Karlsruher Institut für Technologie

Bachelor-Thesis

Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning

by Tobias Heuer

Advisors: Prof. Dr. Peter Sanders
M. Sc. Sebastian Schlag

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Date of submission: 14.08.2015

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 14.08.2015

Abstract

Hypergraphs are generalizations of graphs where an edge can consist of more than two nodes. A reoccurring task is to divide the node set of a hypergraph into k different non-empty parts where we simultaneously want to minimize a partitioning objective. This problem is called the *k-way* hypergraph partitioning problem. Applications can be found in the area of *VLSI* design and the parallelization of computationally intensive problems. A method to solve this problem is the *multi-level* partitioning scheme. This scheme consist of three phases. The hypergraph is first *coarsened*, then *initially partitioned* and at the end the quality of the partition is improved with a *local search* heuristic. A special case of this scheme is the n-Level hypergraph partitioning framework *KaHyPar*. This framework currently uses the external tools *hMetis* and *PaToH* for the initial partitioning phase. In this thesis we develop various initial partitioning methods with the goal to produce the same quality in the same amount of time as *hMetis* in this framework. Our final initial partitioner combines all developed methods into one single initial partitioning algorithm. The solution quality of *KaHyPar* with our initial partitioner is comparable to the quality with *hMetis* and 1% better as the quality with *PaToH* as initial partitioner. The initial partitions produced by our initial partitioning algorithm are 0.6% better than the partitions of *hMetis* and the running time is 16% faster on average.

Zusammenfassung

Hypergraphen sind Generalisierungen von Graphen bei denen eine Kante aus mehr als nur aus zwei Knoten bestehen kann. Eine immer wiederkehrende Aufgabe ist es die Knotenmenge eines Hypergraphen in k verschiedene nicht-leere Teile aufzuteilen, wo wir gleichzeitig versuchen eine Partitionierungszielfunktion zu minimieren. Dieses Problem wird direktes *k-way* Hypergraph Partitionierungsproblem genannt. Einige Anwendungen sind im Bereich des *VLSI*-Design und bei der Parallelisierung von rechenaufwendigen Problemen zu finden. Eine Methode dieses Problem zu lösen ist das *Multilevel*-Partitionierungsschema. Dieses Verfahren besteht aus drei Phasen. Der Hypergraph wird als erstes vergröbert, danach *initial* partitioniert und am Ende wird eine lokale Suchheuristik dazu genutzt um die Partitionierung zu verbessern. Eine Speziallisierung dieses Schema is das *n-level* Hypergraph Partitionierungsframework *KaHyPar*. Dieses Framework benutzt im Moment die externen Anwendungen *hMetis* und *PaToH* für den *initialen* Partitionierungsschritt. In dieser Arbeit entwickelten wir verschiedene initiale Partitionierungsalgorithmen mit dem Ziel die gleiche Qualität in der gleichen Laufzeit im Vergleich zu *hMetis* in diesem Framework zu produzieren. Der letztendlich beste initiale Partitionierer kombiniert alle entwickelten Methoden in einem einzigen Algorithmus. Die Qualität der Lösungen von *KaHyPar* mit unserem initialen Partitionierer ist vergleichbar mit der Qualität mit *hMetis* und 1% besser als die Qualität mit *PaToH* als initialem Partitionierer. Die initialen Partitionen die von unserem initialen Partitionierungsalgorithmus produziert werden sind 0.6% besser als die Partitionen von *hMetis* und die Laufzeit ist im Durchschnitt 16% schneller.

Acknowledgements

I want to thank several persons who supported me on the way to my bachelor thesis. First I want to thank M.Sc. Sebastian Schlag who supervised this work. He was the most important contact person and helped me with words and deeds. He also introduced me into the scientific work and tought me some new techniques in this area. Further he also took the time above his main work to discuss with me theoretical and practical problems in my work. Next, I want to thank my father. Because of his financial support I'm able to concentrate me completely on this work. Also I want to thank my girl friend Alessa who cool down my fuming head after hard days of work and give me the power to continue every day with the same energy and passion. Finally I would like to thank is my mother. Where ever she is now in the heaven she is all the time in my heart with her love she has given me. She said to me in her darkest hour, I should never lose the passion to work on the things I love. I know she would be proud of me and that makes me happy.

Contents

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Contributions . . . 1
1.3 Outline . . . 2

2 Preliminaries **3**
2.1 Definitions and Terminology . . . 3
2.2 Partitioning Objectives . . . 5
2.2.1 Cut Metric . . . 6
2.2.2 Sum of External Degree . . . 6
2.2.3 $(k-1)$ metric . . . 6
2.2.4 Absorption . . . 7

3 Related Work **8**
3.1 Multilevel Paradigm . . . 8
3.2 Coarsening . . . 9
3.2.1 Edge Coarsening . . . 9
3.2.2 Hyperedge Coarsening . . . 9
3.2.3 First Choice . . . 10
3.2.4 Heavy Connectivity Matching/Clustering . . . 10
3.3 Initial Partitioning . . . 11
3.3.1 Random Partitioning . . . 11
3.3.2 Hypergraph Growing Partitioning . . . 11
3.3.3 Greedy Hypergraph Growing Partitioning . . . 12
3.3.4 Label propagation . . . 13
3.3.5 Integer Linear Programming . . . 14
3.3.6 Recursive-Bisection . . . 15
3.4 Refinement . . . 16
3.4.1 Kernighan-Lin . . . 16
3.4.2 Fiduccia-Matheyses . . . 18
3.4.3 Delta-Gain Update . . . 20
3.5 nLevel-Partitioning . . . 21
3.6 State-of-the-art Hypergraph Partitioning Tools . . . 21
3.6.1 hMetis . . . 21
3.6.2 PaToH . . . 22

4 Initial Partitioning **23**
4.1 Direct k-Way Initial Partitioning . . . 23
4.1.1 Random . . . 23
4.1.2 Breadth-First-Search . . . 24
4.1.3 Greedy Hypergraph Growing . . . 26
4.1.4 Label Propagation . . . 35
4.1.5 Additional Implementation Details . . . 37
4.2 Recursive Bisection . . . 38
4.2.1 Adaptive Epsilon . . . 38
4.2.2 Implementation . . . 41

4.3 Recursive Bisection n-Level Hypergraph Initial Partitioning . . . 44
4.3.1 Pool Initial Partitioner . . . 44

5 Experimental Results **46**
5.1 Test-Environment . . . 46
5.1.1 Instances . . . 46
5.1.2 System . . . 46
5.1.3 Methodology . . . 46
5.2 Direct *k-way* and Recursive Bisection Initial Partitioners . . . 47
5.2.1 Multiple Runs . . . 50
5.2.2 Adaptive Epsilon . . . 51
5.3 Recursive Bisection n-Level Initial Partitioner . . . 52
5.3.1 Evaluation of the Coarsening Configuration . . . 52
5.3.2 Evaluation of the *Local Search* Configuartion . . . 54
5.3.3 Composition of the *Pool* Initial Partitioner . . . 57
5.3.4 Experimental Results of the Final Initial Partitioner . . . 58
5.4 Experimental Results of our Initial Partitioner in *KaHyPar* . . . 60

6 Conclusion **62**
6.1 Future Work . . . 62

A Benchmark Instances **68**

B Detailed Parameter Tuning results **70**
B.1 Contraction Limit and Hypernode Weight Bound . . . 70
B.2 Fruitless Moves of the *FM Local Search* algorithm . . . 72

C Detail Results **73**
C.1 Initial Partitioning results on the *medium sized* benchmark set . . . 73
C.2 Initial and Final Cuts of *KaHyPar* . . . 73
C.3 Overall Results of *KaHyPar* with different initial partitioner . . . 74

1 Introduction

Hypergraphs are generalizations of graphs where an edge can consist of more than two nodes. A hyperedge is a subset of the hypernode set of a hypergraph. Therefore the set of all undirected graphs is a subset of the set of all hypergraphs. Every problem, that can be modeled as a graph can therefore be modeled as a hypergraph, too. This leads to fact that, if we are able to provide solutions for hypergraphs problems, we are able to solve the same or similar problems on graphs and much more problems which cannot be represented by a graph.
Taking the example of the hypergraph partitioning problem (HGP) where we want to partition the hypernode set of a hypergraph into k non empty parts. The size of the resulting parts should be between some lower and upper bound and the goal is to minimize or maximize a partitioning objective. An application of hypergraph partitioning is to find in the VLSI physical design. The goal here is to partition a circuit into smaller submodules and to keep the wires as short as possible between it [4]. The nodes represents the gates and edges the wires between the gates of a circuit. Because one wire can connect more than two gates a hypergraph models this problem more precisely than a graph. Another application of the hypergraph partitioning problem (HGP) is to parallize the calculation of the sparse-matrix vector product [6].
The most common solving heuristic for the hypergraph partitioning problem is the multilevel partitioning scheme [6, 16]. This scheme is divided into three phases. The coarsening phase successively merges node pairs with the goal to simplify the original hypergraph for the next step. The resulting hypergraph is then divided into k differents parts in the initial partitioning phase. After the initial partitioning takes place the merged nodes are unpacked in reverse order and a local search algorithm is used to further improve the solution quality. This phase is called the refinement phase. The state-of-the-art hypergraph partitioning frameworks, hMetis and PaToH, implements this scheme.
The HGP moves in the last three decades more and more into focus of research. The main applications area can be summarize in the area of parallelization and to simplify problem instances. This thesis investigates the aspect of the initial partitioning of a hypergraph in a multilevel partitioning scheme.

1.1 Problem Statement

The n-level hypergraph partitioning framework *KaHyPar* is a special case of the multilevel partitioning scheme. For the initial partitioning step, this framework uses the state-of-the-art partitioner *hMetis* as extern tool. The usage of *hMetis* as initial partitioner has several disadvantages. First we can only use this tool as a black box and therefore we cannot control the internal behaviour. The quality of such a framework depends on the quality of the initial partitioner. We cannot change the code base of an extern tool and have to accept the produced results. Therefore we limit the quality of the overall framework. To make *KaHyPar* independent from extern tools, it becomes necessary to develop an own initial partitioner. The goal is to produce with our work the same quality in the same amount of time in this framework as with *hMetis* as initial partitioner. Further the final initial partitioner should be integrated into *KaHyPar* and it shall be verified, if we reach our own goals with our independent initial partitioner.

1.2 Contributions

We have implement and evaluate various direct *k-way* and *recursive bisection* initial partitioning algorithms. In addition to the standard partitioning methods, we adapt the idea of *label prop-*

agation to a working initial partitioning algorithm. We include existing local search heuristics from *KaHyPar* to improve the quality of our partitions after each partitioning. Our *recursive bisection* methods produces 20% better quality and more as their corresponding direct *k-way* implementations. By introducing multiple runs of each initial partitioner on each bisection, we have increased the quality of each evaluated *recursive bisection* partitioner between 20% and 25%. To fulfill the imbalance at the end of *recursive bisection* we developed an adaptable ε'. We adapt ε' before each bisection in a way, such that it allows maximum imbalance while still being small enough to ensure the final imbalance with ε. In our experiments only 32 of 93600 partitions had an imbalance greater than the initial imbalance ε. Further we use the existing framework *KaHyPar* and our developed *recursive bisection* implementation to provide a *recursive bisection n-level* initial partitioner. As initial partitioner in the n-level context we use a combination of all developed variants before, which we call *pool* initial partitioner. This partitioner executes a subset of our implemented initial partitioning algorithm multiple times and take the best produced partition from all runs as final partition.
To evaluate our initial partitioning algorithm we compared our initial partitioner against *hMetis* and *PaToH* as intial partitioner in the direct n-level hypergraph partitioning framework *KaHyPar* on our benchmark set. The solution quality of *KaHyPar* with our initial partitioner is comparable to the quality with *hMetis* and 1% better as the quality with *PaToH* as initial partitioner. The initial partitions produced by our initial partitioning algorithm are 0.6% better than the partitions of *hMetis* and the running time is 16% faster on average.

1.3 Outline

In Section 2 we introduce in theoretical definitions and notations. We define the hypergraph partitioning problem and introduce the most prominent partitioning objectives for this problem. Section 3 gives an overview about related work in the area of hypergraph partitioning. The main topic of this thesis about initial partitioning is presented in Section 4, which describes the developed algorithms. Section 5 presents the evaluation of our work. Further we present the results of our initial partitioner in the direct n-level hypergraph partitioning framework *KaHyPar* in comparison with the state-of-the-art partitioner *hMetis* and *PaToH*. In the last Section 6 we conclude the work and give an overview about future work.

2 Preliminaries

In this section we outline the most relevant definitions and terminology in the area of hypergraph partitioning.

2.1 Definitions and Terminology

Our initial partitioning framework is part of the *KaHyPar* partitioner. The terminology is therefore similiar to [13].

Definition 2.1. An undirected weighted hypergraph $H = (V, E, c, \omega)$ is a set of hypernodes V and a set of hyperedges E with a hypernode weight function $c : V \to \mathbb{R}_{\geq 0}$ and a hyperedge weight function $\omega : E \to \mathbb{R}_{\geq 0}$. A hyperedge e is a subset of V (formally: $\forall e \in E : e \subseteq V$).

A hypergraph generalizes a graph by extending the definition of an edge, which can contain more then only two nodes. Hyperedges are also called nets in the literature [7] and the vertices of a hyperedge are called pins. We can extend the definition of the weight functions c and ω to sets. Let $V' \subseteq V$ and $E' \subseteq E$ then we define

$$c(V') = \sum_{v' \in V'} c(v')$$

$$\omega(E') = \sum_{e' \in E'} c(e')$$

Definition 2.2. In the following we are listing important terminology.

(i) Vertex v is *incident* to a hyperedge $e \Leftrightarrow v \in e$.

(ii) Two vertices v_i and v_j are *adjacent* $\Leftrightarrow \exists e \in E : v_i \in e \wedge v_j \in e$.

(iii) We define $I : V \to \mathcal{P}(E)$, which maps a vertex v to all its *incident* nets.

(iv) The *degree* of a hypernode v is defined as $d(v) = |I(v)|$

(v) A hypernode v is an *isolated* hypernode $\Leftrightarrow d(v) = 0$

(vi) The set $\Gamma(v) = \{u \mid \exists e \in E : v \in E \wedge u \in E\}$ defines all adjacent vertices of a hypernode v. A vertex $u \in \Gamma(v)$ is called *neighbour* of v

(vii) The size of a net e is the cardinality $|e|$ and a hyperedge with size $|e| = 1$ is called *single-node* net.

(viii) Two nets $e_i, e_j \in E$ are parallel, if $e_i = e_j$.

Figure 1 shows an example of a hypergraph. We can define the hypergraph in this figure according to definition 2.1 as follows:

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$E = \{e_1 = \{v_1, v_2, v_3\}, e_2 = \{v_2, v_3\}, e_3 = \{v_3, v_5, v_6\}, e_4 = \{v_4\}\}$$
$$\forall v \in V : c(v) = 1$$
$$\forall e \in E : \omega(e) = 1$$

Vertex v_2 is incident to hyperedges e_1 and e_2. The same vertex is adjacent to vertex v_1, because both are contained in hyperedge e_1. The set of incident nets of hypernode v_2 is $I(v_2) = \{e_1, e_2\}$ and that implies the degree $d(v_2) = |I(v_2)| = 2$. The neighborhood of vertex v_2

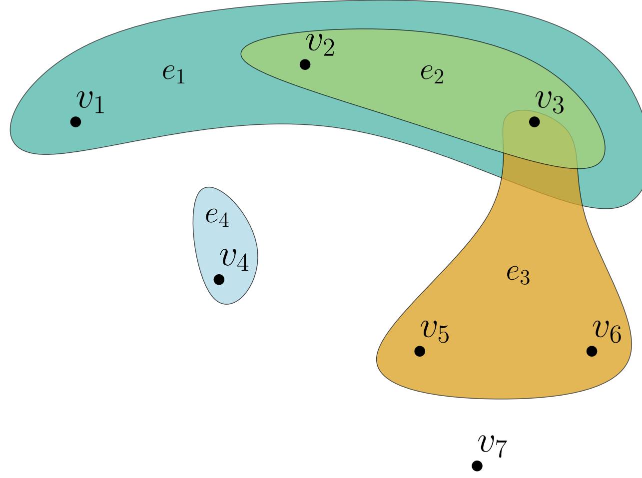


Figure 1: An example of a hypergraph

is $\Gamma(v_2) = \{v_1, v_2, v_3\}$. Vertex v_7 is an isolated hypernode, because $\forall e \in E : v_7 \notin e \Rightarrow d(v_7) = 0$. The size of hyperedge $e_1 = \{v_1, v_2, v_3\}$ is $|e_1| = 3$. e_4 is a *single node* net.

Definition 2.3. Given a hypergraph $H = (V, E, c, \omega)$ and two hypernodes $u, v \in V$. If we contract two hypernodes u and v to one hypernode v', we have to define a new hypergraph $H' = (V', E', c', \omega)$

$$V' = (V \cup v') \setminus \{u, v\}$$

$$E' = \{e \mid e \in E : u, v \notin e\} \cup \{(e \cup v') \setminus \{u, v\} \mid e \in (I(v) \cup I(u))\}$$

$$c'(z) = \begin{cases} c(z), & z \notin \{u, v\} \\ c(v) + c(u), & z = v' \end{cases}$$

Formally a contraction of two hypernodes u and v means to merge the two verticies into one new hypernode v'. The weight of the new hypernode v' is the sum of the weights of the two contracted hypernodes. In every hyperedge e, where either u or v occurs, we delete these pins from the net e and add the new hypernode v'. Contraction of two hypernodes u and v might lead to parallel and *single node* nets. Assume we have two nets $e_i, e_j \in E$ with $e_i \triangle e_j = \{u, v\}$, where $\triangle$ is the symmetric difference. If we contract u and v than $e_i = e_j$. If two nets are parallel after contraction, we can remove e_j and set the hyperedge weight of e_i to $\omega(e_i) = \omega(e_i) + \omega(e_j)$. A *single node* net e can occur, if we want to contract u and v and $e = \{u, v\} \Rightarrow e = \{v'\}$. In the hypergraph partitiong problem (see definition 2.8) such a net can never become cut and we can remove this net. Before we define the hypergraph partitioning problem, we have to formally define a *k-way partition* P of a hypergraph H and some terminology we combine with this partition P.

Definition 2.4. A *k-way partition* $P = \{V_1, ..., V_k\}$ of a hypergraph H is a division of V into k parts, such that the following constraints are fulfilled:

(i) $\bigcup_{i=1}^{k} V_i = V \quad \wedge \quad \forall V' \in P : V' \neq \emptyset$

(ii) $\forall i, j \in \{1, .., k\} : i \neq j \Rightarrow V_i \cap V_j = \emptyset$

(iii) P is *ε-balanced* $\Leftrightarrow \forall V' \in P : c(V') \leq L_{max} := \lceil \frac{c(V)}{k} \rceil (1 + \varepsilon) + \max_{v \in V} c(v)$

Given the definition above a *k-way* partition P of a hypergraph H is a division of the hypernodes

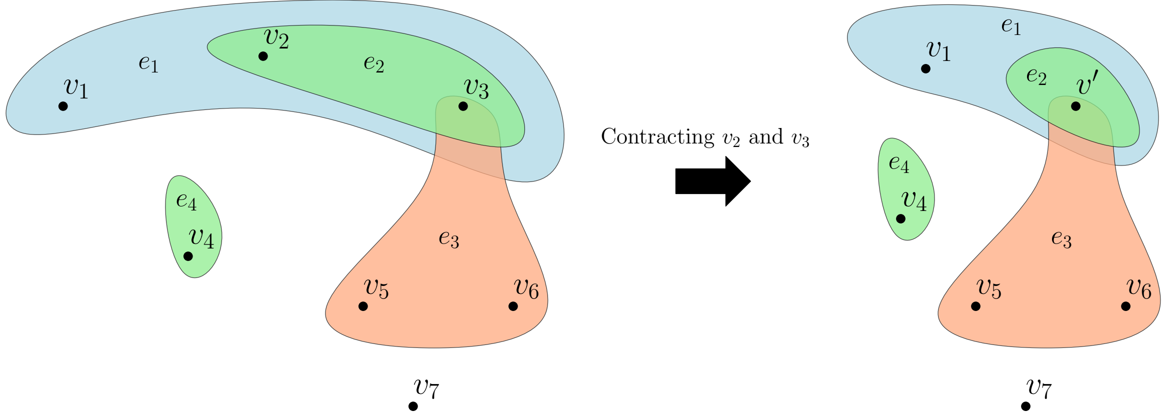


Figure 2: An example of a contraction. Hypernodes v_2 and v_3 are contracted into a single hypernode v'. Hypernode v_2 is incident to hyperedges e_1 and e_2 and v_3 is incident to e_2 and e_3. The resulting hypernode v' is incident to all incident hyperedges of v_2 and $v_3 \Rightarrow v'$ is incident to e_1, e_2 and e_3. The hyperedge e_2 consists only of the contraction pair v_2 and $v_3 \Rightarrow e_2$ becomes a *single node* net.

V into k disjoint subsets, where the weight of each subset has to be smaller than an upper weight bound L_{max}. The k subsets/parts of P are not allowed to be empty.

Definition 2.5. We define the *number of pins* of a net e in a block $V_i \in P$ with

$$\phi(e, V_i) := |\{v \in V_i \mid v \in e\}|$$

A net e is connected to a block V_i, if $\phi(e, V_i) > 0$. A block V_i is adjacent to a vertex $v \notin V_i$, if there $\exists e \in I(v) : \phi(e, V_i) > 0$.

Definition 2.6. Given a *k-way* partition P of a hypergraph H. We define the *connectivity set* of a net e as $\Lambda(e) = \{V_i \mid \phi(e, V_i) > 0\}$. The *connectivity* of a net e is the cardinality of the connectivity set $\lambda(e) := |\Lambda(e)|$.

Definition 2.7. A hyperedge e is *cut*, if $\lambda(e) > 1$ and *internal*, if $\lambda(e) = 1$.

The connectivity set $\Lambda(e)$ of a net e in a *k-way* partition P contains all blocks, which the net e is connected to. A vertex, which is at least in one cut edge is called a border vertex.

Definition 2.8. The *k-way hypergraph partitioning* problem is to find an ε-balanced k-way partition of a hypergraph H, which minimizes the sum of all weights of all cut edges.

$$\kappa(H, P) = \sum_{\substack{e \in E \\ \lambda(e) > 1}} \omega(e) \tag{2.1}$$

2.2 Partitioning Objectives

We present the hypergraph partitioning problem, where we want to minimize the weight of all cut edges of a ε*-balanced* *k-way* partition P. There exist several other objectives which should be minimized or maximized in the hypergraph partitioning context. In the following sections we assume that we have a ε*-balanced* *k-way* partition of a hypergraph $H = (V, E, c, \omega)$.

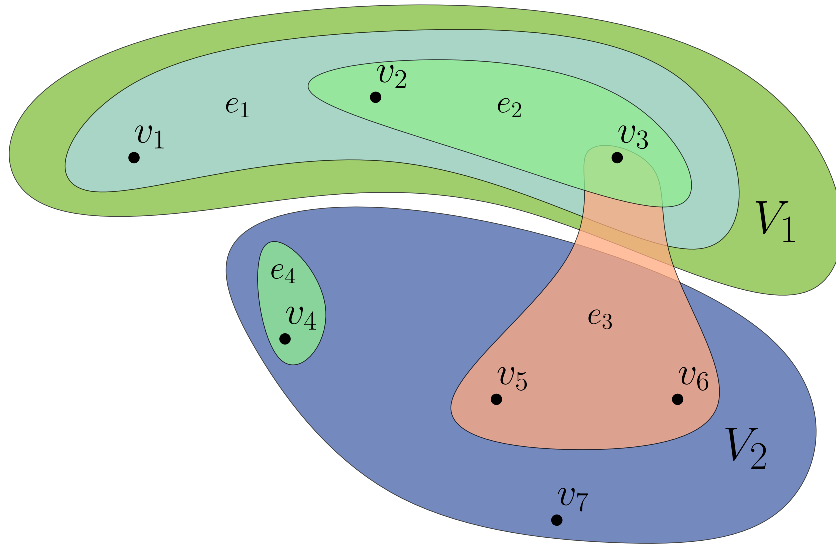


Figure 3: An example of a perfect balanced *2-way* partition $P = \{V_1, V_2\}$ of a hypergraph. The blocks $V_1 = \{v_1, v_2, v_3\}$ and $V_2 = \{v_4, v_5, v_6, v_7\}$ are perfect balanced, because the cardinality of V is odd. There is only one cut hyperedge e_3.

2.2.1 Cut Metric

The cut objective is defined in equation 2.1. This metric is the sum of the weight of all cut edges and we want to minimize it in the hypergraph partitioning context.

2.2.2 Sum of External Degree

The next interesting metric is the sum of external degree (*SOED*). The *external degree* of a hyperedge e is the connectivity of e, if e is a cut net and zero otherwise [17]. The definition of the *SOED* [26] is

$$\nu(H, P) = \sum_{\substack{e \in E \\ \lambda(e) > 1}} \omega(e)\lambda(e) \tag{2.2}$$

The goal of this metric is to minimize the amount of cut edges and if a hyperedge is cut, than we want to minimize the connectivity of that net. If $\omega \equiv 1$ we can explain the *SOED* as sum of the connectivity of all cut edges. Because we sum in both metrics (hyperedge cut and SOED) over the weight of all cut edges and for those nets e $\lambda(e) > 1$, we note that $\kappa < \nu$ for all hypergraphs H and partitions P.

2.2.3 $(k - 1)$ metric

The $(k - 1)$ metric is very similiar to the definition of the *SOED* [28].

$$\tau(H, P) = \sum_{e \in E} \omega(e)(\lambda(e) - 1) \tag{2.3}$$

The goal for this metric is to minimize it.

2.2.4 Absorption

The *Absorption* is defined as follows [2].

$$\psi(H,P) = \sum_{i=1}^{k} \sum_{\substack{e \in E \\ e \cap V_i \neq \emptyset}} \frac{|e \cap V_i| - 1}{|e| - 1} \cdot \omega(e) \tag{2.4}$$

If we have a net e, which is an internal edge of block V_i, the fraction in the inner sum is 1 [2]. If the net e is only connected via one vertex to block V_i, the inner fraction is zero [2]. If we use the absorption metric, the goal is to maximize it. Note, if every net is an internal hyperedge than $\psi(H,P) = \omega(E)$.

3 Related Work

In this section we describe the related work into the area of hypergraph partitioning. First we introduce the multilevel partitioning scheme in Section 3.1. In the following sections we outline different techniques, which are used in the multilevel context. At the end we introduce the n-level partitioning scheme in Section 3.5 and decribe the state-of-the-art partitioner *hMetis* in Section 3.6.1 and *PaToH* in Section 3.6.2.

3.1 Multilevel Paradigm

The *multilevel paradigm* (see figure 4) is used by all state-of-the-art hypergraph partitioner. This paradigm is divided into three phases.
In the first step, the *coarsening phase*, a set of hypernodes are choosen to be contracted. We describe different techniques to choose the hypernodes in section 3.2. The contracted and remaining hypernodes build a smaller hypergraph. This procedure is repeated on the smaller hypergraph until a predefined *contraction limit* is reached. Normally the *coarsening* is performed until $i \cdot k$ hypernodes are left with $i \in \mathbb{N}$.
If the hypergraph is small enough, an *initial partition* is generated in the second phase on the resulting hypergraph of the *coarsening* process. There exist serveral techniques which are detail described in Section 3.3. Random approaches assign hypernodes to a random block of a partition. *Growing*-based techniques grow a cluster around a selected hypernode by assigning those to a block. *Greedy-Growing*-techniques also grow a cluster around a selected vertex, but the hypernodes, which are added to the cluster, are choosen based on a scoring function. We can add a hypernode e.g. to the growing part by calculating the decrease of the hyperedge cut, if we would assign it to the growing block.
In the last phase, the *refinement/uncontraction phase*, the contracted hypernodes are successively uncontracted in reverse order of contraction and the partitioning is projected to the next level finer hypergraph. After projection to the next level the quality of the partition can be improved with local search algorithms. Popular heuristics are the Kernighan-Lin (Section 3.4.1) or Fiduccia-Mattheyses (Section 3.4.2) heuristics.

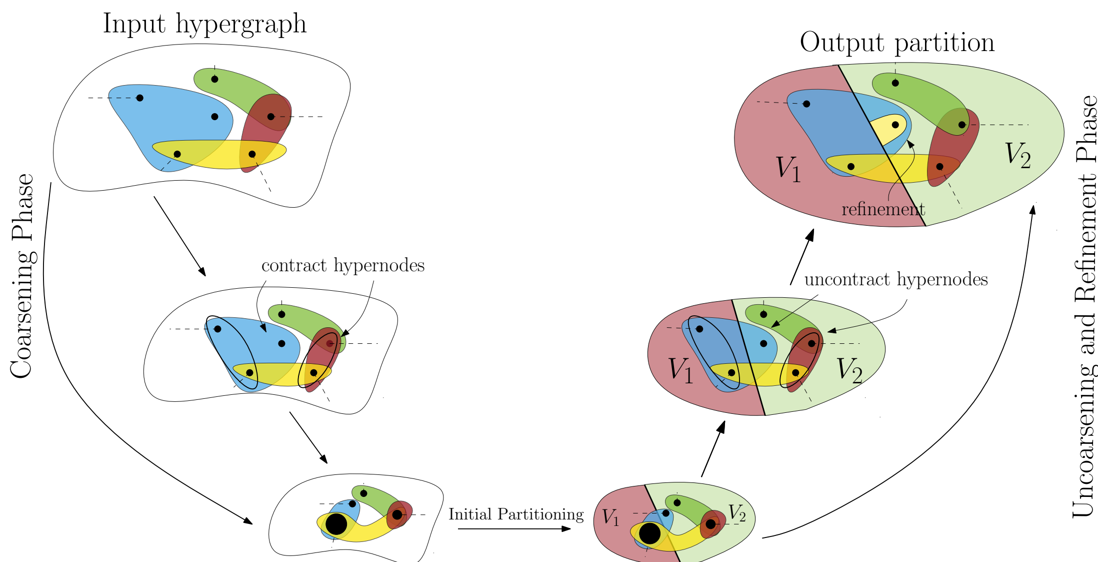


Figure 4: Multilevel partitioning scheme

3.2 Coarsening

The *coarsening* phase is the first step of the multilevel partitioning scheme. Karypis [19] outline three main purposes in this step. First, a coarsening should provide a smaller hypergraph where the initial partitioning of the smaller hypergraph is not significantly worse than on the original hypergraph. The next goal is to provide a coarsened version of a hypergraph where *local search* algorithms become effective during the *uncontraction* phase. The last one is that *coarsening* should reduce the size of hyperedges. *Local search* techinques work significantly better on hypergraphs with small hyperedges, because they are easier to remove from the cut. In general we can distinguish between two classes of coarsening algorithms. *Agglomerative* coarsening schemes choose at one time step two hypernodes for contraction. *Hierarchical* coarsening schemes choose as many hypernode pairs as possible to merge it at one time step [2, 6]. In this section we give a short introduction about various heuristics, which are used by the state-of-the-art partitioner to contract hypernodes of a hypergraph.

3.2.1 Edge Coarsening

Edge coarsening (EC) is a matching-based coarsening scheme and provided by *hMetis* [19]. If a hypernode v should be matched with another one, all unmatched hypernodes $u \in \Gamma(v)$ are considered. The hypernode u with the largest edge weight between v and u forms a matching pair (u, v) (for an example see figure 5). The weight of an edge between two hypernodes u and v is defined as sum of all *edge-weights* of hyperedges that contain u and v. The *edge-weight* of a hyperedge e with weight 1 is defined as $\frac{1}{|e|-1}$. Formally, we define the weight of an edge $\omega' : V \times V \to \mathbb{R}$ between to hypernodes u and v in a hypergraph H with $\forall e \in E : \omega(e) = 1$ as follows:

$$\omega'(v, u) = \sum_{e \in I(v) \cap I(u)} \frac{1}{|e| - 1}$$

The contraction partner $u \in \Gamma(v)$ maximize $\omega'(v, u)$. Ties are broken randomly and the hypernodes are visited in random order. With this definition the hypergraph is implicitly treated as a graph by replacing the hyperedges with the clique expansion [12, 19].
We can take the weight of a hyperedge into account and define the *edge-weight* as $\frac{\omega(e)}{|e|-1}$. We call this coarsening heavy-edge coarsening.

3.2.2 Hyperedge Coarsening

The hyperedge coarsening (HEC) is a hierarchical coarsening scheme and also implement by *hMetis* [15]. HEC looks for an independent set of hyperedges. A independent set $S \subseteq E$ of hyperedges of a hypergraph H is a set, where $\forall e_1, e_2 \in S : e_1 \cap e_2 = \emptyset$. The hyperedges are visited in decreasing order of their weight and a hyperedge is matched, if non of it pins is matched before. We contract all hyperedges instead of hypernode pairs (for an example see figure 5).
A majority of the hyperedges are not contracted during the *hyperedge coarsening*, because the pins of these hyperedges are matched before. This observation has two main disadvantages [15]. The size of hyperedges decrease not significantly after each contraction step. Local search algorithms become therefore less effective. The next problem is the different distribution of hypernode weights at the end of the coarsening process. This destroys potentially the structure of the contracted hypergraph. To solve these two problem Karypis [15] develop the *modified hyperedge coarsening* (MHEC). After the hyperedge coarsening has taken place, the list of

hyperedges is traversed again. All unmatched pins of a hyperedge are matched in this second step.

