases. The experiments are performed on machine E with 16 cores on the instances of the parameter tuning benchmark set C.

For coarsening we consider the contraction limit parameter t, and the shrink factor β. For initial partitioning we test the adaptive portfolio that only repeats promising algorithms, as well as the maximum number of repetitions of each flat algorithm in the portfolio. For refinement we test the number of rounds for LP and FM, as well as the number of seeds in localized FM. Experiments on the Louvain rounds are not shown. We tested 3-9 rounds and saw now difference, which matches previous results by Öhl [Öhl18] who showed that just two rounds suffice for best quality. This is different from usual community detection tasks in social network analysis, where values upwards of 20 rounds are common. For label propagation, we do not show results for the net activation threshold (tested range $50 - 10^4$) and rebalancing with zero gains, as we did not observe any differences between the configurations.

In Figure 4.23 (left) we see the impact of the coarsening limit t on partition quality. Our choice of 160 works best, and 100 or 240 work equally well, whereas larger values beyond 300 are no longer competitive.

On the right of the same figure we show results for the shrink factor β during coarsening. This affects the number of levels created in the hierarchy. Surprisingly we see very little difference between the different tested values. Out of caution we use a fairly conservative value of 2.5 as the default, instead of fast shrinking with larger values. The value 2.5 even narrowly emerges as the top curve.

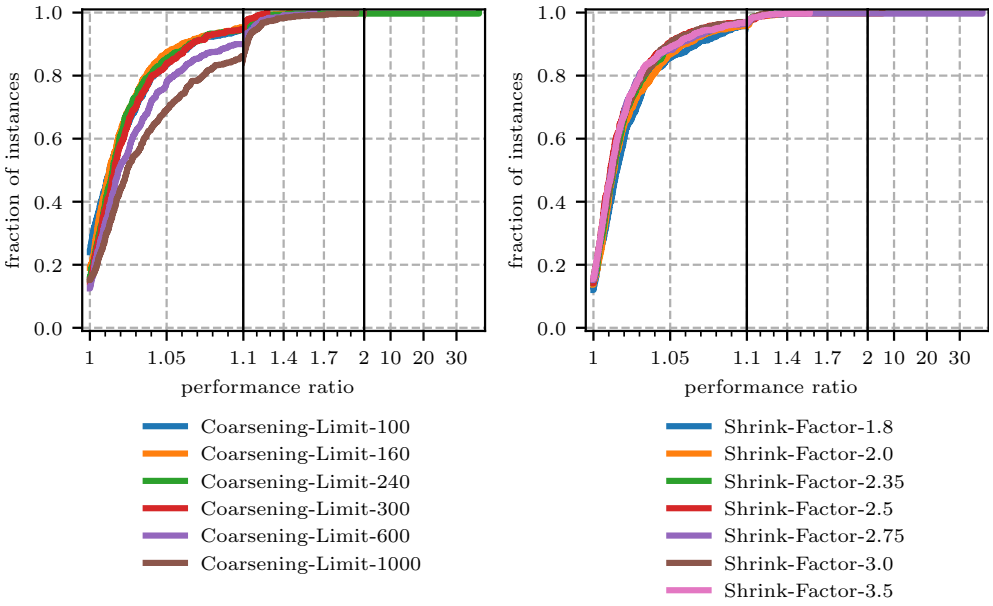

Figure 4.23: Effect of coarsening limit (left) and shrink factor (right) on partition quality.

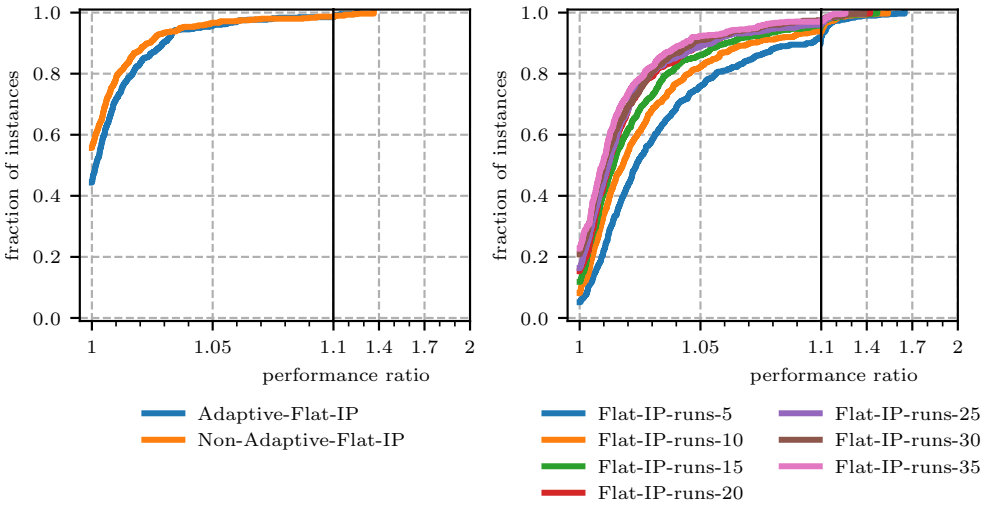

Figure 4.24: Effect of the adaptive portfolio (left) and number of repetitions per flat algorithm (right) on final partition quality.

Parameter	Explanation	Selected Value
-	Louvain: num rounds before contraction	5
-	Louvain: min fraction of nodes moved	1%
η	max net size for heavy-edge rating	1000
t	coarsen until $t \cdot k$ vertices remain	160
$W_{\max}$	max cluster weight	$\lceil \frac{c(V)}{t \cdot k} \rceil$
α	terminate coarsening if $\lvert V_i \rvert / \lvert V_{i+1} \rvert \leq \alpha$	1.01
β	create level $i+1$ once $\lvert V_i \rvert / \lvert V_{i+1} \rvert > \beta$	2.5
-	flat bipartitioning repetitions	20
-	use adaptive	yes
-	min repetitions (only if adaptive)	5
-	LP rounds	5
-	LP max net size for activating neighbors	100
-	LP use zero gain moves for rebalancing	yes
-	max FM rounds	10
-	FM releases vertices	yes
-	FM moves directly on global partition	no
-	FM seeds	25
-	FM stop rule	adaptive
-	FM rollback: allowed balance violation	$1.25 \cdot \varepsilon$
-	FM PQ layout	from-PQs
-	FM gain table	yes
-	IP mode (rb, deep, direct)	rb

Table 4.5: List of parameters used in Mt-KaHyPar

In Figure 4.24 (left) we see that using the adaptive portfolio comes at a small quality penalty. We deem this justified by the savings in running time: 0.16s vs 0.24s geometric mean time, and thus enable the adaptive portfolio in the default configuration. The number of flat algorithm repetitions (right) is more interesting. We see that quality increases if more repetitions are added, as expected. But there are diminishing returns around 20 repetitions, which we therefore choose as our default value. Both plots are based on the quality of the final partition, not the initial k-way partition.

Figure 4.25 (left) shows that the number of LP rounds has almost no impact on partition quality, which is very surprising. FM was not disabled for these measurements, since we want to use the combination of LP and FM, which contributes to this effect. Looking at some outputs, the vast majority of the improvements happen in the first LP round, a few improvements still happen in the second, and by the third almost no moves are made any

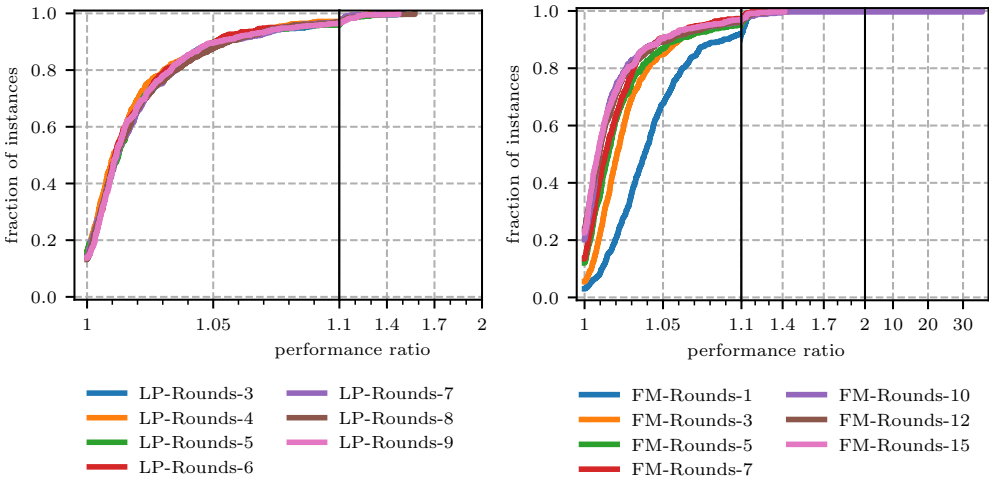

Figure 4.25: Effect of LP rounds (top left), FM rounds (top right). The other refinement algorithm was not disabled for these measurements.

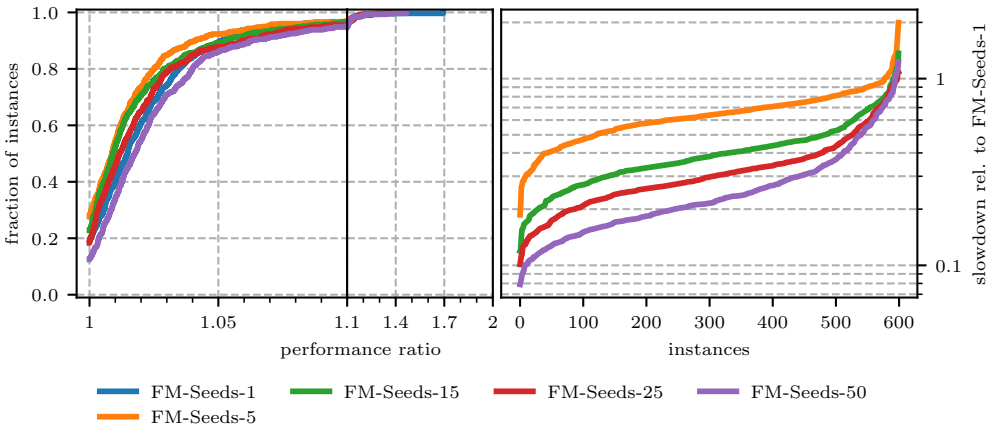

Figure 4.26: Effect of the number of FM seeds on partition quality and running time.

more. This further explains why LP is so fast, as very little work is invested thanks to the active set strategy.

This phenomenon does not carry over to the number of FM rounds (right), where we see drastic improvements when going from 1 to 3 rounds. From 3 to 5 the step is much smaller, and by 10 rounds diminishing returns set in. Therefore, we use 10 rounds as the default value.

For the number of seeds (Figure 4.26) the sweet-spot is at 5, not at 1, because with fewer seeds it is more likely that some of the moves that are reverted would be helpful in the local search of a different seed. On the other hand, grouping too many seeds together diminishes the localization, as unrelated searches are performed together. This can interfere with the hill-climbing capabilities. At more than 5 seeds, the quality slowly degrades, but the running time improves drastically with 80ms geometric mean running time for 15 seeds, 60ms for 25, 50ms for 50 seeds, compared to the 130ms for 5 seeds and 210ms for 1 seed. In this chapter we aim for a fast configuration with moderately good quality, which is why we choose 25 seeds as the default value. In Chapter 6 on our parallel n-level variant we aim for the maximum quality regardless of running time, so we use 5 seeds in that configuration.

4.9 Conclusion

In this chapter, we presented a shared-memory parallel hypergraph partitioning algorithm that guarantees balanced partitions, exhibits very good speedups and partition quality, even compared with sequential algorithms. It is the fastest multilevel hypergraph partitioner to date and offers the best quality out of all parallel algorithms available. This is achieved by carefully engineering and parallelizing each phase of the multilevel framework. On the algorithmic side, we presented parallel versions of agglomerative clustering for coarsening, label propagation refinement and localized k-way FM refinement.

In our experiments we demonstrated that our implementation and design choices are superior to choices made by previous, related implementations [LK13, ASS18a, Sch20]. We falsified the claim that combining LP and FM is superior in terms of running time and quality over using just FM [ASS18a], at least on our benchmark instances. Unfortunately, the gain accuracy techniques were not as impactful as we had hoped, which is caused by the lack of interference between the local searches, thanks to randomization. Using attributed gains showed no significant difference but also does not come at extra running time costs. Using a parallel gain table showed small to moderate quality improvements over other approaches. Yet, the main benefit of the gain table is its superior speed. The largest quality impact is from the global rollback step, which uses our third gain accuracy technique: parallel gain recalculation.

5 Deterministic Parallel Hypergraph Partitioning

Motivation and Context. A program is deterministic if it is guaranteed to produce the same output each time it is run on the same input. For sequential programs this is easy to achieve most of the time. For parallel programs not so much, as randomized scheduling and arbitrary execution interleaving lead to different results in each run. Researchers have advocated the benefits of deterministic parallel algorithms for several decades [Ste90, BFGS]. It is easier to debug the program, to reason about performance and it enables reproducing experimental results.

While some researchers strive for deterministic programming models [LCB11], this comes at high costs, e.g., static scheduling, private on-write copies of shared memory, deterministic ordering of updates at synchronization points, and most importantly the inability to use custom locking and synchronization measures [LCB11]. These approaches focus on deterministic program behavior (a stronger requirement), not just deterministic results.

We want to enjoy the performance benefits of randomized scheduling for skewed inputs (work stealing) and custom synchronization. Therefore we pursue custom measures to force existing algorithms to produce deterministic results. Among these techniques are *hiding* asynchronous move decisions, *sorting* collectively produced results if their order influences the execution in the next program stage, and *tie-breaking* with deterministically generated tags when selecting results.

With the exception of BiPart [MABP21], all published parallel partitioning algorithms so far are non-deterministic. In this chapter, we design, implement and evaluate a scalable multilevel partitioning algorithm with direct k-way refinement that is fully deterministic. To experimentally verify our last claim, we compared partitions from repeated runs. We propose deterministic algorithms for the preprocessing, coarsening and refinement phases of Mt-KaHyPar, and furthermore show how make the rest of the framework deterministic.

Algorithm 5.1: Local Moving Round

```
1 for v ∈ V in random order do in parallel
2     compute and perform best move for v
3     update data structures
```

Algorithm 5.2: Synchronous Local Moving Round

```
1 randomly split vertices into sub-rounds
2 for r = 0 to number of sub-rounds do
3     for v ∈ V in sub-round r do in parallel
4         compute and save best move for v
5     approve saved moves and update data structures
```

Attributions. This chapter is based on a conference publication together with Michael Hamann. I wrote the source code and paper, and conducted the experiments. Michael Hamann contributed to writing the paper. Furthermore, we thank Tobias Heuer for feedback on an early draft.

Non-Determinism in Local Moving. Typical community detection, clustering coarsening and label propagation refinement algorithms are so-called *local moving algorithms*, which follow the structure outlined in Algorithm 5.1: given an initial assignment of vertices to groups, visit vertices in random order in parallel, and improve the solution by greedily moving vertices when they are visited. Since vertices are moved right away, the local optimization decisions depend on non-deterministic scheduling decisions, as they are influenced by changes in their neighborhood.

Our approach to incorporate determinism is illustrated in Algorithm 5.2. It is based on the *synchronous local moving* approach [HSWZ18] to parallelize the Louvain community detection algorithm [BGLL08] in distributed memory scenarios. Instead of performing all moves asynchronously, vertices are split into sub-rounds. The best move for each vertex in the current sub-round is computed with respect to the unmodified groups. The move decisions are deterministic because they are not influenced by other concurrent moves. In a second step, some of the calculated moves are approved and performed, and some are denied, for example due to the balance constraint.

Our algorithmic contributions lie in the details of the approval steps. For example, the refinement approval uses a merge-style parallelization to incorporate vertex weights.

Chapter Overview. This chapter again follows the order of the multilevel framework. In Sections 5.1, 5.2 and 5.4, we discuss how to incorporate determinism into the local moving algorithms for community detection, coarsening and k-way refinement, respectively.

The ingredient that is common to all three local moving algorithms, randomized splitting into sub-rounds is discussed in Section 5.5. Section 5.3 deals with initial partitioning. In Section 5.6, we highlight differences between our approach and BiPart. In Section 5.7, we outline the difficulties of incorporating determinism into parallel localized FM, as this is the only component of Mt-KaHyPar-D for which we do not provide a deterministic version. In Section 5.8, we conduct scalability experiments, a comparison with other partitioning algorithms, and analyze the cost of determinism in terms of partition quality. Finally, we summarize our results in Section 5.9 and provide some ideas for future work.

5.1 Preprocessing

In the preprocessing phase, we compute a community assignment using the parallel Louvain method on the star expansion, as detailed in Section 3.4.5. We directly apply the algorithmic template of Algorithm 5.2 to Louvain. We randomize the visit order by dividing nodes into random sub-rounds. For each sub-round we calculate the best move for each node in parallel, but only apply community volume and community assignment updates after synchronizing. There are no constraints on the communities, so all moves are approved.

Update Order. The difficulty lies in applying community volume updates as we have floating point edge weights. In theory, we could use atomic instructions, but on real machines adding floating point numbers is not associative, which leads to small discrepancies depending on the execution order. Rescaling edge weights to integers is unfortunately not an option, due to the $1/|e|$ factor, so we are stuck with floating points. We would have to multiply each edge weight with the least common multiple of unique net sizes, which would exceed what 64 bit floating points can represent.

Instead, we have to establish a deterministic order in which the volume updates of each community are aggregated. We lexicographically sort the updates by community (primary key) and node ID (secondary key). Applying the updates is done in parallel for different communities. We perform a parallel loop over all updates, and the first entry of a community is responsible for performing the sequential scan. A scan ends once the first update associated with a different community is encountered.

Analysis. To analyze the work and depth, let V'_G denote the nodes in a sub-round. The work is $\mathcal{O}(\sum_{u \in V'_G} \deg(u) + |V'_G| \log(|V'_G|))$. The depth is linear in the maximum number of moves in or out of a community (sequential volume updates) and the maximum degree $\max_{u \in V'_G}(\deg(u))$ for calculating modularity gains, plus the depth of the sorting algorithm.

This is usually poly-logarithmic in $|V'_G|$, but TBB's quick-sort implementation uses sequential partitioning, and thus is linear. We tested a better sorting algorithm [AWFS17] but measured substantial slowdown, so we did not investigate further. To reduce the sorting overhead in practice, we split the updates into two vectors (addition and subtraction) which are sorted independently in parallel, but applied one after another.

The number of moves and degree linear terms in the depth may be reduced to poly-logarithmic by parallelizing the per-vertex gain calculation (parallel for loop over neighbors, atomic fetch-add for weight to neighbor clusters) and aggregating updates within a community in parallel with a deterministic reduce. However, in practice the outer level of parallelism is sufficient, and we do not know the starting and end positions of a community's sub-range which would incur an additional step to compute. In practice, the execution time is dominated by the work for modularity gain calculations and the sorting overheads.

5.2 Coarsening

As discussed in Section 4.1, in the hypergraph coarsening phase we compute a vertex clustering based on heavy-edge-rating, contract the clustering, and repeat on the coarse hypergraph until convergence or the hypergraph is small enough for initial partitioning. We already described a determinism-friendly contraction algorithm and the adaptations needed to make it actually deterministic in Section 4.1.2. In this section, we show how to make the agglomerative heavy-edge clustering algorithm deterministic.

Algorithm 5.3 shows pseudocode for one round (previously called pass and level). Again, we use the template of Algorithm 5.2 to split vertices into sub-rounds (line 1).

Find Target Clusters. For each vertex u in a sub-round, we store the best target cluster according to the rating function in an array of propositions $\mathcal{P}$ (line 8). If there are multiple candidates with the same rating, we pick one uniformly at random. To achieve deterministic selection we use a hash-and-combine function seeded with u as a random number generator. Since the initial partitioning step must be able to compute a feasible partition, we enforce a maximum weight on the clusters $W_{\max}$. To respect this constraint, we filter the target cluster candidates during the selection, using the weights from the previous sub-round. Additionally, some of the calculated moves must be rejected, which leads to the approval step.

Approval. We sort the moves lexicographically by cluster, vertex weight, and lastly for determinism by vertex ID (see line 19). For each target cluster, we then approve the vertices one by one (in order of ascending weight), and reject all of the remaining moves into this cluster once its weight would exceed $W_{\max}$ (line 24). Our implementation iterates over the moves in parallel, and the iteration of the first vertex in the sub-range of a cluster is responsible for performing the moves into the cluster (line 21). Parallel counting sort is not viable in line 19, as the number of clusters is only reduced by a small amount in each sub-round. Furthermore the IDs have not been remapped to a compact range, which is too expensive if performed in each sub-round.

Optimizations. As an optimization we already sum up the cluster weights during the target-cluster calculation step using atomic fetch-and-add instructions (line 9). If all moves into a

Algorithm 5.3: Coarsening Pass

```
1  randomly split vertices into sub-rounds
2  rep[u] ← u, P[u] ← u : ∀u ∈ V
3  weight[u] ← c(u), opportunistic-weight[u] ← c(u) : ∀u ∈ V
4  for r = 0 to number of sub-rounds do
       // find target clusters
5      for u ∈ V in sub-round r do in parallel
6          if weight[u] ≠ c(u) // skip non-singletons
7              continue
8          P[u] ← ComputeHeavyEdgeRating(u)
9          opportunistic-weight[P[u]] +=(atomic) c(u)
       // pre-approve
10     M ← ∅ // moved vertices
11     for u ∈ V in sub-round r do in parallel
12         if P[u] = u
13             continue
14         if opportunistic-weight[P[u]] ≤ W_max
15             rep[u] ← P[u]
16             weight[rep[u]] ← opportunistic-weight[rep[u]] // benign race
17         else
18             add u to M
       // approval step
19     sort M lexicographically by (P[u], c(u), u)
20     for i = 0 to |M| do in parallel
21         if i = 0 or P[M[i − 1]] ≠ P[M[i]]
22             t ← P[M[i]] // target cluster
23             w ← weight[t]
24             for j = i until w + c(M[j]) > W_max do
25                 rep[M[j]] ← t
26                 w += c(M[j])
27             weight[t] ← w
28             opportunistic-weight[t] ← w
29 contract clustering rep
```

target cluster combined do not exceed $W_{\max}$ (line 14), we approve them all. This drastically reduces the number of moves we have to sort. Due to this optimization, calculating the target clusters is the far more expensive step in practice, even though approving the moves requires sorting. There is a data race in line 16, when setting the actual weight of the cluster to the opportunistic weight. All iterations write the same value which makes the race benign.

We collect moves for which this is optimization does not apply in a vector M, filled from thread-local buffers. Moves in M with the same target cluster are guaranteed to exceed $W_{\max}$ in combination. Therefore, the stop condition of the loop in line 24 does not check whether the next cluster is reached. Instead, $W_{\max}$ will be exceeded which stops the loop.

Analysis. Calculating target clusters has the same work and depth bounds as the non-deterministic version. Let V' denote the vertices in the current sub-round. The pre-approval step has $\mathcal{O}(|V'|)$ work and $\mathcal{O}(\log(|V'|)$ depth since constant work is performed in each iteration. The work of the approval step is $\mathcal{O}(|M| \log(|M|)) \subset \mathcal{O}(|V'| \log(|V'|))$. The depth is determined by the depth of the sorting algorithm ($\mathcal{O}(|M|)$ for TBB's quicksort, poly-log usually), and an additive linear term of the maximum number of moves into a cluster. We can reduce the second term from linear to logarithmic: binary search for the end of the target cluster's sub-range, sum up the weights with a parallel prefix sum, and then find the first position that would exceed $W_{\max}$ with binary search again (or track it during the second pass of the prefix sum).

5.3 Initial Partitioning

After the coarsening phase, we compute an initial k-way partition on the coarsest hypergraph using the same approach as in Section 4.2. We perform parallel multilevel recursive bipartitioning and run the portfolio of flat bipartitioning algorithms. The sequential flat algorithms are inherently deterministic. However, care must be taken when selecting which partition to use for refinement. The primary criterion is connectivity followed by imbalance. Additionally, we assign sequentially generated tags to the initial bipartitions, which are used as a third criterion in case a tie break is necessary. In combination with deterministic coarsening and refinement, the overall initial partitioning phase is deterministic.

We do not use the adaptive selection technique for flat bipartitioning since it is non-deterministic. Furthermore, in the existing configuration each thread runs one FM round on all of the bipartitions it computes, and subsequently a second FM round on the best bipartition out of the ones it computed. This is non-deterministic as we do not control which thread performs which bipartitioning runs. Therefore, we implemented a version that keeps a fixed number of the best partitions found across all runs, and after a synchronization refines these with a second round of FM. However, this lacked diversity in preliminary experiments, so we instead increase the number of FM rounds that are run on all bipartitions from one to three, and omit the second refinement.

5.4 Refinement

In the refinement phase, we take an existing k-way partition (from the previous level or initial partitioning) and try to improve it by moving vertices to different parts, depending on

their gain values. Our refinement algorithm is a synchronous version of label propagation refinement [KK00, RAK07]. The vertices are randomly split into sub-rounds. For each vertex in the current sub-round, we compute the highest gain move, and store it if the gain is positive. The gain calculation algorithm (Algorithm 4.4) and thus the work and depth bounds are the same as in the non-deterministic case. Since gain calculation dominates the running time in practice, we also employ the active set strategy, where only neighbors of moved vertices are considered in the next round.

In a second step we approve some of the stored moves, and subsequently apply them in parallel, before proceeding to the next sub-round. This is the interesting part, as just applying all moves does not guarantee a balanced partition.

Approval Idea. We perform a sequence of balance-preserving vertex swaps on each block-pair, prioritized by gain. This approach was first introduced for SHP [Kab+17a], though their work only considers unweighted vertices. For each block-pair $(s, t) \in \binom{[k]}{2}$, we collect the vertices M_{st} that want to move from s to t and M_{ts} from t to s, and sort both sequences descendingly by gain. The vertex ID is used as tie breaker to get a deterministic order.

SHP moves the first $\min(|M_{st}|, |M_{ts}|)$ vertices from each sequence. With unit weights, this does not change the balance of the partition. However, we have to handle non-unit weights, so we are interested in the longest prefixes of M_{st}, M_{ts}, represented by indices i, j, whose cumulative vertex weights $c(M_{st}[0..i]), c(M_{ts}[0..j])$ are equal.

Budgets. The sums do not need to be exactly equal, as long as the resulting partition is still balanced. For each block $t \in [k]$, we have a certain additional weight budget β_t it can take before becoming overloaded.

Block pairs can be handled sequentially one after another or independently in parallel. If they are handled sequentially, we can set $\beta_t = L_{\max} - c(V_t)$, where $c(V_t)$ reflects already approved moves at the time we use it. If they are handled in parallel, we divide this budget equally among the different block-pairs that have moves into t, with the initial $c(V_t)$. Surprisingly, we did not observe quality differences in practice, even when randomizing the order in which block pairs are handled, which is why we handle them in parallel.

The prefixes i, j are called *feasible* if they satisfy the condition $-\beta_s \leq c(M_{st}[0..i]) - c(M_{ts}[0..j]) \leq \beta_t$. The partition obtained from swapping $M_{st}[0..i]$ and $M_{ts}[0..j]$ is balanced if this condition is fulfilled (if and only if for sequential budgets).

The Sequential Algorithm. We compute the prefix indices i, j similar to the way two sorted arrays are merged. Initially, we set $i \leftarrow 0, j \leftarrow 0$. We simultaneously iterate through both sequences and keep track of the so far exchanged weight. If $c(M_{st}[0..i]) - c(M_{ts}[0..j]) < 0$ and M_{st} has moves left, we approve the next move from M_{st} by incrementing i. Otherwise, we approve the next move from M_{ts} by incrementing j. In each iteration, we check whether the current prefixes are feasible. If so, we store the pair i, j as the current best. Longer

prefixes are preferred (since we only consider positive gain moves), so the result is the last feasible pair we encounter.

The Parallel Algorithm. We use a simple divide-and-conquer parallelization for merging sorted arrays. First, we compute the cumulative vertex weights via parallel prefix sums. Then the following algorithm is applied recursively. We binary search for the cumulative weight of the middle of the longer sequence in the shorter sequence. The left and right parts of the sequences can be searched independently. If the right parts contain feasible prefixes, we return them. Otherwise, we return the result from the left parts. The left-most path in the recursion tree is guaranteed to find at least the trivial solution $i = j = 0$ (no move applied). If r denotes the length of the longer sequence, this algorithm has $\mathcal{O}(\log(r)^2)$ depth and $\mathcal{O}(r)$ work.

Optimizations. Since we are interested in the longest prefixes, we can omit the recursive call on the left parts if the prefixes at the splitting points are feasible. This is fairly likely, since the cumulative weights are as close to each other as possible (due to binary search). Further, we can omit the recursive call on the right parts if the cumulative weight at the middle of the longer sequence exceeds the cumulative weight at the end of the shorter sequence plus the appropriate budget. Note that in this case the binary search finds the end and thus the left recursion works on the entirety of the shorter sequence.

We fall back to the sequential algorithm once both sequences have less than 2000 elements. This value worked well in preliminary experiments. As we already computed cumulative weights, we perform the simultaneous traversal from the ends of the sequences. Since we expect to approve the majority of the saved moves, this is likely faster.

There is a merging algorithm with depth $\mathcal{O}(\log(r))$ and $\mathcal{O}(r)$ work [Vis10] based on ranking $r/\log(r)$ equidistant splitters in the opposite sequence and then merging $2r/\log(r)$ (parallel) counter-part sub-sequences of size $< \log(r)$ (sequential merge). However, we cannot expect any performance benefits in practice, because this step is inexpensive compared to gain calculation.

Implementation Details. We collect all positive gain moves of a sub-round in a vector M with thread-local buffers. To organize the moves, we sort M by the move direction using parallel counting sort (k^2 possible keys is decently small). For each block pair (s, t) we sort the two sub-ranges M_{st}, M_{ts} (identified by the positions array from counting sort) by gain and vertex ID (for determinism), using TBB's quicksort. Additionally, we compute the cumulative weights of vertex moves using parallel prefix sums. Subsequently, we run the parallel prefix selection algorithm described above.

Once we have determined the prefixes for all block pairs, we apply the approved moves to the partition. We do this by iterating over M in parallel. A move is approved if its position in M is smaller than the prefix index for its move direction computed in the previous step. This condition can be checked independently in each iteration.

We compute attributed gains, and sum them with a reduction to obtain the overall connectivity reduction. The overall attributed gain can be negative, due to the calculated gains not considering concurrent moves in their neighborhood. In this case, we revert all moves of the current sub-round and double the number of sub-rounds for the next round. This increases the likelihood of finding an overall positive gain.

We add neighbors to the active set for the next round (not sub-round) as in the non-deterministic algorithm. To achieve a deterministic random permutation (sub-round split), we sort the active set using TBB's quicksort after each round (unless we are done). Note that the active set is typically small after the second round, since most moves happen in the first.

5.5 Randomization

Our algorithm to perform the randomized splitting is based on the RandSort and RandDist algorithms for random shuffling [CB05]. We divide the input range into a fixed number of equal-size chunks, which are handled independently in parallel. To achieve deterministic results even when varying the number of threads, we always use 256 chunks. For each chunk, a random number generator is seeded with an input seed and the index of the first element, which is then used to generate 8-bit tags for each element in the chunk. The input range is then sorted by the tags using parallel counting sort, which returns an array of offsets where each tag starts. Multiple tags are further grouped together to form sub-rounds, depending on the number of requested sub-rounds. The order within the same tag is deterministic because counting sort is stable.

This algorithm can be extended to random shuffling. The elements with the same tag are shuffled sequentially. Different tags are handled independently in parallel.

Since the random number generator is used sequentially on a chunk we are stuck with static load balancing. Considering that randomization is never a bottleneck and only $\mathcal{O}(1)$ work is performed per iteration this is fine.

A way to eliminate static load balancing is to use random number generators that allow skipping $r \in \mathbb{N}$ steps in $\mathcal{O}(\log(r))$ time. This would enable using chunks of size $\log(r)$ with overall linear work still. However, such generators are not available in the `C++ STL`.

5.6 Differences to BiPart

We now discuss differences between BiPart and our algorithm. BiPart uses recursive bipartitioning, whereas we use direct k-way, which is superior regarding solution quality [ST97, San89]. Furthermore, our algorithms are randomized. The refinement algorithms are similar.

Both are based on label propagation [RAK07] and the prefix-pair swap method from SHP [Kab+17a]. However, BiPart's refinement ignores vertex weights, which leads to imbalanced partitions that must be repaired by explicit rebalancing. This can be slow and offers little control by how much connectivity degrades, as there is no control on how imbal-

anced the partition is. Our refinement guarantees balanced partitions at all stages without rebalancing.

Additionally, we use sub-rounds for more accurate gains and the active set optimization for better performance. Furthermore, BiPart uses no mechanism to track actual improvements, whereas we use attributed gains to detect and prevent quality-degrading moves.

Their coarsening scheme assigns each vertex to its smallest incident hyperedge (ties broken by ID) and merges all vertices assigned to the same hyperedge. This does not consider hyperedge weights: eliminating a heavy-weight net from the hypergraph (and thus from the possibility of being cut) is in no way preferred to a low-weight net. Furthermore, this scheme offers no control over the weights of the coarse vertices. Preventing large vertex weights is important so that initial partitioning can find balanced partitions, and there is more leeway for optimization. BiPart uses a parallel version of greedy graph growing [CA99] for initial partitioning. However the coarsest hypergraphs are small. This makes achieving speedups difficult. We believe a better investment of the running time is to perform parallel diversified repetitions of sequential algorithms, in order to achieve better partition quality.

5.7 Ideas and Challenges for deterministic FM

While enforcing determinism for label propagation refinement is fairly easy without straying from the spirit of the algorithm, this is certainly not the case for parallel localized FM. We identify the following main challenges and outline ideas to solve them.

During localized expansion, vertices must be acquired. So far this is first come first serve, which is clearly non-deterministic. It is possible to forgo localization and instead pre-assign vertices statically. In the previous chapter we showed that this is detrimental to partition quality, which makes this option unattractive. Instead, we envision a version where we run each localized search for some time. Each search collects vertices it would like to acquire instead of acquiring them right away. At synchronization points, we establish consensus which search gets to own which vertex, and then continue the localized searches.

As in the other local moving algorithms, we need to hide moves from other threads until a synchronization point. This will bloat the thread-local hash-tables for partition data structure deltas, since we cannot clear them as soon as we find an improvement. Abandoning these data structures is not an option due to their performance benefits. However, we have not investigated whether the memory usage would actually be obstructive in practice.

Synchronization would have to happen quite frequently. Otherwise, searches run out of vertices to move, lack up-to-date gains, and may exhibit excessive memory usage. This is bad for long-running searches, which are important, as the searches without improvement terminate quickly.

Much of localized FM's efficiency is due to its completely asynchronous nature in the moving phase. Frequent synchronization will incur bad load imbalance, since threads cannot join the barrier after a certain time or once enough threads are waiting. The searches have to perform an a priori determined workload between synchronizations, e.g., a fixed number

of steps. In practice this could be counteracted (at least partially) by over-subscription, i.e., spawn substantially more searches than threads. The downside is increased interference. More searches want the same vertices and perform hidden moves close to other search regions.

Due to these unresolved challenges and time constraints, we currently do not have a parallel deterministic implementation of FM refinement, but are interested to pursue this in future work. Therefore, our deterministic partitioner only uses label propagation in the refinement phase.

5.8 Experimental Evaluation

Our code is written in `C++17`, uses Intel's `TBB` library for parallelization and is compiled with `g++` version 9.2 with optimization level `-O3` and native architecture optimizations. The experiments are run on machine B and on benchmark set B, comparing our deterministic algorithm with other parallel algorithms. Additionally, we conduct scalability experiments demonstrating very good speedups, examine the impact of the number of sub-rounds on partition quality, and determine the cost of determinism for partitioning.

5.8.1 Configurations

We perform 5 rounds of local moving on each level during refinement, 5 rounds before contracting during preprocessing, and one round before contracting during coarsening. These values are the default values of the configuration in Chapter 4. We call the algorithm and configuration proposed in this work Mt-KaHyPar-SDet, and the equivalent configuration that uses the existing non-deterministic local moving algorithms Mt-KaHyPar-S, where `Det` stands for determinism, and `S` for speed. Additionally, we consider the configuration Mt-KaHyPar-D (for default) from Chapter 4, which has the original non-deterministic initial partitioning configuration (Section 4.2) and uses parallel localized FM.

5.8.2 Parameter Tuning

The supposedly most important parameter is the number of sub-rounds used, as it offers a trade-off between scalability (synchronization after each sub-round) and solution quality (more up-to-date gains). In the following, we show that this is actually not a trade-off, as the number of sub-rounds either does not affect solution quality, or using fewer sub-rounds even leads to better quality.

We made an initial guess of 5 sub-rounds for refinement, and 16 sub-rounds for coarsening and preprocessing, which we use as a baseline configuration when varying each parameter. Figure 5.1 shows the performance profiles. The parameter sets differ, because we had to expand the set for coarsening.

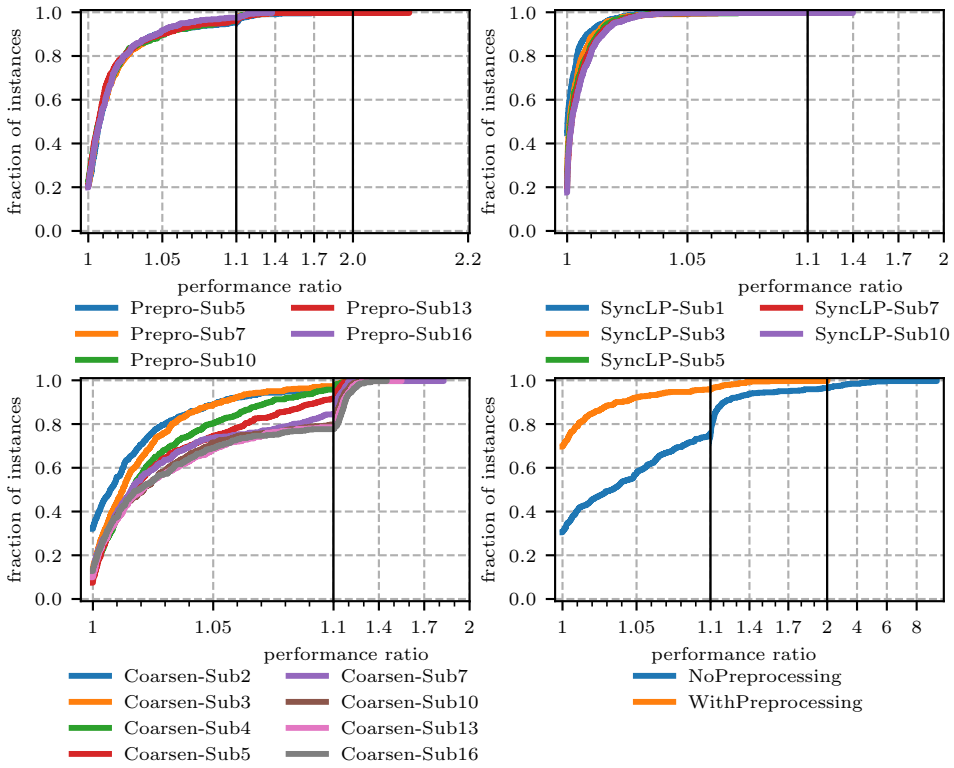

Figure 5.1: Impact of the number of sub-rounds and preprocessing on partition quality.

The largest impact is on the coarsening phase, where two sub-rounds performs the best. Such a small value is surprising. One possible explanation is that high-degree vertices attract low-degree vertices too quickly if synchronization happens too frequently. This is a known weakness of our clustering algorithm, since we do not have a penalizing term for heavy clusters. We only enforce a weight constraint. Hiding this phenomenon for a bit gives slower growing clusters a chance in the tie-breaks.

Using only one sub-round is excluded here, since the clustering algorithm oscillates heavily, with many vertices remaining singletons. This leads to coarsening converging long before the contraction limit is reached and thus initial partitioning takes very long. Due to the same effect, using two sub-rounds is about 12% slower than using three sub-rounds in the geometric mean. Since it gives only slightly worse solution quality, we choose three sub-rounds for coarsening in the main experiments.

For preprocessing, there is almost no impact on solution quality. We stick with our original choice of 16 sub-rounds since the floating-point-aggregation handling is slower if

more vertices are in a sub-round due to the sorting overhead.

For refinement, there are minor differences, but greater than in preprocessing. Here, one sub-round narrowly emerges as the best choice. This is again surprising, as frequently synchronizing should allow for more informed move-decisions. One cause we noticed is that with more sub-rounds the pair-wise swaps did not have a sufficient number of moves to balance, as moves from earlier sub-rounds are not considered. Using such moves as back-up could be included in future versions of the algorithm.

