Karlsruhe Institute of Technology

A Parallel Network Flow-Based Refinement Techinque for Multilevel Hypergraph Partitioning

Master's Thesis of

Lukas Reister

at the Department of Informatics
Institute of Theoretical Informatics, Algorithmics

Reviewer: Prof. Dr. Peter Sanders
Advisor: M.Sc. Tobias Heuer

Date of submission: 19 August 2020

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 19 August 2020

. . . . . .

(Lukas Reister)

Abstract

Recently, a refinement technique that uses max-flow-min-cut computations to improve a k-way partition of a hypergraph was presented. The algorithm was integrated into the sequential hypergraph partitioner KaHyPar and outperforms other state-of-the-art partitioner on a wide range of benchmarks with the drawback of significantly higher running times. In this thesis we present a parallel flow-based refinement algorithm and integrate it in the shared-memory multilevel hypergraph partitioner Mt-KaHyPar.

To improve a given k-way partition, the sequential algorithm grows a region around the cut of two adjacent blocks and improves the cut between them using a max-flow computation on a flow network induced by that region. We make use of the fact, that for $k > 2$ multiple of these pairwise refinement calculations can be executed in parallel. We work out the theoretical foundations to determine which of these pairwise refinements are applicable for a parallel execution. Based on that, we present multiple techniques to schedule the pairwise refinements in parallel and improve the running time of the algorithm. We start with a simple approach, only executing independent calculations on disjoint block-pairs in parallel. To further improve the scaling, we then loosen the restrictions and allow multiple flow calculations that include the same block simultaneously. We further integrate our parallel refinement algorithm in Mt-KaHyPar and combine it with other move-based approaches.

For $k \geq 16$ and 16 threads, we achieve a harmonic mean speedup of 5.68 for the flow-based refinement. Compared to the quality preset of PaToH, we achieve a better solution quality on 67,2% of our benchmark instances while being the factor of 1.95 faster by using 10 threads. We also achieve a comparable solution quality as hMetis-R while we are an order of magnitude faster.

Zusammenfassung

Kürzlich wurde ein Algorithmus vorgestellt, der Max-Flow-Min-Cut-Berechnungen verwendet um eine k-teilige Partition eines Hypergraphen zu verbessern. Der Algorithmus wurde in den sequentiellen Hypergraph-Partitionierer KaHyPar integriert und liefert bessere Ergebnisse als andere moderne Partitionierer für einen großen Teil von Benchmark Instanzen mit dem Nachteil deutlich höherer Laufzeiten. In dieser Arbeit stellen wir einen parallelen flussbasierten Algorithmus vor und integrieren ihn in den parallelen Multilevel-Hypergraph-Partitionierer Mt-KaHyPar.

Um eine gegebene Partition zu verbessern, konstruiert der sequentielle Algorithmus einen Bereich um den *cut* zweier benachbarter Blöcke und verbessert die Partition zwischen ihnen unter Verwendung einer Max-Flow-Berechnung in einem durch diesen Bereich induzierten Flussnetzwerk. Wir nutzen die Tatsache, dass für $k > 2$ mehrere dieser paarweisen Berechnungen parallel ausgeführt werden können. Wir erarbeiten die theoretischen Grundlagen, um zu entscheiden, welche dieser paarweisen Berechnungen für eine parallele Ausführung geeignet sind. Basierend darauf präsentieren wir mehrere Techniken, um die paarweisen Berechnungen parallel auszuführen und die Laufzeit des Algorithmus zu verbessern. Wir beginnen mit einem einfachen Ansatz, bei dem nur unabhängige Berechnungen auf disjunkten Blockpaaren parallel ausgeführt werden. Um die Skalierung weiter zu verbessern, lockern wir dann die Voraussetzungen und erlauben mehrere Flussberechnungen, die denselben Block enthalten. Desweiteren integrieren wir unseren parallelen Algorithmus in Mt-KaHyPar und kombinieren ihn mit anderen bewegungsbasierten Ansätzen.

Für $k \geq 16$ und 16 Threads erreichen wir einen mittleren harmonischen Speedup von 5,68 für die flussbasierten Berechnungen. Im Vergleich zur Qualitätskonfiguration von PaToH erzielen wir auf 67,2% unserer Benchmark-Instanzen eine bessere Partitions-Qualität, während wir mit 10 Threads um den Faktor 1,95 schneller sind. Wir erreichen außerdem eine vergleichbare Qualität der Partitionen wie hMetis-R, während wir deutlich schneller sind.

Contents

1. Introduction

A hypergraph is a generalization of a graph, where a (hyper)edge connects an arbitrary amount of nodes instead of two. The *hypergraph partitioning problem* is about to partition a hypergraph into k disjoint blocks of a bounded size ($\leq 1 + \epsilon$ times the average block size) while we simultaneously want to minimize an objective function.

Fields of application for the hypergraph partitioning problem are *VLSI* design [31], simplifying *SAT* formulas [35, 37] and prallelizing sparse matrix-vector multiplication [6]. The challenge of *VLSI* design is to divide a circuit in two or more blocks and simultaneously keep the wires required to connect the elements in different blocks as short as possible. This reduces signal delays, wiring cost and the total layout area. As wires can connect more than two electrical circuit elements, a hypergraph models the problem more accurately than a graph. To help solving *SAT* formulas, hypergraph partitioning can be used to decompose them into smaller subformulas, that are easier to solve [35].

Solving the hypergraph partitioning problem is known to be NP-hard [33]. Thus to obtain a partition in a practicable amount of time, heuristics are used. The heuristic used by most state of the art hypergraph partitioning sytems is the multilevel paradigm [40, 28, 7, 41, 11]. The main idea is to shrink the hypergraph successively to create a hierarchy of hypergraphs (*coarsening phase*). On the smallest hypergraph an initial partition is obtained using more sophisticated techniques (*initial partitioning phase*). Then we traverse backwards through the levels of the hierarchy. On each level of the hierarchy, a refinement algorithm is used to improve the partition quality (*refinement phase*).

Recently, Heuer and Schlag [24] introduced a refinement algorithm that makes use of the max-flow min-cut theorem [16]. The refinement is done by growing a region around the cut of two adjacent blocks and calculate a minimum (s, t)-cut induced by a maximum flow on a flow network corresponding to that region. They combined their approach with the classical move-based FM algorithm [15] and showed that the algorithm produces the best partition quality for a wide range of applications. The main advantage of a flow-based refinement over the traditional move-based techniques is that it provides a more global view on the problem and does not tend to get stuck in local minima. While it can significantly improve the solution quality, a maximum flow calculation can incur high computational overheads. Therefore, lower running time of the algorithm is an interesting avenue of research. As hypergraph partitioning in many cases is a trade-off between running time and solution quality, decreasing the running time of the algorithm opens up potential to further improve the solution quality.

1.1. Problem Statement

The biggest drawback of the flow-based refinement algorithm by Heuer and Schlag [24] is its high running time, we want to tackle this problem by using parallelism of nowadays available

shared-memory systems. The fundamental question of this thesis is how parallelism can be used to speedup a flow-based refinement technique while simultaneously keep the benefits of a good partition quality. We therefore want to develop a parallel framework for flow-based refinement algorithms using shared memory parallelism. The algorithm by Heuer and Schlag [24] will be used as a basis to our work. The goal is to improve the running time of the algorithm while preserving the partition quality and achieve a good scaling when executing on multiple cores.

1.2. Contributions

We present a parallel flow-based refinement algorithm that executes multiple pairwise refinements on adjacent blocks in parallel. We work out the theoretical foundations to determine which of these pairwise refinements are applicable for a parallel execution. Based on that, we present multiple techniques to execute the pairwise calculations in parallel, starting by only executing independent calculations on disjoint block-pairs in parallel. To improve the scaling, we then loosen the restrictions and allow multiple flow calculations that include the same block simultaneously. We introduce several techniques to prevent or handle the side effects, that occur in a parallel execution. To achieve this we adjust parts of the sequential algorithm. We will show with experiments, that we improved the running time of the algorithm while preserving the partition quality. As we only execute flow calculations on block-pairs in parallel, our approach only improves the running time for $k > 2$.

We integrate our techniques in the parallel hypergraph partitioner Mt-KaHyPar that is currently under development. We analyze the impact of combining our algorithm with other refinement techniques that are implemented in Mt-KaHyPar and determine an optimal configuration. Finally we compare our algorithm with other state of the art hypergraph partitioning systems in terms of running time and partition quality. Using 10 threads, we achieve better solution quality on 67.2% of the instances when compared to the quality preset of PaToH, while being a factor of 1.95 faster. Compared directly to hMetis-R, we are an order of magnitude faster and still achieve a comparable solution quality. For $k \geq 16$ and 16 threads, we achieve a harmonic mean speedup of 5.68 for the flow-based refinement.

1.3. Outline

We start by introducing the basic principles, that are used throughout this thesis in Chapter 2 and present the related work in Chapter 3. Afterwards we introduce our parallel framework in Chapter 4. Additionally we present the multiple techniques used to realize a parallel execution and the integration in Mt-KaHyPar. In Chapter 5 we present the experimental evaluation of our approaches and conclude this thesis with Chapter 6.

2. Preliminiaries

This Chapter introduces the general definitions and terminology used throughout this thesis. As the research presented in this thesis is strongly build on the work of [23] and [40], the terminology used is similar.

2.1. Graphs

A *directed weighted graph* $G = (V, E, c, \omega)$ consists of two finite Sets V and E and two weight functions c and ω. The elements of V are called *nodes* and the elements of E are called *edges*. An edge $e = (u, v)$ is a relation between two nodes $u, v \in V$. The *node weight function* $c : V \to \mathbb{R}_{\geq 0}$ and the *edge weight function* $\omega : E \to \mathbb{R}_{\geq 0}$ assign non negative weights to the nodes and edges of G.

We call two nodes u and v *adjacent* if there exists an $(u, v) \in E$. If two edges e_1 and e_2 share a node $e_1 \cap e_2 \neq \emptyset$ they are referred to as *incident*. The set of neighbors $\Gamma(v)$ of a node consists of all nodes adjacent to it. The degree d of a node is defined as the size of its neighborhood $d(v) = |\Gamma(v)|$.

2.1.1. Contraction

A *contraction* on a graph G is an operation to merge two nodes together. The result of the contraction of two nodes u and v is $G_{(u,v)} = (V \setminus v, E', c', \omega')$. Every edge $(v, w) \in E$ is replaced with (u, w) in E' and every edge $(w, v) \in E$ is replaced with (w, u) in E'. $c'(u) = c(u) + c(v)$ is the new weight of the contracted node. The edge weights of the transformed edges remain the same.

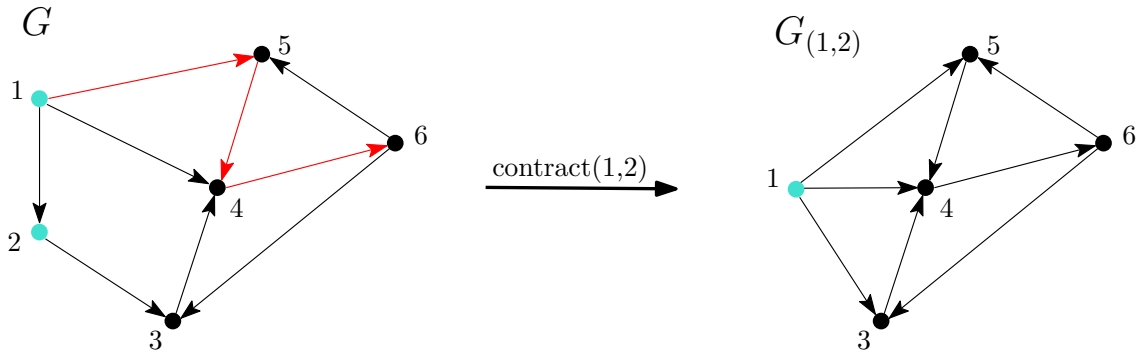

Figure 2.1.: A graph G with a path $\langle 1, 5, 4, 6 \rangle$ on the left side and the corresponding $G_{(1,2)}$ after the contraction of nodes 1 and 2 on the right side.

A *path* $P = \langle v_1, v_2, ..., v_k \rangle$ is a sequence of nodes, where each pair of consecutive nodes is connected by a directed edge. When the first node of a path equals the last node it is called a *cycle*. Figure 2.1 shows a directed graph G with 6 nodes and a path $\langle 1, 5, 4, 6 \rangle$ marked in red. $G_{(1,2)}$ shows the graph after a contraction of the nodes 1 and 2.

A *strongly connected component* (SCC) $C \subseteq V$ is a subset of V, where for each pair of nodes in C there exists a path between them. A directed graph that does not contain any cycles is called an *directed acyclic graph* (DAG). In DAGs it is possible to find an ordering of the nodes, such that a directed edge between two nodes u and v indicates $u \prec v$. Such an ordering is called *topological ordering*.

2.2. Flows

A *flow-network* $G = (V, E, c)$ is a directed Graph with a set of nodes V, a set of edges $E : V \rightarrow V$ and a capacity function $c : E \rightarrow \mathbb{R}_{\geq 0}$. The purpose of a flow-network is to model the flow from a specific *source* $s \in V$ to a *sink* $t \in V$. The flow is modeled with a function $f : E \rightarrow \mathbb{R}_{\geq 0}$ and has to satisfy the following constraints:

1. Capacity constraint: $\forall (u, v) \in E : f(u, v) \leq c(u, v)$
2. Conservation of flow constraint: $\forall v \in V \setminus \{s, t\} : \sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)\in E} f(v, u)$

The capacity constraint guarantees that the flow does not exceed the capacity of the edge. The Conservation of flow constraint ensures that the amount of flows entering a node equals the amount leaving it. The *value* of the flow $|f|$ is the amount that is sent from s to t. It can be measured by the sum of flows leaving the source or the sum of flows entering the sink $|f| = \sum_{(s,v)\in E} f(s, v) = \sum_{(v,t)\in E} f(v, t)$. A flow f is a *maximum flow* if no other flow $|f'| > |f|$ exists.

A useful concept to find a maximum flow of a flow-network are *residual networks*. A residual network to a flow network G with regard to a flow f is defined as $G_f = (V, E_f)$ with:

$$E_f = \{(u, v) \in E | c_f(u, v)\}$$

c_f is called the residual capacity of a node and is defined as:

$$c_f(u, v) = c(u, v) - f(u, v) + f(v, u)$$

The residual capacity models how much more flow can be pushed through an edge $e \in E$ till it is saturated. The residual network also contains reverse edges $\overleftarrow{e} \notin E$ where the residual capacity represents the flow currently pushed through e. A path from s to t in the residual network is called an *augmenting path*. Figure 2.2 illustrates the concept of a flow-network and its corresponding residual-graph. On the left side the flow-network G with the edge capacities and a flow is shown. In the residual-network G_f the flow is illustrated in red and an augmenting path from s to t is highlighted in purple. The current value of the flow in this example is 4.

A *multi-source multi-sink maximum flow problem* is a maximum flow problem with sets of multiple sources S and sinks T instead of a single s and t. The problem is to find a maximum flow from all sources $s \in S$ to all sinks $t \in T$. The problem can be transformed into a maximum

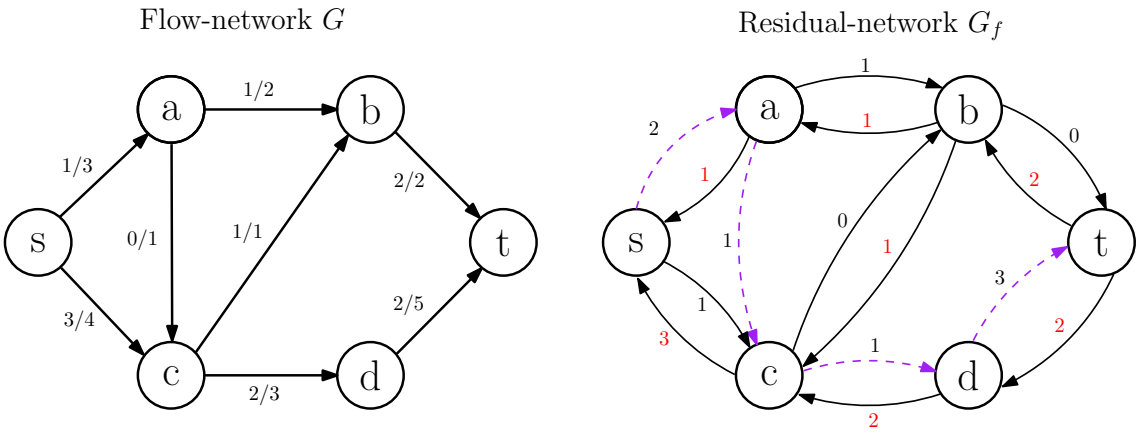

Figure 2.2.: Illustration of concepts related to the maximum flow problem. An example of a flow-network G (left side) and the corresponding residual-network G_f with regard to the flow f ($|f| = 4$).

flow problem with a single source and sink by adding two additional nodes s* and t*. For all $s \in S$ we add an edge $(s*, s)$ with infinite capacity and for all $t \in T$ we add an edge $(t, t*)$ with infinite capacity.

2.2.1. Max-Flow Min-Cut Theorem

The max-flow min-cut theorem is the basis to improve partitions using max-flow calculations:

Theorem 2.2.1 (Max-Flow Min-Cut) *The value of a maximum flow in a flow-network equals the value of a minimum s-t cut.*

A *s-t cut* of a flow-network G is the partition of V in two disjoint sets $(S, V \setminus S)$ with $s \in S$ and $t \notin S$. The value of a s-t cut is defined as:

$$c(S) = \sum_{\substack{(u,v)\in E \\ u\in S \\ v\notin S}} c(u, v)$$

Knowing a minimum flow of a flow-network it is possible to calculate a minimum s-t cut using the residual network [16]. Flow-networks can be used to solve many related problems like maximum bipartite matching or finding a minimum vertex separator. To solve these problems a specific transformation of the Graph to a flow-network is sometimes necessary. A fundamental problem used in [23] is the minimum-weight s-t vertex separator problem.

Definition 2.2.2 (Vertex Seperator Problem) *Let $G = (V, E, c)$ be a graph with the node weight function $c : V \to R_{>0}$. A subset $S \subset V$ is called a s-t vertex separator, when after the removal of all nodes contained in S, s and t are separated in the resulting graph. The weight of a separator $c(S) = \sum_{v\in S} c(v)$ is the sum of its node weights. A vertex separator S is minimal when there exists no S' with $c(S') < c(S)$.*

To find a minimum-weight s-t vertex separator we can use the following transformation to a flow network [23, 45]:

Definition 2.2.3 (Vertex Seperator Transformation) *Let T_V be a transformation of a graph $G = (V, E, c)$ into a flow network $T_V(G) = (V_V, E_V, u_V)$ with $(u_V : E_V \rightarrow \mathbb{R}_{>0})$. T_V is defined as follows:*

- $V_V = \bigcup_{v \in V} \{v', v''\}$
- $\forall v \in V$: *add a directed edge* (v', v'') *with capacity* $u_V(v', v'') = c(v)$
- $\forall (u, v) \in E$: *add two directed edges* (u'', v') *and* (v'', u') *with capacity* $u_V(u'', v') = u_V(v'', u') = \infty$

To obtain the vertex separator we need to get a minimum cut of the network by finding a maximum flow. Only edges of the form (v', v'') will be in the minimum cut set, as all other edges have infinite capacity. The vertex separator is consisting of the corresponding nodes to the cut set.

2.2.2. Max-Flow Algorithms

There exist two main families of algorithms to solve the maximum flow problem. One are the preflow-based algorithms which are based on the push relabel algorithm by Goldberg and Tarjan [18]. The other family is based on augmenting paths in residual networks. They make direct use of the following theorem:

Theorem 2.2.4 (Ford-Fulkerson) *f is a maximum flow if there exists no augmenting path in G_f [16].*

The theorem can easily be understood, as an augmenting path is a path from s to t containing only unsaturated edges. It is therefore possible to increase the flow on an augmenting path and with that also the flow from s to t. The main idea is to push as much flow as possible through an augmenting path while there still exists one. The *Ford-Fulkersonn algorithm* [17] does exactly that and uses a depth first search to find augmenting paths. The maximum running time of this algorithm is $O(|E||f_{max}|)$. The problem is that there exist instances where the value of the maximum flow is exponential in the problem size [14].

Edmonds and Karp improved the algorithm and presented a polynomial-time version [14]. They used breadth-first search to find augmenting paths and therefore always pushed flow on the shortest existing path. The complexity of this algorithm is $O(|E|^2|V|)$.

Another algorithm with a different approach to find augmenting paths was introduced by Boykov and Kolmogorov [5]. The algorithm was designed for the special application of global energy minimization in computer vision and outperformed all existing algorithms in this category. Augmenting paths are found by growing two spanning trees starting by s and t. All edges contained in the trees are unsaturated. When the trees touch, an augmenting path is found and flow is pushed through it. This causes edges to become saturated and therefore be removed from the tree. The algorithm then tries to reconnect the nodes, that are no longer

connected to the tree, by finding new paths to them. The idea is to restore the tree structure and reuse as much as possible. After that, the trees continue to grow and find new augmenting paths. Because the algorithm does not guarantee to find the shortest augmenting paths it has a worst case running time of $O(|E||V|^2|f_{max}|)$ but still performs remarkably well on many instances.