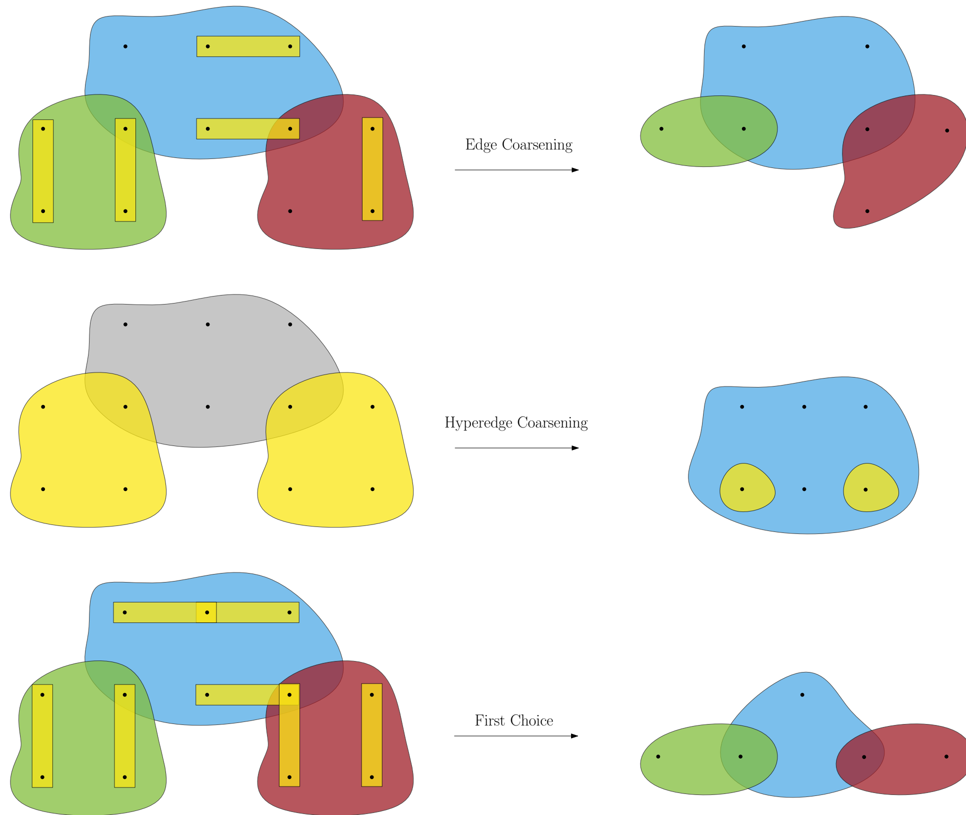


Figure 5: Example of the EC, HEC and FC coarsening scheme. The yellow shapes in all variants indicate the contracted hypernodes in the coarsening schemes.

3.2.3 First Choice

The two coarsening strategies described before both search for an independent set of hyperedges or hypernode-pairs. The problem is that the independence property of such a coarsening can destroy existing clusters of vertices in a hypergraph [19]. Based on this observation Karypis et al. [19] developed the *First Choice* coarsening scheme, which is very similar to *Edge Coarsening*. The difference is, if a hypernode v is visited all matched and all unmatched neighbours $u \in \Gamma(v)$ are considered as contraction partner. We are choosing again the hypernode u which maximizes $\omega'(v, u)$. The chosen hypernode u could already be matched before with another hypernode. Ties are broken randomly, but we favour unmatched hypernodes.

3.2.4 Heavy Connectivity Matching/Clustering

Heavy Connectivity Matching (HCM) is a matching-based coarsening scheme and implemented by PaToH [6]. The algorithm works similar to EC, but uses a different rating function:

$$\omega'(v, u) = |I(v) \cap I(u)|$$

We choose a contraction partner u to an visited hypernode v which maximizes $\omega'(v, u)$. The contraction pair (u, v) has the property to be highly connected via many hyperedges.
PaToH [6] also uses an agglomerative variant of this algorithm, which is similar to *First Choice*.

At the beginning all hypernodes are singleton clusters $C_u = \{u\}$. During *coarsening* we only visit those singleton clusters. If we want to find a contraction partner for a singleton cluster we consider all incident singleton and multinode clusters. We choose the cluster C_v as contraction partner for C_u which maximizes

$$\omega''(C_u, C_v) = \frac{|I(u) \cap (\cup_{v \in C_v} I(v))|}{c(u \cup C_v)}$$

Again the goal is to find highly connected clusters, but very heavy clusters should be avoided with the division by $c(u \cup C_v)$. This method is called the *heavy connectivity clustering* (HCC).

3.3 Initial Partitioning

The goal of the initial partitioning phase is to partition the coarsened hypergraph into k blocks. Since the resulting partition of the multilevel paradigm depends on the quality of the *initial partitioning* step, we review the most prominent initial partitioning algorithms in the following subsections.

3.3.1 Random Partitioning

The easiest way to produce a *k-way* partition is to random assign hypernodes to blocks. Note that no part should be heavier than the maximum allowed partition weight, which is fixed by our ϵ.
Another random partitioning techniques is the *Random Engeenering Method* (REM) [23]. We sort all hypernodes in decreasing order of their weights and assign it in this order. Assignment probabilities to a part V_i are proportional to the hypothetical area remaining before the partition weight of block i will satisfy the minimal bound of partition weights l (see equation 3.2). Assume we have such a lower bound of partition weight l, then we can define the sum of all remaining areas to satisfy this lower bound on all blocks as

$$R = \sum_{i=1}^{K} \max(l - c(V_i), 0) \tag{3.1}$$

Now can define the probability p_i of assigning a hypernode v_j to part V_i

$$\mathrm{p}_i = \max(\frac{l - c(V_i)}{R}, 0) \tag{3.2}$$

This assignment strategy keeps the block weights equal and provides a good degree of randomness. If all parts reach the lower bound, we replace l in (3.1) and (3.2) with the upper bound u.

3.3.2 Hypergraph Growing Partitioning

Hypergraph Growing Partitioning methods (HGP) create a partition by succesively growing blocks around selected seed vertices. Hypernodes, which are connected to the growing partition, are added until the partition satisfies the balance constraint. This assignment can be done in a breath- or depth-first manner. PaToH [5] and hMetis [16] provide this technique in their initial partitioning sets.
The result of the partitioning is sensitive to the selection of the start nodes. Start hypernodes can be selected randomly or we are looking for pseudo peripheral hypernodes [25], which should

be far away from each other. We find these start nodes by selecting a random hypernode and execute a breath-first-search from this hypernode. The last hypernode touched by this search is the second start hypernode.
Diekman [10] developed a framework called *Bubble* (BUB) for finite element meshes which is able to compute a *k-way* partition of a graph. BUB searches for k pseudo peripheral seed nodes. First, one node v with minimum degree is chosen. In finite element meshes this is usually a corner node. From v we search with a breadth-first-search the second initial seed node v_1 as described above. We use these two nodes as start nodes for the next breadth-first-search to find a third start node v_2. This process is repeated until we found k start nodes.
With these initial seed nodes a breadth-first-search partitioning is performed, where the block with the smallest amount of nodes receives the next node. If we touch all nodes, we calculate the center nodes of all blocks and uses them to repeat the whole process. Figure 6 illustrates the bubble algorithm. The algorithm terminates, if the seed nodes stop changing or there are no improvement within 10 iterations.

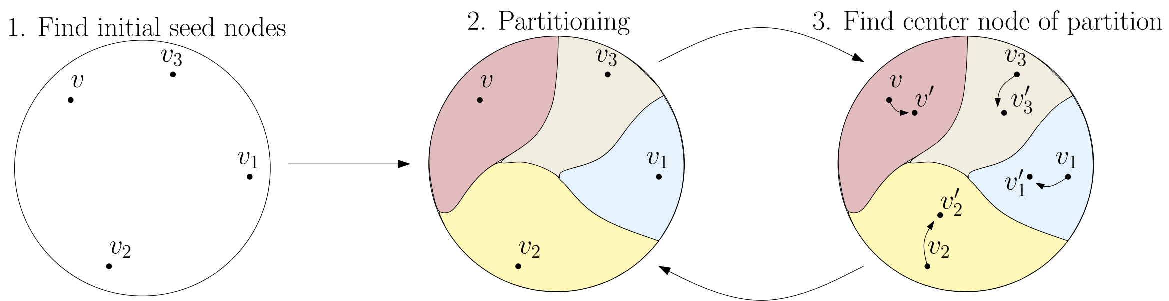


Figure 6: Illustration of the bubble algorithm.

3.3.3 Greedy Hypergraph Growing Partitioning

Karypis and Kumar [18] extend the graph growing partition algorithm to hypergraphs. They grow also a block V_1 around a randomly selected vertex. They define the gain of a hypernode v, which is connected to the growing block, in the reduction of the cut, if we would assign v to V_1. If we want to create a bisection of a hypergraph, we have to add all border vertices v with $v \notin V_1$ and their corresponding gain into a priority queue q. The hypernode with the highest gain in q is added to the growing part V_1. Afterwards the gains of all hypernodes in q are updated and all new border verticies v' (with $v' \notin V_1 \land v' \notin q$) are inserted into the priority queue. The algorithm stops if the weight of V_1 reaches a predefined upper bound. At the end we have two blocks V_1 and $V_2 = V \setminus V_1$.
PaToH [5] uses different gain function in their greedy hypergraph growing partitioning (GHGP) variants.

GHGP Max-Pin. A variant, which prefer moves of hypernodes based on the number of incident hypernodes in the growing block. If we want to define the gain of a move of a hypernode v from his current part V_i to V_j, the gain can be calculated with the following formula

$$g_{Max-Pin}(v, V_i, V_j) = |\{u \mid u \in I(v) \land u \in V_j\}|$$

GHGP Max-Net A variant, which prefer moves of hypernodes based on the number of incident hyperedges that connects the growing block with the hypernode. If we want to define the gain of a move of a hypernode v from his current part V_i to

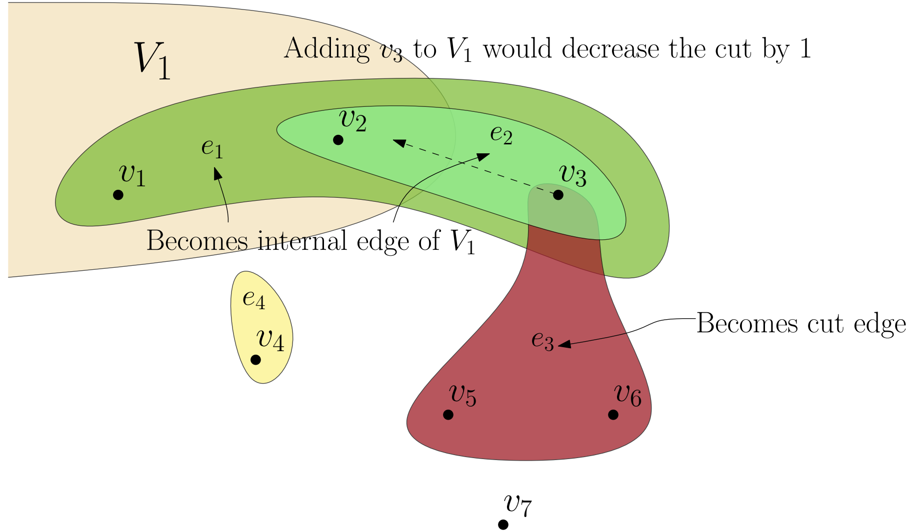


Figure 7: Illustration of moving an unselected hypernode v_3 to the growing partition V_1. The hypernode v_3 decreases the cut by 1. The green edges become internal edges of V_1 and the red hyperedge becomes a cut edge.

V_j, the gain can be calculated with the following formula

$$g_{Max-Net}(v, V_i, V_j) = |\{e \mid e \in \Gamma(v) \wedge V_j \in \Lambda(e)\}|$$

Since GHGP is sensitive to the choice of the start vertex, the most common implementations uses multiple runs [5, 18] and choose the best partition from that runs. Karypis and Kumar [18] note that GGGP is still faster than GGP, because the cuts of GGGP are on average better than of GGP and so fewer runs are needed to produce good solutions. If we want to implement GHGP efficient, similar datastructures are needed, like they are used in the Fiduccia-Mattheyses algorithm (see Section 3.4.2), and delta-gain updates [13] (see Section 3.4.3) should be implemented.

3.3.4 Label propagation

Another idea of an initial partitioning algorithm is outlined by Henne [12] in his future work. He describes how label propagation can be adapted to produce initial partitions.
The basic idea of label propagation is introduced by Raghavan et al. [24] to detect community structures. The algorithm visits the nodes of a graph in random order and to every node is a label assigned, which denotes the current community. If a node v is visited, the occurrences of labels of adjacent nodes are calculated and the label with the maximum occurrence count is chosen as the new label for v. Ties are broken randomly. This process can be repeated until the labels of the nodes did not change any more or a maximum amount of iterations is reached. A pseudocode of label propagation is listed in 1. Because every node is visited once and every edge is visited twice in each iteration the time complexity of this algorithm is near linear.
Henne [12] described the adaption of label propagation to an initial partitioning algorithm as follows. Initially, all but $k \cdot \lambda$ nodes have an empty label and the other nodes are assigned to one of the k blocks randomly for some tuning parameter λ. Label propagation is performed until all nodes with an empty label are assigned to one block. As score function the reduction of the cut can be used to calculate the score to all adjacent labels of a hypernode v.

Algorithm 1: Label Propagation

```
Data: (Hyper)graph H = (V, E, c, ω)
Result: label[1...n]
1 for v ∈ V do
2     label[v] ← v
3 i ← 0
4 while !converged ∧ i ≤ max_iteration do
5     for v ∈ V in random order do
6         label[v] ← argmax_l score(v, l)    // Best adjacent label according to a specific score
              function
7     i++
```

3.3.5 Integer Linear Programming

Kucar [21] formulates the hypergraph partitioning problem as an Integer Linear Programm (ILP). In the following, we briefly introduce on this formulation. First the following indicator variables are needed:

$$x_{ik} = \begin{cases} 1, & \text{if } v_i \in V_k \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

$$y_{jk} = \begin{cases} 1, & \text{if } \Lambda(e_j) = \{V_k\} \\ 0, & \text{othwerwise} \end{cases} \tag{3.4}$$

Variable x_{ik} is true, if hypernode v_i belongs to block k and variable y_{jk} is true, if the hyperedge e_j is an internal net of part V_k.
With the following objective function we try to maximize the number of non cut hyperedges in our ILP:

$$\max \sum_{j=1}^{|E|} \sum_{l=1}^{k} y_{jl} \tag{3.5}$$

To maximize this function hypernodes of a hyperedge should be placed in the same block. If we want to adapt the hypergraph partitioning problem completly to an ILP, we have to define several constraints to our indicator variables. First we have to ensure that all hypernodes can only be in one block. Equation (3.6) takes this into account.

$$\sum_{l=1}^{k} x_{il} = 1, \quad \forall i = 1, ..., |V| \tag{3.6}$$

In the next step we define a lower α_l and upper bound β_l for the part weigts.

$$\alpha_l \leq \sum_{i=1}^{|V|} x_{il} \cdot c(v_i) \leq \beta_l, \quad \forall l = 1, ..., k \tag{3.7}$$

The next inequality 3.8 ensures that y_{jl} is 1, only if all hypernodes $v_i \in e_j$ are part of block V_l.

$$y_{jl} \leq x_{il}, \quad \forall j = 1, ..., |E|, \quad l = 1, ...k, \quad v_i \in e_j \tag{3.8}$$

At the end we have to define the domain of our indicator variables.

$$\forall i \in \{1, ..., |V|\} : \forall l \in \{1, ..., k\} : x_{il} \in \{0, 1\} \tag{3.9}$$

$$\forall j \in \{1, ..., |E|\} : \forall l \in \{1, ..., k\} : y_{jl} \in \{0, 1\} \tag{3.10}$$

If we relax the problem to a *Linear Programm* with a real solution space, we can solve the problem with a *LP* solver. After solving the *LP*, we can relax the solution back to a natural solution.

3.3.6 Recursive-Bisection

Most literature describes only bisection-based partitioning algorithms. The reason is that most partitioning frameworks use only *recursive bisection* to obtain a k-partition of a hypergraph. This scheme bisects the original hypergraph in two parts and afterwards the two resulting blocks are recursively further bisected until we have k parts (see figure 8). After each bisection we can improve our partitioning objective with a *local search* heuristic. Note that with this description k is restricted to be a power of 2.
To extend recursive bisection for any k, we use the definition from Schulz [25]. If we want to divide the hypernode set V of a hypergraph H into k parts, we bisect V in V_1 and V_2 with part weight bounds $c(V_1) \leq (1+\epsilon)\lfloor\frac{k}{2}\rfloor\lceil\frac{c(V)}{k}\rceil$ and $c(V_2) \leq (1+\epsilon)\lceil\frac{k}{2}\rceil\lceil\frac{c(V)}{k}\rceil$. We further divide V_1 into $\lfloor\frac{k}{2}\rfloor$ and V_2 into $\lceil\frac{k}{2}\rceil$ parts. If we use on each bisection the initial imbalance ϵ for the weight bound, it can happen that one of our resulting blocks is imbalanced. Assume $k = 32$ and $\epsilon = 0.05$ and after each bisection, where we have to further divide a subhypergraph $H' = (V', E', c, \omega)$ resulting from bisections of the original hypergraph, the weight of block V_1 is equal to the maximum allowed part weight $\Rightarrow c(V_1) \approx \frac{c(V')}{2}(1+\varepsilon)$. V_1 produces at the end on the original hypergraph an imbalance of $(1+\epsilon)^{\log_2 k} = 1.05^5 \approx 1,27$. Most partitioning frameworks restrict the input ϵ at the beginning of the recursive bisection process. Schulz [25] uses 1% imbalance at each bisection step to reach his default imbalance value of 3%. *hMetis* [14] provides a parameter called *ubfactor*, where the user can specify the imbalance, which is then used at each bisection.

If we bisect a hypergraph we have to extract the two subhypergraphs $H_1 = (V_1, E_1, c, \omega)$ and $H_2 = (V_2, E_2, c, \omega)$. We define the resulting hyperedge sets E_1 and E_2 of a hypergraph $H = (V, E, c, \omega)$ and a bisection $P_2 = \{V_1, V_2\}$ as follows:

$$E_i = \{e \mid e \in E : \Lambda(e) = \{V_i\}\}$$

All nets $e \in E_i$ are internal hyperedges of part V_i and all cut nets are removed in the extraction step. Cut nets cannot be removed from the cut during future recursive bisections and therefore we remove it in the extraction step. If we use another metric, e.g. the $(k-1)$ metric, we also have to minimize the connectivity of a hyperedge. If we remove the cut edges we cannot control the connectivity of these nets anymore. *PaToH* [6] splits a cut edge e into two parts e' and e'', where e' contains all hypernodes from block V_1 and e'' from block V_2. Figure 9 illustrates the two extraction methods of a bipartitioned hypergraph.

The local search algorithms, which can be used after each bisection, have no global view on the problem. Therefore the final solution could be far away from the optimum [3, 4, 19]. A direct *k-way* partitioning method has a global view on the problem and therefore be able to compute potentially better results [19]. The possibility that a *local search* heuristic finds a positive move on the original hypergraph of recursive bisection is very low, because large nets are still part of the hypergraph and cut nets have a large number of pins in both parts of the bisection [3]. Also a direct *k-way* algorithm can work with tighter balance constraints and can explore a greater part of the solution space than a recursive bisection method [19]. Much research has been done to produce a good direct *k-way* partitioning algorithm, but all algorithms are inferior to the multilevel recursive bisection scheme [19].

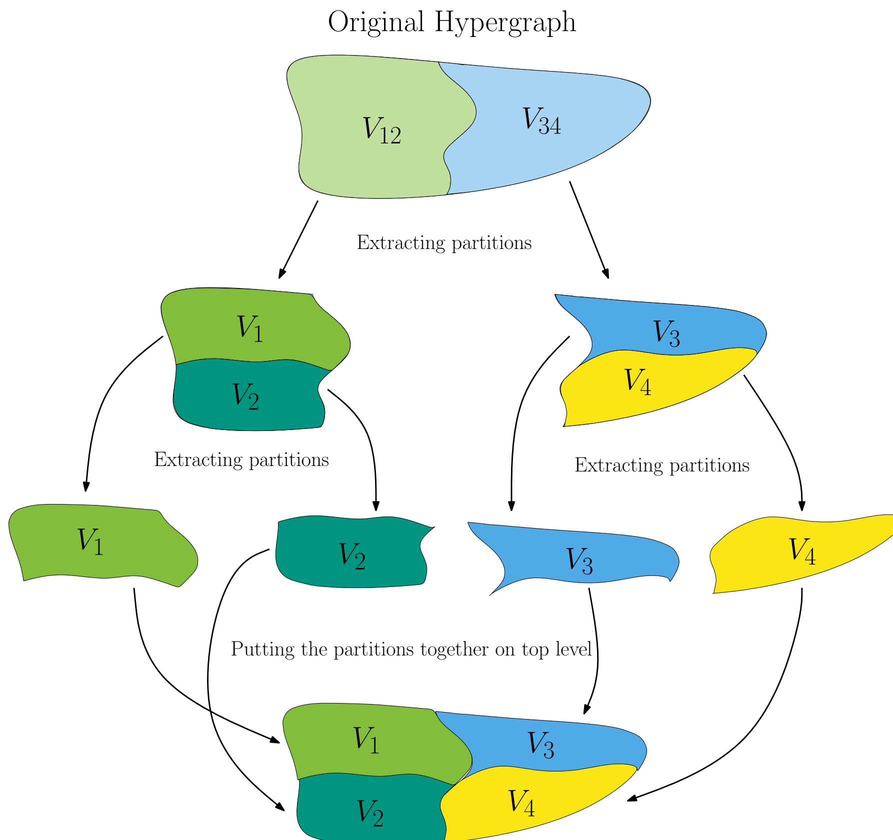


Figure 8: Example of Recursive Bisection. We want to partition a hypergraph into four parts. We divide the original hypergraph into two parts and recursively bisect the resulting parts until we have our four parts. At the end we put the pieces on the original hypergraph together.

3.4 Refinement

3.4.1 Kernighan-Lin

Kerninghan and Lin introduce one of the first local search algorithms for the partitioning problem [20]. We present the simplest variant of this heuristic where we assume that we have a graph $G = (V, E, \omega)$ with a edge weighting function ω and let $P = (V_1, V_2)$ be a perfectly balanced two-partition of the graph. Assume we have a perfectly balanced partition $P^* = (V_1^*, V_2^*)$ with a minimum cut. We can produce this partition with a sequence of interchanging of two nodes $s = ((a_1, b_1), ..., (a_k, b_k))$ from every arbitrary perfect balanced partition P.
Let us assume we have a perfectly balanced two partition $P = (A, B)$. The external costs of a node $a \in A$ is defined as $E_a = \sum_{v \in B} \omega(a, v)$ and the internal costs of a node $a \in A$ is defined as $I_a = \sum_{v \in A} \omega(a, v)$. The both definitions are used vice versa for all $b \in B$. Further difference between the external and internal costs is defined as $D_v = E_v - I_v \ \forall v \in V$.

Definition 3.1. Assume we want to interchange two nodes $a \in A$ and $b \in B$ of a perfectly balanced partition $P = (A, B)$. The reduction of the edge cut, is a function $g : V \times V \rightarrow \mathbb{Z}$ with $g(a, b) = D_a + D_b - 2\omega(a, b)$.

Now we are able to define an algorithm. First all D values of all nodes are calculated. Then we have to search two nodes $a \in A$ and $b \in B$ where $g(a, b)$ is maximized. Afterwards the exchange of a and b is performed and these two nodes are locked for further interchanges. The

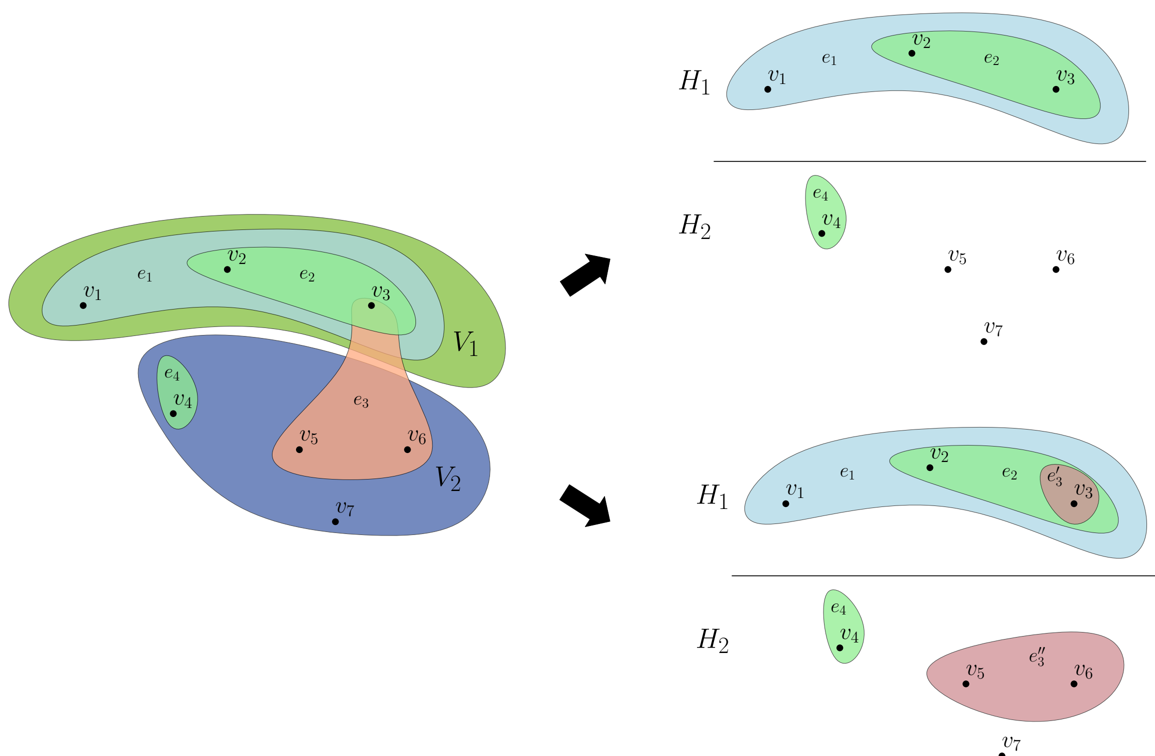


Figure 9: Example of extraction methods. On the top-right side we see the extracted subhypergraphs, when minimizing the cut metric. Hyperedge e_3 is a cut edge and can be removed in both resulting subhypergraphs. On the buttom-right side we see the extraction of the subhypergraphs, when the $(k-1)$ metric is optimized. Hyperedge e_3 is split into two nets e'_3 and e''_3. e'_3 is part of the subhypergraph H_1 and contains the nodes which are in e_3 and V_1. e''_3 is part of the subhypergraph H_2 and contains the nodes which are in e_3 and V_2.

gain $g_i = g(a, b)$ is saved in a interchange gain sequence s. After each interchange all D values have to be updated:

$$D'_x = D_x + 2\omega(x, a) - 2\omega(x, b) \quad \forall x \in A \setminus \{a\}$$

$$D'_y = D_y + 2\omega(y, b) - 2\omega(y, a) \quad \forall y \in B \setminus \{b\}$$

$\forall x \in A$ the edge (x, a) is an internal edge of block A. After the exchange (x, a) became an external edge and we have to add $\omega(x, a)$ to E_x and substract the same value from I_x. If we want to update D_x we have to add $2\omega(x, a)$. The consideration of subtracting $2\omega(x, b)$ from D_x are equivalent. This process is repeated until all nodes are locked. If all nodes are locked, we have to find a point $1 \leq k \leq |s|$ where $\sum_{i=1}^{k} g_i$ is maximized. The first k exchanges of node pairs are performed on our partition P and the whole process is repeated until no further improvement is possible. In algorithm 2 is a pseudocode of the described algorithm.
The algorithm has a running time of $\mathcal{O}(|V| \cdot |E| \log(|E|))$ in his initial version. The maximum gain pair (a, b) in line 8 of algorithm 2 can be calculated by searching for the maximum D_a and D_b value [20]. The resulting pair (a, b) of this linear scan is only possibly a maximum gain pair, but we are reducing the running time to $\mathcal{O}(|V|^2)$. We only have to update those D values in line 13 of algorithm 2, which are incident to a or b.

Algorithm 2: Kerninghan-Lin heuristic

```
Data: (Hyper)graph H = (V, E, ω) and perfect balanced partition P = (A, B)
Result: Improved partition P' = (A', B')
1  initialize all D values
2  A' ← A
3  B' ← B
4  do
5      X ← A'
6      Y ← B'
7      for i = 1 until |V|/2 do
8          (a^i, b^i) ← argmax_{a^i ∈ X, b^i ∈ Y} g(a^i, b^i)
9          g_i ← g(a^i, b^i)
10         X ← X \ {a^i} ∪ {b^i}
11         Y ← Y \ {b^i} ∪ {a^i}
12         lock(a^i, b^i)
13         update all D values
14     k ← argmax_{1 ≤ k ≤ |V|/2} Σ_{i=1}^{k} g_i
15     g ← Σ_{i=1}^{k} g_i
16     if g > 0 then
17         for i = 1 until k do
18             A' ← A' \ {a^i} ∪ {b^i}
19             B' ← B' \ {b^i} ∪ {a^i}
20 while g > 0
21 return P' = (A', B')
```

3.4.2 Fiduccia-Matheyses

The most prominent and used *local search* heuristic is the Fiduccia-Matheyses algorithm [11]. This algorithm is an adaption of the Kerninghan-Lin algorithm (see section 3.4.1). The algorithm works in its simplest variant for an arbitrary bipartition $P = (A, B)$ and with arbitrary balance constraints for each block of the partition.

The main differences between Fiduccia-Matheyses and Kerninghan-Lin is that only one hypernode is moved per pass from one block to another. Furthermore a special datastructure is used for finding the maximum gain value, and to update and remove gain values of hypernodes. Before we start to describe this algorithm, we describe the special datastructure which is called *bucket queue*. An example of this datastructure is shown in figure 10. The gain $g : V \rightarrow \mathbb{R}$ of a hypernode v is the reduction of the hyperedge cut, if we move this vertex from his current block to the other block of a bisection.

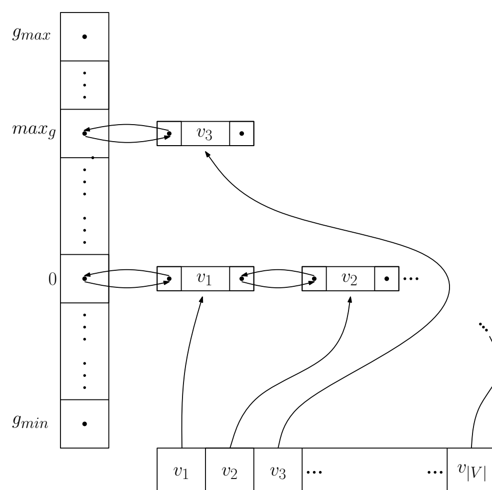


Figure 10: Example of the bucket queue datastructure.

If we want to find out the maximum/minimum possible gain of a move with the assumption that $\forall e \in E : \omega(e) = 1$, we have to search for a node $v_{max} = \text{argmax}_{v \in V}\ d(v)$ with maximum

degree. The maximum gain g_{max}, which can occur during a pass, is defined as $g_{max} = d(v_{max})$. This can occur, if all incident edges of v_{max} were removed from the cut after we move this node. Vice versa the minimum gain g_{min} is defined as $g_{min} = -g_{max}$. With this observation we can save the gain of a hypernode into an array with size $2g_{max}$ from $-g_{max}$ to g_{max}. An array entry i contains a double-linked list with all hypernodes with gain i. To find the hypernode with the maximum gain efficiently, we hold a pointer max_g to that entry in the datastructure. If this node is removed, we have to update the max_g pointer. To update and remove arbitrary hypernodes into the datastructure efficiently a mapping is stored from all hypernodes to the corresponding entry in the double-linked list. The gain of a hypernode can be updated in $\mathcal{O}(1)$, if the hypernode is accessed over our mapping and added to the new double-linked list introduced by the new gain. We only have to check, if the new gain is greater than max_g and in this case we have to update the max_g pointer.