Finally, we should justify why we use the community detection preprocessing. We have shown the quality impacts in Chapter 4 already. However, to incorporate determinism we had to deal with quite unpleasant floating point issues for the volume updates, so we re-confirm the choice. Figure 5.1 (bottom right) shows significantly worse quality when the preprocessing is turned off. The plot shows the qualiy of the final partition, not the initial partition.

5.8.3 Speedups

In Figure 5.2 we show self-relative speedups of the overall algorithm and the separate components, plotted against the sequential running time on that particular instance. In addition to the scatter plot, we show rolling geometric means with window size 50. The overall geometric mean speedups of the full partitioning process are 3.91, 7.04, 12.79, 21.32, 28.73, 29.09 for 4, 8, 16, 32, 64, and 128 threads, respectively, and the maximum speedups are 4.9, 8.7, 15.8, 29.1, 48.9, and 72.6. Since our algorithms are memory-bound workload types these are very good results, and they are better than the speedups for the default configuration.

On about 37% of the runs with 4 threads, and 0.32% of runs with 8 threads, we observe super-linear speedups which occur in all phases except initial partitioning. We identified two reasons for this. First, even sequential runs had running time fluctuations, and as super-linear speedups occur mostly on instances with small sequential times, the speedups are more easily affected. Secondly, we sort vectors that are filled in non-deterministic order. Sorting algorithms in library implementations (such as TBB's quicksort) have checks for presorted sub-sequences to speed up execution. This leads to fluctuating amounts of work in different runs.

Looking at speedups for the individual phases, we see that most phases exhibit very consistent speedups, even for small sequential running times. Only initial partitioning exhibits sub-par speedups on larger instances, which is due to load imbalance from long running sequential FM refinement. It may be worthwhile to detect and prune overly long running searches, though this is clearly non-deterministic.

With 128 threads (only rolling geometric means shown for readability), the running times still improve, though not as drastically. Only small instances show a slight slowdown, predominantly in initial partitioning. Each socket has 64 cores, so the second memory socket is used, and thus some slowdown is expected. By default, we set the memory allocation

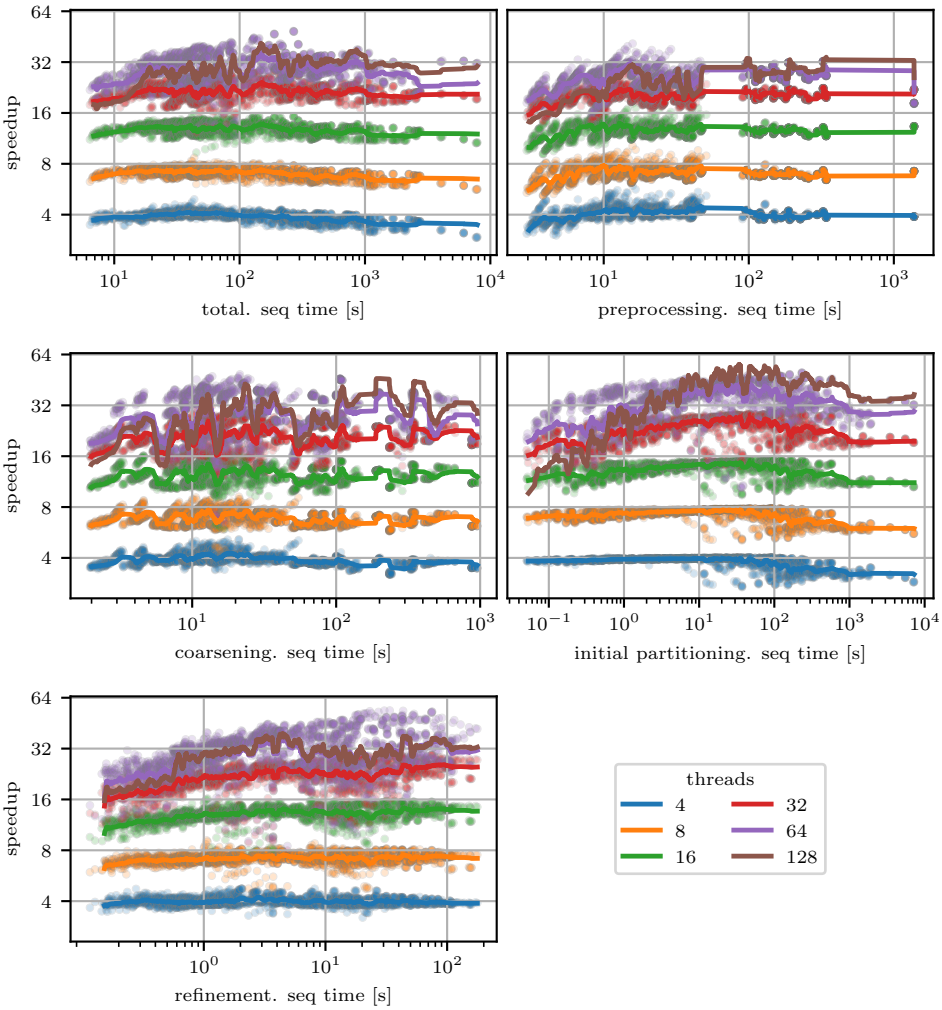

Figure 5.2: Speedups for Mt-KaHyPar-SDet in total as well as its components separately. The x-axis shows the sequential time in seconds, the y-axis the speedup. The lines are rolling geometric means (window size 50) of the per-instance speedups (scatter).

policy to interleaved if multiple sockets are used, in order to distribute memory latencies evenly.

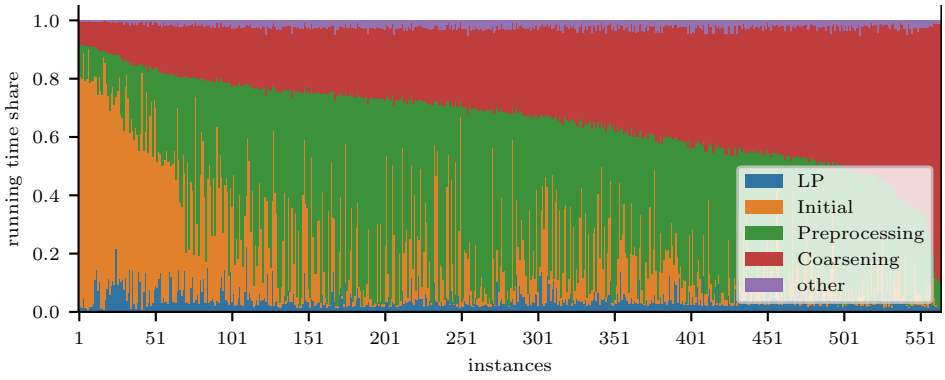

Figure 5.3: Shares of running time spent in each of the components of Mt-KaHyPar-SDet.

5.8.4 Running Time Share

From the sequential times in the speedup plots we already get a broad idea how much running time is spent in which phase of the multilevel algorithm. Figure 5.3 shows this break-down into phases per instance when run on 64 cores. Coarsening and community detection preprocessing make up the bulk of the work, whereas label propagation refinement is negligible compared to the other two. While these are all local moving algorithms, the cluster IDs for label propagation are small and thus cluster ratings always fit in cache, which is not the case for community detection and coarsening. Calculating moves is the most expensive step in these algorithms, so these results are expected. Initial partitioning generally takes less time than preprocessing and coarsening, but there are some instances where it takes the longest. Often this is caused by coarsening converging before reaching the vertex limit, such that initial partitions are computed on larger hypergraphs than desired. This emphasizes again that it is important to parallelize each phase.

5.8.5 Comparison with other Algorithms

Figure 5.4 (left) shows performance profiles comparing our new algorithm with its non-deterministic predecessors, the non-deterministic distributed algorithm Zoltan [Dev+06] as well as the deterministic BiPart algorithm [MABP21]. Each algorithm was run with 64 threads. Note that the number of threads does not impact the partitions computed by BiPart and Mt-KaHyPar-SDet. We omit comparisons with sequential algorithms, since determinism is not a challenge for them, and we have established Mt-KaHyPar-D from Chapter 4 as the state-of-the-art baseline for fast algorithms. Slow algorithms based on n-level (un)coarsening (sequential KaHyPar) or many repetitions and V-cycles (hMetis) represent a vastly different trade-off and would not finish on these instances in reasonable time.

As expected, Mt-KaHyPar-D finds the highest quality solutions out of the compared

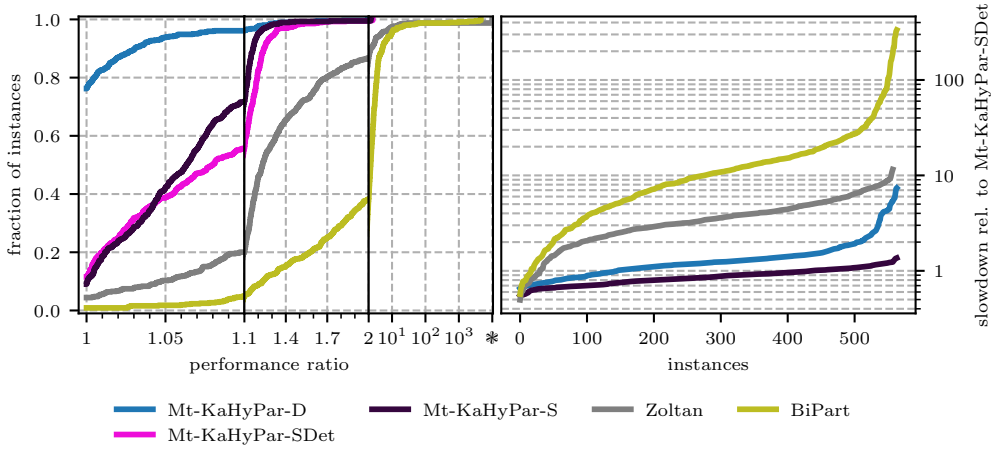

Figure 5.4: On the left: solution quality of BiPart, Zoltan, our algorithm Mt-KaHyPar-SDet and the existing Mt-KaHyPar variants. The ❋ symbol marks timeouts (6 instances for Zoltan). On the right: slowdown relative to our algorithm. The instances on the x-axis are sorted independently.

algorithms, due to FM. It contributes the best solutions on about 75% of the instances, followed by Mt-KaHyPar-SDet and Mt-KaHyPar-S which are similar. However, Mt-KaHyPar-S is slightly better than Mt-KaHyPar-SDet as the former converges faster towards 1, and there is a noticeable gap starting at factor 1.05. BiPart is far off. It contributes only 6 of the best solutions, and its quality is off by more than a factor of 2 on more than 50% of the instances; on some instances even by three orders of magnitude. Zoltan is situated between BiPart and Mt-KaHyPar-S. In a direct comparison, Mt-KaHyPar-SDet computes better partitions than BiPart on 551 of the 564 instances with a geometric mean performance ratio of 1.0032 compared to BiPart's 2.3805. Recall that performance ratio refers to partition quality, not running time.

In Figure 5.4 (right), we report relative slowdowns, i.e., the running time of the other algorithm divided by running time of the baseline Mt-KaHyPar-SDet. Mt-KaHyPar-S is faster on 406/564 instances but never by a factor of more than 2. BiPart is between *one and two orders of magnitude slower* than the two speed variants of Mt-KaHyPar. A partial reason for this is shown in Figure 5.5 which plots self-relative speedups of BiPart. Most speedups are below 2 and the largest speedup is about 7. Unfortunately, the plot is not very legible but on the other hand this is not necessary. In theory, the algorithms employed in BiPart should scale as well as ours. The poor scaling in practice is due to the implementation and the parallelization library.

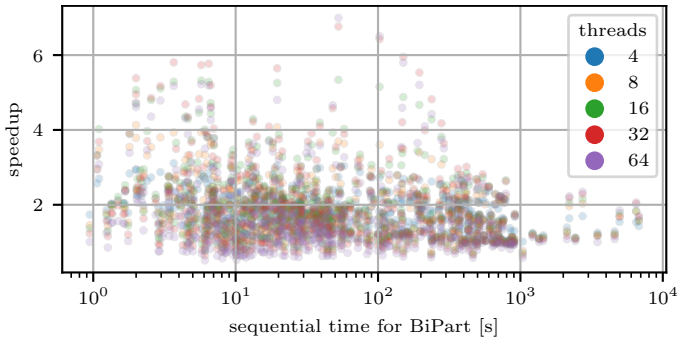

Figure 5.5: Instance-wise speedups for BiPart. The rolling geometric means are omitted.

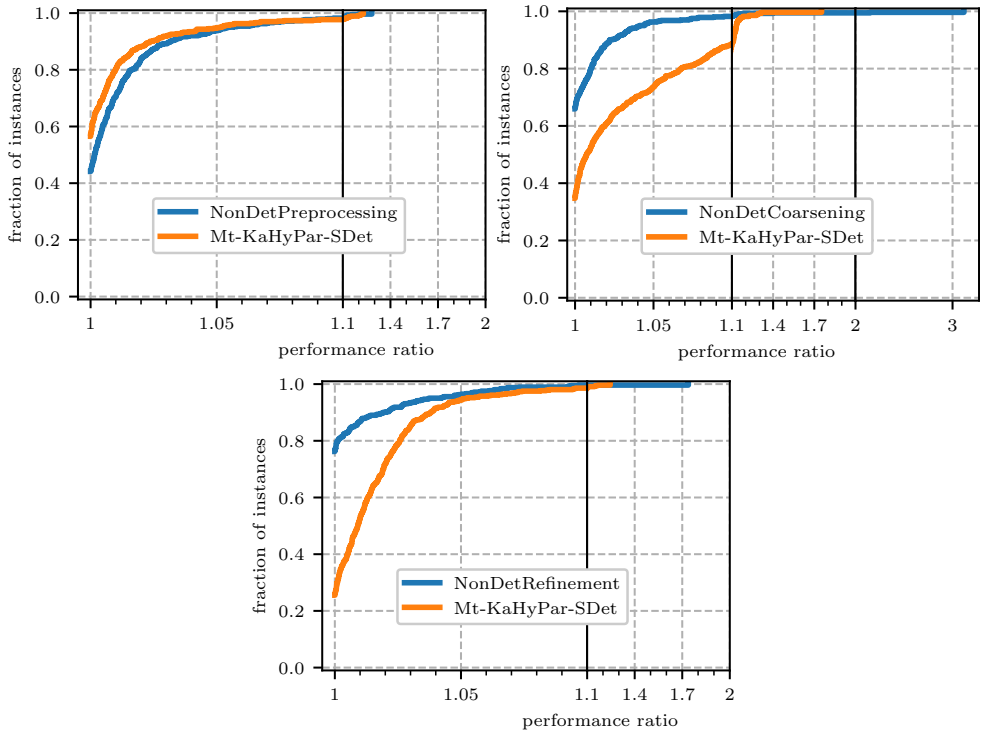

Figure 5.6: Impact of determinism in each component on the final partition quality.

5.8.6 The Cost of Determinism

In this section, we investigate in which phase the solution quality gets lost, by swapping out one component for its non-deterministic counterpart in each of the plots in Figure 5.6. Interestingly, the biggest quality loss comes from coarsening, whereas deterministic preprocessing even improves quality. The loss in refinement is expected due to the lack of up-to-date gains and the inability to leverage zero gain moves for rebalancing and diversification.

For coarsening, the results are unexpected, particularly because similar local moving algorithms for community detection [HSWZ18] are not as affected by out-of-date gains. One reason for this is that these algorithms perform multiple global rounds, where vertices can back out of their first cluster assignment. We only use one round and even prematurely terminate the round to avoid coarsening too aggressively. Performing a second round, where only already clustered vertices may reassess their assignment, may be beneficial and we leave this for future research. Additionally, we unsuccessfully experimented with two features of the non-deterministic coarsening, but both affected the partition quality negatively. We considered adapting hyperedge sizes to the current clustering in the rating function, and *stable leader chasing*, where oscillations (vertices joining each other) and cyclic joins are resolved by merging all involved vertices.

5.9 Conclusion and Future Work

We presented the first scalable, deterministic parallel hypergraph partitioning algorithm. Our experiments show that determinism does incur sacrifices regarding both solution quality and running time compared to the previous non-deterministic version. These are small enough to justify if determinism is desirable for the application.

In future work we would like to make additional refinement algorithms deterministic, such as parallel localized FM or flow-based refinement. For localized FM, we already identified the main challenges, and we believe these are solvable with engineering effort. Making flow-based refinement deterministic is easier. The idea is to schedule two-way refinements on independent block pairs, where we can synchronize after each block participated in one refinement. On each block pair, we solve incremental maximum flow problems. The flow algorithm need not be deterministic, as the source-side and sink-side cuts are unique. The flow assignment only influences the insertion order into a data structure we use to construct the next flow problem. Therefore, we only need to make the insertion order deterministic, which is not a performance-critical part.

Further work is necessary to investigate the quality issues in the coarsening phase. On the engineering side, we would like to eliminate the overhead for sorting when aggregating moves into the same cluster (for preprocessing and coarsening). Promising options are semi-sorting or the hash-and-distribute approach employed for identical net detection. We still need to sort within a bucket, but can do so sequentially on a smaller input.

6 Parallel n-Level Hypergraph Partitioning

Motivation and Context. Our parallel multilevel algorithm Mt-KaHyPar-D from Chapter 4 finds moderately worse partitions than KaHyPar-CA. As we already ruled out that parallelism impacts partition quality, the component responsible for this difference is the n-level uncoarsening, thanks to its fine-grained refinement granularity. At first glance, this method appears inherently sequential, as only one vertex is contracted at a time. Correspondingly, in each step of refinement, only a single vertex is uncontracted, allowing a highly localized search for improvements.

Contributions. To bridge this remaining quality gap, we present an approach to parallel n-level (un)coarsening in this chapter. We devise a parallel execution order for contractions based on a representation of contractions as a forest and on-the-fly conflict resolution. The contractions are performed on a highly intrusive semi-dynamic hypergraph data structure with fine-grained locking. After initial partitioning, the forest is decomposed into *batches*, each containing a fixed number of vertices, which can be uncontracted independently in parallel and are used as seeds for highly localized refinement. The result is a highly scalable system (self-relative speedups around 25 on 64 cores), which computes partitions with the same quality as sequential KaHyPar-CA.

Attributions. This chapter is based on a joint publication [GHSS22] with Tobias Heuer, Peter Sanders and Sebastian Schlag, as well as a follow-up work presented in the Master thesis of Moritz Laupichler [Lau21] which was supervised by Tobias and me. Tobias and I wrote the paper, with editing by Peter Sanders and Sebastian Schlag. The n-level (un)coarsening code was written by Tobias Heuer with consulting by me. To make the approach feasible, I

Algorithm 6.1: Parallel n-level Hypergraph Partitioning

```
   Input: Hypergraph H = (V, E), number of blocks k
   Output: k-way partition Π of H
1  ∀v ∈ V : rep[v] ← v                                   // initialize empty forest F
2  while n > t · k do
3      for u ∈ V in random order do in parallel
4          v ← HeavyEdgeRating(u)
5          if (u, v) can be safely added to F
6              rep[v] ← u and contract v onto u
7  Π ← InitialPartition(H, k)
8  B = ⟨B_1, ..., B_l⟩ ← ConstructBatches(F)
9  for B ∈ B do
10     for v ∈ B do in parallel
11         uncontract v from rep[v]
12         Π[v] ← Π[rep[v]]
13     improve Π with localized refinement on B
```

worked on overall performance engineering of the framework, with a particular focus on eliminating overheads in the refinement routines.

6.1 Overview

Algorithm 6.1 shows a high-level pseudocode of our parallel n-level framework. Contracting and uncontracting vertices in a strict order is inherently sequential, which is why we have to relax the n-level paradigm. For the coarsening phase, we parallelize the loop over the active vertices in line 3 to select contraction partners. Contractions are performed on-the-fly as in the sequential algorithm, completely asynchronously with the contraction partner selection. To this end, we propose a new low-overhead hypergraph data structure in Section 6.2, and describe how to implement contractions and uncontractions on it.

In Section 6.3 we address two challenges regarding contractions. A vertex can only be contracted once all contractions we want to perform onto it are finished. We define the *contraction forest* $\mathcal{F}$ as the rooted forest induced by the array rep of vertices pointing to the vertex they are contracted onto (their parent). This *children before parents* condition seems intuitively necessary and correct. If a parent u is contracted before its child v, it does not know all of its incident nets yet, which would have to be propagated to rep$[u]$. In Section 6.2 we additionally pin-point which data structure invariants we need this condition for. The second challenge is to incrementally construct $\mathcal{F}$ without adding cycles (see line 5), since we do not know $\mathcal{F}$ beforehand.

For initial partitioning, we use the same approach as in Chapter 4, but plug in the n-level

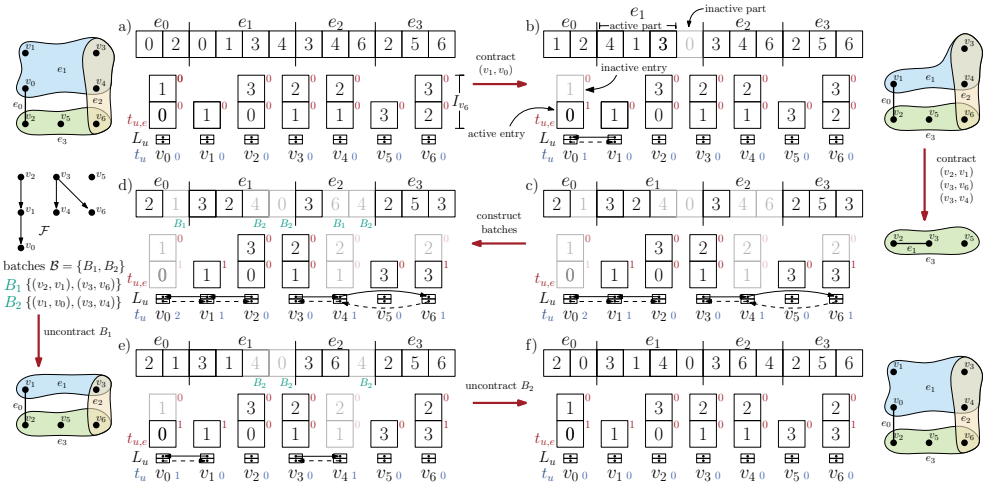

Figure 6.1: Contractions and uncontractions applied on the dynamic data structure.

coarsening and uncoarsening for recursive multilevel bipartitioning.

The uncoarsening phase starts in line 8. We construct a sequence of batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$ of contracted vertices, such that $|B_i| \approx b_{\max}$, where $b_{\max}$ is an input parameter. Batches are processed one after another, enabling the uncontraction of vertices in subsequent batches. This leads to $\mathcal{O}(n/b_{\max})$ levels. Vertices in the same batch are uncontracted in parallel, such that $b_{\max}$ offers a trade-off between parallelism and levels.

In sequential n-level partitioning, the uncontractions are performed in the exactly reverse order of the contractions. This is done to *correctly* project the partition up in the hierarchy, but is of course inherently sequential. We show that it suffices to only revert the contractions of siblings in reverse order to correctly project the partition, in addition to parents before children. This enables parallel uncoarsening as detailed in Section 6.4.

After uncontracting each batch, we run highly-localized refinement algorithms around the batched vertices. In Section 6.5 we discuss how to adapt the gain table to n-level uncoarsening. In order to avoid reinitialization after each batch, we have to update it through the information obtained from uncontractions. Localizing the existing parallel refinement algorithms is straight-forward. Therefore, the main contributions of this chapter are the parallel n-level coarsening and uncoarsening algorithms.

After presenting our experimental results, we discuss a follow-up work where uncontractions are performed completely asynchronously as well.

6.2 Semi-Dynamic Hypergraph Data Structure

To support concurrent on-the-fly contraction and uncontraction, we use a semi-dynamic hypergraph data structure that improves the approach of KaHyPar [Sch20, p. 100]. In KaHyPar, the pins of the nets are represented as a CSR (the sub-range storing the pins of a certain net is called its pin-list), whereas the incident nets of the vertices are represented using adjacency-lists, i.e., as a separate vector for each vertex.

A contraction (u, v) merges $\mathrm{I}(v)$ into $\mathrm{I}(u)$ and removes v from the hypergraph. Adjacency lists are the more flexible data structure for dynamic hypergraphs. It is possible to add entries to the incident nets vector without rebuilding the entire data structure, which is why this choice was made. With adjacency lists $\mathrm{I}(v) \setminus \mathrm{I}(u)$ is copied to $\mathrm{I}(u)$.

With repeated contractions onto the same vertex this can lead to quadratic memory usage in the worst case, and is therefore not suitable for hypergraphs that are very large or have a skewed degree distribution. Following the approach of Bader and Cong [BC06] for parallelizing Boruvka's minimum spanning tree algorithm, we organize merged vertices in a linked list, such that no copy and reallocation operations are necessary to merge adjacency lists. In preliminary experiments, this data structure was roughly 5% slower than KaHyPar's approach due to iteration overheads, but allows us to handle larger and highly skewed instances. Note that Bader and Cong used this to speed up the contraction algorithm, whereas our intention is to save memory.

The key idea is to remove $I(u) \cap I(v)$ from $I(v)$ (instead of adding $I(v) \setminus I(u)$ to $I(u)$). $I(u)$ is obtained by iterating over both the representation of the current $I(u)$ and the remaining entries of $I(v)$. For each vertex $u \in V$, we store an array I_u which is initialized with the incident nets of u on the input hypergraph. We organize all vertices contracted onto u as well as u itself in a doubly-linked list L_u, so that the *current state* for $I(u)$ is obtained by iterating over all I_w arrays for $w \in L_u$. When contracting v onto u, we remove any incident net of u from the arrays I_w for $w \in L_v$ and append L_v to L_u. Since this flattens the list, L_u contains all descendants of u in $\mathcal{F}$.

The pin-lists of nets are stored in CSR format. Each pin-list is split into an *active* and *inactive* part. The active part contains the pins that are currently part of the net, the inactive part contains pins that were previously part of that net, but were contracted.

The data structure and all operations, which we describe in more detail in the following, are illustrated with an example of multiple contraction and uncontraction steps in Figure 6.1. In each step, the top part shows the current state of the pin-lists, the bottom part shows the incident net arrays I_w and lists L_w.

6.2.1 Remove and Restore Incident Nets.

To remove and later restore entries from an incident net array I_w, we additionally store a counter t_w which counts in how many contractions I_w was modified, as well as a marker $t_{w,e}$ for each entry of I_w. The counter and markers are initially set to zero. Entries with markers $\geq t_w$ are *active*, i.e., were not removed yet. To remove a set X of entries from I_w, we

Algorithm 6.2: Contraction Operation

```
1  c(u) ← c(u) + c(v)
2  for e ∈ I(v) do                                   // edit pin-lists
3      if u in pin-list of e                         // e ∈ I(u) ∩ I(v)
4          remove v from pin-list of e
5          mark e in bitset X
6      else                                          // e ∈ I(v) \ I(u)
7          replace v by u in the pin-list of e
8  for w ∈ L_v do                                    // edit incident net arrays
9      remove active entries in X from I_w
10 append L_v to L_u
```

increment t_w and iterate over the previously active entries of I_w (now marked with $t_w - 1$). If the entry is not in X, we set its marker to t_w. Otherwise, we swap the entry and its marker to the end of the active part but keep its marker at $t_w - 1$, thereby marking the entry inactive. This maintains the invariant that the entries of I_w are sorted by decreasing markers, so that iterating over active entries of I_w has no overhead. In particular, the iterator for $I(u)$ has a time complexity of $\mathcal{O}(\deg(u) + |L_u|)$. Recall that $I(u)$ changes during coarsening and is made up of the active entries in the I_w arrays for $w \in L_u$.

To restore entries, we just decrement t_w, so that we consider entries marked with $t_{w,e} = t_w - 1$ as active again. The restore operations must be performed in reverse order of the remove operations, in order to restore the correct entries in the sorted order. This corresponds exactly to the dependency children before parents in $\mathcal{F}$ during coarsening, and parents before children during uncoarsening.

In Figure 6.1 b) we contract v_0 onto v_1. We remove e_1 from the incident net array of v_0, but keep e_0. Therefore, we increment t_0 to 1, and set $t_{0,0}$ to 1, while leaving $t_{0,1}$ at 0. In step f (which uncontracts v_1), we decrement t_0 to 0 and thus mark e_1 as active again in the incident net array of v_0.

6.2.2 Contraction Operation

Algorithm 6.2 shows the pseudocode for contracting a vertex v onto another vertex u. To edit the pin-lists, we iterate over the incident nets $e \in I(v)$ and search for the position of u in e (see line 3).

If $u \notin e$, we replace v by u, reusing its slot, see line 7. For example in Figure 6.1 b) v_0 is replaced by v_1 in e_0 (first slots of e_0).

If $u \in e$ already, we remove v by swapping it to the end of the active part and decrement the current size of e, see line 4. In Figure 6.1 b) v_0 is removed from e_1 by moving it from its slot (first) to the inactive part (last slot). Additionally, we mark e in a bitset X if $u \in e$. After

Algorithm 6.3: Uncontraction Operation

```
1  Π(v) ← Π(u)
2  restore the sublist L_v from L_u
3  for w ∈ L_v do
4  |   t_w ← t_w − 1
5  |   for active entries e ∈ I_w do
6  |   |   if t_{w,e} = t_w                          // e ∈ I(u) ∩ I(v)
7  |   |   |   restore v in the pin-list of e
8  |   |   else                                      // e ∈ I(v) \ I(u)
9  |   |   |   replace u by v in pin-list of e
10 c(u) ← c(u) − c(v)
```

editing the pin-lists, we use this bitset to remove all nets $e \in I(u) \cap I(v)$ from the incident net arrays I_w for $w \in L_v$, see line 9.

To enable concurrent contractions, we use a separate lock for each net to synchronize edits to the pin-lists. In line 10, the set $I(u)$ may change due to concurrent contractions onto u, which is why it is not thread-safe to initialize X by iterating over $I(u)$. If multiple vertices that are contracted concurrently onto u share a net e, only the first pin-list edit of e can do the replacement (if u was not already in e). All subsequent edits of e, triggered by contraction onto u, remove their pins and mark e in their thread-local bitset X.

Operations on the incident net arrays I_w for $w \in L_v$ are not synchronized (see line 9), since I_w is only modified by contracting vertices in L_v. These are the descendants of v in $\mathcal{F}$ and therefore their contractions must be finished before the contraction of v starts.

If $c(u) + c(v)$ exceeds the maximum vertex weight $W_{\max}$, we discard the contraction. We use a separate lock for each vertex $u \in V$ to synchronize modifications to L_u and $c(u)$. According to the parallel loop in line 3 of Algorithm 6.1, we would not need a lock on u since only one vertex gets contracted onto it per pass. However, we may reverse the roles of v and u as a performance optimization if v has very high degree, so we can edit the smaller incident nets list.

6.2.3 Uncontraction Operation

Algorithm 6.3 shows the pseudocode for uncontracting a vertex v that is contracted onto a vertex u. To splice L_v out of L_u in line 2, we additionally store the last vertex in L_v at the time v is contracted. To restore the incident nets of v that were removed, we iterate over all vertices $w \in L_v$ and decrement the counter t_w in line 4. This reactivates all entries of I_w that became inactive due to contracting v, i.e., had marker $t_{w,e} = t_w$. The other active nets are marked with $t_{w,e} > t_w$, which were not incident to u at the time of contraction and thus not removed.

To restore the pin-lists, we iterate over all active nets $e \in I_w$ and if $t_{w,e} = t_w$, we restore

v from the inactive part of the pin-list, see line 7. In Figure 6.1 f) v_4 is restored from the inactive part of e_2. Otherwise, if $t_{w,e} > t_w$, we replace u by v, see line 9. For example v_0 replaces v_1 in e_0 in Figure 6.1 f).

In the sequential setting, contractions are undone in the reverse order in which they were performed, so if v was removed, it is the first entry in the inactive part of e. In this case it suffices to increment the current size of e to restore v in line 7.

In the parallel setting, we perform all uncontractions in the current batch in parallel, so v can be anywhere in the inactive part of e's pin-list. After constructing the batches, we sort each pin-list including the inactive entries by the batches in which the pins are uncontracted (see net e_2 in Figure 6.1 d). Then, all pins of e that have to be restored in the current batch can be activated simultaneously by appropriately incrementing the current size of e (as seen in parts e and f of Figure 6.1). Only one thread that triggers the restore case on a net performs the restore operation, which we ensure with the atomic time-stamping trick from Section 2.2.4.

6.2.4 Implementation Details

To speed up iteration over incident nets, we store a second doubly-linked list where vertices w without active entries in I_w are removed. This improves the iterator's complexity to $\mathcal{O}(\deg(u))$. Since the edit operations that require locks are very fine-grained, we implement locking with spinlocks using atomic test-and-set operations. This applies to all locks used in this chapter.

6.3 Coarsening

In this section, we describe how to implement the coarsening phase for n-level partitioning. The main conceptual tool for this is the forest $\mathcal{F}$ of contractions, represented by the array rep of parent pointers.

Parallelization. A vertex can be contracted as soon as all of its children in $\mathcal{F}$ have been contracted. Different sub-trees and siblings can be contracted in parallel. Hence, we traverse $\mathcal{F}$ in bottom-up fashion. The contracted hypergraph will be the same regardless of the exact execution order. What may differ is the representation in the data structure (which pin was replaced and which was removed). This will have an effect on the intermediate levels later on, but for now we can safely ignore this.

Incremental Construction. We do not know $\mathcal{F}$ in advance, but start with $\text{rep}[v] = v$ for all $v \in V$ and incrementally build it. For finding good contractions to perform, we use the same heavy-edge-rating algorithm as the $\log(n)$-level version (Algorithm 4.2) but construct the ratings directly at vertices instead of their clusters. In the following, we describe how to dynamically extend $\mathcal{F}$ in a thread-safe manner that still enables the bottom-up parallelization.

Conditions. Let (u, v) be a contraction suggested by the heavy-edge-rating function. We have to ensure that $\mathcal{F}$ remains a rooted forest, i.e., v must still be a root ($\text{rep}[v] = v$) and we cannot add cycles. Additionally, u must not have been contracted yet or be in the process of contraction (bottom-up).

This has two consequences: First, if u's contraction has started, we instead contract v onto a suitable ancestor of u. In Algorithm 6.4 we detect cycles and replace u by an ancestor of u that is not contracted yet. This is what we call a *safe ancestor*. In practice u is a safe ancestor in the overwhelming majority of cases, but for correctness the scheme below is still necessary. Secondly, if there are unfinished contractions onto v, we cannot contract v right away. We explicitly allow multiple concurrent contractions onto the same vertex. Therefore Algorithm 6.5 shows how to avoid waiting when there are still pending contractions.

Algorithm 6.4: Find Safe Ancestor

```
   Input: Vertex pair (u, v) to contract
 1 lock(v)
 2 if rep[v] ≠ v
 3 |   unlock(v) and return                          // other thread contracts v
 4 while rep[u] ≠ u and pending[u] = 0 do
 5 |   u ← rep[u]
 6 |   if u = v
 7 |   |   unlock(v) and return                      // cycle in F
 8 if v < u
 9 |   lock(u)
10 else
11 |   unlock(v), lock(u), lock(v)                   // avoid deadlock
12 |   if rep[v] ≠ v
13 |   |   unlock(v), unlock(u) and return          // other thread contracts v
14 if rep[u] = u or pending[u] > 0
15 |   x ← u
16 |   while rep[x] ≠ x do
17 |   |   x ← rep[x]
18 |   |   if x = v
19 |   |   |   unlock(v), unlock(u) and return      // cycle in F
20 |   rep[v] ← u                                    // safe ancestor found
21 |   pending[u] ← pending[u] + 1
22 |   unlock(u), unlock(v)
23 |   return ContractOrTransferResponsibility(u, v)
24 else
25 |   unlock(u) and go to line 4                    // retry with different ancestor
```

Detecting Contraction Starts. We use a zero-initialized array pending, where pending$[x]$ stores the number of vertices y with rep$[y] = x$ whose contraction is not finished. If pending$[x] = 0$, it is safe to contract x. If additionally rep$[x] \neq x$, we assume that the contraction of x onto rep$[x]$ has started. This makes x *unsafe* to contract onto. Note that this does not mean that it has started, just that it is safe to start.

Cycle Detection and Safe Ancestor. We discuss Algorithm 6.4 first. First, v is locked, so that no other thread can write to rep$[v]$ (line 1). If rep$[v] \neq v$, then (u, v) is discarded (line 2), as another thread has already elected to contract v and will run this algorithm. Otherwise, we walk the path towards the root of u's tree in $\mathcal{F}$ by chasing the rep entries to find the lowest ancestor w of u, for which either rep$[w] = w$ or pending$[w] > 0$, i.e., the contraction of w has not started (line 4). If v is found on this path, the contraction is discarded, as it would add a cycle to $\mathcal{F}$ (line 6,7). We replace u with w.

If no cycle is found, then u is locked (line 9) and we check rep$[u]$ and pending$[u]$ again (line 14). If they changed, we release u and keep walking up to find a higher ancestor of u (line 25). Otherwise, u is the desired candidate. We finish the walk up to the root to check for cycles (line 15-19). If no cycles are found, u is a safe ancestor, so we set rep$[v] \leftarrow u$ and increment pending$[u]$ by 1 (line 20, 21). We then unlock v and u and call Algorithm 6.5 to perform the contraction itself (line 22,23).

To avoid deadlocks in Algorithm 6.4, we acquire the lock of the vertex with lower ID first (line 8-13). If $u < v$, we have to intermediately release the lock for v, so we re-check whether rep$[v] = v$ and discard the contraction if this is not the case any longer.

Algorithm 6.5: Contract Or Transfer Responsibility

```
   Input: Vertex pair (u, v) to contract
 1 while u ≠ v do
 2     if pending[v] > 0                                  // without lock
 3         return                                         // transfer responsibility
 4     lock(v)
 5     if pending[v] = 0
 6         unlock(v)
 7         performContraction(u, v)                       // Algorithm 6.2
 8         lock(u), pending[u] −= 1, unlock(u)
 9         v ← u
10         u ← rep[u]
11     else
12         unlock(v) and return                           // transfer responsibility
```

Contract or Transfer. If v has pending contractions $\text{pending}[v] > 0$, we cannot perform the contraction right away. To avoid waiting until $\text{pending}[v] = 0$, we return so we can perform the next iteration of the coarsening pass (the parallel loop in line 3 of Algorithm 6.1). Instead, the thread that reduces $\text{pending}[v]$ to 0 is responsible for contracting $(\text{rep}[v], v)$. If $\text{pending}[v] = 0$ already, we call Algorithm 6.2 to contract (u, v), and subsequently decrement $\text{pending}[u]$ by 1. If this reduces $\text{pending}[u]$ to 0 and $\text{rep}[u] \neq u$, we recursively apply this process to $(\text{rep}[u], u)$.

Identical Net Detection. There is one last detail left for the coarsening phase. For the sake of performance in all components (not just coarsening) it is crucial to detect and remove identical nets. Doing this on-the-fly would introduce additional dependencies for the batches in the uncoarsening phase, which is why we decided against it. Instead, we remove identical nets and nets consisting of a single pin at the synchronization point after a coarsening pass. We adapt the algorithm from Section 4.1.2 to the dynamic data structure.

6.4 Batch-Synchronous Uncoarsening

For the uncoarsening phase, our goal is to create a sequence of batches $\mathcal{B} = \langle B_1, \ldots, B_l \rangle$, where $\mathcal{B}$ is a partition of the contracted vertices into disjoint sets such that $\forall B \in \mathcal{B} : |B| \approx b_{\max}$. Here, $b_{\max}$ is an input parameter that interpolates between parallelism (high values) and uncoarsening granularity. Each batch B_i will be chosen such that we can uncontract the vertices $v \in B_i$ in parallel. Refinement is applied after each batch. Processing B_i will resolve the last dependencies required to uncontract the next batch B_{i+1}. Clearly, the uncontraction of a vertex v can only start once the uncontraction of $\text{rep}[v]$ is finished, i.e., all of its ancestors are uncontracted. Therefore, we construct the batches via a top-down traversal of $\mathcal{F}$.

There is an additional ordering dependency to achieve a correct algorithm that we have not introduced yet. In the sequential version, contractions are undone in the reverse order they were performed. This is not possible in the parallel setting, since it would be inherently sequential and we do not even have a strict contraction order. In the following we show that reverse order is necessary for correctness but only between siblings (vertices contracted onto the same parent). Subsequently, we describe the batch construction algorithm that must take these dependencies into account.