An extension of the algorithm from Boykov and Kolomogrov is presented in [20] and [19]. The *incremental breath first search algorithm* (IBFS) ensures that the augmenting paths found are always the shortest existing paths and therefore guarantees a polynomial running time of $O(|E||V|^2)$. To achieve this the algorithm stores the distance of nodes to the source and the sink and grows the trees in a breath first manner. The algorithm has been shown to perform comparable and often better than the Boykov Kolomogrov algorithm.

2.3. Hypergraphs

A *hypergraph* is a generalization of a graph, where the edges are not always connecting two nodes but instead connect a arbitrary amount of nodes.

Definition 2.3.1 (Hypergraph) *A weighted undirected hypergraph $H = (V, E, c, \omega)$ is defined as a set of hypernodes V and a set of hyperedges E with hypernode-weights $c : V \to \mathbb{R}_{>0}$ and hyperedge-weights $\omega : E \to \mathbb{R}_{>0}$. Each hyperedge is a subset of V.*

In this thesis we use hypernodes/vertices and hyperedges/nets when referring to hypergraphs and nodes and edges when referring to graphs. The vertices of a net are called *pins*. We extend the weight functions c and ω for sets of hypernodes $V' \subseteq V$ and sets of hyperedges $E' \subseteq E$ as follows: $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. A hypernode v is *incident* to a hyperedge e when $v \in e$. The set of all incident hyperedges of a vertex is called $I(v)$. Vertices are called *adjacent* when there exists a net that has both vertices as a pin. The degree of a hypernode v is $d(v) = |I(v)|$. The size of a net $|e|$ is the number of its pins. Because hyperedges have an arbitrary amount of pins there are multiple concepts to describe subgraphs:

Definition 2.3.2 (Subhypergraph) *A subhypergraph H_A induced by $A \subseteq V$ is defined as $H_A = (A, E', c, \omega)$ with $E' = \{e \cap A | e \in E \wedge e \cap A \neq \emptyset\}$.*

Definition 2.3.3 (Section Hypergraph) *A section hypergraph $H \times A$ is defined as $H \times A = (A, E', c, \omega)$ with $E' = \{e | e \in E \wedge e \subseteq A\}$.*

Both graphs are induced by removing vertices, but a section hypergraph only contains hyperedges, that are fully contained in $A \subseteq V$.

It is possible to represent a hypergraph as an undirected graph. The two most common ways are the *clique* and the *bipartite* representation [27]. In the clique representation each hyperedge is replaced with an edge for each pair of vertices in the hyperedge. In the bipartite representation each hyperedge is replaced with an additional node. The node is connected to each pin of the hyperedge. The node weights c and edge weights ω depend on the problem

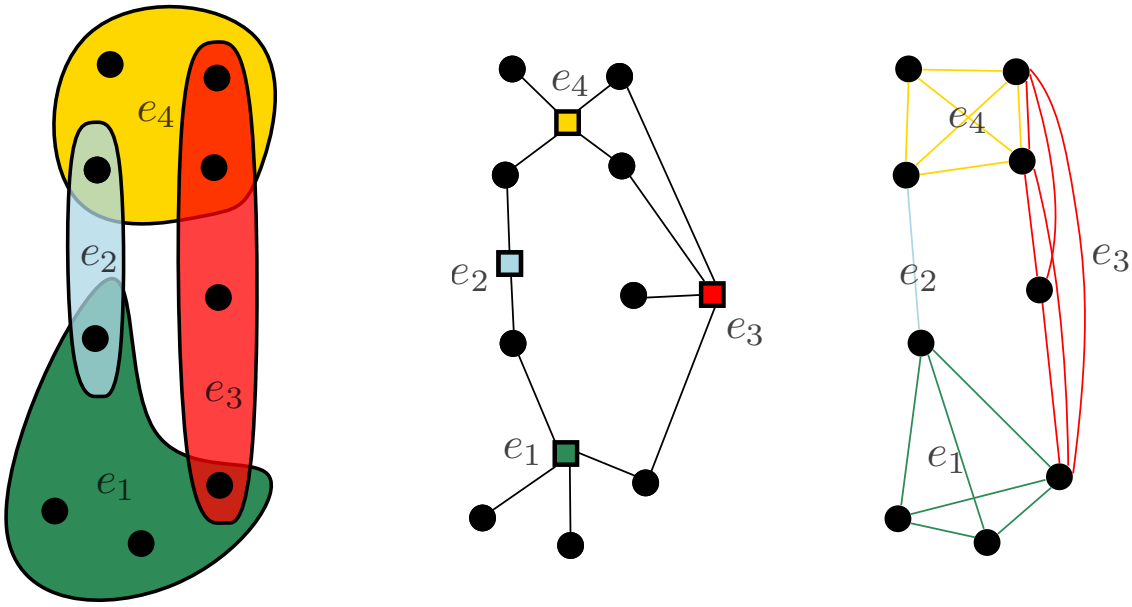

Figure 2.3.: A hypergraph (left) with its corresponding bipartite (middle) and clique (right) representations.

domain. Figure 2.3 shows an example of a hypergraph with 9 hypernodes and 4 hyperedges and the corresponding clique and bipartite representations.

2.4. Hypergraph Partitioning

Definition 2.4.1 (Hypergraph Partition) *The k-way hypergraph partitioning problem is to find an ϵ-balanced k-way partition Π of a hypergraph H that minimizes an objective function over the cut nets.*

A k-way partition is a partition of the vertices of a hypergraph in k disjoint blocks $\Pi = \{V_1, ..., V_k\}$ with $\bigcup_{i=0}^{k} V_i = V$ and $V_i \neq \emptyset$. A k-way partition is ϵ-balanced when every block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$.

The set of blocks that a hyperedge e has pins in, is called *connectivity set* $\Lambda(e, \Pi) = \{V_i \in \Pi | V_i \cap e \neq \emptyset\}$. The *conncetivity* $\lambda(e, \Pi)$ of a net with respect to a partition Π is defined as the cardinality of its connectivity set. Hyperedges with a connectivity greater than one are called *cut nets*. There are several objective functions to minimize over cut nets, the two most common functions are the *cut metric* $\omega_H(\Pi)$ and the *connectivity metric* $(\lambda - 1)_H(\Pi)$[13]. The goal of the cut metric is to minimize the sum of the weight of all cut nets:

$$\omega_H(\Pi) = \sum_{\substack{e \in E \\ \lambda(\Pi, e) > 1}} \omega(e)$$

The connectivity metric also considers how many blocks a cut net is connecting. The connectivity minus one is added as factor to the sum:

$$(\lambda - 1)_H(\Pi) = \sum_{\substack{e \in E \\ \lambda(\Pi,e)>1}} (\lambda(e,\Pi) - 1) * \omega(e)$$

The optimization of both functions is known to be NP-hard [33].

Another concept used a lot in this thesis is the *quotient graph*. It connects all adjacent blocks of a partition. Blocks are called adjacent when there exists a cut net connecting them.

Definition 2.4.2 (Quotient Graph) *$Q = (\Pi, E')$ is a graph which contains an edge between each pair of adjacent blocks of a k-way partition Π of a hypergraph H with $E' = \{(V_i, V_j) | \exists e \in E : V_i, V_j \in \Lambda(e, \Pi)\}$.*

3. Related Work

In this Chapter we give an brief overview of the current state of hypergraph partitioning and introduce the basic concepts used by state of the art hypergraph partitioners. We also show how parallelism is currently used in hypergraph partitioning and how flow-based approaches are used as refinement techniques.

3.1. Hypergraph Partitioning

There exist a wide range of hypergraph partitioning systems and algorithms used in them. The most commonly used systems are KaHyPar[40], hMetis[28] and PaToH[7]. The algorithms presented in this thesis are integrated in the KaHyPar hypergraph partitioner family, more specifically the shared-memory hypergraph partitioner Mt-KaHyPar that is currently under development. Zoltan [11] and Par*k*way [41] are two other partitioners supporting parallelism. All of the named partitioners use the multilevel approach presented in Section 3.1. The partitioners can also differ in the way that they partition the hypergraph in k blocks. The *direct k-way* approach splits the hypergraph directly in k blocks while the *recursive bisection* method splits the graph in two blocks and continues to do so recursively until k blocks are reached. Due to the relevance to this thesis we give a brief introduction to the parallelism used in other systems in Section 3.1.2 and the Mt-KaHyPar framework in Section 3.1.3.

3.1.1. Multilevel Paradigm

As the hypergraph partitioning problem is known to be NP-hard, it is not feasible to calculate a perfect partition. Therefore heuristics are used to find partitions in an acceptable amount of time. The most prominent heuristic used by most partitioning systems is the *multilevel paradigm*. The multilevel paradigm was first introduced by Barnard and Simon [3] to improve the running time of a graph partitioning algorithm. It consists of three main phases, namely the *coarsening phase*, the *initial-partitioning phase* and the *uncoarsening/ refinement phase*.

The idea behind the multilevel paradigm is to shrink the hypergraph using contractions and create a hierarchy of successively smaller instances. Then finding an initial partition on the smallest hypergraph using an algorithm of choice. The last phase is to propagate back through the hierarchy by undoing the contractions. On each level of the the hierarchy a refinement-algorithm is executed to improve the partition. Figure 3.1 shows an illustration of the multilevel paradigm.

To find a good initial partition, the smallest hypergraph should still represent the structure of the input hypergraph. To achieve this matching- or clustering-algorithms are used during the coarsening phase. The algorithm used to find a good initial partition can have a comparatively high running time as it is only executed on a small hypergraph.

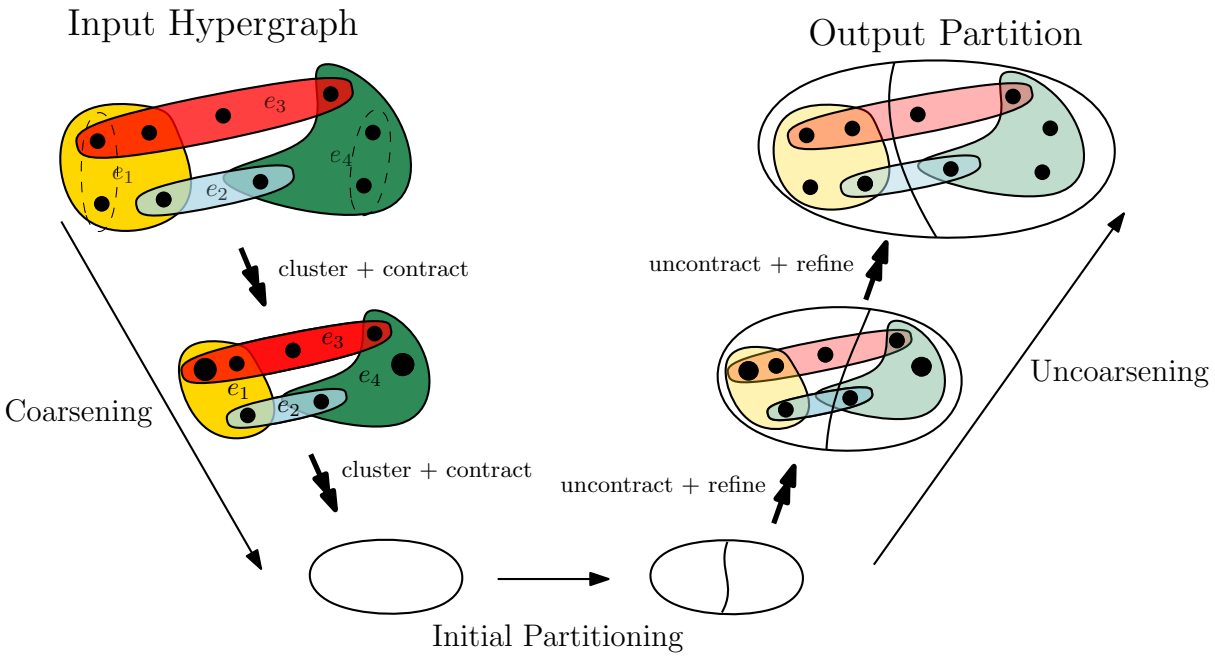

Figure 3.1.: Illustration of the partitioning process using the multilevel paradigm

During the refinement phase, after each uncontractioning-step a refinement algorithm is executed to improve the partition. This is possible as higher levels of the hierarchy offer more degrees of freedom. It is possible to use multiple refinement techniques consecutively or for example use a expensive refinement algorithms on only a few levels. How much level a hierarchy has, varies between the different refinement systems. Most of them use a approximately logarithmic number of levels, but it is also possible to contract only one node per level as shown in the n-level approach by KaHyPar [40]. The number of levels is a trade-off between running time and solution quality.

3.1.2. Parallelism in (Hyper-)Graph Partitioner Systems

In this Section we give a brief overview of other graph and hypergraph partitioning systems using parallelism. As the purpose of this thesis is to construct a parallel refinement algorithm we will mainly focus on this phase. Most partitioners use a distributed memory approach for parallelism. ParMetis [29] and ParHIP [36] are examples for graph partitioners, Zoltan [11] and Par*k*way [41] for hypergraph-partitioners using distributed memory. The only frameworks using shared memory are the graph partitioners MT-Metis[30] and MT-KaHIP[1]. Other than Mt-KaHyPar there currently does not exist any shared memory hypergraph partitioner except for [8] which only executes the coarsening phase in parallel.

In the **coarsening** phase there exist two main approaches to preserve the global structure of the graph. The first one being matching based techniques [8, 11]. The problem with searching for matchings in parallel is that there can be conflicts where two different vertices get matched to the same vertex. To solve this problem in a distributed memory environment an additional

communication step is executed before applying the matchings. In a shared memory environment a global matching vector on which conflicts can be resolved before applying them or a lock-based approach can be used. The second approach is to use a clustering based algorithm. The problem here is to prevent contracted vertices from getting to heavy. The shared memory version presented by [8] uses a locking mechanism for this. ParHIP [36] and MT-KAHIP [1] use an adapted version of the parallel size-constraint label propagation algorithm [26].

In the **initial partitioning** phase it is common to execute the initial partitioning algorithm multiple times as the graph on the coarsest level is comparatively small. Afterwards the partition with the best quality is chosen. In a parallel environment it is common that the sequential algorithm is executed on each processor a defined number of times and the best solution out of all is chosen afterwards.

For the **refinement** phase there exist multiple algorithms to improve the quality on each level of the hierarchy. The most prominent being the *greedy move/ Label Propagation* algorithm. For each boundary vertex a *gain* is calculated. The gain implies how a move of the vertex would affect the quality of the partition. The vertex with the highest gain is selected to move, if the move would not violate the balance constraint. The algorithm is executed in rounds. If a vertex is moved in a round, itself and its neighbors are locked for the rest of the round but become candidates to move in the next round. This greedy algorithm is used by the majority of parallel (hyper)graph partitioners [1, 11, 29, 42, 30, 36].

As moves are executed in parallel there can occur problems. When two adjacent vertices switch their blocks concurrently in different directions, it can have a negative impact on the partition quality despite both having a positive gain value before the move. To prevent this several techniques are used. One way is to apply moves in two alternating phases only allowing vertices to move from blocks with a lower index to a higher one and vise versa in the second round[41, 30]. Another method used by [29] is to compute a coloring of the graph, such that adjacent vertices have a different color. The moves are split in steps where only vertices of the same color can move, to prevent conflicts.

Another problem is to maintain the balance constraint while moving vertices concurrently. In distributed systems the balance can be maintained locally [42]. After each round a communication step between the processors is executed to update the balance. In case of a balance-violation the corresponding moves are reverted. Another solution is to commit the moves to a root processor which applies the moves sequentially after a phase and rejects moves that would violate the balance constraint[41, 11].

MT-Metis [30] extends the greedy algorithm to the *hill-climbing* algorithm. Instead of single moves, a sequence of moves, so called *hills*, are executed. The hills are build using a priority queue associated with a gain. If the move of a complete hill has positive gain it is executed. This technique helps to better escape local minima.

Another refinement algorithm that produces better quality partitions than the greedy one is the *FM* algorithm[15]. The first phase of the algorithm is to find a sequence of feasible moves always using the move with the best gain. In the second phase the moves are reverted back to the prefix with the best quality in that sequence. The problem with parallelizing this algorithm is that the best gain moves are stored in a priority queue which is hard to maintain in parallel. To solve this PT-Scotch [9], KaPPa [26] and Par*k*way [41] execute a sequential 2-way

FM refinement pairwise on adjacent blocks in the quotient graph. The disadvantage of this approach is that the scalability is dependent on the number of blocks.

The only framework using a parallel version of the direct k-way FM algorithm with rollback ability is [1]. They start a local search on each processor using different boundary nodes. After the local search, a root thread recalculates the gains sequentially and rolls back to the partition with the best quality. In Mt-KaHyPar this approach is extended to hypergraphs and improved by also parallelizing the second phase.

Another refinement algorithm that will be introduced in Section 3.2 is the flow-based refinement. There currently does not exist a parallel version of a flow-based refinement for hypergraphs.

3.1.3. Mt-KaHyPar

The algorithms presented in this thesis are integrated into the Mt-KaHyPar framework. Mt-KaHyPar is a hypergraph partitioning system using shared memory parallelism that is currently in development. It uses the multi-level paradigm. All three phases of the multilevel approach are executed in parallel.

In the coarsening phase a parallel clustering based approach is implemented. On each level, the hypergraph is reduced by a factor of up to 2.5, making the amount of levels approximately algorithmic in the size of the hypergraph. The procedure is repeated until the contraction limit $c = 160k$ is reached. In the initial partitioning phase the same approach as in KaHyPar [22] is executed in parallel.

For the refinement, on each level of the hierarchy, two different algorithms are implemented. The first one being a parallel version of the greedy algorithm used by hMetis-K[28] called *size constrained label propagation* in MT-KaHiP [1]. The second algorithm is a parallel version of the FM local-search algorithm [15]. The approach by [1] is extended for hypergraphs and improved by also executing the second phase of the algorithm in parallel. The subject of this thesis is to add a parallel version of the flow-based refinement by Heuer and Schlag [23, 24] as a third refinement algorithm. The parallel algorithm will be presented in Chapter 4.

During all three algorithms the balance constraint can be guaranteed, but intermediate violations to the balance constraint have shown to substantially improve the solution quality. To make sure that the final partition is balanced an additional rebalancing step is executed at the end.

3.2. Flow-Based Refinement

In this Section we will show how flow-based approaches are used as a refinement algorithm for hypergraphs. We will first summarize how flows are used for refinement on graphs an then show how the approach was generalized and improved for hypergraphs.

3.2.1. Flow-Based Refinement for Graphs

The max flow min cut theorem 2.2.1 naturally suggests to use max flow calculations to improve a cut-based metric of graph partition. Sanders and Schulz use a flow-based refinement in

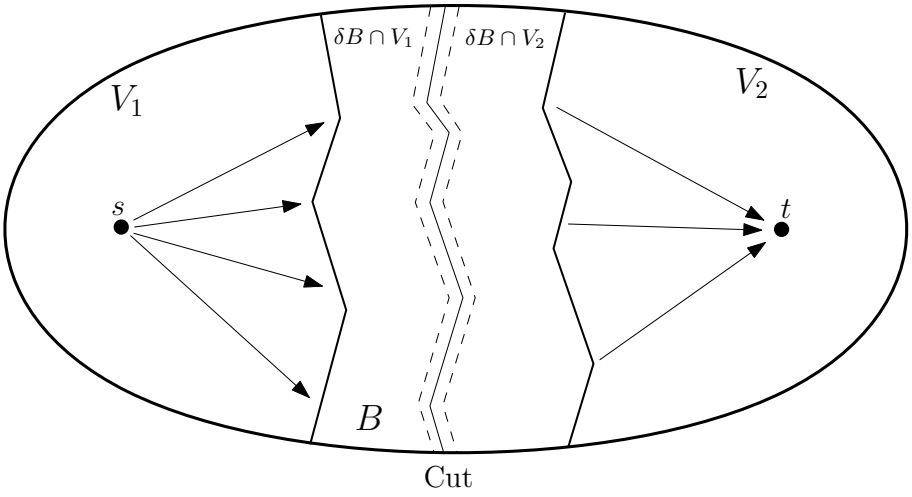

Figure 3.2.: Illustration of the flow-network build by KaFFPa [39].

the multilevel graph partitioner KaFFPa [39]. They extract a region around the cut between two blocks and construct a flow network corresponding to that region. They improve the cut between the blocks using a max-flow computation on the flow network. The region is build using two Breath-First-Searches (BFS) in the involved blocks. The BFS is initialized with all nodes adjacent to the cut nets connecting both blocks and stops when the weight of the nodes would exceed $(1+\epsilon)\lceil\frac{c(V)}{2}\rceil - c(V_2)$. The upper bound for the weight of the region ensures that the balance constraint still holds after the Max-Flow-Min-Cut computation.