Algorithm 3: Fiduccia-Matheyses heuristic

```
Data: (Hyper)graph H = (V, E, ω), ϵ and a bipartition P = (V_1, V_2) with
      V_i ≤ ⌈c(V)/2⌉(1 + ϵ)   i ∈ {0, 1}
Result: Improved partition P' = (V'_1, V'_2)
1  b_0 ← initialize with gain values from V_1, if we move those hypernodes from V_1 to V_2
2  b_1 ← initialize with gain values from V_2, if we move those hypernodes from V_2 to V_1
3  V'_1 ← V_1
4  V'_2 ← V_2
5  do
6      X_1 ← V'_1
7      X_2 ← V'_2
8      for i = 1 until |V| ∨ (b_0.empty() ∧ b_1.empty()) do
9          do
10             p_i ← argmax_{j∈{0,1}} b_{j,max}
11             g_i ← b_{p_i,max}
12             v_i ← b_{p_i,max}.node()
13             b_{p_i}.remove(v_i)
14         while c(V_p) + c(v_i) ≥ ⌈c(V)/2⌉(1 + ϵ)
15
16         X_{p_i} ← X_{p_i} \ {v_i}
17         X_{1−p_i} ← X_{1−p_i} ∪ {v_i}
18         lock(v_i)
19         update gain of b_0 and b_1
20     k ← argmax_{1≤k≤|V|} Σ_{i=1}^{k} g_i
21     g ← Σ_{i=1}^{k} g_i
22     if g > 0 then
23         for i = 1 until k do
24             V'_{p_i} ← V'_{p_i} \ {v_i}
25             V'_{1−p_i} ← V'_{1−p_i} ∪ {v_i}
26 while g > 0
27 return P' = (A', B')
```

The main algorithm requires as input a hypergraph H, an initial imbalance ε and a bipartition $P = (V_1, V_2)$, which fullfil the balance constraints. First the two bucket queues b_1 and b_2 have to be initialized, which both hold the gain values of hypernodes of the moves to part V_1 or V_2. Then the algorithm is very similiar to the Kerninghan-Lin algorithm. Except that we have to

take care, if a move with maximum gain should be performed, that the block weights are within the balance constraints after a move (line 9 - 14). If there are several nodes with maximum gain we choose one randomly. After a move the moved hypernode is locked and the gain values in b_1 and b_2 are updated. The second part of algorithm 3 (line 20 - 25) is similiar to the Kerninghan-Lin algorithm. We apply the moves which lead to the highest gain and repeat the process until we no further improve the quality of the solution. To provide fast updates of the gain values after a move of a hypernode v, only hypernodes $u \in \Gamma(v)$ have to be updated. If we want to further improve the updates we can use delta-gain updates (see section 3.4.3). Fiduccia and Matheyses [11] show that no more than four update operations have to be performed during the update process.

3.4.3 Delta-Gain Update

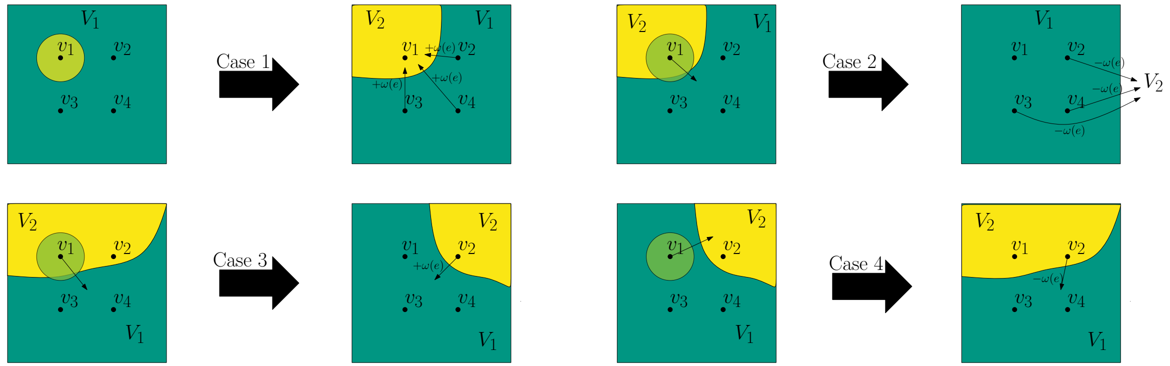


Figure 11: Four cases of delta gain updates.

As mentioned before we only have to update the gains of all incident pins $u \in \Gamma(v)$ of a moved hypernode v. For hypergraphs the gain of the pins of a net changes only in four cases [11,13]. In figure 11 the four cases are illustrated. We present a list of the four cases, where we describe the state of the net e before and after a move of a hypernode v and the resulting update operations:

(i) Before the move of hypernode $v \in e$ the net e has connectivity $\lambda(e) = 1$ and $\Lambda(e) = \{V_i\}$. The hyperedge e contributes $-\omega(e)$ to the gain of all its pins. If we move the hypernode v to block V_j the net e becomes a cut net. Therefore, if we move another pin $u \in e$ after this move from V_i to any block $\neq V_i$, the cut would not change. We have to add to all calculated gain values of all pins $p \in e \cap V_i$ the value $+\omega(e)$.

(ii) Before the move of hypernode $v \in e \cap V_j$ the net e has connectivity $\lambda(e) = 2$, $\Lambda(e) = \{V_i, V_j\}$ and $|V_j \cap e| = 1$. Moving v to V_i removes e from the cut. Before the move all moves of pins $u \in e \cap V_i$ to any block $\neq V_i$ were zero gain moves, if we only consider net e. After the move all moves of pins $u \in e \cap V_i$ to any block $\neq V_i$ would make e a cut net again, if we move it. Therefore we have to add $-\omega(e)$ to all calculated gain values of moves to any block $\neq V_i$.

(iii) Before the move of hypernode $v \in e \cap V_j$ the net e has connectivity $\lambda(e) = 2$, $\Lambda(e) = \{V_i, V_j\}$ and $|V_j \cap e| = 2$. The move of v is a zero gain move, but only one pin of e is still left in V_j. Moving this pin to V_i would remove e from the cut. Therefore we have to add $+\omega(e)$ to the calculated gain of the remaining pin $p \in V_j$ for moving it to V_i.

(iv) Before the move of hypernode $v \in e \cap V_i$ the net e has connectivity $\lambda(e) = 2$, $\Lambda(e) = \{V_i, V_j\}$ and $|V_j \cap e| = 1$. The move of v from V_i to V_j is a zero gain move. Before the move the remaining pin $p \in V_j$ would remove e from the cut and after the move of hypernode

v the net e has two pins in V_j. Therefore we have to add $-\omega(e)$ to the calculated gain of pin p for moving it to V_i.

3.5 nLevel-Partitioning

The n-level partitioning scheme is an extreme version to the multilevel partitioning scheme. There still all three phases of the multilevel paradigm: coarsening, initial partitioning and refinement phase. The difference is that the n-level scheme only contract two nodes at each level [22]. Furthermore Osipov [22] outlined that the n-level hierarchy leads to quadratic space consumption and the contraction of two nodes instead of finding a heavy matching leads to a non-uniform distribution of node weights. Osipov and Sanders [22] solve these two problems by providing a dynamic graph datastructure and taking the node weight into account in a rating function for the contraction of two nodes, which avoids very heavy nodes during contraction.
Schlag [13] adapts the n-level concept to hypergraph partitioning. In the coarsening phase the hypernode pairs are rated with following function:

$$r(u,v) := \frac{1}{c(u) \cdot c(v)} \sum_{e \in (I(u) \cap I(v))} \frac{\omega(e)}{|e|-1} \tag{3.11}$$

In [13] three different coarsening strategies are presented, which only differ in the way they re-rating the adjacent vertices of a contracted hypernode pair (u, v). For each hypernode v a rating to all adjacent pins $u \in \Gamma(v)$ is calculated and the pair (u, v) with the highest rating is stored in a priority queue. After all ratings are calculated the hypernode-pair (u, v) with the highest rating is contracted. The *full*-variant recalculate all ratings of all adjacent vertices after a contraction. Since the re-rating can become very expensive, there exists two different approaches to update the ratings. In the first method only the ratings of neighbours $w \in \Gamma(u) \cup \Gamma(v)$ of a contraction pair (u, v) are updated, which chooses either v or u as contraction partner in the initial rating phase. This method is called *partial*. In the second variant the ratings are not updated immediately. The adjacent vertices are invalided. If a contraction pair (u, v) in the following is chosen and either u or v is *invalid*, the rating is re-calculated and the priority queue is updated. This method is called *lazy*.

3.6 State-of-the-art Hypergraph Partitioning Tools

3.6.1 hMetis

hMetis was developed by Karypis and Kumar [16] and it is one of the state-of-the-art hypergraph partitioning tools. *hMetis* is optimized for partitioning VLSI instances. It uses the multilevel paradigm.
The coarsening strategies are EC, HEC and FC, which are described in section 3.2. As initial partitioning they perform random, hypergraph growing partitioning in a breadth-first-fashion and greedy hypergraph growing. They mentioned that the initial partitioning with smallest cut not necessarily leads to the smallest cut after the *uncontraction* phase [15]. They use i initial partitionings in the *refinement* phase and propagate on each level the best $x\%$ to the next level finer hypergraph. In this phase of the multilevel paradigm they use the classical *FM local search* algorithm with some adaptions. Only k moves are performed and only two passes [15]. Another refinement algorithm is the *Hyperedge Refinement* algorithm (HER) [16]. This algorithm moves groups of vertices with the goal to remove a whole net from the cut.
Two modes of multilevel partitioning are provided. *hMetis* can perform direct *k-way* and

recursive bisection multilevel partitioning. In the recursive bisection mode the multilevel process is performed on each bisection.

3.6.2 PaToH

PaToH is developed by Catalyurek and Aykanat [6] and it is one of the state-of-the-art hypergraph partitioning tools, too. PaToH is optimized for partitioning sparse matrix instances and uses the multilevel paradigm.
In the coarsening phase PaToH uses HCM and HCC, which are described in section 3.2. In the initial partitioning phase they use eleven different initial partitioning methods which can be categories into random, hypergraph growing and greedy hypergraph growing variants. For a closer overview on all different variants, we refer to the PaToH manual [5]. PaToH also have a parameter for the number of different initial partitionings which should be constructed for the refinement phase [5]. In the refinement phase PaToH uses Boundary FM (BFM). BFM only looks at boundary hypernodes and moves are only performed from an over-loaded to an under-loaded partition [6]. The refinement stops if no feasible move remains or $\max(50, 0.001|V|)$ moves with no decrease into the cut are performed [6].

4 Initial Partitioning

The goal of this work is to implement and evaluate different initial partitioning algorithms for the n-level hypergraph partitioning framework *KaHyPar*. For more information about this framework we refer the reader to [13]. This framework relies upon *hMetis* to produce an initial partition. With this work we try to develop an initial partitioning method which produces equal or better cuts than *hMetis* on the coarsened hypergraph instances in the same amount of time. After we integrate our partitioner in *KaHyPar*, we want to archieve better cuts as with *hMetis* as initial partitioner.
In the first phase of this work we implement several of the described algorithms in Section 3.3. We adapt these algorithms to direct *k-way* initial partitioners and evaluate the various implementations. We implement a random, breadth-first-search, greedy hypergraph growing and label propagation initial partitioner, which are explained in Section 4.1. In Section 4.1.3 we describe several implementations of greedy hypergraph growing and different gain functions. Afterwards the algorithms are embedded into a recursive bisection framework, which we describe in Section 4.2. This framework can handle an arbitrary number of blocks. We can adapt before each bisection the initial imbalance ε, such that the final partitions are guaranteed to satisfy the imbalance. This technique allows us to explore a greater solution space during recursive bisection than other existing implementations.
Finally, the recursive bisection implementation and the existing n-level hypergraph partitioning framework *KaHyPar* is used to build a recursive bisection n-level initial partitioner in Section 4.3. In every bisection step, we coarsen the hypergraph, then we choose an initial partitioning algorithm to bisect the hypergraph and then we uncoarsen the contracted node and use a *local search* heuristics to improve the quality.

4.1 Direct k-Way Initial Partitioning

4.1.1 Random

The simplest algorithm to produce a *k-way* partition is to randomly assign the hypernodes to a block of a partition. This algorithm is used as a baseline for all other partitioning methods. Other methods should normally produce partitions with better quality. But to assign all hypernodes in expected $\mathcal{O}(|V|)$ running time has the advantage that it is possible to run this algorithm more often than other variants in the same amount of time.
Algorithm 4 shows the pseudocode of our random initial partitioner. This partitioner produces a *k-way* partition $P = (V_1, ..., V_k)$ of a hypergraph $H = (V, E, c, \omega)$ and tries to ensure that all weights of all blocks V_i are below $w_{max} = \lceil \frac{c(V)}{k} \rceil (1 + \varepsilon)$ for a given ε. We visit all nodes of the hypergraph and for all nodes we randomly choose a block V_p. If the resulting weight $c(V_i \cup \{v\}) \leq w_{max}$, we assign hypernode v to part V_p, otherwise we choose another random block V_p and try to assign it again. If there is no part, such that the resulting block weight $c(V_i \cup \{v\})$ for an arbitrary $i \in \{1, ..., k\}$ is smaller than w_{max} we assign v to the last calculated random partition V_p. Note, if we reach this state the resulting partition P is not ε-balanced. In most cases we can satisfy the balancing constraint, but if the graph is small and the weights of the hypernodes are large, it may occur that there are few nodes left at the end which could not be assigned to a part.
On the first view the algorithm has a worst case running time of $\mathcal{O}(k|V|)$. This is because it may occur that we have to try several parts to assign a hypernode v. Because of its simplicity and the fact that the worst case only occurs at the end of the partitioning process, the algorithm

runs fast in practice. The idea to assign the hypernodes random to a block seem to be not a promising method, but if the hypergraph is small enough, we can try to run this algorithm multiple times and use the best result.

Algorithm 4: Random-based partitioning algorithm

```
Data: Hypergraph H = (V, E, c, ω), ε, k
Result: k-way partitioning P = (V_1, ..., V_k)
1  w_max ← ⌈c(V)/k⌉(1 + ε)
2  P ← (∅, ..., ∅)    // |P| = k
3  for v ∈ V do
4      tryAssignmentToPart ← (0, ..., 0)    // |tryAssignmentToPart| = k
5      sum ← 0
6      p ← −1
7      do
           // In this loop we have to search for a valid block p for hypernode v
8          if p ≠ −1 ∧ !tryAssignmentToPart[p] then
9              tryAssignmentToPart[p] ← 1
10             sum ← sum + p
11             if sum = k(k+1)/2 then
                   // All blocks has been discovered an there is no block to assign v such that
                      resulting block weight is smaller than w_max
12                 break do-while loop
13         p ← choose random p ∈ {1, ..., k}
14     while c(V_i ∪ {v}) ≤ w_max
15     V_p ← V_p ∪ {v}
16 return P = (V_1, ..., V_k)
```

4.1.2 Breadth-First-Search

The next algorithm we present is based on the concept of hypergraph growing partitioning as described in Section 3.3.2. To create a bisection with this variant, we randomly choose a hypernode and perform a breath-first-search (BFS) from that vertex until half of the hypergraph is discovered. The vertices touched by our search constitutes part V_1 and all untouched V_2. In this section we extend the hypergraph growing partitioning algorithm to produce direct *k-way* partition.

First we describe an optimization of the BFS traversal on hypergraphs. Normally, in graphs, we select a start node and push it into a queue. Afterwards we pop a node v from the queue. Then we iterate over all adjacent nodes of v and push all unmarked nodes into the queue. If we push a node into the queue, we mark it as *in_queue*. We repeat this procedure until the queue is empty. This algorithm has a running time of $\mathcal{O}(|V| + |E|)$. Appling the same algorithm to hypergraphs can result in a running time of $\mathcal{O}(|V| + |E|(\max_{e \in E} |e|)^2)$. This is caused by the fact that a hyperedge contains more than two nodes. Assume we push the pins of a hyperedge e into the queue and mark them as *in queue*. This has the effect that every time we pop an element of that net e from the queue we again iterate over all pins, but all of them are already marked and nothing happens. If we want to prevent that case we have to mark all hyperedges as *in queue* as well. If we visit the first pin of a net e we have to consider all pins of e and push them into the queue, if they are not marked already. If we pop the next hypernode from the

queue of net e, we have not to iterate over all pins in e again. Based on this observation the first pin of net e contributes $\mathcal{O}(|e|)$ and the remaining $\mathcal{O}(1)$ to the overall running time, if we pop them from the queue. Every edge contributes $|e| + \sum_{i=1}^{|e|-1} 1 = 2|e| - 1$ to the overall running time. The running time of the BFS traversal in hypergraphs with the adaption described above can be estimated as follows:

$$\sum_{e \in E} 2|e| - 1 \leq |E|(2 \max_{e \in E} |e| - 1) \leq 2(\max_{e \in E} |e|)|E| = \mathcal{O}((\max_{e \in E} |e|)|E|)$$

Additionally, we have to initialize the datastructures for marking hypernodes and hyperedges. This can be done in $\mathcal{O}(|V| + |E|)$. This leads to an overall running time of $\mathcal{O}(|V| + (\max_{e \in E} |e|)|E|)$.

Our direct *k-way* hypergraph growing partitioning algorithm is shown in Algorithm 5. At the beginning of the algorithm we initialize k queues for the BFS traversal. Queue i contains the hypernodes, which are adjacent to the growing block V_i. To start with the hypergraph growing algorithm we have to choose k different start vertices. As described in Section 3.3.2 we can choose those hypernodes by searching for pseudo peripheral hypernodes. This is done by choosing a random start node and perform a BFS traversal from that node. The last hypernode touched by this search is taken as the second start node. We repeat this procedure with both start vertices as start nodes for the next BFS to find the third start hypernode. This procedure is repeated until we have k different start hypernodes. Start vertex s_i is inserted into queue q_i to grow block V_i.

The assignment process starts in line 6. We grow the blocks in a *round-robin*-fashion until there are no further hypernodes left to be assigned. In each round, we assign one additional hypernode v to a part V_i. After the assignment of vertex v, block V_{i+1} can choose its next hypernode. If we reach part V_k we start this process again from $i = 1$ until all hypernodes are assigned. A part V_i can choose its new hypernode from its corresponding queue q_i. Note that during the assignment process the blocks of the partition can grow different fast, because the weights of the nodes are not uniform. For this reason we only assign hypernodes to blocks, which are enabled. A part can become disable in two cases. In the first case, if there are no further hypernodes in its corresponding queue and all hypernodes are assigned to a part. In the second case, if we found a hypernode v, but the resulting part weight of V_i would become to heavy after the assignment of v to V_i. The assignment process of a hypernode v to a part V_i works as follows:

(i) Choose an unassigned hypernode v from q_i (line 13).

(ii) If there is no unassigned hypernode $v \in q_i$, we randomly search for an unassigned hypernode (line 18).

(iii) If we found a hypernode v in step (i) or (ii), we check if the resulting weight $c(V_i \cup \{v\})$ is less to or equal than $\lceil \frac{c(V)}{k} \rceil (1 + \varepsilon)$.

(iv) If the resulting weight of $V_i \cup \{v\}$ fulfils our balance constraints, we push all adjacent neighbors $u \in \Gamma(v)$, which are not marked *in_queue* or already assigned to another part, into q_i and assign hypernode v to V_i.

Note, if we push all adjacent hypernodes into the queue after the assignment of v, we use all the optimizations described at the beginning of this section (in line 21 - 27). To mark the hypernodes and hyperedges, we use k different bitsets, one for each q_i. The running time of this algorithm is $\mathcal{O}(k(|V| + (\max_{e \in E} |e|)|E|))$, because we perform k different breadth-first-searches and each has a running time of $\mathcal{O}(|V| + (\max_{e \in E} |e|)|E|)$.

We are also able to assign all hypernodes to a block V_i before initial partitioning. A BFS traversal then only operates on those hypernodes of block V_i. Note that if we assign all hypernodes to a part $i \neq -1$, we only perform $k - 1$ breath-first-searches.

Algorithm 5: BFS-based partitioning algorithm

```
Data: Hypergraph H = (V, E, c, ω), ε, k
Result: k-way partitioning P = (V_1, ..., V_k)
1  w_max ← ⌈c(V)/k⌉(1 + ε), P ← (∅, ..., ∅)    // |P| = k
2  markHypernodes ← {{0,..,0},..,{0,..,0}}    // |markHypernodes| = k, ∀i ∈ {1,..,k}|markHypernodes[i]| = |V|
3  markHyperedges ← {{0,..,0},..,{0,..,0}}    // |markHyperedges| = k, ∀i ∈ {1,..,k}|markHyperedges[i]| = |E|
4  startNodes ← calculateStartNodes(H, k)
5  Q ← (q[k] = {startNodes[1]}, ..., q[k] = {startNodes[k]})    // Array of k different queues
6
7  while there are unassigned nodes do
8      for i = 1...k do
9          if V_i is enabled then
10             v ←⊥
11             if q[i] is not empty then
                   // searching for a valid hypernode v which is not assigned to a any part
12                 do
13                     v ← q[i].front(),   q[i].pop()
14                 while v is already assigned ∧ q[i] is not empty
15
16         if v = invalid_hypernode ∨ v is already assigned then
               // We have to choose a new random unassigned hypernode, because the queue q_i is
                  empty or no unassigned hypernode was contained into q_i.
17             v ← search for a random unassigned hypernode
18
19         if v ≠ invalid_hypernode then
20             if c(V_i ∪ {v}) ≤ w_max then
21                 V_i ← V_i ∪ {v}
22                 for e ∈ I(v) do
23                     if !markHyperedges[i][e] then
24                         for u ∈ e do
25                             if !markHypernodes[i][u] then
26                                 q[i].push(u)
27                                 markHypernode[i][u] ← 1
28                         markHyperedge[i][e] ← 1
29             else
30                 if q_i is empty then
31                     disable V_i
32         else
33             disable V_i
34 return P = (V_1, ..., V_k)
```

4.1.3 Greedy Hypergraph Growing

Greedy hypergraph growing partitioning (GHGP) is an extension of the hypergraph growing algorithm and described in Section 3.3.3. The difference to HGP is that we choose the next

vertex which should be assigned to a block V_i according to a gain function. In this Section we present our direct *k-way* greedy hypergraph growing algorithms. We provide three different implementations of GHGP and three different gain functions, which all provides their own *delta gain updates* (see Sections 3.3.3 and 3.4.3). If we combine all variants we have nine different greedy hypergraph growing algorithms.

Gain functions

The first gain function is the classical *FM* variant and the other two functions are the *Max-Pin* and *Max-Net* variants used by PaToH [5]. A gain function is mapping $g : V \times P \times P \rightarrow \mathbb{R}$. This mapping assigns each move of a hypernode $v \in V$ from block $V_{from} \in P$ to $V_{to} \in P$ a gain value. To store gain values we use k different prority queues $PQ = \{pq_1, ..., pq_k\}$. If we insert a hypernode-gain pair (v, g) into priority queue pq_i, it means that the move of hypernode v to part V_i has gain g.

FM gain function

The gain of a hypernode v is the decrease of the hyperedge cut, if we move it from its current block V_i to V_j with $i \neq j$. If the hyperedge cut e.g. is 100 and we move a hypernode to another part, which results in a new cut of 97, the *FM* gain is +3.
As mentioned at the end of Section 4.1.2, we are able to assign all hypernodes to a block V_i before initial partitioning and use the vertices from that block to grow a partition. If we have a look at an unpartitioned hypergraph H, all hypernodes have default part -1. We call all hypernodes *unassigned*. This leads to the fact that the connectivity of all hyperedges e is $\lambda(e) = 0$ in the beginning. If we assign all hypernodes to an arbitrary block V_i, the connectivity of all hyperedges e changes to $\lambda(e) = 1$. Figure 12 illustrates two different behaviors of the FM gain function in the same situation, if we use the vertices from block V_2 to grow a partition P (left figure) and if all hypernodes are unassigned before initial partitioning (right figure). In the left situation the connectivity of the hyperedges e_1, e_2 and e_3 is $\lambda(e_1) = \lambda(e_2) = 2$ and $\lambda(e_3) = 1$. Moving v_3 from partition V_2 to V_1 removes e_1 and e_2 from the cut, but makes e_3 a cut edge $\Rightarrow g(v, V_2, V_1) = 1$. If we look at the same situation in the right figure, hyperedge e_1, e_2 and e_3 has connectivity $\lambda(e_1) = \lambda(e_2) = 1$ and $\lambda(e_3) = 0$. Moving v_3 from -1 to V_1 would only change the connectivity of e_3 to $\lambda(e_3) = 1$. No hyperedge becomes a cut edge and no net is removed from the cut $\Rightarrow g(v_3, V_2, V_1) = 0$. If we did not assign all hypernodes to a block before initial partitioning, we can only increase the connectivity of a hyperedge during initial partitioning $\Rightarrow \forall v \in V : \forall i, j \in \{1, ..., k\} : g(v, V_i, V_j) \leq 0$. To remove a net from the cut, we have to decrease the connectivity of a hyperedge. Keep in mind the two different behaviors of the *FM* gain function.
The calculation of the FM gain of a hypernode v, which should be moved from his current part V_{from} to V_{to} is list in Algorithm 6. To calculate the decrease in the cut we have to look at each net e incident to v and either decide if the move removes hyperedge e from the cut or makes it a cut net. A net e becomes a cut hyperedge, if the connectivity is $\lambda(e) = 1$ and there are no pins in the destination block V_{to} of the move. Note, if we move an unassigned hypernode, it can happen that $\Lambda(e) = \{V_{to}\}$ and this move would not increase the connectivity of e. Therefore we have to check, if $V_{to} \cap e = \emptyset$. Only in this case the connectivity of a hyperedge e increases and net e becomes a cut hyperedge after the move. If the connectivity of net e is not $\lambda(e) = 1$, we have to check if the move removes hyperedge e from the cut. If we did not assign all hypernodes before initial partitioning to an arbitrary block, we can never decrease the cut with a move,

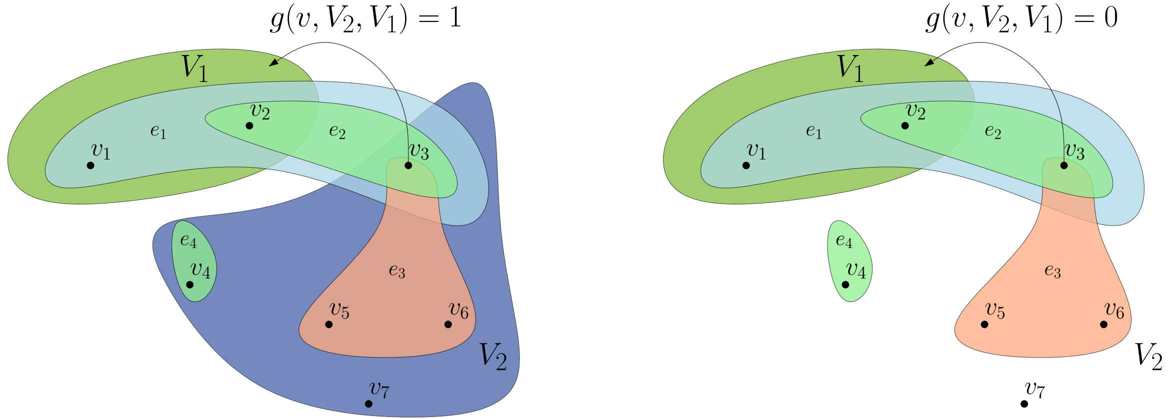


Figure 12: Main problem of FM gain function during initial partitioning. Left we assign all hypernodes to block V_2 before initial partitioning and in the right all hypernodes are left unassigned.

because all moves from source part -1 never decrease the connectivity of a hyperedge. In this case we skip the resulting check for a cut decrease. The precondition for a cut decrease is that the connectivity of hyperedge e is $\lambda(e) = 2$. If part V_{from} only contains hypernode v and V_{to} contains all remaining pins of hyperedge e, the move would decrease the connectivity of e and remove it from cut. The gain for moving a hypernode v can be calculated in $\mathcal{O}(|I(v)|)$, because our hypergraph datastructures provides constant time operations for all necessary operations. After a move of a hypernode v we have to update the gain values of all neighbors $u \in \Gamma(v)$. Therefore we implement delta gain updates as described in Section 3.4.3. If we move only unassigned hypernodes, there are two special cases, which are illustrated in Figure 13. If all hypernodes of a hyperedge are unassigned, all moves to an arbitrary block are zero gain moves. If we assign the first hypernode v at this net e to part V_i, we have to decrease the gain of all moves of a hypernode $u \in e \setminus \{v\}$ to a part V_j with $i \neq j$ by the value of $\omega(e)$. These moves would increase the connectivity of e to 2 and make it a cut net. If net e has connectivity $\lambda(e) = 1$ with $\Lambda(e) = \{V_i\}$ and we move an unassigned hypernode v to V_j with $i \neq j$, e becomes a cut net. Therefore we have to increase the gains of all calculated moves of a hypernode $u \in e \setminus V_i$ to a block $\neq V_i$ by the value of $\omega(e)$. Before we move hypernode v to V_j, all moves of unassigned hypernodes $u \in e$ to a block $\neq V_i$ would make e a cut net. After the move e is a cut net and all moves of unassigned hypernode did not increase the cut anymore.

Algorithm 6: FM Gain Calculation

```
Data: Hypergraph H = (V, E, c, ω), Hypernode v, Partition V_from and V_to
Result: Gain g, for moving hypernode v from block V_from to V_to
1 for e ∈ I(v) do
2     if λ(e) = 1 ∧ V_to ∩ e = ∅ then
3         g ← g − ω(e)
4     else if λ(e) = 2 ∧ unassigned_part ≠ −1 then
5         if |V_from ∩ e| = 1 ∧ |V_to ∩ e| = |e| − 1 then
6             g ← g + ω(e)
7 return g
```

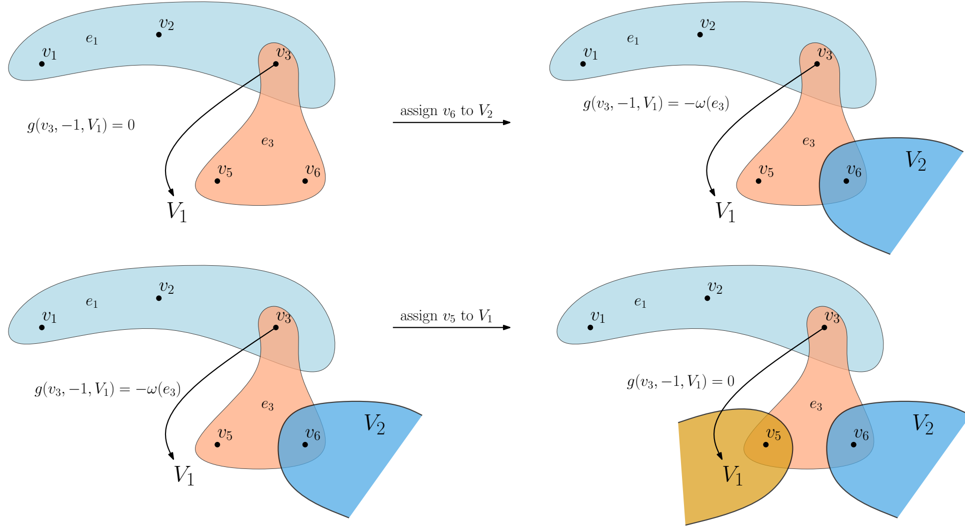


Figure 13: Two additional delta gain update cases, if we move only unassigned hypernodes.

Max-Pin gain function

The next gain function we implement is the *Max-Pin* gain function used by PaToH [5]. The gain of a move of a hypernode v from part V_{from} to V_{to} is calculated by counting the number of pins $p \in \Gamma(v)$, which are already assigned to V_{to}. This gain function prefers moves of hypernodes with a large number of pins adjacent to that vertex, which are assigned to V_{to}. Assume we have a net e of size $|e| = 20$, $|V_{from} \cap e| = 3$ and $|V_{to} \cap e| = 17$. The *FM* gain function only takes a move from V_{from} to V_{to} into account, if $|V_{from} \cap e| = 1$ and all remaining pins are in V_{to}. In the example above all moves from V_{from} to V_{to} are zero gain moves, if we only consider net e. The *Max-Pin* gain function signalizes a large number of pins in V_{to} for each move to that part for a hypernode $v \in V_{from} \Rightarrow \forall v \in V_{from} : g(v, V_{from}, V_{to}) = 17$. With this function we are working towards to remove e from the cut. An example of that variant is shown in figure 14.

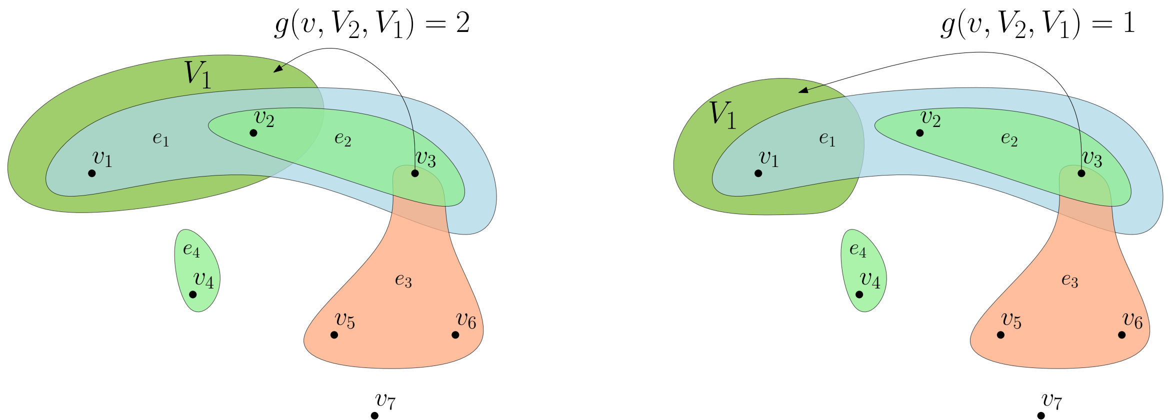


Figure 14: Example of the *Max-Pin* gain function (left figure) and *Max-Net* gain function (right figure).

To improve the running time of this variant we also provide delta gain updates. If we move a hypernode v from V_{from} to V_{to}, all calculated gains for moving hypernodes $u \in \Gamma(v)$ to part V_{from} are decremented and to part V_{to} are incremented. Before moving hypernode v, we count it for each calculated gain of moving a hypernode $u \in \Gamma(v)$ to block V_{from}. After we move

hypernode v to V_{to} the gains of all moves of hypernodes $u \in \Gamma(v) \cap V_{from}$ decrease, because v is part of V_{to} now. Otherwise the gains of moves of hypernodes $u \in \Gamma(v)$ to V_{to} increase. The pseudocode of *Max-Pin* delta gain updates is listed in algorithm 7.