6.4.1 Sibling Dependencies

Consider the scenario in Figure 6.2, with vertices $u, v_1, v_2 \in V$ with $\text{rep}[v_1] = \text{rep}[v_2] = u$, and two nets $e_1, e_2 \in E$ with $v_1, v_2 \in e_i$ but $u \notin e_i$. If the contractions of v_1 and v_2 happen at the same time, and u replaces v_1 in e_1 and v_2 in e_2, then v_2 gets removed in e_1, and v_1 gets removed in e_2 (moved to the inactive part). If we uncontract v_1 in an earlier batch than v_2, then u would be incident to e_2 but not e_1 until v_2 is uncontracted, since we replace u by v_1 in

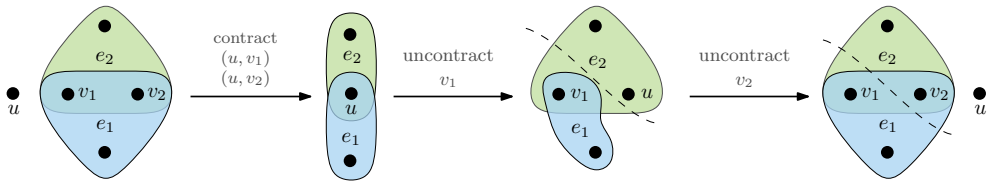

Figure 6.2: Inconsistent state due to concurrent contraction of both v_1 and v_2 onto u, where u replaces v_1 in e_1 and v_2 in e_2. By uncontracting v_1 before v_2, it replaces u in e_1 again, but u should still be incident to e_1, since v_2 is still contracted onto u.

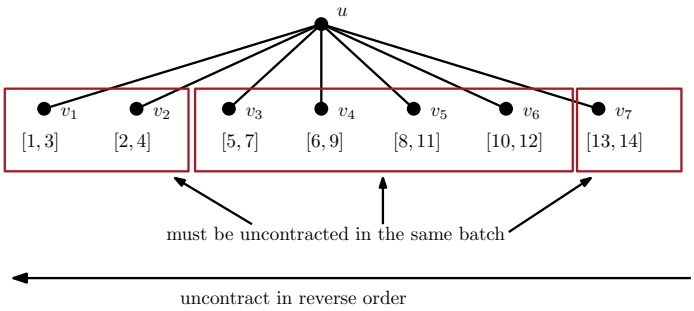

Figure 6.3: Sibling interval counters. Siblings are sorted by the finish time of their contraction, such that we can uncontract them in reverse order. The red boxes show transitive closures of overlapping time intervals.

e_1 but not yet by v_2 in e_2. This is an inconsistent state because v_2 is in e_1 and was contracted onto u, and thus u should be incident to e_1.

The Problem. This is not an issue if we only care about fully uncontracting the hypergraph and projecting the partition. However, we perform refinement, moving vertices after each batch, and doing so in inconsistent intermediate states causes a problem. Assume, we uncontract v_1 from u and then move u and v_1 into different blocks (third part in Figure 6.2). Then e_1 is not in the cut (dashed line) because $u \notin e_1$. When we uncontract v_2 from u we add it to the same block as u and thus inadvertently add e_1 to the cut since now $v_2 \in e_1$ (fourth part). Note that this is just through the uncontraction operation. This violates a fundamental property of the multilevel framework: the projected partition must have the same connectivity and imbalance as that on the previous level. Fortunately these inconsistent states are restricted to siblings, since they are caused by intertwining different roles (replace or remove/restore) of their shared parent in different shared nets.

The Solution. We fix this by enforcing that all siblings that were contracted *at the same time* are reverted in the same batch. A similar argument holds for the case of v_1 being contracted *strictly earlier* than v_2 (no time overlap). Here, v_2 must be uncontracted in an earlier or the same batch as v_1.

To detect time overlaps, we atomically increment a counter at the parent before starting and after finishing a contraction operation. For each contracted vertex v, this yields an interval $[s_v, e_v]$ with start time s_v and end time e_v. If the intervals of two siblings overlap, we assume they were contracted at the same time, otherwise one is strictly earlier than the other. We need to compute the transitive closure of siblings with overlapping intervals, as well as order them decreasingly if one is strictly earlier than the other. Since comparing for equality with interval overlaps is not transitive, we instead sort them in decreasing order of e_v. Then, the siblings in a transitive closure are ordered consecutively and can be found with a rightward sweep from the first interval, by checking whether the next interval overlaps with the union of intervals in the closure so far and extending the union interval if they overlap. An example is shown in Figure 6.3. Here, the siblings are sorted with respect to their contraction finish time, and the red boxes correspond to transitive closures.

Strictly speaking, siblings in a transitive closure need not be reverted in the same batch, but we would have to delay refinement until there are no partially uncontracted transitive closures, which seems more obstructive.

6.4.2 Batch Construction

After the coarsening phase, we need to construct batches of contracted vertices, to uncontract in parallel during the uncoarsening phase. We traverse $\mathcal{F}$ top-down in BFS order, using two FIFO queues: Q for the current layer and Q' for the next layer. Additionally, we have a batch B_{cur} that we are currently adding contracted vertices to, which we append to $\mathcal{B}$ once $|B_{\text{cur}}| \geq b_{\max}$ and then proceed with a new empty batch. An example batch construction is shown in Figure 6.4.

BFS Details. Q and Q' store elements (u, v_i), where u is a vertex and v_i is the i-th child of u in the order sorted by finish time e_v of the contraction operation. For elements in Q, we maintain the invariant that u is uncontracted in a batch before B_{cur}, so that its children can be added to B_{cur}. Furthermore, v_i is the first child of u that has not been added to any batch yet. To initialize Q, we insert all entries (r, w_1), where r is a root of $\mathcal{F}$ and w_1 is the first child of r.

We pop elements from Q until it is empty, and then swap it with Q'. Now, let (u, v_i) be the current element we popped from Q, and let T denote the transitive closure of v_i. For each $v \in T$, we add v to B_{cur}, and push (v, w_1) to Q', where w_1 is the first child of v. Additionally, we push (u, v_j) to the end of Q, where v_j is the first child of u outside T, if any.

This reinsertion aims to minimize the number of contractions with the same parent in a batch, which reduces synchronization overheads during uncontraction operations (list

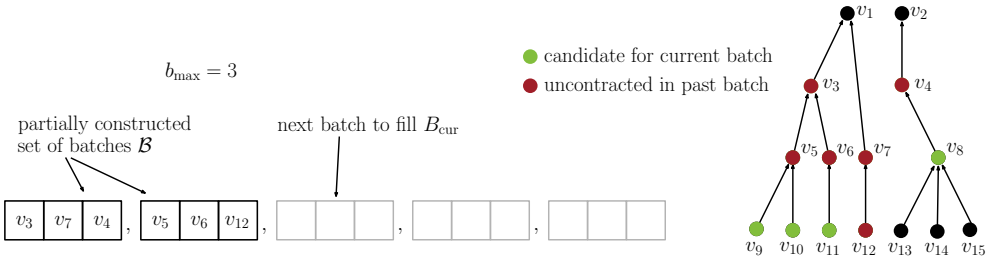

Figure 6.4: Batch Construction via BFS from the roots v_1, v_2 of $\mathcal{F}$. In this situation v_8 will be added to the third batch (currently under construction) since it is at depth 2. The other eligible vertices (v_9, v_{10}, v_{11}) are at depth 3, so only two of them will be added before opening a new batch, since $b_{\max} = 3$.

splicing, vertex weights) and reduces the overlap of local searches. The work of this algorithm is $\mathcal{O}n$.

Parallelization. We traverse the different trees of $\mathcal{F}$ in parallel. $\mathcal{F}$ has as many trees as vertices left in the coarsest hypergraph, so at least around $t \cdot k$, which is more than a common number of cores. We statically assign roots to threads. To approximate a BFS order across trees, we perform one BFS per thread on all of its assigned roots. The threads collaborate on filling batches, and we keep multiple batches open, as threads may progress at different rates.

6.5 Refinement

We perform refinement after uncontracting a batch of vertices. To make this efficient, the refinement should be *localized*, i.e., focus on areas close to the uncontracted vertices first and then gradually expand. We reuse the two refinement algorithms from Section 4.3: label propagation and localized FM.

More Localized Refinement. FM is already localized; we only use vertices from the current batch as seeds instead of all boundary vertices. Similarly, label propagation does not iterate over all vertices but just the vertices in the batch. Still, neighbors of moved vertices are activated, so the search expands to vertices not in the batch.

Global Refinement. We complement the localized refinement with a run of parallel localized FM on the entire hypergraph. We choose the same synchronization point as the one used to restore single-pin and identical nets: after contractions from a full coarsening pass are reverted. This corresponds to the refinement available in traditional multilevel algorithms.

Gain Table Revisited. Recall that FM uses a gain table to look up gains, that is updated when vertices are moved. With the $\log(n)$-level algorithm, we initialize the table from scratch on each level. With $\Omega(n)$ levels this incurs too much overhead. Instead, we update the previous gains based on updates to the $\Phi(e, i)$ values made while uncontracting.

To make this efficient, we employ the same trick as Algorithm 4.4 and Algorithm 4.7 where we do not compute benefits and penalties directly. Instead of $p_i(u)$, we store and update $\overline{p_i}(u) := \omega(\mathrm{I}(u)) - p_i(u)$, such that we can compute the gain as $g_i(u) = b(u) - \omega(\mathrm{I}(u)) + \overline{p_i}(u)$.

Consider a contraction (u, v) that should be reversed. The values $b(v)$ and $\overline{p_i}(v)$ are initialized to zero. The values $b(u)$ and $\overline{p_i}(u)$ are taken from the previous levels. Analogously to the uncontraction operation in Algorithm 6.2.3, on each incident net e of v we distinguish two cases: (i) whether v was replaced by u in e, or (ii) whether u and v were both incident to e before the contraction; confer line 6.

Case 1: Replace. If v was replaced by u during coarsening, then u is now being replaced by v. Hence, we subtract $\omega(e)$ from $\overline{p_i}(u)$ and add it to $\overline{p_i}(v)$ for all $i \in \Lambda(e)$. Recall that $\Lambda(e) = \{i \mid \Phi(e, i) \geq 1\}$. If $\Phi(e, \Pi(u)) = 1$, we additionally subtract $\omega(e)$ from $b(u)$ and add it to $b(v)$. In this case, $\Phi(e, \Pi(u))$ does not change, as v is now a pin of e, but u no longer is.

Case 2: Restore. In the second case, where u and v are both incident to e after the uncontraction, we increase $\Phi(e, \Pi(u))$ by one since v is assigned to $\Pi(u)$ and now both u and v are pins of e in $\Pi(u)$. If $\Phi(e, \Pi(u)) = 2$, we have to reduce $b(u)$ by $\omega(e)$, since moving u out of $\Pi(u)$ would no longer remove $\Pi(u)$ from $\Lambda(e)$. Regardless of $\Phi(e, \Pi(u))$ we add $\omega(e)$ to each $\overline{p_i}(v)$ for $i \in \Lambda(e)$.

Algorithm 6.6 shows the approach again in pseudocode, with the following parallelization included. As in Section 6.2.3, we distinguish the two cases based on the timestamp in the incident net arrays. There is no need to search in the pin-list which case applies, as is done for sequential KaHyPar [Sch20].

Parallelization. While updates to v are done exclusively by one thread, updates to u can happen from multiple threads. Therefore, we use atomic fetch-and-add instructions for updates to u.

$\Lambda(e)$ does not change during uncontractions, since $\Phi(e, i)$ is only modified in the second case, where $\Phi(e, i) \geq 1$. Therefore, iteration over $\Lambda(e)$ is thread-safe. Modification and reads on $\Phi(e, \Pi(u))$ are thread-safe because they are protected by a net-specific lock. Updates to the gain table are done outside locks, but for simplicity the benefit updates are displayed inside here. The lock in the replace case is necessary to do the replacement itself, though the slot can be found outside the lock. This is important for the following issue.

Uncoarsening in parallel introduces one more intricacy. In the restore case, u might have been replaced by some $v' \neq v$ due to a concurrent uncontraction (u, v'). Therefore if $\Phi(e, \Pi(u)) = 2$, we search for a pin v' in the active part of e with $\Pi(v') = \Pi(u)$ and $v' \neq v$, and then subtract from $b(v')$ instead of $b(u)$. If u was not replaced, we simply find $v' = u$.

Algorithm 6.6: Update Gain Table for Uncontraction

```
1  for e ∈ I(v) do
2      if v replaces u // case 1: replace
3          find u's slot in e
4          lock(e)
5          replace u with v at the found slot
6          if Φ(e, Π(u)) = 1
7              b(u) −=_atomic ω(e)
8              b(v) += ω(e)
9          unlock(e)
10         for i ∈ Λ(e) do
11             p_i(u) −=_atomic ω(e)
12             p_i(v) += ω(e)
13     else // case 2: restore
14         lock(e)
15         Φ(e, Π(u)) += 1
16         if Φ(e, Π(u)) = 2
17             for v′ ∈ e do // subtract from potential replacement of u
18                 if v′ ≠ v and Π(v′) = Π(u)
19                     b(v′) −=_atomic ω(e)
20                     break
21         unlock(e)
22         for i ∈ Λ(e) do
23             p_i(v) += ω(e)
```

For correctness it is necessary to perform the scan forward from the beginning of the active part. The reasoning about correctness is very delicate, in fact we were lucky that this was not a bug. The current batch may contain further vertices contracted onto u that are restored. These are assigned to $\Pi(u)$ as well, but are the wrong vertex to take $\omega(e)$ off the benefit. Fortunately, these were moved behind u or its potential replacement v' during coarsening, into the inactive part at the time. Thus, we exploit the fact that the sorting algorithm, used to sort the pin-lists by batch, is stable, in order to find u or v' first. Note that there are no pins of e in $\Pi(u)$ that were not contracted onto u, since $\Phi(e, \Pi(u))$ was 1 prior to this. Furthermore, the scan happens only for the first restore case; subsequent ones have $\Phi(e, \Pi(u)) > 2$.

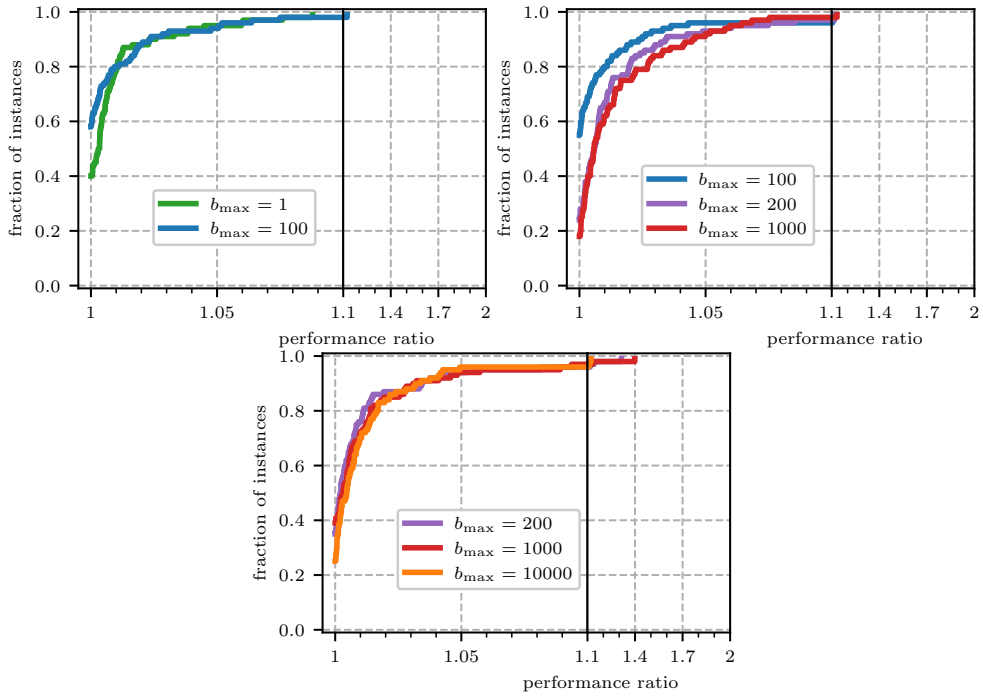

Figure 6.5: Impact of the batch size b_{max} on partition quality.

6.6 Experimental Evaluation

We integrated our parallel *n*-level approach into the Mt-KaHyPar framework. Our new algorithm is called Mt-KaHyPar-Q (for quality preset). While the speedup experiments are performed on machine C (they took six weeks which was unattainable for a repetition), all other experiments on set B are performed on machine B in this section.

In the following, we first determine an appropriate value for the batch size parameter b_{max}. Subsequently, we conduct speedup experiments, look at running time shares of the multilevel phases, and analyze the refinement algorithm's behavior. We follow up with the familiar horse-race comparisons against prior works. Since Mt-KaHyPar-Q is primarily targeted at medium size instances, we present experiments on set A before set B.

6.6.1 Batch Size Configuration

In Figure 6.5 we compare the solution quality when varying the batch size parameter b_{max}. For this experiment we used machine A with 20 cores on the second parameter tuning benchmark set D. We tested the different values 1 (which corresponds to sequential uncoarsening), 100,

Table 6.1: Geometric mean speedups for each phase, reported separately for 16 and 64 threads, and all instances or instances that took at least 100 seconds sequentially. The last row shows the percentage of instances that took at least 100 seconds sequentially.

instances	threads	total	coarsen	initial	uncoarsen	LP	FM
all	16	11.79	11.04	10.21	11.01	8.04	11.22
all	64	23.41	25.05	18.16	22.9	15.91	19
$\geq 100s$	16	12.33	12.69	11.15	11.79	10.73	12.04
$\geq 100s$	64	25.52	35.39	18.53	24.84	25.94	21.8
% $\geq 100s$		71.8	30.8	7.8	33.8	18.5	39.3

200, 1000, and 10000. Comparing $b_{\max} = 1$ and $b_{\max} = 100$ shows a marginal advantage for $b_{\max} = 100$. Using $b_{\max} = 100$ is slightly better than $b_{\max} = 200$. Finally, there is no visible difference between $b_{\max} = 200$, $b_{\max} = 1000$ and $b_{\max} = 10000$. The fastest configuration is $b_{\max} = 1000$, which is 1.64 times faster (geometric mean) than $b_{\max} = 100$ and 13.06 times faster than $b_{\max} = 1$. We therefore choose $b_{\max} = 1000$ for all further experiments.

6.6.2 Scalability

For the speedup experiment we used a subset of set B containing 77 out of the 94 hypergraphs. These were selected such that Mt-KaHyPar-Q on 64 cores finishes in under 800 seconds. The speedup experiment still took 6 weeks to complete on machine C.

In Figure 6.6 we summarize self-relative speedups for the algorithmic components of Mt-KaHyPar-Q and the whole framework (top left), with varying number of threads 4, 16, 64. For initial partitioning, we removed 4 outliers below 1, in order to avoid skewing the plot further. These all stem from the same instance `circuit5M.mtx`, where non-determinism during coarsening leads to a larger initial partitioning instance. Additionally, Table 6.1 shows geometric mean speedups for all instances, and such where the particular phase took at least 100 seconds.

The overall geometric mean speedup of Mt-KaHyPar-Q is 3.68 on 4 cores, 11.79 on 16 cores and 23.41 on 64 cores. If we only consider instances with a single-threaded running time $\geq$ 100s, we achieve a geometric mean speed up of 25.52 on 64 cores.

The scaling behavior of coarsening, uncoarsening and localized FM are crucial to the overall scalability of our framework, since they account for the majority of the work performed. Coarsening and uncoarsening both have similar speedups with 11.04 and 11.01 for 16 cores, whereas coarsening performs significantly better with 64 cores (25.05 vs 22.9).

Both localized refinement algorithms yield reliable speedups. However, since we run label propagation before FM, it scales slightly better for 64 cores, as it could potentially remove boundary vertices from the cut, which then reduces the available parallelism for FM. Initial partitioning shows the least promising speedups out of all components, but is substantially

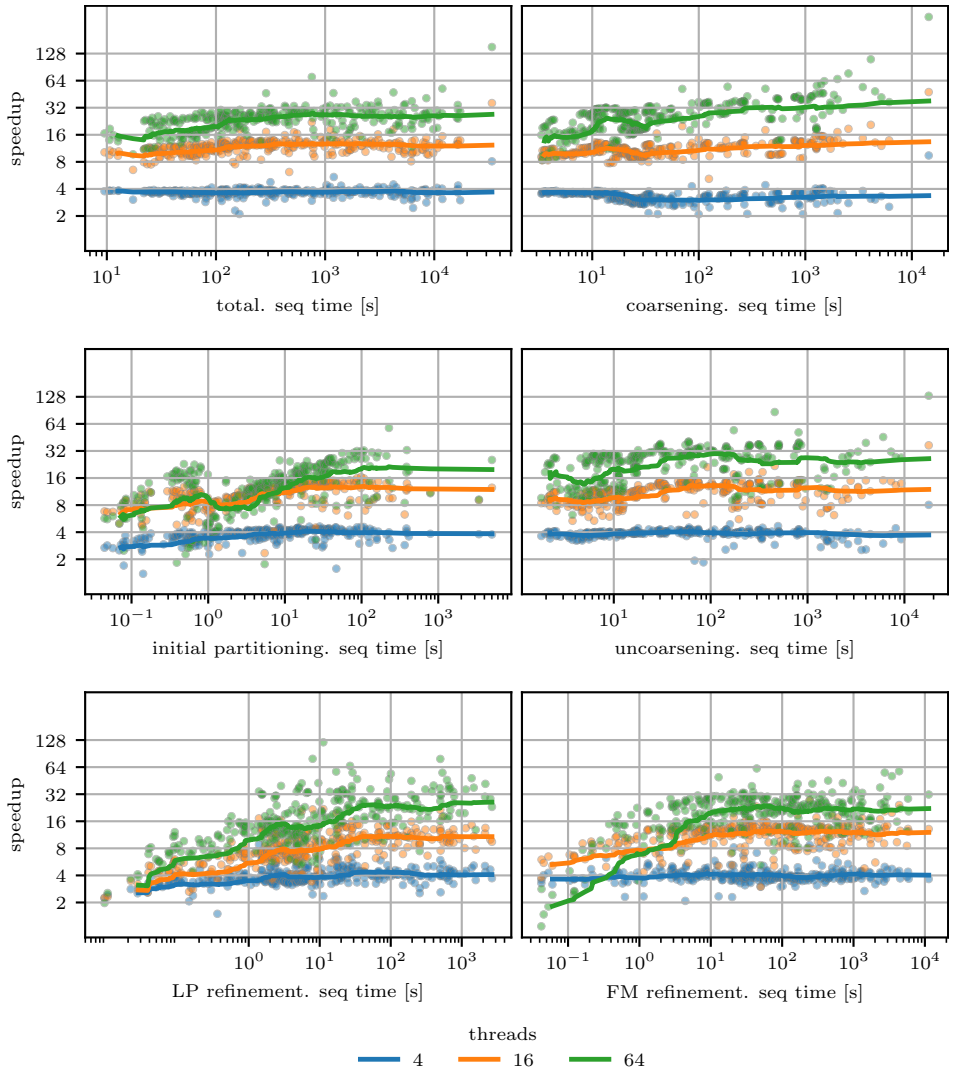

Figure 6.6: Relative speedups of Mt-KaHyPar-Q in total, as well as its components shown separately. The x-axis shows the sequential time in seconds, whereas the y-axis shows the speedup. The lines are rolling geometric means with window size 50 of the per-instance speedups (scatter).

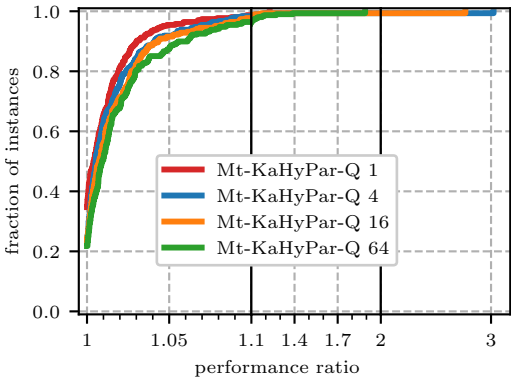

Figure 6.7: Partition quality of Mt-KaHyPar-Q with increasing number of threads on set B.

faster, running in less than 100 seconds on 92.2% of the instances (see last row of Table 6.1).

On some instances we obtain super-linear speedups. This is not surprising since the algorithms are non-deterministic. Among other things, we have observed refinement converging in fewer rounds. The extreme outliers, however, are caused by coarsening and uncoarsening (up to 412), where non-deterministic contraction decisions cause highly varying running times. The coarsening slows down once only few vertices are left.

Figure 6.7 shows that increasing the number of threads does adversely affect the solution quality, but only by a fairly small amount. We observed that the localized searches interfere with one another, which may reasonably explain this behavior. A possible remedy would be to further diversify the uncontraction batches, an avenue we pursue later in Section 6.7.4.

6.6.3 Refinement Statistics

This phenomenon is further explained by the statistics in Figure 6.8, where we show how often calculated gains were wrong and how often moves had to be reverted due to accidental balance violations. Looking back at Figure 4.6 we see that for n-level the calculated gains are wrong more often than for Mt-KaHyPar-D. The majority is still around 0% but there are a lot more outliers in the 5-10% range. This is caused by interference between threads, as they operate in parallel on the same areas, whereas in Mt-KaHyPar-D the randomization spreads out the areas of the hypergraph where vertices are moved. For label propagation, fewer reverts are caused by accidental balance violations, which is explained by the fact that fewer moves are performed between synchronization points (end of round) than in Mt-KaHyPar-D. However, localized FM reverts a much larger fraction of moves during global rollbacks: the median for Mt-KaHyPar-D was below 5%, whereas for Mt-KaHyPar-Q it is around 30%. We also see larger gain fluctuations.

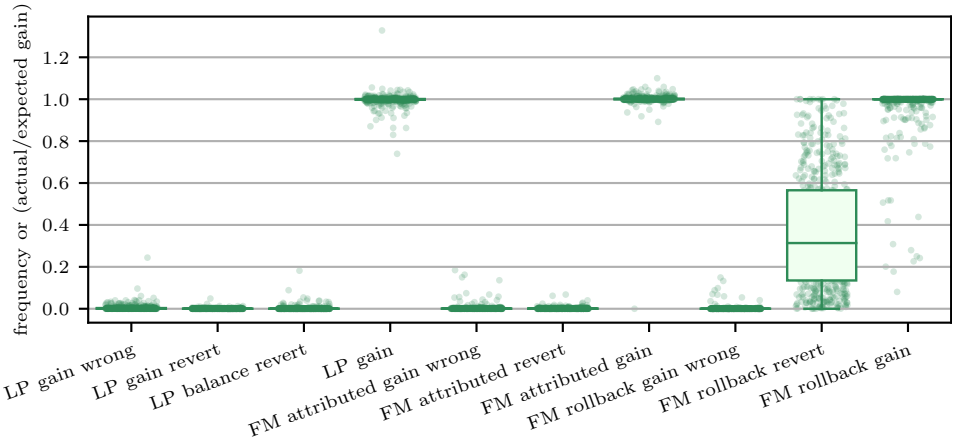

Figure 6.8: Intervention frequency of gain accuracy techniques, and overall gain fluctuation for Mt-KaHyPar-Q on set B.

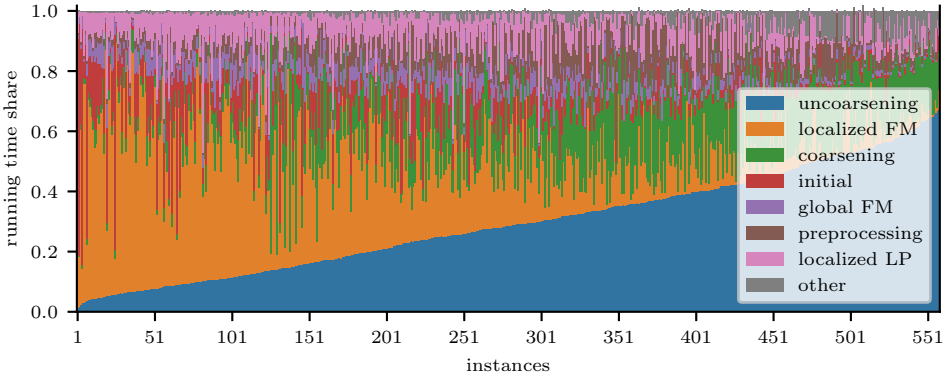

Figure 6.9: Fractions of time spent in each of the components of Mt-KaHyPar-Q plotted per instance of set B.

6.6.4 Running Time Shares

Next we look at Figure 6.9 to see which phases of the framework take up the most time. For each phase we plot the fraction of the total running time. The instances are sorted by the fraction for uncoarsening, which is generally the slowest phase, together with localized FM. Combined they make up around 60% of the total time. Adding 5-10% for localized LP and 5-10% for global FM (run at synchronization points for restoring identical nets), the refinement phase makes up around 70-80% of the total time. On around half of the instances

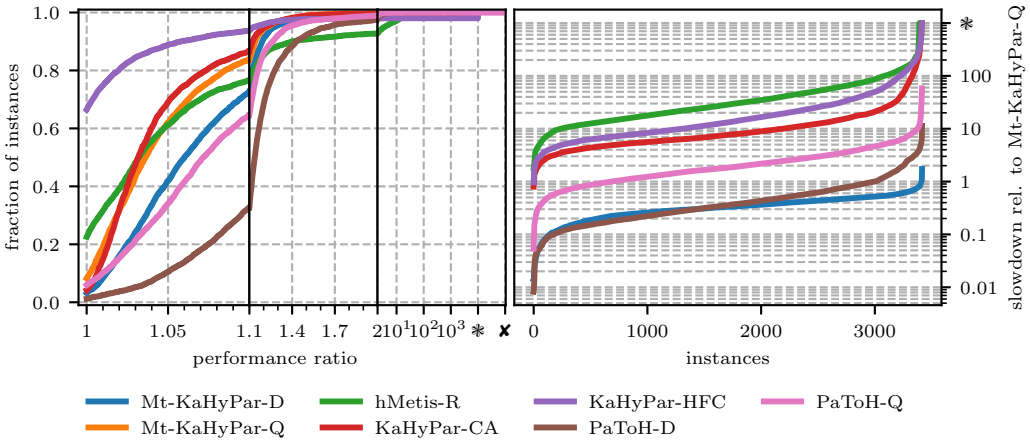

Figure 6.10: Partition quality and running time of Mt-KaHyPar-Q and competitors on set A.

coarsening is negligible, on the remaining half it takes between 10-20%. Initial partitioning is negligible for the most part, though there are some outlier instances.

6.6.5 Comparison with Sequential Algorithms

Now we turn our attention to comparisons with existing state-of-the-art hypergraph partitioning algorithms. As Mt-KaHyPar-Q primarily targets medium-size instances we start with benchmark set A. Figure 6.10 displays the quality and running time results for the full pool of algorithms and Figure 6.11 shows some direct comparisons.

From Figure 6.10 (right) we see that Mt-KaHyPar-Q on 10 cores is moderately faster than PaToH-Q and significantly faster than hMetis and the sequential KaHyPar variants: in particular around a factor of 10 compared to KaHyPar-CA. The geometric mean running times are 3.13s for Mt-KaHyPar-Q (10 cores), 28.14s for KaHyPar-CA, 48.95s for KaHyPar-HFC, 93.2s for hMetis-R, 1.17s for PaToH-D, 5.86s for PaToH-Q, and 0.96s for Mt-KaHyPar-D (10 cores). PaToH-D and Mt-KaHyPar-D are substantially faster, but compute partitions with worse quality.

Mt-KaHyPar-Q finds partitions of similar quality as KaHyPar-CA and hMetis-R, and is only beaten by KaHyPar-HFC due to its flow-based refinement. Considering the entire algorithms pool, hMetis-R contributes more of the best partitions, but converges slower towards 1 than the KaHyPar variants.

Looking at the direct comparison in Figure 6.11 (bottom right), we see that Mt-KaHyPar-Q is slightly better than hMetis-R and contributes more of the best partitions. It reaches the 1.1 factor at 90% of the instances, whereas hMetis-R only does so at 80%. Mt-KaHyPar-D is moderately worse: at 50% it reaches factor 1.02, at 80% it reaches 1.04, and at 95% it reaches

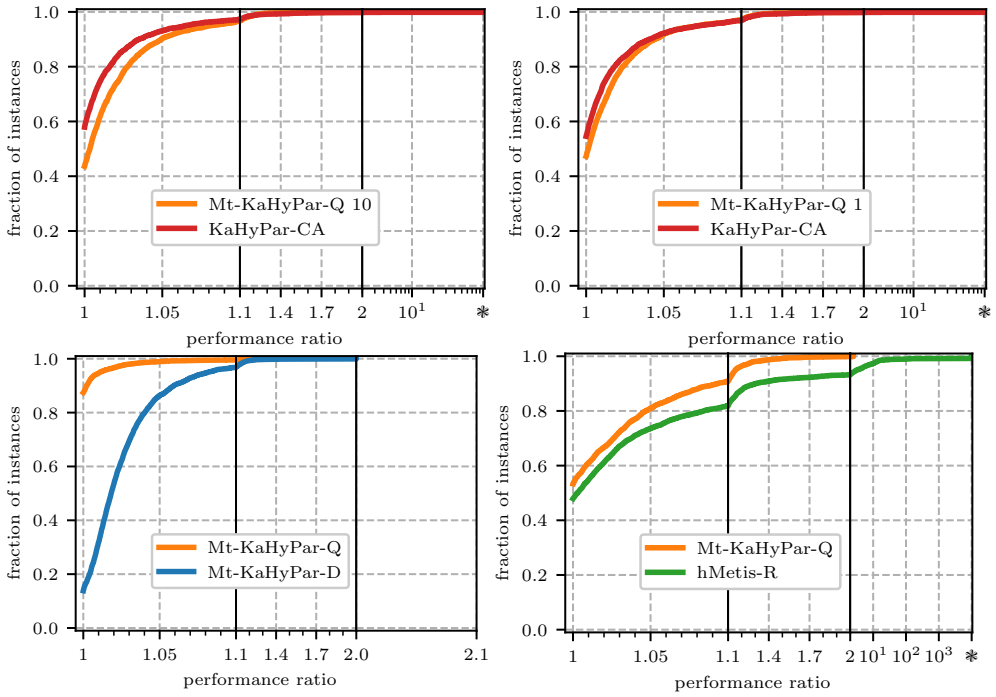

Figure 6.11: Direct comparisons of Mt-KaHyPar-Q with competitors of similar quality. Top row KaHyPar-CA versus Mt-KaHyPar-Q with 10 cores (left) and 1 core (right). Bottom row: Mt-KaHyPar-D (left) and hMetis-R(right).

1.1 (bottom left). Comparing Mt-KaHyPar-Q and KaHyPar-CA directly (top row), we see that KaHyPar-CA is still barely better if 10 cores are used, and this diminishes if only 1 core is used.

Due to varying running times, we present effectiveness tests in Figure 6.12. Here, Mt-KaHyPar-Q beats hMetis-R and KaHyPar-CA convincingly, thanks to the parallelism. The plot versus hMetis-R looks similar on 1 core (not shown), the one versus KaHyPar-CA does not (same quality). Furthermore, Mt-KaHyPar-Q almost reaches the quality of KaHyPar-HFC and is almost reached by Mt-KaHyPar-D. The effectiveness tests comparing default and *n*-level Mt-KaHyPar on set B look similar, but slightly better for Mt-KaHyPar-D.

6.6.6 Comparison with Parallel Algorithms

Next we look at the comparison with fast and parallel algorithms on set B, using 64 cores for the parallel algorithms. Figure 6.13 shows the results. Mt-KaHyPar-Q contributes the best solutions on around 70% of the instances, followed by Mt-KaHyPar-D and PaToH-D

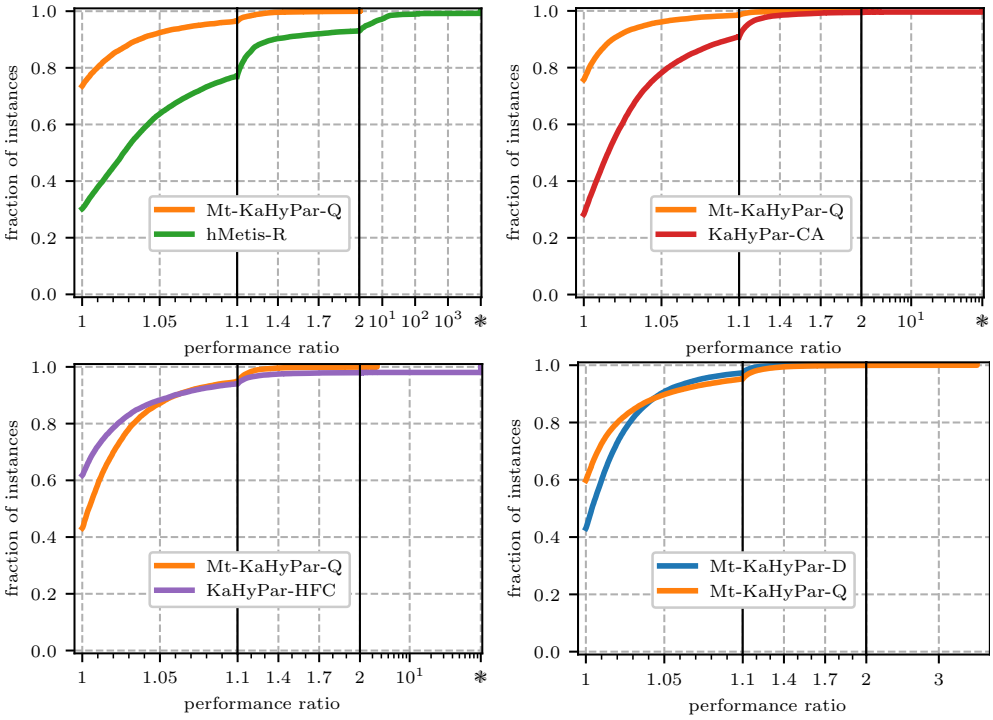

Figure 6.12: Effectiveness tests with virtual instances on benchmark set A, comparing Mt-KaHyPar-Q (on 10 cores) with KaHyPar-HFC, KaHyPar-CA and hMetis-R and Mt-KaHyPar-D (on 10 cores).

on around 17%. It reaches the 1.1 factor at 90% of the instances, as does Mt-KaHyPar-D. In the first bucket range $[1.0, 1.1]$ Mt-KaHyPar-D stays below Mt-KaHyPar-Q. At the median marker for the instances, it is off by about 2%. As shown previously, BiPart and Zoltan are far off in terms of partition quality, whereas PaToH-D is situated in the midfield.

Concerning running times, Mt-KaHyPar-Q is slightly faster than BiPart, but slower than Zoltan and much slower than Mt-KaHyPar-D. The geometric mean running times are 23.96s for Mt-KaHyPar-Q, 29.15s for BiPart, 10.64s for Zoltan, 3.9s for Mt-KaHyPar-D and 47.63s for PaToH-D.

6.7 Asynchronous Uncoarsening

Batch-synchronous uncoarsening performs remarkably well, considering the $\Theta(n/b_{\max})$ synchronization points. Through writing the paper we gained some further intuition how

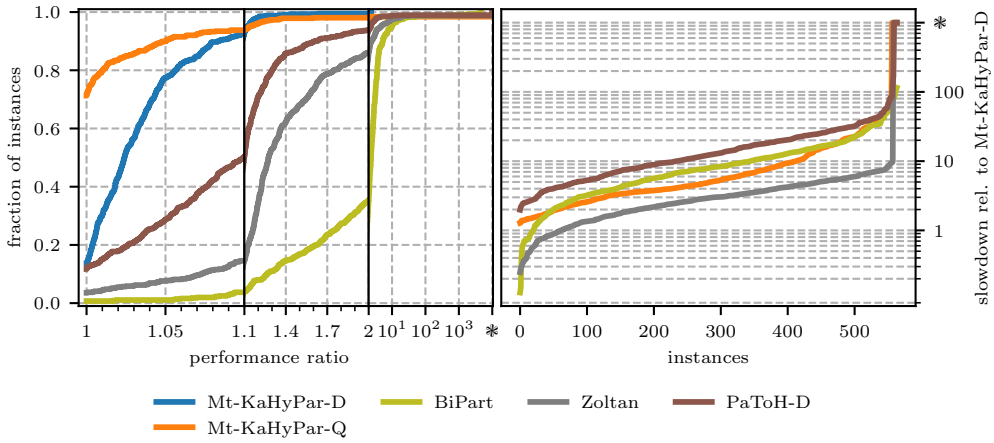

Figure 6.13: Partition quality and running time of Mt-KaHyPar-Q and competitors on set B.

parallel n-level should work, which led us to believe that completely asynchronous uncoarsening is feasible and potentially practical. The goal is to perform uncontractions and refinement at the same time, without any synchronization for batches. Localized refinement should be run on the parent and child right after the uncontraction. Therefore this approach is truly n-level, not $n/b_{\max}$ levels.