For a region $B \subseteq V$ they define its *border* $\delta B = \{u \in B | \exists (u,v) \in E : v \notin B\}$. To construct the flow network all border nodes of the first block $\delta B \cap V_1$ are connected to a source node and all border nodes of the second block $\delta B \cap V_2$ are connected to a sink node. This ensures that no additional edge becomes a cut edge and the max-flow-min-cut computation results in a partition with an equal or better quality than before. Figure 3.2 illustrates the region and the flow network build by [39].

3.2.1.1. Adaptive Flow Iterations

Sanders and Schulz introduced several techniques to improve the flow-based refinement. One is to execute the maximum flow calculation multiple times and adapt the size of region depending on the result of the computations. The size depends on an input parameter α. ϵ is replaced with $\epsilon' = \alpha\epsilon$ in the upper bound for the weight of the region. If a calculation found an improvement α is increased to $min\{2\alpha, \alpha'\}$, where α' is a predefined upper bound. If no improvement was found α is decreased to $max\{\frac{\alpha}{2}, 1\}$. This technique is called *adaptive flow iterations*.

3.2.1.2. Active Block Scheduling

Active block scheduling is a technique to use a two way refinement algorithm for direct k-way partitioning. It operates on the quotient graph and sequentially schedules adjacent blocks pairwise. The scheduling is executed in rounds. A block-pair is scheduled in a round if at least one block is active. In the first round all blocks are active and all adjacent block-

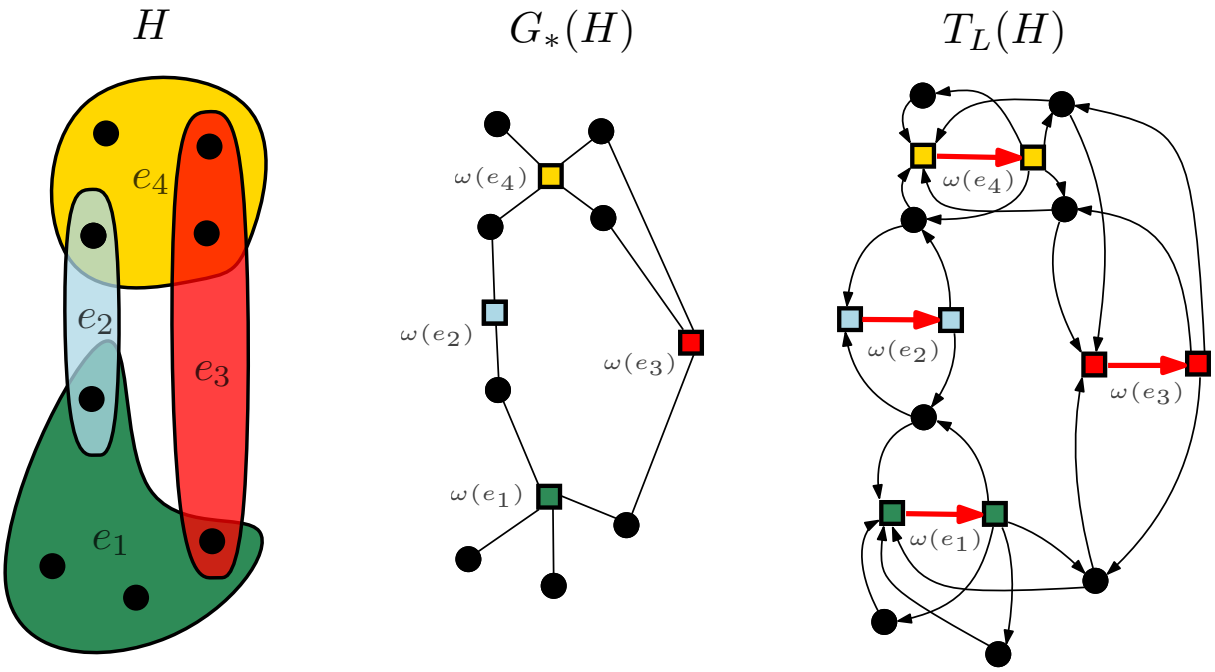

Figure 3.3.: Transformation of a hypergraph H (left) to its bipartite graph representation $G_*(H)$ (middle) and the corresponding Lawler network $T_L(H)$ (right).

pairs are scheduled. When a refinement finds an improvement on a block-pair, both are set active in the next round. The algorithm terminates when there are no more active blocks. As refinement algorithm, any two-way technique can be used, like the FM-algorithm or a flow-based refinement.

3.2.2. Flow-Based Refinement for Hypergraphs

To use a flow-based algorithm for the refinement of a hypergraph partition, parts of the hypergraph need to be transformed to flow networks. The problem of finding a minimum (s,t)-cutset of a hypergraph can be transformed to the problem of finding a minimum (s,t)-vertex separator in its bipartite graph representation $G_*(H)$. In $G_*(H)$ each node representing a hypernode of H has infinite capacity and each node representing a hyperedge has its edge-weight $\omega(e)$ as capacity. Lawler [32] presented a way to transform the vertex separator problem to a flow problem:

Definition 3.2.1 (Lawler Transformation) *Let T_L be the transformation of a hypergraph $H = (V, E, c, \omega)$ into a flow network $T_L(H) = (V_L, E_L, c_L)$ proposed by Lawler [32]. $T_L(H)$ is defined as follows:*

- $V_L = V \cup \bigcup_{e \in E} \{e', e''\}$
- $\forall e \in E$: *we add a directed edge (e', e'') with capacity $c_L(e', e'') = \omega(e)$*
- $\forall v \in V$ *and* $\forall e \in I(v)$: *we add tow directed edges (v, e') and (e'', v) with capacity $c_L(v, e') = c_L(e'', v) = \infty$*

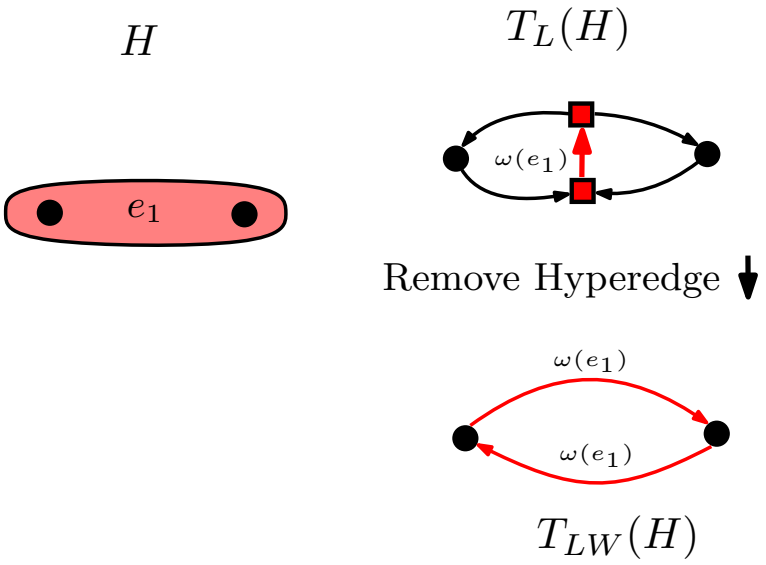

Figure 3.4.: Illustration of the removal of a hyperedge with two pins in the Lawler network and replacing it with graph-edges.

By finding a maximum flow in $T_L(H)$ we are able to obtain a minimum cut in $T_L(H)$. The minimum cut can be mapped directly to a minimum (s,t)-vertex separator of $G_*(H)$ and therefore minimizes the cut metric in H. Figure 3.3 illustrates the transformations.

In Section 2.2.2 we explained that the runtime of max-flow algorithms strongly depends on the number of nodes and edges in the flow-network. Liu and Wong [34] and Heuer and Schlag [24] improve the Lawler network by introducing and combining multiple techniques to reduce the number of nodes and edges in the network.

[34] shows that it is possible to remove the corresponding nodes e' and e'' of a hyperedge with two pins in the Lawler network. Let v and u be the hypernodes connected by the hyperedge e in H, then two directed graph-edges (v,u) and (u,v) with the capacity $c(v,u) = c(u,v) = \omega(e)$ are added to the flow-network instead. The different transformations are illustrated in Figure 3.4. We refer to the network that is retrieved with this method as $T_{LW}(H)$.

Another technique introduced by [24] is to remove hypernodes from the network and replace them with *shortcut* edges. Figure 3.5 illustrates the removal of hypernode with three adjacent hyperedges. To minimize the number of nodes and edges [24] proposes the network $T_L(H, V_d(3))$ where all hypernodes with a degree less or equal than three are removed.

[24] combines the two approaches to be effective on all types of hypergraphs. The $T_{Hybrid}(H)$ network makes use of both techniques. All hypernodes v with $d(v) \leq 3$ are removed if they are not adjacent to a hyperedge of size two. All hyperedges of size two are removed as shown in $T_G(H)$.

3.2.3. Max-Flow-Min-Cut Refinement Framework

[24] presents a dircet k-way flow-based refinement algorithm that can be used in a multilevel hypergraph refiner system. The algorithm is executed on pairs of blocks from the quotient graph. The block-pairs are scheduled using the active block scheduling from Section 3.2.1.2.

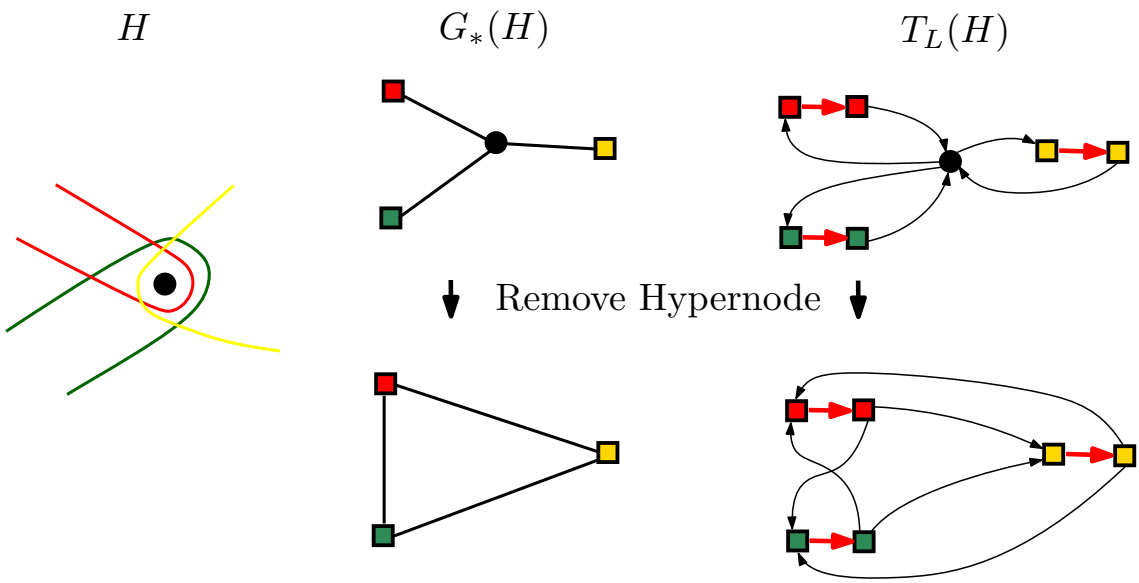

Figure 3.5.: Illustration of the removal of a hypernode and replacing it with shortcut edges.

The algorithm first builds a region B around the cut of the two blocks using two BFS. Then builds a flow network $T(H_B)$ using the techniques from Section 3.2.2 on the subhypergraph $H_B = (V_B, E_B)$ induced by the region B.

To configure the flow problem, two additional nodes s and t are added to the network and connected to some nodes of $T(H_B)$. The sets of nodes that are connected to s and t are called S and T. To chose S and T [24] extends the definition of a border for hypergraphs. They distinguish between *internal* border nodes $\overrightarrow{B} = \{v \in V_B | \exists e \in E : \{u, v\} \subseteq e \wedge u \notin V_B\}$ and *external* border nodes $\overleftarrow{B} = \{u \notin V_B | \exists e \in E : \{u, v\} \subseteq e \wedge v \in V_B\}$. The sources and sinks are chosen to minimize the according metric. For the connectivity metric the sources S and sinks T are chosen as follows:

- $S = \{e' | e \in I(\overleftarrow{B} \cap V_1)\}$
- $T = \{e'' | e \in I(\overleftarrow{B} \cap V_2)\}$

[23] shows that this source and sink configuration guarantees that the refinement does not worsen the partition quality $(\lambda - 1)(\Pi_{new}) \leq (\lambda - 1)(\Pi_{old})$ and the improvement of cut value in the flow network equals the global improvement of the connectivity metric when applied to the hypergraph.

To solve the max flow-problem, multiple max-flow algorithms like the Edmonds Karp algorithm [14] or IBFS [19] are supported. In a *most balanced minimum cut* step a well balanced partition with an optimal quality is chosen. Picard and Queyranne [38] showed that it is possible to compute all minimum (s, t)-cuts of a graph with one maximum (s, t)-flow computation. An adjusted version of the algorithm is used to find a good balanced partition with the min-cut property. If the quality or the balance is improved, the vertices are moved accordingly in the hypergraph. The algorithm additionally uses the approach of adaptive flow iterations from Section 3.2.1.1. As only a minority of the max-flow calculations lead to an improvement several heuristics are used to abort unpromising flow executions and speed up the algorithm.

Gottesbüren [21] further improved the flow based refinement algorithm, by solving a sequence of incremental maximum flow problems directly on the hypergraph. They add vertices to the flow problem dynamically till a partition induced by the max flow calculation satisfies the balance constraint. The algorithm also uses the active block scheduling technique and is executed on block-pairs that correspond to edges of the quotient graph.

4. Parallel Flow-Based Refinement Framework

In Section 3.2.2 we presented a flow-based refinement algorithm for multilevel hypergraph partitioning. In this Chapter we will introduce techniques to parallelize the algorithm for a shared-memory environment. The presented techniques will be integrated in the Mt-KaHyPar framework introduced in Section 3.1.3. In general, our approach is to schedule pairwise flow-based refinements of adjacent blocks in parallel. We will first give a brief introduction to our parallel framework and how it interacts with the different scheduling approaches. We will then present multiple scheduling techniques, starting with a simple approach scheduling only independent block-pairs. We will then loosen the restriction and allow multiple max-flow refinements on one block, to improve the scaling. We present several techniques to handle the problems coming with this and to improve the partition quality of the results.

4.1. Algorithm Overview

In this Section we will introduce the general framework of our algorithm. We will present the workflow of our flow-based refinement, that is executed on each level of the multilevel-hierarchy. We show where and how parallelism is used to speedup the algorithm.

The basis of our parallel algorithm is the flow-based refinement technique by Heuer and Schlag [24] presented in Section 3.2. First, we construct the quotient graph , containing an edge between each adjacent pair of blocks in the partition. With all the block-pairs we execute the active block scheduling strategy from Section 3.2.1.2. On each block-pair we execute the adaptive flow iterations technique presented in Section 3.2.1.1. An iteration includes growing a region around the cut between the involved blocks, building the corresponding flow-network, calculating the maximum flow and finding the most balanced minimum cut induced by the maximum flow. If a flow calculation found an improvement, the moves are applied to the hypergraph and the size of the region is doubled for the next round of flow-based refinement on the same blocks. If no improvement was found, the size of the region is halved in the next round until its size of the region is again equal to the original/starting region.

For $k > 2$ the quotient graph can contain multiple edges and therefore induce multiple pairwise flow refinements per round of the active block scheduling strategy. Our extension of the algorithm is to execute as much as possible of this pairwise refinements in parallel. To decide which block-pairs should be executed in parallel we use a scheduling-interface. The workflow of our framework and the interactions with the scheduler are illustrated in Figure 4.1. At the beginning of each round of the active block scheduling algorithm, the block-pairs that are initially executed in parallel are obtained from the scheduler. After one of these pairwise

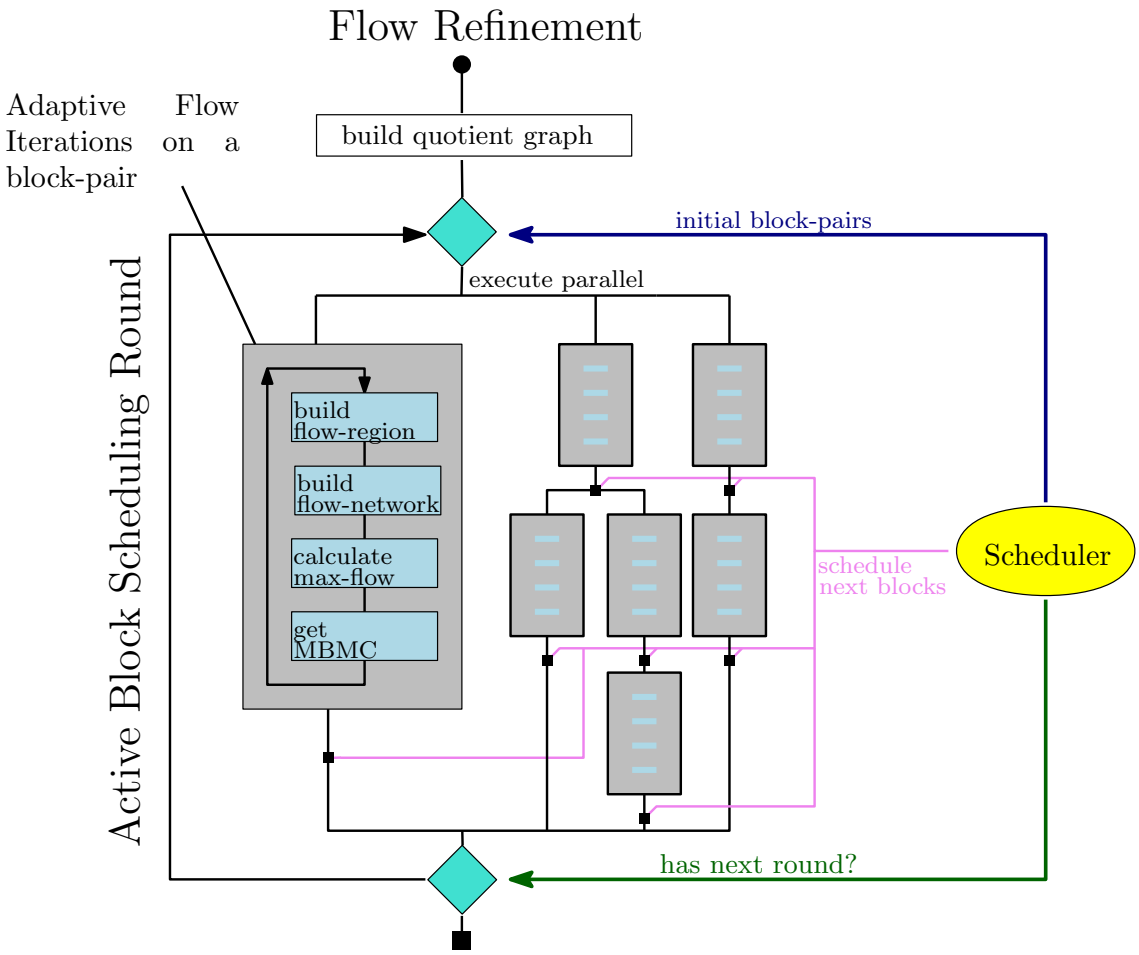

Figure 4.1.: Illustration of the framework for our parallel flow refinement showing interactions with the scheduler.

refinements terminates, the scheduler decides which block-pairs should be scheduled next. We repeat this process for each round of the active block scheduling.

In the following we will introduce multiple scheduling techniques that implement our scheduling interface. Scheduling multiple flow problems in parallel requires that they satisfy some constraint, such that they fullfil the cut property if we apply the resulting min-cuts to the partition. Section 4.2 will introduce our central lemma that multiple flow problem must satisfy if we schedule them in parallel. The Sections 4.2.1 - 4.2.3 discuss parallel scheduling techniques that are derived from our lemma. In Section 4.2.4 we further improve the scaling by removing unnecessary synchronization steps from the algorithm. Section 4.2.5 covers how we need to modify the most balanced minimum cut heuristic, to be able to execute it in parallel.

4.2. Parallel Flow Calculations

In this Section we will introduce our different scheduling techniques. We will show when we can safely execute two flow calculations in parallel and then derive the actual scheduling techniques from that.

To formally describe the flow calculations, we will introduce some terminology that is also used in [24]. We define $H_B = (V_B, E_B)$ as the subhypergraph of $H = (V, E)$ induced by a corridor B computed in the bipartition $\Pi_{i,j} = (V_i, V_j)$. A hypergraph flow problem $\mathcal{F}(B)$ consists of a flow problem $\mathcal{N}_B = (\mathcal{V}_B, \mathcal{E}_B)$ derived from H_B and two additional nodes s and t that are connected to some nodes $v \in \mathcal{V}_B$. A flow problem has the cut property if the resulting min-cut bipartition $\Pi_{\mathcal{F}(B)}$ induced by the max-flow calculation of H_B does not increase the $(\lambda - 1)$-metric when applied to H. This is the case, when applying the moves induced by $\Pi_{\mathcal{F}(B)}$ to the hypergraph results in a less or equal cut value than $cut_H(\Pi_{i,j})$. Additionally we define the set of all border nets $\overleftrightarrow{E_B} = \mathrm{I}(\overleftarrow{B}) \cap \mathrm{I}(\overrightarrow{B})$ for a flow problem $\mathcal{F}(B)$.