Algorithm 7: *Max-Pin* delta gain update

```
Data: (Hyper)graph H = (V, E, c, ω), k Priority-Queues Q = {pq_1, ..., pq_k}, Hypernode v,
      Partition V_from and V_to
1 for u ∈ Γ(v) do
2     if u ∈ pq_to then
3         pq_to.u ← pq_to.u + 1
4     if u ∈ pq_from then
5         pq_from.u ← pq_from.u − 1
```

Max-Net gain function

The last gain function we introduce is the *Max-Net* gain function [5]. The gain of a move of a hypernode v from block V_{from} to V_{to} is defined as the number of hyperedges $e \in I(v)$ where $V_{to} \in \Lambda(e)$. To calculate the gain, we iterate over all nets e incident to hypernode v and check if $|e \cap V_{to}| \geq 1$. We increment a counter for every edge, where the condition is *true* and return this counter as our gain value. The intentioned idea is that we try to remove potentially as many nets from the cut as possible. The next example outline one disadvantage of this function. Assume a situation, where we have a net e with $\forall e' \in E \setminus \{e\} : e \cap e' = \emptyset$. Net e has connectivity $\lambda(e) = 2$ and only one pin p in block V_i. Moving p to the other part $V_j \in \Lambda(e) \setminus \{V_i\}$ would remove e from the cut. The *Max-Net* gain is $g(p, V_i, V_j) = 1$, because p is only incident to e. But 1 is a very low gain for this function, because the *Max-Net* gain is for all moves of hypernodes greater than zero. Therefore the move of p would potentially not perform, although it would remove e from the cut.

Algorithm 8: *Max-Net* delta gain update

```
Data: (Hyper)graph H = (V, E, c, ω), k Priority-Queues Q = {pq_1, ..., pq_k}, Hypernode v,
      Partition V_from and V_to
1 for e ∈ I(v) do
2     if |e ∩ V_from| = 0 ∨ |e ∩ V_to| = 1 then
3         for u ∈ e do
4             if |e ∩ V_from| = 0 ∧ u ∈ pq_from then
5                 pq_from.u ← pq_from.u − 1
6             if |e ∩ V_to| = 1 ∧ u ∈ pq_to then
7                 pq_to.u ← pq_to.u + 1
```

We also provide delta gain updates for this variant. The gain of a hypernode changes, if the connectivity of an incident net changes. There are two cases a move of a hypernode v from V_{from} to V_{to} can modify the connectivity of a hyperedge $e \in I(v)$. If v is the only remaining hypernode in $e \cap V_{from}$, the move decreases the connectivity of e. In this case, we decrement all calculated gain values of a hypernode $u \in e$ to part V_{from}. If the hypernode v is the first vertex in V_{to} after the move, we have to increment the calculated gains of all hypernodes $u \in e$ to part V_{to} by the value of one. Note, if we move only unassigned hypernodes, we can never decrease the connectivity of a net and the gain values are monotonically increasing. The pseudocode for *Max-Net* delta gain updates is given in Algorithm 8.

Implementaions

We have developed three direct k-*Way* partitioning algorithms for greedy hypergraph growing partitiong (GHGP). In this section we present all three methods and outline their differences and similarities.
Each variant uses k different priority queues to store the gain for moving a hypernode to an other block. If a hypernode v is inserted into a priority queue $pq[i]$ with gain g_v it means that moving hypernode v from its current part to V_i has gain g_v. The priority queue provides logarithmic time operations for inserting, removing and updating elements. For every greedy hypergraph growing partitioner we need to choose k start nodes $S = \{s_1, ..., s_k\}$, one for each block. We search again for pseudo peripheral hypernodes as described in Section 4.1.2 and 3.3.2. We insert start node s_i with its corresponding gain g_{s_i} into pq_i. Start hypernode s_i is used to grow block V_i.

Sequential Greedy Hypergraph Growing

Algorithm 9: Sequential Greedy Hypergraph Growing Algorithm

```
Data: Hypergraph H = (V, E, c, ω), ε, k
Result: k-way partition P = (V_1, ..., V_k)
1  P ← (∅, ..., ∅)    // P.size() = k
2  PQ ← (pq_1, ..., pq_k)    // |PQ| = k
3  S ← calculateStartNodes(H, k)
4  for i from 1 until k do
5  |   g_{s_i} ← calculateGain(H, partOf(s_i), V_i)
6  |   pq_i.insert((s_i, g_{s_i}))
7  ε' ← 0 // Relaxing the input ε to avoid empty partitions
8  w_max ← ⌈c(V)/k⌉(1 + ε')
9  for i = 1...k do
10 |   v ← pq_i.max()
11 |   while c(V_i ∪ v) ≤ w_max do
12 |   |   V_i ← V_i ∪ {v}
13 |   |   update all gains in all pq_i of all hypernodes u ∈ Γ(v) and insert all u ∉ pq_i into pq_i
   |   |       with their corresponding gain // see algorithm 10
14 |   |   if pq_i is empty then
   |   |   |   // If pq_i is empty and we haven't reached the partition bound w_max we have to
   |   |   |       choose a new start node.
15 |   |   |   u ← search for a random unassigned hypernode
16 |   |   |   pq_i.insert((u, calculateGain(H, partOf(u), V_i)))
17 |   |   v ← pq_i.max()
18 w_max ← ⌈c(V)/k⌉(1 + ε)
19 assign all remaining unassigned hypernodes
20
21 return P = (V_1, ..., V_k)
```

The first initial partitioner is called *sequential* GHG partitioner, because we grow one block after the other. Algorithm 9 shows the pseudocode of this variant. First, like in all GHGP variants, we initialize the k priority queues and calculate k start nodes. At the beginning we set

our initial imbalance ε to $\varepsilon' := 0$ and use it for the maximum allowed part weight bound w_{max}. This avoids empty blocks at end of the partitioning process. This case occurs, if the initial imbalance ε is large and the maximum weight $w_{max} := \lceil \frac{c(V)}{k} \rceil (1+\varepsilon)$ is assigned to each part. If we finally want to grow the last blocks, it is possible that the weight of all remaining unassigned hypernodes is smaller than w_{max} and if this case occurs at a block V_{k-i} for an $i \in \{0, ..., k-1\}$, the following $k-i+1$ parts are empty. In the main partitioning process we iterate over all parts and grow block V_i until we found a hypernode v where $c(V_i \cup \{v\}) > w_{max}$. We choose the next hypernode v, which should be assigned to part V_i, according to the maximum gain in pq_i. If the vertex v can be assigned to V_i, we remove it from all priority queues, assign it to V_i and perform the corresponding delta gain updates for all neighbours of v that are already into a priority queue. After this we have to insert all $u \in \Gamma(v)$, which are not already inserted into pq_i with their corresponding gain for moving u to V_i. The implementation of the insertion of the new gain values and update of all calculated gains after a move is shown in algorithm 10. Note that in this GHGP variant delta gain updates are very fast, because only the active pq_i of block V_i contains most hypernodes. All pq_j with $j > i$ contains only the start node for part V_j. All pq_l with $l < i$ only contain the hypernodes which could not be assigned to part V_l, because the resulting weight of this part would become greater than w_{max}. If the current pq_i is empty after delta gain updates and insertion of all unassigned hypernodes $u \in \Gamma(v) \land u \notin pq_i$, we have no further hypernodes to assign to part V_i. It is possible that V_i has a weight less than our partition weight bound w_{max}. In this case we randomly choose a new unassigned start node and continue with the assignment process. Growing the blocks sequentially can leave some hypernodes unassigned at the end of the partitioning process, because we redefine our ε to $\varepsilon' = 0$. Therefore the sum of all part weights $\sum_{V' \in P} c(V')$ is less or equal than $c(V)$ after we grow the last block. At the end we use again the upper bound $w_{max} := \lceil \frac{c(V)}{k} \rceil (1+\varepsilon)$ with initial imbalance ε and assign the remaining hypernodes in all priority queues pq_i to their corresponding part V_i. To assign all remaining hypernodes, we choose the hypernode from a priority queue, where the gain for moving it to a part V_i is maximized. If there are still unassigned hypernodes left after assigning all remaining hypernodes $v \in pq_i$ with $i \in \{1, ..., k\}$, we visit them in random order and assign hypernode v to part V_i, where the gain is maximized. This behaviour is summarized in line 25 and 26.

Algorithm 10: Insertion, Deletion and Update of the new gain values after a move

```
Data: (Hyper)graph H = (V, E, c, ω), Priority-Queues Q = {pq_1, ..., pq_k}, Hypernode v, Block
      V_from and V_to
1 delete v in all pq_i
2 deltaGainUpdate(H, PQ, v, V_from, V_to)
3 for e ∈ I(v) do
4     if e is not marked in_queue then
5         for u ∈ e do
6             if u ∉ pq_to ∧ u is unassigned then
7                 g_u ← calculateGain(H, V_from, V_to)
8                 pq_to.insert((u, g_u))
9         mark e as in_queue
```

In this variant, we use a separate assignment procedure to assign the remaining unassigned hypernodes after we grow all blocks, instead of repeating the *sequential* assignment process. There are two reason to do this in algorithm 9. First, if we repeat the procedure with the upper bound the algorithm tends to assign all remaining hypernodes to the first parts. To distribute the remaining unassigned hypernodes more evenly, we use the seperate assignment

procedure. Repeating the *sequential* assignment process would increase the cost of delta gain updates, because all *priority queues* are filled with hypernodes from the first iteration. If we further want to have the advantage of fast delta gain updates, we have to use this additional assignment process.

Global Greedy Hypergraph Growing

Algorithm 11: Global Greedy Hypergraph Growing Algorithm

```
Data: Hypergraph H = (V, E, c, ω), ε, k
Result: k-way partitioning P = (V_1, ..., V_k)
1  P ← (∅, ..., ∅)   // P.size() = k
2  PQ ← (pq_1, ..., pq_k)   // Array of binary max-heaps of size k
3  S ← calculateStartNodes(H, k)
4  for i from 1 until k do
5  │ g_{s_i} ← calculateGain(H, partOf(s_i), V_i)
6  └ pq_i.insert((s_i, g_{s_i}))
7
8  ε' ← 0 // Relaxing the input ε to avoid empty partitions
9  w_max ← ⌈c(V)/k⌉(1 + ε')
10 while there are unassigned nodes do
11 │ for i from 1 to k do
12 │ │ if pq_i is empty then
13 │ │ │ u ← search for a randomly chosen unassigned hypernode
14 │ │ └ pq_i.insert((u, calculateGain(H, partOf(u), V_i)))
   │ └
15 │ i ← argmax_{i∈{1,...,k}} pq_i.maxGain()
16 │ v ← pq_i.max()
17 │
18 │ if every V_i is disabled then
   │ │ // Every part is disabled and we use the input ε to assign all remaining unassigned
   │ │    hypernodes.
19 │ │ w_max ← ⌈c(V)/k⌉(1 + ε)
20 │ │ enable every partition V_i again
21 │ else if V_i is enabled ∧ c(V_i ∪ v) ≤ w_max then
22 │ │ V_i ← V_i ∪ {v}
23 │ │ update all gains in all pq_i of all hypernodes u ∈ Γ(v) and insert all u ∉ pq_i into pq_i
   │ │    with their corresponding gain // see algorithm 10
24 │ else
25 │ └ disable V_i
   └
26 return P = (V_1, ..., V_k)
```

The next GHGP implementation is the *global* algorithm. This variant chooses the move with the maximum gain among all possible moves from all priority queues. This move is called the *global* maximum gain move. To find this move, we search in all k priority queues for the hypernode with the highest gain and perform this move. The pseudocode of this variant is shown in algorithm 11. The initialzation step between line 1 and 7 is the same as in algorithm 9. At the beginnning we redefine our input ε to $\varepsilon' := 0$ in line 9, because we want to avoid empty parts. To find the *global* maximum gain move in the assigment process, we first insert into each

empty priority queue the gain of a randomly choosen unassigned hypernode. Afterwards we determine the highest gain among all k priority queues by a simple array scan (line 19). Ties are broken randomly to guarantee evenly growing partitions. We are able to enable or disable a block V_i for receiving further hypernodes. We have to take care that we only consider moves to enabled parts in line 19. All parts are initial enabled. If an assignment of a hypernode to an enabled block failed, because the resulting weight $c(V_i \cup \{v\})$ is greater than w_{max}, we disable this block. If all parts are disabled, we repeat the assignment process with the initial imbalance ε for our partition weight bound $w_{max} := \lceil \frac{c(V)}{k} \rceil (1 + \varepsilon)$ and enable all blocks again. If we find a valid move to a block V_i, we perform it. Afterwards we insert all new gain values of adjacent unassigned hypernodes into pq_i and perform delta gain updates (line 25 - 28 and algorithm 10). We repeat the assignment process until no unassigned hypernodes are left.

Round-Robin Greedy Hypergraph Growing

Algorithm 12: Round-Robin Greedy Hypergraph Growing Algorithm

```
Data: (Hyper)graph H = (V, E, c, ω), ε, k
Result: k-way partitioning P = (V_1, ..., V_k)
1  P ← (∅, ..., ∅)    // P.size() = k
2  PQ ← (pq_1, ..., pq_k)    // Array of binary max-heaps of size k
3  S ← calculateStartNodes(H, k)
4  for i from 1 until k do
5      g_{s_i} ← calculateGain(H, partOf(s_i), V_i)
6      pq_i.insert((s_i, g_{s_i}))
7
8  w_max ← ⌈c(V)/k⌉(1 + ε)
9  while there are unassigned nodes do
10     for i from 1 to k do
11         if V_i is enabled then
12             if pq_i is empty then
13                 u ← search for a random unassigned hypernode
14                 pq_i.insert((u, calculateGain(H, partOf(u), V_i)))
15             v ← pq_i.max()
16             if c(V_i ∪ v) ≤ w_max then
17                 V_i ← V_i ∪ {v}
18                 update all gains in all pq_i of all hypernodes u ∈ Γ(v) and insert all u ∉ pq_i into
                       pq_i with their corresponding gain // see algorithm 10
19             else
20                 disable V_i
21 return P = (V_1, ..., V_k)
```

The last implementation variant is called *round-robin* greedy hypergraph growing partitioner. This variant is similar to the BFS partitioner implementation in Section 4.1.2. In each round of the assignment process, each block V_i can choose one additional hypernode with the highest gain value in pq_i. The initialization step in line 1 until 7 is again similar to all other GHGP variants. The difference is at the beginning of the assignment process. We do not need to redefine our initial imbalance ε, because all parts grow simultaneously. In the assignment

phase we iterate over all parts and choose the hypernode v with the highest gain value in pq_i. If the resulting weight $c(V_i \cup \{v\})$ is smaller or equal than w_{max}, we assign it to V_i, otherwise we disable block V_i. In one iteration over all blocks only enabled parts are considered for receiving further hypernodes. This step is repeated until all hypernodes are assigned.

4.1.4 Label Propagation

The *Label Propagation*-based initial partitioning algorithm is inspired by the ideas of Henne [12]. The label propagation algorithm and the main idea of label propagation initial partitioning is described in section 3.3.4. In this Section we describe our implementation of a direct *k-way* label propagation initial partitioner.
Each hypernode has a label. A label represents the current block, which a hypernode is assigned to. An empty label represents an unassigned hypernode. Changing a label of a hypernode v from i to j means moving v from its current block V_i to V_j. As score function we use the *FM* gain function. If we choose a new label for a hypernode v, we choose a label of an adjacent hypernode $u \in \Gamma(v)$ where gain for moving v to the corresponding label/block is maximized.

Algorithm 13: Label Propagation-based Partitioning Algorithm

```
Data: (Hyper)graph H = (V, E, c, ω), ε, k
Result: k-way partitioning P = (V_1, ..., V_k)
1  P ← (∅, ..., ∅)    // P.size() = k
2  S ← calculateStartNodes(H, k)
3  for i from 1 until k do
4      B_i ← random subset B ⊆ Γ(s_i) ∧ 1 ≤ |B| ≤ 5
5      V_i ← V_i ∪ B_i
6
7  w_max ← ⌈c(V)/k⌉(1 + ε)
8  cur_iterations ← 0
9  while not converged ∧ cur_iterations ≤ i_max do
10     for v ∈ V in random order do
11         V_from ← partOf(v)
12         V_to ← compute max gain move to a incident label/partition to
13         if V_to ≠ V_from ∧ c(V_to ∪ {v}) ≤ w_max then
14             V_to ← V_to ∪ {v}
15             converged ← false
16     if converged ∧ there are unassigned hypernodes then
           // If the algorithm is converged, but there are still unassigned nodes the hypergraph
              consist of more than one connected components and we have to assgin a small
              amount of unassigned hypernodes to continue.
17         B ← random subset B ⊆ V with 1 ≤ |B| ≤ 5 ∧ ∀v ∈ B : v is unassigned
18         for v ∈ B do
19             V_to ← argmin_{V∈P} c(V)
20             V_to ← V_to ∪ {v}
21             converged ← false
22
23 return P = (V_1, ..., V_k)
```

Algorithm 13 shows the pseudocode of our label propagation initial partitioner. At the beginning we choose k pseudo peripherial start hypernodes (see section 3.3.2). In line 3 until 6 we

choose five different hypernodes around each start node s_i with a breath-first-search and assign them to part V_i. This procedure avoids empty parts at end of the partitioning process. Assume we assign only the start node s_i to block V_i. In the first round of label propagation a start node s_i decides to change its label. No other hypernode has label i. Therefore no hypernode would change its label to part i anymore, because only adjacent parts of a hypernode v are considered for the score calculation. With the assignment of five additional hypernodes around each start vertex, the probability of empty parts is minimized.
In the main loop we iterate over all hypernodes in random order until the algorithm has converged or we reach a predefined maximum number of iterations. For each hypernode v we have to calculate all scores to labels/blocks of adjacent hypernodes $u \in \Gamma(v)$. The new label of v is the label, where the score is maximized. If this label/block V_{to} differs from its current and the resulting weight of the part $V_{to} \cup \{v\}$ is smaller than our partition weight bound w_{max}, we change the label/block of v to V_{to}. If we iterate over all hypernodes and no hypernode changed its label, we have to check if there are still unassigned hypernodes. Assume we have a hypergraph $H = (V_1 \cup V_2, E, c, \omega)$ where $\forall A \subseteq V_1 : \forall B \subseteq V_2 : A \cup B \notin E$ and $V_1 \cap V_2 = \emptyset$. H consists of two connected components. If the start node set S is a subset of V_1, no hypernode $v \in V_2$ has an adjacent label and therefore none of them are able to get assigned to a part during the label propagation algorithm. This is caused by the fact that only labels of adjacent hypernodes are considered for score calculation. If this situation occurs, we choose a random set B of unassigned hypernodes with a size smaller or equal to five. We assign each hypernode of this set to a part with minimum weight. Then we repeat the process until the algorithm has converged and no unassigned hypernodes are left.
To calculate the new label for a hypernode v, we need to determine an adjacent label/block where the FM score is maximized. Since we call this method in each round of label propagation $|V|$ times, we have to implement it very efficient. Algorithm 14, shows the implementation of this calculation. The pseudocode is structured into two sections. First, we calculate the FM gain for moving hypernode v to all adjacent parts and then we search for the move with maximum gain. To store the FM gain to all adjacent blocks we use an array of size k, which is called *tmp_gains*. We iterate over all hyperedges $e \in I(v)$ and check if the net is a internal or cut net. If the hyperedge e is an internal net and $\Lambda(e) = \{V_i\}$ with $i \in \{1, ..., k\}$, moving a hypernode v to a block V_j with $j \neq i$ would increase the cut, except if v is the only assigned hypernode in net e. In this case the connectivity set $\Lambda(e) = \{V_i\}$ changes only to $\Lambda(e) = \{V_j\}$. If e becomes a cut net, we add $\omega(e)$ to *tmp_gains*[i] and to *internal_weight*. In *internal_weight* we sum the weight of all internal nets $e \in I(v)$. We substract *internal_weight* from all *tmp_gains*[i] with $i \in \{1, ..., k\}$ at the end of the gain calculation. By adding $\omega(e)$ to *tmp_gains*[i] we ensure that only the move to block V_i is a zero gain move. All other moves to an other part ($\neq V_i$) makes e a cut net. We can handle the internal net case then in $\mathcal{O}(1)$ instead of $\mathcal{O}(k)$. If the net e is already a cut net, we can only reduce the cut with a move of a hypernode v, if $\lambda(e) = 2$ and $e \cap V_{from} = \{v\}$. If this condition is true, we add the edge weight $\omega(e)$ to *tmp_gains*[*to*].
In the next phase we have to search for the maximum score in *tmp_gains*, which represents the new label/block of hypernode v. If hypernode v is an unassigned hypernode and there exists a neighbour $u \in \Gamma(v)$ with a non empty label, we assign it to the label where the score is maximized (ensure by setting g_{max} to $-\infty$ in line 21). Otherwise, we only want to move hypernode v, if the move decrease the cut (ensure by setting g_{max} to zero in line 19). A gain value *tmp_gains*[i] is only considered, if the corresponding part/label V_i is adjacent to hypernode v (line 24). Now we can iterate over all gains temporally calculated and subtract the weight of all internal hyperedges before comparing it with the current maximum gain. If the move of hypernode v to a part V_i produces a feasible weight of block V_i and if the gain of this move is greater as the current maximum gain, we save it as our new maximum gain and continue.

The first phase of this algorithm 14 has a worst case running time of $\mathcal{O}(|I(v)|k)$, because in the worst case every net $e \in I(v)$ has connectivity $\lambda(e) = k$. The next phase is a simple array scan with constant time operations, so we have a running time of $\mathcal{O}(k)$. The worst case running time of the maximum gain move calculation is $\mathcal{O}(|I(v)|k)$.

Algorithm 14: Computation of Max Gain Move to Incident Label

```
Data: (Hyper)graph H = (V, E, c, ω), Hypernode v
Result: Max Gain Move (V_max, g_max)
1  tmp_gains ← (0, ..., 0) // Array of size k
2  internal_weight ← 0
3  V_from ← partOf(v)
4
5  for e ∈ I(v) do
6      if λ(e) = 1 ∧ |e ∩ V_from| > 1 then
7          V_i ← corresponding label/block in net e
8          internal_weight ← internal_weight + ω(e)
9          tmp_gains[i] ← tmp_gains[i] + ω(e)
10     else
11         for V_to ∈ Λ(e) do
12             if λ(e) = 2 ∧ |e ∩ V_from| = 1 ∧ V_from ≠ V_to then
13                 tmp_gains[to] ← tmp_gains[to] + ω(e)
14
15 (V_max, g_max) ← (V_from, 0)
16 if from = unassigned_part then
17     (V_max, g_max) ← (V_from, −∞)
18 for i from 1 until k do
19     if ∃e ∈ I(v) : V_i ∈ Λ(e) then
20         tmp_gains[i] ← tmp_gains[i] − internal_weight
21         if c(V_i ∪ {v}) ≤ w_max ∧ tmp_gains_i > g_max then
22             (V_max, g_max) ← (V_i, tmp_gains_i)
23 return (V_max, g_max)
```

4.1.5 Additional Implementation Details

We implement two heuristics to improve our direct *k-way* partitioners. Every partitioner is able to perform a rollback operation to the best cut seen during initial partitioning. After each initial partitioning we can use a *local search* algorithm to improve the quality.
If we assign all hypernodes to a block V_i before initial partitioning, we can store every move of a hypernode from a part V_i to V_j and calculate the cut after each move. A partition P is feasible, if it fulfils the balance constraints and all hypernodes are assigned. If we assign only unassigned hypernodes during initial partitioning, the partition is feasible after the last hypernode is assigned. Therefore we cannot rollback to best cut, because the last partition is the only feasible solution. Otherwise, it is possible that there exists more than one valid partition P, which are produced during initial partitioning. In this case we can rollback at the end of the initial partitioning to a partition P_{best} with minimum cut.
As additional improvement of our initial partitions we use a *k-Way FM local search* algorithm, which is part of the n-level hypergraph partitioning framework *KaHyPar*.

4.2 Recursive Bisection

Alternative to direct k-*Way* partitioning, a k-partition can be obtained with the recursive bisection method. Recursive bisection is a method which takes a hypergraph H and bisect it in two parts. The resulting blocks are further bisected recursively until we have k parts. For a detailed description of this method we refer the reader to Section 3.3.6. If we bisect a hypergraph into two near equal weighted parts, k is restricted to be a power of 2. To partition a hypergraph into k parts with recursive bisection, where $k \neq 2^n$, the weight of the resulting parts V_1 and V_2 of each bisection has to be less or equal than two different weight bounds w_1 and w_2 with $c(V_1) \leq w_1$ and $c(V_2) \leq w_2$. This is caused by the fact that we want to further bisect the first resulting part in $\lfloor \frac{k}{2} \rfloor$ and the second in $\lceil \frac{k}{2} \rceil$ parts. For a bisection, where we want to further divide a subhypergraph $H' = (V', E', c, \omega)$ into $k' \leq k$ parts, we allow a maximum weight of $w_1 = \frac{c(V')}{k'} \lfloor \frac{k'}{2} \rfloor (1 + \varepsilon')$ for the first block and $w_2 = \frac{c(V')}{k'} \lceil \frac{k'}{2} \rceil (1 + \varepsilon')$ for the second one. ε' is adapted at each bisection, such that it allows maximum imbalance while still being small enough to ensure the final imbalance with ε. Section 4.2.1 introduces the main problem with the usage of the initial imbalance ε on each bisection and the description of our *adaptive epsilon* ε'. If we want to further divide the two parts V_1 and V_2 of each bisection, we have to extract the subhypergraphs $H_1 = (V_1, E_1, c, \omega)$ and $H_2 = (V_2, E_2, c, \omega)$. The hyperedge sets E_1 and E_2 contains only internal nets of V_1 and V_2 (see Section 3.3.6). The implementation details of our recursive bisection partitioner is described in Section 4.2.2.
We decide to execute recursive bisection in reverse pre-order. We extract the second part and furhter bisect it before the first part, because there exists a small problem with the execution in pre-order. Assume we execute recursive bisection in pre-order and want to divide a hypergraph H into $k = 4$ parts. First we bisect the original hypergraph H into V_1 and V_2. Afterwards we extract block V_1 as subhypergraph H'. The bisection of H' produces the final blocks V_1 and V_2 on the original hypergraph. We assign the hypernodes of H' in H to their corresponding block and go back to the original hypergraph H, where we have to extract partition V_2. We have bisected hypergraph H at the beginning into V_1 and V_2. Further we have assigned the final block V_2 on the original hypergraph H as well, which results from the bisection of the extracted subhypergraph H' of part V_1. If we extract part V_2 on the original hypergraph H, we extract the final part V_2 and V_2 of the first bisection. The result is that partition V_2 is the end of recursive bisection empty. Figure 15 illustrates this problem.

4.2.1 Adaptive Epsilon

At this section we work out a formulation for an ε', which can be adapted before each bisection, which leads to a greater exploration of the solution space during recursive bisection. Furthermore it is guaranteed that we fulfil the imbalance with ε at the end.
Using the initial imbalance parameter ε in each bisection step can lead to an imbalanced partition at the end of recursive bisection. Assume block V_1 has the maximum possible part weight $c(V_1) \approx \frac{c(V)}{2}(1 + \varepsilon)$ which is allowed at the first bisection step. The same case occurs at the next bisection with the subhypergraph of part $V_1 \Rightarrow c(V_1') \approx \frac{c(V)}{4}(1 + \varepsilon)^2$. If we want to divide a hypergraph into k parts with recursive bisection and the described situation above occurs, the weight of the final block V_1 is $c(V_1) \approx \frac{c(V)}{k}(1 + \varepsilon)^{\log_2(k)}$, if $k = 2^n$ with $n \in \mathbb{N}_{>0}$. This is the worst case part weight of a block. We can avoid this case, if we restrict our initial imbalance ε. A solution is to choose an $\varepsilon' \leq \varepsilon$ for each bisection, which ensure that every part is less or equal than the maximum allowed block weight at the end of recursive bisection. We can choose a fix value for ϵ' or we can calculate it according to the initial imbalance ε and k.
Our imbalance is defined in definition 2.4. Every part $V' \in P$ should have a weight less than

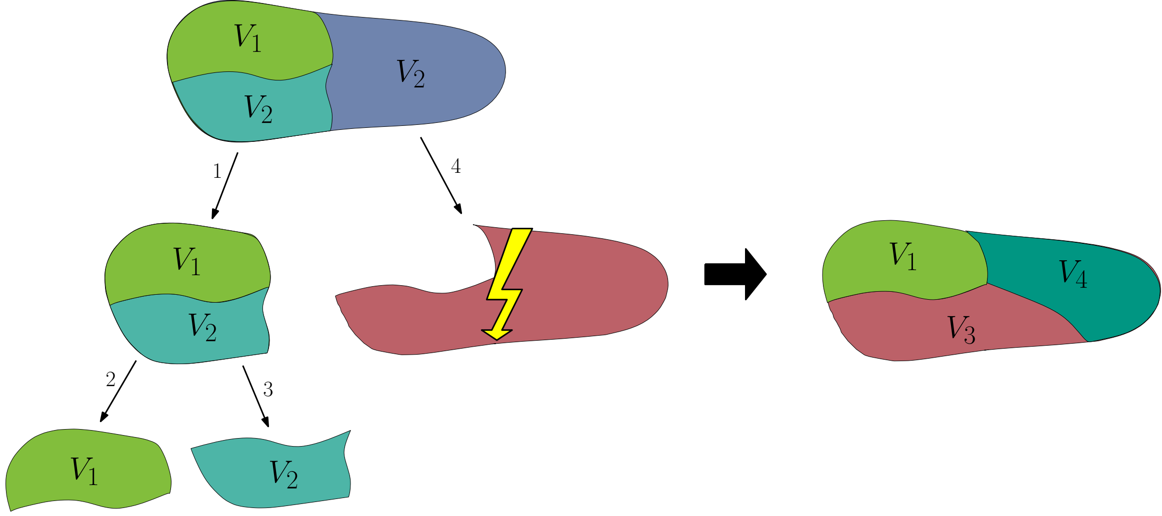


Figure 15: Problem with the pre-order traversal of recursive bisection. The number on the arrow denotes the execution order. On the right side you see the final partition, if we execute recursive bisection with pre-order traversal.

or equal to $L_{max} := \lceil \frac{c(V)}{k} \rceil (1+\varepsilon) + \max_{v \in V} c(v)$ for a given input ε. To define our new ε' we use another definition, which is $L'_{max} := \frac{c(V)}{k}(1+\varepsilon)$. This definition simplifies the calculation and if we can ensure that the resulting weights of each part at end of recursive bisection fulfils the imbalance with L'_{max}, we can fulfil the imbalance with L_{max}, since $L'_{max} \leq L_{max}$. First, we work out a formulation for the adaptive ε', if $k = 2^n$ and at end of this section we show our observations for the general case, if $k \neq 2^n$.
Assume $k = 2^n$ with $n \in \mathbb{N}_{>0}$. The final block V_1 is created after $\log_2(k)$ bisections, if we recursively further divide the first block before the second. Assume that on each bisection the first block has the maximum possible part weight. The final weight of block V_1 after the last bisection is

$$c(V_1) := \frac{c(V)}{2^{\log_2(k)}}(1+\varepsilon)^{\log_2(k)} = \frac{c(V)}{k}(1+\varepsilon)^{\log_2(k)} \tag{4.1}$$

This is the heaviest part weight of a block which can occur at the end of recursive bisection. We can replace V_1 with an arbitrary block $V' \in P$, because the heaviest possible weight for it is the same. By replacing ε with ε' in $c(V')$ we ensure that the weight in equation 4.1 is less or equal than L'_{max}.

$$\begin{aligned} c(V') = \frac{c(V)}{k}(1+\varepsilon')^{\log_2(k)} \leq \frac{c(V)}{k}(1+\varepsilon) = L'_{max} \\ \Rightarrow \varepsilon' \leq (1+\varepsilon)^{\log_2(k)^{-1}} - 1 \end{aligned} \tag{4.2}$$

We have a restriction ε' for the initial imbalance ε, which ensures that all blocks $V' \in P$ have a weight less than or equal to L_{max} at the end of recursive bisection by using ε' for each bisection. If we use on each bisection the same ε', we restrict the solution space unnecessary. The restricted ε' was constructed based on a worst case scenario. If a bisection produces two blocks with equal weight, we can choose ε' much greater for the following bisections. It may be desirable to adapt ε' before each bisection. By using the *adaptive epsilon* ε', which we calculate before each bisection again, we can ensure that the resulting weights of all parts after the end of recursive bisection is less or equal than L'_{max} and we can explore a greater solution space as with the usage of a fix ε' for each bisection. Since $L'_{max} \leq L_{max}$, we fulfil our original imbalance as well.