This section is based on the Master thesis of Moritz Laupichler [Lau21], which was supervised jointly with Tobias Heuer. In many places it contains a substantial amount of additional details (for example search space diversification), and we only show a subset of the experiments.

6.7.1 Asynchronous Uncoarsening Scheme

We still have to abide by the same uncoarsening rules as before: parents before children and siblings in reverse order; or together if they were contracted at the same time. Hence, the algorithm works on *uncontraction groups*, which we define as the transitive closure of siblings that were contracted at the same time. We traverse $\mathcal{F}$ top-down in BFS order, by using a priority queue with the depth in $\mathcal{F}$ as the key for the groups. A group is *eligible* when its parent is uncontracted and prior sibling uncontractions are finished. Once an uncontraction group is finished, we mark the next sibling uncontraction of the parent as eligible, as well as the first group of each child (where it acts as parent), and insert them into the queue. Initially the first groups of the roots of $\mathcal{F}$ are eligible.

The threads keep polling groups from the queue until all are uncontracted. For a group, we perform all uncontractions first, before running localized refinement around the boundary vertices among the children and parent. To prevent vertices that participate in an uncontrac-

tion from being moved and vice versa, we use vertex-specific locks that are shared between the local search and uncontraction parts of the code. For each move, the vertex must be locked, and for each uncontraction group the parent must be locked. The children need not be locked, since we delay their *activation* until all children of the group are uncontracted. This means the local search routines do not consider them as part of the hypergraph yet.

We only use *try-lock* operations. If the try-lock on the parent of the top-priority uncontraction group fails, we reinsert it and poll a new uncontraction group. The priority queue is randomized, such that we likely get a different entry with each poll.

6.7.2 Asynchronous Localized Refinement

For refinement, we run label propagation first such that it is responsible for finding the easy moves. We only hold the vertex-lock for the duration of the move and again only use try-lock operations. If the try-lock fails, we store the move in a vector, to try again in a second pass, but still do not insist. If the lock fails a second time, we discard the move.

Subsequently we run localized FM on the same seeds. Here, we must hold on to the locks of all vertices in the PQs and those already moved, releasing them only once the search ends. We do not perform a global rollback with gain recalculation, since the asynchronous uncoarsening lacks the concept of a round. Instead, we rely on attributed gains when applying moves to the global partition in the local rollback step.

6.7.3 Gain Table

For the same reason, the trick with recomputing benefits of moved vertices at the end of a round no longer works. Hence, we use the gain table version with k benefit values as outlined in Section 4.3.5. This version supports correct updates through any set of moves.

Uncontractions. A major simplifying factor for updating the gain table through uncontractions in Section 6.5 was that $\Lambda(e)$ does not change during uncontractions. This is no longer the case with concurrent moves. Instead of performing the updates while holding the net-lock, we create a snapshot of $\Lambda(e)$ and then perform penalty updates outside the lock. Furthermore, the added benefit values require more updates, whereas the penalty updates stay the same.

Let (u, v) be a contraction to revert, and let $e \in \mathrm{I}(v)$. There are again the two cases where v replaces u in e, or v is restored in e. If v replaces u, we now need to shift $\omega(e)$ from all $b_i(u)$ to $b_i(v)$ for $i \in \Lambda(e)$ with $\Phi(e, i) = 1$. If v is restored, we add $\omega(e)$ benefit to all $b_i(v)$ with $\Phi(e, i) = 1$. In this case we need to additionally subtract $\omega(e)$ from $b_{\Pi(u)}(w)$ for all $w \in e \setminus v$ if $\Phi(e, \Pi(u)) = 2$. This is the expensive new part; for an uncontraction the synchronous version never required a gain update to all pins of a net.

Moves. The gain updates due to moves are as described in Section 4.3.5 for the variant with k benefits. Unfortunately, the updates may be applied to the wrong vertices since concurrent

uncontractions may edit the pin-list. This affects updates due to uncontractions equally, since we have a new case where we need to scan the pin-list in the restore case.

To guarantee correctness, the gain table updates would have to be performed inside the net-lock, which we cannot afford for large nets. A potential remedy is to create a snapshot of the pin-list inside the lock, and then use the snapshot to perform the updates outside. Asymptotically the snapshot creation is just as expensive as the gain update, but the latter does random memory access (with atomic instructions that need to flush write buffers no less), whereas the former only does a scan. If the net is small ($|e| < \eta = 1000$), we simply perform the gain update inside the lock. Otherwise, we create a snapshot but drastically reduce the number of pins we have to snapshot with the following optimization.

Optimize Snapshots. A pin is called *stable* if it will not be replaced in future uncontractions. Initially, only leaves of $\mathcal{F}$ are stable. For simplicity, we make a vertex stable in all of its incident nets, once all uncontractions from it are done.

We partition the pin-lists into two parts: stable pins first, then unstable pins. We only need to snapshot unstable pins inside the lock, the stable pins can be scanned outside the lock. Once a pin becomes stable we swap it to the boundary and increment the start pointer for unstable pins. A further optimization is to consider that we have a synchronization point for restoring identical nets. Parents with all uncontractions after this point are considered stable as well. Hence, the number of unstable pins is fairly small.

6.7.4 Explicit Search Space Diversification

While the batch-synchronous version already exhibited some interference between FM searches, this problem is exacerbated for the asynchronous version since uncontractions and label propagation happen concurrently to FM searches. We attempt to remedy this by restricting in which *regions* of the hypergraph we perform uncontractions and thus subsequent localized refinement. More precisely, we ensure that we do not uncontract a group whose parent shares *any* (!) net with a parent of an uncontraction on a different thread. The nets stay blocked until the associated localized refinement is done. However, the blocked region is not expanded as the localized search expands; only the parent's nets are blocked. Yet, this still counteracts interference from uncontractions. The implementation uses a bound on the Jaccard similarity, but preliminary experiments showed that forcing complete dissimilarity works best, even though this seems fairly aggressive. If after a number of tries, there is still no uncontraction group with a disjoint region available, we pick the least similar out of those tried. This approach is also applicable to the batch-synchrous variant, but the implementation needs more work to make it truly viable.

6.7.5 Experimental Evaluation

We omit the usual speedup plot and instead report the harmonic mean (!) speedups taken from the Master thesis. With 64 cores of machine C, a 21.81 overall speedup is achieved,

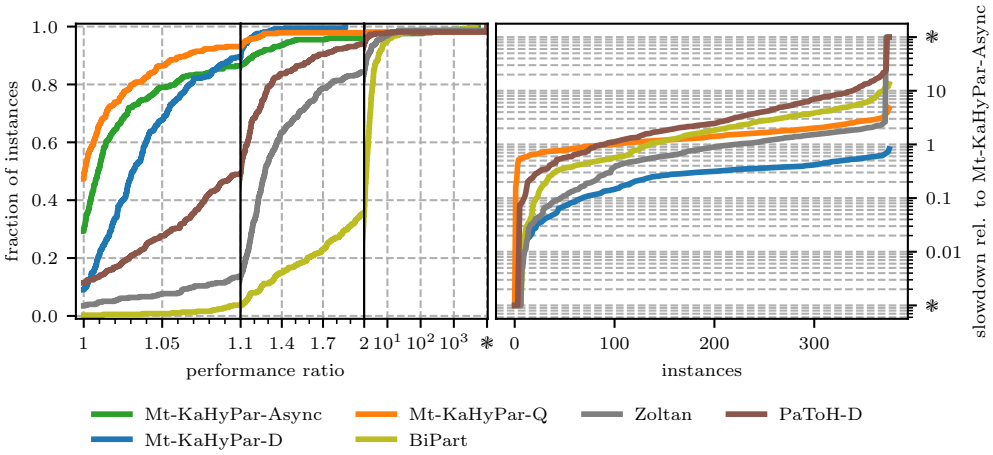

Figure 6.14: Partition quality and running time of the asynchronous n-level uncoarsening variant Mt-KaHyPar-Async and competitors on set B.

which improves to 29.14 for instances that take longer than 100s on 1 core, and to 32.32 for instances that take longer than 1000s. For uncoarsening plus refinement the speedups are 21.6 overall, 32.51 for $\geq$ 100s, and 39.46 for $\geq$ 1000s. This last aggregate still includes 23% of the instances. Again, there are some super-linear speedups due to non-determinism.

Looking at partition quality in Figure 6.14 (left), we see that the asynchronous version does come with a small quality penalty and is situated between Mt-KaHyPar-D and Mt-KaHyPar-Q in the first range $[1.0, 1.1]$. In the second range $[1.1, 2]$ it stays below Mt-KaHyPar-D, indicating that robustness can be improved. The geometric mean running times are 22.66s for asynchronous Mt-KaHyPar-Q and 30.7s for batch-synchronous Mt-KaHyPar-Q.

6.8 Conclusion

In this chapter, we demonstrated that the powerful n-level scheme can be parallelized very efficiently on shared-memory architectures, without significant sacrifices in solution quality. Our new system outperforms most existing partitioners in terms of solution quality and achieves good speedups over its sequential counterpart. From a theoretical perspective, we introduced the contraction forest – formed by any valid sequence of contractions – as a useful concept to maintain consistent states while performing concurrent contractions. Further, we presented a decomposition of this forest into batches – each containing a fixed number of vertices – which can be uncontracted in parallel and enables highly localized searches in a very fine-grained hierarchy. Moreover, we showed how to lift the restriction to batches and instead perform uncoarsening completely asynchronously, at the same time as

refinement. This leads to better speedups due to fewer synchronization points, but sacrifices some solution quality due to increased interference.

7 Flow-Based Hypergraph Partitioning

In this chapter, we investigate maximum flows as a tool for partitioning heuristics. In contrast to local vertex moving algorithms they were only scarcely used due to their complexity. This chapter is based on three papers [GHW19a, GHSW20, GHS22], published jointly with Michael Hamann and Dorothea Wagner; MH, DW and Sebastian Schlag; as well as Tobias Heuer and Peter Sanders. The first paper [GHW19a] deals with porting an existing flow-based bipartitioning algorithm FlowCutter [HS18a] from graphs to hypergraphs. As it is too slow, we investigate methods to accelerate FlowCutter, which are based on using bigger terminal sets. An extreme version of this ultimately corresponds to an extension of an existing flow-based refinement algorithm [SS11]. The advantage of our approach is that the solution space is more effectively explored, which leads to smaller cuts. This first paper is restricted to bipartitioning and unweighted instances. FlowCutter itself is an accidental re-invention of the FBB algorithm by Yang and Wong [YW96], who originally proposed it on hypergraphs, computing flows via the Lawler network. Our contribution over their work is the faster flow computation directly on the hypergraph, and using it as a refinement algorithm. We furthermore contribute several extensions such as handling disconnected inputs, distance-based piercing (refinement only), and keeping the assignment of vertices only incident to cut nets flexible. In the second paper [GHSW20] we integrate flow-based refinement with FlowCutter into the existing flow-based refinement scheme of KaHyPar [HSS19], now supporting weighted inputs and k-way partitioning via scheduling. The advantage of our approach is faster speed due to the thoroughly engineered flow algorithm and better partition quality due to FlowCutter. In the third paper [GHS22] we parallelize the approach from the second paper, by employing a parallel push-relabel algorithm to compute maximum flows, and parallelizing the scheduling of different block pairs.

Attributions. The source code for the first two papers was written by me. The papers were largely written by me, with editing by the co-authors. For the third paper, the parallel flow algorithm was implemented by me, whereas the parallel scheduling was implemented by Tobias Heuer. Both contributed equally to the write-up.

Overview. This chapter is split into two parts: sequential FlowCutter refinement and its parallelization. After the first part on sequential refinement, we present the experimental evaluations from the first two papers [GHW19a, GHSW20] with the first (Section 7.7) restricted to 2-way partitions but providing more internal details, whereas the second (Section 7.8) focuses on horse-race comparisons with more algorithms.

We discuss a lot of related work in the first part, in order to supply specific details close to where we use them. This is the reason we kept their discussion brief in Chapter 3. Sections 7.1 and 7.2 are largely related work, presenting FlowCutter and flow-based refinement in a way that sets up the rest of the chapter nicely. The novelties proposed by us in these sections are ensemble terminals, using FlowCutter as a refinement algorithm, and distance-based piercing. In Section 7.3 we show how to implement Dinitz' maximum flow algorithm [Din70] directly on the hypergraph, and discuss the running time benefits of this approach. In Section 7.4, we show how to run FlowCutter on disconnected hypergraphs. After some minor points and the first two sets of experiments, we continue with the parallel flow algorithm in Section 7.9, and parallel scheduling in Section 7.12. These components are evaluated separately: in Section 7.11 on computing flows and 2-way refinement and Section 7.13 with the main horse-race comparison and refinement analysis.

7.1 FlowCutter

Algorithm 7.1: FlowCutter Core Routine

```
1  S ← {s}, T ← {t}
2  while true do
3  |   augment flow to maximality regarding S, T
4  |   derive source- and sink-side cut S_r, T_r ⊂ V
5  |   if (S_r, V \ S_r) or (V \ T_r, T_r) balanced
6  |   |   return balanced partition
7  |   if c(S_r) ≤ c(T_r)
8  |   |   S ← S_r ∪ selectPiercingNode()
9  |   else
10 |   |   T ← T_r ∪ selectPiercingNode()
```

FlowCutter solves a sequence of incremental maximum flow problems until a balanced bipartition is found. Algorithm 7.1 shows pseudocode for the approach. We start with an

initial set of sources and sinks (traditionally consisting of one node each), and an initial flow assignment of zero. In each iteration, first the previous flow is augmented to a maximum flow regarding the current source set S and sink set T. Subsequently, the node sets $S_r, T_r \subset V$ of the source- and sink-side cuts are derived. This is done via residual (parallel) BFS (forward from S for S_r, backward from T for T_r). The node sets induce two bipartitions $(S_r, V \setminus S_r)$ and $(V \setminus T_r, T_r)$. If neither is balanced, all nodes on the side with smaller weight are transformed to a source if $c(S_r) \leq c(T_r)$ or a sink otherwise. To find a different cut in the next iteration, one additional node is added, called the *piercing node*. Thus, the bipartitions contributed by the currently smaller side will be more balanced in future iterations. Since the smaller side is grown, this process will converge to a balanced partition.

The running time of FlowCutter is $\mathcal{O}(\zeta m)$, where ζ is the final cut weight. This makes it well-suited for graphs with tiny cuts such as road networks. The bound stems from a pessimistic implementation that augments one flow unit in $\mathcal{O}(m)$ work with the Edmonds-Karp algorithm [HS18a, YW96]. Deriving cut-sides also takes $\mathcal{O}(m)$ work per bipartition. The time bound carries over to a hypergraph implementation with the Lawler network, leading to $\mathcal{O}(\zeta p)$ running time.

FlowCutter leaves three design decisions to the implementing engineer: the flow algorithm, the heuristic to select piercing nodes, and the choice of initial terminals s, t, of which only the latter two have an impact on partition quality, whereas the first has the largest impact on running time. The piercing heuristic is arguably the most important piece for partition quality, since a bad choice can make the rest of the run useless. The terminals are also important for both quality and running time. If the final cut weight ζ is not small, and there are many incremental cuts whose weights differ only slightly, FlowCutter will be excruciatingly slow. This is the case for modern hypergraph partitioning instances, so there is a need for a terminal selection method that yields good partitions and fast FlowCutter runs.

Piercing. It is possible that we pierce but the cut weight does not increase, so we get better balance for free. Hence, a natural piercing heuristic is to select a node that does not increase the cut weight, if such a node exists [HS18a, YW96]. This is called the *avoid augmenting paths* heuristic, since such a node is not reachable from the opposite side: $v \notin T_r$ if the source side is grown and $v \notin S_r$ if the sink side is grown. If we pick such a node, there is no new flow to augment, and therefore we cannot afford to spend $\mathcal{O}(m)$ work to derive the next cut-sides, but this is not necessary. We take the previous cut-sides and grow the just pierced side via residual BFS from the new piercing node. The added work for this can be charged to the $\mathcal{O}(m)$ work for computing the previous cut-sides, since only one side was grown and the other did not change.

This also has an implication on the piercing node selection. The candidates are restricted to nodes incident to cut arcs, such that we can perform $\mathcal{O}(m)$ iterations (at least one cut arc changes per iteration) with $\mathcal{O}(\zeta)$ work in each iteration, for checking whether there is a candidate that avoids augmenting paths. Cut arcs are collected while deriving cut-sides.

Hamann and Strasser [HS18a] consider a secondary heuristic based on hop distances from the initial terminals, but its geometric interpretation seems only useful in road networks or instances with good low-dimensional embeddings. In our preliminary experiments it performed worse than random tie-breaking, a secondary heuristic that is also employed by Yang and Wong [YW96].

For hypergraphs, the direct piercing candidates are never hypernodes, but the out-node of a cut hyperedge (if piercing the source side) or the in-node (piercing sink side). All pins of that hyperedge effectively become terminals immediately because of the infinite capacity arcs from the in-node or out-node. Therefore, it may make sense to select the hyperedges to pierce based on some measure how many of its pins are not assigned to a side yet. We unsuccessfully experimented with several rating schemes: preferring many or little, absolute or relative numbers, and avoid creating hyperedges with pins in both sides. Unfortunately, we did not observe a significant difference to random tie-breaking. The only piercing heuristic that had measurable impact is a hop-distance (from cut) heuristic that only works in the context of flow-based refinement, which we describe in the next section.

Terminals. The second design option is the choice of initial terminals. Hamann and Strasser [HS18a] as well as Yang and Wong [YW96] use two randomly sampled nodes as terminals. As mentioned, this is too slow for modern instances, so there is a need for larger terminal sets, where FlowCutter has to compute fewer incremental cuts. Since nodes of the same terminal are assigned to the same block, we should be confident they belong together.

Li, Lillis, and Cheng [LLC95] compute a linear ordering (minimizing distance of adjacent nodes) using a heuristic spectral approach, and take the first, respectively last 10% as terminals. The ordering is also used for piercing node selection by picking candidates closest to the terminals of the pierced side. Their application is linear placement in VLSI design, but the merit of this approach in other contexts has not been investigated.

We experimented with pseudo-peripheral terminals (see Section 4.2) to no avail. What worked well for us is ensemble classification, and flow-based refinement (described in the next section), which can be interpreted as an extreme version of ensemble terminals (with an ensemble of size one).

Ensemble classification is a technique used in machine learning to build a strong classifier from multiple weak ones. We compute multiple $\Pi_1, \ldots, \Pi_r$ bipartitions with PaToH [CA99] (an extremely fast multilevel partitioner). Let $x \equiv y \Leftrightarrow \forall i = 1, \ldots, r : \Pi_i(x) = \Pi_i(y)$ denote the equivalence relation in which two nodes are equivalent if they are clustered together in each of the r bipartitions. An equivalence class is likely in the same block of a good bipartition and is thus suited as a terminal set. We order the equivalence classes by size in descending order and group two successive classes as one terminal pair. Generally speaking, the larger equivalence classes make for better terminal pairs, since the subsequent FlowCutter run is faster and the random tie-breaking piercing makes less errors. In this sense, the terminal set size is not strictly a parameter for trading off solution quality versus running time, as too small terminals generally result in bad quality. Rather, there is a sweet-spot to be found,

which is of course dependent on the instance and random choices and thus hard to find.

7.2 Flow-Based Refinement

Flow-based refinement is a technique that was first developed by Sanders and Schulz [SS11] for graph partitioning, and subsequently ported to hypergraphs by Heuer, Sanders, and Schlag [HSS19] by plugging in the Lawler network [Law73] and optimizing some cases in the model (e.g., small nets). The idea is to select a set of nodes B that are allowed to be moved between the two blocks V_0, V_1 of a given bipartition Π. We use $S = V_0 \setminus B$ and $T = V_1 \setminus B$ as terminals, such that S remains in V_0, T remains in V_1 and M may be assigned to either side in a minimum $S - T$ cut. The size (or weight) of B offers a trade-off between cut weight and running time. Sanders and Schulz as well as Heuer, Sanders and Schlag use one minimum cut computation instead of FlowCutter. Hence, the larger B the more likely it is that the found bipartition is not balanced. If the bipartition induced by the minimum cut is balanced and has smaller cut weight (or equal cut and better balance), it is accepted as the new currently best solution. The difference between the weight of the original cut nets and the flow value equals the decrease in connectivity, even if used on k-way partitions. We do not include nets that are connected to both S and T, since they cannot be removed from the cut. This method can also be used to optimize the cut-net metric on k-way partitions, by removing nets with pins in a third block.

Weight Constraint and Rescaling. If the weight of $B \cap V_i$ is restricted to $c(B \cap V_i) \leq (1+\varepsilon)\frac{c(V)}{k} - c(V_{1-i})$ the resulting bipartition is guaranteed to be balanced, no matter how the nodes in B are assigned. But, this restricts the cut optimization capabilities too much, so Sanders and Schulz propose to rescale ε with a parameter $\alpha \geq 1$, i.e., $c(B \cap V_i) \leq (1+\alpha\cdot\varepsilon)\frac{c(V)}{k} - c(V_{1-i})$ is the new constraint. If the result is accepted, α is doubled (capped at 16), if it is rejected α is halved, and another flow computation with the new α is started. This is repeated until $\alpha < 1$.

FlowCutter. A weakness of this approach is that a lot of computation time is wasted recomputing flow from scratch if the bipartition is barely imbalanced. Similarly, the much smaller B in the next iteration can waste a lot of cut reduction potential. FlowCutter shines in this scenario. With only few incremental flow problems and thus little additional work, we can trade off some cut weight for balance, to arrive at a bipartition that is accepted in the refinement framework. FlowCutter eliminates the need for the rescaling framework, and offers the finest possible granularity without the need to recompute flows from scratch.

In our experiments we consider two configurations: the above with $\alpha = 16$ fixed (no rescaling) used in the papers on FlowCutter refinement in sequential KaHyPar [GHSW20] and Mt-KaHyPar [GHS22], as well as $c(B \cap V_i) \leq c(V_i) - \beta c(V)$ with $\beta = 0.4$ for the default imbalance $\varepsilon = 0.03$ used in the papers on FlowCutter refinement for bipartitioning with

PaToH [GHW19a] and sequential KaHyPar [GHSW20]. These are simply different methods, but the latter usually yields smaller flow models and is thus faster.

Identifying Nodes to Move. Intuitively, the nodes we consider for moving should be close to the cut. Sanders and Schulz construct $B \cap V_0$ and $B \cap V_1$ separately with two BFSs restricted to V_0 and V_1, initialized with the respective boundary nodes. The visited nodes constitute B, and the BFSs are run until the weight constraint is tight. We call this the *corridor* approach, since a corridor around the cut is extracted.

In the first paper [GHW19a] we considered a second method, where we use PaToH to compute B by splitting off a third block from the input bipartition. In terms of cut size this method works equally well as the corridor approach, but it is slower, so we focus solely on the corridor approach.

Distance-Based Piercing. We can use the original bipartition as a guide for the piercing heuristic. To avoid that bad piercing decisions make it impossible to recover parts of the original cut, we use hop-distances from the original cut as a secondary criterion to avoiding augmenting paths, preferring larger distances from the cut. The distances are already computed in the corridor-approach, so this heuristic comes at no additional cost. Vertices from the other side of the original cut are rated with distance -1, i.e., chosen only after one side has been entirely added to the corresponding terminal vertices. We maintain the boundary vertices in a bucket priority queue and select candidates uniformly at random from the highest-rated non-empty bucket. New terminal vertices are removed lazily.

Improving Balance. Once the bipartition is balanced, Li et al. [LLC95] propose to keep piercing as long as no augmenting path is created in order to further improve the balance of the bipartition at no additional cut. Sanders and Schulz [SS11] as well as Heuer et al. [HSS19] use a more explicit approach: repeated randomized topological ordering of the Picard-Queyranne DAG [PQ82]. We follow the approach of Li et al. but incorporate repetitions by adding the ability to track and revert vertex and net assignments, as well as restore the state of the bucket PQs for piercing to the initial balanced partition, without resorting to copies. This heuristic is useful if integrated in a framework that uses other refinement mechanisms, such as FM local search. FM only considers balance-preserving moves, such that partitions with small imbalance give FM more leeway for optimization.

7.3 Maximum Flow on Hypergraphs

In this section, we show how to compute maximum flows directly on the hypergraph, instead of constructing the Lawler network. This is done purely for efficiency purposes, which is why we discuss several engineering details. We focus on augmenting path algorithms in this section, since we use Dinitz algorithm [Din70] in the first two papers [GHW19a,

GHSW20]. Later on, in Section 7.9 we also discuss a push-relabel algorithm, in the context of parallelization.

7.3.1 Pistorius-Minoux Generalized

Pistorius and Minoux [PM03] propose a method to run the augmenting path based Edmonds-Karp flow algorithm [EK72] directly on hypergraphs with unit capacity nets. The advantage of implementing flow algorithms directly on the hypergraph is the ability to identify and skip cases in which we cannot push any flow. If a unit capacity net is already saturated, we only need to follow one pin, the one sending flow in, instead of scanning the entire net. This corresponds to scanning the in-node of that net but not the out-node. The other advantage is that less memory is used which also has a running time impact since data is packed more tightly together. Furthermore, we can fuse the scan of residual arcs for in-nodes and out-nodes if the out-node must be scanned, since both comprise a scan of the pin-list. If the out-node is reached the in-node is always reached as well (in a forward traversal).

On an early testing instance with an average net size of 12.6 and maximum net size of 36, the unit capacity Pistorius-Minoux approach with Edmonds-Karp gave a 15x speedup over the Lawler network. This impressive result led us to abandon plain Lawler networks. We generalize the approach of Pistorius and Minoux to non-unit capacities and arbitrary augmenting path based algorithms, such as Dinitz algorithm [Din70]. In the following, we present the general version directly, and then discuss how it simplifies on unit capacities.

Let $\widetilde{f}(u,e)$ denote the amount of flow that vertex u sends into net e. Negative values indicate that u receives flow from e. Let $\widetilde{f}(u,e)^+ := \max(\widetilde{f}(u,e),0)$ denote the *absolute flow* u sends into e and $\widetilde{f}(u,e)^- := \max(-\widetilde{f}(u,e),0)$ the absolute flow u receives from e. Then, $f(e)$ can also be written as $\sum_{u \in e} \widetilde{f}(u,e)^+ = \sum_{u \in e} \widetilde{f}(u,e)^-$. We can push up to $\text{cap}(e) - f(e) + \widetilde{f}(u,e)^- + \widetilde{f}(v,e)^+$ flow from u via e to another pin $v \in e$. This is the residual capacity of e. The $\text{cap}(e) - f(e)$ term stems from the path (u, e_i, e_o, v), the $\widetilde{f}(u,e)^-$ stems from (u, e_o, v), and the $\widetilde{f}(v,e)^+$ term stems from (u, e_o, e_i, v). Figure 7.1 shows these three paths in the fourth, second and third step, respectively.

If $\text{cap}(e) - f(e) = 0$ and $\widetilde{f}(u,e)^- = 0$, we only need to iterate over the pins $v \in e$ with $\widetilde{f}(v,e) > 0$. On the Lawler network, we would have to scan the arc (e_i, v) for *every* pin $v \in e$. With unit capacity, there is at most one pin $v \in e$ with $\widetilde{f}(v,e) > 0$ and one $v' \in e$ with $\widetilde{f}(v,e) < 0$. Storing these in separate arrays flow-from and flow-to constitutes the approach of Pistorius and Minoux. With non-unit capacity, there can be many such pins. To avoid using $\mathcal{O}(p)$ extra memory, we partition the pin-lists of the nets into three subranges $\widetilde{f}(v,e)\{<0, =0, >0\}$. When the sign of $\widetilde{f}(v,e)$ changes (from pushing flow), we insert pin v into the correct subrange by performing swaps with the range boundaries.

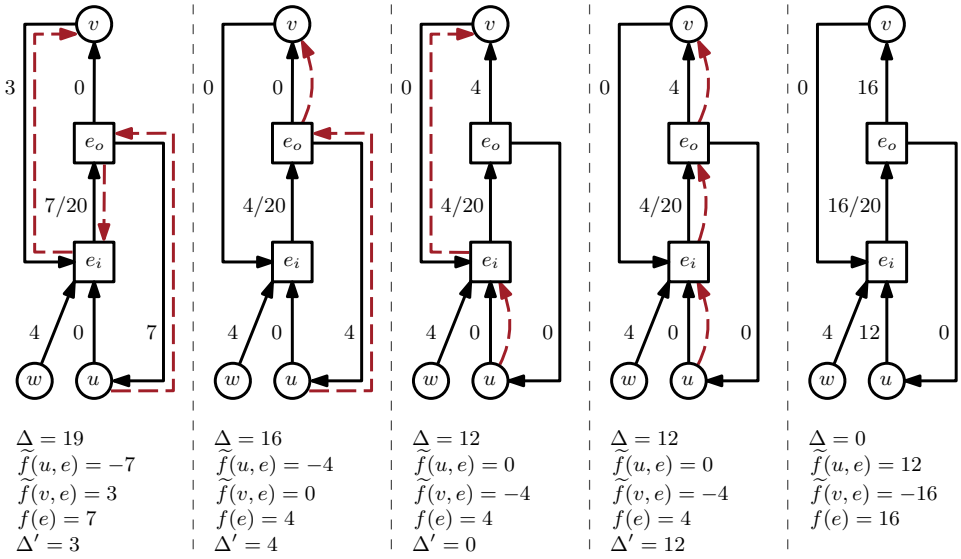

Figure 7.1: Example illustrating the four steps for pushing $\Delta = 19$ units of flow from u via hyperedge $e = \{u, v, w\}$ to v. Black arcs show the direction of the flow, dashed red arrows the direction we want to push flow in the Lawler network. The arc (e_0, w) is omitted for readability. Current values of Δ, $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, $f(e)$, and Δ' are shown at the bottom for each state.

7.3.2 Flow Routing

Pushing flow over a hyperedge is the elementary operation necessary to implement any flow algorithm on hypergraphs. Let Δ be the amount of flow to push. We update the values $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and $f(e)$ in four steps (see Figure 7.1 for an example). These steps correspond to paths $p_1 = (u, e_o, e_i, v)$, $p_2 = (u, e_o, v)$, $p_3 = (u, e_i, v)$, $p_4 = (u, e_i, e_o, v)$ in the residual Lawler network. The order of these steps is important to correctly update $f(e)$. First, we push $\Delta' := \min(\Delta, \widetilde{f}(u,e)^-, \widetilde{f}(v,e)^+)$ along p_1 by setting $f(e) \leftarrow f(e) - \Delta'$, $\widetilde{f}(u,e) \leftarrow \widetilde{f}(u,e) + \Delta'$, $\widetilde{f}(v,e) \leftarrow \widetilde{f}(v,e) - \Delta'$, and reduce the amount to push $\Delta \leftarrow \Delta - \Delta'$. Then, we push $\Delta' := \min(\Delta, \widetilde{f}(u,e)^-)$ along p_2, by updating $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and Δ as before. Note that we do not update $f(e)$ since the bridge arc (e_i, e_o) is not in p_2. Analogously to p_2, we push $\Delta' := \min(\Delta, \widetilde{f}(v,e)^+)$ along p_3. Finally, we push the remaining Δ along p_4 and update $f(e)$, $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and Δ as for p_1.

The order of these updates stems from the intention to avoid circulation of flow, i.e., a vertex should not have incoming and outgoing flow from the same hyperedge. Hence, we push back incoming flow first, reduce the delta, and then push the remaining delta in the

desired direction. In particular the first path (u, e_o, e_i, v) is not considered in the residual capacity since it is covered by the other three, but it is necessary to correctly drain flow from it in the routing procedure. If we spent double the memory to store incoming and outgoing flow separately, we could avoid this more complicated routing procedure. Yet, pushing flow accounts for only a negligible part of the running time, so we deem saving the memory worthwhile. In Section 7.9 we consider a parallel flow algorithm, where this is no longer possible, and we have to store incoming and outgoing flow separately for thread safety.

Note that the Lawler network is just used as a means of illustration, to identify the appropriate update steps. In the implementation, we do not actually construct the Lawler network. We only update the $\widetilde{f}(u, e)$, $\widetilde{f}(v, e)$, and $f(e)$ values as shown at the bottom of Figure 7.1.

7.3.3 Switching Directions

With FlowCutter, we solve incremental maximum flow problems. We keep the existing flow assignment and only need to run an augmenting path algorithm from the last piercing vertex [HS18a, YW96]. Since we can pierce both the source and sink side, we must therefore be able to push flow from the new sink to the existing sources. This does not mean pushing flow back, but transpose the network including flow direction, and then augment flow. Instead of transposing the entire data structure, we multiply the flow with a direction multiplier 1 for forward and -1 for backward. Additionally, we swap the data structures for in- and out-nodes in the flow algorithm (distance labels for Dinitz), special marker values for membership in S or T (fused into distance labels to save memory), the indices for the pinlist subranges $\widetilde{f}(v, e)\{< 0, = 0, > 0\}$, as well as the last piercing nodes of each side, such that the entire transposition takes only $\mathcal{O}(1)$ time. We can then run the regular forward implementation of the flow algorithm. In our implementation, this is all neatly hidden away behind an interface to access flow values, such that the flow algorithm implementation remains legible. We assume the arithmetic operation overhead for multiplying flow values is negligible, since the workload is heavily dominated by random memory accesses for labels and flow values.

7.3.4 Dinitz Algorithm

We now turn to the actual flow algorithm, namely Dinitz algorithm [Din70]. It works in two phases that are repeated until no flow can be augmented any more. The first phase is a BFS to construct distance labels in the residual network. If a sink is found during the BFS, the flow is not maximal and the second phase is invoked. Otherwise, the algorithm terminates, and the nodes visited by the BFS induce the side of the cut that was pierced last.

In the second phase, the distance labels induce a layered network. Flow is augmented along shortest paths in the layered network, i.e., the distance label always increases by one along the augmenting path. This is implemented as a DFS with backtracking, by following arcs leading to nodes with one higher distance label. When a sink is found, the smallest residual capacity (bottleneck capacity) on the path induced by the call-stack is pushed along

Algorithm 7.2: Construct Layered Network (BFS)

```
1  Q ← last piercing node
2  augment ← false
3  while Q not empty do
4      u ← Q.pop()
5      d ← dist[u] + 1
6      if u ∈ T
7          augment ← true
8      for e ∈ I(u) do
9          if d < out-dist[e]
10             if cap(e) − f(e) + f̃(u, e)⁻ > 0
11                 out-dist[e] ← d
12                 pin-it[e] ← H.first-pin[e]
13                 in-dist[e] ← d
14                 flow-sending-pin-it[e] ← H.first-flow-sending-pin[e]
15                 for v ∈ e do
16                     if d < dist[v]
17                         dist[v] ← d
18                         Q.push(v)
19                         net-it[v] ← H.first-net[v]
20         else if d < in-dist[e]
21             in-dist[e] ← d
22             flow-sending-pin-it[e] ← H.first-flow-sending-pin[e]
23             for v ∈ e with f̃(v, e)⁺ > 0 do
24                 if d < dist[v]
25                     dist[v] ← d
26                     Q.push(v)
27                     net-it[v] ← H.first-net[v]
28 return augment
```

the path. The search continues at one below the lowest saturated arc on the call stack. There is at least one saturated arc on the path, because the amount pushed was the bottleneck capacity.

Saving Traversals. When the algorithm terminates, we transpose the instance to also derive the other cut-side, and then transpose back, since we need both cut-sides for FlowCutter. This traversal also sets the data structures for the second phase, such that after piercing, we can transpose, augment flow using the already computed layered network (from an old piercing node to the new one), transpose back, and then run regular Dinitz. This saves one

BFS, which saves some running time in practice.

Layered Network Construction. We emulate Dinitz algorithm on the layered network without explicitly constructing it. For this, we need distance labels for each of the three node types: hypernodes (dist$[u]$), in-nodes (in-dist$[e]$) and out-nodes (out-dist$[e]$). Algorithm 7.2 shows pseudocode for our hypergraph implementation of the first phase. Instead of placing in-nodes and out-nodes in the BFS queue, we fuse their scans (line 15 for out-nodes in Algorithm 7.2, line 23 for in-nodes) with the hypernode scan loop (line 8). For each vertex/hypernode u we scan its incident nets $e \in \mathrm{I}(u)$. The corresponding out-node e_o is scanned, if it was not scanned yet (dist$[u] + 1 <$ out-dist$[e]$), and there is residual capacity $\mathrm{cap}(e) - f(e) > 0$ on the bridge arc (e_i, e_0) or u receives flow from e already ($\widetilde{f}(u, e)^- > 0$). If so, we set the distance label and a pin iterator that is used in the DFS for both the in-node and out-node. If we do not scan the out-node, we check whether the in-node has not been visited yet. If so, we visit all pins v that send flow into e, i.e., $\widetilde{f}(v, e)^+ > 0$, using the corresponding sub-range in our data structure. Note that we use the same distance label for in-node and out-node, regardless of which specific path leads to them.

Flow Augmentation. Algorithm 7.3 shows the pseudocode for the flow augmentation phase (DFS). The implementation emulates recursion in an iterative code, using an explicit call-stack on the heap to avoid stack overflows on huge instances, and to avoid function call overheads in hot parts of the code. Again, the loops are fused to avoid pushing nets' in-nodes or out-nodes on the stack. The iterators are used to scan each incident net list and each pin list only once, as well as each list of flow-sending pins only once. They are not a byproduct of the iterative implementation, as they are also used in purely recursive implementations. The iterators can only be incremented after backtracking from a node, i.e., if it leads to a dead end. Lines 22-24 implement the backtracking for dead ends. Resetting the distance in line 24 is necessary since the pin-iterator on the previous layer still points to u. With the reset, the check dist$[v] = d$ fails (u in current layer in the role of v in the previous layer), so the iterator is incremented.

Theoretical Complexity and Running Time in Practice. Due to the backtracking the DFS may take super-linear time (also on plain graphs). In particular, the backtracking after flow augmentation (line 27) causes the search to reuse the path from the source up to the lowest saturated arc. Recall the running time of $\mathcal{O}(n^2 m)$ of Dinitz algorithm on graphs, which translates to $\mathcal{O}(\min(n, m)^2 \cdot p)$ on hypergraphs. The graph bound stems from performing at most n repetitions of BFS and DFS, since each DFS increases the number of layers by at least one in the next BFS, and the $\mathcal{O}(nm)$ time bound for DFS, which is due to reusing partial paths. Note not $(n + m)^2$ for the hypergraph bound since $\min(n, m)$ is a bound on the layers in our version. Let l denote the number of layers. In the DFS, at most l forward steps are performed between either backtracking (eliminating the incoming arc) or saturating (and thus eliminating) an arc from the layered network, which can happen at most m times. Flow

Algorithm 7.3: Augment Flow (DFS)

```
1  stack.push(last piercing node)
2  while stack not empty do
3      u ← stack.top()
4      d ← dist[u] + 1
5      w ← ⊥
6      for e ∈ I(u) starting at net-it[u] do
7          if out-dist[e] = d and cap(e) − f(e) + f̃(u, e)⁻ > 0
8              for v ∈ e starting at pin-it[e] do
9                  if dist[v] = d
10                     w ← v
11                     break
12             pin-it[e]++
13         if w = ⊥ and in-dist[e] = d
14             for v ∈ e with f̃(v, e)⁺ > 0 starting at flow-sending-pin-it[e] do
15                 if dist[v] = d
16                     w ← v
17                     break
18             flow-sending-pin-it[e]++
19         if w ≠ ⊥
20             break
21         net-it[u]++
22     if w = ⊥
23         stack.pop()
24         dist[u] ← ∞
25     else
26         if w ∈ T
27             lowest-bottleneck ← PushBottleneckCapacityAlongStack(stack)
28             stack.popDownTo(lowest-bottleneck)
29         else
30             stack.push(w)
```

augmentation also costs $\mathcal{O}(l)$ time, which thus proves the DFS bound. In practice, we often observe just a constant number of repetitions, and the DFS behaves linearly in p, since the instances often have low diameter. The running time shares are roughly 40% for BFS and 60% for DFS, though numbers vary across instances.

7.4 Disconnected Hypergraphs

By default, FlowCutter only works on connected inputs, which is often a reasonable assumption. However, out of the 488 hypergraphs in benchmark set A, 173 are disconnected, so we cannot run plain FlowCutter on them. We could partition each component separately and glue the solutions together, but we may get much smaller cuts if we split some components unevenly. Often there are some small components that need not be cut, so partially this is rather a packing problem.