To ensure that all flow problems satisfy the cut property in [24], all border nets that contain a pin, which is an external border node and is contained in one of the involved blocks, are added to the source resp. the sink set:

Definition 4.1. *(Sources and Sinks) For a flow problem $\mathcal{F}(B)$ induced by the region B on the block pair (V_i, V_j) with the set of border nets $\overleftrightarrow{E_B}$ and the external border nodes $\overleftarrow{B}$, the source set S and the sink set T are defined as follows:*

- $S = e'$ *for all* $e \in \overleftrightarrow{E_B} : e \subset \overleftarrow{B} \cap V_i$
- $T = e''$ *for all* $e \in \overleftrightarrow{E_B} : e \subset \overleftarrow{B} \cap V_j$

This ensures that each flow problem satisfies the cut property if executed sequential. In parallel, one flow problem can move external border nodes of another flow problem which might change S and T. This can cause a violation of the cut property for a flow problem as illustrated in Figure 4.2. At the beginning (top-left) we have a fraction of a hypergraph that shows three blocks V_1, V_2 and V_3. On the block-pair (V_1, V_2) a flow calculation is executed. The region $B_{1,2}$ induced by the calculation is indicated in blue. v_1 even though it is adjacent to the cut net e_1 is not part of the region. This is possible as it can be part of another region what we will show later. v_1 therefore is part of $\overleftarrow{B} \cap V_1$ and e_1 is part of the source set. On the bottom left

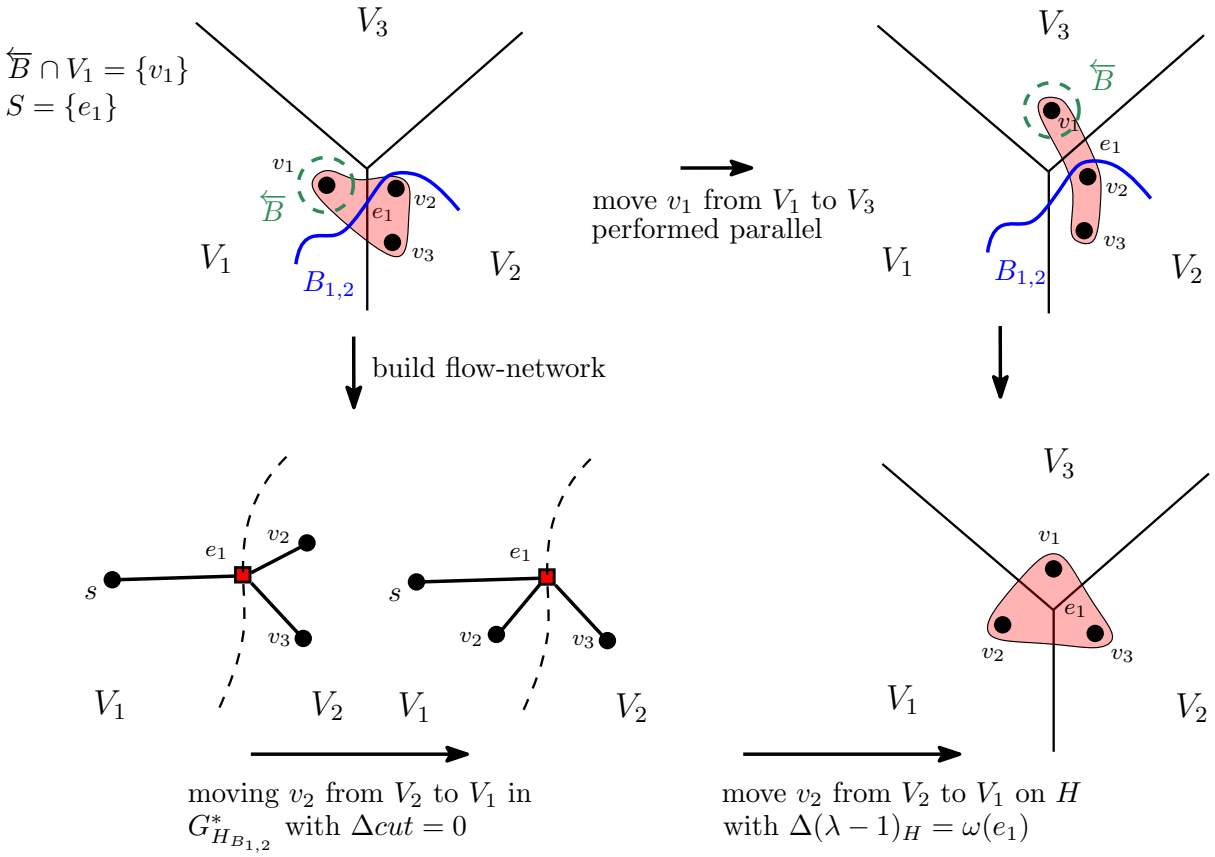

Figure 4.2.: Illustration of the problem with the modification of the source and sink configuration in parallel. The concurrent move of v_1 causes a violation to the cut property of the flow calculation on block-pair (V_1, V_2).

a move induced by the max-flow min-cut calculation is illustrated. The move is shown in the bipartite graph representation corresponding to the flow-network. The move has no impact on the local cut value of the graph. Parallel another flow calculation on the block-pair (V_1, V_3) moves v_1 from V_1 to V_3 (top-right). In this partition v_1 would no longer be part of $\overleftarrow{B} \cap V_1$ and e_1 would not become a source node in the flow graph of (V_1, V_2), but the sources can not get updated in the middle of the calculation. On the bottom right the local move is submitted to the hypergraph, but has a negative impact on the partition quality. Thus the cut property of the flow problem $\mathcal{F}(B_{1,2})$ could be violated.

The following lemma defines properties that two flow problems must satisfy such that both fullfil the cut property if executed in parallel:

Lemma 4.2. *Let $\Pi = \{V_1, \ldots, V_k\}$ be a k-way partition of a hypergraph $H = (V, E)$ and $\mathcal{F}(B_{i,j})$ and $\mathcal{F}(B_{k,l})$ two hypergraph flow problems induced by the corridors $B_{i,j}$ and $B_{k,l}$ computed in the bipartitions $\Pi_{i,j}$ and $\Pi_{k,l}$ with s and t connected to vertices as defined in Defintion 4.1. Applying the resulting min-cut bipartitions $\Pi_{\mathcal{F}(B_{i,j})}$ and $\Pi_{\mathcal{F}(B_{k,l})}$ on Π improves the $(\lambda - 1)$-metric in H if*

1. *$B_{i,j} \cap B_{k,l} = \emptyset$ and*
2. *if $\{i, j\} \cap \{k, l\} \neq \emptyset$ then $\overleftarrow{B}_{i,j} \cap B_{k,l} = \emptyset$ (or vice versa).*

Proof. $B_{i,j} \cap B_{k,l} = \emptyset$ ensures that both flow problems consist of disjoint sets of vertices. Therefore they never move the same vertex. For both flow problmes, $\mathcal{F}(B_{i,j})$ and $\mathcal{F}(B_{k,l})$, if executed sequential , [23] already showed that the cut property is satisfied if the source and sink is configured as defined in Defintion 4.1. Therefore, we have to show that each source and sink configuration is not changed by the other flow calculation. The source and sink configuration of $\mathcal{F}(B_{i,j})$ depends on the sets $\overleftarrow{B}_{i,j} \cap V_i$ and $\overleftarrow{B}_{i,j} \cap V_j$. To affect the sources and sinks of $\mathcal{F}(B_{i,j})$, a move induced by $\mathcal{F}(B_{k,l})$ needs to move a vertex in or out of these volumes. The same holds vice versa. A flow calculation can only induce a move of a vertex that is contained in its corridor. If the block-pairs of the flow problems are disjoint ($\{i, j\} \cap \{k, l\} = \emptyset$), the moves induced by the flow problems can not include a block of the other flow problem and therefore the source and sink configurations of the other problem are not affected.

If the flow problems share a block, we additionally have to show that condition 2.) ensures that $\overleftarrow{B}_{i,j} \cap V_i$ and $\overleftarrow{B}_{i,j} \cap V_j$ are not modified. To move a vertex that is contained in the external border nodes of the other flow problem, an external border node must be part of the corridor of the flow problem, that induces the move. Condition 2.) ensures that none of the external border nodes of one flow problem are contained in the other one. Therefore the external border nodes of both flow problems can not change their blocks. Thus the sets $\overleftarrow{B}_{i,j} \cap V_i$ and $\overleftarrow{B}_{i,j} \cap V_j$ are not effected by moves induced by $\mathcal{F}(B_{k,l})$ and vice versa. The source and sink configuration of $\mathcal{F}(B_{i,j})$ and $\mathcal{F}(B_{k,l})$ are therefore not influenced by each other and both cut properties are still satisfied when executed in parallel. □

With the help of lemma 4.2 we can determine block-pairs on which we can execute flow problems in parallel while still maintaining their cut property. The easiest way to satisfy the conditions of lemma 4.2 is to schedule only disjoint block-pairs in parallel. By doing this $\{i, j\} \cap \{k, l\} = \emptyset$ always holds between two flow calculations $\mathcal{F}(B_{i,j})$ and $\mathcal{F}(B_{k,l})$. If we want to schedule more than one flow problem on a block, we need to modify our flow problems

to make sure that $\overleftarrow{B}_{i,j} \cap B_{k,l} = \emptyset$ or $\overleftarrow{B}_{k,l} \cap B_{i,j} = \emptyset$ holds for all flow calculations $\mathcal{F}(B_{i,j})$ and $\mathcal{F}(B_{k,l})$ that are executed in parallel. We will start by presenting a simple scheduling technique that only executes disjoint block-pairs in parallel.

4.2.1. Parallel Flow Calculations on Disjoint Block-Pairs

As described before, by only scheduling disjoint block-pairs in parallel, lemma 4.2 ensures the cut property for all flow problems. Therefore our first approach aims to schedule a maximum number of pairwise flow-based refinement between blocks that form a matching in the quotient graph. The algorithm starts by computing an initial matching M greedily. All edges $(V_i, V_j) \in M$ are eligible for parallel execution according to Lemma 4.2. Once a pairwise flow-based refinement terminates, our scheduler removes the corresponding edge from M and tries to extend M with the next non-scheduled block-pair. To guarantee that the edges in M form a matching in the quotient graph, our algorithms maintains a vector of size k that indicates whether a block is part of the current matching M or not. Read and writes to that vector are protected via an exclusive lock. The first approach of our parallel algorithm is called *Matching Scheduling*.

4.2.2. Parallel Flow Calculation on All Block-Pairs

The disadvantage of our first approach is that we can only schedule disjoint block-pairs in parallel, which unnecessarily restricts scalability, if the number of blocks is rather small. The advantage of the matching scheduling is, that lemma 4.2 guarantees us the cut property for all scheduled flow problems without further modification to them. But according to lemma 4.2, all block-pairs are feasible for a parallel execution. When two block-pairs share a block, we additionally need to modify our flow problems to still meet the conditions of lemma 4.2.

Typically, the vertices that are involved in a pairwise flow-based refinement are only a fraction of the vertices contained in the corresponding blocks. Therefore, our next approach aims to release the matching restriction and tries to schedule flow problems with disjoint vertex sets. In the following, we will present a postprocessing technique for the flow regions, that ensures the cut property of all flow problems that run in parallel according to lemma 4.2. However, the proposed approach can worsen solution quality. Therefore, we additionally implement a variant that does not aim to satisfy the cut property and show how to handle conflicting moves.

4.2.2.1. Hypernode Ownership

To construct the region of a flow problem we start two BFS around the cut of both involved blocks. To make sure that two different flow calculations do not move the same vertices, we need to make sure that both regions induced by the calculations are disjoint sets of hypernodes. To ensure this, we use an atomic bit set of size $|V|$. Hypernodes are acquired by raising the corresponding bit to one via a compare-and-swap operation. Only if a thread successfully performs the compare-and-swap operation, the vertex is added to the region. After each iteration of the adaptive block scheduling the hypernodes are released and a new region with a different size is grown. Unlike to the sequential algorithm it is no longer guaranteed, that a

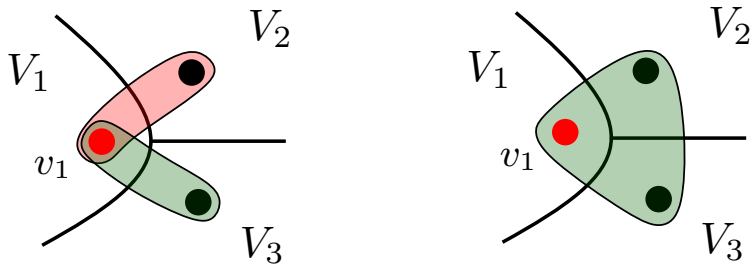

Figure 4.3.: Examples of a hypernode being adjacent to cut nets between multiple block-pairs.

BFS starting with the same vertices, always acquires the same region. Thus the regions built in parallel can differ from the regions built sequential.

The two BFS are initialized with all hypernodes adjacent to cut-nets connecting both blocks involved in the flow calculation. When we execute multiple flow calculations on one block, we can not guarantee that all of these nodes are included in the region. As illustrated in Figure 4.3, a hypernode can be adjacent to cut nets of multiple block-pairs. In both examples v_1 would be element of the initial hypernodes to start the BFS for $B_{1,2}$ and for $B_{1,3}$. As previously stated a hypernode can only be owned by one flow calculation. Such hypernodes are acquired according to the "first come, first serve" principle.

4.2.2.2. Building the Flow-Problems

Without further modification to the flow problems, we can no longer guarantees the cut property for our flow problems with lemma 4.2. The problem is that two flow problems that share a block, can influence the source and sink configuration of each other when executed in parallel. In the following we will present two different approaches to deal with this problem.

The fact that the regions are disjoint vertex sets is not sufficient to guarantee the cut property. Additionally we need to make sure, that if two flow problems share a block, that no flow problem includes an external border node of the other flow problem. To achieve this we exclude some vertices from the regions before building the flow problem. Per definition of the external border nodes, it is sufficient to exclude all external border nodes of flow problems that are currently running, when we build a new flow problem. By doing this we also ensure that a running flow problem does not include an external border node of our new problem.

To formally describe all vertices, that we need to exclude, we define B^* as the set of all regions currently induced by a flow calculation. For a region $B_{i,j}$ we define $B^+(B_{i,j}) := \{B_{k,l} \in B^* \setminus B_{i,j} : \{k,l\} \cap \{i,j\} \neq \emptyset\}$ as all regions that share a block with $B_{i,j}$. Further we define the set:

$$\overleftarrow{B}_{B^+}(B) := \bigcup_{B' \in B^+(B)} \overleftarrow{B}_{B'}$$

For a region B the set $\overleftarrow{B}_{B^+}(B)$ contains all vertices, that are currently external border nodes of flow problems that share a block with B. We ensure the cut-property defined in lemma 4.2 by removing all vertices contained in $B \cap \overleftarrow{B}_{B^+}(B)$ from B. The flow problem induced by B therefore does only contain the vertices $B \setminus B \cap \overleftarrow{B}_{B^+}(B)$ instead of all vertices of B.

We remove the vertices after growing the region and before building the flow network. With this modification and lemma 4.2 follows that all flow problems still have the cut property even in a parallel execution. A problem with the proposed approach is that growing a region and building the according flow network are not atomic operations. We therefore do not have a strict "happens before" relation between the steps. But for a single flow problem, we can always guarantee that its region is fully grown, when it begins to remove vertices from it. Considering two flow problems executed in parallel, the flow problem reaching the vertex removal step later, will therefore always remove vertices after both regions are fully grown. This makes sure that the conditions of lemma 4.2 are always satisfied and both flow problems have the cut property. When both flow problems remove vertices concurrently, it is possible that the flow problem, that reaches the hypernode removal step first, unnecessarily removes vertices.

We determine the vertices, that need to be excluded from a region B, by iterating over all hypernodes $v \in B$ and check for every adjacent hyperedge $e \in I(v)$, if it contains a pin that is acquired by another calculation. Per definition of the external border nodes, such an edge exists for each vertex that we have to exclude from the flow problem. All pins of such an hyperedge, that are part of the region, need to be excluded from the flow network. To realize this, we need to store for each hypernode by which flow calculation it is owned. More specific other flow calculations need to be able to find out by which block-pair a hypernode is owned to check if they share a block. We realize this by considering the nodes with an atomic integer instead of a bit. Zero indicating the vertex is not owned by a flow calculation and $ik + j$ indicating it is owned by the block-pair (V_i, V_j). We guarantee correctness via compare-and-swap operations.

4.2.2.3. Scheduling Multiple Flow Calculations on One Block

Now we present how the actual scheduling is done to minimize the side effects while simultaneously maximizing the degree of parallelism. To schedule a new block-pair, we introduce a new method called `getMostIndependentEdge`. To schedule a new block-pair we keep track of the number of flow calculations scheduled on each block. Let $t(V_i)$ be the number of flow calculations currently running that include V_i. To find the most *independent* block-pair (or edge of the quotient graph), we iterate over all block-pairs that still need to be scheduled in this round and schedule the one that minimizes the following function:

$$dependence(V_i, V_j) = max(t(V_i), t(V_j))$$

The number of threads available for the application is available during run-time as an input parameter. To maximize the utilization of the hardware, we initially schedule as many block-pairs as we have threads available. To determine these block-pairs, we us our new method. When a calculation terminates we schedule exactly one new, if available, using the `getMostIndependentEdge` method again. By doing this we maximize the parallelism while simultaneously minimizing the side effects between the calculations. The new scheduling approach is called *All-Block Scheduling*.

4.2.3. An Optimized Approach for Parallel Flow Calculations on All Block-Pairs

With the matching scheduling approach we can execute independent flow problems without further modification, but limit the scalability. With our second approach we improved the

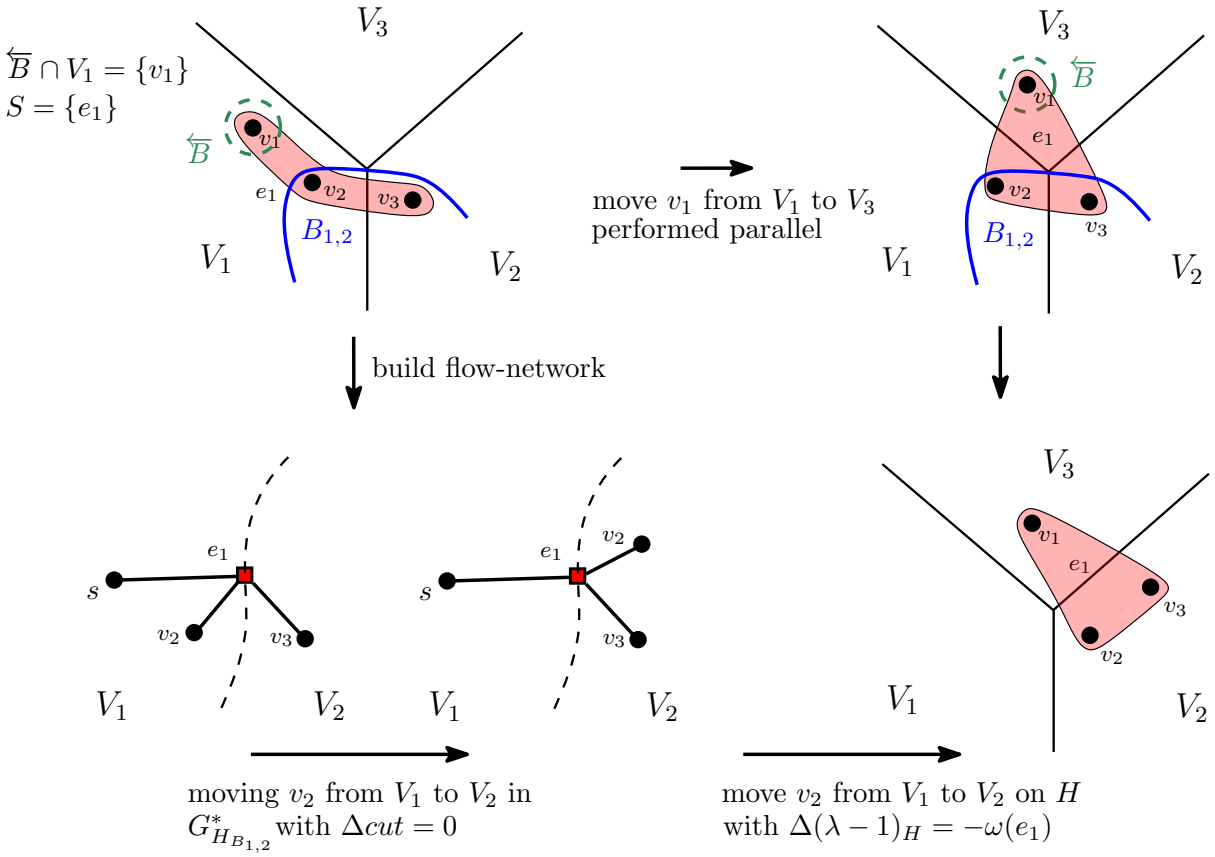

Figure 4.4.: Illustration of a possible positive impact by the modification of the source and sink configuration in parallel. The concurrent move of v_1 leads to an improvement of the connectivity metric in H when the calculation on (V_1, V_2) applies its move, despite the move having no impact on the cut value in $G^*_{H_{B_{1,2}}}$.

scalability, but during test experiments it was observed that it has a noticeable negative impact on the partition quality. We remove more vertices than actually necessary. But it is impossible to predict which vertices will actually be moved by other flow calculations in parallel. It would be desirable to have an approach that combines the best of both approaches.