Assume we have a subhypergraph $H' = (V', E', c, \omega)$, which was created from bisections of the original hypergraph H during the recursive bisection process. The task is to further divide H' in $k' = 2^{n'}$ parts with $k' \leq k$ with recursive bisection. To calculate the *adaptive epsilon* ε' for the bisection of subhypergraph H', we have to calculate the heaviest possible block weight of a part V''. The considerations for the weight of this part $V'' \in P$ is the same as for equation 4.1 and the calculation of ε' is equal to the calculation in inequality 4.2.

$$\begin{aligned} c(V'') = \frac{c(V')}{k'}(1+\varepsilon')^{\log_2(k')} \leq \frac{c(V)}{k}(1+\varepsilon) = L'_{max} \\ \Rightarrow \varepsilon' \leq (\frac{k' \cdot c(V)}{k \cdot c(V')}(1+\varepsilon))^{\log_2(k')^{-1}} - 1 \end{aligned} \tag{4.3}$$

If we want to ensure that the resulting parts at the end of recursive bisection with $k = 2^n$ have a weight less or equal than L'_{max}, we have to set our ε' to $(1+\varepsilon)^{\log_2(k)^{-1}} - 1$ for each bisection. At the first bisection step the weight of the two resulting blocks should not exceed $\frac{c(V)}{2}(1+\varepsilon')$. If we are on an arbitrary recursive bisection step and want to further bisect a subhypergraph H' into $k' = 2^{n'} \leq k$ parts, we have to set our ε' to $(\frac{k' \cdot c(V)}{k \cdot c(V')}(1+\varepsilon))^{\log_2(k')^{-1}} - 1$. The weight of the resulting parts of the bisection of subhypergraph H' should also not exceed $\frac{c(V')}{2}(1+\varepsilon')$. We have defined a adaptable restriction for the maximum allowed imbalance for each bisection, if $k = 2^n$ with $n \in \mathbb{N}$. The next step is to expand the definition of the *adaptive epsilon* for an arbitrary $k \in \mathbb{N}$.

Theorem 4.1. Given a hypergraph $H = (V, E, c, \omega)$, which should be divided into k parts with recursive bisection. The resulting weights of each part of the final partition $P = \{V_1, ..., V_k\}$ should not exceed $L'_{max} := \frac{c(V)}{k}(1+\varepsilon)$ for a given initial imbalance ε. If we want to further divide a subhypergraph $H' = (V', E', c, \omega)$ into $k' \leq k$ parts, which results from bisections of the original hypergraph H, we can adapt ε' before the bisection of H' to

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

The *adaptive epsilon* ε' ensures that P is ε-balanced $\Rightarrow \forall V_i \in \{1, ..., k\} : c(V_i) \leq L'_{max} \leq L_{max}$.

Proof. We have shown Theorem 4.1 for a $k = 2^n$ with $n \in \mathbb{N}_{>0}$. Now we have to show it for an arbitrary $k \in \mathbb{N}_{>0}$. If we want to divide a hypergraph H into k parts, we have to ensure that the weight of the first block V_1 does not exceed $\frac{c(V)}{k}\lfloor\frac{k}{2}\rfloor(1+\varepsilon')$ and the second one V_2 does not exceed $\frac{c(V)}{k}\lceil\frac{k}{2}\rceil(1+\varepsilon')$ at the first bisection, because we want to further divide V_1 in $\lfloor\frac{k}{2}\rfloor$ blocks and the second V_2 in $\lceil\frac{k}{2}\rceil$ blocks. If we want to proof Theorem 4.1 for an arbitrary $k \in \mathbb{N}$, we have to show that each resulting block at the end of recursive bisection, if we use the *adaptive epsilon* ε' for each bisection, has a weight less than or equal to L'_{max}. To proof this we have to calculate the heaviest possible part weight of an block at the end of recursive bisection. Let us consider an arbitrary path $p = (k_0 \to k_1 \to ... \to k_{\lceil \log_2(k) \rceil})$ through all levels of recursive bisection, which indicates in how many parts one of the two blocks of the bisection on level $i \in \{0, ..., \lceil \log_2(k) \rceil\}$ should be further divided. Figure 16 illustrates this path p. We can more precisely define path p as follows:

$$\begin{aligned} k_0 = k \\ k_i = \max(\lfloor\frac{k_{i-1}}{2}\rfloor, 1) \vee k_i = \lceil\frac{k_{i-1}}{2}\rceil \end{aligned} \tag{4.4}$$

This path means that we have to divide hypergraph H into k_0 parts. One of the resulting parts of the first bisection should be further divided into k_1 parts. If we are on an arbitrary level i (see Figure 16) and have to further divide a subhypergraph H', resulting from bisections along our path p, in k_{i-1} parts, we have to ensure that the resulting weight of the corresponding part, which we further want to divide in k_i parts, does not exceed $\frac{c(V')k_i}{k_{i-1}}(1+\varepsilon')$. Because we consider an arbitrary path of partition sizes through each level of recursive bisection we can choose either $\lfloor\frac{k_{i-1}}{2}\rfloor$ or $\lceil\frac{k_{i-1}}{2}\rceil$ for the value of k_i in equation 4.4. If we choose $\lfloor\frac{k_{i-1}}{2}\rfloor$, we have to ensure that k_i cannot become zero, since it makes no sense to divide an subhypergraph H' into zero parts. We assume that each bisection produce the maximum possible weight for the block we further want to divide along path p. Now we are able to calculate heaviest possible part weight of the resulting block V'.

$$c(V') = \frac{c(V)\prod_{i=0}^{\lceil\log_2(k)\rceil-1} k_{i+1}}{\prod_{i=0}^{\lceil\log_2(k)\rceil-1} k_i}(1+\varepsilon')^{\lceil\log_2(k)\rceil} = \frac{c(V)k_{\lceil\log_2(k)\rceil}}{k_0}(1+\varepsilon')^{\lceil\log_2(k)\rceil} = \frac{c(V)}{k}(1+\varepsilon')^{\lceil\log_2(k)\rceil} \tag{4.5}$$

The products in the fraction cancel each other off except k_0 and $k_{\lceil\log_2(k)\rceil}$. Since we define $k_0 = k$ and $k_{\lceil\log_2(k)\rceil} = 1$, we can replace them in equation 4.5. We have calculated the heaviest possible weight of a block, which can occur, if we divide a hypergraph H into $k \in \mathbb{N}$ parts with recursive bisection. To finish the proof, we have to make the same considerations as for equation 4.3. Assume again we have a subhypergraph $H' = (V', E', c, \omega)$, which results from bisections of the original hypergraph H, and should be further divided into $k' \leq k$ parts with $k' \in \mathbb{N}$. Let us consider again an arbitrary path $p' = (k_0 \rightarrow k_1 \rightarrow \ldots \rightarrow k_{\lceil\log_2(k')\rceil})$ where we define $k_0 = k'$. We can calculate the heaviest possible part weight along that path p' of the resulting block V'' like in equation 4.5.

$$c(V'') = \frac{c(V')\prod_{i=0}^{\lceil\log_2(k')\rceil-1} k_{i+1}}{\prod_{i=0}^{\lceil\log_2(k')\rceil-1} k_i}(1+\varepsilon')^{\lceil\log_2(k')\rceil} = \frac{c(V')k_{\lceil\log_2(k')\rceil}}{k_0}(1+\varepsilon')^{\lceil\log_2(k')\rceil} = \frac{c(V')}{k'}(1+\varepsilon')^{\lceil\log_2(k')\rceil} \tag{4.6}$$

We have only to estimate $c(V'')$ with the estimation of ε' of Theorem 4.1 in equation 4.6 to finish the proof.

$$\begin{aligned} c(V'') = \frac{c(V')}{k'}(1+\varepsilon')^{\lceil\log_2(k')\rceil} &\leq \frac{c(V')}{k'}(1+(\frac{k'\cdot c(V)}{k\cdot c(V')}(1+\varepsilon))^{\lceil\log_2(k')\rceil^{-1}}-1)^{\lceil\log_2(k')\rceil} \\ &= \frac{c(V)}{k}(1+\varepsilon) = L'_{max} \end{aligned} \tag{4.7}$$

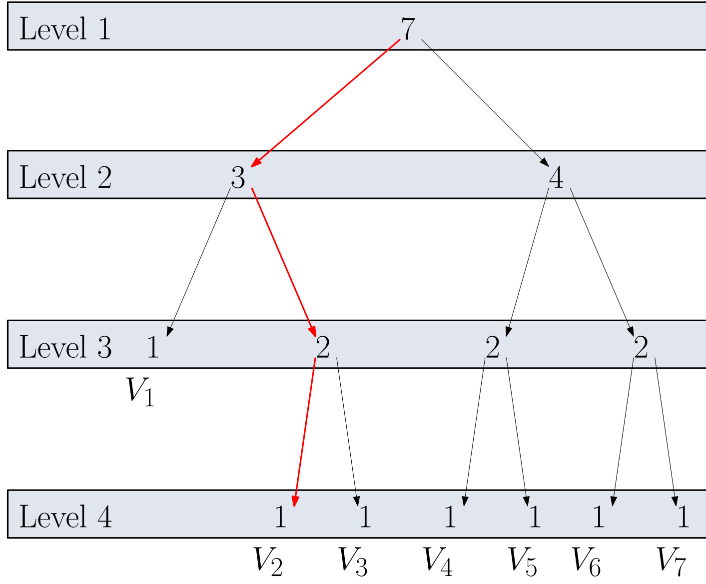
□

4.2.2 Implementation

If we bisect a hypergraph $H = (V, E, c, \omega)$ into $P = (V_1, V_2)$ and extract one of the resulting parts V_i with $i \in \{1, 2\}$ into a subhypergraph

$$H' = (V_i, E', c, \omega)$$

$$E' = \{e \mid e \in E : \Lambda(e) = \{V_i\}\}$$



Worst case partition weights:

$V_1 = \frac{c(V)\cdot 3\cdot 1}{7\cdot 3}(1+\epsilon')^2 = \frac{c(V)}{7}(1+\epsilon')^2$

$V_2 = \frac{c(V)\cdot 3\cdot 2\cdot 1}{7\cdot 3\cdot 2}(1+\epsilon')^3 = \frac{c(V)}{7}(1+\epsilon')^3$

$V_3 = \frac{c(V)\cdot 3\cdot 2\cdot 1}{7\cdot 3\cdot 2}(1+\epsilon')^3 = \frac{c(V)}{7}(1+\epsilon')^3$

$V_4 = \frac{c(V)\cdot 4\cdot 2\cdot 1}{7\cdot 4\cdot 2}(1+\epsilon')^3 = \frac{c(V)}{7}(1+\epsilon')^3$

$V_5 = \frac{c(V)\cdot 4\cdot 2\cdot 1}{7\cdot 4\cdot 2}(1+\epsilon')^3 = \frac{c(V)}{7}(1+\epsilon')^3$

$V_6 = \frac{c(V)\cdot 4\cdot 2\cdot 1}{7\cdot 4\cdot 2}(1+\epsilon')^3 = \frac{c(V)}{7}(1+\epsilon')^3$

$V_7 = \frac{c(V)\cdot 4\cdot 2\cdot 1}{7\cdot 4\cdot 2}(1+\epsilon')^3 = \frac{c(V)}{7}(1+\epsilon')^3$

$\rightarrow$ $p = (7 \rightarrow 3 \rightarrow 2 \rightarrow 1)$

Figure 16: Example of a recursive bisection tree. The red path shows an example of a path of partition sizes through each level of recursive bisection. On the right side an example of the possible worst case parts weights is shown of each block.

we need to define a mapping $f : V_i \rightarrow V$, which maps the hypernodes from H' to H. This mapping is necessary, because our extraction method creates a new hypergraph and renumber the hypernodes.

Definition 4.1. Assume we have to divide a hypergraph $H = (V, E, c, \omega)$ with recursive bisection into k parts. Given an extraction chain of subhypergraphs $U = \{H_0, H_1, ..., H_l\}$, where $H_0 = H$ and H_i with $1 \leq i \leq l$ results from an extracted part of H_{i-1}. For all extraction chains U exists a mapping chain $M = \{g_0, ..., g_{l-1}\}$, where $g_j : V_{j+1} \rightarrow V_j$ with $0 \leq j \leq l-1$. We can map a hypernode from an subhypergraph $H_i \in U$ with $1 \leq i \leq l$ to the original hypergraph H with the following function $f : V \times U \rightarrow V$

$$f(v, H_i) = \begin{cases} f(g_{i-1}(v), H_{i-1}) & i > 0 \\ v & i = 0 \end{cases} \tag{4.8}$$

The pesudocode of our recursive bisection implementation is given in algorithm 15. We implemented recursive bisection with a stack. The extraction and mapping chain from defintion 4.1 are stored in stacks U and M. The stack O stores on top the range (k_1, k_2) of parts, which the subhypergraph H' on top of U should be further divided. This means that H' should be further divided into $k' = k_2 - k_1$ parts. The blocks, which are produced with recursive bisection starting from H', represents blocks k_1 until k_2 on the original hypergraph H.
During recursive bisection each hypergraph on the stack is associated with one of three possible states:

- *unpartitioned* (line 20-26): The hypergraph H is unpartitioned.
- *partitionTwoExtracted* (line 26-31): The hypergraph H is bisected and V_2 is already extracted.

- *finished* (line 18-19): The hypergraph H is finally divided in the range of parts (k_1, k_2) on top of O, if H is on top of the stack U.

At the beginning, we push the base hypergraph H on top of stack U with the state *unpartitioned* and we push the range $(1, k)$ on top of O. This means we want to divide hypergraph H into k parts. We can now start recursive bisection. At the beginning of the *While*-Loop (line 4), we fetch the current hypergraph-state pair (H', S) and the range (k_1, k_2) of parts, which the hypergraph H' should be further divided. Afterwards we calculate our current *adaptive epsilon* ε' and the part weight bounds w_1 and w_2 which bound the resulting part weights of the bisection. If $k_2 - k_1 \equiv 0$, we have reached a subhypergraph H' which represents part V_{k_1} on the original hypergraph. We assign all hypernodes from subhypergraph H' to block V_{k_1} in H. To map the hypernodes of H' to H we use the mapping function f from definition 4.1 and our mapping chain M.

Algorithm 15: Recursive Bisection

Data: (Hyper)graph $H = (V, E, c, \omega)$, ε, k, InitialPartitioner *ip*
Result: *k-way* partitioning $P = (V_1, ..., V_k)$

```
1  Initalize hypergraph-state stack U, mapping stack M and partition size stack O
2  U.push((H, unpartioned)), O.push((1, k)) // means H should be further partitionen into
       partition 1 until k
3
4  while U is not empty do
5      (H' = (V', E', c, ω), S) ← U.top()
6      (k1, k2) ← O.top()
7      k' ← k2 − k1 + 1 // Hypergraph H' should be further divided into k' parts
8      ε' ← ((k'·c(V))/(k·c(V'))(1 + ε))^(log2(k')^−1) − 1
9      w1 ← c(V)/k ⌊k/2⌋(1 + ε'), w2 ← c(V)/k ⌈k/2⌉(1 + ε')
10
11     if k2 − k1 = 0 then
12         Initialize extraction chain U' from U and mapping chain M' from M
13         Initialize mapping function f' : V × U' → V // see definition 4.1
14         for v ∈ V' do
15             V_k1 ← V_k1 ∪ {f'(v, H')}
16         U.pop(), M.pop(), O.pop()
17     else if S ≡ finish then
18         U.pop(), M.pop(), O.pop()
19     else if S ≡ unpartition then
20         (V1, V2) ← ip.partition(H', 2, w1, w2) // c(V1) ≤ w1 ∧ c(V2) ≤ w2
21         (H'' = (V'', E'', c, ω), g'') ← extract partition V2 from H' // g'' : V'' → V'
22         U.pop()
23         U.push((H', partitionTwoExtracted)), U.push((H'', unpartition))
24         M.push(g''), O.push((k1 + ⌊k'/2⌋, k2))
25     else if S ≡ partitionTwoExtracted then
26         (H'' = (V'', E'', c, ω), g'') ← extract partition V1 from H' // g'' : V'' → V'
27         U.pop()
28         U.push((H', finish)), U.push((H'', unpartition))
29         M.push(g''), O.push((k1, k1 + ⌊k'/2⌋ − 1))
30 return P = (V1, ..., Vk)
```

If $k_2 - k_1 \neq 0$, we have to decide based on the state of the current subhypergraph H' what we have to do. If $H' = (V', E', c, \omega)$ is associated with the state *unpartitioned*, we have to bisect it. We are able to run our initial partitioner several times in line 20 and take the best bisection from all runs. After the bisection we extract part V_2 as subhypergraph $H'' = (V'', E'', c, \omega)$ and push it onto the stack U with state *unpartioned*. We push the corresponding mapping $g' : V'' \rightarrow V'$ on top of stack M and the range of blocks $(k_1 + \lfloor \frac{k_2-k_1}{2} \rfloor, k_2)$, which the hypergraph H'' should be further divided, onto stack O. The state of hypergraph H' changes from *unpartitioned* to *partitionTwoExtracted*. If the state of subhypergraph H' is *partitionTwoExtracted*, we have to extract block V_1 as subhypergraph H'' and put it onto the stack U with the state *unpartioned*. We push the corresponding mapping on top of stack M and the range of parts $(k_1, k_1 + \lfloor \frac{k_2-k_1}{2} \rfloor - 1)$ on top of stack U. The state of hypergraph H' changes from *partitionTwoExtracted* to *finished*. If the subhypergraph H' onto stack U is associated with the state *finished*, we can remove the hypergraph H', mapping and part range from U, M and O.

4.3 Recursive Bisection n-Level Hypergraph Initial Partitioning

We have presented in the sections before many direct *k-way* and recursive bisection initial partitioning algorithms for the n-level partitioning framework *KaHyPar*. Now we use *KaHyPar* and our recursive bisection implementation to provide a recursive bisection n-level initial partitioning method. We use the n-level partitioning scheme on each bisection. We coarsen the hypergraph until a predefined number of hypernodes are left. In the next step we choose one of our initial partitioning algorithms to produce a bisection on the coarsened hypergraph. At the end we uncontract the contracted hypernodes and use a *local search* heuristic to improve the quality of the bisection. To use *KaHyPar* as initial partitioner, we have to evaluate the best configuration of it for this use case.
The available coarsener are described in Section 3.5. For our recursive bisection n-level initial partitioner we choose the *partial* coarsening method. For the coarsening phase we have to setup the contraction limit t and the hypernode weight limit s. We contract hypernodes until $k \cdot t$ hypernodes are left and no contraction of two hypernodes u and v should produce a hypernode w with a weight $c(w) > c_{max} := s\frac{c(V)}{t \cdot k}$. Furthermore we have to evaluate our initial partitioning algorithms in the recursive bisection n-level partitioning context.
The last step is to choose a *local search* algorithm for the uncoarsening phase. Since we using n-Level hypergraph partitioning in a recursive bisection context, we only perform a bisection on the coarsest hypergraph. We have tested and evaluated two *local search* algorithms for our recursive bisection n-Level initial partitioner. We tested a *ScLap*-based [12] and a *FM local search* heuristic.

4.3.1 Pool Initial Partitioner

We implement an initial partitioner which calls a subset of our initial partitioning algorithms sequentially and chooses the best partition of all runs. We call it *pool* initial partitioner. Since the resulting hypergraph in the recursive bisection n-Level context has only $2t$ hypernodes left at the end of the coarsening phase, initial partitioning becomes very fast and we can perform more than one run of each initial partitioner. The pool initial partitioner contains a subset of our partitioners $A = \{ip_1, ..., ip_l\}$ and calls each partitioner n times.
The pseudocode of the *pool* partitioner is shown in Algorithm 16. Each partitioner performs n initial partitioning runs. If an initial partitioner produces a partition with a better cut than the cut of the current best partition, we only apply the new partition as our new best partition,

if the imbalance of this partition P is less than or equal to ε or if the current imbalance is less or equal than the imbalance from the best partition. The pool initial partitioner calls every partitioner in A $n = 20$ times. In Section 5.3 we evaluate different compositions for A.

Algorithm 16: Pool Initial Partitioner

```
Data: (Hyper)graph H = (V, E, c, ω), ε, k, n
Result: k-way partitioning P = (V_1, ..., V_k)
1 P_best ← (∅, ..., ∅)
2 A ← {ip_1, ..., ip_l}
3 for i from 1 until l do
4     P ← ip_i.partition(H, ε, k) // Each partitioner call themselve n times.
5     if cut(P) ≤ cut(P_best) then
6         if imbalance(P) ≤ ε ∨ imbalance(P) ≤ imbalance(P_best) then
7             P_best ← P
8 return P_best
```

5 Experimental Results

In this chapter we present the experimental results of our work. In Section 5.2 we present the results of the initial partitioning methods based on direct *k-way* and recursive bisection. We run all partitioners in different configurations and outline the most interesting results. Furthermore we test the quality and running time of our recursive bisection methods with multiple runs at each bisection. In Section 5.3 we evaluate our recursive bisection n-level initial partitioner. We have made numerous amounts of tests to find the best configuration for it. At the end we integrated the final initial partitioning algorithm into *KaHyPar* and compare ourselves against *hMetis* and *PaToH* as initial partitioner in Section 5.4.

5.1 Test-Environment

5.1.1 Instances

For our experiments we used instances of two benchmark sets from the two big application areas of hypergraph partitioning. We used the ISPD98 Circuit Benchmark Suite [1] and a subset of the Florida Sparse Matrix Collection [9]. We divide the benchmark sets in *medium-sized* and *large* instances. The former are used for parameter tunning during our development process and the latter are used for the final tests of our partitioner in *KaHyPar* as initial partitioner in Section 5.4. An overview of our benchmark instances is shown in tables 15 and 16.

5.1.2 System

We implement our algorithms in `C++` and compile it with `gcc` version 4.9.2 with the flags `-std=c++14 -O3 -mtune=native -march=native`. The system used for parameter tunning instance tests is running *Ubuntu 12.04* and contains two Intel Xeon X5355 2.66 GHz processors where each has four cores. Furthermore the system has four L1-Caches of size 32 KiB, two L2-Cache of size 4 MiB and the main memory has a size of 24 GiB. Experiments where we only have to verify a quality property is executed in parallel on several systems with the architecture described above. Experiments regarding running time, we execute each on a single core on one system.
The final tests of our initial partitioner in *KaHyPar* with the *large* benchmark instances used a system with two Intel Xeon E5-2670 processors clocked at 2.6 GHz where each has eight cores. The system has eight 256 KiB L2-Caches, one 20 MiB L3-Cache and the main memory has a size of 64 GiB.

5.1.3 Methodology

We have coarsened the *medium-sized* benchmark instances with *KaHyPar* ten times with ten different seeds for all $k \in \{2, 4, 8, 16, 32, 64\}$. For each hypergraph and k we have ten different coarsened hypergraph versions of the initial input hypergraph with $160k$ hypernodes. For each instance we execute our initial partitioner with five different seeds. If we analyse the overall results (e.g. cut) of an experiment, we calculated the average results for each hypergraph and k value and then the geometric mean overall average results. For each test we use imbalance $\varepsilon = 0.03$. We compared the results of our algorithms in section 5.2 and 5.3 with the results from the recursive bisection variant of *hMetis* and *PaToH*.
For the final tests in Section 5.4 we execute *KaHyPar* with our initial partitioner with ten different seeds on each instance of the *full* benchmark set, with $k \in \{2, 4, 8, 16, 32, 64, 128\}$ and

$\varepsilon = 0.03$. We compare the results of *KaHyPar* with our initial partitioner against the usage of the recursive bisection variants of *hMetis* and *PaToH* as initial partitioner.

5.2 Direct *k-way* and Recursive Bisection Initial Partitioners

In this section we present the experimental results of our direct *k-way* and recursive bisection methods described in Section 4.1 and 4.2. The results of the first experiments are summarized in Table 1. This table shows the cuts and the running times of each algorithm with different configurations in comparison with *hMetis* and *PaToH*. We execute the algorithms with six different configuration which are described in the first row of table 1. The first entry in each configuration is either +1 or −1, which indicates, whether we assign all hypernodes to block V_1 or leave all unassigned before initial partitioning. The next two parameters in the configuration are R for *rollback* and FM for *FM local search* heuristic which are both described in section 4.1.5. With +/−, we denote whether (+) or not (−) we use the corresponding technique. Furthermore we shorten the names of the *Greedy*, *BFS*, *Random* and *Label Propagation* initial partitioner to *greedy* (section 4.1.3), *bfs* (section 4.1.2), *random* (section 4.1.1) and *lp* (section 4.1.4). The abbreviation *RB* in a name of an algorithm indicates that the recursive bisection variant of this algorithm is used. The different greedy hypergraph growing variants are *greedy-S* for *sequential*, *greedy-G* for *global* and *greedy-R* for *round-robin*. The different gain functions are *FM* for the classical Fidducia-Mattheyeses (see Section 3.4.2), *MP* for *Max-Pin* and *MN* for *Max-Net* gain function (both described in Section 4.1.3). The hyperedge cut in the column *Avg[%]* of each initial partitioner is relative to the cut produced by *hMetis* and the running times are measured in seconds. We seperate the direct *k-way* and recursive bisection methods in Table 1 and the partitioners within the separation are sorted after their cuts of the first experiment in ascending order.

In the first experiment $(+1, -R, -FM)$ we evaluate the quality of our initial partitioner without any improvement technique. In this test we assign all hypernodes to block V_1 before initial partitioning. Note that only the growing based partitioning algorithms are able to do this. We recognized that nearly all *greedy* implementations are better than the remaining methods. Within the *greedy* variants the recursive bisection implementations based on the *FM* gain function produced the best results. Furthermore the *Max-Pin* performs better than the *Max-Net* gain function. On the second lowest place of the first experiment is the direct *k-way greedy round-robin* algorithm based on the *FM* gain function. If we assign a hypernode to a block in a *greedy* method, we have to delete it from all remaining priority queues. For an explanation of this worse result, we examined the number of hypernodes which are deleted from the priority queues in line 1 of algorithm 10. We observed that the number of deleted hypernodes are two times bigger than the number of the other direct *k-way greedy FM* variants. Therefore many potentially maximum gain moves on top of a priority queue are deleted and the gain of a chosen move is far worse than the gains of the *greedy sequential* or *global* variant. The *random* initial partitioner produces the worst results. Furthermore the recursive bisection variant of *greedy global* and *round-robin* have the same cut results, because we only move hypernodes from V_1 to V_0 on each bisection. Note that in the first experiment all hypernodes are assigned to block V_1 before initial partitioning. Therefore moves to block V_1 are not considered during a bisection (V_1 is disabled) and on this reason the *global* maximum gain move is the same, which is chosen by the *greedy round-robin* partitioner (*greedy-R* ignores block V_1).

The next experiment $(-1, -R, -FM)$ evaluates the quality of the algorithms, if all hypernodes are left unassigned before initial partitioning and no improvement technique is used. Most variants seem to perform worse compared to the results of the first experiment. Except all direct *k-way greedy round-robin* initial partitioners. The worse performance of the most *FM*-based

	Config.	(+1,-R,-FM)		(-1,-R,-FM)		(+1,+R,-FM)		(+1,-R,+FM)		(-1,-R,+FM)		(+1,+R,+FM)	
	Algorithm	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]
	hMetis	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494
	PaToH	3565.27	0.392	3565.27	0.392	3565.27	0.392	3565.27	0.392	3565.27	0.392	3565.27	0.392
Recursive Bisection	greedy-R_{FM}^{RB}	45.44	0.017	92.61	0.021	42.11	0.02	**22.63**	0.023	28.59	0.029	22.72	0.026
	greedy-G_{FM}^{RB}	45.44	0.017	102.2	0.021	42.11	0.02	**22.63**	0.023	29.13	0.029	22.72	0.026
	greedy-S_{FM}^{RB}	45.48	0.017	169	0.019	45.54	0.02	**23.05**	0.023	39.52	0.031	23.07	0.026
	greedy-G_{MP}^{RB}	95.05	0.034	86.88	0.047	91.06	0.037	27.84	0.042	**26.47**	0.054	27.4	0.045
	greedy-R_{MP}^{RB}	95.05	0.018	82.34	0.024	91.06	0.021	27.84	0.026	**27.26**	0.031	27.4	0.029
	greedy-S_{MP}^{RB}	95.61	0.019	94.19	0.023	95.56	0.021	27.67	0.026	**27.33**	0.03	27.62	0.029
	lp^{RB}	90.34	0.049	90.34	0.049	90.34	0.049	**65.12**	0.055	65.12	0.055	65.12	0.055
	bfs^{RB}	113.06	0.01	124.36	0.01	108.92	0.012	29.71	0.018	116.85	0.013	**29.21**	0.021
	greedy-G_{MN}^{RB}	120.04	0.011	110.49	0.015	116.26	0.014	39.46	0.021	**38.04**	0.023	38.66	0.023
	greedy-R_{MN}^{RB}	120.04	0.011	71.41	0.015	116.26	0.014	39.46	0.021	**26.95**	0.022	38.66	0.023
	greedy-S_{MN}^{RB}	120.34	0.011	119.12	0.014	120.34	0.014	39.22	0.02	**38.88**	0.022	39.19	0.023
	$random^{RB}$	242.41	0.005	242.41	0.005	242.41	0.01	**76.85**	0.018	76.85	0.018	76.85	0.024
direct *k-way*	greedy-S_{FM}	53.24	0.016	179.55	0.02	53.24	0.018	42.95	0.019	130.83	0.023	**42.93**	0.021
	greedy-G_{FM}	61.9	0.018	94	0.021	61.15	0.02	50.81	0.024	73.14	0.028	**50.49**	0.026
	greedy-G_{MP}	83.05	0.016	82.35	0.017	82.24	0.018	62.45	0.022	**60.21**	0.023	61.91	0.024
	greedy-S_{MP}	83.56	0.015	82.34	0.015	83.55	0.016	57.22	0.018	**57.07**	0.018	57.18	0.02
	greedy-R_{MP}	84.22	0.018	79.84	0.018	82.43	0.019	62.6	0.02	62.92	0.021	**62.3**	0.022
	lp	86.05	0.033	86.05	0.033	86.05	0.033	**79.94**	0.036	79.94	0.036	79.94	0.036
	greedy-R_{MN}	94.11	0.011	72.68	0.01	92.46	0.012	70.72	0.014	**58.93**	0.013	69.94	0.016
	greedy-G_{MN}	111.49	0.011	111.05	0.011	110.69	0.013	86.49	0.018	**84.35**	0.017	85.95	0.019
	bfs	110.5	0.007	112.94	0.006	108.68	0.008	81.27	0.01	112.94	0.006	**81.24**	0.012
	greedy-S_{MN}	112.65	0.01	111.1	0.01	112.64	0.011	82.05	0.013	82.16	0.013	**82.04**	0.015
	greedy-R_{FM}	121.73	0.021	79.52	0.021	119.58	0.022	105.23	0.024	**62.82**	0.025	106.63	0.026
	random	242.52	0.001	242.52	0.001	242.52	0.003	180.66	0.006	**180.66**	0.006	180.66	0.008

Table 1: Results of our direct *k-way* initial partitioners with different configurations. The *Avg.*[%] column represents the geometric mean overall average cuts relative to the results of *hMetis*. E.g. an value of 45% means that the cut of the corresponding initial partitioner is 45% larger than the result of *hMetis*. The best cut of each initial partitioner is highlighted bold.

partitioners can be explained with the behaviour of the *FM* gain function, if all hypernodes are left unassigned before initial partitioning. We outlined in Section 4.1.3 that in this case a move can never decreases the cut of a partition, because these moves only increase the connectivity of a hyperedge. Most moves are zero gain moves at the beginning and the choice of the next hypernode which should be added to a growing block follows a random fashion. For the significantly better quality of the direct *k-way greedy round-robin* initial partitioner we have only a assumption which is derived from the observation of the first experiment. As mentioned before, if we decide to left all hypernodes unassigned before initial partitioning, the assignment of hypernodes is more random (because of many zero gain moves at the beginning) than if we assign all to a block V_1 before. This leads to a more uniform distribution of assigned hypernodes. In this experiment we observed that the number of deleted hypernodes in algorithm 10 is much less than the same of the *greedy round-robin* methods of the first experiment. We assume that if we move only unassigned hypernodes during partitioning, we have a more evenly distributed assignment of hypernodes and the blocks growing together more slowly. Two blocks V_i and V_j growing together, if $\exists e \in E : \{V_i, V_j\} \subseteq \Lambda(e)$. Therefore we have a significantly lower number of deleted hypernodes into each priority and for this reason the gains of moves are higher compared to the *greedy-R_{FM}* partitioner of the first experiment. Based on this observation we did not

assign all hypernodes before initial partitioning, if we use a direct *k-way greedy round-robin* variant.

In the third column of Table 1 $(+1, +R, -FM)$ we tested the *rollback* technique with the same configuration as in the first experiment. Since we mentioned in Section 4.1.5 that *rollback* only works, if we assign all hypernodes to a block before initial partitioning, we execute this experiment only in the $+1$ configuration. We can compare the results of this configuration with our first experiment $(+1, -R, -FM)$. Nearly all variants benefit from the *rollback* technique. The quality of the recursive bisection variants increased more with this technique as the direct *k-way* initial partitioner. This can be explained that we are able to execute a *rollback* operation after each bisection. A partition of a direct *k-way* initial partitioning method becomes an valid partition only nearly at end of the partitioning process. There are only a small number of nodes which can be used for *rollback*.