For the refinement version, this is not so critical. We rarely observe disconnected inputs, since even if the cut spans across multiple components, none of them are visited fully by the BFSs, such that terminals are placed in them. This is because small components are already packed together without cutting them in the input partition, because PaToH has an initial partitioning heuristic tailored towards packing. However, for plain FlowCutter the issue persists. In Section 7.7, we use this even with the refinement version, in Section 7.8 and 7.13 we do not, because it complicates the implementation.

An obvious approach for handling disconnected hypergraphs is connecting components artifically. We refrained from this because a component which intersects neither the initial S nor T would become assimilated by only one side and thus could never be split. A similar issue arises, if we instead permit adding piercing nodes from components without terminals. The first side to add a terminal takes the entire component, since there is no opposite terminal.

Use Pareto Bipartitions. Instead, we exploit the fact that FlowCutter computes a series of cuts with different trade-offs between cut and balance. We run the core algorithm on each component up to $\varepsilon = 0$, and systematically try possible combinations from the Pareto sets of each component. This can be stated as a generalization of subset sum. In the subset sum problem we are given a multiset of positive integers $A = \{a_1, \ldots, a_z\}$, the items, and a target sum W, and want to know whether a subset of A sums to W.

Subset Sum. Finding a bipartition with zero cut is equivalent to subset sum, where the items are the sizes of the components and W is the minimum size of the smaller block. We are interested in any subset summing to at least W. Let A be sorted in increasing order and let $Q(i, S)$ be a boolean variable, which is exactly if a subset of the first i items sums to S. The standard pseudo-polynomial time dynamic program (DP) [CLRS01, Section 35.5] for subset sum computes solutions for all possible target sums. It fills the DP table Q by iterating through the items in increasing order and setting $Q(i, S)$ to true if $Q(i-1, S-a_i)$ or $Q(i-1, S)$ is true. For filling row i, only row $i-1$ is required, so the memory footprint is not quadratic.

Splitting Cost Subset Sum. We now turn to non-zero cut bipartitions by allowing to split items in different ways and associating costs with the splits. Let $C_1, \ldots, C_z$ denote the decomposition into connected components. We have multiple bipartitions P_i in the Pareto

set on component C_i, at most one for every possible size of the smaller side, i.e., $|P_i| \leq |C_i|/2$. These correspond directly to the different ways we can split items. The associated cost is the cut size.

We modify the standard subset sum DP to minimize the added cuts instead of finding any subset. We ensure every item is split only one way in a solution. For each C_i, we iterate through the possible bipartitions in the Pareto set, and try the smaller and larger side of the component bipartition for the smaller side of the global bipartition. The costs of the possible global bipartitions are stored in the DP table, storing the best one for each possible size of the smaller side $0, ..., n/2$. The worst case asymptotic running time of this DP is $\mathcal{O}(\sum_{i=1}^{z} \sum_{j=1}^{i-1} |P_i||P_j|)$.

Optimizations. We propose some optimizations to make the approach faster in practice. First we solve standard subset sum to check whether there is an ε-balanced bipartition with zero cut. This actually applies to eight hypergraphs in our benchmark set.

Our main optimization is what we call *gap-filler*. We find the largest $g \in \mathbb{N}$ such that for every $x \in [0, g]$ there are connected components, whose sizes sum to x. Computing g is possible in $\mathcal{O}(n)$ time. Let $C_1, \ldots, C_z$ be sorted by increasing size, which takes $\mathcal{O}(n)$ time using counting sort. Then $g = \sum_{j=1}^{r-1} |C_j|$ for the smallest r such that $|C_r| > 1 + \sum_{j=1}^{r-1} |C_j|$. It is never beneficial to split the components $C_1, \ldots, C_{r-1}$.

For most hypergraphs in set A, we do not have to invoke the DP because we split only the largest component due to the gap-filler optimization. For the hypergraphs on which we do invoke the DP, its running time is negligible. Nonetheless, it is easy to construct a worst case instance, where the quadratic running time is prohibitive. For a robust algorithm, we propose to sample bipartitions from every P_i so that the worst case running time falls below some input threshold, though we stress that this has not been implemented in our framework. The samples should include perfectly balanced bipartitions to guarantee that a balanced partition on H can be combined from those on $H[C_i]$.

7.5 Isolated Vertices

In this section, we discuss an optimization particular to hypergraphs, with which we may achieve a balanced bipartition earlier, which is important for both cut weight and running time. A vertex $v \notin S \cup T$ is called *isolated* if all of its incident nets have pins in both S and T. An isolated vertex can be moved without affecting the cut, because $v \notin S_r \cup T_r$ (due to flow maximality) and its incident nets remain in both the source- and sink-side cut over the course of the algorithm. More precisely, at each bipartition, we can freely reassign them to balance the bipartition as much as as possible. Solving this assignment problem optimally constitutes a subset sum problem. With unweighted vertices, it is completely trivial and comes at no additional cost, so we integrate this in the unweighted version [GHW19a].

The results are not as pronounced for the refinement as for plain FlowCutter, because nets with pins in the initial S and T are removed, and in the multilevel version they are negligible.

For completeness sake, we still describe our approach for non-uniform vertex weights below, since the multilevel version introduces non-uniform weights through coarsening, even for uniform weight inputs. It is implemented in the code, and was used for initial experiments with sequential KaHyPar [GHSW20], but disabled for the experiments in Section 7.8 as well as Section 7.13, even though its running time was negligible during profiling.

Introducing non-uniform vertex weights makes the subset sum problem non-trivial, because arbitrary divisions of the total vertex weight are not necessarily possible. Since isolated vertices remain isolated, the problem instances are incremental. To solve the problem, we use the pseudo-polynomial DP for subset sum. It maintains a lookup table of partition weights that are summable with isolated vertices. After obtaining a new bipartition, we update the DP table to incorporate potential new isolated vertices. For each new isolated vertex v, we iterate over the subset sums x in the table and insert $c(v) + x$ if it was not yet a subset sum.

As an optimization, we maintain a list of ranges that are summable (consecutive entries). The balance check takes constant time per range. To merge ranges efficiently, we store a pointer from each DP table entry to its range in the list. When a new subset sum x is obtained, we check whether $x - 1$ and $x + 1$ are also subset sums, and extend or merge ranges as appropriate. Since entries in the DP table cannot be reverted easily, we do not add isolated vertices to the DP after the first balanced partition is found (improve balance by piercing as long as no augmenting path is created).

7.6 Repetition Interleaving

To improve solution quality, one can perform multiple randomized repetitions of FlowCutter and take the minimum cut. Recall that the running time of one repetition is $\mathcal{O}(\zeta m)$, where ζ is the final cut size. If the repetitions are run one after another, the running time depends on the worst of the found cuts. Instead, Hamann and Strasser [HS18a] propose to run them simultaneously, such that the overall running time depends on the best found cut. We should interleave the executions by always running the repetition with the currently smallest flow value. In their implementation[1], the execution only yields once a cut is found. This breaks the claimed running time because the final cut of the current execution may be larger than the final overall cut. Instead, we invest $\mathcal{O}(m)$ work for flow augmentation and then yield, to resume the instance with the smallest flow value. In the context of road networks, this small detail does not matter since incremental cut sizes often differ by only one and the different executions find similar cut sizes. However for our hypergraph instances these assumptions are wrong and it makes a noticeable difference.

[1] https://github.com/kit-algo/flow-cutter/blob/master/flow_cutter.h#L649

Table 7.1: Average and quantile speedups of the hybrid and interleaved execution strategies over consecutive execution.

	gmean	min	0.1	0.25	median	0.75	0.9	max
hybrid	1.46	0.4	0.96	1.07	1.29	1.55	2.57	49.66
interleaved	1.61	0.55	1.0	1.14	1.38	1.74	2.66	175.47

7.7 2-way Flat Experiments

We now turn to the first set of experiments stemming from the first paper [GHW19a] on flow-based refinement. We evaluate our flow-based refinement with FlowCutter using PaToH as the initial partitioner. This algorithm is called ReBaHFC (for Refinement and Balancing HyperFlowCutter). In addition we consider running plain FlowCutter (dubbed HyperFlowCutter or HFC here) with random and ensemble terminal pairs.

The experiments are conducted on set A and a cluster of machines of type D. Each partitioner in the evaluation is run five times (as opposed to the usual ten on set A) with different random seeds, and we report the arithmetic mean cut in the plots. The code[2] is restricted to $k = 2$ and unweighted instances, which is why we present a second set of experiments in the next section; for arbitrary k, support for weights and integrated in the multilevel partitioner KaHyPar.

We consider two imbalance values $\varepsilon = 0.03$ which is a commonly used default value in the literature, and $\varepsilon = 0$, i.e., perfect balance. Plain FlowCutter is not even remotely competitive on $\varepsilon = 0.03$, as opposed to results by Hamann and Strasser on the Walshaw benchmark [HS18a]. Therefore, we also consider the perfectly balanced case $\varepsilon = 0$, where it provides the best quality out of the tested algorithms.

7.7.1 Repetition Interleaving

Table 7.1 shows the average speedup and some quantiles when interleaving the execution of 20 random terminal vertex pairs, instead of running them one after another; repeated for 5 random seeds. Here plain HyperFlowCutter is run until $\varepsilon = 0$. Because consecutive execution exhibits more memory locality, we also tested a hybrid strategy where the instance with the currently smallest flow is allowed to make multiple progress iterations. Interleaving outperforms consecutive execution by a factor of 1.61 in the geometric mean, a factor of 1.38 in the median, a factor of 2.66 on the 0.9 quantile, whereas the 0.1 quantile has a factor of 1. This shows that saving work is more important than memory locality.

[2] `github.com/kit-algo/HyperFlowCutter`

Table 7.2: Overview by hypergraph class, how often ReBaHFC improves the initial partition.

Algorithm	ε	all	SPM	Dual	Primal	Lit	DAC	VLSI
ReBaHFC-Q	0.00	37.3	39.8	47.4	23.9	33.0	66.0	34.4
ReBaHFC-D	0.00	49.2	58.4	59.8	27.4	41.3	58.0	47.8
ReBaHFC-Q	0.03	52.5	47.2	51.5	50.9	57.8	82.0	76.7
ReBaHFC-D	0.03	64.5	61.5	65.9	56.7	71.1	76.0	86.7

7.7.2 ReBaHFC versus PaToH

ReBaHFC Configuration. With ReBaHFC we use only one initial partition computed with PaToH. The imbalance for the initial partition is set to the same value as the desired imbalance ε for the output partition, which proved superior to larger imbalances on initial partitions. The block-size parameter β should depend on ε, so we settled on $\beta = 0.4$ for $\varepsilon = 0.03$ and $\beta = 0.46$ for $\varepsilon = 0$. In the technical report [GHW19b] we conduct a thorough parameter study for these choices. We compute one initial partition and run five differently randomized FlowCutter repetitions in interleaved fashion. This number seems to provide decent quality without increasing running time too much. We consider two variants: ReBaHFC-D, which uses PaToH with default preset and ReBaHFC-Q, which uses PaToH with quality preset.

Improvement over Initial Partition. Table 7.2 reports how often ReBaHFC improves the initial partition, for the different hypergraph classes of the benchmark set. As expected, ReBaHFC-Q could improve fewer solutions than ReBaHFC-D since the PaToH baseline is already better. Furthermore, ReBaHFC has more opportunities for refinement with $\varepsilon = 0.03$, in particular on the DAC and VLSI instances, whereas it struggles with the Primal and Literal SAT instances for $\varepsilon = 0$.

Direct Comparison. In Figure 7.2 we show performance profiles with direct comparisons between ReBaHFC and the corresponding PaToH version, as well as ReBaHFC-D versus PaToH-Q for $\varepsilon = 0.03$. While ReBaHFC consistently improves upon its PaToH baseline, ReBaHFC-D does not catch up to PaToH-Q in a direct comparison. If we take the minimum cut out of the five seeds instead of the arithmetic mean, they achieve equal partition quality (not shown). Note that the PaToH runs use different random seeds than the internal calls in ReBaHFC, so it is possible for stand-alone PaToH to find smaller cuts. Otherwise, the curve of ReBaHFC would start at 1.

7.7.3 Comparison for $\varepsilon = 0.03$

We now turn to the horse-race comparison for the default imbalance parameter $\varepsilon = 0.03$. As plain FlowCutter is not competitive, we consider only ReBaHFC here. Since KaHyPar-HFC

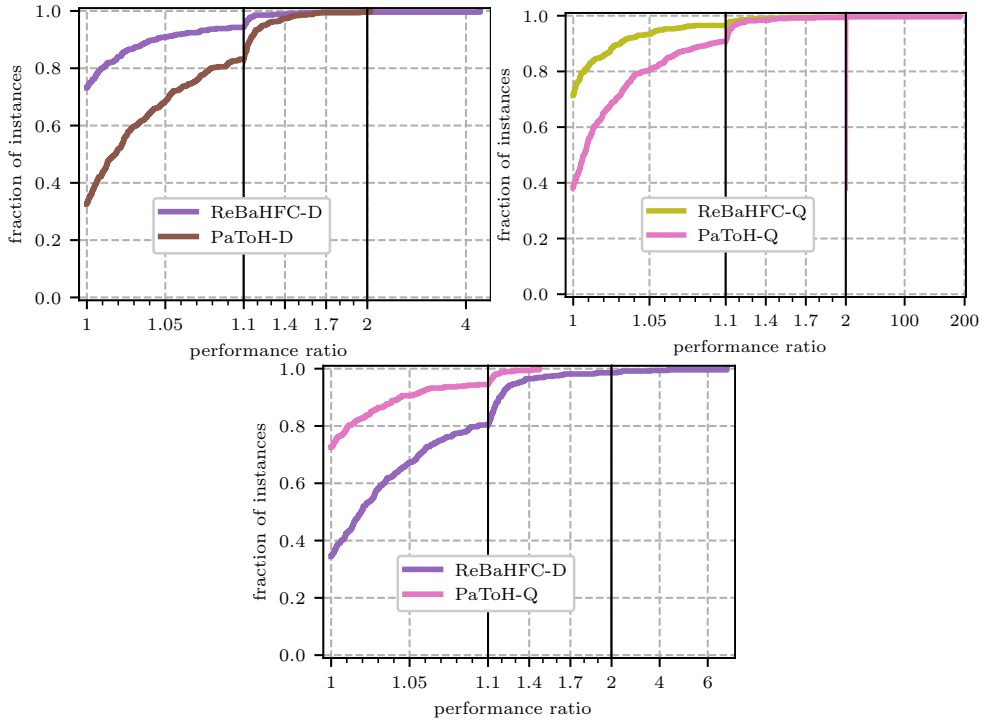

Figure 7.2: Direct solution quality comparison of ReBaHFC versus PaToH for $\varepsilon = 0.03$.

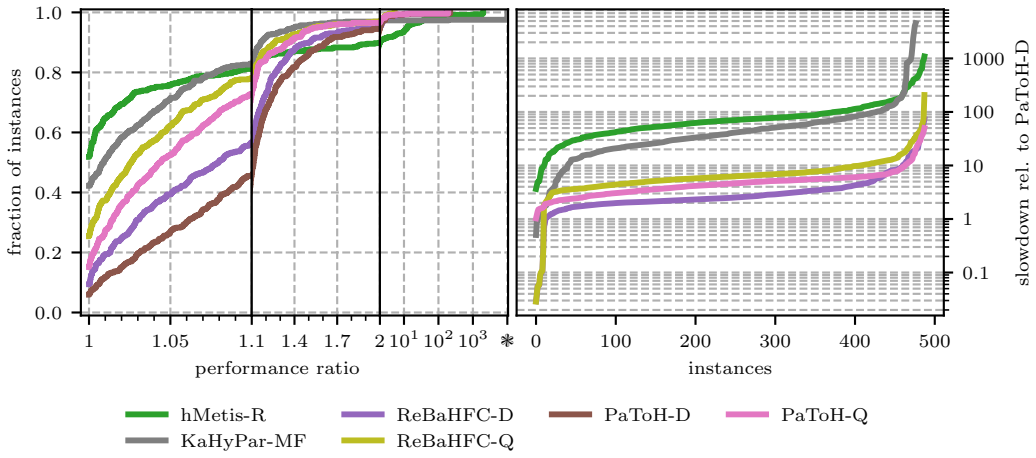

Figure 7.3: Quality and running times of ReBaHFC and competitors on set A for $\varepsilon = 0.03$.

did not exist yet, we consider the previous flow-based refinement KaHyPar-MF instead, as this was the best KaHyPar version at the time.

The geometric mean running times are 0.5s for PaToH-D, 1.4s for ReBaHFC-D, 2.27s for PaToH-Q, 3.1s for ReBaHFC-Q, 17.08s for KaHyPar-MF and 34.15s for hMetis-R. While hMetis-R is the best algorithm regarding partition quality, contributing around 52% of the best solutions, it is also the slowest, followed closely by KaHyPar-MF at 42%. There is a factor 3 gap in running time between ReBaHFC-D and its PaToH baseline, however there are some instances where it is actually faster. These are instances where the DP for disconnected inputs solves the instance with zero cut, or much smaller components are partitioned. The running time gap between PaToH-Q and ReBaHFC-Q is smaller, because there is less opportunity for refinement. FlowCutter reaches the flow bound from the initial partition faster (smaller cut than with PaToH-D) if it cannot improve the partition. Regarding partition quality, ReBaHFC-D is situated between PaToH-D and PaToH-Q, whereas ReBaHFC-Q is between PaToH-Q and KaHyPar-MF. Furthermore, we note that while hMetis-R is the top competitor in the first segment $[1.0, 1.1]$, it falls off in the second segment $[1.1, 2.0]$, being overtaken by PaToH-D at around 1.55, and by ReBaHFC-D at 1.4. This indicates that the worst solutions by the other competitors are closer to the best than the worst of hMetis.

Looking at effectiveness tests in Figure 7.4, we see that ReBaHFC-D almost catches up to KaHyPar-MF, and ReBaHFC-Q is actually better. We see a similar picture for hMetis-R and ReBaHFC-Q except the gap is slightly larger due to hMetis being less robust. Furthermore, ReBaHFC-D overtakes hMetis already at the 1.05 factor. Lastly, PaToH-Q narrowly beats ReBaHFC-D in effectiveness tests, indicating that V-cycles are stronger than flow-based refinement in the medium quality tier, whereas ReBaHFC-Q narrowly beats PaToH-Q.

7.7.4 Perfect Balance

Even though the setting $\varepsilon = 0$ has received no attention in hypergraph partitioning and only some attention in graph partitioning [SS13, MMS09, CBM07, BH10, BH11a, BH11b, DW12], we consider it here. Previous studies on perfectly balanced partitioning for graphs have focused on running time intensive metaheuristics such as evolutionary algorithms [SS13, BH11a, BH10] or tabu search [BH11b] and even an exact branch-and-bound algorithm [DW12]. Therefore, we include KaHyPar-EVO [ASS18c], the evolutionary algorithm of KaHyPar as well as plain HyperFlowCutter in addition to the already considered algorithms. While we report results for hMetis, we note that it rejects $\varepsilon < 0.002$ for bipartitions as input. We run it with $\varepsilon = 0.002$ but it is therefore often unable to compute balanced partitions.

Plain HyperFlowCutter Configuration. With plain HyperFlowCutter we want to push the envelope on solution quality for $\varepsilon = 0$. We do this regardless of running time because ReBaHFC already provides a good time-quality trade-off. Therefore we use up to $q = 100$ terminal pairs, the maximum value used for the graph variant [HS18a], and call this configuration HFC-100. Based on experiments in the technical report [GHW19b], we use 3

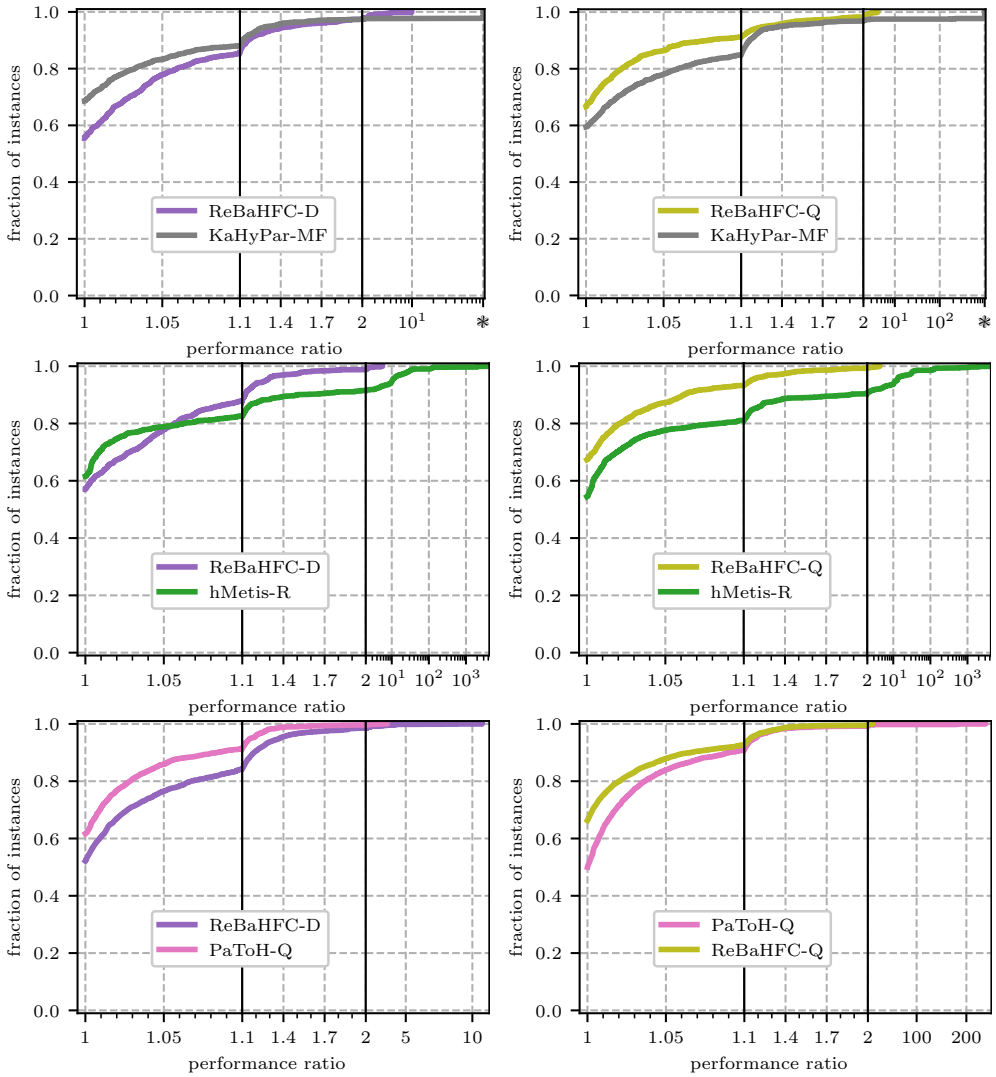

Figure 7.4: Effectiveness tests with virtual instances on benchmark set A, comparing ReBaHFC with KaHyPar-MF, hMetis-R and PaToH-Q for $\varepsilon = 0.03$.

ensemble terminal pairs and 97 random vertex pairs. For the ensemble terminals we compute 10 bipartitions with PaToH-D. The reported running time always includes the time for these runs.

On 42 of the 488 hypergraphs, plain HyperFlowCutter with 100 terminal pairs exceeds the eight hour time limit. One downside of interleaving executions is that the solution is only available once all terminal pairs have been processed. Instead of interleaving all 100 executions, we run four waves of $\langle 1, 5, 14, 80 \rangle$ terminal pairs consecutively and interleave execution within waves. An improved bipartition is available after every wave, so that, even if the time limit is exceeded, a solution is available as long as the first wave has been completed. We chose wave sizes, so that completing waves four and three corresponds to 100 and 20 terminal pairs, respectively, as these values were used in [HS18a]. The first wave consists of the first ensemble terminal pair, the second/third wave consist of 5/14 random terminal pairs and the fourth wave consists of 78 random plus two ensemble terminals. There are 438 hypergraphs for which the fourth wave finishes, 35 for which the third but not the fourth wave finishes, 6 for the second, 1 for the first. Furthermore, there are 8 hypergraphs which are partitioned with zero cut, using just the subset sum preprocessing.

Evolutionary KaHyPar Configuration. The evolutionary algorithm KaHyPar-EVO generates, manages and improves a pool of solutions until a time limit is exceeded, and outputs the minimum cut out of all generated solutions. We set the instance-wise time limit to the maximum of the running times of HFC-100 and KaHyPar-MF to evaluate whether KaHyPar-EVO can yield better solution quality when given the same running time as HFC-100. As opposed to the original paper, we configure KaHyPar-EVO to use flow-based refinement, which further improves solution quality. KaHyPar-MF is unable to find any balanced bipartition on 4 hypergraphs, whereas KaHyPar-EVO always finds one, as the latter computes multiple solutions. Furthermore, KaHyPar-MF exceeds the time limit on 7 hypergraphs and KaHyPar-EVO on an additional 17, without reporting intermediate solutions.

Horse-Race Comparison. Figure 7.5 shows the relative running times and performance profiles of all tested algorithms. HFC-100 produces the best solutions on 244 hypergraphs (50%), followed by ReBaHFC-Q (142). This shows that with exorbitant running time, HFC-100 produces high quality solutions for $\varepsilon = 0$. However the time-quality trade-off is clearly in favor of ReBaHFC-Q, especially since the solution quality of the latter is closer to the best cut for the instances on which it does not find the best cut, as opposed to HFC-100. Both KaHyPar and KaHyPar-EVO do not perform well, reaching the 50% instance marker only at a factor 2. PaToH is better than KaHyPar for $\varepsilon = 0$ because it includes a KL [KL70] refinement pass as opposed to KaHyPar which only uses FM [FM82]. Lastly, hMetis-R finds perfectly balanced partitions on only 30% of the instances, out of which it however contributes 13% of the best partitions.

The geometric mean running times are 35.29s for hMetis-R, 7.58s for KaHyPar-MF, 850.16s for KaHyPar-EVO, 714.51s for HFC-100, 0.52s for PaToH-D, 0.67s for ReBaHFC-D, 2.36s for PaToH-Q, and 2.4s for ReBaHFC-Q. Noticeably the running time gap between PaToH-D and ReBaHFC-D is much smaller than for $\varepsilon = 0.03$. This is partially because we used a different block-size parameter α, and partially because there is less opportunity for refinement.

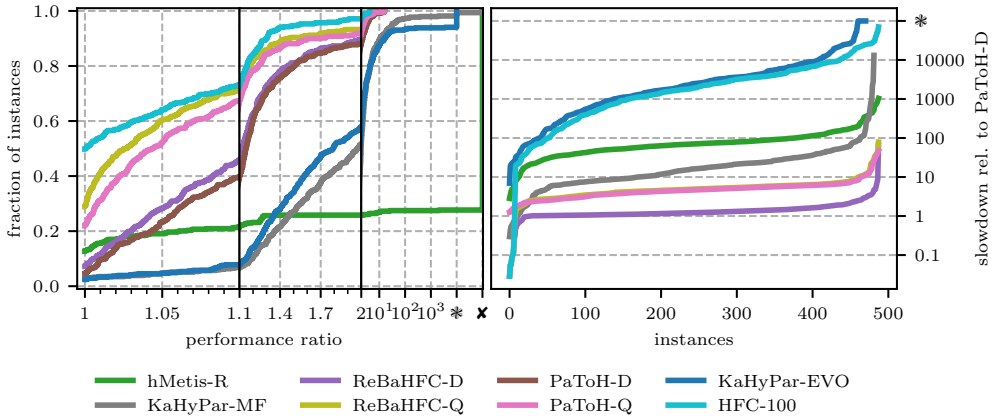

Figure 7.5: Quality and running times of ReBaHFC and competitors on set A for $\varepsilon = 0.0$

7.8 k-way Multilevel Experiments

Algorithm 7.4: k-way Flow-Based Refinement

Input: Hypergraph $H = (V, E, c, \omega)$, k-way partition Π of H

$\mathcal{Q} \leftarrow$ `ConstructQuotientGraph`(H, Π)
while improvement found **do**
  for $(V_i, V_j) \in \mathcal{Q}$ with V_i, V_j active **do**
    $B := B_i \cup B_j \leftarrow$ `GrowRegion`(H, V_i, V_j) // $B_i \subseteq V_i, B_j \subseteq V_j$
    $(\mathcal{N}, s, t) \leftarrow$ `ConstructFlowHypergraph`(H, B)
    $(M, \Delta_{\text{gain}}) \leftarrow$ `FlowCutterRefinement`$(\mathcal{N}, s, t)$
    if $\Delta_{\text{gain}} \geq 0$
      `ApplyMoves`(H, Π, M)
      mark V_i, V_j as active

In our second paper on flow-based refinement [GHSW20], we integrated refinement with FlowCutter into KaHyPar. The novelty is that we must now support weighted inputs. To refine k-way partitions we use the active block scheduling of Sanders and Schulz [SS11] as already implemented in KaHyPar-MF [HSS19] and outlined in Algorithm 7.4. The algorithm is organized in multiple rounds, in which the blocks that contributed to an improvement in the previous round are scheduled for pair-wise refinement.

In the evaluation presented in this section, we consider two configurations KaHyPar-HFC and KaHyPar-HFC-Eco, which differ in the way the size constraints for the movable vertices are chosen. KaHyPar-HFC uses the same method as KaHyPar-MF (confer Section 7.2),

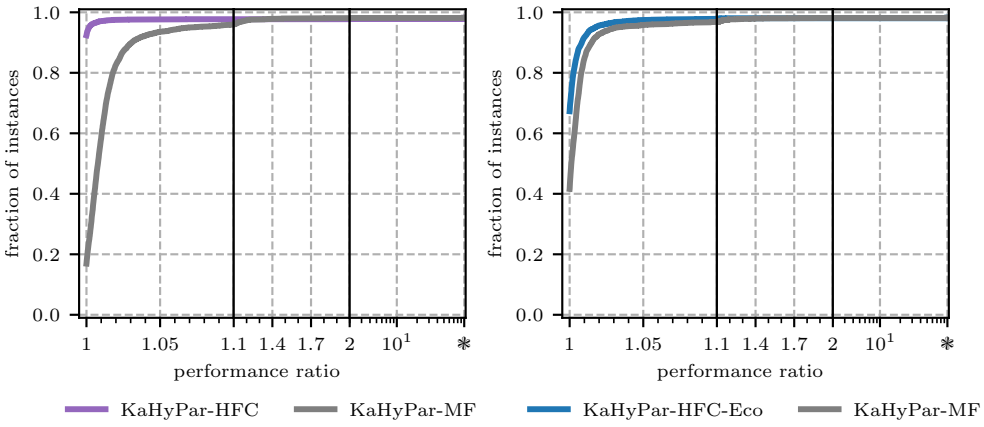

Figure 7.6: Partition quality of KaHyPar-HFC and KaHyPar-HFC-Eco versus KaHyPar-MF.

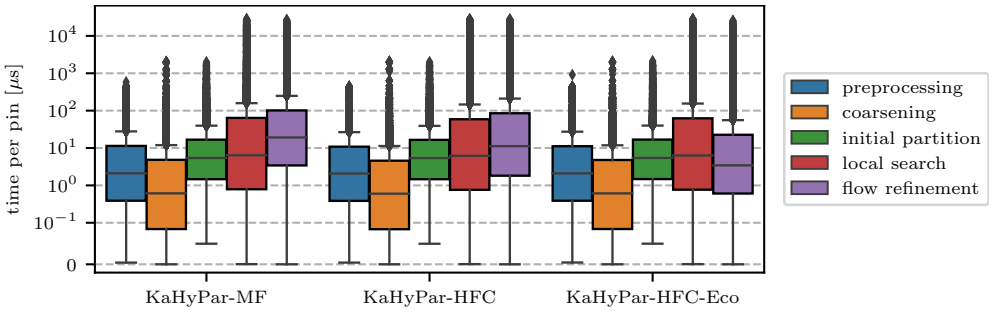

Figure 7.7: Running time per pin spent in the different phases of KaHyPar variants with flow-based refinement.

whereas KaHyPar-HFC-Eco uses the method of ReBaHFC. Note that the terminology differs from the paper, but the slower KaHyPar-MF rule has emerged as the default configuration, and is used in later experiments in this chapter. As in the previous section we use a cluster of machines of type D, $\varepsilon = 0.03$, 8 hours timelimit and benchmark set A, but consider different values of $k \in \{2, 4, 8, 16, 32, 64, 128\}$ now and run 10 repetitions with different seeds (the usual setup for set A).

Comparison with KaHyPar-MF. KaHyPar-HFC-Eco computes solutions with better, equal, or worse quality than KaHyPar-MF on 1933, 367, 1056 instances, respectively. On the remaining 60 instances neither finished within the time limit. As Figure 7.6 (right) shows, the performance ratios are consistently better, though not by a large margin.

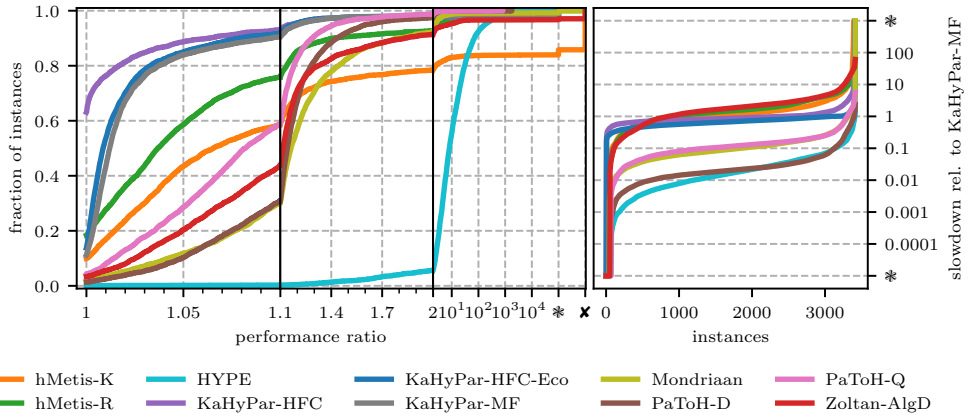

Figure 7.8: Partition quality and running time of KaHyPar-HFC and competitors on set A for $\varepsilon = 0.03$.

The median of the flow-based refinement time ratios (KaHyPar-HFC-Eco divided by KaHyPar-MF) is 0.18, the 75th percentile is 0.26, and the 90th percentile is 0.51. Hence, the flow-based refinement of KaHyPar-HFC-Eco is significantly faster than that of KaHyPar-MF. As a rough estimate, flow-based refinement constitutes about 40% of KaHyPar-MF's overall running time [HSS19]. With a geometric mean running time of 44.84s, KaHyPar-HFC-Eco is about 33% faster than KaHyPar-MF at 67.07s. The improved running time is partially due to faster flow computation and partially due to smaller flow hypergraphs. Since KaHyPar-HFC-Eco uses smaller flow hypergraphs, the improved solution quality can be attributed to using FlowCutter in the refinement.

With 62.49s, KaHyPar-HFC is moderately faster than KaHyPar-MF and computes solutions of better, equal, or worse quality on 2776, 381, 198 instances, respectively (with 61 instances on which neither finished within the time limit). The partition quality of KaHyPar-HFC is thus clearly better. The flow hypergraphs have the same maximum size in these two variants. Hence, the faster flow computation more than compensates the additional work incurred by FlowCutter refinement.

Figure 7.7 shows box plots for the different phases of KaHyPar. The running times of preprocessing, coarsening, and initial partitioning remain unchanged, as they are not influenced by the refinement phase. Local search and flow-based improvement both modify the solution and thus influence one another. The plots show that the running time of local search remains largely unchanged, while our variants substantially reduce the running time of flow-based refinement. For the Eco variant, flow-based refinement even becomes faster than local search.

Table 7.3: Overview of geometric mean running times and number of instances with timeouts, errors, or imbalanced partitions.

	KaHyPar-HFC	KaHyPar-HFC-Eco	KaHyPar-MF	hMetis-R	hMetis-K
gmean time (s)	62.49	44.84	67.07	96.55	73.65
timeouts	75	66	63	63	27
imbalanced	0	0	0	0	484
error	0	0	0	0	0

	PaToH-Q	PaToH-D	Zoltan-AlgD	Mondriaan	HYPE
gmean time (s)	7.48	1.47	107.60	6.44	1.03
timeouts	0	0	12	19	0
imbalanced	0	0	99	3	0
error	0	0	0	4	1

Full Comparison. We now compare the three KaHyPar variants with other state-of-the-art algorithms. In contrast to previous chapters, we add hMetis-K [KK00], Mondriaan [VB05], Zoltan-AlgD [SCS19a] and HYPE [May+18], as they were still included in the experimental setup for this paper. In later papers they were excluded to focus on a smaller set of non-dominated competitors. As HYPE produces substantially worse partitions when randomized, we report only the results of one non-randomized run.

Figure 7.8 (left) shows that KaHyPar-HFC outperforms all competing algorithms in terms of partition quality, and that hMetis-R emerges as the best competitor outside the KaHyPar variants. KaHyPar-HFC computes the best solutions on 63% of all instances, KaHyPar-HFC-Eco on 14%, and hMetis-R on 18%, as shown by their $\rho_a(1)$ values.

Recall that these values alone do not permit a ranking between the algorithms. Both KaHyPar-MF and KaHyPar-HFC-Eco compete with KaHyPar-HFC for the best solutions on similar instances, and thus end up with a lower $\rho_a(1)$ value. Compared individually, KaHyPar-HFC-Eco is better than hMetis-R on 69.9% of the instances. Additionally, KaHyPar-HFC-Eco and KaHyPar-MF approach the profile of KaHyPar-HFC much faster.

The KaHyPar variants are all within a factor of 1.1 of the best solution on over 90% of the instances, and within 1.4 on over 97%, whereas hMetis-R achieves 76% and 90%. PaToH-Q and PaToH-D solve more instances than hMetis-R within factors of roughly 1.2 and 1.4, and more instances than hMetis-K within 1.1 and 1.2. Mondriaan is similar to PaToH-D and Zoltan-AlgD settles between PaToH-D and PaToH-Q. The only non-multilevel algorithm HYPE is considerably worse, with only 5.7% of solutions within a factor 2 of the best.

Figure 7.8 (right) shows relative running times for each instance. We categorize the algorithms into two groups. Algorithms in the first group, consisting of KaHyPar, hMetis and Zoltan-AlgD, invest substantial running time to aim for high-quality solutions. On the other hand, PaToH, Mondriaan and HYPE aim for fast running time and reasonable solution quality. The results show that while PaToH gives the best time-quality trade-off, KaHyPar-HFC is

the best algorithm for high-quality solutions, whereas KaHyPar-HFC-Eco offers the best time-quality trade-off in the first group. The geometric mean running times can be found in Table 7.3, which match the ranking suggested by the running time plot.

Failed Runs. The table additionally contains statistics on timeouts, imbalanced solutions (all seeds yielded an imbalanced partition), and errors. On one instance, HYPE output an infeasible objective value. Mondriaan reported a not further classified error on 4 instances. Regarding infeasible solutions, hMetis-K computes imbalanced partitions on 484 instances, Zoltan-AlgD on 99 and Mondriaan on 3 instances. To the best of our understanding, the hMetis-K paper [KK00] does not mention a constraint or penalty on maximum vertex weights during coarsening. Heavy vertices make it difficult for initial partitioning to find balanced partitions, which could be a possible explanation for these effects.

In a rerun of one seed of the experiments, where we disabled some expensive timing measurements, KaHyPar-HFC only times out on 60 instances and KaHyPar-HFC-Eco on 57. On the instances that still time out, initial partitioning often dominates the running time. This can be solved by implementing techniques to sparsify the coarsest hypergraph.

7.9 Parallel Push-Relabel on Hypergraphs

When developing Mt-KaHyPar, we set out to parallelize all of the powerful techniques in sequential KaHyPar. FlowCutter refinement is certainly the most powerful among the refinement techniques. We see two avenues for parallelization: scheduling two-way refinement on independent block pairs, and plugging a parallel maximum flow algorithm into FlowCutter. The scheduling is more straight-forward, so we focus on parallel maximum flow first.

Maximum flow algorithms are notoriously difficult to parallelize efficiently [SV82, BBS15, AS95, KÖ19] and often achieve only mediocre speedups. Push-relabel algorithms are the most amenable to parallelization [AS95, KÖ19, BBS15]. The synchronous push-relabel approach of Baumstark et al. [BBS15] is a recent algorithm that sticks closely to the sequential FIFO algorithm and thus exhibits good results. We pick their algorithm, since it does not seem to do much more work than the sequential algorithm, whereas asynchronous push-relabel can exhibit substantially more work when run with more threads [BBS15].