The reason why we excluded some vertices from the flow network, was that we lost the guarantee that our flow problems satisfy the cut property. In Figure 4.2 we presented an example where this had a negative impact on connectivity metric of the hypergraph. In Figure 4.4 we see an example where this problem can actually have a positive impact on the global partition quality. It shows a slightly different scenario. Note that the move performed by the calculation on (V_1, V_3) (top-right) worsens the connectivity of the partition in our picture. Such a move is still plausible as we only consider a fraction of the hypergraph and the move could remove a heavier edge ,that is not displayed, from the cut. The move of v_2 has no impact on the cut value of the flow network, but can still be performed to improve the balance. Applying the move to the hypergraph (bottom-right) improves the connectivity metric of the hypergraph by $\omega(e_1)$.

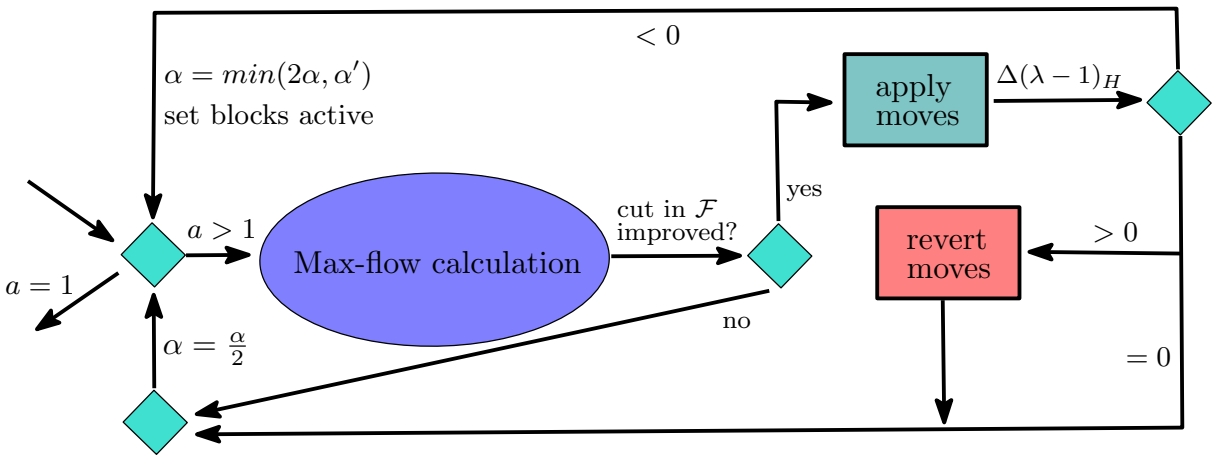

Figure 4.5.: Adapted workflow of an adaptive flow iteration without removing vertices. After the moves are applied to the hypergraph, we determine the real impact on the connectivity metric and update the parameters accordingly. If the moves had a negative impact, we revert them.

Our next approach is therefore to build the flow network with all hypernodes of the region and assume that the side effects will overall have a less negative impact than removing the vertices. By doing this we give up the cut property of the flow problems and have to handle the conflicts explicitly. The main problem is that we can no longer decide if we found an improvement as before, since parallel flow calculations can influence each other. Moves were only applied if the cut value was reduced. Also the update of the adaptive flow iteration variable α and if we set the involved blocks active in the next round of the active block scheduling depends on the fact that an improvement was found or not.

To solve this, we calculate the actual impact on the connectivity of the partition when applying the moves to the hypergraph. Moving a node v in our framework includes an atomic update to the Φ values of all incident nets $e \in I(v)$ and all blocks $V_i \in \Pi$. We can calculate the actual gain of a move by tracking if a move reduced a $\Phi(e, V_i)$ value of an adjacent hyperedge to zero or increased it to one. By adding up the gain values of all applied moves we obtain the impact of our flow calculation. As the incident nets are locked one at a time it is possible that this scheme attributes the contributions to a move executed in parallel by another thread. But this happens much less frequently than two flow problems influencing each other and the consequences of one inaccurate gain are acceptable. We therefore treat the value obtained with this method as the real impact of the moves. We then update α and set the blocks active according to the obtained value. It is possible that we have a negative impact on the connectivity. In this case we reverse our applied moves. If the impact is zero, we do not reverse the moves to minimize side effects with other calculations. In Figure 4.5 the new procedure of an adaptive flow iteration is illustrated.

4.2.4. Removing Synchronization-Steps

Our last optimization technique tackles a different problem of the algorithm that comes with the active block scheduling approach. Previously, we only executed a single round of the

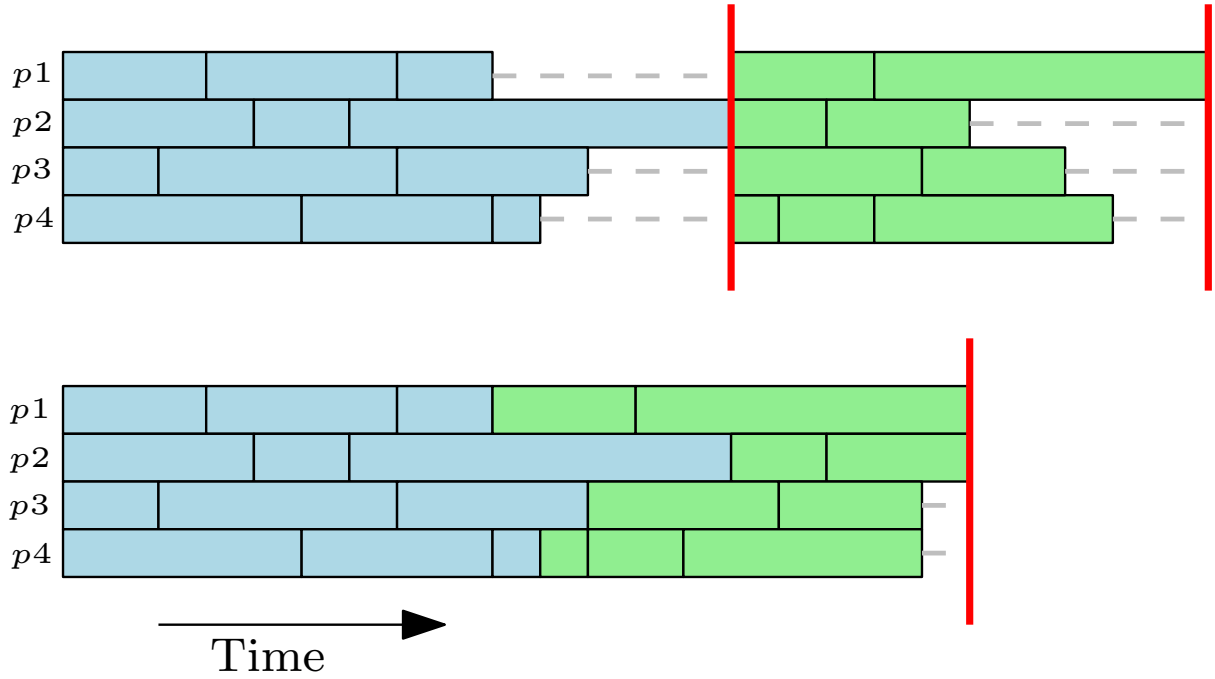

Figure 4.6.: Illustration of the negative effect that multiple synchronization steps can have on the running time.

active block scheduling in parallel. In each round we process each block-pair of the quotient graph exactly once. The end of each round worked like a synchronization step. In Figure 4.6 the negative effects that multiple synchronization steps can have on the running-time of an algorithm are illustrated. The blue and green rectangles illustrate work-packages (e.g. the flow calculation on a block pair) that are executed in parallel. The red lines represent synchronization steps. In both cases the same amount of work is done, but one synchronization step less significantly reduces the running-time.

Often synchronization steps are inevitable, as some things can only be done sequentially. In our case the only thing that is done sequentially between two rounds is finding all block-pairs that must be executed in the next round and select the ones that initially should be executed in parallel. The goal of the last optimization is to remove the synchronization steps between the rounds of the active block scheduling by executing all calculation in as long as there are active blocks. We call this technique *No-Sync*. An important note is that all flow calculations are still assigned to a specific round of the active block scheduling to keep the benefits of the technique.

A problem coming with this approach is that we do not know all block-pairs, that need to be executed, at the beginning of the parallel execution. We define the set R that contains all block-pairs, that are currently executed, and X that contains all block-pairs that still need to be executed. Which block-pairs need to be executed in a round depends on the results of the last round. Therefore block-pairs need to be added dynamically to X. We perform a flow-based refinement on two blocks if at least one of both blocks is active in this round. Previously we stored the status of all blocks in a vector containing an entry for each block. After determining which edges need to be executed in a round, we reseted the vector, setting all blocks inactive. As we now want to execute multiple rounds in parallel, we can never safely reset the vector.

Therefore we store the information for active blocks in a two-dimensional data-structure a that is growing with each round. $a(x, V_i) = true$ indicating that V_i is active in round x.

When a block-pair (V_i, V_j) is set active in round x we determine if we have to add new block-pairs to X. The only dependency we have between flow calculations is that we can not schedule multiple calculations on the same block-pair. We define $N(V_i, x, j) \cup N(V_j, x, i)$ as the set of block-pairs we add to X when we set (V_i, V_j) active in round x with:

$$N(V_i, x, j) = \bigcup_{z=1}^{i-1} \{(V_z, V_i) : z \neq j, (V_z, V_i) \notin X \cup R\} \cup \bigcup_{z=i+1}^{k} \{(V_i, V_z) : z \neq j, (V_i, V_z) \notin X \cup R\}$$

To make sure (V_i, V_j) is also scheduled in the next round we add it to X after the flow calculation is done. With this technique we nearly realize the sequential active block scheduling. All block-pairs that would be scheduled in the next round caused by the activation of our blocks are either added to X, are already in X or are currently running ($\in R$). Meaning they will be executed in the future or are already running. If the block-pair is currently running in a round $y = x - 1$ the pair will be added to X again, even if no improvement was found on it. The only scenario where an adjacent block-pair is not refined after finding an improvement, is when the block-pair is already running in a round $y \neq x - 1$ and does not find an improvement. We accept this modification to the algorithm as it most likely has very little effect on the partition-quality.

Note that the quotient graph can dynamically change, since moves can introduce new cut nets which induce new edges between two blocks of the partition in the quotient graph. To deal with this we consider every block-pair as an edge of the quotient graph and use one of the speedup heuristics by [24], which aborts a flow calculation immediately if the cut between two blocks is smaller than a fixed threshold.

As setting the blocks active is done in parallel we need to make sure that only the first flow calculation, that is setting a block active, schedules the new block-pairs. We ensure this by using atomic variables in a and executing a compare-and-swap operation to determine if the block was already active. When two flow calculations would add new block-pairs at the same time it can happen that a block-pair is added to X twice. To prevent this we use an exclusive lock while adding the block-pairs.

As we add block-pairs to X dynamically it is possible that at some point we have less parallel flow calculations available than threads, but add more block-pairs in the future. Instead of scheduling exactly one new block-pair after a flow calculation terminates, we therefore try to schedule till we have as much parallel executions as threads available.

4.2.5. Parallel Most Balanced Minimum Cut

Another part of the algorithm we did not cover so far is the most balanced minimum cut heuristic. As [24] and [39] we make use of the fact that with one maximum (s, t)-flow calculation it is possible to obtain all minimum (s, t)-cuts. After calculating a maximum flow we iterate over a number of possible minimum (s, t)-cuts and chose the one with the best balance. Heuer and

Schlag [24] uses as the global imbalance of the partition $imbalance(\Pi) = max(\frac{c(V_i)}{\lceil \frac{c(V)}{k} \rceil} - 1), V_i \in \Pi$. If the best partition induced by the max-flow calculation is feasible ($imbalance(\Pi) \leq \epsilon$) and either improves the metric or the balance of the partition, the moves are applied to the hypergraph. In parallel the global imbalance is additionally influenced by moves performed by other calculations. We therefore have to consider the impact of multiple flow calculations within the same block, before applying the (s, t)-cut to the hypergraph.

As a flow calculation on a block-pair (V_i, V_j) only moves vertices between the involved blocks, the only values that change and affect the imbalance are $c(V_i)$ and $c(V_j)$. In the matching scheduling approach we can simply consider the two involved blocks in isolation, as they are not influenced by other calculations. Therefore we define the *local imbalance* of a block-pair $imbalance(V_i, V_j) = max(\frac{c(V_i)}{\lceil \frac{c(V)}{k} \rceil} - 1, \frac{c(V_j)}{\lceil \frac{c(V)}{k} \rceil} - 1)$.

For the other two scheduling approaches, that allow more than one calculation on a block, additional work is required. The values of $c(V_i)$ and $c(V_j)$ can be influenced by other calculations in parallel. As we have an ownership of hypernodes by a flow calculation, the calculation also owns a fraction of the part-weight $c(V_i)$ of a block. For a flow calculation on a block-pair (V_i, V_j) with the region $B_{i,j}$ we define the *acquired part weights* $q_i(B_{i,j}) := c(V_i \cap B_{i,j})$ and $q_j(B_{i,j}) := c(V_j \cap B_{i,j})$. This is the fraction of the part weight that is controlled by the flow calculation on (V_i, V_j) and can not be influenced by other calculations. To determine the local balance between two blocks we additionally need the values of the *not acquired part weights* $n_i(B_{i,j}) := c(V_i \setminus B_{i,j})$ and $n_j(B_{i,j}) := c(V_j \setminus B_{i,j})$. This fraction of the block-weight can be owned and changed by other calculations in parallel.

To be able to obtain the acquired and not acquired part weights in parallel we introduce a two dimensional data structure W. For every block i, $W(i, i)$ contains the fraction of the block weight that is currently not owned by any calculation. When we build a region $B_{i,j}$, we update the values $W(i, j) = q_i(B_{i,j})$ and $W(i, i) = W(i, i) - q_i(B_{i,j})$ for both blocks. Therefore the acquired part weights of a block i are stored in $W(i, j) = q_i(B_{i,j}), j \in (1, \ldots, k)$. The not acquired part weight for a block-pair (V_i, V_j) can be obtained as follows:

$$n_i(B_{i,j}) = \sum_{\substack{x \in (1,..,k) \\ x \neq j}} W(i, x)$$

$$n_j(B_{i,j}) = \sum_{\substack{x \in (1,..,k) \\ x \neq i}} W(j, x)$$

To protect the parallel access to W, we use a read-write lock for each block. We obtain the not acquired part weights before we execute the most balanced minimum cut heuristic. If a feasible partition is found and the moves are applied to the hypergraph, the calculation releases the acquired weights and updates the values in W accordingly. To make sure no other calculation influenced the balance and the imbalance calculations done during the most balanced minimum cut are still correct, we can check if the not acquired part weight still equals our earlier obtained value. If the not acquired part weight changed during the most balanced minimum cut step, we could restart the procedure. With that technique we are able to guarantee a balanced partition,

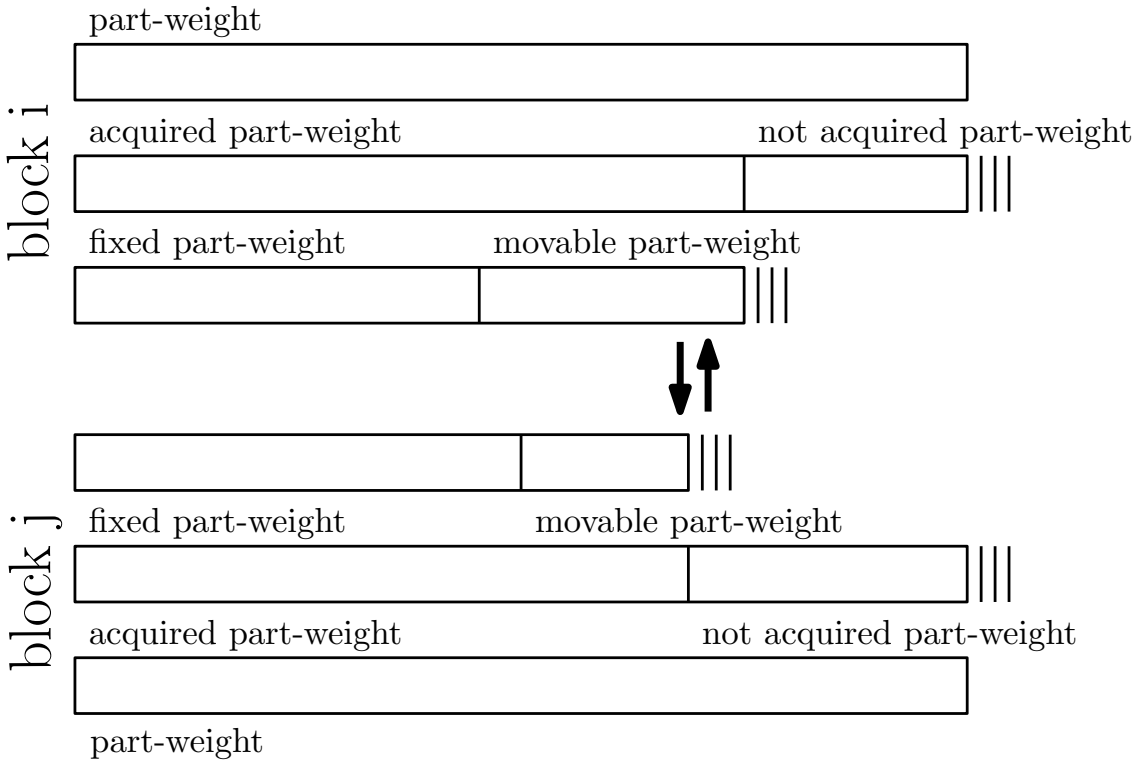

Figure 4.7.: Fractions of the part-weight used by the most balanced minimum cut calculations. The movable part-weights and the not acquired part-weights can be influenced by other calculations executed in parallel.

but the possible repetition of the most balanced minimum cut step could have negative impact on the running time. As our framework includes an rebalancing step after the refinement, we omit the check after the most balanced minimum cut and accept the possibility of a temporarily imbalanced partition.

Additionally we introduce another optimization to the most balanced minimum cut process. We can further split the acquired fraction of the part weight into *fixed part weight* and *movable part weight*. Fixed meaning here the part weight that is corresponding to vertices that will remain in their block in every minimum (s,t)-cut. We can determine these by checking which nodes are reachable from the source resp. the sink in the residual graph after the maximum flow calculation. For each vertex ,that is part of the movable part weight, there exists a minimum (s,t)-cut where the vertex has to change its block. The different fractions of the part weights are illustrated in Figure 4.7. After determining all fixed hypernodes in the residual graph, we can check if one of the fixed part weights added to the not acquired part weight already exceeds $(1 + \epsilon)\lceil\frac{c(V)}{k}\rceil$. If that is the case, we can abort the most balanced minimum cut, because no minimum (s, t)-cut induces a feasible partition.

As we can never safely compute the global balance of our partition in parallel, determining if applying a minimum (s, t)-cut to the hypergraph improves the balance is difficult. We therefore only calculate a most balanced minimum cut if the (s, t)-cut induces a strictly positive improvement to the cut value in the flow network and not to only improve the balance.

5. Experiments

In this Chapter we will evaluate the performance of the presented parallel max-flow-min-cut refinement technique. First, we will compare the different scheduling techniques and find an optimal configuration for our parallel flow-based refinement. Then we will combine the flow-based refinement with other refinement algorithms implemented in Mt-KaHyPar and compare the different combinations in running time and quality. We will further inspect the scaling of our algorithm for different numbers of threads. Finally we will compare our optimal setup with other state of the art hypergraph partitioning systems.

5.1. Instances

To evaluate our algorithms, we use multiple benchmark sets. Our full benchmark set (A) is the set from Heuer and Schlag [25]. It contains 488 hypergraphs from different application areas. The instances are derived from the ISPD98 VLSI Circuit Benchmark Suite [2], the DAC 2012 Routability-Driven Placement Contest [43], the SuiteSparse Matrix Collection [10] and the international SAT Competition 2014 [4]. For parameter tuning experiments we use a representative subset of the full benchmark set containing 100 instances (set B). For our scaling experiments we use a random subset (set C), of 30 hypergraphs, from a benchmark set originally containing 90 large hypergraphs.[1]

5.2. System and Methodology

For our experiments we use a machine consisting of two *Intel(R) Xeon(R) E5-2683* clocked at 2.10 GHz with 16 cores each and 504 GB main memory. In the following we refer to this machine as $P1$. For the comparison with other sequential hypergraph partitioning systems on the full benchmark set, we used nodes of a cluster with *Intel Xeon Gold 6230* (2 Sockets with 20 cores each) clocked at 2.1 GHz with 96 GB main memory ($P2$). For the comparison with the parallel hypergraph partitioner Zoltan we used an *AMD EPYC ROME 7702P* with one socket and 64 cores clocked at 2.0-3.5 GHz and 1025GB main memory ($P3$). The executed code is written in `C++17` and compiled using `g++9.2` with the flags `-O3 -mtune=native -march=native`. We use the Intel Thread Building Blocks library as parallelization library [44].