In the next two experiments $(+1, -R, +FM)$ and $(-1, -R, +FM)$ we activate our *FM local search* heuristic without *rollback*. This technique improves the quality of the partitions more than the *rollback* technique. We can see again that the recursive bisection variants benefit more from *FM local search* than the direct *k-way* initial partitioning methods. We can execute this technique after each bisection (recursive bisection) instead only at the end of the partitioning process (direct *k-way*). If we rank the initial partitioners again according to their average cut nearly all recursive bisection methods would have a better placement than all direct *k-way* methods. The *FM local search* heuristic improved the average cut of the bfs^{RB} initial partitioner about 83% in comparison to the first experiment $(+1, -R, -FM)$, which is in this experiment $(+1, -R, +FM)$ one of our best partitioning algorithms. The recursive bisection variants of the *greedy* initial partitioner based on the *Max-Pin* and *Max-Net* gain function benefits also from the *FM local search* technique. The quality of both gain functions in the recursive bisection context increases between 70% and 90% in comparison with the results of the first experiment. If we compare the quality of the direct *k-way greedy-R* partitioners of experiment $(+1, -R, +FM)$ and $(-1, -R, +FM)$, we can observe the same behaviour as in the corresponding experiments without the *FM local search* technique. This confirms our decision to left the hypernodes unassigned before initial partitioning, if we run a direct *k-way greedy round-robin* method. The running times of all initial partitioners increase with *FM local search* by a factor of 2 on average. Our best initial partitioners are the recursive bisection variants of the *greedy* algorithms based on the *FM* gain function, if we assign all hypernodes to block V_1 before initial partitioning.

The last experiment uses the *rollback* and *FM local search* technique. Note that we first perform *rollback* and than a *FM local search*. Comparing these results with the same experiment without *rollback*, it can be observed that the results do not differ significantly. This leads to the conclusion that a better initial partition not necessarily leads to a better solution after we execute a *FM local search*. We have shown that the *rollback* technique in combination with a *FM local search* technique did not lead to significantly better results. Therefore we only use the *FM local search* algorithm without *rollback* in the following experiments.

For a conclusion of the first experiments we list the most interesting results at the following:

(i) Recursive bisection methods produce partitions with better quality than the direct *k-way* variants.

(ii) The *FM local search* algorithm improved the quality of partitions of the recursive bisection methods about 25% - 160% and of the direct *k-way* methods about 5% - 60%.

(iii) The direct *k-way greedy round-robin* partitioners performing better, if all hypernodes are unassigned before initial partitioning.

(iv) A *FM local search* heuristic in combination with the *rollback* technique produces equal cuts than without *rollback* on average.

Based on this observations we applied the following adaptions for the next experiments:

(i) We focus on optimizing the recursive bisection methods ⇒ Using only the recursive bisection variants for further experiments.

(ii) If we use a *greedy round-robin* method, we left the hypernodes unassigned before initial partitioning.

(iii) We disable *rollback* and enable the *FM local search* heuristic.

5.2.1 Multiple Runs

The *lp* initial partitioner in the experiment before only produced partitions with mediocre quality. This algorithm has the slowest running time of all partitioners. This is caused by the fact that in each round we have to calculate the gain of a move to all adjacent blocks for each hypernode. Since this partitioner did not assign all hypernodes to a block before initial partitioning, the *rollback* technique cannot be used. The direct *k-way* label propagation initial partitioner seems to not profit very well from the *FM local search* technique. An interesting behaviour of the label propagation initial partitioner is shown in table 2.

Table 2 shows the minimum, average and maximum cuts of each recursive bisection method

Config.	(+1,-R,+FM)			(-1,-R,+FM)		
Algorithm	Min.[%]	Avg.[%]	Max.[%]	Min.[%]	Avg.[%]	Max.[%]
hMetis	1029.69	1099.11	1170.04	1029.69	1099.11	1170.04
PaToH	1053.49	1131.04	1201.88	1053.49	1131.04	1201.88
bfs^{RB}	3.29	29.79	97.33	79.03	170.15	281.13
$\text{greedy-S}_{MN}^{RB}$	4.02	37.62	88.9	3.24	37.23	85.4
lp^{RB}	4.15	56.3	160.03	4.15	56.3	160.03
$\text{greedy-S}_{MP}^{RB}$	4.42	23.36	59.57	4.4	22.39	60.74
$\text{greedy-G}_{MP}^{RB}$	5.06	23.4	57.81	4.22	21.41	55.62
$\text{greedy-R}_{MP}^{RB}$	5.06	23.4	57.81	4.29	23.94	74.52
$\text{greedy-G}_{MN}^{RB}$	5.76	37.67	85.66	5.24	34.07	85.77
$\text{greedy-R}_{MN}^{RB}$	5.76	37.67	85.66	3.3	23.26	83.46
$\text{greedy-R}_{FM}^{RB}$	6.69	34.84	71.89	3.88	25.37	76.58
$\text{greedy-G}_{FM}^{RB}$	6.69	34.84	71.89	3.82	27	73.16
$\text{greedy-S}_{FM}^{RB}$	7.19	34.44	80.27	3.26	38.9	108.47
random^{RB}	7.66	66.79	139.3	7.66	66.79	139.3

Table 2: Min., Avg. and Max. cuts of our initial partitioners for $k = 2$ with a *FM local search* algorithm. The cut of each initial partitioner is relative to the cut of *hMetis*.

relative to the corresponding result of *hMetis* for $k = 2$. We take the results from the *FM local search* experiments $(+1, -R, +FM)$ and $(-1, -R, +FM)$. The partitioners are sorted according to their minimum cut in ascending order. The label propagation initial partitioner has the biggest range between the minimum and maximum cut. In general we can observe that the minimum and maximum cut of each initial partitioner significantly differ and the minimum cuts come very close to the minimum cut of *hMetis*. We want to show with this table that we

have to introduce for our recursive bisection methods multiple runs of each initial partitioner on each bisection to converge against the minimum cut.

	random ^{RB}		bfs^{RB}		greedy-S$^{RB}_{FM}$		greedy-G$^{RB}_{FM}$		greedy-R$^{RB}_{FM}$		greedy-S$^{RB}_{MP}$	
runs	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]
hMetis	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494
1	76.85	0.038	29.71	0.034	23.05	0.045	22.63	0.045	28.59	0.055	27.67	0.050
2	66.09	0.056	19.73	0.050	15.24	0.071	15.28	0.071	18.69	0.092	20.28	0.081
5	57.05	0.102	12.82	0.095	10.62	0.143	10.87	0.144	10.65	0.194	15.20	0.167
10	52.12	0.175	9.19	0.164	8.36	0.260	8.36	0.260	7.20	0.356	13.05	0.305
15	49.29	0.246	7.78	0.229	7.34	0.369	7.23	0.372	5.75	0.511	12.09	0.437
20	47.57	0.313	6.88	0.294	6.68	0.476	6.67	0.481	4.94	0.663	11.56	0.565
25	46.25	0.380	6.33	0.359	6.36	0.588	6.16	0.594	4.43	0.818	11.16	0.695
50	42.34	0.718	4.90	0.677	5.24	1.118	5.12	1.133	3.11	1.581	10.15	1.339
75	40.34	1.057	4.22	0.997	4.72	1.652	4.59	1.668	2.53	2.335	9.66	1.981

	greedy-G$^{RB}_{MP}$		greedy-R$^{RB}_{MP}$		greedy-S$^{RB}_{MN}$		greedy-G$^{RB}_{MN}$		greedy-R$^{RB}_{MN}$		lp^{RB}	
runs	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]	Avg.[%]	t[s]
hMetis	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494	3518.88	2.494
1	27.84	0.050	27.26	0.059	39.22	0.040	39.46	0.039	26.95	0.038	65.12	0.095
2	20.51	0.082	17.49	0.100	31.51	0.063	32.42	0.061	16.87	0.061	46.37	0.154
5	15.82	0.171	11.22	0.211	26.66	0.121	27.68	0.119	10.79	0.118	28.23	0.333
10	13.36	0.310	8.10	0.388	24.07	0.216	24.72	0.209	7.79	0.207	19.92	0.624
15	12.38	0.443	6.96	0.559	23.15	0.309	23.51	0.296	6.76	0.294	16.47	0.911
20	11.77	0.574	6.23	0.728	22.53	0.396	22.91	0.381	6.05	0.378	14.51	1.201
25	11.44	0.709	5.85	0.891	21.93	0.487	22.41	0.468	5.63	0.464	12.81	1.503
50	10.35	1.357	4.63	1.719	20.60	0.924	21.19	0.887	4.62	0.881	9.14	2.951
75	9.75	2.012	4.10	2.550	20.00	1.360	20.40	1.305	4.14	1.300	7.58	4.317

Table 3: Results of the multiple runs experiment of our recursive bisection variants. The cuts are relative to the results of hMetis.

In Table 3 we see the convergence behavior of each recursive bisection variant, if we use multiple runs on each bisection. The best result is shown by greedy-R$^{RB}_{FM}$ which is only 2.53% worse than *hMetis* with 75 runs on each bisection. Nearly all partitioners seem to converge against the *minimum cut*, which is presented in table 2 except the lp^{RB} initial partitioner. This is caused by the huge difference of its minimum and maximum cut. Note, we execute this experiment in parallel on several systems. The running times are only approximate values in Table 3. The running time of each partitioner increases linear with the number of repetitions. For the final choice of the multiple runs parameter, we have to evaluate the running time and quality of our initial partitioners in the recursive bisection n-level context. Furthermore we have to take care that the average running time of our final recursive bisection n-level initial partitioner, which we integrated into *KaHyPar*, did not exceed the average running time of *hMetis*.

5.2.2 Adaptive Epsilon

At the end of this section we want to show that in practice our partitions are balanced, if we adapt the initial imbalance ε on each bisection to ε' (see Section 4.2.1). We take the imbalance values of our first experiment $(+1, -R, -FM)$ without an execution of a *FM local search* algorithm and *rollback*. Table 4 shows the average and maximum imbalance of each tested hypergraph with input $\varepsilon = 0.03$. The last column shows how many test runs produced an imbalanced partition. For each hypergraph we execute 7200 test runs (6 k values $\cdot$ 10 different coarsened instances $\cdot$ 5 different seeds $\cdot$ 24 initial partitioners $= 7200$). Only 32 partitions of 93600 have an imbalance greater than $\varepsilon = 0.03$. If we have look at the maximum imbalance

values this is only caused by the fact that in some cases a hypernode with maximum weight has to be assigned at the end of the partitioning process.

Hypergraph	Avg. imbalance	Max. imbalance	count
bcsstk29	0.023	0.03	0
bcsstk30	0.024	0.03	0
bcsstk31	0.024	0.03	0
bcsstk32	0.024	0.03	0
ibm01	0.024	0.03	2
ibm02	0.024	0.029	0
ibm03	0.025	0.033	8
ibm04	0.025	0.032	4
ibm05	0.024	0.029	0
ibm06	0.025	0.031	2
ibm07	0.025	0.033	4
memplus	0.024	0.032	12
vibrobox	0.023	0.03	0

Table 4: Average and maximum imbalance of our first experiment without refinement and rollback with $\varepsilon = 0.03$. The last coloumn shows the amount of imbalanced hypergraphs. On each hypergraph we perform 7200 test runs.

5.3 Recursive Bisection n-Level Initial Partitioner

In this section we evaluate the optimum parameter configuration for our recursive bisection n-Level initial partitioner (*RBNL*). We conclude this chapter with the results of our final initial partitioner on the *medium sized* instances in comparison with *hMetis* and *PaToH*. Since the recursive bisection algorithms produce better results than the direct *k-way* methods we continue only with the *RB* methods. Furthermore we include our *pool* initial partitioner into the experiments, since this partitioner is a result of our n-level development process. We used the *basic* algorithm set in our *pool* initial partitioner for the following experiments, which is shown in Table 8.

5.3.1 Evaluation of the Coarsening Configuration

In this experiment we want to find the optimum parameter setting for the contraction limit t and the hypernode weight limit s for the *coarsening* phase of our *RBNL* initial partitioners. We contract the hypergraph in the coarsening process at each bisection until $2t$ hypernodes are left and no hypernode v resulting from a contraction should be heavier than $c_{max} := s \cdot \frac{c(V)}{2t}$. Since the parameters s and t correlate with each other we have to test several combinations. The testing parameters are $s \in \{1.0, 1.5, 2.0, 2.5, 3.0, 3.5\} =: S$ and $t \in \{25, 50, 75, 100, 125, 150, 200, 250, 300\} =: T$. We test each possible combination $\Rightarrow$ Test-Set $M_{st} = \{(s,t) \in S \times T\}$. Further we execute each initial partitioner 20 times on each bisection. The *pool* partitioner executes each algorithm in its initial partitioner set 20 times on each bisection, too. In the *uncoarsening* phase we used a *FM local search* heuristic.
The heatmap in Figure 17 shows the result of the experiment for our *RBNL* pool initial partitioner. The brighter the color in the heatmap, the better the quality of results. The best results are archived between $t = 150$ and $t = 200$. If we set $s = 2.5$ with $t = 150$ or $s = 3.5$ with $t = 200$ we can reach the optimum results. The smaller our contraction limit, the worse is

the result of the quality of the partitions. If we use a small value for t, the initial partitioning is less important, because the coarsened hypergraph is very small and the *local search* heuristic changes the solution during *uncontraction* much more than with a larger value for t. Further the weight of the hypernodes would become heavier, if t is small. Therefore the initial partitioning becomes harder, because we have to take care that we fulfil the imbalance. To choose a parameter assignment for s and t we have to analyse the results of the other initial partitioning methods in the recursive bisection n-Level context.

The heatmap in Figure 18 shows the result of our *RBNL greedy global FM* initial partitioner.

Figure 17: Results of the experiment with parameter s and t of our *RBNL* pool initial partitioner.

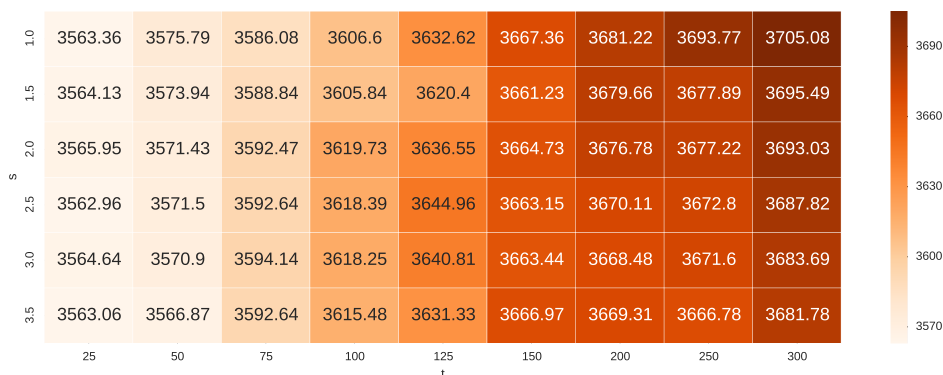


Figure 18: Results of the experiment with parameter s and t of our *greedy global FM* initial partitioner with 20 runs at each bisection step.

All other algorithms produced similiar results (except the pool initial partitioner). If we interpret the results it seem to be advantageous to use the smallest t value for this initial partitioning method to archieve the best results. On the first view this seem to conflict with the results in Figure 17. As mentioned before using a small value for t means to let the local search heuristic during the *refinement* phase do most of the partitioning work. The initial partitioning result is then less important. Further we destroy with a small t value a large part of the structure of the hypergraph and therefore limit the quality of the solution. We assume that the t value, where the RBNL method of an initial partitioner produces the best results, is a measure for the quality of an initial partitioning method. If the best cut result in the RBNL context is archieved with a large t value, the quality of the produced partitions of the corresponding initial partitioner is very well. If we have a good initial partitioning algorithm we can take away the work from the local search algorithm during the *uncontraction* phase. We can show this behavior, if we

execute the same experiment with the *RBNL greedy global FM* initial partitioner with 100 runs on each bisection instead of 20. The result of this experiment is shown in the heatmap in Figure 19. Note, if we execute the initial partitioner with 100 runs on each bisection the quality of the partitions are better than with only 20 runs (see Table 3). In this experiment with 100 runs the optimum t value moves from $t = 25$ (with 20 runs) to $t = 50$ and $t = 75$ (with 100 runs). This verifies our assumption that the optimum t value, where the initial partitioner in the RBNL context produces the best results, is a measure for the quality of the initial partitioner.

We can conclude that all initial partitioning methods (except the pool initial partitioner) are

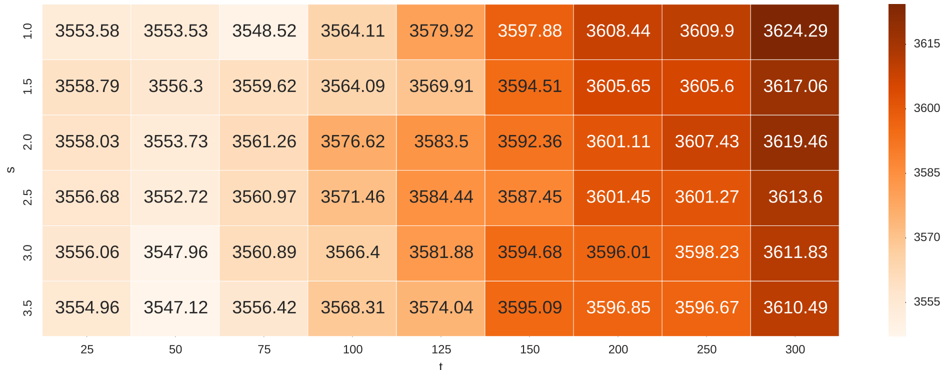


Figure 19: Results of the experiment with parameter s and t of our *greedy global FM* initial partitioner with 100 runs at each bisection step.

not strong enough to allow a greater t value than 50. If we use our pool initial partitioner, we have a method which is strong enough. It is desirable to choose t greater than 50, because we did not destroy the structure of a hypergraph as much as with $t \leq 50$. Furthermore we can take away the work from the *local search* algorithm during the *refinement* phase. The only initial partitioning method which allows a greater t value than 50 is the *pool* partitioner. It seems that this method becomes our initial partitioner, which we want to integrate in *KaHyPar*. Therefore we set $t = 150$ and $s = 2.5$. We choose $s = 2.5$, because if we use a greater value for s the contracted hypernodes become heavier and this makes it harder to fulfil the imbalance.

5.3.2 Evaluation of the *Local Search* Configuartion

In the next experiment we evaluate which *local search* heuristic we have to choose for the *refinement* phase of our recursive bisection n-level initial partitioner. The framework currently supports a *FM local search* and a *ScLap*-based algorithm. The *FM* local search is based on the Fiduccia-Matheysses algorithm described in section 3.4.2. For detail implementation details we refer the reader to [13]. The basic idea of *Label Propagation* is described in section 3.3.4 and the adaption to use it as a *local search* heuristic is described by Henne [12]. The *ScLap*-based refinement algorithm is two times faster than the *FM* local search algorithm. Table 5 shows the minimum and average cuts of our recursive bisection n-level initial partitioner in comparison with *hMetis* and *PaToH*. We use 20 runs of each initial partitioner on each bisection. The contraction limit of our n-Level recursive bisection initial partitioner is set to $t = 160$. Further we use 20 fruitless moves of our *FM* local search algorithm (see Section 4.3) and the maximum number of iterations of the *ScLap*-based algorithm is set to 3. The *Pool* initial partitioner executes each partitioner in his algorithm set 20 times. Both columns of the table are sorted according to the average cut of each partitioner and the resulting cuts of each initial partitioner

is relative to the cut of the partitioner which produces the best results.
First we note that our *RBNL* pool initial partitioner with the *FM* local search heuristic produces in average 0.73% better cuts than *hMetis* on the *medium sized* benchmark set. The minimum cut is also 1% better than the cut produced by *hMetis*. With the *ScLap*-based algorithm the result of our *RBNL* pool initial partitioner is 2.7% worse than *hMetis*. The pool initial partitioner combines all advantages of each developed partitioning algorithm. If one partitioner produces worse results another method working better. The second best variant is our *RNBL bfs* initial partitioner with the *FM* local search algorithm. All variants, except the *random* initial partitioner, produces average cuts which are only 10% worse than the cuts of *hMetis*, when used together with *FM*. If we have a look at the same results of the *ScLap*-based *local search* algorithm the most partitioners have average cuts which are more than 10% worse than the cuts of *hMetis*. The advantage of the *ScLap*-based algorithm is the running time, but our first goal is to optimize the quality and we decided to use the *FM* local search algorithm.

FM local search			ScLap-based local search		
Algorithm	Min.[%]	Avg.[%]	Algorithm	Min. [%]	Avg.[%]
pool	3362.5	3493.23	hMetis	3396.49	3518.88
hMetis	1.01	0.73	PaToH	1.33	1.31
PaToH	2.36	2.06	pool	1.14	2.7
bfs^{RB}	1.29	2.25	greedy-R$^{RB}_{FM}$	3.01	5.82
greedy-R$^{RB}_{FM}$	1.35	2.4	greedy-R$^{RB}_{MP}$	5.73	8.86
greedy-R$^{RB}_{MP}$	2.16	3.17	greedy-R$^{RB}_{MN}$	5.1	10.02
greedy-R$^{RB}_{MN}$	1.86	3.31	greedy-G$^{RB}_{FM}$	5.35	10.21
lp^{RB}	1.75	3.39	lp^{RB}	5.27	10.37
greedy-G$^{RB}_{FM}$	1.87	4.85	greedy-S$^{RB}_{FM}$	5.22	10.49
greedy-S$^{RB}_{FM}$	1.93	4.86	bfs^{RB}	6.01	11.2
greedy-G$^{RB}_{MP}$	3.04	5.09	greedy-S$^{RB}_{MP}$	10.66	16.17
greedy-S$^{RB}_{MP}$	3.27	5.1	greedy-G$^{RB}_{MP}$	11.41	17.22
greedy-S$^{RB}_{MN}$	5.27	10.27	greedy-S$^{RB}_{MN}$	22.79	35.07
greedy-G$^{RB}_{MN}$	5.46	10.48	greedy-G$^{RB}_{MN}$	23.95	36.74
random^{RB}	8.26	15.92	random^{RB}	34.65	48.75

Table 5: Results of our experiments with the *FM* and *ScLap*-based *local search* heuristic. The cut values are relative to best cut in the first row of the table. Both Columns of the table are sorted after the average cut in ascending order.

In the next experiment we evaluate the maximum number of fruitless moves f of our *FM local search* algorithm during the *refinement* phase. This parameter limits the maximum number of moves which the *FM* heuristic can make without improving the cut. If we reach f fruitless moves after one *uncontraction* step the *FM* algorithm stops and performs a *rollback* operation to best cut. The results of this experiment are summarized in Table 6 and 7. Table 6 presents the cut of each *RBNL* initial partitioner for $f \in \{5, 10, 20, 30, 40, 50, 75, 100, 150, 200, 300\}$. The results are sorted according to the cut with $f = 5$ in ascending order. All cut values are relative to the result of the *RBNL* pool initial partitioner. Table 7 shows the running times of each initial partitioner. Since we execute this experiment in parallel on several systems these values

are only approximate values.

We notice that the quality of each partitioner increase significantly until $f = 50$. Note that

	$f=5$	$f=10$	$f=20$	$f=30$	$f=40$	$f=50$	$f=75$	$f=100$	$f=150$	$f=200$	$f=300$
Algorithm	Avg.[%]	Avg.[%]	Avg.[%]	Avg.[%]	Avg.[%]	Avg.[%]	Avg.[%]	Avg.[%]	Avg.[%]	Avg.[%]	Avg.[%]
pool	3520.43	3502	3493.23	3491.02	3490.02	3489.56	3488.82	3488.54	3488.55	3487.85	3486.02
greedy-R^{RB}_{FM}	2.62	2.53	2.4	2.36	2.32	2.31	2.3	2.28	2.28	2.27	2.22
bfs^{RB}	3.18	2.6	2.25	2.14	2.09	2.07	2.04	2.02	2	1.98	1.9
greedy-R^{RB}_{MP}	4.3	3.75	3.17	2.99	2.91	2.85	2.81	2.78	2.75	2.73	2.56
greedy-R^{RB}_{MN}	4.36	3.65	3.31	3.14	3.08	3.03	2.96	2.94	2.92	2.89	2.69
greedy-G^{RB}_{FM}	6.71	5.76	4.85	4.41	4.17	4.03	3.68	3.53	3.31	3.18	2.78
greedy-S^{RB}_{FM}	6.96	5.86	4.86	4.47	4.16	3.94	3.68	3.51	3.19	3.12	2.73
greedy-S^{RB}_{MP}	7.69	6.25	5.1	4.62	4.35	4.24	4.07	3.98	3.91	3.86	3.63
greedy-G^{RB}_{MP}	7.73	6.35	5.09	4.67	4.46	4.33	4.16	4.06	4	3.92	3.65
lp^{RB}	9.75	7.49	4.86	3.73	3.25	2.99	2.79	2.71	2.69	2.67	2.52
greedy-S^{RB}_{MN}	15.64	12.4	10.27	9.15	8.51	8.22	7.51	7.22	6.74	6.41	5.85
greedy-G^{RB}_{MN}	16.14	12.64	10.48	9.39	8.82	8.47	7.93	7.5	6.97	6.55	6.03
random^{RB}	23.02	18.94	15.92	14.51	13.46	12.86	11.92	11.45	10.91	10.49	9.22

Table 6: Cut values of our *RBNL* initial partitioners with different maximum numbers of fruitless moves.

	$f=5$	$f=10$	$f=20$	$f=30$	$f=40$	$f=50$	$f=75$	$f=100$	$f=150$	$f=200$	$f=300$
Algorithm	t[s]	t[s][%]	t[s][%]	t[s][%]	t[s][%]	t[s][%]	t[s][%]	t[s][%]	t[s][%]	t[s][%]	t[s][%]
bfs^{RB}	0.4	0.44	0.5	0.56	0.61	0.67	0.8	0.94	1.21	1.48	1.99
random^{RB}	0.41	0.45	0.52	0.58	0.64	0.69	0.83	0.96	1.23	1.48	1.96
greedy-R^{RB}_{MN}	0.45	0.49	0.55	0.61	0.67	0.72	0.86	1	1.28	1.55	2.08
greedy-S^{RB}_{MN}	0.46	0.5	0.57	0.63	0.69	0.74	0.88	1.02	1.3	1.57	2.08
greedy-G^{RB}_{MN}	0.46	0.5	0.57	0.63	0.69	0.74	0.89	1.02	1.3	1.57	2.08
greedy-S^{RB}_{FM}	0.48	0.52	0.6	0.66	0.72	0.78	0.92	1.06	1.33	1.61	2.13
greedy-G^{RB}_{FM}	0.49	0.53	0.6	0.66	0.72	0.78	0.92	1.06	1.34	1.61	2.13
greedy-S^{RB}_{MP}	0.51	0.55	0.62	0.69	0.74	0.79	0.94	1.08	1.34	1.62	2.16
greedy-G^{RB}_{MP}	0.51	0.56	0.62	0.69	0.74	0.8	0.94	1.07	1.35	1.63	2.16
greedy-R^{RB}_{FM}	0.58	0.62	0.69	0.75	0.81	0.86	1.01	1.15	1.43	1.71	2.26
greedy-R^{RB}_{MP}	0.58	0.62	0.68	0.75	0.81	0.86	1.01	1.16	1.43	1.71	2.24
lp^{RB}	0.81	0.83	0.9	0.98	1.06	1.13	1.31	1.49	1.84	2.21	2.86
pool	1.82	1.93	2.06	2.19	2.29	2.4	2.68	2.96	3.51	4	4.8

Table 7: Running times of our *RBNL* initial partitioners with different maximum numbers of fruitless moves.

our bfs^{RB} n-level initial partitioner becomes our second best method for $f \geq 20$. The running times of each partitioner in Table 7 increase near linearly. If we increase the maximum number of fruitless moves from $f = 20$ to $f = 50$, the running times of each initial partitioner would increase about $20\% - 35\%$. For the final assignment of parameter f we have to make a trade off between running time and quality. The quality of the cut of each initial partitioner increases only until $f = 50$ significantly. With the focus that our *pool* initial partitioner becomes the final initial partitioner, we choose therefore $f = 50$ as our final maximum number of fruitless moves.

5.3.3 Composition of the *Pool* Initial Partitioner

The *pool* initial partitioner produced the best results in the *RBNL* context. Since this partitioner executes a subset of our developed initial partitioning algorithms and take the best result from all executions, we have to find the best subset of algorithms for it. Table 8 shows the results of our experiment with different compositions of subsets. The composition *full* with all initial partitioning algorithms produces the best quality. The *basic* set is the composition we used in the experiment before. The running times and values indicates that the quality of the pool initial partitioner depends on the number of different initial partitioning algorithms which are used. The final composition of our *RBNL pool* initial paritioner is the *mix3* set. The quality of this variant is close to *full* and the running time is about 20% faster.

In the last experiment we evaluate the number of multiple runs, which each initial partitioner

Pool-Type	Avg.[%]	$\frac{Pool-Type\ t[s]}{full\ t[s]}$	Composition
full	3476.84	1	greedy-R^{RB}_{FM}, greedy-G^{RB}_{FM}, greedy-S^{RB}_{FM}, greedy-R^{RB}_{MP}, greedy-G^{RB}_{MP}, greedy-S^{RB}_{MP}, greedy-R^{RB}_{MN}, greedy-G^{RB}_{MN}, greedy-S^{RB}_{MN}, bfs^{RB}, lp^{RB}, $random^{RB}$
mix3	3479.6	0.79	greedy-R^{RB}_{FM}, greedy-S^{RB}_{FM}, greedy-R^{RB}_{MP}, greedy-G^{RB}_{MP}, greedy-R^{RB}_{MN}, greedy-G^{RB}_{MN}, bfs^{RB}, lp^{RB}, $random^{RB}$
mix1	3482.03	0.78	greedy-G^{RB}_{FM}, greedy-S^{RB}_{FM}, greedy-R^{RB}_{MP}, greedy-S^{RB}_{MP}, greedy-R^{RB}_{MN}, greedy-S^{RB}_{MN}, bfs^{RB}, lp^{RB}, $random^{RB}$
greedy_full	3482.09	0.81	greedy-R^{RB}_{FM}, greedy-G^{RB}_{FM}, greedy-S^{RB}_{FM}, greedy-R^{RB}_{MP}, greedy-G^{RB}_{MP}, greedy-S^{RB}_{MP}, greedy-R^{RB}_{MN}, greedy-G^{RB}_{MN}, greedy-S^{RB}_{MN}
basic	3489.56	0.63	greedy-G^{RB}_{FM}, greedy-S^{RB}_{FM}, greedy-G^{RB}_{MP}, greedy-G^{RB}_{MN}, bfs^{RB}, lp^{RB}, $random^{RB}$
mix2	3490.94	0.58	greedy-R^{RB}_{FM}, greedy-G^{RB}_{MP}, greedy-S^{RB}_{MN}, bfs^{RB}, lp^{RB}, $random^{RB}$
adaptive	3495.17	0.57	Contains the *full* set and choose at each partitioning a subset of six initial partitioner random.
greedy	3510.53	0.39	greedy-R^{RB}_{FM}, greedy-G^{RB}_{FM}, greedy-S^{RB}_{FM}
no_greedy	3526.6	0.35	bfs^{RB}, lp^{RB}, $random^{RB}$
greedy_maxpin	3533.51	0.39	greedy-R^{RB}_{MP}, greedy-G^{RB}_{MP}, greedy-S^{RB}_{MP}
greedy_maxnet	3537.74	0.34	greedy-R^{RB}_{MN}, greedy-G^{RB}_{MN}, greedy-S^{RB}_{MN}

Table 8: Experiment results with different compositions of the pool initial partitioner.

of the *pool* partitioner algorithm subset (*mix3* variant, see Table 8) should be executed on each bisection. Table 9 shows the results of this experiment. We choose for our final partitioner 20 runs. The quality of the partitions is only 0.32% worse than with 75 runs and the running time is 3 times faster with 20 runs. Furthermore with 20 runs we are still 20% faster than *hMetis* (2.49s). Note, that the cut values did not accord with the cut values in Table 8, because the code base changed between these two experiments.

runs	Min.[%]	Avg.[%]	t[s]
75	3356.688	3484.424	6.53
50	+0.003	+0.06	4.51
40	+0.17	+0.14	3.7
30	+0.14	+0.19	2.89
20	+0.17	+0.32	2.07
10	+0.21	+0.54	1.24

Table 9: Multiple runs experiment with our RBNL *pool* initial partitioning algorithm with the *mix3* composition.

5.3.4 Experimental Results of the Final Initial Partitioner

In this subsection we evaluated the best configuration of our recursive bisection n-level initial partitioner with several experiments. In the following we list the main adaptions:

(i) *Coarsening* phase: Contraction Limit $t = 150$ and hypernode contraction weight limit $s = 2.5$.

(ii) *Initial Partitioning* phase: As final initial partitioner for our RBNL partitioner we use the *pool* initial partitioner with the *mix3* composition (see Table 8). We execute each algorithm of the *mix3* set 20 times on each bisection and take the best from all runs as final partition.

(iii) *Uncoarsening* phase: We use a *FM local search* heuristic during the uncontraction phase and set the maximum number of fruitless moves to $f = 50$.

We conclude this chapter with the final results of our recursive bisection n-Level initial partitioner with all adaptions described in this section in comparison with *hMetis* and *PaToH* on the coarsened instances of the *medium sized* benchmark set in Table 10 and 11. Our *RBNL* initial partitioner produces 1.13% better cuts in average than *hMetis* and the runtime is about 20% faster. For all k we produce better minimum and average cuts as *hMetis* and *PaToH* except for $k = 2$ where *hMetis* produces 0.18% better cuts in average.

	All Benchmarks			VLSI Benchmarks			SPM Benchmarks		
Algorithm	Min.	Avg.	t[s]	Min.	Avg.	t[s]	Min.	Avg.	t[s]
KaHyPar-Pool	3354.82	3479.6	2.09	2816.18	2907.87	2.35	3897.85	4070.47	1.56
hMetis	+1.24	+1.13	2.49	+0.76	+0.45	3.05	+1.96	+2.16	1.61
PaToH	+2.6	+2.46	0.39	+2.19	+1.9	0.45	+3.09	+3.15	0.3

Table 10: Result of our final initial partitioner on the coarsened instances of the *medium sized* benchmark set.