There is prior work by Lillis and Cheng [LLC95] on a push-relabel algorithm directly on the hypergraph, but it constructs all paths to push from a vertex u to another vertex v with a nested loop over the incident nets and then over the pins. This takes quadratic time in the net sizes and is thus impractical. For push-relabel it turns out that an on-hypergraph implementation is better than the Lawler expansion, but as opposed to Dinitz, it is better to intermediately push to nets instead of directly between vertices.

In the following, we first outline the synchronous algorithm on graphs, then describe a so far undocumented bug followed by our fix, and conclude with implementation details and intricacies of using FlowCutter with preflows. In this section the variable n is used for the number of nodes in the flow network, which is $|V| + 2|E|$ for the Lawler network of a

hypergraph $H = (V, E)$. For the majority of this section, we assume the perspective of plain graphs again, since the results carry over to hypergraphs via the Lawler network. Only in the sections on the efficient implementation, where we describe pseudocode and optimizations, we consider hypergraph specifics again.

7.9.1 Synchronous Parallel Push-Relabel

Algorithm 7.5: Synchronous Push Relabel Outline

```
1  active, next-active ← ∅
2  SaturateSourceArcs()      // add nodes with exc(v) > 0 and d(v) < n to next-active
3  while next-active not empty do
4      swap active and next-active
5      if work exceeds threshold
6          GlobalRelabeling()
7          work ← 0
8      next-active ← ∅
9      for u ∈ active do in parallel          // aggregate work performed with reduction
10         Discharge(u)                       // add pushed to nodes to next-active
11     for u ∈ active do in parallel
12         d(u) ← d'(u)
13         exc(u) += exc'(u)
14         exc'(u) ← 0
15     for u ∈ next-active do in parallel
16         exc(u) += exc'(u)
17         exc'(u) ← 0
```

Recall that an active node is a node v with $\text{exc}(v) > 0$. Let $e = (v, w)$ be an arc of v. If $\text{cap}(e) - f(e) > 0$ and $d(v) = d(w) + 1$, the arc is called admissible. The discharging operation repeatedly scans all arcs of an active node, pushing the maximum residual capacity on admissible arcs. After a scan, no admissible arcs are left, so the node is relabeled.

The synchronous parallel algorithm proceeds in rounds in which all active nodes are discharged in parallel. Algorithm 7.5 shows an outline. The flow is updated globally, but relabels and excess differences are hidden until the end of the round. More precisely, we maintain a second array d' with new distance labels and a second array exc' in which excess differences are aggregated using atomic fetch-and-add. There is no synchronization for d' because a node is only relabeled by one thread. After all nodes have been discharged, the distance labels d are updated to the local labels d' (see line 12) and the excess deltas are applied (see line 13 and 16). The discharging operations thus use the labels and excesses from the previous round. The distance condition in the admissibility definition becomes

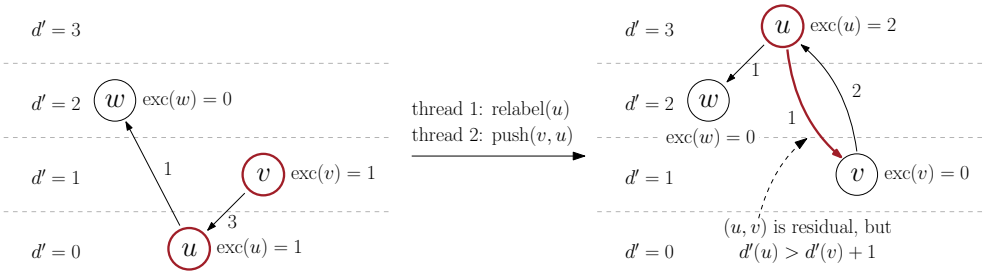

Figure 7.9: A conflict in the parallel discharge routine (adapted from Ref. [KÖ19]). The numbers on the arcs denote their residual capacities.

$d'(v) = d(w) + 1$, i.e., use the fresh label for the discharged node, but the old label for the target. We discuss more details on the discharging routine later.

Discharging rounds are repeated until there are no nodes with $\text{exc}(v) > 0$ and $d(v) < n$ left, at which point a minimum cut is available. The rounds are interleaved with global relabeling [CG97], after linear push and relabel work (line 6), using parallel reverse BFS, but no gap-relabeling heuristic is employed [CG97].

There is a race condition on the flow value of an arc, if both of its endpoints are active. To avoid concurrently pushing in both directions, a deterministic winning criterion on the old distance labels d is used to determine which direction to push, if both nodes are active. If $d(v) < d(w) - 1$ or $d(v) = d(w) + 1$ or $v < w$, then v wins the comparison and is allowed to push on (v, w), whereas w is not allowed to push on (w, v). The condition $v < w$ is used as tie-breaker for the case $d(v) = d(w)$. The intention behind this criterion is that one node may be relabeled past the other, to ensure progress. If an arc (v, w) cannot be pushed due to the criterion, the discharge terminates after the current scan. In this case v still has an outgoing admissible arc, and thus may not be relabeled in this round, and there are no further admissible arcs left to push.

7.9.2 A Bug in the Synchronous Algorithm

Unfortunately, there is a still a bug in the above algorithm. The parallel discharge routine does not protect against push-relabel conflicts [KÖ19] as illustrated in Figure 7.9. In particular the winning criterion does not help. A node u may be relabeled too high if it concurrently receives flow from a residual arc (v, u) with $d'(v) = d(u) + 1$. The arc (u, v) may not be observed as residual yet, and thus u may set its new label $d'(u) > d'(v) + 1$, violating correctness of the distance labels. We noticed the bug when the algorithm terminated prematurely with non-maximal flow, even though all excess nodes had label $> n$.

Our fix is to collect mislabeled excess nodes during global relabeling. The global relabeling repairs the wrong distance labels. A node is mislabeled, if it is visited by the relabeling BFS, has excess and was not added to the active nodes for the next discharging round. When the

regular termination criterion is triggered, we run global relabeling, and restart the main loop if new active nodes are found.

The additional work is already accounted for, because we need to extract the sink-side cut anyways. Hence, we already set the markers for the sink-side when running this extra global relabeling, since we presume that we can terminate most likely. In practice, this bug occurs very rarely (which makes it hard to find), and in fact none of the termination check relabel runs find new active nodes, because they are already found during regular runs, which justifies our presumption.

We also consider a second independent fix where we prevent nodes from being pushed to after being relabeled, or from being relabeled after being pushed to; whichever comes first. We implement this with an atomic compare-and-swap on a state variable for the node. The compare-and-swap must succeed before either operation is allowed to execute. The possible states are not modified, relabeled, or pushed. We reset the states in the update loop for levels and excess values. Note that relabeled nodes may be relabeled again, and nodes that were pushed to may be pushed to again, just not the other operation.

We use the first fix, but investigate the difference between these two approaches in Section 7.11, showing that on our instances, there is no difference in performance. As mentioned, the bug occurs rarely, which explains this result.

7.9.3 Optimizations

In this section we discuss two optimizations that substantially improve performance on our instances in practice. The first is particular to hypergraphs and relevant in both sequential and parallel settings, whereas the second is also relevant for plain graphs but is presumably much more important for the parallel setting than the sequential.

Restricting Capacities. Recall that only bridge arcs $(e_{\text{in}}, e_{\text{out}})$ have finite capacity $(\omega(e))$ in the Lawler network. Since $(e_{\text{in}}, e_{\text{out}})$ is the only outgoing arc of e_{in} with non-zero capacity, the flow (but not preflow) on arcs (u, e_{in}) is also bounded by $\omega(e)$. Adding these capacities during the preflow stage is a trivial optimization, but it reduces running time for one flow computation on our largest instance from over two hours to 14 seconds, when using 16 cores! That is an improvement by a factor of more than 500. It also boosts the available parallel work, since hypernodes are not immediately relieved of all their excess.

Without this optimization the minimum cut contains only bridge arcs. Now it may contain arcs of the form (u, e_{in}). This matters when tracking cut hyperedges (for collecting piercing candidates), which are detected by checking if e_{in} and e_{out} are on different sides. Therefore, we do not check this capacity and thus visit e_{in} nodes during the forward residual BFSs to derive the source-side cut.

Avoid Pushing Flow Back. Once the correct flow value is found, the algorithm could terminate in theory. This is often achieved in very few discharging rounds ($< 1\%$). At this

point the full amount of flow has reached the sink, the cut is saturated, but there is still excess that can be pushed back. Furthermore, we observed that the number of active nodes roughly follows a power law distribution: many active nodes in the beginning, decreasing rapidly until only a few active nodes remain in later rounds. This limits the available parallelism, which is why we would like to terminate in as few rounds as possible.

We terminate once all nodes with $\text{exc}(u) > 0$ have $d(u) \geq n$, which is most often detected by global relabeling. At this stage only little work is performed per round, and thus it takes many rounds to trigger global relabeling. We perform additional relabeling, if the flow value has not changed for some rounds (500), and only few active nodes (< 1500) were available in each. In case we were wrong and have to continue pushing flow, we do not perform the additional relabeling again. work is still performed.

7.9.4 Intricacies with Preflows and FlowCutter

In this section, we discuss (some unexpected) challenges we faced during the implementation that arose from using FlowCutter with preflows. The major difference to actual flows is that there are nodes with positive excess left.

Source-Side Cut. A maximum preflow only yields a sink-side cut via the reverse residual BFS, but we also need the source-side cut. We can run flow decomposition [CG97] to push excess back to the source, to obtain an actual flow. However, flow decomposition is difficult to parallelize [BBS15]. Instead, we initialize the forward residual BFS with all non-sink excess nodes. This finds the reverse paths that carry flow from the source to the excess nodes, which is what we need.

Sink-Side Piercing. When transforming a node with positive excess to a sink, its excess must be added to the flow value. This only happens when piercing, as sink-side nodes have no excess, if they are not sinks yet.

Maintain Distance Labels. Between two consecutive flow computations, we want to reuse the distance labels to avoid re-initialization overheads. However, as the labels are a lower bound on the distance from the sink, piercing on the sink side invalidates the labels. Additionally, no new excess nodes are created. In this case, we run global relabeling to fix the labels and collect the existing excess nodes, before starting the main discharge loop. When piercing on the source side the labels remain valid and new excesses are created by saturating the arcs of the new sources. The new excesses are added to the active nodes and we do not run an additional global relabeling. The existing excess nodes are collected during regular global relabel runs; at the latest for the termination check.

7.9.5 Hypergraph Implementation

In this section, we discuss the on-hypergraph implementation of parallel push-relabel, focusing on the discharging routines. We implement three separate discharging routines for in-nodes, out-nodes and hypernodes since they differ in the way residual capacities are calculated, dispatching to the correct one in the loop over active nodes (line 10, Algorithm 7.5). As opposed to the Dinitz implementation, we have to store in-nodes and out-nodes as actual entities in the data structures, since push and relabel are local operations.

Algorithm 7.6: TryPush

```
   Input: Arc (u, v), residual capacity Δ, excess g, potential new level l, skipped
 1 if Δ > 0
 2     if d'(u) = d(v) + 1
 3         win ← d(u) = d(v) + 1 or d(u) < d(v) − 1 or (d(u) = d(v) and u < v)
 4         if exc(v) > 0 and not win                          // only check if v is active
 5             skipped ← true
 6         else
 7             next-active.tryAdd(v)                          // deduplicate
 8             g −= Δ
 9             exc'(v) +=_atomic Δ
10             return true
11     else if d'(u) ≤ d(v)
12         l ← min(l, d(v))
13 return false
```

For hypernodes (Algorithm 7.7), we iterate over the incident nets and try to push flow to the corresponding in-nodes (line 6) and out-nodes (line 12). For in-nodes (Algorithm 7.8) and out-nodes (Algorithm 7.9), we iterate over the pins of the net and try to push to them (line 10 and 6 respectively), as well as over the bridge arc (line 5 and 10 respectively). We make a local copy g of the excess of the discharged node (line 1), so we can use the $\text{exc}(u)$ values to detect if a node is active, as they are not modified during the round. Updates to the excess are applied to $\text{exc}'(u)$ at the end of each discharge (line 19 for Algorithm 7.7, omitted for the others), which are then applied to exc after all active nodes have been discharged. Note again that in the discharging routines $n = |V| + 2|E|$ denotes the number of nodes in the Lawler network, not $|V|$.

Algorithm 7.6 shows the `TryPush` routine that is run on each outgoing arc. It takes the amount Δ to be pushed (residual capacity), the origin node u and target node v, and returns whether the push is successful (distances fit, $\Delta > 0$ and arc is won). If $\Delta > 0$ but the distances do not fit (line 11,12), we record the label of v as a candidate for relabeling. If the distances do fit (line 2), v is active and the arc is not won (line 3,4), we cannot push on it. Thus, we record that the arc is skipped, such that we terminate the discharge after the current scan.

Algorithm 7.7: Discharge Hypernode

```
   Input: Hypernode u
1  g ← exc(u), n ← |V| + 2|E|, skipped ← false
2  while g > 0 and d'(u) < n and not skipped do
3      l ← n
4      for e ∈ I(u) do
5          Δ = min(g, cap(e) − f(u, e_i))          // restricted capacity optimization
6          if TryPush((u, e_i), Δ, g, l, skipped)
7              f(u, e_i) += Δ
8              if g = 0 break
9      if g = 0 break
10     for e ∈ I(u) do
11         Δ = f(e_o, u)                           // exc(u) ≥ f(e_o, u)
12         if TryPush((u, e_o), Δ, g, l, skipped)
13             f(e_o, u) −= Δ                      // push back to out-node
14             if g = 0 break
15     if g > 0 and not skipped                    // relabel
16         d'(u) ← l + 1
17 if g > 0 and d'(u) < n
18     next-active.tryAdd(u)
19 exc'(u) −=_atomic (exc(u) − g)
```

Algorithm 7.8: Discharge In-Node

```
   Input: Net e
1  g ← exc(e_i), n ← |V| + 2|E|, skipped ← false
2  while g > 0 and d'(e_i) < n and not skipped do
3      l ← n
4      Δ_b ← min(g, cap(e) − f(e_i, e_o)
5      if TryPush((e_i, e_o), Δ, g, l, skipped)
6          f(e_i, e_o) += Δ_b
7          if g = 0 break
8      for v ∈ e do
9          Δ = f(v, e_i)
10         if TryPush((e_i, v), Δ, g, l, skipped)
11             f(v, e_i) −= Δ
12             if g = 0 break
13     if g > 0 and not skipped                    // relabel
14         d'(e_i) ← l + 1
15 ...
```

Algorithm 7.9: Discharge Out-Node

```
   Input: Net e
1  g ← exc(e_o), n ← |V| + 2|E|, skipped ← false
2  while g > 0 and d'(e_o) < n and not skipped do
3      l ← n
4      for v ∈ e do
5          Δ = f(e_o, v)
6          if TryPush((e_o, v), Δ, g, l, skipped)
7              f(e_o, v) += Δ
8              if g = 0 break
9      Δ_b ← min(g, f(e_i, e_o))
10     if TryPush((e_o, e_i), Δ, g, l, skipped)
11         f(e_i, e_o) −= Δ_b
12         if g = 0 break
13     if g > 0 and not skipped                          // relabel
14         d'(e_o) ← l + 1
15 ...
```

We have to terminate because we cannot relabel u, as the arc (u, v) is admissible. In case of success (line 6-10), we update the local excess copy g of u, $\text{exc}'(v)$ and add v to the active nodes in the next round. To avoid duplicate insertion, we use atomically updated timestamps. The flow value of the arc is updated outside the function since we store the flow only in one direction, such that we either need to add or subtract Δ.

We store $f(u, e_i), f(e_o, u), f(e_i, e_o)$, i.e., the values at the regular arcs of the Lawler network, but not $f(e_i, u), f(u, e_o), f(e_o, e_i)$ (the corresponding negatives) since we do not have the arcs constructed in memory for pointers to their back-arcs. Instead, we have back-pointers at pin-list entries pointing to the corresponding entry in the incident nets list of the pin. With these, we can retrieve the required flow values $f(u, e_i), f(e_o, u)$ when discharging in-nodes and out-nodes, respectively. For arcs of the form $(u, e_i), (e_o, u), (e_i, e_o)$ we add Δ, and for the reverse arcs $(e_i, u), (u, e_o), (e_o, e_i)$ we subtract Δ, as can be seen in the discharging routines.

7.10 Bulk Piercing

In this section, we propose a simple optimization that enables FlowCutter to progress faster towards a balanced bipartition by piercing more aggressively. This is important since the incremental flow problems at later stages often lack good parallelism, since only little flow is augmented. The proposed approach is rather brute force, but it works well enough in practice. It can be seen as a hybrid between the rescaling in KaHyPar-MF [HSS19] and regular piercing, transitioning carefully between them. Regarding partition quality, our experiments

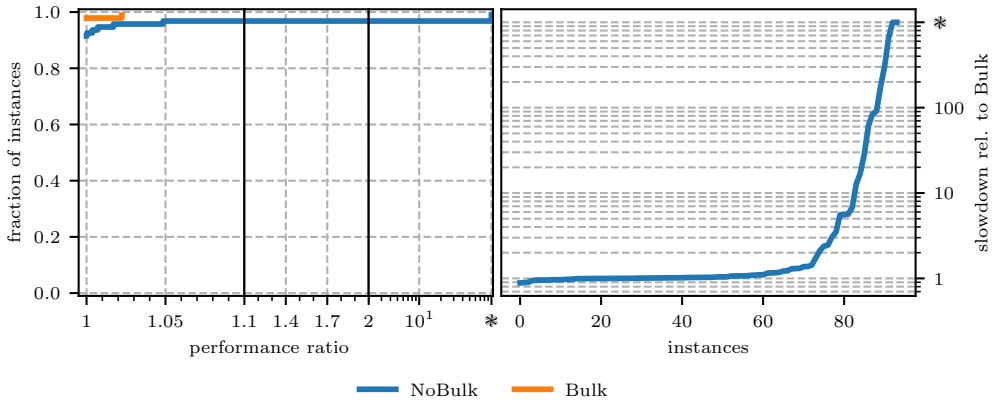

Figure 7.10: Effects of bulk piercing on partition quality and running time. Measured on extracted flow hypergraphs of set B with 32 cores.

show there is no difference between regular piercing and the new *bulk piercing*, but bulk piercing is substantially faster.

The performance of FlowCutter is much better than the $\mathcal{O}(\zeta m)$ bound in practice, as the first cut is often close to the final cut. Only few augmenting iterations are needed and much less than $\mathcal{O}(m)$ work is spent per flow unit [GHW19a], with most work spent on the initial flow. Still, the flow augmented per iteration at later stages is often small: at most the capacity of arcs incident to the piercing node. On large instances, we observed that the number of required iterations also increases substantially, which further inhibits parallelism. We accelerate convergence by piercing multiple nodes per iteration, but only if we cannot avoid augmenting paths. This increases the amount of augmentation work per iteration (good for parallelism) and reduces the number of iterations.

To ensure a poly-log iteration bound, we set a geometrically shrinking goal of weight to add to each side per iteration. The initial goal for the source side is set to $\beta(\frac{c(V)}{2} - c(S))$, and analogously the initial goal for the sink side is $\beta(\frac{c(V)}{2} - c(T))$. The term $\frac{c(V)}{2} - c(S)$ is the weight to add for perfect balance. The parameter $\beta \in (0, 1)$ is a geometric shrinking factor that is multiplied with the initial weight goal in each iteration. If a goal is not met, its remainder is added to next iteration's goal. More precisely, in the j-th iteration, we add $\beta^j(\frac{c(V)}{2} - c(S))$ to the current weight goal.

We track the average weight added per node and from this estimate the number of piercing nodes needed to match the current goal. To boost measurement accuracy, we pierce only one node for the first few rounds, and then enable bulk piercing. The sides have distinct measurements and goals, so that we do not pierce too aggressively when the smaller side flips. This scheme (with $\beta = 0.55$, and 5 regular iterations before bulk) reduces running time on our largest instances from beyond two hours (time limit) to less than 10 minutes, when

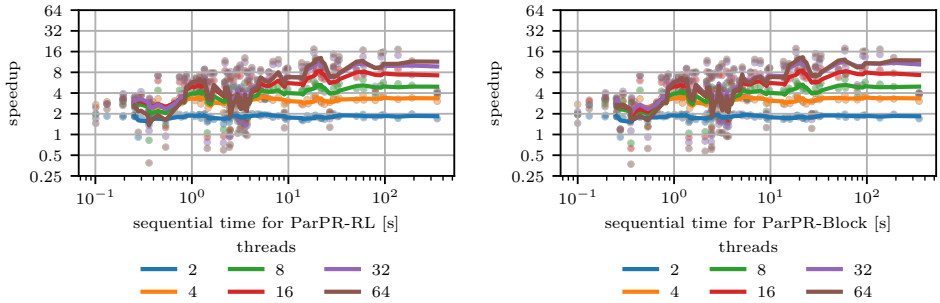

Figure 7.11: Speedups for flow computation with parallel push relabel on set B. On the left is the variant that picks up mislabeled excess nodes during global relabeling. On the right is the variant that blocks relabeled nodes from being pushed to and vice versa. The window size for the rolling geometric means is set to 5.

run on a single core.

Figure 7.10 shows performance profiles of FlowCutter with and without bulk piercing, as well as running times relative to bulk piercing. We constructed the instances by extracting flow hypergraphs from bipartitions of the instances in set B. The measurements are done on 32 cores. We see no quality difference, but the bulk piercing version can solve two instances that previously timed out, and is much faster on some instances that take very long with the non-bulk version. The geometric mean running times are 1.38s with bulk piercing and 2.79s without bulk piercing.

Additionally, if we can avoid augmenting paths, and assigning all $V \setminus (S_r \cup T_r)$ to the smaller side still keeps this side as the smaller one, we assign them all straight away. This allows us to skip having to repeatedly step through the piercing and side-growing routines (with startup overheads for parallel BFS), even though we will ultimately assign them all anyways.

7.11 Flow Algorithm Experiments

Speedups. In Figure 7.11 we report speedups for the two parallel push-relabel variants, with instances sorted by sequential time. The instances are the aforementioned extracted flow hypergraphs. The algorithms differ in the way the discovered bug is addressed: ParPR-RL is the variant that runs additional relabeling, ParPR-Block is the variant that blocks relabels or pushes. Both variants achieve essentially the same speedups. With 2 and 4 cores near-perfect speedups are recorded. With 8 cores the speedups are around 5-6 for the larger instances, and similarly for 16 cores the speedups are around 8. With 32 and 64 cores we reach up to 16, but on the smaller instances we also observe slow-downs.

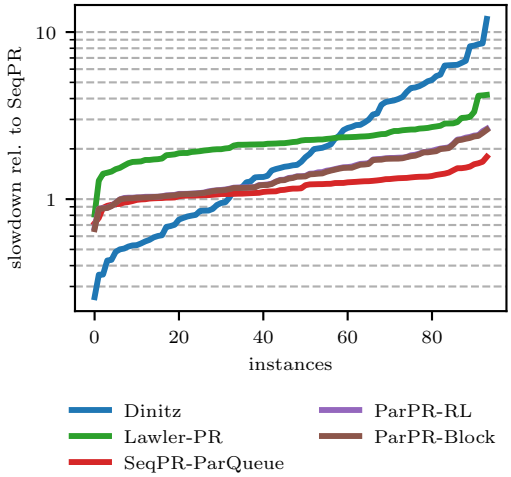

Figure 7.12: Running time comparison of flow algorithms for plain flow computation on set B. All algorithms were run sequentially.

Running Times. In Figure 7.12 we present running times relative to a sequential implementation of FIFO push-relabel, which uses a simple queue and does not need synchronization mechanisms. In addition to the two parallel push-relabel variants (run on one core), we consider an implementation of Dinitz, sequential push-relabel with the parallel queue data structures, and a push-relabel variant directly on the Lawler network. We can see that the Lawler variant is a factor 2-3 slower, i.e., the implementation directly on the hypergraph is worth it, but the savings are not as drastic as they were for Dinitz. Dinitz is faster than push-relabel on about $\frac{1}{3}$ of the instances, and slower on about $\frac{2}{3}$. Furthermore, we see that using the parallel data structures comes at a cost, though always below a factor 2. Finally, the two parallel variants exhibit the same running time (their curves are almost identical). We see that the synchronization mechanisms incur a further slowdown.

2-way FlowCutter Refinement Speedups. After looking at speedups for the flow algorithms, we now look at a full execution of FlowCutter with ParPR-RL as parallel flow algorithm, run on the same instances. With 2 and 4 cores we still observe near-perfect speedups, but starting at 8 cores the speedups are not as good as the plain flow algorithm. This is not surprising since even with bulk piercing, there are some iterations that only have small flow augmentation work. Notably using 64 cores over 32 seems to incur a slow-down except for 7 instances where we still reach a speedup of 16.

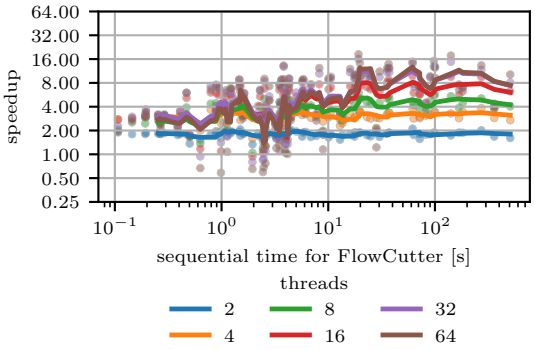

Figure 7.13: Speedups for FlowCutter refinement on set B. The window size for the rolling geometric means is set to 5.

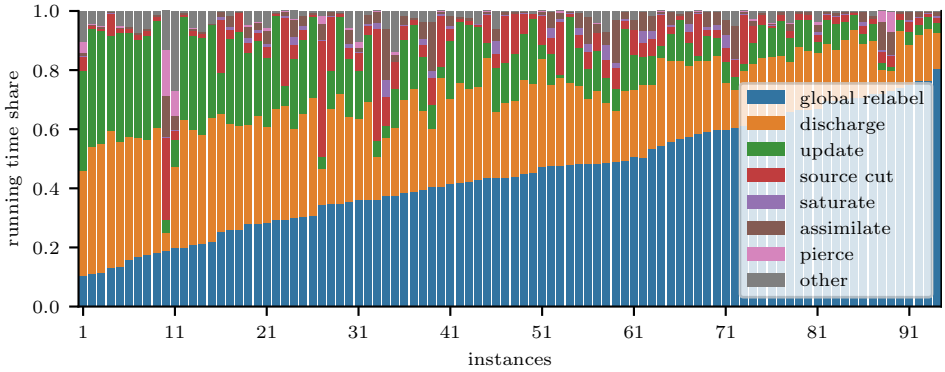

Figure 7.14: Fractions of time spent in each of the components of FlowCutter refinement plotted per instance.

Running Time Shares. In Figure 7.14 we break the running time of FlowCutter refinement down into its components: global relabeling, discharging, applying excess and level updates, and saturating source nodes, which are part of the flow algorithm, as well as piercing, deriving the source-side cut, and determining the smaller side and settling nodes (assimilate). Deriving the sink-side cut is accounted for in the global relabeling time already. These measurements were done on 32 cores of machine B on the extracted flow hypergraphs of set B.

Global relabeling accounts for the largest shares, which is partially a configuration choice and partially a side effect of including the time to derive sink-side cuts. In preliminary experiments we determined that global relabeling is the best way to accelerate the algorithm. It was almost impossible to perform too much of it. As expected, discharging makes up the

second largest share, followed by applying excess and level updates. Piercing and assimilating are negligible, which shows that the bottleneck is indeed the flow algorithm.

7.12 Parallel Scheduling

So far, we have only parallelized the refinement on bipartitions, but for integration in a k-way partitioner such as Mt-KaHyPar, we need to schedule the 2-way flow refinement on different block pairs. This is a much easier avenue for parallelization than the flow algorithm since the different refinement calls are completely (or largely) independent. The goal is thus to maximize the number of block pairs we can schedule at the same time, without compromising partition quality.

A simple scheme would be to schedule non-overlapping block pairs in the quotient graph $\mathcal{Q}$ in parallel, i.e., a maximal matching, in order to maximize the utilized parallelism. We can either synchronize after one matching, construct a new one and restart, or dynamically maintain it. This scheme ensures that the different refinements are completely independent and do not interfere with one another. This means that all of the calculated gains are correct, and no accidental balance violations can occur.

However, there are at most $\frac{k}{2}$ non-overlapping block pairs. At later stages the number is much smaller even, since there are fewer active blocks or, in some cases, just a few hub blocks and many low-degree blocks in $\mathcal{Q}$. This restricts the available parallelism too much, in particular for small k. Therefore, we remove the restriction to non-overlapping block pairs, but restrict the number of concurrent refinements to reduce the interference between overlapping searches.

Apply Moves. Let $M = \{(u, s, t) \mid u \in V_i \cup V_j \text{ and } (s = i, t = j \text{ or } s = j, t = i)\}$ denote the set of moves output by flow-based refinement on V_i, V_j. The semantics are that vertex u is moved from block s to t. There are three issues that can occur due to the removed restriction.

1. a vertex of M may no longer be in the block expected by M
2. the calculated gain Δ_{exp} does not equal the actual gain Δ when applied to the partition
3. applying the moves may make the partition imbalanced

To mitigate these issues, we apply moves to the global partition only while holding a global lock, see lines 7-16 in Algorithm 7.10. This does not impact performance, since the bulk of the work is performed outside the lock (line 3-5). First, we remove all vertices from M that are not in the expected block (line 8). Then, we calculate the block weights if all remaining moves were applied (line 9). If balanced (line 10), we apply the remaining moves to the global partition (line 11). During this, we use attributed gains to calculate the actual gain Δ. If $\Delta < 0$, we accidentally worsened the partition, and thus revert the moves (line 13). Otherwise, we found an improvement and mark i and j as active for further refinement in the next round.

Algorithm 7.10: Parallel Flow-Based k-way Refinement

```
   Input: Hypergraph H = (V, E, c, ω), k-way partition Π of H
1  Q ← ConstructQuotientGraph(H, Π)
2  while ∃ active (V_i, V_j) ∈ Q do in parallel
3      B := B_i ∪ B_j ← GrowRegion(H, V_i, V_j)            // B_i ⊆ V_i, B_j ⊆ V_j
4      (N, s, t) ← ConstructFlowHypergraph(H, B)
5      (M, Δ_exp) ← FlowCutterRefinement(N, s, t)
6      if Δ_exp ≥ 0                                        // potential improvement
7          lock()
8          M ← {(u, s, t) ∈ M | Π(u) = s}
9          a ← c({u | (u, j, i) ∈ M}) − c({u | (u, i, j) ∈ M})
10         if c(V_i) + a ≤ L_max and c(V_j) − a ≤ L_max
11             Δ ← ApplyMoves(H, Π, M)
12             if Δ < 0                                    // accidentally worsened
13                 RevertMoves(H, Π, M)
14             else                                        // verified improvement
15                 mark V_i and V_j as active
16         unlock()
```

Block Pair Queue. In contrast to Algorithm 7.4, we partially removed the concept of rounds from the scheduler in Algorithm 7.10, in order to eliminate the synchronization point at the end of a round. Instead, we use a FIFO queue that contains the active block pairs, which threads poll from until empty, to implement the loop in line 2. The queue in our implementation is concurrent, but using a lock would not be harmful here.

For the sake of determining which block pairs to add to the queue, we still use the rounds concept. Each round is associated with an array of size k to mark blocks that become active in the next round. If we find a verified improvement (line 14,15), we mark the two involved blocks as active for the next round, and push them into the FIFO queue if not yet contained, together with their adjacent blocks in $\mathcal{Q}$. The rounds are stored interleaved in the queue, and each block pair in the queue is associated with a round number. A round ends once all of its block pairs have been processed and all prior rounds have ended. If the relative improvement at the end of a round is less than 0.1%, we immediately terminate the entire algorithm, even if refinements from later rounds are running.

Restrict Concurrent Searches. Removing the restriction to non-overlapping block pairs introduces a lot of parallelism, but also the potential for interference between the searches. Therefore, we manually limit the number of concurrent searches to $\tau \cdot k$, where τ is a parameter to tune. Lower values of τ can reduce conflicts, but restrict parallelism, and thus put more emphasis on obtaining good speedups in 2-way refinement. Higher values of τ increase parallelism, but cause more interference. In Section 7.13.4, we determine $\tau = 1$ as a

good choice. This seems like just a factor 2 more than the matching, however at later stages there are much less than $\frac{k}{2}$ block pairs in the matching. In the experiments, we show that we do observe interference between searches with this approach (sometimes heavy interference), however the partition quality is fortunately not negatively affected when adding more cores.

Integration in Mt-KaHyPar. In previous chapters, we introduced two variants of Mt-KaHyPar: the default version Mt-KaHyPar-D and the n-level version Mt-KaHyPar-Q. We equip both with flow-based refinement to obtain the configurations Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F. For Mt-KaHyPar-D, we first run label propagation refinement, then FM, then flow-based refinement. Recall that for Mt-KaHyPar-Q, we use the synchronization points at which we restore identical nets to run further refinement. Here, we first run FM and then flow-based refinement. Running flow-based refinement within regular n-level is impractical since it is not localized. We run all refinement algorithms on each level multiple times in combination and stop if the relative improvement is less than 0.25%.

7.13 Parallel k-way Refinement Experiments

We now turn to evaluating our parallel flow-based refinement on k-way partitions with both parallelism sources. As in Chapter 6, the experiments on set B use $k \in \{2, 4, 8, 16, 32, 64\}$ if run on 64 cores, and $k \in \{2, 8, 16, 64\}$ if run on different numbers of cores. We look at the running time shares of the different phases, analyze the refinement behavior, and conduct scalability experiments. Because flow-based refinement is employed in the same way in Mt-KaHyPar-Q-F as in Mt-KaHyPar-D-F, we focus on Mt-KaHyPar-D-F in this part. Subsequently we perform the horse-race comparisons, first with sequential algorithms and other Mt-KaHyPar variants on set A, then with fast and parallel algorithms on set B, this time involving Mt-KaHyPar-Q-F as well.

7.13.1 Running Time Shares

Figure 7.15 shows that flow-based refinement makes up the biggest part of the running time of Mt-KaHyPar-D-F, by a huge margin. For each instance and phase we plot the fraction of the total time spent in that phase. The instances are sorted by the fraction for flow-based refinement (brown).

In Figure 7.16 we break flow-based refinement down into four separate phases: selecting movable vertices (grow region, BFS), assembling the flow hypergraph, running FlowCutter, and applying the calculated moves. We further plot the behavior for the different numbers of threads 1, 4, 16, 64 as separate box and scatter plots. We sum up the measured times of each phase and plot the fraction of the total sum. The algorithm works on different block pairs in parallel, therefore we measure the execution times of each block pair and sum them up. As a disclaimer, the timings are therefore not guaranteed to be an accurate representation of

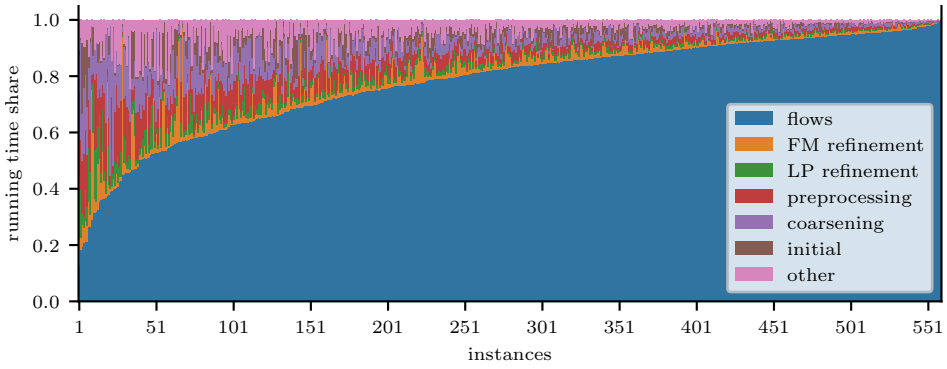

Figure 7.15: Fractions of time spent in each of the components of Mt-KaHyPar-D-F, plotted per instance.

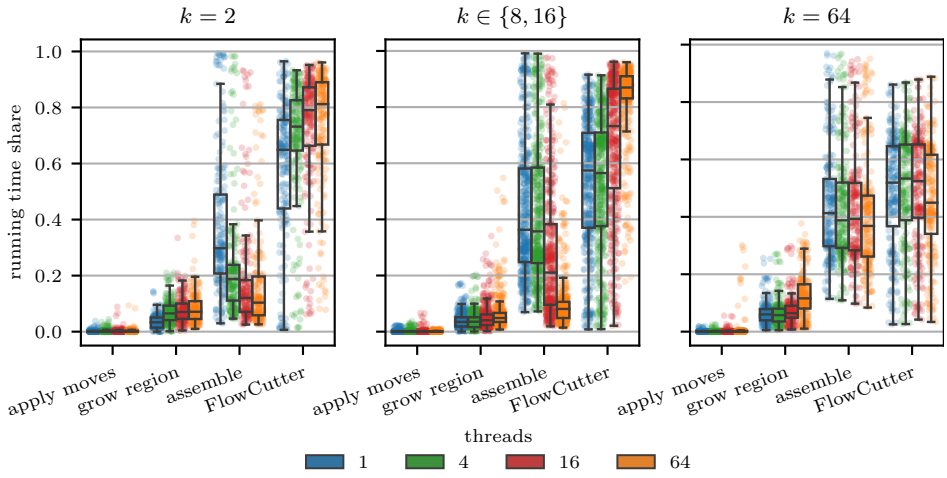

Figure 7.16: Fractions of time spent in each of the phases of flow-based refinement, separated by k and the number of threads used.

the overall work, except when $k = 2$ or when we use 1 core, because the flow algorithm is parallelized as well.

Applying moves is usually negligible as expected. Only for $k = 64$ and 64 cores we observe some outliers in the 20-30% area. Region growing takes around 10% for $k = 2, 64$, and 5% for $k = 8, 16$, though the share for 64 cores is slightly higher because the BFS is not parallelized, as it is not the bottleneck. The actual bottleneck is FlowCutter (and therein computing flows), as would be expected. The share is the largest for $k = 2$ because computing flows is the

only component with super-linear running time (in practice as well) and smaller values of k yield larger flow hypergraphs. Therefore, the share decreases for $k = 8, 16$ and even more for $k = 64$, where assembling the flow hypergraph makes up around 40% of the running time. With increased thread count, the share of FlowCutter increases. As we observe this for $k = 2$, it is not a side-effect of the measurements being inaccurate, but a matter of worse speedups than assembly, which is much easier to parallelize.

7.13.2 Scalability

In Figure 7.17 we plot instance-wise self-relative speedups of parallel flow-based refinement and Mt-KaHyPar-D-F as a whole (bottom right), with rolling geometric means as curves. The plot for flow-based refinement is split into three segments $k = 2$ (top left), where only the flow algorithm is parallelized, $k = 64$ (bottom left) where only the scheduling parallelism is used, and $k \in \{8, 16\}$ (top right), where both parallelism sources are used (depending on the number of cores). For Mt-KaHyPar-D-F we use window size 50, for the other three plots we use window size 10 due to the reduced number of instances.

The overall geometric mean speedup of Mt-KaHyPar-D-F is 3.1 on 4 cores, 7.4 on 16 cores, and 10.62 on 64 cores. If we only consider instances with a single-threaded running time greater than 100s, we achieve a geometric mean speedup of 14.5 on 64 cores.

For $k = 2$ we see that the speedups are largely comparable with those in Figure 7.11 and 7.13. The speedups exhibit very large variance, but the overall trend indicated by the rolling geometric mean curves, is that we achieve better speedups if there is sufficient work as indicated by the sequential running time. Looking back at Figure 7.16, we recall that there were some instances where assembling the flow hypergraph takes particularly long on one thread. These are the outliers in the top left plot, e.g., `nlpkkt200` with a speedup of 80.05 on 64 cores.

For $k = 64$ we achieve a geometric mean speedup of 18.48 on 64 cores. In this case, all parallelism is leveraged in the scheduler, and none in the flow algorithm, which explains why we obtain more reliable speedups than for all other k. As the outer parallel construct, the scheduler is the more amenable parallelism source. For $k \in \{8, 16\}$, both parallelism sources are used. The speedups are slightly better than for $k = 2$, but not as robust or reliable as for $k = 64$.

In Figure 7.18 we plot the partition quality of Mt-KaHyPar-D-F with 1, 4, 16 and 64 cores. We see that using more cores does not adversely affect the partition quality.