To indicate a parallel execution with multiple threads, we add a suffix with the number of threads to the algorithm description in the plots. For sequential partitioners we omit the suffix. For each hypergraph we execute multiple runs with different k. For each k we additionally execute multiple seeds. To aggregate the running time and the solution quality for a hypergraph and a specific k, we use the arithmetic mean over all seeds. To further aggregate over multiple

[1] The full set can be found under `http://algo2.iti.kit.edu/heuer/alenex21/` (Set B).

instances (hypergraph and k), we use the geometric mean for the running time and the harmonic mean for speedups. In the comparison with other partitioners, we used a time limit of 8 hours for a single execution. We use ⏲ to indicate that an instance exceeded the time limit and ✗ when all seeds produced an imbalanced partition.

To compare the solution quality of different algorithms, the performance profiles introduced by [12] are used. For a set of algorithms $\mathcal{A}$ and a set of instances $\mathcal{I}$, $q_A(I)$ describes the quality of an algorithm $A \in \mathcal{A}$ on an instance $I \in \mathcal{I}$. On the y-axis we plot for each algorithm the fraction of instances, that is better than the best solution times τ, where τ is on the x-axis: $q_A(I) \leq \tau \cdot min_{A' \in \mathcal{A}} q_{A'}(I)$. For $\tau = 1$ the y-value of an algorithm therefore indicates for how many percent of the instances the algorithm produced the best solution.

For all executions our algorithm is configured as proposed by [23]. The adaptive flow iteration variable α is set to 16 and as a maximum flow algorithm IBFS [19] is used. We always use the most balanced minimum cut heuristic and all speedup heuristics proposed by [23]. As objective function we use the connectivity metric and an imbalance parameter of $\epsilon = 0.03$.

5.3. Comparison of the different Scheduling Approaches

In our first experiment we compare the different scheduling techniques and optimizations, that we presented in Section 4. We compare the matching scheduling (match) and the all-blocks scheduling (all-blocks). The all-blocks scheduling is configured with the optimization described in Section 4.2.3 (Opt) and the no-sync approach from section 4.2.4 (S). A + in the algorithm description is indicating that the corresponding optimization was used. E.g. `all-blocks(+Opt,-S)` refers to our all blocks scheduler with the optimization from Section 4.2.3 and without the no-sync approach. We evaluate the performance of our scheduling algorithm by executing them on our subset (B) using machine $P1$ and 16 threads. We executed 5 seeds for every instance and every $k \in \{4, 8, 16, 32, 64\}$.

Figure 5.1 shows the different running times of the algorithms per k as well as their geometric mean. As expected, is the matching scheduling slower than the other approaches, because due to the matching restriction, we potentially schedule less block-pairs in parallel. The difference is more significant for smaller k. As there are overall less block-pairs to execute in parallel, the matching restriction has a larger impact. For larger k the running time of the matching scheduling is similar to the other algorithms. For larger k, we fully utilize all cores even with the matching scheduling and the advantages of scheduling multiple calculations on one block become less relevant.

When we compare the all-blocks scheduling with and without the no-sync optimization, we see that removing the synchronization steps between the rounds of the active bock scheduling, overall improved the running time. The impact is smaller than removing the matching restriction, but it is also more significant for smaller k. The more block-pairs we schedule in parallel, the less impact the synchronization steps seem to have on the overall running time.

When we examine the impact of removing vertices in terms of running time, the results differ for increasing k. For $k = 4$ removing the vertices is slower. For $k = 8$ and $k = 16$ we observe the opposite and for larger k the running times become nearly equal.

Based on the running times, our best scheduling configuration is the all-blocks scheduling with or without removing vertices. To determine the best configuration for our parallel

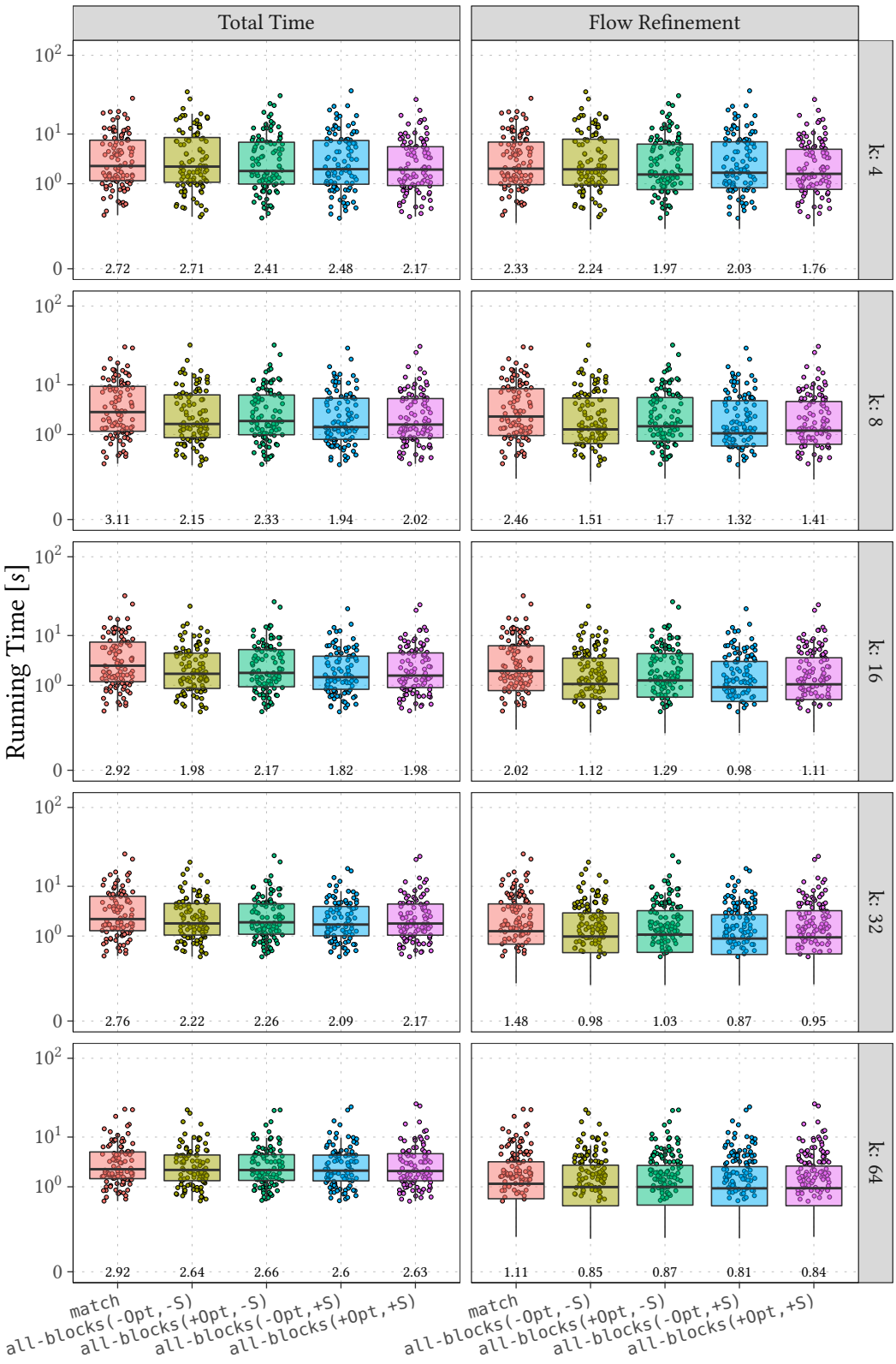

Figure 5.1.: Running times per k and their geometric mean values for the different scheduling approaches and optimization techniques presented in Chapter 4.

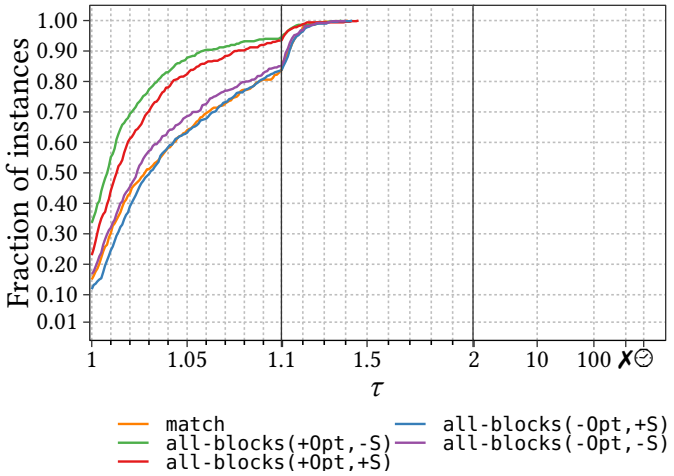

Figure 5.2.: Comparison of the partition quality for the different scheduling approaches and optimization techniques presented in Chapter 4.

framework, we will further inspect the influence of the different approaches on the partition quality. In Figure 5.2 the quality of the different approaches are illustrated. As we see, not removing the vertices, produces the best solution quality. Test experiments showed, that if we are not removing vertices, we actually improve the solution quality by using multiple threads. The matching scheduling remains very stable in terms of solution quality, when executed with multiple threads.

The direct comparison of the all-blocks scheduling with both optimizations activated to the other approaches is shown in Figure A.1. Using the all-blocks scheduling, not removing the vertices, produces the best solution quality on 70.2% of the instances when compared directly to removing the vertices. Not using the no-sync approach shows a slight improvement to the solution quality in the all-blocks scheduling. Further test experiments showed that this difference diminishes when we combine our flow-based refinement with other refinement algorithms. As we will show in the next section, combining the flow-based refinement with other refinement algorithms is always preferable. Due to the slight advantage in running time, we therefore chose the all-blocks scheduling with both optimization techniques active as our best configuration.

5.4. Refinement Configuration

As we integrate our approach in the Mt-KaHyPar framework, we additionally need to examine how our flow-based refinement works together with other refinement algorithms. In Mt-KaHyPar two additional refinement algorithms, namely a variant of the classical FM algorithm [15] and a greedy refinement technique based on label propagation (LP) [1], are implemented. In the following we refer to our flow-based refinement as Maximum Flow (MF). To determine the best configuration, we tested all possible combinations of the three available refinement algorithms. As it is preferable that an improvement is found by a faster algorithm, we always

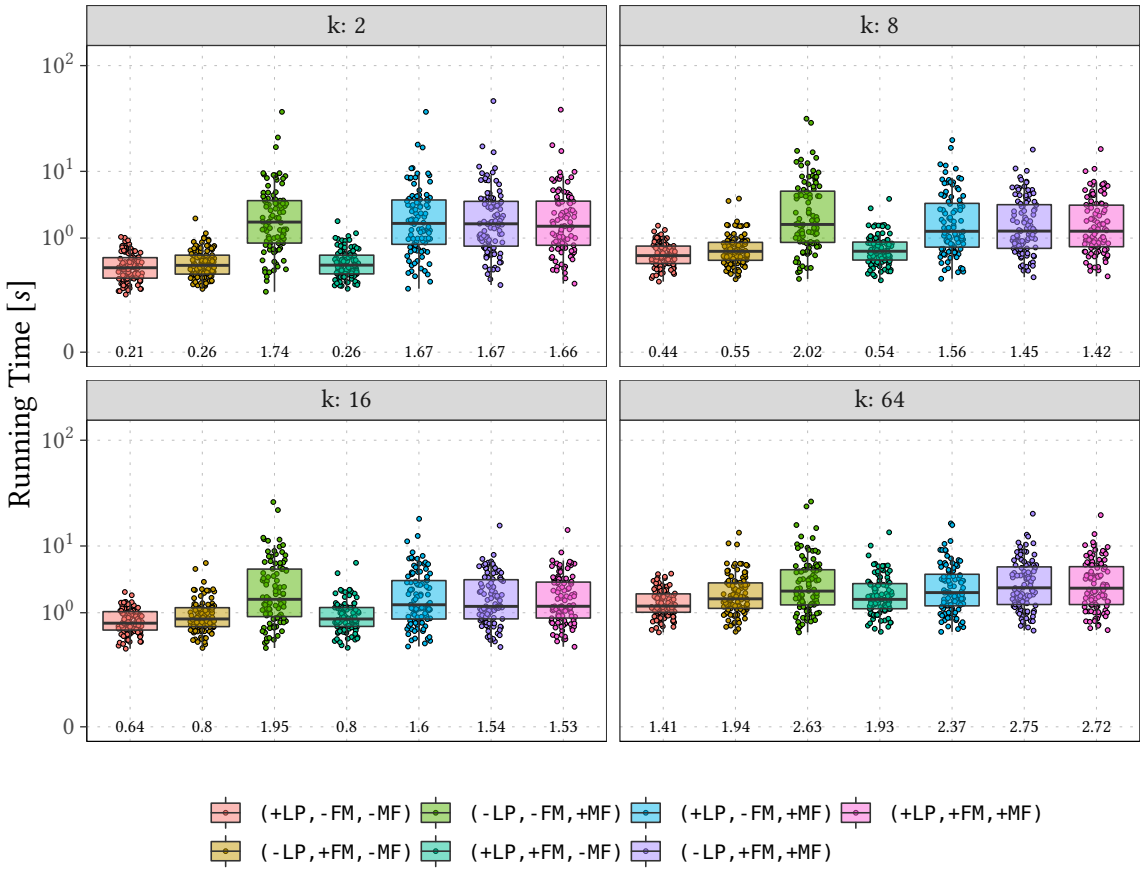

Figure 5.3.: Effect on the running time, when combining the different refinement algorithms implemented in Mt-KaHyPar with our flow-based approach. Illustrates the running times per k and their geometric mean for the different combinations.

execute the algorithms in the following order: (1) LP (2) FM (3) MF. For the experiment, we again used machine $P1$, the subset B and executed 5 seeds for every instance with $k \in \{2, 8, 16, 64\}$.

We first consider the running times of the different combinations. Figure 5.3 shows the running times and the geometric mean per k for all combinations of the different refinement algorithms. The first thing one notices, is that all combinations including the flow-based refinement have a longer running time than the ones without it. For smaller k the difference is more significant. Another observation is that combining the flow-based refinement with other algorithms reduces the running time in most cases. The best combination in terms of running time with the flow-based refinement activated is using all three algorithms.

To determine the best combination of refinement techniques, we will further inspect the impact on the partition quality. In Figure 5.4 the solution quality of the different combinations is shown. The two best configurations in terms of quality are the ones including the FM-algorithm and the flow-based refinement. When the FM-algorithm is active, the label propagation algorithm has no impact on the solution quality. As they are both move based algorithms, the

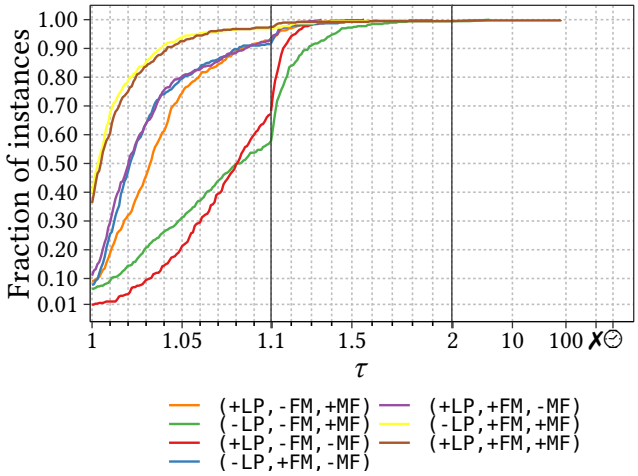

Figure 5.4.: Impact on the partition quality by the different combinations of refinement algorithms. Direct comparisons can be found in the Appendix in Figure A.2

greedy label propagation approach will very rarely find improvements, that the FM-algorithm can not find. Another observation is that the flow-based refinement as a single refinement algorithm is comparable to the greedy label propagation approach. The direct comparison of some of the combinations is added in the Appendix in Figure A.2. Adding a move based approach to the flow-based refinement significantly improves the partition quality. Combining the flow-based refinement with both move based approaches improves the solution quality on 91% of the instances. Adding the flow-based refinement to both move based approaches improves the best partition on 75.3% of the instances.

The conclusion of these experiments is that the flow-based refinement should never be used as a single refinement algorithm, as combining it with a move based approach significantly improves the partition quality and mostly even the running time. Even though the label propagation has very little impact in terms of solution quality we still activate it in our best configuration due to a slight improvement in running time. We therefore choose (+LP,+FM,+MF) as our best configuration. In the following we refer to this algorithm as Mt-KaHyPar-MF.

5.5. Scalability

To determine the scalability of our algorithm, we executed it on our large hypergraph set C using machine $P1$. For every hypergraph and $k \in \{2, 8, 16, 64, 128\}$ we executed three seeds using 1, 4, 16 and 32 threads. As our machine consists of two CPU's with 16 cores each, for 32 threads, we additionally have NUMA-affects that negatively influence our running time. In Figure 5.5 and Table 5.1 the speedup per k for the total running time and the flow refinement are illustrated. We represent the speedup of each instance as a point and the harmonic mean speedup over all instances with with a single-threaded running time $\geq x$ seconds with a line.

As expected, we have no speedup for $k = 2$ in the flow refinement. Only considering the flow refinement time for $k = 2$, more threads actually result in a worse running time. The more we increase k, the better the speedup of the flow refinement. Using 32 threads compared to

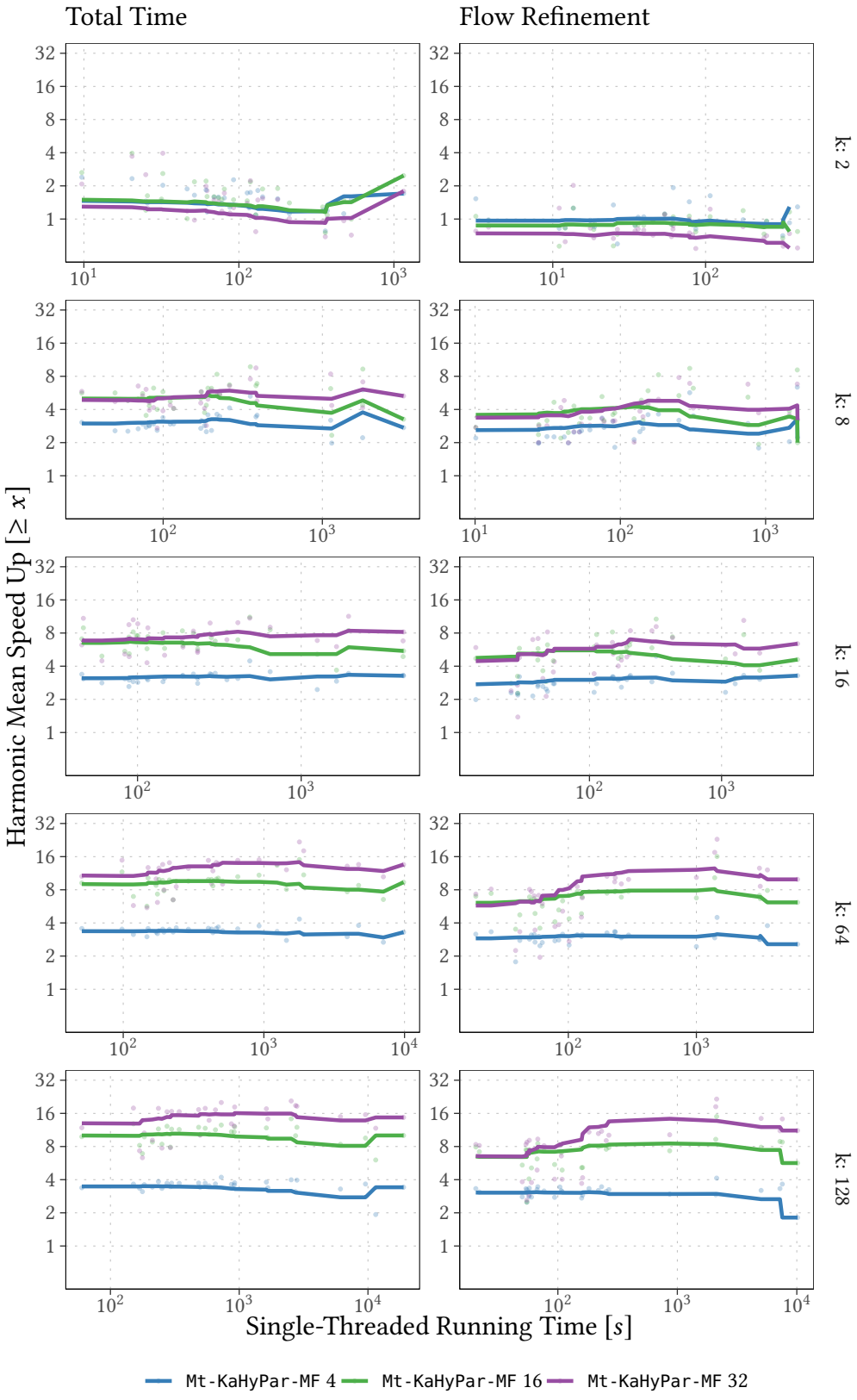

Figure 5.5.: Arithmetic mean speedup per instance and the cumulative harmonic mean speedup (lines) of the total running time and the flow-based refinement per k.

threads \ k	2	8	16	64	128
4	0.98	2.60	2.74	2.91	3.06
16	0.88	3.58	4.75	6.12	6.49
32	0.74	3.37	4.50	5.79	6.57

Table 5.1.: Harmonic mean speedup per k of the flow-based refinement using 4, 16 and 32 threads.