	KaHyPar			hMetis			PaToH		
k	Min.	Avg.	t[s]	Min.	Avg.	t[s]	Min.	Avg.	t[s]
2	1031.24	1101.19	0.34	**1029.69**	**1099.11**	0.24	1053.49	1131.04	0.08
4	**2086.75**	**2216.44**	0.87	2133.78	2236.13	0.77	2151.2	2273.64	0.19
8	**3375.6**	**3493.35**	1.75	3444.04	3583.31	1.93	3478.19	3588.59	0.32
16	**4704.33**	**4804.84**	3.14	4737.98	4869.03	4.31	4797.94	4897.67	0.57
32	**5882.26**	**5999.77**	5.41	5947.57	6059.23	8.82	6014.21	6122.68	0.89
64	**7092.36**	**7221.06**	9.27	7199.91	7307.08	18.19	7309.33	7421.74	1.44

Table 11: Result of our final initial partitioner on the coarsened instances of the *medium sized* benchmark set for each k.

5.4 Experimental Results of our Initial Partitioner in *KaHyPar*

We integrate our initial partitioning framework into *KaHyPar* and compare our best (*Pool* initial partitioner) and second best initial partitioner (*BFS* initial partitioner) with *hMetis* and *PaToH* as initial partitioner in this framework. We evaluate the results on the *full* benchmark set. Table 12 and 13 show the results of the n-Level hypergraph partitioning framework *KaHyPar* with different initial partitioners.

In table 12 we can see the results of *KaHyPar* on the overall benchmark set. Our initial

	All Benchmarks			VLSI Benchmarks			SPM Benchmarks			Large Benchmarks		
Initial Partitioner	Min.	Avg.	t[s]	Min	Avg.	t[s]	Min.	Avg.	t[s]	Min.	Avg.	t[s]
hMetis	7937.32	**8110.24**	3.42	**4506.62**	**4615.9**	4.35	13979.69	14249.87	2.69	**12812.13**	**13086.39**	3.93
KaHyPar-Pool	**7936.92**	8118.07	2.94	4511.43	4627.93	3.43	**13963.38**	**14240.28**	2.52	12841.49	13135.2	3.51
PaToH	7973.8	8198.2	0.63	4529.33	4685.88	0.63	14037.7	14343.23	0.64	12902.29	13254.4	0.85
KaHyPar-BFS	8321.3	8938.15	0.45	4794.89	5174.52	0.6	14441.22	15439.22	0.34	13437.26	14513.6	0.53

Table 12: Result of *KaHyPar* on the full benchmark set with different initial partitioning algorithms. The column t[s] shows the running time of the initial paritioners.

	KaHyPar-Pool			hMetis			PaToH			KaHyPar-BFS		
k	Min.	Avg	t[s]	Min.	Avg.	t[s]	Min.	Avg	t[s]	Min.	Avg	t[s]
2	1650.14	**1701.07**	0.31	1653.69	1703.19	0.23	1647.86	1712.91	0.11	1666.48	1882.64	0.01
4	3534.75	3651.96	0.83	3524.77	**3650**	0.66	3522.35	3692.98	0.21	3651.483	4197.328	0.08
8	6032.93	6230.13	1.72	6048.9	**6219.74**	1.68	6081.94	6303.51	0.38	6329.843	6985.93	0.24
16	9303.81	9535.33	3.27	9249.73	**9472.36**	3.94	9325.06	9597.27	0.67	9835.25	10492.19	0.61
32	13291.17	13524.3	6.06	13288.52	**13495.93**	8.68	13355.2	13631.6	1.13	14085.37	14603.72	1.4
64	18486.78	18713.81	10.87	18462.37	**18671.37**	18.48	18674.79	18934.47	1.91	19674.55	20154	2.92
128	24664.56	**24877.79**	19.69	24806.41	25007.44	33.41	24963.45	25199.37	3.14	26315.82	26734.76	6.03

Table 13: Result of *KaHyPar* on the full benchmark set for each k with different initial partitioning algorithms. The column t[s] shows the running time of the initial paritioners.

partitioner produces results comparable to those of *hMetis* (0.09% worse). We archieve 1% better results than *PaToH*. On the sparse matrix instances our initial partitioner produces the best solutions. Also the running time is about 16.33% faster than *hMetis*. With our *Pool* initial partitioner *KaHyPar* archieves the best minimum cut in comparison with *hMetis* and *PaToH*.

If we have a look on Table 13, we can see that the *KaHyPar-Pool* initial partitioner computes

	All Benchmarks			
Initial Partitioner	Init. Min.	Min.	Init Avg.	Avg.
KaHyPar-Pool	**10655.34**	**7936.92**	**10957.09**	8118.07
hMetis	10738.71	7937.32	11024.4	**8110.19**

Table 14: Initial cuts (Init. Min. & Avg.) and final cuts of *KaHyPar* with *hMetis* and the *Pool* partitioner as initial partitioner on the full benchmark set.

the best partitions for $k = 2$ and $k = 128$. Our cuts are 0.52% better for $k = 128$ than the cuts

of *hMetis* and 1.28% better than *PaToH*. On maximum we are only 0.66% worse than *hMetis* for $k = 16$. We produce the best minimum cuts for $k = 8$ and $k = 128$.
Table 14 shows the average initial partitioning cuts of our *Pool* initial partitioner and *hMetis* in comparison with the final cuts of *KaHyPar* on the *full* benchmark set. We can see that the *Pool* initial partitioner produces 0.61% better initial cuts as *hMetis*, but the final cuts of *KaHyPar* are nearly the same as with *hMetis*.
The goal of this work was to replace *hMetis* as initial partitioner in *KaHyPar* with an own initial partitioner which produces the same results in the same amount of time. With this final experiment we have shown that our initial partitioner archieves the same results in the n-Level context as *hMetis* and is about 16% faster.

6 Conclusion

In this thesis we developed several direct *k-way* and recursive bisection initial partitioning methods for the n-level hypergraph partitioning framework called *KaHyPar*. We implemented several standard partitioning techniques which are the BFS, random and greedy hypergraph growing initial partitioners. We introduce three different greedy hypergraph growing implementation techniques and provide three different gain functions. Further we adapt the idea of *Label Propagation* to a working initial partitioning algorithm. Also we developed a *Recursive Bisection* implementation which can use one of our implemented partitioning algorithms to produce bisections. We work out a new imbalance definition for our *recursive bisection* method, which allows us to calculate an ϵ' before each bisection to ensure that every resulting block at the end of that process fulfil our predefined imbalance with the initial imbalance ϵ. The advantage is that we do not restrict the solution space on each bisection that much as if we use a uniform ϵ' for each bisection. At the end we use the existing n-level hypergraph partitioning framework *KaHyPar* to develop a recursive bisection n-level initial partitioner which uses one of our provided algorithms as initial partitioner. For the n-level context we implemented the *pool* initial partitioner which executes a subset of our developed initial partitioning methods and chooses the best partition of all executed variants as final partition. We extensively evaluate different configurations of our methods. We finally choose the *pool* initial partitioner for our recursive bisection n-Level partitioner.
We integrate our initial partitioner into *KaHyPar* and compare the results of the overall framework with our partitioner against the usage of *hMetis* and *PaToH* as initial partitioner. For the comparison we use the recursive bisection variants of *hMetis* with default parameters and *PaToH* with the default quality preset.
Our initial partitioner produce 0.61% better initial cuts on average than *hMetis* on the *full* benchmark instances and is about 16% faster. *KaHyPar* produces comparable cuts with our initial partitioner on average as with *hMetis* and 1% better cuts as *PaToH* as initial partitioner. Also *KaHyPar* generates the best solutions on average for $k = 2$ and $k = 128$ with our initial partitioner and produces the best minimum cuts. On the sparse matrix instances *KaHyPar* works best with the *pool* initial partitioner.

6.1 Future Work

During our work on the initial partitioning framework for *KaHyPar* we find several interesting approaches and adaptions in different areas.
We have thought about a new variant of recursive bisection where we choose a huge ϵ' at beginning and decide based on the weight of the resulting blocks in how many parts we further divide the resulting parts. This approach has several advantages. First we do not restrict our ϵ' as much as with the *adaptive epsilon* ε'. This leads to a greater exploration of the solution space. Furthermore, the *rollback* technique gains more importance, because with a large ϵ' we can explore more feasible solutions during the bisection process. After a subhypergraph is bisected, we have to decide in how many parts we further divide the resulting blocks V_1 and V_2. For this decision we define the partition weight bound for a single block $w_{max} = \frac{c(V)}{k}(1+\epsilon)$. We further divide block V_1 in $k_1 = \min\{i \mid c(V_1) \leq i \cdot w_{max}\}$ and V_2 in $k_2 = \min\{i \mid c(V_2) \leq i \cdot w_{max}\}$ parts. Note that if we divide V_i in $k' < k_i$ parts one of the resulting blocks at end of recursive bisection is imbalanced, since $\frac{c(V_i)}{k'} > w_{max}$ with $i \in \{1, 2\}$. If we want to further divide the two resulting parts of a bisection in k_1 and k_2 parts, they must fulfil the condition $k_1 + k_2 = k$. Assume the situation where the weight of the hypergraph is $c(V) = 200$, $\epsilon = 0.1$ and we want to divide the hypergraph in $k = 4$ parts. The partition weight bound is $w_{max} = 55$. The first

bisection produces a weight of the two resulting blocks of $c(V_1) = 125$ and $c(V_2) = 75$. We have to choose $k_1 = 3$ and $k_2 = 2$ with the definition above $\Rightarrow k_1 + k_2 = 3 + 2 = 5 > 4 = k$. If $k_1 + k_2 > k$, we have to move hypernodes from one block to the other until $k_1 + k_2 = k$. It is not possible to choose one k_i smaller without moving a hypernode, because than one of the resulting blocks is imbalanced at the end. If $k_1 + k_2 < k$, we can increase k_1 or k_2 until $k_1 + k_2 = k$. Note that this procedure do not produce an imbalanced partition at the end. We believe that this variant produces better cuts as our actual method, because we explore a greater solution space and the *rollback* technique takes more effect.
The next open task in our work is a redefinition of the *FM* gain. The *FM* gain function takes a move into account, if they increase or decrease the cut. Maybe it is more advantageously to take the gain in the future, which a move can produce into the calculation. Their exists several concept e.g. *higher-level gains*, but they are all runtime intensive and slow down the partitioning process. We focus on an other gain function which describes the gain and potential in the future in one function. This function $g : V \times E \times P \times P \to \mathbb{R}$ describes the potential for moving a hypernode v, which is contained in a net e, from block V_i to V_j

$$g(v, e, V_i, V_j) = \omega(e) \cdot \frac{1 + |e \cap V_j| - |e \cap V_i|}{|e| - 1} \tag{6.1}$$

We can define the gain of a move as follows:

$$g'(v, V_i, V_j) = \sum_{e \in I(v)} g(v, e, V_i, V_j) \tag{6.2}$$

The gain function in equation 6.1 has interesting properties, which we summarize in the following listing:

(i) If $\lambda(e) = 2$, $|e \cap V_i| = 1$ and $|e \cap V_j| = |e| - 1$, than $g(v, e, V_i, V_j) = \omega(e)$
(ii) If $\lambda(e) = 1$, $|e \cap V_i| = |e|$ and $|e \cap V_j| = 0$, than $g(v, e, V_i, V_j) = -\omega(e)$
(iii) If $|e \cap V_i| > |e \cap V_j|$ than $g(v, e, V_i, V_j) \leq 0$
(iv) If $|e \cap V_i| \leq |e \cap V_j|$ than $g(v, e, V_i, V_j) > 0$
(v) For all moves of a hypernode v in a net e from V_i to V_j the value of the gain is within $-\omega(e) \leq g(v, e, V_i, V_j) \leq \omega(e)$

Property i) and ii) ensures that if a move decrease or increase the cut, this modified gain function behave like the original *FM* gain function. The next two properties are important for the future gain of a move. It is more promising to move a hypernode v, which is contained in a net e, from his current block V_i to an other V_j, where $|e \cap V_i| \leq |e \cap V_j|$. We can remove V_i from e with less moves than V_j. The last property means that the maximum and minimum value of g is reached, if a move decrease or increase the cut. In all other situation the gain is between those two values. This function seem to open a promising area for future work. It is possible to use this function in our greedy initial partitioner or in the *uncoarsening* phase as gain function for a *local search* algorithm in a multilevel partitioning framework. A open question for this function is the implementation of delta gain updates to provide acceptable running times. We integrate this function without delta gain updates in our greedy initial partitioner and test it manually. On the first view it decrease the cut on the sparse matrix instances in comparison with the *FM* variant by a huge amount. At the VLSI instances the modify gain function provides near equal results, because these instances has a low average hyperedge size and in this case the behaviour is near equal to the *FM* gain function.
In the future we want to test and evaluate an adaption of the *Iterative Local Search* (ILS) algorithm from Cong [8]. An ILS algorithm normally create an initial solution for a problem and repeat a loop where the solution is pertubated and than the solution is improved with a

local search technique. To pertubate a solution means to change the existing solution in a way that we move away from the neighbourhood of this solution, but not to change them completely. In combination with a *local search* heuristic we have the advantage to jump out of local minima and explore an other one near to the old one. We implement a first version and using the *Stable Net Removal* technique [8] to pertubate the solution. *Stable Nets* are hyperedges which are before the execution of *local search* algorithm cut edges and after. Shibuya [27] has shown that about 80% of the cut edges are stable nets and hard to remove from the cut. By removing a small amount of these nets from the cut we can perturbate our solution. As *local search* heuristic we used the *ScLap*-based algorithm which provides very fast *local searches* on small hypergraphs and therefore we can make much more iterations than with a *FM local search* algorithm. The tuning and evaluation of this technique is a task for the future.
Finally we want to provide different multiple runs parameters for each initial partitioner in the algorithm set of the *pool* initial partitioner. Every partitioning method has different running times and qualities. If we choose different multiple runs parameters for each partitioner we can maybe decrease the running time and increase the quality of the pool initial partitioner. The pool partitioner also offers the possibility to parallize the execution of the initial partitioner and further speed up the partitioning process.

References

[1] Charles J Alpert, Andrew E Caldwell, Andrew B Kahng, and Igor L Markov. Partitioning with terminals: a "new" problem and new benchmarks. In *Proceedings of the 1999 international symposium on Physical design*, pages 151–157. ACM, 1999.

[2] Charles J Alpert and Andrew B Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI journal*, 19(1):1–81, 1995.

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

[4] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *Preprint*, 2013.

[5] Ümit Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). pages 1479–1487, 2011.

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

[7] Ü Çatalyürek and C Aykanat. PaToH (Partitioning Tool for Hypergraphs). *Encyclopedia of Parallel Computing*, pages 1–9, 2011.

[8] Jason Cong, Honching Peter Li, Sung Kyu Lim, Toshiyuki Shibuya, and Dongmin Xu. Large scale circuit partitioning with loose/stable net removal and signal flow based clustering. pages 441–446, 1997.

[9] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[10] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive fem. *Parallel Computing*, 26(12):1555–1581, 2000.

[11] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. pages 175–181, 1982.

[12] Vitali Henne. Label Propagation for Hypergraph Partitioning. 2015.

[13] Vitali Henne, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, and Christian Schulz. n -Level Hypergraph Partitioning.

[14] G Karypis and V Kumar. hMETIS 1.5: A hypergraph partitioning package. 1998.

[15] George Karypis. Multilevel Hypergraph Partitioing. Technical report, 2002.

[16] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):69–79, 1999.

[17] George Karypis and Vipin Kumar. Multilevel k-way Hypergraph Partitioning.

[18] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[19] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[20] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[21] Dorothy Kucar, Shawki Areibi, and Anthony Vannelli. Hypergraph partitioning techniques. *Dynmaics of continous discrete and impulsive systems series A*, 11:339–368, 2004.

[22] Vitaly Osipov and Peter Sanders. n-level graph partitioning. pages 278–289, 2010.

[23] David A Papa and Igor L Markov. Hypergraph partitioning and clustering. *Approximation algorithms and metaheuristics*, pages 61–1, 2007.

[24] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[25] Peter Sanders and Christian Schulz. *High quality graph partitioning.* PhD thesis, 2012.

[26] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*, XX(Xx):1–14, 2003.

[27] T. SHIBUYA. Smincut : Vlsi placement tool using min-cut. *Fujitsu Scientific & Technical Journal*, 31(2):197–207, 1995.

[28] Aleksandar Trifunovic and William J Knottenbelt. Parkway 2.0: A parallel multilevel hypergraph partitioning tool, 2004.

A Benchmark Instances

Table 15: Used hypergraphs from the ISPD98 Circuit Benchmark Suite. *Medium size* instances are highlighted bold.

Hypergraph	$\|V\|$	$\|H\|$	$\|p\|$	HE-Size				HN-Degree			
				μ	σ	min	max	μ	σ	min	max
ibm01	12752	14111	50566	3.58	3.34	2	42	3.97	2.33	1	39
ibm02	19601	19584	81199	4.15	5.45	2	134	4.14	2.29	1	69
ibm03	23136	27401	93573	3.41	3.11	2	55	4.04	3.45	1	100
ibm04	27507	31970	105859	3.31	2.92	2	46	3.85	4.65	1	526
ibm05	29347	28446	126308	4.44	4.29	2	17	4.30	2.35	1	9
ibm06	32498	34826	128182	3.68	3.28	2	35	3.94	1.84	1	91
ibm07	45926	48117	175639	3.65	3.05	2	25	3.82	2.41	1	98
ibm08	51309	50513	204890	4.06	5.01	2	75	3.99	6.18	1	1165
ibm09	53395	60902	222088	3.65	3.13	2	39	4.16	3.22	1	173
ibm10	69429	75196	297567	3.96	3.56	2	41	4.29	3.22	1	137
ibm11	70558	81454	280786	3.45	2.60	2	24	3.98	3.17	1	174
ibm12	71076	77240	317760	4.11	3.72	2	28	4.47	4.68	1	473
ibm13	84199	99666	357075	3.58	3.01	2	24	4.24	3.34	1	180
ibm14	147605	152772	546816	3.58	2.94	2	33	3.70	3.18	1	270
ibm15	161570	186608	715823	3.84	3.51	2	36	4.43	3.29	1	306
ibm16	183484	190048	778823	4.10	3.61	2	40	4.24	2.77	1	177
ibm17	185495	189581	860036	4.54	4.07	2	36	4.64	2.49	1	81
ibm18	210613	201920	819697	4.06	3.96	2	66	3.89	1.90	1	97

Table 16: Used hypergraphs from the Florida Sparse Matrix Collection. *Medium size* instances are highlighted bold.

				HE-Size				HN-Degree			
Hypergraph	$\|V\|$	$\|H\|$	$\|p\|$	μ	σ	min	max	μ	σ	min	max
vibrobox	12328	12328	342828	27.81	16.09	9	121	27.81	16.09	9	121
bcsstk29	13992	13992	619488	44.27	15.64	5	71	44.27	15.64	5	71
memplus	17758	17758	126150	7.10	22.04	2	574	7.10	22.04	2	574
bcsstk30	28924	28924	2043492	70.65	31.72	4	219	70.65	31.72	4	219
bcsstk31	35588	35588	1181416	33.20	15.18	2	189	33.20	15.18	2	189
bcsstk32	44609	44609	2014701	45.16	15.48	2	216	45.16	15.48	2	216
finan512	74752	74752	596992	7.99	6.28	3	55	7.99	6.28	3	55
af_shell9	504855	504855	17588875	34.84	1.28	20	40	34.84	1.28	20	40
audikw_1	943695	943695	77651847	82.28	42.45	21	345	82.28	42.45	21	345
ldoor	952203	952203	46522475	48.86	11.95	28	77	48.86	11.95	28	77
ecology2	999999	999999	4995991	5.00	0.06	3	5	5.00	0.06	3	5
ecology1	1000000	1000000	4996000	5.00	0.06	3	5	5.00	0.06	3	5
thermal2	1228045	1228045	8580313	6.99	0.81	1	11	6.99	0.81	1	11
af_shell10	1508065	1508065	52672325	34.93	0.90	15	35	34.93	0.90	15	35
G3_circuit	1585478	1585478	7660826	4.83	0.64	2	6	4.83	0.64	2	6
kkt_power	2063494	2063494	14612663	7.08	7.40	1	96	7.08	7.40	1	96
nlpkkt120	3542400	3542400	96845792	27.34	3.09	5	28	27.34	3.09	5	28
nlpkkt160	8345600	8345600	229518112	27.50	2.70	5	28	27.50	2.70	5	28

B Detailed Parameter Tuning results

B.1 Contraction Limit and Hypernode Weight Bound

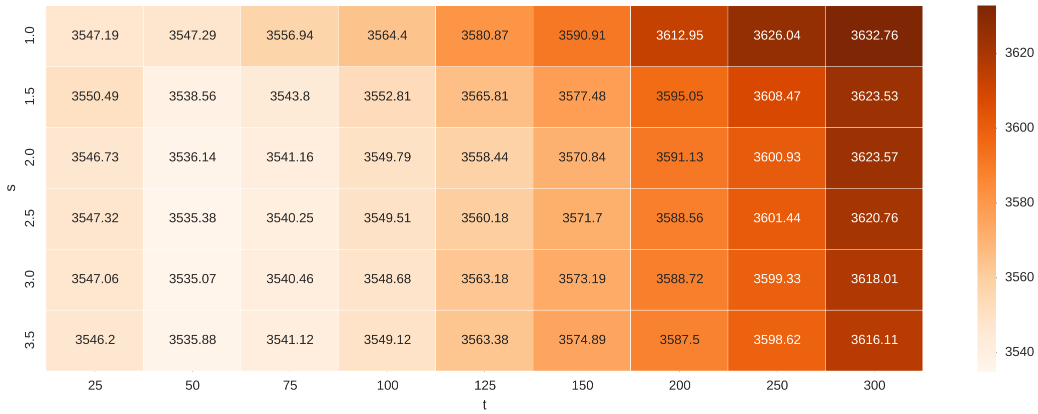


Figure 23: Results of the experiment with parameter s and t of our *bfs* initial partitioner with 20 runs at each bisection step.

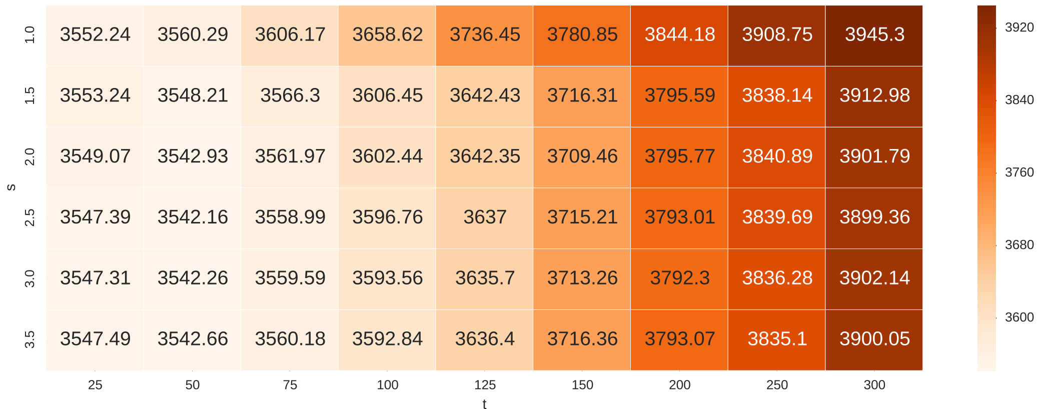


Figure 24: Results of the experiment with parameter s and t of our *lp* initial partitioner with 20 runs at each bisection step.

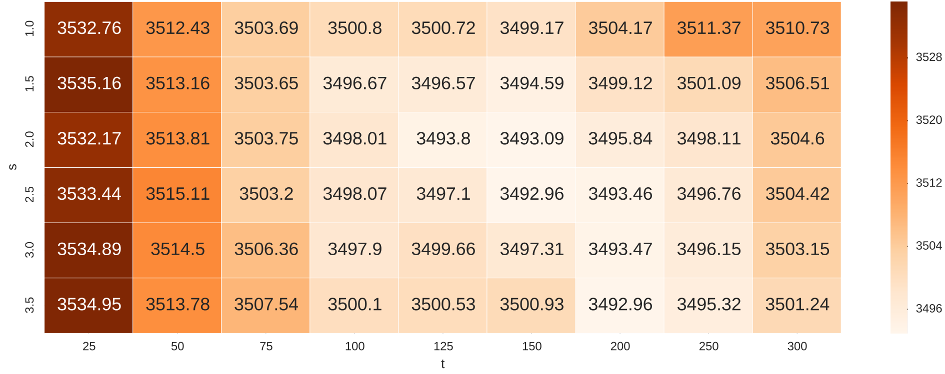


Figure 20: Results of the experiment with parameter s and t of our *RBNL* pool initial partitioner.

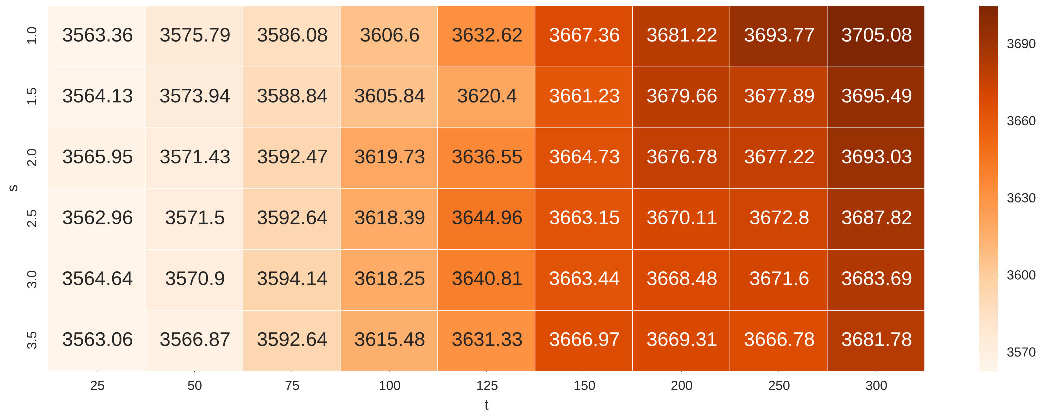


Figure 21: Results of the experiment with parameter s and t of our *greedy global FM* initial partitioner with 20 runs at each bisection step.

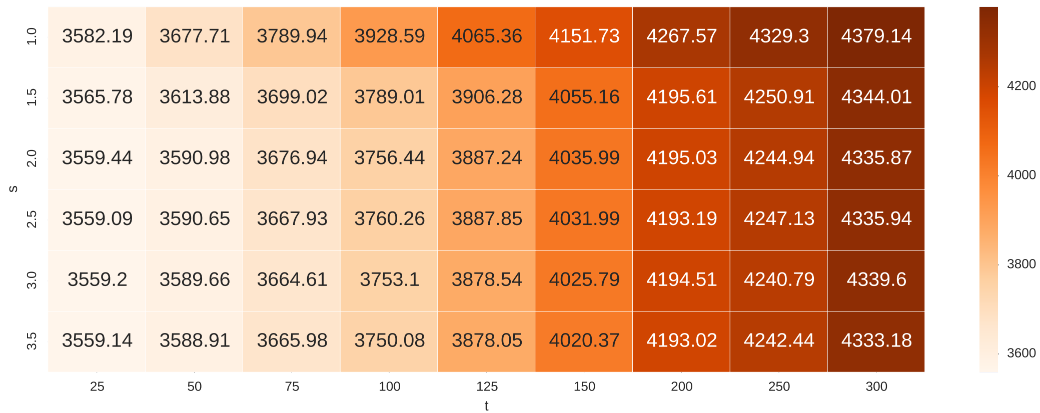


Figure 22: Results of the experiment with parameter s and t of our *random* initial partitioner with 20 runs at each bisection step.

B.2 Fruitless Moves of the *FM Local Search* algorithm

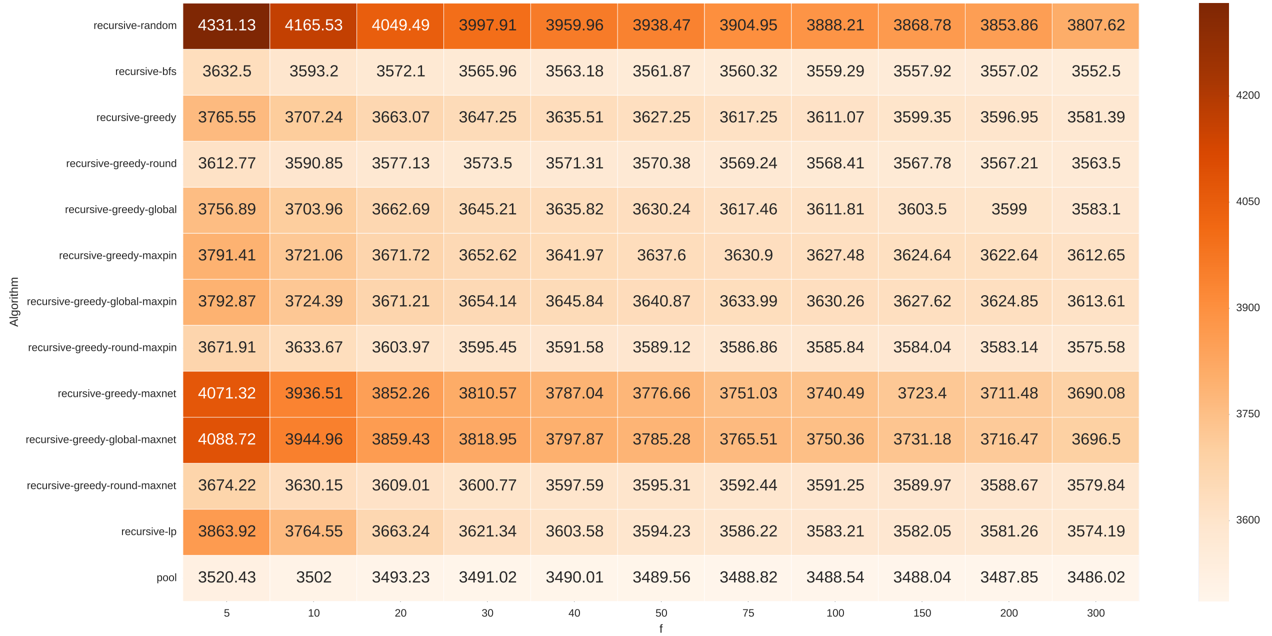


Figure 25: Cut values of our *RBNL* initial partitioners with different maximum numbers of fruitless moves.

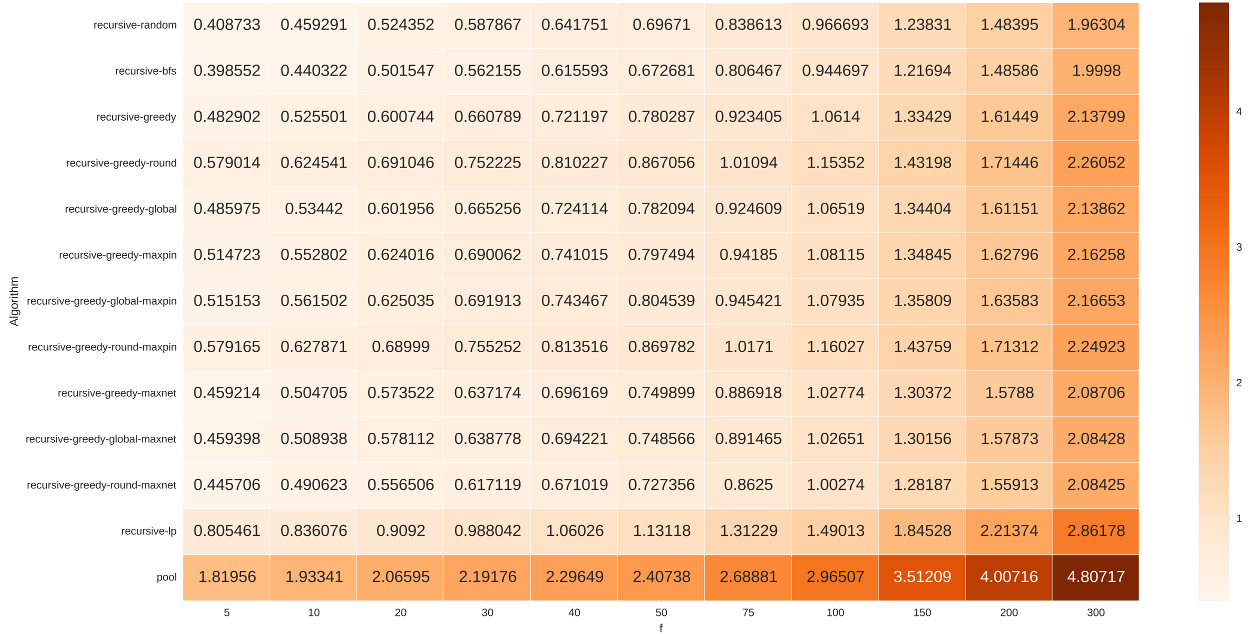


Figure 26: Running times of our *RBNL* initial partitioners with different maximum numbers of fruitless moves.