7.13.3 Refinement Analysis

Next, we look at the behavior of flow-based refinement under interference from concurrent searches. In Figure 7.19 we plot the fraction of block pairs for which FlowCutter claims an improvement, relative to all scheduled block pairs. Out of these, we plot the fraction of actual improvements (verified while applying moves) and searches with zero gain, incorrect gain value, whether this event lead to a revert, or whether accidental balance violations lead

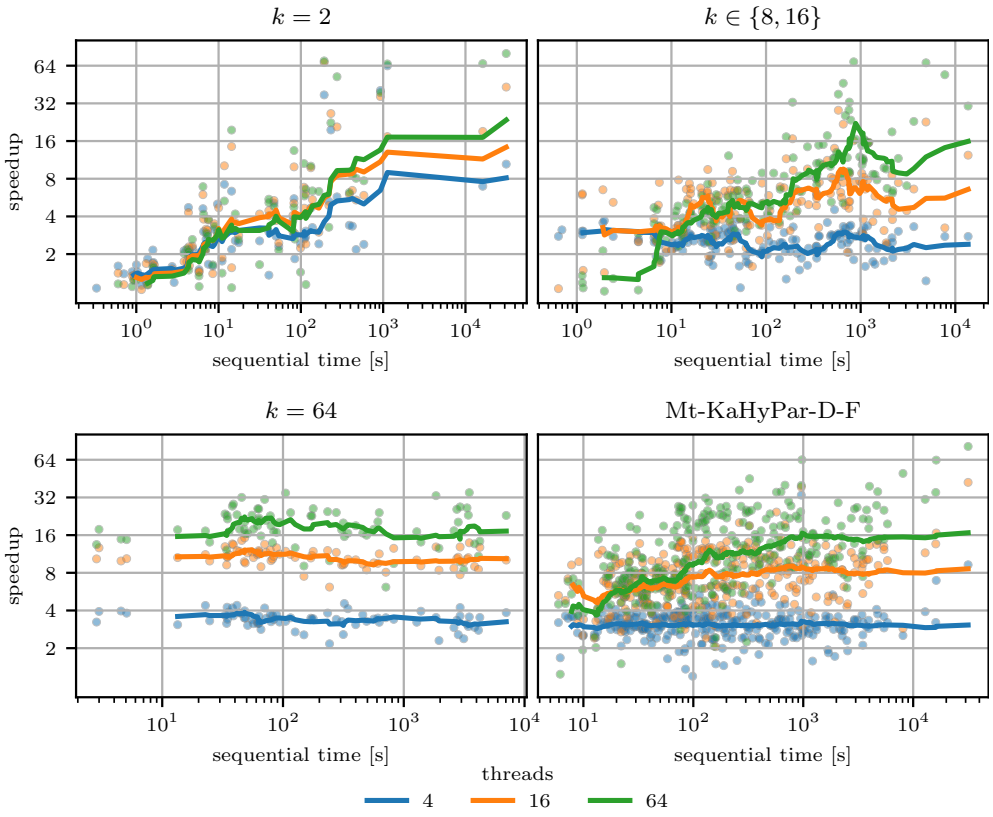

Figure 7.17: Speedups for flow-based refinement with different values of k (because different parallelism sources are used), as well as Mt-KaHyPar-D-F in total for all k (bottom right). The x-axis shows the sequential time in seconds, the y-axis the speedup. The lines are rolling geometric means (window size 50 for Mt-KaHyPar-D-F, 10 for the per k plots) of the per-instance speedups (scatter).

to a revert. These events are shown per instance and per number of threads. Around 35% of scheduled block pairs claim an improvement, and the number increases when adding more threads, which is the first sign of interference. With 1 core these are all successful, with 4 cores still around 95%, 16 cores around 90%, and 64 cores around 85%, though the outliers increase. The number of zero gain improvements are quite steady across different thread counts, but surprisingly high at around 40% of claimed improvements. In this case the balance was improved, which in turn can lead to positive gain improvements in later rounds or the other refinement algorithms.

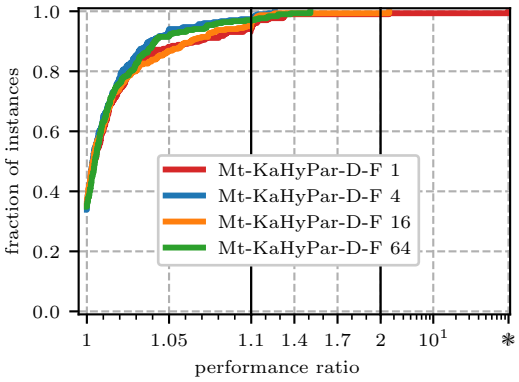

Figure 7.18: Partition quality with increasing number of threads. We can see that there is no quality penalty incurred by using more cores.

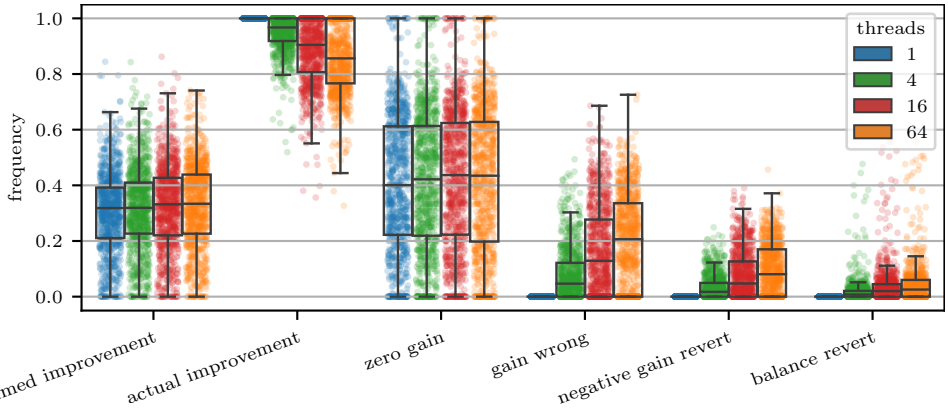

Figure 7.19: Flow-based refinement statistics per instance and different numbers of threads.

The number of searches with incorrect gains increases rapidly with more cores, reaching around 20% for 64 cores. Still, this translates to a revert due to negative gains in about half of the cases. Notably, for $k = 2$ the claimed gain or balance is never wrong, which we see from clusters at 0, even when 64 cores are used. Furthermore, for the other k, balance reverts are more rare than negative gain reverts.

The data for Figure 7.19 does not contain information how these statistics affect the actual gain, which we care about the most. In order to save running time, we only repeated this experiment with 64 cores, which we show in Figure 7.20. As a sanity check, the event statistics look the same as in Figure 7.19. The rightmost entry shows that the gain is affected. We plot the sum of actual gains divided by the sum of expected gains, showing that we

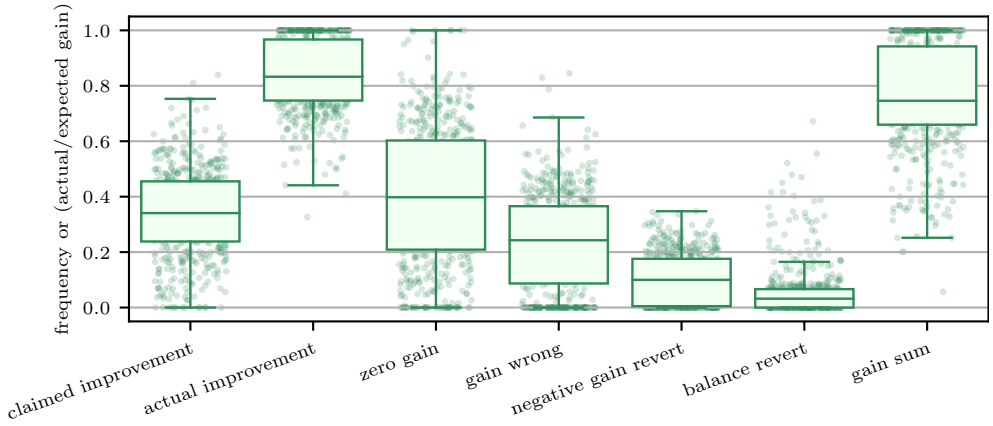

Figure 7.20: Flow-based refinement statistics for 64 cores, with tracked gain values.

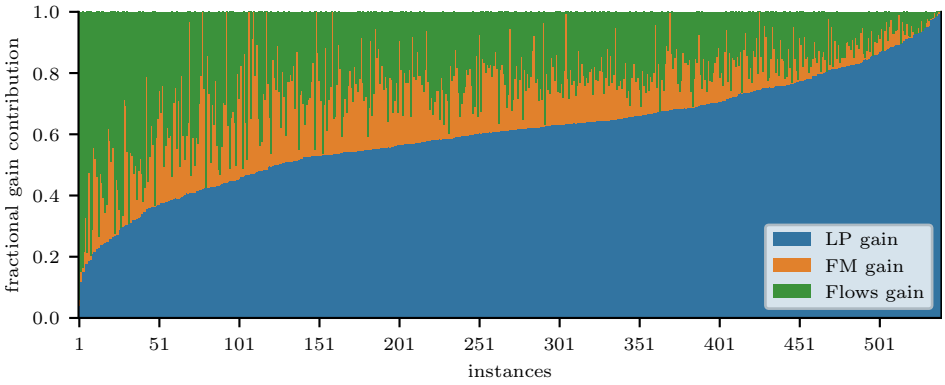

Figure 7.21: Contribution to the overall gain made by each refinement algorithm, plotted per instance.

achieve around 75-80% of the expected gain. As we saw earlier in Figure 7.18, this does not actually yield worse partition quality overall, because there may be additional repetitions, or the other refinement algorithms are able to compensate.

Finally, in Figure 7.21 we show how much each of the three refinement algorithms contributed to the overall gain (difference from initial partition to final partition). Again, label propagation contributes the majority, since it is run first and can perform the easy improvements, but the share is lower than without flow-based refinement, confer Figure 4.5. Flow-based refinement and FM have similar shares, with flows coming out ahead by a small margin.

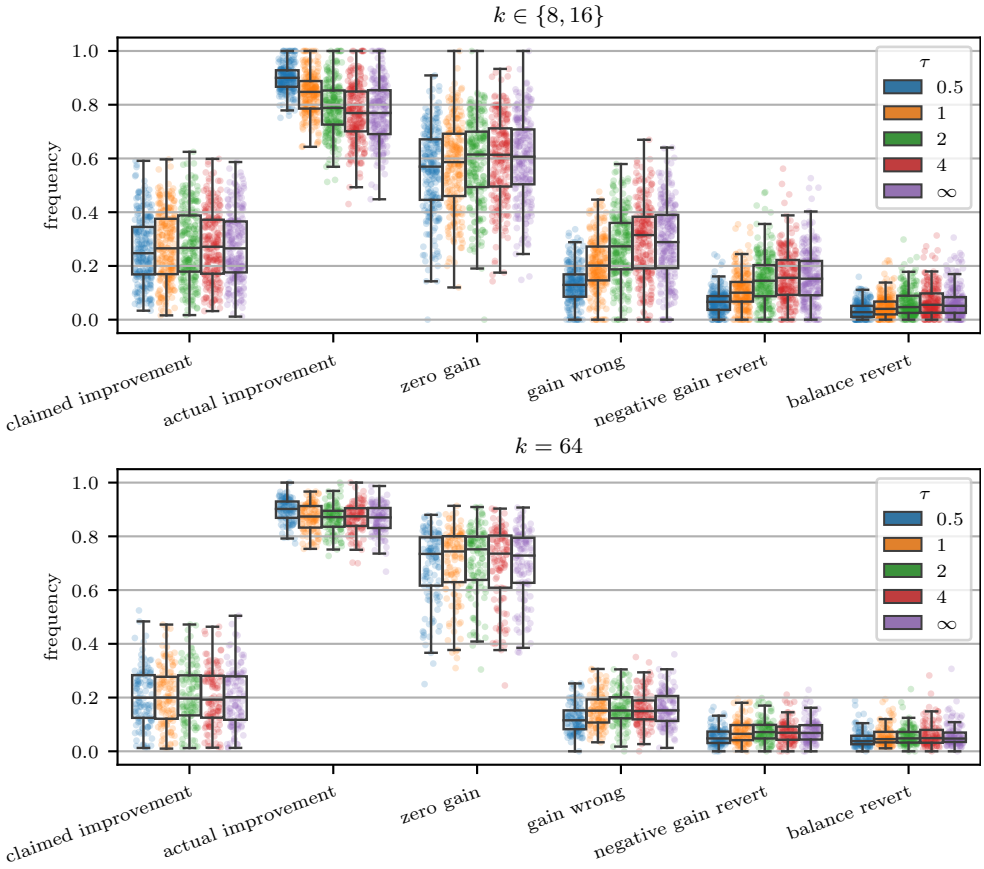

Figure 7.22: Flow-based refinement statistics per instance and different values of τ.

7.13.4 Configuring τ

In this section, we justify our choice of $\tau = 1$ for the factor to restrict concurrent searches. To save running time, we use only a subset of set B consisting of 19 instances: 5 VLSI, 5 SPM and 9 SAT instances. We perform three repetitions with different random seeds and use $k \in \{2, 8, 16, 64\}$ on machine C with 64 cores.

Figure 7.22 shows the per-instance event statistics we used before, but this time for different values of $\tau \in \{0.5, 1, 2, 4, \infty\}$. With $\tau = 0.5$, we have the same level of parallelism as the non-overlapping version in the beginning of a round, and with $\tau = \infty$ there is no restriction. The plot is split into two parts with $k \in \{8, 16\}$ and $k = 64$, as these should exhibit different levels of interference. We can see that the level of interference increases significantly with

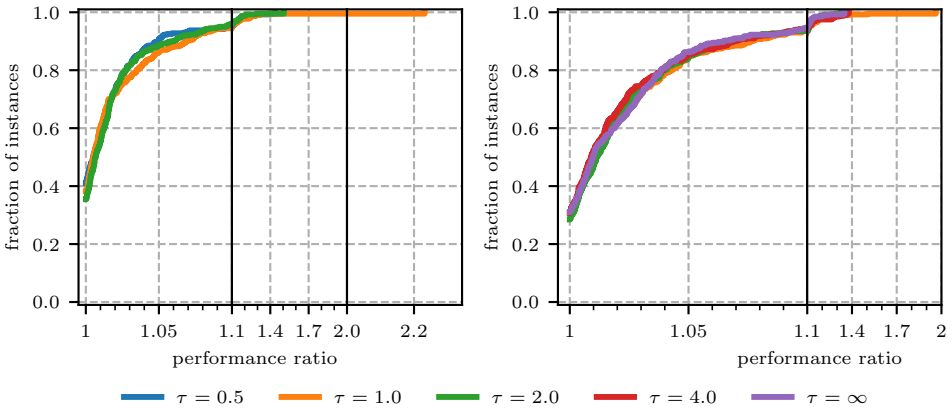

Figure 7.23: Solution quality of Mt-KaHyPar-D-F with different values of τ.

τ for $k \in \{8, 16\}$, and slower for $k = 64$. This is also reflected in the running times, where we observe an increase at $\tau > 1$ due to increased conflicts. Applying the move sequences fails more often for larger values of τ, which slows down the convergence of the scheduling algorithm. The geometric mean running times are 14.63s for $\tau = 0.5$, 12.35s for $\tau = 1$, 12.87s for $\tau = 2$, 13.34s for $\tau = 4$, and 13.01s for $\tau = \infty$.

Fortunately, Figure 7.23 shows that the increased interference affects partition quality only mildly. As $\tau = 1$ is the fastest, we choose this value.

7.13.5 Comparison with Sequential Algorithms

After analyzing the refinement algorithm, we now turn to evaluating it against state-of-the-art sequential partitioning algorithms on set A. As before, Mt-KaHyPar is run on 10 cores. Figure 7.24 summarizes the quality and running time results. We see that Mt-KaHyPar-Q-F and KaHyPar-HFC have the same quality, as they now have the same relevant feature set, followed closely by Mt-KaHyPar-D-F which catches up to their curves at factor 1.08 with 90% of the instances. These three algorithms stand out as the highest quality partitioners by a significant margin, due to refinement with FlowCutter. In Figure 7.25 we present the performance profiles of just the variants with flow-based refinement, and Mt-KaHyPar-D for a less cluttered plot. In a direct comparison Mt-KaHyPar-Q-F computes better partitions than PaToH-D on 94.7%, PaToH-Q on 87.7%, Mt-KaHyPar-D on 97.3%, Mt-KaHyPar-D-F on 73.5%, Mt-KaHyPar-Q on 95.7%, and KaHyPar-HFC on 51.3% of the instances.

Meanwhile, Mt-KaHyPar-Q-F has a similar running time as PaToH-Q as can be seen in the right plot. The geometric mean running times are 2.79s for Mt-KaHyPar-D-F, 5.11s for Mt-KaHyPar-Q-F, 48.95s for KaHyPar-HFC, and 5.86s for PaToH-Q. Further geometric mean times are, 1.17s for PaToH-D, 0.96s for Mt-KaHyPar-D, 3.13s for Mt-KaHyPar-Q and 93.2s

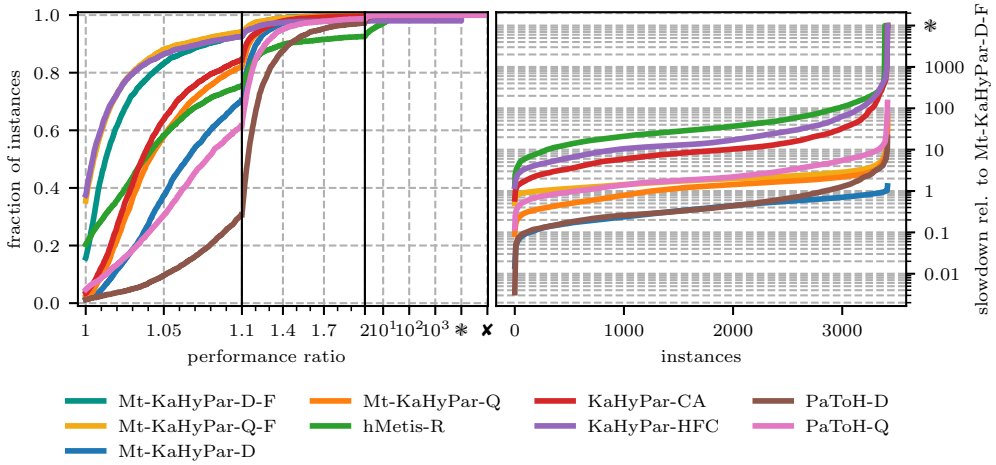

Figure 7.24: Partition quality and running time on benchmark set A, using 10 cores for Mt-KaHyPar.

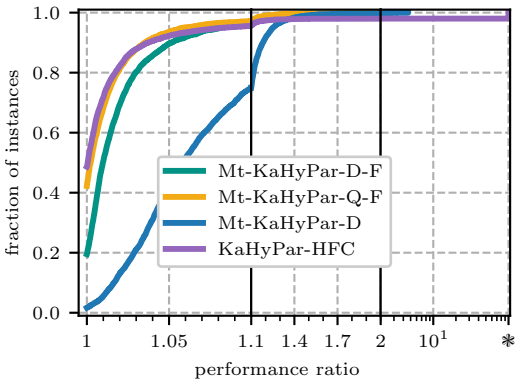

Figure 7.25: Partition quality on benchmark set A of just the variants with flow-based refinement and Mt-KaHyPar-D.

for hMetis-R. Using 10 cores, we achieve a speedup of nearly 10 over KaHyPar-HFC in the geometric mean. This is not just due to parallelism, but also due to further engineering.

The median improvement of Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F compared to the configurations that use no flow-based refinement is 4.2% and 2.7%, while only incurring a slowdown by a factor of 2.9 and 1.6. To put this into perspective, the quality preset of PaToH (PaToH-Q) improves the default preset (PaToH-D) by 5.3% in the median and is a factor of 5 slower. The median improvement of hMetis-R compared to PaToH-Q is 2.6% while it is a

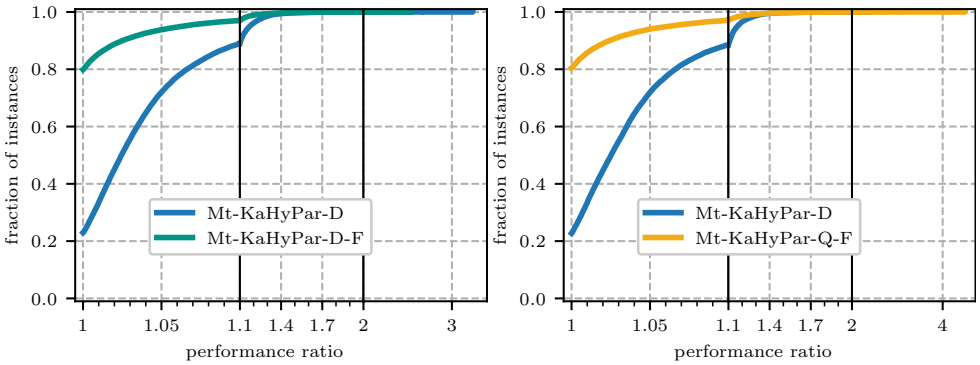

Figure 7.26: Effectiveness tests with virtual instances on benchmark set A, comparing Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F with Mt-KaHyPar-D (on 10 cores). As opposed to n-level, flow-based refinement cannot be beaten by repetitions.

factor of 15.9 slower. The solutions produced by Mt-KaHyPar-Q-F are 3% better than those of hMetis-R in the median and it has a similar running time as PaToH-Q.

Comparing our two parallel partitioners with flow-based refinement, we see that Mt-KaHyPar-Q-F gives only minor quality improvements over Mt-KaHyPar-D-F (median improvement is 0.6% whereas without flow-based refinement it is 1.9%). This demonstrates the effectiveness of flow-based refinement.

Furthermore, we present effectiveness tests with virtual instances in Figure 7.26. Whereas Mt-KaHyPar-D was competitive with Mt-KaHyPar-Q and even KaHyPar-HFC in this setup (this time parallel versus non-parallel mattered), we see that Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F are still more effective.

In conclusion, we achieved the solution quality of the currently hiqhest quality sequential partitioner in a fast parallel code.

7.13.6 Comparison with Parallel Algorithms

In Figure 7.27 we present solution quality and running time results on set B, where each algorithm was run on 64 cores. Again, the quality results of Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F are comparable, with a small advantage for the n-level variant. Yet, Mt-KaHyPar-D-F is a factor of 1.73 faster at 28.93s versus 50.11s geometric mean running time. The other geometric mean running times are 3.89s for Mt-KaHyPar-D, 23.96s for Mt-KaHyPar-Q, 29.15s for BiPart, 10.64s for Zoltan and 47.63s for PaToH-D. In terms of partition quality, the flow-based variants clearly outperform the competition.

The median improvement of Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F compared to the configurations that use no flow-based refinement is 5.2% and 3.4%, while they are slower by a factor of 7.4 and 2.1. The improvements and slowdowns are more pronounced than on set A.

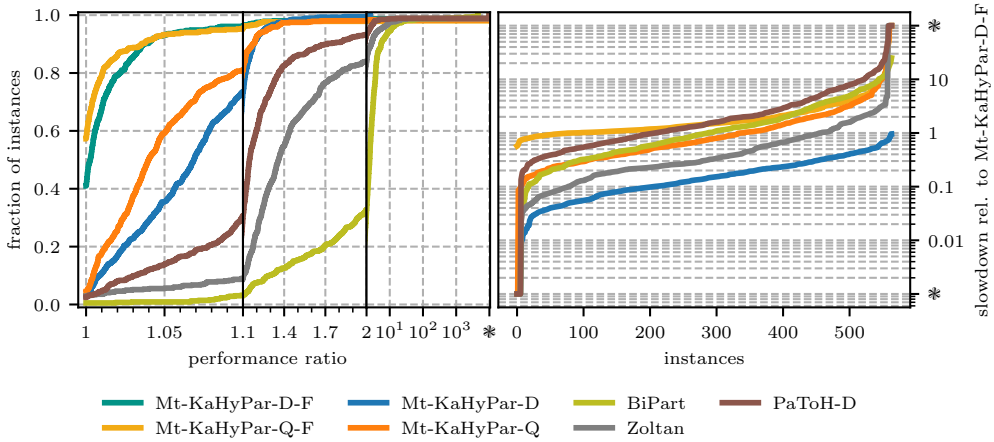

Figure 7.27: Partition quality and running time on benchmark set B. All algorithms run on 64 cores, except PaToH.

The slowdowns are expected since the size of the flow problems scales linearly with instance sizes, while the complexity of the flow-based refinement is super-linear. Mt-KaHyPar-D-F is slower than Zoltan and BiPart, but computes partitions that are 33% better than those of Zoltan and twice as good as those of BiPart in the median.

7.14 Conclusion

In this chapter, we presented an extension and thorough engineering of the powerful flow-based refinement technique. The main ingredient for speed is an on-hypergraph implementation of the flow algorithm. In extensive experiments we demonstrated the quality merits of our approach. In a second part, we parallelize flow-based refinement with two parallelism avenues: scheduling and the flow algorithm. While the speedups are not as impressive as our prior works, they are consistent with speedup results of flow algorithms in the literature. Overall, our system Mt-KaHyPar now possesses the full feature set of the highest quality sequential partitioner KaHyPar but in a parallel form, and is therefore the new state of the art. The Mt-KaHyPar-D configuration is the fastest multilevel partitioner and achieves very competitive quality, whereas Mt-KaHyPar-D-F/Mt-KaHyPar-Q-F achieve the highest quality (previously held solely by sequential KaHyPar-HFC) and are significantly faster than KaHyPar thanks to parallelism. Hence, Mt-KaHyPar-D is the method of choice for extremely large instances, whereas Mt-KaHyPar with flow-based refinement is the method of choice for medium-size instances where partition quality really matters.

8 Conclusion

Summary. In this dissertation we presented efficient parallelizations of all relevant techniques employed in the highest quality sequential partitioning framework. We achieve the same partition quality but in a fast parallel code, thus making our algorithms the state of the art both in terms of speed and partition quality.

More specifically, our default configuration is the fastest multilevel partitioner and achieves very competitive solution quality. Compared only with parallel algorithms beyond our own follow-up works, it offers both the highest quality and fastest speed.

The two versions with flow-based refinement achieve the highest partition quality at the cost of increased running times, which are however still competitive with previous algorithms other than our default configuration. The parallel n-level variant without flow-based refinement has similar running time as the $\log(n)$-level version with flow-based refinement, but the latter achieves better solution quality. Yet, n-level is still relevant in combination with flow-based refinement, and is interesting from an academic standpoint.

Contributions Summarized. The ingredients to these achievements are community-aware coarsening using the parallel Louvain algorithm, parallel agglomerative hypergraph clustering for heavy-edge rating, parallel n-level coarsening, parallel initial partitioning via multilevel recursive bipartitioning with work-stealing and parallel runs of a bipartitioning portfolio, parallel label propagation refinement, parallel localized FM refinement, parallel n-level uncoarsening, and parallel flow-based refinement with FlowCutter.

We contributed substantial engineering efforts to speed up localized FM and enhance its optimization capabilities. In our experiments, we thoroughly analyzed the impact of each component and design choice. Among these components are gain accuracy techniques such as attributed gains, parallel gain tables, and parallel gain recalculation. The latter two

improve partition quality and speed in both n-level and $\log(n)$-level refinement, whereas attributed gains only contribute to partition quality in n-level refinement.

Perhaps the most surprising result is the efficient parallelization of n-level (un)coarsening, particularly the fully asynchronous version. The main optimizations are projecting gain table values via uncontractions, and restoring ranges of newly activated pins in the semi-dynamic hypergraph data structure.

For flow-based refinement, we engineered a hypergraph version of FlowCutter by implementing flow algorithms directly on the hypergraph without constructing the Lawler network. On-hypergraph implementations have substantial running time benefits over the Lawler network. We fixed a previously undocumented bug in a synchronous parallel push-relabel algorithm, and developed optimizations such as imposing additional capacities, and a mechanism to detect early termination. Furthermore, we developed techniques for running FlowCutter on disconnected hypergraphs, show how to avoid redundant computations, and efficiently integrate Dinitz algorithm in the incremental maximum flow problems of FlowCutter.

Finally, we presented deterministic versions of the local moving algorithms for community preprocessing, coarsening and label propagation refinement, culminating in a deterministic version of our framework. In our experiments, we showed that there is a small price to pay in terms of partition quality and running time, whereas the speedups were better than for the non-deterministic version. We outlined challenges and ideas for deterministic versions of localized FM and flow-based refinement, which serve as a starting point for future work.

Future Work. Beyond this, the research on shared-memory parallelization of existing techniques appears saturated. Future work should focus on transferring the high quality techniques to distributed memory and other computational architectures (e.g., GPUs). Efficient distributed implementations of FM face enormous challenges, as our localized version is inherently very fine-grained. Flow-based refinement on the other hand is fairly amenable to transfer since scheduling block pairs is coarse-grained. Refinement on a block pair should be run on one host, but can still be run in parallel using the existing shared-memory implementation. Label propagation refinement and coarsening are well understood in distributed memory. So far a highly scalable implementation is missing, but work on a plain graph version is currently in progress.

Another missing gap is experimental studies of algorithms with theoretical approximation guarantees, albeit non-constant factor approximation. We cannot expect remotely competitive quality or running time to the established well-understood heuristics. This is therefore purely for academic purposes. At the moment we lack an understanding of how bad the gap actually is. The structure of these algorithms is usually recursive bipartitioning into more than k blocks, and subsequent packing along the recursion tree to reach the desired k. It might be worthwhile to combine the theoretically sound components in such algorithms with heuristic components in practical works to get the best of both worlds.

Most of the research in recent years has focused on the refinement phase, as it is easy to

understand where improvements come from. The coarsening phase on the other hand has seen very little progress. While clustering intuitively seems the best approach, the success of community-aware coarsening showed that simple local greedy optimization is inadequate. A recent work [SSS22] incorporates spatial proximity in embeddings into the rating and restrictions. It showed promising results in terms of quality, but is completely infeasible in terms of running time. Therefore, future research should work towards better understanding of the coarsening methods.

Finally, it would be interesting to study variations of the partitioning problem. A notable example is judicious partitioning, where we are interested in minimizing the maximum load. The *load* of a block is the weight sum of nets incident to vertices in the block. There is no need for a balance constraint, as a locally optimal solution has balanced loads. This problem is a data distribution tool in a phylogenetic interference bioinformatics application [Baa+19].

As only the maximum load contributes to the objective function, *all vertex moves* have zero gain if there is a sufficient gap to the second highest load. This makes the existing greedy approaches essentially unusable. So far, this NP-hard problem has received little algorithmic attention. Most works focus on proving bounds for specific hypergraph classes. The only algorithm known so far [Tan+17, Baa+19] iterates through possible objective values and checks whether this value is attainable by greedily solving a minimum set cover problem.

Still, it is possible to perform refinement to some extent by moving out of the block with the highest load. We are currently working on a paper, where we apply greedy initial partitioning and refinement heuristics to the judicious problem formulation with promising results. In spite of the mentioned challenges, our methods outperform the mentioned set-cover approach, both in terms of load (moderately) and running time (enormously).

Bibliography

[ABR20] Seher Acer, Erik G. Boman, and Sivasankaran Rajamanickam. **SPHYNX: Spectral Partitioning for HYbrid aNd aXelerator-enabled systems**. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW*, pages 440–449, 2020. DOI: 10.1109/IPDPSW50202.2020.00082.
Cited on page 49.

[ACU08] Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. **Multi-level Direct k-Way Hypergraph Partitioning With Multiple Constraints and Fixed Vertices**. In *Journal of Parallel Distributed Computing* volume 68:5, pages 609–625, 2008. DOI: 10.1016/j.jpdc.2007.09.006.
Cited on page 58.

[AFHM08] Michael Armbruster, Marzena Fügenschuh, Christoph Helmberg, and Alexander Martin. **A Comparative Study of Linear and Semidefinite Branch-and-Cut Methods for Solving the Minimum Graph Bisection Problem**. In *Integer Programming and Combinatorial Optimization, 13th International Conference, IPCO 2008, Bertinoro, Italy, May 26-28, 2008, Proceedings*. Ed. by Andrea Lodi, Alessandro Panconesi, and Giovanni Rinaldi. Volume 5035 of Lecture Notes in Computer Science, pages 112–124. Springer, 2008. DOI: 10.1007/978-3-540-68891-4_8.
Cited on page 49.

[AH19] Pablo Andrés-Martínez and Chris Heunen. **Automated Distribution of Quantum Circuits via Hypergraph Partitioning**. In *Physical Review A* volume 100:3, American Physical Society (APS), Sept. 2019. DOI: 10.1103/physreva.100.032308.
Cited on page 2.

[AHSS17] 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, Jan. 2017. DOI: 10.1137/1.9781611974768.3.
Cited on pages 4, 7, 35, 47, 60, 68.

[AK95] Charles J. Alpert and Andrew B. Kahng. **Recent Directions in Netlist Partitioning: A Survey**. In *Integration* volume 19:1-2, pages 1–81, Elsevier, 1995. DOI: 10.1016/0167-9260(95)00008-4.
Cited on page 1.

[Akh19] Yaroslav Akhremtsev. **Parallel and External High Quality Graph Partitioning**. PhD thesis. Karlsruhe Institute of Technology, Germany, 2019. DOI: 10.5445/IR/1000098895.
Cited on page 21.

[AL08] Reid Andersen. and Kevin J. Lang. **An Algorithm for Improving Graph Partitions**. In *Proc. of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 651–660, 2008. DOI: 10.5555/1347082.1347154.
Cited on page 46.

[Alp98] Charles J. Alpert. **The ISPD98 Circuit Benchmark Suite**. In *International Symposium on Physical Design (ISPD)*, pages 80–85, Apr. 1998. DOI: 10.1145/274535.274546.
Cited on pages 1, 19.

[AR06] Konstantin Andreev and Harald Räcke. **Balanced Graph Partitioning**. In *Theory of Computing Systems* volume 39:6, pages 929–939, 2006. DOI: 10.1007/s00224-006-1350-7.
Cited on page 48.

[AS95] Richard J. Anderson and João C. Setubal. **A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem**. In *J. Parallel Distributed Comput.* volume 29:1, pages 17–26, 1995. DOI: 10.1006/jpdc.1995.1103.
Cited on page 178.

[ASS18a] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. **High-Quality Shared-Memory Graph Partitioning**. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. Ed. by Marco Aldinucci, Luca Padovani, and Massimo Torquati. Volume 11014 of Lecture Notes in Computer Science, pages 659–671. Springer, 2018. DOI: 10.1007/978-3-319-96983-1_47.
Cited on pages 4, 5, 6, 22, 37, 41, 42, 44, 48, 51, 64, 68, 73, 76, 106.

[ASS18b] Robin Andre, Sebastian Schlag, and Christian Schulz. **Memetic Multilevel Hypergraph Partitioning**. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*, pages 347–354. ACM, 2018. DOI: 10.1145/3205455.3205475.
Cited on pages 20, 39, 51.

[ASS18c] Robin Andre, Sebastian Schlag, and Christian Schulz. **Memetic multilevel hypergraph partitioning**. In *Genetic and Evolutionary Computation Conference, (GECCO)*, pages 347–354, 2018. DOI: 10.1145/3205455.3205475.
Cited on page 171.

[ASS20] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. **High-Quality Shared-Memory Graph Partitioning**. In *IEEE Transactions on Parallel Distributed Systems* volume 31:11, pages 2710–2722, 2020. DOI: 10.1109/TPDS.2020.3001645.
Cited on pages 24, 81, 83, 86, 101.

[AWFS17] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. **In-Place Parallel Super Scalar Samplesort (IPSSSSo)**. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*. Ed. by Kirk Pruhs and Christian Sohler. Volume 87 of LIPIcs, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/LIPIcs.ESA.2017.9.
Cited on page 109.

[Baa+19] Ivo Baar, Lukas Hübner, Peter Oettig, Adrian Zapletal, Sebastian Schlag, Alexandros Stamatakis, and Benoit Morel. **Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning**. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 175–184. IEEE, 2019. DOI: 10.1109/IPDPSW.2019.00038.
Cited on page 205.

[BBR97] Roberto Battiti, Alan A. Bertossi, and Romeo Rizzi. **Randomized Greedy Algorithms for the Hypergraph Partitioning Problem**. In *Randomization Methods in Algorithm Design, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, December 12-14, 1997*. Ed. by Panos M. Pardalos, Sanguthevar Rajasekaran, and José Rolim. Volume 43 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 21–35. DIMACS/AMS, 1997. DOI: 10.1090/dimacs/043/02.
Cited on page 40.

[BBS15] Niklas Baumstark, Guy E. Blelloch, and Julian Shun. **Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm**. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*. Ed. by Nikhil Bansal and Irene Finocchi. Volume 9294 of Lecture Notes in Computer Science, pages 106–117. Springer, 2015. DOI: 10.1007/978-3-662-48350-3_10.
Cited on pages 8, 178, 182.

[BC06] David A. Bader and Guojing Cong. **Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs**. In *Journal of Parallel Distributed Computing* volume 66:11, pages 1366–1378, 2006. DOI: 10.1016/j.jpdc.2006.06.001.
Cited on page 128.

[BCR97] Lorenzo Brunetta, Michele Conforti, and Giovanni Rinaldi. **A Branch-and-Cut Algorithm for the Equicut Problem**. In *Mathematical Programming* volume 77, pages 243–263, 1997. DOI: 10.1007/BF02614373.
Cited on page 49.

[BDHJ14] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. **The SAT Competition 2014**. 2014.
Cited on page 19.

[BFGS] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. **Internally Deterministic Parallel Algorithms Can Be Fast**. In *PPoPP 2012*. DOI: 10.1145/2145816.2145840.
Cited on pages 6, 107.

[BGLL08] Vincent D. Blondel, Jean Guillaume, Renaud Lambiotte, and Etienne Lefebvre. **Fast Unfolding of Communities in Large Networks**. In *Journal of Statistical Mechanics: Theory and Experiment*: 10, 2008.
Cited on pages 35, 38, 39, 108.

[BH10] Una Benlic and Jin-Kao Hao. **An Effective Multilevel Memetic Algorithm for Balanced Graph Partitioning**. In *IEEE International Conference on Tools with Artificial Intelligence*, pages 121–128. IEEE Computer Society, 2010. DOI: 10.1109/ICTAI.2010.25.
Cited on page 171.

[BH11a] Una Benlic and Jin-Kao Hao. **A Multilevel Memetic Approach for Improving Graph k-Partitions**. In *IEEE Transactions on Evolutionary Computation* volume 15:5, pages 624–642, Oct. 2011. DOI: 10.1109/TEVC.2011.2136346.
Cited on page 171.

[BH11b] Una Benlic and Jin-Kao Hao. **An Effective Multilevel Tabu Search Approach for Balanced Graph Partitioning**. In *Computers & Operations Research* volume 38:7, pages 1066–1075, July 2011. DOI: 10.1016/j.cor.2010.10.007.
Cited on page 171.

[BJ92] Thang Nguyen Bui and Curt Jones. **Finding Good Approximate Vertex and Edge Partitions is NP-Hard**. In *Information Processing Letters* volume 42:3, pages 153–159, 1992. DOI: 10.1016/0020-0190(92)90140-Q.
Cited on page 3.

[BMSW13] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. **Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge**. Volume 588 of. American Mathematical Society, 2013.
Cited on page 27.

[Bra+08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. **On Modularity Clustering**. In *IEEE Transactions on Knowledge and Data Engineering* volume 20:2, pages 172–188, IEEE, 2008.
Cited on page 38.

[Bul+16] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. **Recent Advances in Graph Partitioning**. In *Algorithm Engineering - Selected Results and Surveys*. Volume 9220. Lecture Notes in Computer Science. Springer, 2016, pages 117–158. DOI: 10.1007/978-3-319-49487-6_4.
Cited on page 27.

[CA99] Ümit V. Catalyurek and Cevdet Aykanat. **Hypergraph-Partitioning-based Decomposition for Parallel Sparse-Matrix Vector Multiplication**. In *IEEE Transactions on Parallel and Distributed Systems* volume 10:7, pages 673–693, IEEE, 1999. DOI: 10.1109/71.780863.
Cited on pages 1, 2, 5, 8, 19, 22, 33, 35, 39, 48, 53, 56, 62, 63, 116, 156.

[CB05] Guojing Cong and David A. Bader. **An Empirical Analysis of Parallel Random Permutation Algorithms on SMPs**. In *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems, September 12-14, 2005 Imperial Palace Hotel, Las Vegas, Nevada, USA*. Ed. by Michael J. Oudshoorn and Sanguthevar Rajasekaran, pages 27–34. ISCA, 2005.
Cited on page 115.

[CBM07] Pierre Chardaire, Musbah Barake, and Geoff P. McKeown. **A PROBE-Based Heuristic for Graph Partitioning**. In *IEEE Transactions on Computers* volume 56:12, pages 1707–1720, 2007. DOI: 10.1109/TC.2007.70760.
Cited on page 171.

[ÇDKU12] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Ucar. **Multithreaded Clustering for Multi-Level Hypergraph Partitioning**. In *IEEE Transactions on Parallel and Distributed Systems*, pages 848–859, 2012.
Cited on pages 36, 56, 81.