16, barely increases the speed up for most of the instances. Only for large k and a high single threaded running time, a significant improvement compared to 16 threads is observable. The speedup using 16 threads is mostly stable for different single threaded running times.

Further experiments using the Intel Vtune Profiler showed that the scaling of our algorithm is mainly limited by memory bandwidth. Using more threads, the percentage of memory bound instructions significantly increases. This prevents a better scalability of our algorithm.

5.6. Comparison with other Hypergraph Partitioner

We now compare our algorithm with other state of the art hypergraph partitioning systems. We compare us to PaToH [7] using three different configurations: PaToH-D (default), PaToH-S (speed) and PaToH-Q (quality). We also use hMetis-R [28] and two variants of KaHyPar. KaHypar-CA being a n-level partitioner using similar move based approaches as Mt-KaHyPar and KaHyPar-HFC, that additionally uses the flow-based refinement by [21]. We also added the default version of Mt-KaHypar without the flow-based refinement to the comparison. With all partitioners, we executed the full benchmark set A for $k \in \{2, 4, 8, 16, 32, 64, 128\}$. For each instance, we executed 10 seeds on machine $P2$. We executed Mt-KaHyPar-MF using 1, 10 and 20 threads and Mt-KaHyPar with 10 threads.

Figure 5.6 shows the running times for each partitioner. Considering the running times, we categorize the partitioners in two different categories. Algorithms in the first group are KaHyPar-Ca, KaHyPar-HFC and hMetis-r. These partitioning systems invest a substantial running time and aim for a high-quality solution. The second category, consisting of the three variants of Patoh, aims for good quality-time trade-off. We can assign our algorithm to the second group. We achieve a better mean running time compared to Patoh-Q with 10 threads. Incorporating the flow-based refinement slows down Mt-KaHyPar by a factor of two. The detailed running times per k are added in the Appendix (Figure A.3). As expected, does our algorithm slow down Mt-KaHyPar less for large k.

The partition quality of our algorithm does not decrease significantly when using multiple threads (Figure A.4). As it achieves a better quality than PaThoH-Q while being faster by a factor of two and in general we can assume, that a system with 10 threads is available, we will use Mt-KaHyPar-MF 10 to compare our algorithm with the other hypergraph paritioners in terms of solution quality.

To better classify our algorithm, we will compare it with our defined categories separately. In Figure 5.7 we compare the partitioner in both categories in terms of solution quality. On the

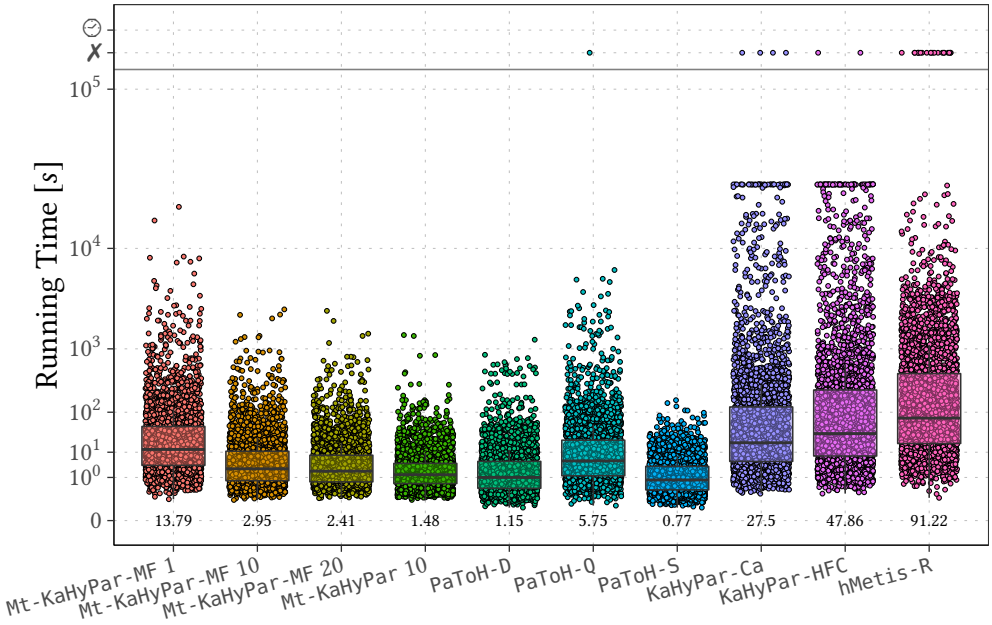

Figure 5.6.: Comparison of the running time of Mt-KaHyPar-MF with other state of the art hypergraph partitioners on the full benchmark set A using machine *P2*.

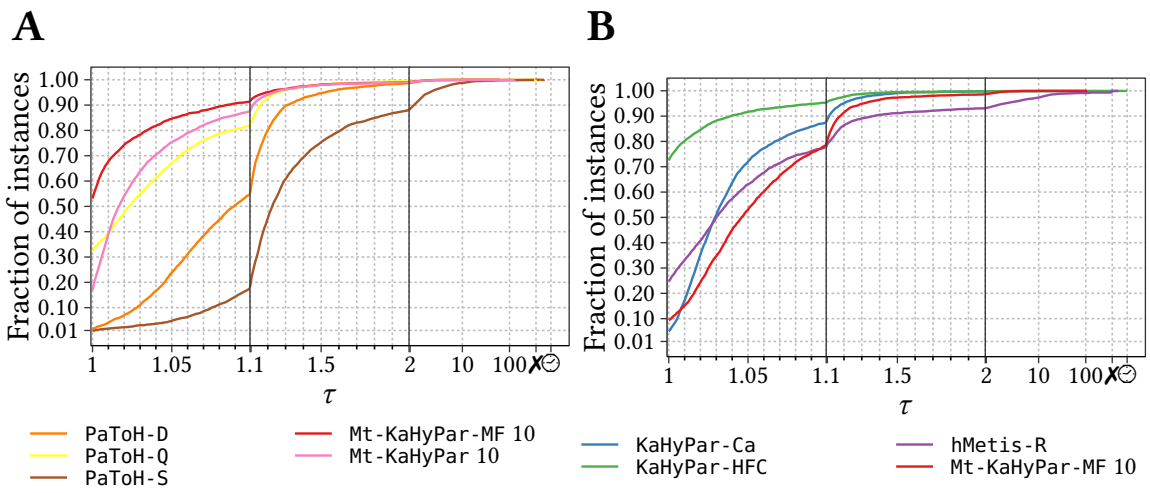

Figure 5.7.: Performance Profiles comparing the solution quality of Mt-KaHyPar-MF with other hypergraph partitioners.

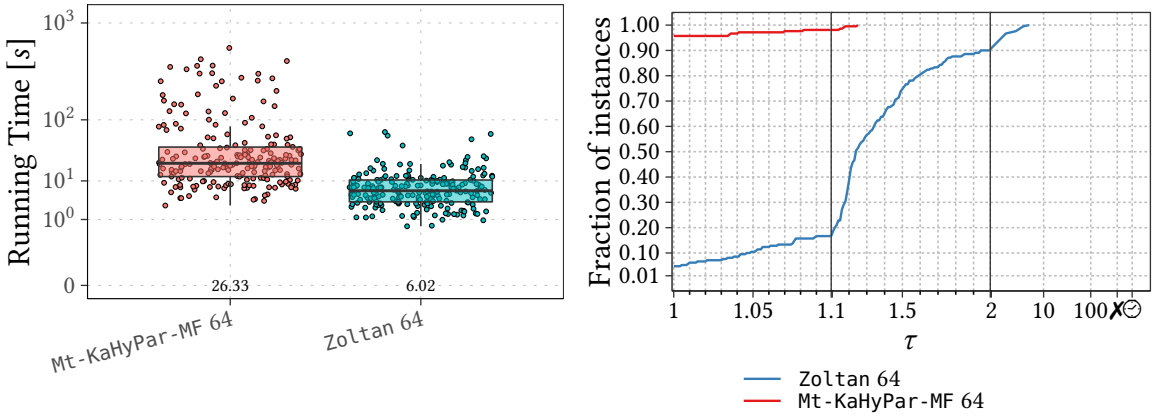

Figure 5.8.: Direct comparison in terms of running time and solution quality with Zoltan using machine $P3$ and 64 threads on our large hypergraph set C.

left side (A) we compare our algorithm with the partitioners aiming for a good quality-time trade-off and on the right side (B) we compare it to the partitioners with a significantly higher running time. In A, we can see that we outperform all partitioners with a comparable or better running time than ours. Our algorithm produced the best solution quality for 53.1% of the instances. The only version of PaToH being competitive is Patoh-Q, which we also outperform in terms of running time. When compared directly to the other algorithms, MT-KaHyPar-MF produced the best partition for 96.6%(PaToH-S), 86.6%(PaToH-D), 74.6% (Mt-KaHyPar) and 67.2% (PaToH-Q) of the instances. The plots showing the direct comparison can be found in the Appendix in Figure A.5.

In B we can see that our solution quality is comparable to hMetis-R and KaHyPar-CA, although they have a considerably larger running time than our algorithm. When compared directly to hMetis-R we produced the best solution quality for 47.9% ot the instances (Figure A.5). The clearly best algorithm in terms of solution quality is KaHyPar-HFC.

Finally we compare our algorithm with the parallel hypergraph partitioner Zoltan on machine $P3$ and our large hypergraph set C. We executed 5 seeds for every instance and $k \in \{2, 4, 8, 16, 32, 64, 128\}$ using 64 threads. Figure 5.8 shows the direct comparison in running time and solution quality. While Zoltan is faster by a factor of 4.37 in the mean running time, we outperform it in terms of solution quality on 95.7% of the instances. The detailed comparison of the running times per k can be found in Figure A.6 in the Appendix.

6. Conclusion

In this thesis we present a parallel flow-based refinement algorithm for shared-memory multilevel hypergraph partitioning. We schedule multiple pairwise flow calculations on adjacent blocks of the quotient graph in parallel, to improve the running time and scalability of the algorithm. The flow-based refinement by Heuer and Schlag [24] was used as a basis for our parallel framework. For large k, we achieve a significant improvement of the running time with multiple threads. For $k \geq 16$ and 16 threads, we achieve a harmonic mean speedup of 5.68 for the flow based refinement. The main weakness of our approach is that we only improve the running time for $k > 2$, as for a bipartition only one pairwise refinement is executed.

We work out the theoretical basis and present multiple techniques to schedule pairwise flow-based refinements in parallel. We start by only scheduling disjoint block-pairs, that form a matching in the quotient graph. To further improve the level of parallelism, we allow multiple flow-based refinements on one block. We introduce a technique, which still guarantees the correctness of the flow-based calculations in parallel, but has a negative impact on the partition quality. To improve the partition quality, we loosen restrictions that were necessary for the correctness of our parallel flow calculations and show how to ensure correctness in a postprocessing step.

We integrate our algorithm in the shared-memory multilevel hypergraph partitioning framework Mt-KaHyPar, which is currently under development. We combine the flow-based refinement with other implemented refinement algorithms and call our approach Mt-KaHyPar-MF. Incorporating the flow-based refinement significantly improves the solution quality, while it slows down the running time by a factor of two. As a standalone refinement algorithm the flow-based refinement does not perform well in terms of running time and partition quality when compared to a move-based algorithm. Therefore the flow-based refinement should always be used in combination with a move-based approach. For small k, our flow-based refinement dominates the running time. For larger k the flow-based refinement has still a non neglible impact on the running time, but the share on the total running time becomes less significant.

Finally we compared our approach with other state of the art hypergraph partitioning systems. We achieve a better solution quality on 67,2% of the instances, when compared to the quality preset of PaToH, while having a better mean running time by a factor 1.95 by using 10 threads. Compared to hMetis-R, we achieve a comparable solution quality, while being an order of magnitude faster. Directly compared to the default version of Mt-KaHyPar, we achieve a better solution quality on 74.6% of the instances by including our flow-based refinement algorithm, while slowing down the geometric mean running time by a factor of 2. The only partitioners that outperform us in terms of solution quality are the two versions of KaHyPar. Especially KaHyPar-HFC, using the flow-based refinement by [21], significantly improves the solution quality compared to our algorithm, but is slower than our algorithm (executed with 10 threads) by a factor of 16. Our approach therefore offers a good trade off between solution quality and running time, compared to other state of the art hypergraph paritioners.

6.1. Future Work

As we only execute multiple flow calculations on block-pairs in parallel, our algorithm does not scale for small k. To further improve the scaling a way to speed up a single flow calculation on a block-pair using parallelism should be developed. One way to achieve this could be to further split the flow problems, while trying to preserve the solution quality.

As shown by the comparison with other hypergraph partitioning systems, the KaHyPar-HFC algorithm clearly outperforms our approach in terms of partition quality. As the algorithm also uses the active block scheduling and executes multiple flow calculations on block-pairs, some of our techniques could be used to parallelize the algorithm and integrate it in Mt-KaHyPar. To execute multiple flow calculations on one block some adjustments are necessary, as the flow problems in [21] are growing dynamically.

Additionally, constructing the flow network is responsible for a large amount of our running time. This causes our algorithm to be memory bound and prevents a better scaling. As the HFC-algorithm works directly on the hypergraph a further improvement in scalability should be achievable.

Bibliography

[1] Y. Akhremtsev, P. Sanders, and C. Schulz. "High-Quality Shared-Memory Graph Partitioning". In: *European Conference on Parallel Processing (Euro-Par)*. Springer-Verlag, Aug. 2017, pp. 659–671.

[2] C. J. Alpert. "The ISPD98 Circuit Benchmark Suite". In: *Proceedings of the 1998 International Symposium on Physical Design* (1998), pp. 80–85.

[3] S. T. Barnard and H. D. Simon. "A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems". In: *Concurrency: Practice and Experience* 6.2 (1994), pp. 101–117.

[4] A. Belov et al., eds. *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Finland: University of Helsinki, 2014.

[5] Y. Boykov and V. Kolmogorov. "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.9 (2004), pp. 1124–1137.

[6] Ü. V. Catalyurek and C. Aykanat. "Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication". In: *IEEE Transactions on Parallel and Distributed Systems* 10 (1999), pp. 673–693.

[7] Ü. V. Catalyurek and Cevdet Ph.D. "PaToH (Partitioning Tool for Hypergraphs)". In: Jan. 2011, pp. 1479–1487.

[8] Ü. V. Catalyurek et al. "Multithreaded Clustering for Multi-level Hypergraph Partitioning". In: *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012* (2012), pp. 848–859.

[9] C. Chevalier and F. Pellegrini. "PT-Scotch: A Tool for Efficient Parallel Graph Ordering". In: *Parallel Computing* 34.6–8 (July 2008), pp. 318–331.

[10] T. A. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software* 38.1 (2011), 1:1–1:25.

[11] K. D. Devine et al. "Parallel Hypergraph Partitioning for Scientific Computing". In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing* (2006), 10–pp.

[12] E. Dolan and J. Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Mathematical Programming* 91 (2001), pp. 201–213.

[13] W. E. Donath. "Logic Partitioning". In: *Physical Design Automation of VLSI Systems* (1988), pp. 65–86.

[14] J. Edmonds and R. M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *J. ACM* 19 (1972), pp. 248–264.

[15] C. M. Fiduccia and R. M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In: *Proceedings of the 19th Design Automation Conference*. DAC '82 (1982), pp. 175–181.

[16] D. R. Ford and D. R. Fulkerson. *Flows in Networks*. USA: Princeton University Press, 2010.

[17] L. R. Ford and D. R. Fulkerson. "Maximal Flow Through a Network". In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. DOI: 10.4153/CJM-1956-045-5.

[18] A. V. Goldberg and R. E. Tarjan. "A New Approach to the Maximum Flow Problem". In: *JOURNAL OF THE ACM* 35.4 (1988), pp. 921–940.

[19] A. V. Goldberg et al. "Faster and More Dynamic Maximum Flow by Incremental Breadth-First Search". In: *Algorithms - ESA 2015 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings* 9294 (Sept. 2015), pp. 619–630.

[20] A. Goldberg et al. "Maximum Flows by Incremental Breadth-First Search". In: *19th European Symposium on Algorithms (ESA 2011)* (Jan. 2011).

[21] L. Gottesbüren et al. "Advanced Flow-Based Multilevel Hypergraph Partitioning". In: *18th International Symposium on Experimental Algorithms (SEA 2020)* (2020), 11:1–11:15.

[22] T. Heuer. "Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning". Bachelor Thesis. Karlsruhe Institute of Technology, 2015.

[23] T. Heuer. "High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations". Masters Thesis. Karlsruhe Institute of Technology, 2018.

[24] T. Heuer, P. Sanders, and S. Schlag. "Network Flow-Based Refinement for Multilevel Hypergraph Partitioning". In: *Journal of Experimental Algorithmics* 24 (Sept. 2019), 2.3:1–2.3:36.

[25] T. Heuer and S. Schlag. "Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure". In: *16th International Symposium on Experimental Algorithms (SEA 2017)*. Leibniz International Proceedings in Informatics (LIPIcs) (2017), 21:1–21:19.

[26] M. Holtgrewe, P. Sanders, and C. Schulz. "Engineering a Scalable High Quality Graph Partitioner". In: *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium* (2010), pp. 1–12.

[27] T. C. J. Hu and K. E. Moerder, eds. *Multiterminal Flows in a Hyper-graph*. 1985.

[28] G. Karypis and V. Kumar. "Multilevel k-way Hypergraph Partitioning". In: *Proceedings - Design Automation Conference* 11 (Dec. 1998).

[29] G. Karypis and V. Kumar. "Parallel Multilevel k-Way Partitioning Scheme for Irregular Graphs". In: *Siam Review* 41.2 (1999), pp. 278–300.

[30] G. Karypis and D. Lasalle. "A Parallel Hill-Climbing Refinement Algorithm for Graph Partitioning". In: Proceedings of the International Conference on Parallel Processing (2016), pp. 236–241.

[31] G. Karypis et al. “Multilevel Hypergraph Partitioning: Applications in VLSI Domain”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 69–79.

[32] E. L. Lawler. “Cutsets and Partitions of Hypergraphs”. In: *Networks* 3 (1973), pp. 275–285.

[33] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.

[34] H. Liu and D. F. Wong. “Network Flow Based Multi-Way Partitioning with Area and Pin Constraints”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17 (1998), pp. 50–59.

[35] Z. Mann and P. Papp. “Formula Partitioning Revisited”. In: *Fifth Pragmatics of SAT workshop*. 2014, pp. 41–56.

[36] H. Meyerhenke, P. Sanders, and C. Schulz. “Parallel Graph Partitioning for Complex Networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (2017), pp. 2625–2638.

[37] D. Papa and I. Markov. “Hypergraph Partitioning and Clustering”. In: *Handbook of Approximation Algorithms and Metaheuristics* (2007).

[38] J. Picard and M. Queyranne. “On the Structure of All Minimum Cuts in a Network and Applications”. In: *Math. Program.* 22 (1982), p. 121.

[39] P. Sanders and C. Schulz. “Engineering Multilevel Graph Partitioning Algorithms”. In: *19th European Symposium on Algorithms (ESA)*. Springer, 2011, pp. 469–480.

[40] S. Schlag. “High-Quality Hypergraph Partitioning”. In: (2020).

[41] A. Trifunović and W. J. Knottenbelt. “Parallel Multilevel Algorithms for Hypergraph Partitioning”. In: *Journal of Parallel and Distributed Computing* 68.5 (2008), pp. 563–581.

[42] A. Trifunović and W. J. Knottenbelt. “Towards a Parallel Disk-Based Algorithm for Multilevel k-way Hypergraph Partitioning”. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* (Apr. 2004).

[43] N. Viswanathan et al. “The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite”. In: *DAC Design Automation Conference 2012* (2012), pp. 774–782.

[44] M. Voss, R. Asenjo, and J. Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. 1st. USA: Apress, 2019.

[45] D. B. West. *Introduction to Graph Theory*. 2nd ed. Prentice Hall, Sept. 2000.