[CG97] Boris V. Cherkassky and Andrew V. Goldberg. **On Implementing the Push-Relabel Method for the Maximum Flow Problem**. In *Algorithmica* volume 19:4, pages 390–410, 1997. DOI: 10.1007/PL00009180.
Cited on pages 17, 180, 182.

[CKM00a] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. **Improved Algorithms for Hypergraph Bipartitioning**. In *Proceedings of ASP-DAC 2000, Asia and South Pacific Design Automation Conference 2000, Yokohama, Japan*, pages 661–666. ACM, 2000. DOI: 10.1145/368434.368864.
Cited on page 6.

[CKM00b] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. **Iterative Partitioning with Varying Node Weights**. In *VLSI Design* volume 2000:3, pages 249–258, 2000. DOI: 10.1155/2000/15862.
Cited on page 11.

[CKM99] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. **Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning**. In *Algorithm Engineering and Experimentation, International Workshop ALENEX '99, Baltimore, MD, USA, January 15-16, 1999, Selected Papers*. Ed. by Michael T. Goodrich and Catherine C. McGeoch. Volume 1619 of Lecture Notes in Computer Science, pages 177–193. Springer, 1999. DOI: 10.1007/3-540-48518-X_11.
Cited on page 31.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. **Introduction to Algorithms**. second. MIT Press, 2001.
Cited on pages 12, 165.

[CS93] Jason Cong and M'Lissa Smith. **A Parallel Bottom-Up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design**. In *30th Conference on Design Automation (DAC)*, pages 755–760, June 1993.
Cited on page 1.

[CVB21] Philip S Chodrow, Nate Veldt, and Austin R Benson. **Generative Hypergraph Clustering: From Blockmodels to Modularity**. In *Science Advances* volume 7:28, eabh1303, American Association for the Advancement of Science, 2021. DOI: 10.1126/sciadv.abh1303.
Cited on page 61.

[CZJM10] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. **Schism: a Workload-Driven Approach to Database Replication and Partitioning**. In *Proceedings of the VLDB Endowment* volume 3:1, pages 48–57, 2010. DOI: 10.14778/1920841.1920853.
Cited on page 2.

[DD96] Shantanu Dutt and Wenyong Deng. **VLSI Circuit Partitioning by Cluster-Removal using Iterative Improvement Techniques**. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*. Ed. by Rob A. Rutenbar and Ralph H. J. M. Otten, pages 194–200. IEEE Computer Society / ACM, 1996. DOI: 10.1109/ICCAD.1996.569591.
Cited on page 32.

[Dev+06] 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*, 10–pp, 2006. DOI: 10.1109/IPDPS.2006.1639359.
Cited on pages 21, 33, 37, 41, 43, 48, 121.

[DGRW12] Daniel Delling, Andrew V. Goldberg, Ilya P. Razenshteyn, and Renato Fonseca F. Werneck. **Exact Combinatorial Branch-and-Bound for Graph Bisection**. In *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*. Ed. by David A. Bader and Petra Mutzel, pages 30–44. SIAM / Omnipress, 2012. DOI: 10.1137/1.9781611972924.3.
Cited on page 49.

[DH11] Timothy A. Davis and Yifan Hu. **The University of Florida Sparse Matrix Collection**. In *ACM Transactions on Mathematical Software* volume 38:1, pages 1:1–1:25, ACM, Nov. 2011. DOI: 10.1145/2049662.2049663.
Cited on page 19.

[DHKY20] Timothy A Davis, William W Hager, Scott P Kolodziej, and S Nuri Yeralan. **Algorithm 1003: Mongoose, a Graph Coarsening and Partitioning Library**. In *ACM Transactions on Mathematical Software (TOMS)* volume 46:1, pages 1–18, ACM New York, NY, USA, 2020. DOI: 10.1145/3337792.
Cited on page 36.

[Din70] Yefim Dinitz. **Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation**. In *Soviet Mathematics-Doklady* volume 11:5, pages 1277–1280, 1970.
Cited on pages 16, 154, 158, 159, 161.

[DKÇ13] Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. **Hypergraph Sparsification and Its Application to Partitioning**. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 200–209, 2013. DOI: 10.1109/ICPP.2013.29.
Cited on pages 15, 58, 60.

[DM02] Elizabeth D. Dolan and Jorge J. Moré. **Benchmarking Optimization Software with Performance Profiles**. In *Mathematical Programming* volume 91:2, pages 201–213, Springer, 2002. DOI: 10.1007/s101070100263.
Cited on page 23.

[Dut93] Shantanu Dutt. **New Faster Kernighan-Lin-type Graph-Partitioning Algorithms**. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*. Ed. by Michael R. Lightner and Jochen A. G. Jess, pages 370–377. IEEE Computer Society / ACM, 1993. DOI: 10.1109/ICCAD.1993.580083.
Cited on page 29.

[DW12] Daniel Delling and Renato F. Werneck. **Better Bounds for Graph Bisection**. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*. Ed. by Leah Epstein and Paolo Ferragina. Volume 7501 of Lecture Notes in Computer Science, pages 407–418. Springer, 2012. DOI: 10.1007/978-3-642-33090-2_36.
Cited on pages 49, 171.

[EK72] Jack Edmonds and Richard M. Karp. **Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems**. In *Journal of the ACM* volume 19:2, pages 248–264, 1972. DOI: 10.1145/321694.321699.
Cited on pages 16, 159.

[Fer+98] Carlos Eduardo Ferreira, Alexander Martin, C. Carvalho de Souza, Robert Weismantel, and Laurence A. Wolsey. **The Node Capacitated Graph Partitioning Problem: A Computational Study**. In *Math. Program.* volume 81, pages 229–256, 1998. DOI: 10.1007/BF01581107.
Cited on page 49.

[FF15] Andreas Emil Feldmann and Luca Foschini. **Balanced Partitions of Trees and Applications**. In *Algorithmica* volume 71:2, pages 354–376, 2015. DOI: 10.1007/s00453-013-9802-3.
Cited on page 48.

[FF56] Lester Randolph Ford and Delbert R Fulkerson. **Maximal Flow through a Network**. In *Canadian Journal of Mathematics* volume 8, pages 399–404, Cambridge University Press, 1956. DOI: 10.4153/CJM-1956-045-5.
Cited on page 16.

[Fie75] Miroslav Fiedler. **A Property of Eigenvectors of Nonnegative Symmetric Matrices and its Application to Graph Theory**. In *Czechoslovak mathematical journal* volume 25:4, pages 619–633, Institute of Mathematics, Academy of Sciences of the Czech Republic, 1975.
Cited on page 49.

[FM82] 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.
Cited on pages 1, 27, 30, 71, 173.

[FS21] Marcelo Fonseca Faraj and Christian Schulz. **Buffered Streaming Graph Partitioning**. In *CoRR* volume abs/2102.09384, 2021. arXiv: 2102.09384. URL: https://arxiv.org/abs/2102.09384.
Cited on page 50.

[GH21] Lars Gottesbüren and Michael Hamann. **Deterministic Parallel Hypergraph Partitioning**. In *CoRR* volume abs/2112.12704, 2021. arXiv: 2112.12704. URL: https://arxiv.org/abs/2112.12704.
Cited on pages 15, 52.

[GHS22] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. **Parallel Flow-Based Hypergraph Partitioning**. In tech. rep., 2022.
Cited on pages 8, 153, 157.

[GHSS21] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. **Scalable Shared-Memory Hypergraph Partitioning**. In *23rd Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2021. DOI: 10.1137/1.9781611976472.2.
Cited on pages 35, 51.

[GHSS22] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. **Shared-Memory *n*-level Hypergraph Partitioning**. In *24th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Jan. 2022. DOI: 10.1137/1.9781611977042.11.
Cited on pages 52, 64, 125.

[GHSW20] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. **Advanced Flow-Based Multilevel Hypergraph Partitioning**. In *18th International Symposium on Experimental Algorithms (SEA)*, 2020. DOI: 10.4230/LIPIcs.SEA.2020.11.
Cited on pages 4, 8, 22, 51, 95, 153, 154, 157, 158, 159, 167, 174.

[GHW19a] Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. **Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm**. In *27th European Symposium on Algorithms (ESA)*, pages 52:1–52:17, 2019. DOI: 10.4230/LIPIcs.ESA.2019.52.
Cited on pages 4, 8, 153, 154, 158, 166, 168, 186.

[GHW19b] Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. **Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm**. In 2019.
Cited on pages 169, 171.

[GJ79] M. R. Garey and D. S. Johnson. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. Volume 174 of. W.H. Freeman, San Francisco, 1979.
Cited on page 11.

[GK21] Johnnie Gray and Stefanos Kourtis. **Hyper-optimized tensor network contraction**. In *Quantum* volume 5, page 410, Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, 2021. DOI: 10.22331/q-2021-03-15-410.
Cited on pages 1, 2.

[GL16] Aditya Grover and Jure Leskovec. **node2vec: Scalable Feature Learning for Networks**. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, pages 855–864. ACM, 2016. DOI: 10.1145/2939672.2939754.
Cited on page 37.

[Got+20] Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. **Engineering Exact Quasi-Threshold Editing**. In *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*. Ed. by Simone Faro and Domenico Cantone. Volume 160 of LIPIcs, pages 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/LIPIcs.SEA.2020.10.
Cited on page 49.

[Got+21] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. **Deep Multilevel Graph Partitioning**. In *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*. Ed. by Petra Mutzel, Rasmus Pagh, and Grzegorz Herman. Volume 204 of LIPIcs, pages 48:1–48:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/LIPIcs.ESA.2021.48.
Cited on pages 6, 21, 22, 36, 37, 42, 43, 46, 48, 101.

[Gra69] Ronald L. Graham. **Bounds on Multiprocessing Timing Anomalies**. In *SIAM Journal of Applied Mathematics* volume 17:2, pages 416–429, 1969.
Cited on page 11.

[GSSB15] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. **A Top-Down Parallel Semisort**. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*. Ed. by Guy E. Blelloch and Kunal Agrawal, pages 24–34. ACM, 2015. DOI: 10.1145/2755573.2755597.
Cited on page 15.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. **Exact Routing in Large Road Networks Using Contraction Hierarchies**. In *Transportation Science* volume 46:3, pages 388–404, 2012. DOI: 10.1287/trsc.1110.0401.
Cited on page 47.

[GT88] Andrew V. Goldberg and Robert Endre Tarjan. **A New Approach to the Maximum-Flow Problem**. In *Journal of the ACM* volume 35:4, pages 921–940, 1988. DOI: 10.1145/48014.61051.
Cited on page 17.

[Heu15] Tobias Heuer. **Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning**. Bachelor Thesis. Karlsruhe Institute of Technology, Aug. 2015.
Cited on pages 40, 51, 62, 63.

[HHK97] Lars W. Hagen, Dennis J.-H. Huang, and Andrew B. Kahng. **On Implementation Choices for Iterative Improvement Partitioning Algorithms**. In *IEEE Transactions on Computer Aided Design of Integrated Circuits Systems* volume 16:10, pages 1199–1205, 1997. DOI: 10.1109/43.662682.
Cited on page 31.

[HL93] Bruce Hendrickson and Robert Leland. **A Multilevel Algorithm for Partitioning Graphs**. In tech. rep., 1993.
Cited on pages 3, 34.

[HL95] Bruce Hendrickson and Robert W. Leland. **An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations**. In *SIAM J. Sci. Comput.* volume 16:2, pages 452–469, 1995. DOI: 10.1137/0916028.
Cited on page 49.

[HMS21] Tobias Heuer, Nikolai Maas, and Sebastian Schlag. **Multilevel Hypergraph Partitioning with Vertex Weights Revisited**. In *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*. Ed. by David Coudert and Emanuele Natale. Volume 190 of LIPIcs, pages 8:1–8:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/LIPIcs.SEA.2021.8.
Cited on page 11.

[HS17] 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, June 2017. DOI: 10.4230/LIPIcs.SEA.2017.21.
Cited on pages 4, 5, 23, 37, 38, 53, 60, 95.

[HS18a] Michael Hamann and Ben Strasser. **Graph Bisection with Pareto Optimization**. In *ACM Journal of Experimental Algorithmics* volume 23, 2018. DOI: 10.1145/3173045.
Cited on pages 7, 153, 155, 156, 161, 167, 168, 171, 173.

[HS18b] Lorenz Hübschle-Schneider and Peter Sanders. **Communication Efficient Checking of Big Data Operations**. In *IEEE Transactions on Parallel and Distributed Systems*, pages 650–659, 2018. DOI: 10.1109/IPDPS.2018.00074.
Cited on page 60.

[HSS10] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. **Engineering a Scalable High Quality Graph Partitioner**. In *IEEE Transactions on Parallel and Distributed Systems*, pages 1–12, 2010.
Cited on page 36.

[HSS19] Tobias Heuer, Peter Sanders, and Sebastian Schlag. **Network Flow-Based Refinement for Multilevel Hypergraph Partitioning**. In *ACM Journal of Experimental Algorithmics (JEA)* volume 24:1, pages 2.3:1–2.3:36, ACM, Sept. 2019. DOI: 10.1145/3329872.
Cited on pages 7, 22, 153, 157, 158, 174, 176, 185.

[HSWZ18] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. **Distributed Graph Clustering Using Modularity and Map Equation**. In *European Conference on Parallel Processing (Euro-Par)*, pages 688–702, 2018. DOI: 10.1007/978-3-319-96983-1_49.
Cited on pages 6, 15, 43, 61, 108, 124.

[Hua+20] Cupjin Huang, Fang Zhang, Michael Newman, Junjie Cai, Xun Gao, Zhengxiong Tian, Junyin Wu, Haihong Xu, Huanjun Yu, Bo Yuan, et al. **Classical simulation of quantum supremacy circuits**. In *arXiv preprint arXiv:2005.06787*, 2020.
Cited on page 2.

[Jia+19] Wenkai Jiang, Jianzhong Qi, Jeffrey Xu Yu, Jin Huang, and Rui Zhang. **HyperX: A Scalable Hypergraph Framework**. In *IEEE Transactions on Knowledge and Data Engineering* volume 31:5, pages 909–922, 2019. DOI: 10.1109/TKDE.2018.2848257.
Cited on page 4.

[JSA21] Nazanin Jafari, Oguz Selvitopi, and Cevdet Aykanat. **Fast shared-memory streaming multilevel graph partitioning**. In *Journal of Parallel and Distributed Computing* volume 147, pages 140–151, Elsevier, 2021. DOI: 10.1016/j.jpdc.2020.09.004.
Cited on page 50.

[Kab+17a] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. **Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner**. In *Proceedings of the VLDB Endowment*. Volume 10 of, pages 1418–1429, 2017. DOI: 10.14778/3137628.3137650.
Cited on pages 2, 22, 33, 42, 43, 48, 98, 113, 115.

[Kab+17b] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Alessandro Presta, and Yaroslav Akhremtsev. **Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner**. In *arXiv preprint arXiv:1707.06665*, 2017.
Cited on page 98.

[KAKS99] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. **Multilevel Hypergraph Partitioning: Applications in VLSI Domain**. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* volume 7:1, pages 69–79, IEEE, 1999. DOI: 10.1109/92.748202.
Cited on pages 1, 6, 22, 33, 39, 48, 53.

[Kam+19] Bogumił Kamiński, Valérie Poulin, Paweł Prałat, Przemysław Szufel, and François Théberge. **Clustering via Hypergraph Modularity**. In *PloS one* volume 14:11, e0224307, Public Library of Science San Francisco, CA USA, 2019. DOI: 10.1371/journal.pone.0224307.
Cited on page 61.

[Kar74] Alexander V. Karzanov. **Determining the Maximal Flow in a Network by the Method of Preflows**. In *Soviet Mathematics Doklady*. Volume 15 of, pages 434–437, 1974.
Cited on page 17.

[KK00] George Karypis and Vipin Kumar. **Multilevel *k*-way Hypergraph Partitioning**. In *VLSI Design* volume 2000:3, pages 285–300, 2000. DOI: 10.1155/2000/19436.
Cited on pages 6, 22, 35, 40, 48, 62, 113, 177, 178.

[KK95] George Karypis and Vipin Kumar. **Multilevel Graph Partitioning Schemes**. In *Proceedings of the 1995 International Conference on Parallel Processing, Urbana-Champain, Illinois, USA, August 14-18, 1995. Volume III: Algorithms & Applications*. Ed. by Kyle A. Gallivan, pages 113–122. CRC Press, 1995.
Cited on page 36.

[KK98] George Karypis and Vipin Kumar. **Multilevel k-way Partitioning Scheme for Irregular Graphs**. In *Journal of Parallel and Distributed Computing* volume 48:1, pages 96–129, 1998. DOI: 10.1006/jpdc.1997.1404.
Cited on pages 32, 33, 36, 39, 40, 44, 68.

[KK99] George Karypis and Vipin Kumar. **Parallel Multilevel k-Way Partitioning Scheme for Irregular Graphs**. In *Siam Review* volume 41:2, pages 278–300, SIAM, 1999.
Cited on pages 37, 40, 41, 42.

[KL70] Brian W. Kernighan and Shen Lin. **An Efficient Heuristic Procedure for Partitioning Graphs**. In *The Bell System Technical Journal* volume 49:2, pages 291–307, Feb. 1970.
Cited on pages 27, 28, 29, 33, 173.

[KNS09] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. **Partitioning Graphs into Balanced Components**. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*. Ed. by Claire Mathieu, pages 942–949. SIAM, 2009.
Cited on page 48.

[KÖ19] Gökçehan Kara and Can C. Özturan. **Graph Coloring Based Parallel Push-relabel Algorithm for the Maximum Flow Problem**. In *ACM Transactions on Mathematical Software* volume 45:4, pages 46:1–46:28, 2019. DOI: 10.1145/3330481.
Cited on pages 178, 180.

[KQDK14] K Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. **SWORD: Workload-Aware Data Placement and Replica Selection for Cloud Data Management Systems**. In *The VLDB Journal* volume 23:6, pages 845–870, Springer, 2014. DOI: 10.1007/s00778-014-0362-1.
Cited on pages 1, 2.

[Kra21] Robert Krause. **Community Detection in Hypergraphs with Application to Partitioning**. Bachelor Thesis. Karlsruhe Institute of Technology, June 2021.
Cited on page 61.

[KRC00] Stefan E. Karisch, Franz Rendl, and Jens Clausen. **Solving Graph Bisection Problems with Semidefinite Programming**. In *INFORMS J. Comput.* volume 12:3, pages 177–191, 2000. DOI: 10.1287/ijoc.12.3.177.12637.
Cited on page 49.

[Kri84] Balakrishnan Krishnamurthy. **An Improved Min-Cut Algorithm for Partitioning VLSI Networks**. In *IEEE Transactions on Computers* volume 33:5, pages 438–446, 1984. DOI: 10.1109/TC.1984.1676460.
Cited on pages 1, 32.

[Lar06] Jesper Larsson Träff. **Direct graph k-partitioning with a Kernighan–Lin like heuristic**. In *Operations Research Letters* volume 34:6, pages 621–629, Nov. 2006. DOI: 10.1016/j.orl.2005.10.003.
Cited on pages 32, 68, 79.

[LaS+15] Dominique LaSalle, Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. **Improving Graph Partitioning for Modern Graphs and Architectures**. In *Proceedings of the 5th Workshop on Irregular Applications - Architectures and Algorithms, IA3 2015, Austin, Texas, USA, November 15, 2015*, pages 14:1–14:4. ACM, 2015. DOI: 10.1145/2833179.2833188.
Cited on page 36.

[Lau21] Moritz Laupichler. **Asynchronous n-Level Hypergraph Partitioning**. Master Thesis. Karlsruhe Institute of Technology, Nov. 2021.
Cited on pages 125, 148.

[Law73] Eugene L. Lawler. **Cutsets and Partitions of Hypergraphs**. In *Networks* volume 3:3, pages 275–285, 1973. DOI: 10.1002/net.3230030306.
Cited on pages 17, 157.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. **Dthreads: Efficient Deterministic Multithreading**. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. Ed. by Ted Wobber and Peter Druschel, pages 327–336. ACM, 2011. DOI: 10.1145/2043556.2043587.
Cited on page 107.

[Len90] Thomas Lengauer. **Combinatorial Algorithms for Integrated Circuit Layout**. John Wiley & Sons, Inc., 1990. DOI: 10.1017/S0263574700015691.
Cited on pages 1, 3.

[LK13] Dominique LaSalle and George Karypis. **Multi-Threaded Graph Partitioning**. In *IEEE Transactions on Parallel and Distributed Systems*, pages 225–236, 2013. DOI: 10.1109/IPDPS.2013.50.
Cited on pages 6, 22, 37, 41, 42, 44, 62, 90, 106.

[LK16] Dominique LaSalle and George Karypis. **Parallel Hill-Climbing Refinement Algorithm for Graph Partitioning**. In *45th International Conference on Parallel Processing (ICPP)*, pages 236–241, 2016. DOI: 10.1109/ICPP.2016.34.
Cited on pages 22, 45, 48, 101.

[LLC95] Jianmin Li, John Lillis, and C.K. Cheng. **Linear decomposition algorithm for VLSI design applications**. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, pages 223–228. IEEE Computer Society, 1995. DOI: 10.1109/ICCAD.1995.480016.
Cited on pages 156, 158, 178.

[LR04] Kevin J. Lang and Satish Rao. **A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts**. In *Proc. of 10th International Integer Programming and Combinatorial Optimization Conference*. Volume 3064 of LNCS, pages 383–400. Springer, 2004. DOI: 10.1007/978-3-540-25960-2_25.
Cited on page 45.

[MABP21] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. **BiPart: A Parallel and Deterministic Hypergraph Partitioner**. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–174, 2021. DOI: 10.1145/3437801.3441611.
Cited on pages 6, 21, 43, 48, 107, 121.

[May+18] 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.
Cited on pages 23, 48, 177.

[Meh84] Kurt Mehlhorn. **Data Structures and Algorithms 1: Sorting and Searching**. Volume 1 of EATCS Monographs on Theoretical Computer Science. Springer, 1984. DOI: 10.1007/978-3-642-69672-5.
Cited on page 75.

[MLLS17] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. **Spinner: Scalable Graph Partitioning in the Cloud**. In *33rd IEEE International Conference on Data Engineering, ICDE*, pages 1083–1094. IEEE Computer Society, 2017. DOI: 10.1109/ICDE.2017.153.
Cited on page 43.

[MMS09] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. **A new diffusion-based multilevel algorithm for computing graph partitions**. In *Journal of Parallel and Distributed Computing* volume 69:9, pages 750–761, 2009. DOI: 10.1016/j.jpdc.2009.04.005.
Cited on page 171.

[MP14] Zoltán Á. Mann and Pál A. Papp. **Formula Partitioning Revisited**. In *5th Pragmatics of SAT Workshop*, pages 41–56, 2014. DOI: 10.29007/9skn.
Cited on pages 2, 19.

[MP17] Zoltán Ádám Mann and Pál András Papp. **Guiding SAT Solving by Formula Partitioning**. In *International Journal of Artificial Intelligence Tools* volume 26:4, pages 1750011:1–1750011:37, 2017. DOI: 10.1142/S0218213017500117.
Cited on page 2.

[MSS14] Henning Meyerhenke, Peter Sanders, and Christian Schulz. **Partitioning Complex Networks via Size-Constrained Clustering**. In *13th International Symposium on Experimental Algorithms*. Volume 8504 of LNCS, pages 351–363. Springer, 2014. DOI: 10.1007/978-3-319-07959-2_30.
Cited on pages 36, 40.

[MSS17] Henning Meyerhenke, Peter Sanders, and Christian Schulz. **Parallel Graph Partitioning for Complex Networks**. In *IEEE Transactions on Parallel and Distributed Systems* volume 28:9, pages 2625–2638, IEEE, 2017. DOI: 10.1109/TPDS.2017.2671868.
Cited on pages 37, 40, 42, 43.

[Mül+15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. **Cache-Efficient Aggregation: Hashing Is Sorting**. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, pages 1123–1136. ACM, 2015. DOI: 10.1145/2723372.2747644.
Cited on page 15.

[NG04] Mark E. J. Newman and Michelle Girvan. **Finding and Evaluating Community Structure in Networks**. In *Physical Review* volume 69, American Physical Society, Feb. 2004.
Cited on page 38.

[Öhl18] Clemens Öhl. **Algorithm Configuration for Hypergraph Partitioning**. Bachelor Thesis. Karlsruhe Institute of Technology, May 2018.
Cited on page 102.

[OS10] Vitaly Osipov and Peter Sanders. **n-Level Graph Partitioning**. In *18th European Symposium on Algorithms (ESA)*, pages 278–289, 2010. DOI: 10.1007/978-3-642-15775-2_24.
Cited on pages 4, 33, 43, 47, 76.

[PM03] Joachim Pistorius and Michel Minoux. **An Improved Direct Labeling Method for the Max–Flow Min–Cut Computation in Large Hypergraphs and Applications**. In *International Transactions in Operational Research* volume 10:1, pages 1–11, 2003. DOI: 10.1111/1475-3995.00389.
Cited on page 159.

[PM07] David A. Papa and Igor L. Markov. **Hypergraph Partitioning and Clustering**. In *Handbook of Approximation Algorithms and Metaheuristics*. 2007. DOI: 10.1201/9781420010749.ch61.
Cited on pages 19, 31.

[PQ82] Jean-Claude Picard and Maurice Queyranne. **On the Structure of All Minimum Cuts in a Network and Applications**. In *Math. Program.* volume 22:1, page 121, 1982. DOI: 10.1007/BF01581031.
Cited on page 158.

[PS19] Richard John Preen and Jim Smith. **Evolutionary n-Level Hypergraph Partitioning With Adaptive Coarsening**. In *IEEE Transactions on Evolutionary Computing* volume 23:6, pages 962–971, 2019. DOI: 10.1109/TEVC.2019.2896951.
Cited on pages 39, 54.

[PZ21] Feng Pan and Pan Zhang. **Simulating the Sycamore quantum supremacy circuits**. In *arXiv preprint arXiv:2103.03074*, 2021.
Cited on page 2.

[Räc08] Harald Räcke. **Optimal Hierarchical Decompositions for Congestion Minimization in Networks**. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*. Ed. by Cynthia Dwork, pages 255–264. ACM, 2008. DOI: 10.1145/1374376.1374415.
Cited on page 48.

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. **Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks**. In *Physical Review E* volume 76:3, page 036106, APS, 2007. DOI: 10.1103/PhysRevE.76.036106.
Cited on pages 35, 36, 40, 113, 115.

[San89] L. A. Sanchis. **Multiple-Way Network Partitioning**. In *IEEE Transactions on Computers* volume 38:1, pages 62–81, 1989. DOI: 10.1109/12.8730.
Cited on pages 1, 32, 69, 115.

[Sch+16] 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, 2016. DOI: 10.1137/1.9781611974317.5.
Cited on pages 47, 62.

[Sch13] Christian Schulz. **High Quality Graph Partitioning**. PhD thesis. Karlsruhe Institute of Technology, 2013.
Cited on page 49.

[Sch20] Sebastian Schlag. **High-Quality Hypergraph Partitioning**. In 2020. DOI: 10.5445/IR/1000105953.
Cited on pages 6, 7, 19, 20, 27, 31, 32, 33, 34, 39, 48, 55, 79, 81, 86, 95, 106, 128, 138.

[SCS19a] Ruslan Shaydulin, Jie Chen, and Ilya Safro. **Relaxation-Based Coarsening for Multilevel Hypergraph Partitioning**. In *Multiscale Modeling and Simulation* volume 17:1, pages 482–506, 2019. DOI: 10.1137/17M1152735.
Cited on pages 22, 48, 177.

[SCS19b] Ruslan Shaydulin, Jie Chen, and Ilya Safro. **Relaxation-Based Coarsening for Multilevel Hypergraph Partitioning**. In *Multiscale Modelling and Simulation* volume 17:1, pages 482–506, 2019. DOI: 10.1137/17M1152735.
Cited on page 37.

[Sen01] Norbert Sensen. **Lower Bounds and Exact Algorithms for the Graph Partitioning Problem Using Multicommodity Flows**. In *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*. Ed. by Friedhelm Meyer auf der Heide. Volume 2161 of Lecture Notes in Computer Science, pages 391–403. Springer, 2001. DOI: 10.1007/3-540-44676-1_33.
Cited on page 49.

[Ser+16] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. **Clay: Fine-Grained Adaptive Partitioning for General Database Schemas**. In *Proceedings of the VLDB Endowment* volume 10:4, pages 445–456, VLDB Endowment, 2016. DOI: 10.14778/3025111.3025125.
Cited on page 2.

[Sha+16] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. **Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks**. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. Ed. by Katerina J. Argyraki and Rebecca Isaacs, pages 455–468. USENIX Association, 2016.
Cited on page 2.

[SK12] Isabelle Stanton and Gabriel Kliot. **Streaming Graph Partitioning for Large Distributed Graphs**. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2012. DOI: https://doi.org/10.1145/2339530.2339722.
Cited on page 50.

[SK72] Daniel G. Schweikert and Brian W. Kernighan. **A Proper Model for the Partitioning of Electrical Circuits**. In *Proceedings of the 9th Design Automation Workshop, DAC '72, Dallas, Texas, USA, June 26-28, 1972*. Ed. by Harlow Freitag, J. Michael Galey, Robert B. Hitchcock Sr., J. M. Saindon, Herbert M. Wall, Donald J. Humcke, and J. R. Hanne, pages 57–62. ACM, 1972. DOI: 10.1145/800153.804930.
Cited on pages 1, 29.

[SM16] Christian L. Staudt and Henning Meyerhenke. **Engineering Parallel Algorithms for Community Detection in Massive Networks**. In *IEEE Transactions on Parallel and Distributed Systems* volume 27:1, pages 171–184, 2016.
Cited on pages 5, 39, 61.

[SMR16] George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. **Complex Network Partitioning Using Label Propagation**. In *SIAM Journal of Scientific Computing* volume 38:5, 2016. DOI: 10.1137/15M1026183.
Cited on page 42.

[Sou35] R. V. Southwell. **Stress-Calculation in Frameworks by the Method of "Systematic Relaxation of Constraints"**. In *Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences* volume 151:872, pages 56–95, The Royal Society, 1935.
Cited on page 34.

[SS11] Peter Sanders and Christian Schulz. **Engineering Multilevel Graph Partitioning Algorithms**. In *19th European Symposium on Algorithms (ESA)*, pages 469–480. Springer, 2011. DOI: 10.1007/978-3-642-23719-5_40.
Cited on pages 7, 33, 36, 39, 44, 45, 68, 153, 157, 158, 174.

[SS12] Peter Sanders and Christian Schulz. **Distributed Evolutionary Graph Partitioning**. In *12th Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 16–29, 2012. DOI: 10.1137/1.9781611972924.2.
Cited on page 39.

[SS13] Peter Sanders and Christian Schulz. **Think Locally, Act Globally: Highly Balanced Graph Partitioning**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Volume 7933 of Lecture Notes in Computer Science, pages 164–175. Springer, 2013. DOI: 10.1007/978-3-642-38527-8_16.
Cited on page 171.

[SSS20] Justin Sybrandt, Ruslan Shaydulin, and Ilya Safro. **Hypergraph Partitioning With Embeddings**. In *IEEE Transactions on Knowledge and Data Engineering*, IEEE, 2020. DOI: 10.1109/TKDE.2020.3017120.
Cited on page 37.

[SSS22] Justin Sybrandt, Ruslan Shaydulin, and Ilya Safro. **Hypergraph Partitioning With Embeddings**. In *IEEE Transactions on Knowledge and Data Engineering* volume 34:6, pages 2771–2782, 2022. DOI: 10.1109/TKDE.2020.3017120.
Cited on page 205.

[SST03] Meinolf Sellmann, Norbert Sensen, and Larissa Timajev. **Multicommodity Flow Approximation Used for Exact Graph Partitioning**. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*. Ed. by Giuseppe Di Battista and Uri Zwick. Volume 2832 of Lecture Notes in Computer Science, pages 752–764. Springer, 2003. DOI: 10.1007/978-3-540-39658-1_67.
Cited on page 49.

[ST17] Christian Schulz and Jesper Larsson Träff. **Better Process Mapping and Sparse Quadratic Assignment**. In *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Ed. by Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman. Volume 75 of LIPIcs, pages 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/LIPIcs.SEA.2017.4.
Cited on page 34.

[ST97] Horst D. Simon and Shang-Hua Teng. **How Good is Recursive Bisection?** In *SIAM Journal of Scientific Computing* volume 18:5, pages 1436–1445, 1997. DOI: 10.1137/S1064827593255135.
Cited on pages 33, 115.

[Ste90] Guy L. Steele. **Making Asynchronous Parallelism Safe for the World**. In *POPL 90*. Ed. by Frances E. Allen, pages 218–231. ACM Press, 1990. DOI: 10.1145/96709.96731.
Cited on pages 6, 107.

[SV82] Yossi Shiloach and Uzi Vishkin. **An O(n^2 log n) Parallel Max-Flow Algorithm**. In *Journal of Algorithms* volume 3:2, pages 128–146, 1982. DOI: 10.1016/0196-6774(82)90013-X.
Cited on page 178.

[SW91] John E. Savage and Markus G. Wloka. **Parallelism in Graph-Partitioning**. In *Journal of Parallel and Distributed Computing* volume 13:3, pages 257–272, 1991. DOI: 10.1016/0743-7315(91)90074-J.
Cited on page 40.

[Tan+17] Tunzi Tan, Jihong Gui, Sainan Wang, Suixiang Gao, and Wenguo Yang. **An Efficient Algorithm for Judicious Partition of Hypergraphs**. In *Combinatorial Optimization and Applications - 11th International Conference, COCOA 2017, Shanghai, China, December 16-18, 2017, Proceedings, Part II*. Ed. by Xiaofeng Gao, Hongwei Du, and Meng Han. Volume 10628 of Lecture Notes in Computer Science, pages 466–474. Springer, 2017. DOI: 10.1007/978-3-319-71147-8_33.
Cited on page 205.

[TDKU21] Fatih Taşyaran, Berkay Demireller, Kamer Kaya, and Bora Uçar. **Streaming Hypergraph Partitioning Algorithms on Limited Memory Environments**. In *arXiv preprint arXiv:2103.05394*, 2021.
Cited on page 50.

[TGRV14] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. **Fennel: Streaming graph partitioning for massive scale graphs**. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014. DOI: https://doi.org/10.1145/2556195.2556213.
Cited on page 50.

[TK04] 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, 2004. DOI: 10.1007/978-3-540-30182-0_79.
Cited on pages 22, 42, 48.

[UB13] Johan Ugander and Lars Backstrom. **Balanced Label Propagation for Partitioning Massive Graphs**. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*. Ed. by Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis, pages 507–516. ACM, 2013. DOI: 10.1145/2433396.2433461.
Cited on pages 40, 42, 43.

[VB05] B. Vastenhouw and R. H. Bisseling. **A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication**. In *SIAM Review* volume 47:1, pages 67–95, 2005. DOI: 10.1137/S0036144502409019.
Cited on pages 22, 48, 177.

[Vis+12] 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, June 2012. DOI: 10.1145/2228360.2228500.
Cited on page 19.

[Vis10] Uzi Vishkin. **Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques**. 2010.
Cited on page 114.

[Wah21] Noah Wahl. **Improving Parallel Refinement Algorithms for Hypergraph Partitioning**. Bachelor Thesis. Karlsruhe Institute of Technology, May 2021.
Cited on page 90.

[Wal03] Chris Walshaw. **An Exploration of Multilevel Combinatorial Optimisation**. In *Multilevel Optimization in VLSICAD*, pages 71–123. Springer US, 2003. DOI: 10.1007/978-1-4757-3748-6_2.
Cited on page 35.

[Wal04] Chris Walshaw. **Multilevel Refinement for Combinatorial Optimisation Problems**. In *Annals of Operations Research* volume 131:1, pages 325–372, 2004. DOI: 10.1023/B:ANOR.0000039525.80601.15.
Cited on page 37.

[WC00] Chris Walshaw and Mark Cross. **Parallel Optimisation Algorithms for Multilevel Mesh Partitioning**. In *Parallel Computing* volume 26:12, pages 1635–1660, 2000. DOI: 10.1016/S0167-8191(00)00046-6.
Cited on page 45.

[WW93] Dorothea Wagner and Frank Wagner. **Between Min Cut and Graph Bisection**. In *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*. Ed. by Andrzej M. Borzyszkowski and Stefan Sokolowski. Volume 711 of Lecture Notes in Computer Science, pages 744–750. Springer, 1993. DOI: 10.1007/3-540-57182-5_65.
Cited on page 47.

[YW96] Hannah Honghua Yang and D.F. Wong. **Efficient Network Flow Based Min-Cut Balanced Partitioning**. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* volume 15:12, pages 1533–1540, 1996. DOI: 10.1007/978-1-4615-0292-0_41.
Cited on pages 1, 7, 153, 155, 156, 161.

[YWCC18] Wenyin Yang, Guojun Wang, Kim-Kwang Raymond Choo, and Shuhong Chen. **HEPart: A Balanced Hypergraph Partitioning Algorithm for Big Data Applications**. In *Future Generation Computer Systems* volume 83, pages 250–268, Elsevier, 2018. DOI: 10.1016/j.future.2018.01.009.
Cited on page 2.

List of Figures

List of Tables

List of Algorithms

List of Publications

Journal Articles

[1] **More recent advances in (hyper) graph partitioning**. In *ACM Computing Surveys*, 2022. Ümit Çatalyürek, Karen Devine, Marcelo Faraj, *Lars Gottesbüren*, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier and Dorothea Wagner.

[2] **High-Quality Hypergraph Partitioning**. In *ACM Journal of Experimental Algorithmics (JEA)*, 2022. Sebastian Schlag, Tobias Heuer, *Lars Gottesbüren*, Yaroslav Akhremtsev, Christian Schulz and Peter Sanders.

[3] **Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies**. In *Algorithms*, 2019. *Lars Gottesbüren*, Michael Hamann, Tim Niklas Uhl and Dorothea Wagner.

Articles in Conference Proceedings

[1] **Deterministic Parallel Hypergraph Partitioning**. In *European Conference on Parallel Processing 2022 (EuroPar). Lars Gottesbüren* and Michael Hamann.

[2] **Parallel Flow-Based Hypergraph Partitioning**. In *Symposium on Experimental Algorithms 2022 (SEA). Lars Gottesbüren*, Tobias Heuer and Peter Sanders.

[3] **A branch-and-bound algorithm for cluster editing**. In *Symposium on Experimental Algorithms 2022 (SEA).* Thomas Bläsius, Philipp Fischbeck, *Lars Gottesbüren*, Tobias Heuer, Michael Hamann, Jonas Spinner, Christopher Weyand and Marcus Wilhelm.

[4] **Shared-Memory n-level Hypergraph Partitioning**. In *Proceedings of the Symposium on Algorithm Engineering and Experiments 2022 (ALENEX). Lars Gottesbüren*, Tobias Heuer, Peter Sanders and Sebastian Schlag.

[5] **Deep Multilevel Graph Partitioning**. In *European Symposium on Algorithms 2021 (ESA). Lars Gottesbüren*, Tobias Heuer, Peter Sanders, Daniel Seemaier and Christian Schulz.

[6] **PACE Solver Description: The KaPoCE Exact Cluster Editing Algorithm**. In *Internation Symposium on Parameterized and Exact Computation 2021 (IPEC).* Thomas Bläsius, Philipp Fischbeck, *Lars Gottesbüren*, Tobias Heuer, Michael Hamann, Jonas Spinner, Christopher Weyand and Marcus Wilhelm.

[7] **PACE Solver Description: The KaPoCE Heuristic Cluster Editing Algorithm**. In *Internation Symposium on Parameterized and Exact Computation 2021 (IPEC).* Thomas Bläsius, Philipp Fischbeck, *Lars Gottesbüren*, Tobias Heuer, Michael Hamann, Jonas Spinner, Christopher Weyand and Marcus Wilhelm.

[8] **Scalable Shared-Memory Hypergraph Partitioning**. In *Symposium on Algorithm Engineering and Experiments 2021 (ALENEX). Lars Gottesbüren*, Tobias Heuer, Peter Sanders and Sebastian Schlag.

[9] **Advanced Flow-Based Multilevel Hypergraph Partitioning**. In *Symposium on Experimental Algorithms 2020 (SEA). Lars Gottesbüren*, Michael Hamann, Sebastian Schlag and Dorothea Wagner.

[10] **Engineering Exact Quasi-Threshold Editing**. In *Symposium on Experimental Algorithms 2020 (SEA). Lars Gottesbüren*, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner and Sven Zühlsdorf.

[11] **Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm**. In *European Symposium on Algorithms 2019 (ESA). Lars Gottesbüren*, Michael Hamann and Dorothea Wagner.