A. Appendix

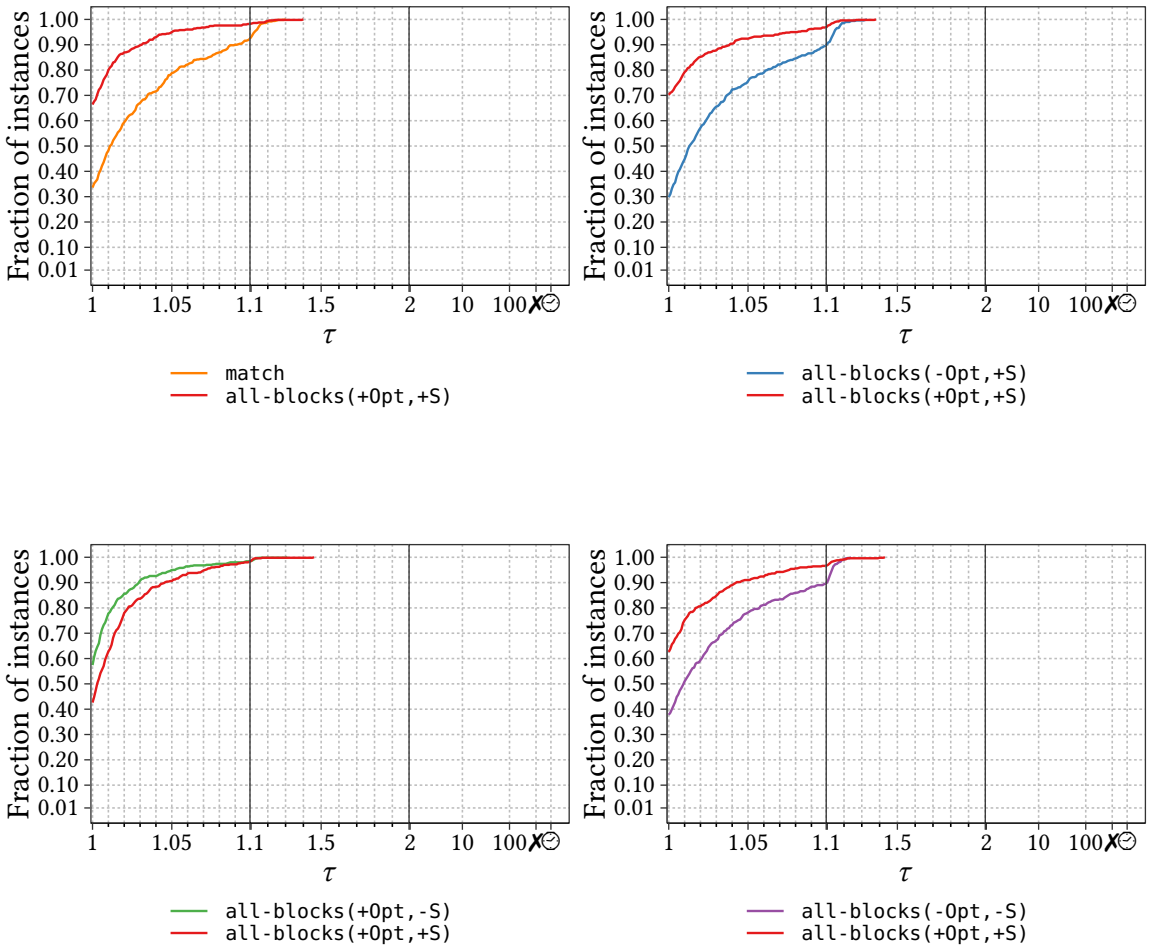

Figure A.1.: Performance Profiles comparing the solution quality of the all-blocks scheduling approach using both optimization techniques (`all-blocks(+Opt,+S)`) with other scheduling techniques.

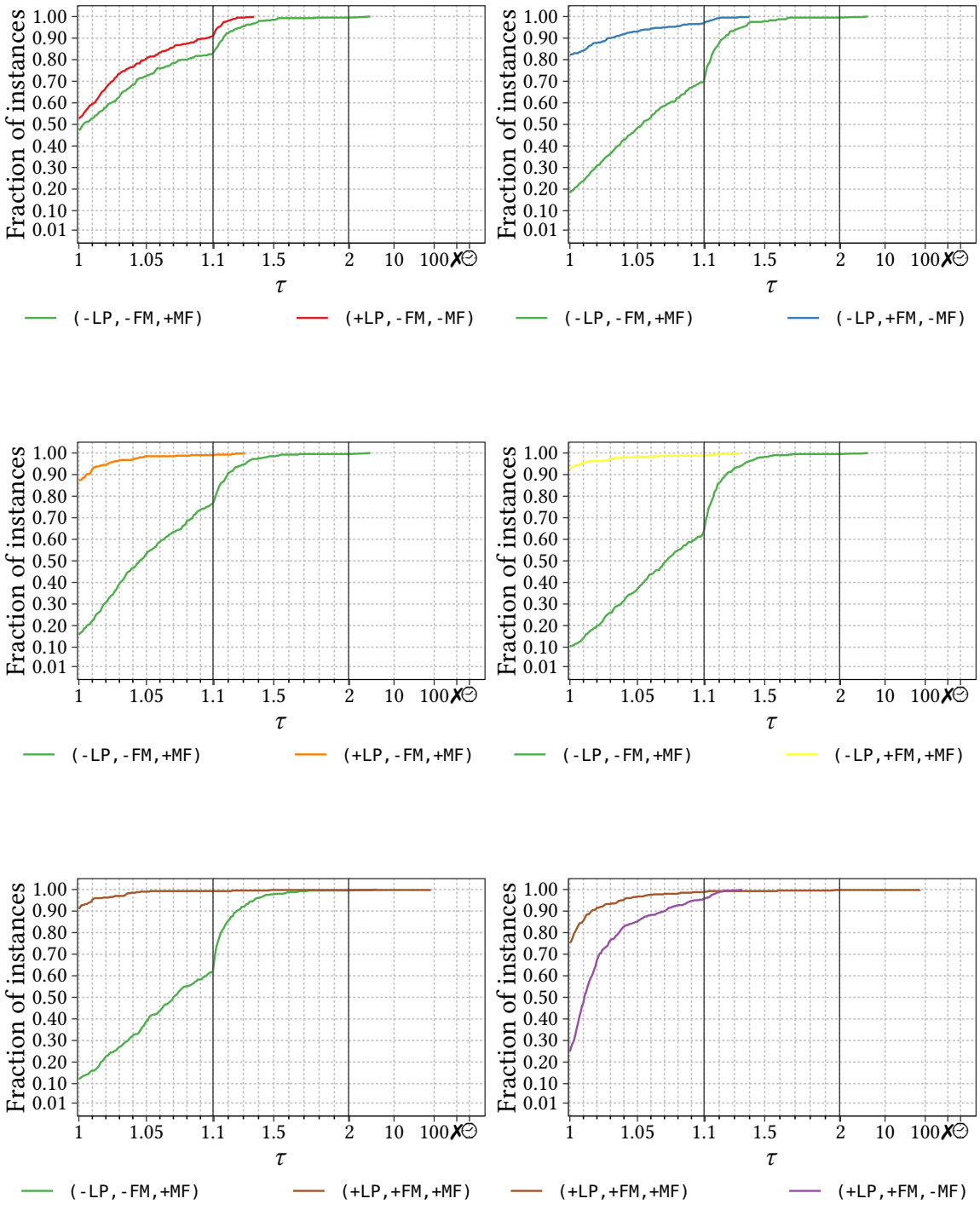

Figure A.2.: Direct comparison of the different combinations of refinement algorithms in terms of solution quality.

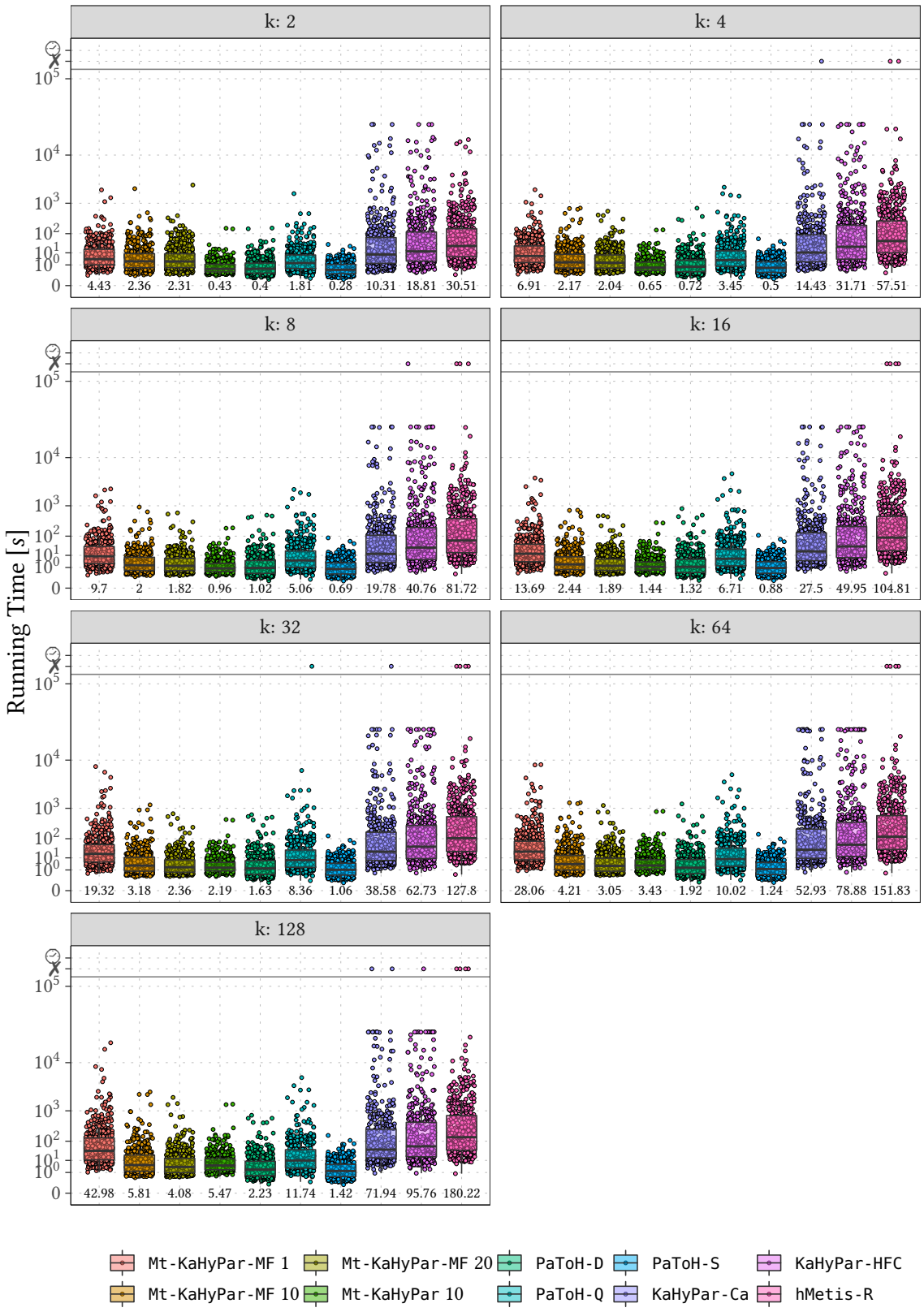

Figure A.3.: Absolute running times per k and their geometric mean for the different hypergraph partitioner. The values were obtained by executing our full benchmark set A on machine $P2$ with a time limit of 8 hours.

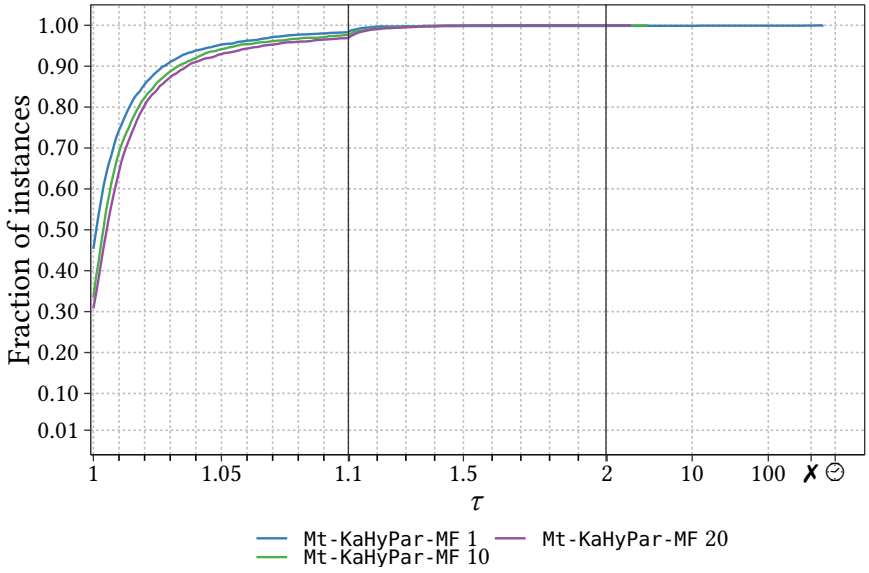

Figure A.4.: Performance plot comparing the solution quality of Mt-KaHyPar-MF with an increasing number of threads.

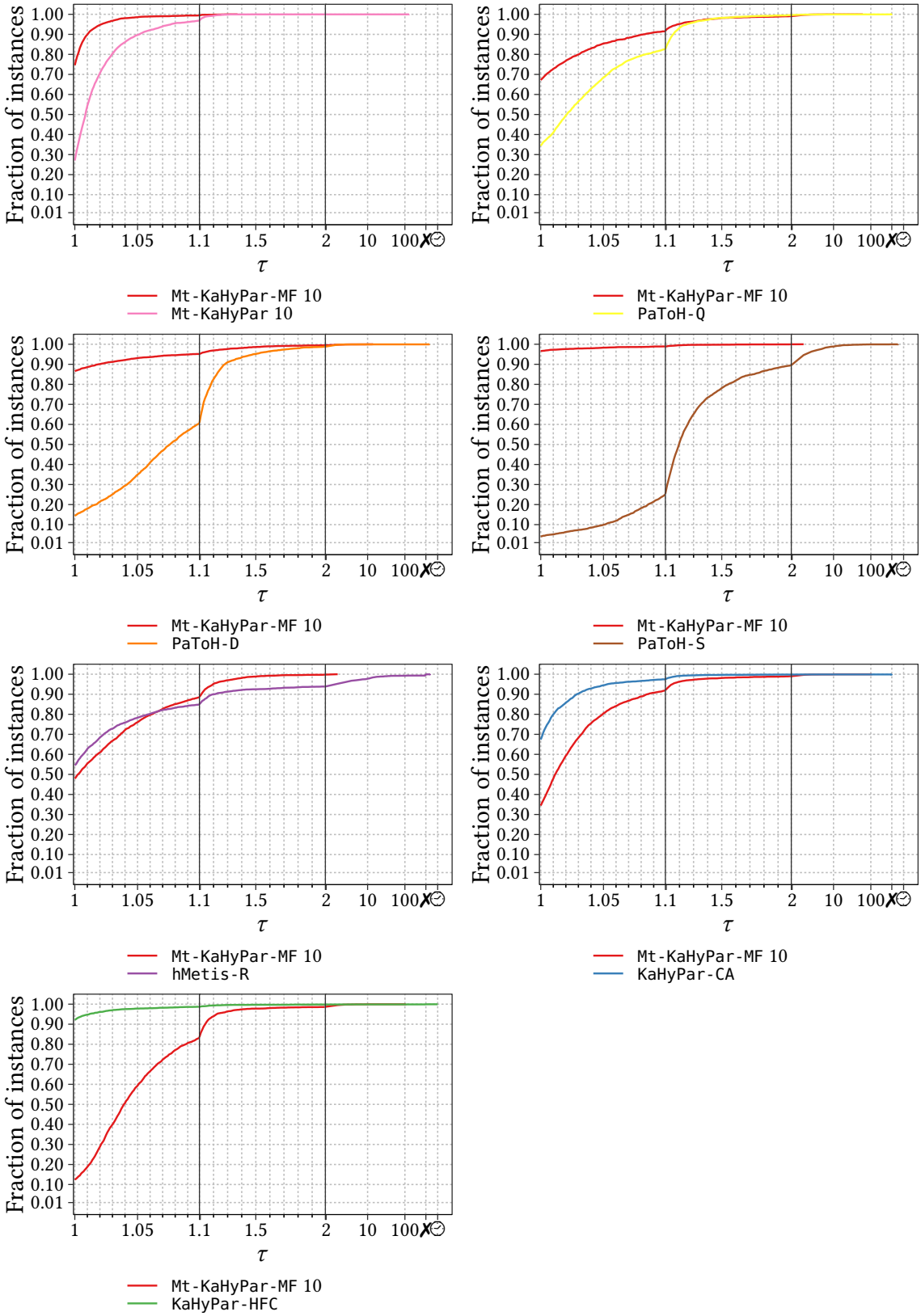

Figure A.5.: Performance plots directly comparing the solution quality of Mt-KaHyPar-MF (using 10 threads) with other state of the art hypergraph partitioners on our full benchmark set A.

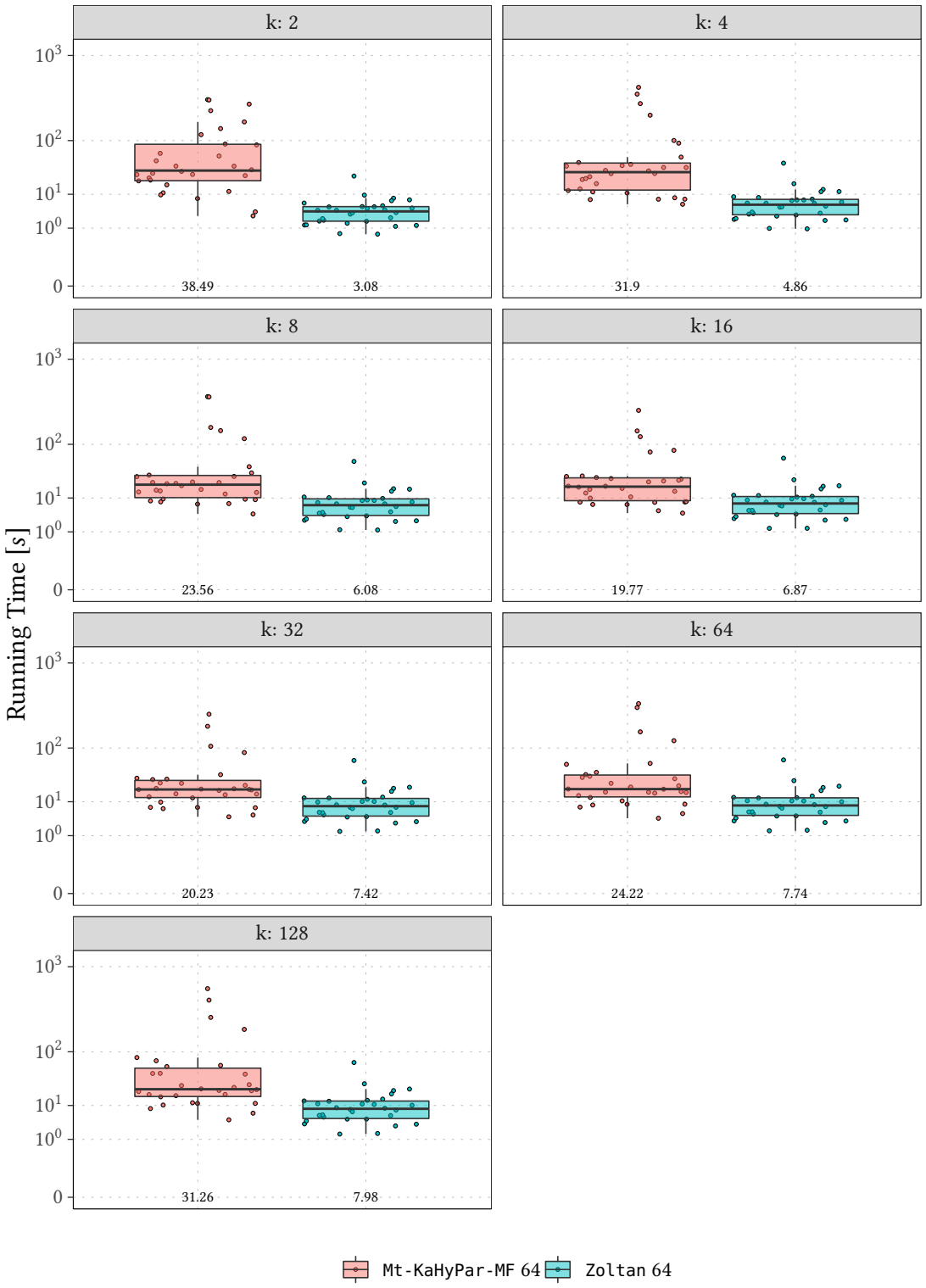

Figure A.6.: Direct comparison of the running time per k with Zoltan on our large hypergraph set C using machine $P3$ with 64 threads.