C Detail Results

C.1 Initial Partitioning results on the *medium sized* benchmark set

	All Benchmarks			VLSI Benchmarks			SPM Benchmarks		
Algorithm	Min.	Avg.	t[s]	Min.	Avg.	t[s]	Min.	Avg.	t[s]
KaHyPar-Pool	3354.82	3479.6	2.09	2816.18	2907.87	2.35	3897.85	4070.47	1.56
hMetis	+1.24	+1.13	2.49	+0.76	+0.45	3.05	+1.96	+2.16	1.61
PaToH	+2.6	+2.46	0.39	+2.19	+1.9	0.45	+3.09	+3.15	0.3

Table 17: Result of our final initial partitioner on the coarsened instances of the *medium sized* benchmark set.

	KaHyPar			hMetis			PaToH		
k	Min.	Avg.	t[s]	Min.	Avg.	t[s]	Min.	Avg.	t[s]
2	1031.24	1101.19	0.34	1029.69	1099.11	0.24	1053.49	1131.04	0.08
4	2086.75	2216.44	0.87	2133.78	2236.13	0.77	2151.2	2273.64	0.19
8	3375.6	3493.35	1.75	3444.04	3583.31	1.93	3478.19	3588.59	0.32
16	4704.33	4804.84	3.14	4737.98	4869.03	4.31	4797.94	4897.67	0.57
32	5882.26	5999.77	5.41	5947.57	6059.23	8.82	6014.21	6122.68	0.89
64	7092.36	7221.06	9.27	7199.91	7307.08	18.19	7309.33	7421.74	1.44

Table 18: Result of our final initial partitioner on the coarsened instances of the *medium sized* benchmark set for each k.

C.2 Initial and Final Cuts of *KaHyPar*

	All Benchmarks			
Initial Partitioner	Init. Min.	Min.	Init Avg.	Avg.
KaHyPar-Pool	**10655.34**	**7936.92**	**10957.09**	8118.07
hMetis	10738.71	7937.32	11024.4	**8110.19**

Table 19: Initial cuts (Init. Min. & Avg.) and final cuts of *KaHyPar* with *hMetis* and the *Pool* partitioner as initial partitioner on the full benchmark set.

	VLSI Benchmarks				SPM Benchmarks			
Initial Partitioner	Init. Min.	Min.	Init Avg.	Avg.	Init. Min.	Min.	Init Avg.	Avg.
KaHyPar-Pool	**5919.57**	4511.43	**6073.54**	4627.93	**19179.8**	**13963.38**	**19767.37**	**14240.28**
hMetis	5962.62	**4506.62**	6101.4	**4615.9**	19340.48	13979.69	19919.61	14249.71

Table 20: Initial cuts (Init. Min. & Avg.) and final cuts of *KaHyPar* with *hMetis* and the *Pool* partitioner as initial partitioner on the *VLSI* and *Sparse Matrix* instances. Best result in a column is highlighted bold.

	KaHyPar-Pool				hMetis			
k	Init. Min.	Min.	Init Avg.	Avg.	Init. Min.	Min.	Init Avg.	Avg.
2	2588.72	**1650.14**	2757.07	**1701.07**	**2588.27**	1653.69	**2749.35**	1703.19
4	**5330.4**	3534.75	**5595.95**	3651.96	5399.72	**3524.77**	5646.43	**3650**
8	**8732.07**	**6032.93**	**8995.53**	6230.13	8801.99	6048.9	9054.69	**6219.74**
16	**12520.4**	9303.81	**12776.48**	9535.33	12606.31	**9249.73**	12857.1	**9471.91**
32	**16767.86**	13291.17	**17031.4**	13524.3	16882.48	**13288.52**	17121.25	**13495.93**
64	**22013.75**	18486.78	**22250.24**	18713.81	22170.46	**18462.37**	22387.06	**18671.54**
128	**28003.9**	**24664.56**	**28217.73**	**24877.79**	28372.97	24806.41	28571.41	25007.44

Table 21: Initial cuts (Init. Min. & Avg.) and final cuts of *KaHyPar* with *hMetis* and the *Pool* partitioner as initial partitioner on the full benchmark set for each k. Best results in comparison are highlighted bold.

C.3 Overall Results of *KaHyPar* with different initial partitioner

Overall results of our final experiment of *KaHyPar* with the *BFS* and *Pool* initial partitioner and *hMetis* and *PaToH* as initial partitioner. The running time in the following table is the running time of the initial partitioner.

		KaHyPar-BFS			KaHyPar-Pool			PaToH			hMetis		
H	k	Min	Avg	t	Min	Avg	t	Min	Avg	t	Min	Avg	t
G3_circuit	2	2142	2238.0	0.02	2142	2204.6	0.13	2142	2234.1	0.08	2142	2212.50	0.10
	4	5353	6050.7	0.03	5376	5476.8	0.40	5370	5552.6	0.15	5368	5482.20	0.26
	8	9654	10760.5	0.09	9666	9827.4	0.98	9720	9905.4	0.30	9694	9839.10	0.70
	16	16849	17747.4	0.22	15424	15800.6	2.19	15605	16166.7	0.63	15369	15966.20	1.91
	32	25720	26862.4	0.60	23319	24073.0	4.69	23526	24145.3	1.41	23240	23712.30	5.36
	64	39201	40871.8	1.46	37189	37698.4	9.92	37304	37901.4	2.95	36726	37572.00	14.34
	128	63630	64560.9	3.71	61487	61999.7	20.53	61149	62074.4	5.61	61668	61916.10	34.84
af_shell10	2	5250	5250.0	0.01	5250	5250.0	0.09	5250	5250.0	0.07	5250	5250.00	0.09
	4	11170	12082.0	0.02	10855	11094.5	0.25	11020	11190.0	0.12	10830	11202.00	0.21
	8	20225	21552.0	0.05	20715	21275.0	0.58	20090	21088.0	0.21	20830	21161.50	0.52
	16	32835	34647.5	0.11	32095	32954.5	1.25	32305	32747.5	0.40	31585	32451.50	1.44
	32	52010	54078.5	0.28	50170	51221.0	2.64	50045	50991.0	0.77	50475	51560.50	3.60
	64	77560	79550.0	0.70	74185	75300.0	5.55	74780	75533.5	1.50	73175	75137.00	8.87
	128	115135	116681.5	1.80	110525	111875.5	11.71	110305	111238.0	2.98	111035	111796.00	19.38
af_shell9	2	1810	2152.5	0.01	1810	1869.0	0.08	1810	1870.0	0.04	1810	1869.00	0.06
	4	4480	5691.5	0.01	4420	4460.0	0.23	4270	4441.0	0.07	4430	4456.00	0.17
	8	8335	9744.5	0.03	8530	8567.5	0.55	8525	8597.5	0.14	8305	8556.00	0.47

H	k	KaHyPar-BFS			KaHyPar-Pool			PaToH			hMetis		
		Min	Avg	t	Min	Avg	t	Min	Avg	t	Min	Avg	t
	16	16655	17406.0	0.10	15745	16061.5	1.24	15680	15920.0	0.29	15660	16028.00	1.32
	32	26800	27668.0	0.26	25800	26370.0	2.68	26020	26294.5	0.55	25860	26291.00	3.41
	64	41000	42911.0	0.74	40640	41073.5	5.75	40140	40850.5	1.33	40570	40859.00	9.12
	128	62510	63981.0	1.99	59690	60604.5	12.44	59540	60420.0	2.79	60530	60711.50	21.03
audikw_1	2	10563	12471.3	0.02	10563	10643.4	0.35	10563	10643.4	0.15	10563	10643.40	0.21
	4	32073	35247.3	0.07	31917	32148.9	1.21	32094	32199.0	0.39	32085	32189.40	0.81
	8	74730	78999.3	0.36	74406	75081.9	3.38	74916	75208.2	1.10	74817	75060.60	3.07
	16	124920	130746.3	1.49	123402	124444.2	8.49	122382	124246.8	2.69	122547	124431.00	10.61
	32	184194	189902.7	4.63	180918	183517.8	19.48	182397	183739.2	5.99	181539	183848.40	31.29
	64	262887	265367.7	11.85	256731	258853.8	39.78	256581	258495.6	10.77	257145	258365.70	77.17
	128	361593	366080.7	25.01	357555	358993.8	71.06	357834	359432.1	18.82	357438	358449.90	151.63
bcsstk29	2	360	362.4	0.00	360	361.2	0.10	360	361.2	0.07	360	361.80	0.10
	4	1080	1104.0	0.01	1080	1088.4	0.30	1080	1092.0	0.06	1080	1093.20	0.22
	8	2226	2377.2	0.06	2184	2221.8	0.65	2190	2221.7	0.11	2184	2228.40	0.64
	16	3552	3860.2	0.16	3576	3630.9	1.24	3580	3668.4	0.20	3558	3638.60	1.63
	32	5604	5742.0	0.52	5166	5225.7	2.52	5156	5300.6	0.38	5202	5275.80	4.46
	64	8248	8465.5	2.45	6972	7089.2	6.86	7497	7625.5	0.93	6998	7108.40	14.50
	128	10839	10994.3	8.03	9479	9537.1	18.31	9865	9865.0	2.29	10782	10823.30	26.10
bcsstk30	2	527	569.5	0.00	527	529.2	0.17	527	529.2	0.06	527	529.20	0.11
	4	1486	1802.3	0.04	1482	1524.7	0.56	1482	1536.9	0.14	1482	1521.20	0.41
	8	3362	4187.5	0.12	3103	3206.0	1.31	3080	3199.6	0.26	3103	3154.70	1.26
	16	7210	7665.2	0.36	6567	6690.9	2.65	6648	6763.8	0.48	6555	6668.00	3.30
	32	11259	11783.7	0.89	10139	10270.1	4.78	10498	10639.7	0.72	10201	10354.70	7.88
	64	16578	16915.0	2.28	14212	14457.0	17.06	14550	14850.0	1.18	14526	14724.70	18.60
	128	21346	21614.8	7.08	18718	19017.8	17.11	19317	19597.2	2.32	19789	20048.30	33.84
bcsstk31	2	665	772.7	0.01	665	682.1	0.20	664	691.9	0.07	664	685.70	0.11
	4	1782	1998.7	0.04	1628	1721.3	0.62	1628	1738.4	0.15	1628	1690.60	0.41
	8	3418	3884.6	0.14	3198	3328.1	1.43	3284	3420.9	0.29	3236	3351.00	1.28
	16	5718	6406.3	0.39	5699	5760.3	2.79	5523	5723.3	0.52	5596	5716.20	3.53
	32	9424	9791.6	0.87	8767	8975.8	5.01	8806	9024.8	0.90	8826	8934.30	8.56
	64	13594	14124.1	1.91	12831	13034.4	8.66	13110	13231.1	1.54	12935	13059.90	19.12
	128	19697	19909.3	4.23	17907	18016.2	15.16	18335	18473.3	2.34	17991	18080.70	37.44
bcsstk32	2	946	1023.4	0.01	831	872.7	0.14	831	901.5	0.04	831	831.60	0.10
	4	1677	2370.2	0.03	1594	1778.2	0.39	1648	1838.2	0.09	1594	1741.60	0.30
	8	3749	4353.7	0.10	3627	3711.2	0.96	3567	3773.9	0.19	3615	3731.40	0.90
	16	6649	7089.4	0.26	6181	6395.4	1.98	6288	6452.5	0.35	6170	6339.60	2.40
	32	10447	10742.7	0.63	9836	10001.0	3.67	9924	10039.2	0.69	9840	9951.50	5.79
	64	15336	15788.8	1.39	14072	14363.0	6.72	14276	14557.6	1.15	14013	14302.00	13.83
	128	21875	22453.3	3.28	19847	20108.0	12.15	20297	20548.8	1.84	19834	20018.00	29.04
ecology1	2	2000	2000.0	0.01	2000	2000.0	0.09	2000	2000.0	0.10	2000	2000.00	0.27
	4	3676	3939.4	0.02	3700	3735.9	0.25	3636	3730.6	0.08	3690	3743.90	0.19
	8	6151	7053.7	0.04	6424	6640.5	0.58	6512	6791.0	0.16	6487	6635.00	0.51
	16	10443	10809.1	0.11	10126	10347.3	1.24	10094	10321.0	0.30	10160	10288.80	1.36
	32	15905	16198.8	0.28	15483	15971.9	2.63	15714	16013.1	0.61	15689	15983.20	3.55
	64	22728	23475.0	0.71	22441	22845.9	5.46	22579	22857.3	1.28	22750	22900.10	8.74
	128	33129	33985.9	1.78	32915	33374.8	11.41	32230	33067.7	2.62	32833	33291.70	19.71
ecology2	2	1998	1999.8	0.01	1998	1999.6	0.09	1998	1999.6	0.05	1998	1999.60	0.07
	4	3689	3884.8	0.02	3630	3721.2	0.25	3652	3720.9	0.08	3673	3734.20	0.19
	8	6363	6849.7	0.05	6368	6660.8	0.58	6487	6688.1	0.15	6533	6631.20	0.51
	16	10393	10769.3	0.11	10100	10310.1	1.26	10102	10298.5	0.30	10141	10318.50	1.35
	32	15651	16234.1	0.29	15577	15844.1	2.64	15387	15959.1	0.58	15381	15983.10	3.43
	64	22736	23329.8	0.70	22645	22876.3	5.50	22380	22749.2	1.31	22380	22818.90	8.73
	128	32367	33814.7	1.77	33003	33251.3	11.47	32526	32890.4	2.69	32625	33246.80	19.28
finan512	2	146	146.0	0.00	146	146.0	0.18	146	146.0	0.06	146	146.00	0.10
	4	292	343.1	0.04	292	292.0	0.67	292	292.0	0.15	292	292.00	0.38
	8	584	737.3	0.17	584	591.3	1.78	584	584.0	0.40	584	591.30	1.25
	16	1314	1539.8	0.47	1168	1175.3	3.77	1168	1182.6	0.71	1168	1168.00	4.10
	32	2336	2524.9	0.93	2336	2336.0	6.81	2336	2336.0	1.27	2336	2336.00	10.45
	64	9500	9766.4	2.05	9056	9111.6	12.24	9068	9212.5	2.15	9049	9120.80	23.71
	128	20156	20517.0	8.69	18303	18560.9	41.85	20094	20634.3	3.52	18255	18452.70	43.39
ibm01	2	203	290.3	0.01	203	245.7	0.18	205	258.9	0.29	203	248.40	0.14
	4	513	621.2	0.05	520	601.5	0.47	515	589.9	0.09	583	599.20	0.39
	8	931	1024.8	0.14	879	895.9	0.86	868	898.4	0.16	862	880.60	0.99
	16	1309	1410.1	0.30	1249	1269.0	1.51	1258	1278.4	0.24	1242	1265.50	2.07
	32	1827	1910.2	0.56	1651	1680.8	2.51	1659	1709.4	0.36	1653	1683.40	4.10
	64	2568	2629.1	1.00	2218	2250.9	4.15	2243	2291.7	0.61	2229	2241.40	8.19
	128	3500	3565.0	1.19	2922	2947.2	6.02	2973	2973.0	0.78	2959	2973.10	11.22
ibm02	2	350	411.2	0.01	356	369.2	0.35	352	368.5	0.52	352	365.50	0.31
	4	748	987.4	0.07	705	717.1	0.87	698	741.3	0.19	704	721.50	0.83
	8	2111	2280.5	0.20	2004	2077.1	1.55	2000	2110.4	0.33	2002	2060.20	2.21
	16	3479	3582.4	0.47	3395	3427.3	2.52	3379	3459.5	0.48	3372	3409.50	5.06
	32	4463	4594.1	0.96	4340	4370.7	3.76	4283	4382.2	0.62	4371	4403.30	8.92
	64	5349	5406.1	1.67	5137	5158.1	5.63	5136	5185.7	0.84	5179	5218.70	14.04
	128	6203	6348.8	3.00	5902	5934.3	8.78	6027	6027.0	1.31	6087	6113.20	19.90
ibm03	2	957	976.7	0.01	957	966.9	0.35	954	974.4	0.08	957	967.70	0.28
	4	1829	2032.2	0.09	1753	1817.9	0.78	1699	1808.8	0.17	1721	1776.20	0.79
	8	2693	2968.0	0.29	2646	2719.9	1.41	2580	2693.3	0.25	2513	2641.10	1.70
	16	3543	3664.3	0.57	3321	3398.1	2.40	3305	3433.9	0.38	3307	3383.90	3.29
	32	4265	4352.8	2.28	4025	4075.0	7.52	4072	4098.7	0.54	4020	4052.30	5.91

		KaHyPar-BFS			KaHyPar-Pool			PaToH			hMetis		
H	k	Min	Avg	t	Min	Avg	t	Min	Avg	t	Min	Avg	t
	64	5030	5135.0	1.86	4679	4741.8	5.90	4773	4798.8	0.79	4708	4750.90	10.90
	128	6221	6295.4	3.07	5659	5683.8	9.34	5756	5791.3	1.17	5776	5809.50	16.96
ibm04	2	593	619.2	0.01	590	609.0	0.33	585	612.1	0.08	590	607.60	0.25
	4	1805	1875.8	0.08	1754	1782.0	0.87	1747	1789.5	0.18	1757	1787.50	0.85
	8	3094	3202.1	0.25	2860	2946.2	1.61	2901	2986.9	0.33	2860	2916.40	1.95
	16	4055	4225.0	0.57	3876	3921.1	2.64	3837	3947.5	0.48	3778	3869.00	3.87
	32	5248	5357.6	1.10	4994	5053.4	4.23	5039	5092.2	0.65	4982	5045.20	6.95
	64	6459	6565.4	3.74	6126	6178.4	6.37	6186	6223.7	1.06	6091	6126.10	12.15
	128	7800	7934.8	3.22	7260	7294.2	9.99	7353	7440.4	1.44	7314	7375.90	18.87
ibm05	2	1725	1777.5	0.01	1727	1735.3	0.52	1725	1734.7	0.17	1727	1735.30	0.44
	4	3034	3295.2	0.10	2966	3013.4	1.07	2938	3055.3	0.20	2979	3023.30	1.16
	8	4362	4663.1	0.34	4230	4331.7	1.74	4302	4535.9	0.34	4405	4572.60	2.60
	16	5573	5756.2	0.83	5467	5558.0	2.78	5486	5571.2	0.45	5269	5408.30	4.50
	32	6166	6273.1	1.60	5974	6090.1	4.20	6046	6127.7	0.60	5900	5992.40	7.82
	64	6793	6946.8	2.80	6584	6645.2	6.39	6496	6565.3	0.85	6445	6530.10	14.11
	128	7507	7610.9	4.35	7218	7277.5	10.00	7087	7181.4	1.30	7116	7225.80	20.73
ibm06	2	982	1109.1	0.01	982	1040.7	0.44	982	1034.5	0.10	1043	1068.10	0.35
	4	1665	1852.0	0.09	1481	1594.0	0.96	1493	1651.0	0.18	1509	1637.50	0.95
	8	2412	2575.9	0.35	2345	2393.8	1.80	2346	2392.5	0.34	2370	2397.90	2.11
	16	3339	3538.8	0.66	3186	3226.6	2.88	3180	3260.8	0.50	3231	3255.60	4.19
	32	4444	4528.5	1.43	4159	4197.1	4.75	4238	4276.9	0.66	4160	4226.80	7.62
	64	5483	5656.1	2.50	5123	5193.4	7.14	5211	5250.0	0.95	5122	5164.50	13.31
	128	6833	6918.2	7.14	6225	6258.7	22.14	6314	6379.9	1.48	6313	6347.20	20.24
ibm07	2	931	964.2	0.02	934	965.5	0.50	912	954.4	0.10	931	957.30	0.37
	4	2283	2598.6	0.13	2264	2293.4	1.22	2201	2319.8	0.27	2253	2298.20	1.13
	8	3549	3834.7	0.45	3410	3478.8	2.15	3415	3547.0	0.38	3391	3452.80	2.46
	16	4977	5178.6	0.88	4710	4833.6	3.53	4734	4898.9	0.64	4600	4782.80	5.05
	32	6284	6483.0	1.55	5998	6130.4	5.40	6108	6191.1	0.91	5979	6069.50	9.09
	64	7928	8091.7	2.68	7571	7633.5	8.31	7564	7664.5	1.29	7516	7588.90	16.46
	128	9693	9816.8	8.21	9110	9162.0	24.46	9195	9233.3	1.79	9133	9177.80	24.61
ibm08	2	1146	1362.8	0.01	1146	1165.7	0.43	1163	1168.5	0.13	1146	1166.10	0.34
	4	2409	2651.0	0.08	2341	2364.0	1.04	2343	2381.5	0.26	2344	2369.20	0.99
	8	3707	3863.8	0.32	3486	3532.0	1.86	3504	3564.1	0.38	3496	3541.60	2.16
	16	4888	5123.4	0.63	4438	4680.6	3.06	4624	4751.7	0.55	4595	4645.00	4.35
	32	6392	6575.3	1.33	6033	6132.3	4.95	5976	6148.6	0.82	5968	6068.00	8.95
	64	8257	8436.7	2.28	7751	7834.3	7.93	7846	7906.8	1.33	7778	7882.40	16.18
	128	10019	10119.9	3.74	9321	9371.8	12.33	9392	9455.7	1.80	9466	9503.70	27.28
ibm09	2	621	667.5	0.01	621	623.4	0.37	621	624.6	0.09	621	624.30	0.23
	4	1795	2294.4	0.11	1701	1739.5	0.96	1732	1789.5	0.20	1694	1724.00	0.78
	8	2942	3246.5	0.33	2782	2889.1	1.82	2824	2909.9	0.34	2682	2836.20	1.81
	16	4431	4621.6	0.76	3980	4039.8	3.18	3914	4063.1	0.54	3943	4014.40	4.11
	32	6062	6260.1	1.49	5466	5565.9	5.48	5414	5604.9	0.94	5485	5533.20	8.23
	64	7901	8120.6	2.61	7301	7419.4	8.65	7498	7570.3	1.24	7327	7381.00	15.93
	128	10177	10384.3	4.39	9441	9494.9	13.61	9509	9647.8	1.99	9393	9477.90	25.41
ibm10	2	1296	1623.5	0.02	1289	1307.4	0.53	1295	1458.0	0.14	1293	1383.00	0.36
	4	2453	2905.9	0.14	2489	2574.4	1.28	2431	2555.6	0.33	2419	2528.60	1.11
	8	4493	4899.2	0.46	4064	4295.3	2.44	4188	4368.0	0.54	4198	4288.70	2.87
	16	6568	6982.1	1.03	6025	6182.9	4.32	5970	6216.7	0.87	6048	6244.00	5.66
	32	8986	9499.9	2.08	8369	8531.6	6.98	8495	8655.0	1.22	8547	8631.10	11.02
	64	12362	12574.5	3.39	11619	11768.5	10.64	11676	11863.8	1.74	11729	11795.00	20.61
	128	15410	15585.5	5.83	14600	14699.1	16.38	14708	14822.5	2.62	14645	14777.30	32.55
ibm11	2	1076	1383.4	0.02	1073	1145.7	0.48	1067	1179.8	0.11	1077	1171.00	0.34
	4	2841	3088.7	0.13	2472	2574.4	1.19	2546	2833.0	0.25	2436	2530.60	0.99
	8	3989	4219.5	0.42	3548	3794.9	2.20	3582	3845.9	0.45	3549	3783.90	2.28
	16	5735	6326.5	0.94	5245	5427.7	3.90	5304	5502.8	0.70	5116	5364.40	5.02
	32	8102	8486.6	1.76	7507	7582.9	6.47	7349	7630.5	1.07	7351	7524.90	9.87
	64	10650	10821.1	3.26	9659	9779.1	10.03	9897	9989.1	1.72	9659	9722.30	18.67
	128	13791	14004.3	5.35	12729	12850.0	15.48	12887	13026.8	2.41	12780	12834.10	30.23
ibm12	2	1951	2389.3	0.02	1939	2041.0	0.62	1937	2045.8	0.15	2040	2055.30	0.39
	4	4014	4643.0	0.19	4006	4128.5	1.52	3952	4111.0	0.36	3840	4130.10	1.18
	8	6271	6940.0	0.54	6123	6324.4	2.89	6073	6411.6	0.50	6194	6319.60	2.97
	16	8938	9358.8	1.27	8261	8492.9	4.74	8408	8719.4	0.84	8519	8770.30	5.54
	32	11519	11910.7	2.34	10800	11021.7	7.69	10920	11159.6	1.40	10727	10941.00	11.33
	64	15267	15454.0	4.44	14248	14386.7	12.02	14575	14783.7	2.06	14185	14311.30	21.90
	128	19409	19568.4	6.71	18286	18484.2	18.03	18400	18665.2	2.72	18123	18258.10	36.30
ibm13	2	832	1101.6	0.02	832	832.2	0.52	832	860.5	0.19	832	832.20	0.34
	4	2211	2396.7	0.14	2006	2148.2	1.29	1915	2175.3	0.29	1940	2085.60	1.04
	8	3691	3958.3	0.39	3050	3400.8	2.47	3153	3565.7	0.52	2969	3159.30	2.58
	16	6075	6626.1	1.04	5362	5705.0	4.46	5641	5950.1	0.91	5540	5745.60	5.97
	32	8510	9044.9	2.07	7800	7989.3	7.32	7887	8068.0	1.23	7835	7958.70	11.69
	64	12758	13123.2	3.48	12026	12139.7	11.59	12225	12335.7	1.97	12012	12053.60	22.40
	128	16675	16862.7	11.80	15500	15623.6	35.22	15671	15819.9	2.85	15428	15511.60	35.26
ibm14	2	1914	2216.1	0.03	1890	1924.3	0.80	1891	1954.9	0.18	1891	1927.70	0.50
	4	3624	4226.0	0.21	3402	3446.6	1.88	3357	3469.0	0.40	3344	3417.20	1.40
	8	5802	6489.8	0.69	5139	5334.3	3.59	5165	5394.6	0.68	5170	5290.60	3.60
	16	8812	9537.9	1.53	8407	8538.1	6.18	8457	8669.9	1.20	8330	8527.60	8.05
	32	13255	13799.3	3.20	13025	13169.6	10.20	12906	13220.1	1.98	12823	13079.40	15.74
	64	18532	18885.6	5.62	17634	17773.9	15.89	17746	17870.1	2.79	17505	17666.80	28.90
	128	23719	23901.8	9.53	22363	22509.3	23.94	22524	22734.5	3.94	22192	22408.60	45.95

H	k	KaHyPar-BFS			KaHyPar-Pool			PaToH			hMetis		
		Min	Avg	t	Min	Avg	t	Min	Avg	t	Min	Avg	t
ibm15	2	2814	3433.1	0.03	2747	2762.5	0.76	2747	2777.9	0.16	2747	2762.40	0.50
	4	5409	6388.1	0.21	5043	5131.8	1.81	5060	5370.5	0.47	5039	5167.90	1.48
	8	7676	8543.2	0.68	6712	6970.8	3.43	6858	7232.8	0.71	6777	6975.30	3.58
	16	10066	11003.5	1.44	9037	9359.4	6.04	9239	9576.6	1.34	8716	9206.00	7.70
	32	15685	16224.5	3.13	13497	13837.7	10.27	13964	14327.7	1.95	13833	13963.70	16.19
	64	20377	21027.9	5.90	18657	18857.9	16.61	18874	19168.5	2.93	18655	18891.10	30.74
	128	27280	27545.2	9.43	24913	25251.1	25.88	25307	25611.1	4.50	25003	25205.40	51.27
ibm16	2	1976	2446.2	0.03	1957	2042.6	0.82	2017	2080.5	0.18	1981	2084.40	0.52
	4	4706	5222.0	0.26	4274	4430.7	2.11	4217	4410.3	0.47	4261	4379.20	1.66
	8	7453	8357.1	0.76	6851	7049.2	4.06	6964	7217.9	0.78	6692	6986.10	4.12
	16	11738	12344.0	1.95	10993	11183.2	7.01	10643	11014.1	1.34	10780	11002.70	8.81
	32	16580	17285.8	3.74	15330	15584.4	11.35	15350	15747.0	2.23	15206	15442.20	16.85
	64	22283	22803.8	6.86	20739	20858.2	18.15	20866	21129.7	2.99	20509	20643.20	31.88
	128	28477	28796.0	11.31	26571	26704.6	27.46	26815	27019.5	4.87	26260	26469.70	51.10
ibm17	2	2570	3207.6	0.04	2429	2505.7	1.03	2348	2541.4	0.26	2376	2503.00	0.68
	4	5954	7404.8	0.30	5884	6015.1	2.54	5700	6048.5	0.61	5604	5928.30	2.12
	8	10524	12096.2	0.96	9630	10051.5	4.60	9766	10134.7	1.02	9573	9915.20	4.85
	16	16736	17558.0	2.13	14732	15250.5	7.92	14784	15447.8	1.82	14511	14906.40	10.52
	32	21785	22395.2	4.27	19883	20166.0	13.21	20411	20596.0	2.72	19705	19961.20	20.49
	64	28197	28944.8	7.93	26492	26801.8	20.88	26459	27045.4	3.73	26481	26886.90	37.38
	128	36450	37175.5	13.08	34899	35047.8	32.39	35077	35308.7	5.10	34548	34796.40	63.59
ibm18	2	1742	2176.3	0.03	1548	1838.7	0.68	1542	1705.5	0.16	1542	1746.20	0.51
	4	3345	4599.2	0.16	3081	3279.8	1.63	3089	3313.6	0.43	3098	3220.00	1.56
	8	6626	7315.5	0.50	5757	6010.7	3.34	5902	6154.3	0.72	5819	6031.90	4.02
	16	9427	10591.8	1.28	8740	8963.3	5.78	8807	9022.0	1.25	8724	8853.10	8.99
	32	14195	15165.1	2.75	13017	13224.8	9.76	13022	13243.8	1.85	13008	13323.40	17.29
	64	19871	20328.1	4.40	18271	18578.6	15.18	18093	18689.9	2.81	18224	18483.20	32.16
	128	25529	26188.3	7.65	24161	24357.6	23.19	24465	24585.1	3.97	24154	24258.50	54.12
kkt_power	2	10097	11294.5	0.26	10097	10923.6	4.38	10097	10933.7	0.69	10097	11049.80	1.89
	4	17307	21066.5	3.38	17291	19012.1	11.85	18693	19315.5	1.79	17291	18972.60	6.30
	8	33999	38914.1	9.41	31879	35168.7	26.17	33397	35589.1	3.33	33270	34824.00	12.18
	16	58714	71761.2	19.44	62532	68753.7	45.70	60242	66638.4	6.01	57238	59887.40	20.40
	32	104305	113071.0	35.26	93112	99893.5	72.18	96923	104737.5	10.70	93040	96507.10	36.45
	64	163317	170931.0	53.21	149574	153735.7	105.49	157286	163453.7	15.88	142552	147647.20	66.10
	128	233381	241166.4	79.58	217425	222496.3	146.91	225001	230578.0	25.60	206270	211111.10	105.19
ldoor	2	3164	3364.2	0.01	3164	3217.9	0.09	3108	3201.1	0.06	3136	3220.70	0.08
	4	6482	7337.4	0.02	6279	6384.7	0.27	6251	6508.6	0.09	6328	6439.30	0.20
	8	11305	13602.4	0.05	11452	11872.7	0.63	11830	12059.6	0.20	11389	11865.70	0.51
	16	22323	23832.2	0.13	19873	20284.6	1.37	20041	20633.9	0.35	20097	20553.40	1.50
	32	37373	38491.2	0.31	34454	34983.2	2.92	33635	34601.7	0.72	34195	34890.10	3.70
	64	57533	59372.6	0.83	53046	53897.2	6.23	52843	53482.1	1.42	52591	53498.90	9.52
	128	87514	88840.8	2.16	80717	81623.5	13.27	81046	81755.1	3.21	79877	81332.30	21.85
memplus	2	2917	2966.0	0.00	2908	2925.4	0.13	2907	2921.1	0.03	2906	2916.60	0.11
	4	4834	4934.2	0.04	4477	4514.7	0.23	4215	4445.6	0.04	4200	4901.90	0.27
	8	5453	5599.4	0.18	5180	5273.9	0.53	5065	5157.4	0.07	5536	5850.80	0.52
	16	5910	6129.4	0.54	5807	5831.8	1.12	5648	5773.4	0.28	6375	6540.90	1.07
	32	6642	6797.4	1.78	6499	6539.2	5.80	6271	6356.2	0.27	6528	6570.80	2.00
	64	7118	7179.4	3.19	6915	6979.8	5.26	6946	6978.7	0.60	6960	7009.60	3.95
	128	7409	7486.5	4.48	7196	7241.1	8.60	7288	7288.0	0.66	7298	7346.40	8.90
nlpkkt120	2	58560	59814.1	0.04	58810	60278.1	0.51	58800	59629.9	0.31	58560	60223.30	0.37
	4	119200	123423.0	0.16	118370	121379.0	1.57	119156	121638.3	0.60	120070	122772.10	1.17
	8	181994	198829.1	0.62	179820	183881.7	3.87	180587	182220.6	1.62	180462	182893.50	3.72
	16	292773	303178.1	1.98	289862	294259.7	9.84	290233	294332.0	4.07	288628	293564.70	11.53
	32	413436	419753.2	6.40	409959	412907.9	23.89	411192	413526.5	9.73	408704	413523.90	37.45
	64	538054	546594.6	19.66	524848	532066.3	59.03	528560	536923.7	26.18	530373	535203.70	113.58
	128	733731	742871.1	57.94	726004	731651.7	144.44	727704	731869.3	60.91	725136	731231.80	297.85
nlpkkt160	2	104532	107071.0	0.08	103880	105871.1	0.51	104112	106388.8	0.42	104112	105955.60	0.47
	4	216792	221293.4	0.19	216118	221155.8	1.48	215170	221128.9	0.84	214174	220290.40	1.15
	8	320640	334096.0	0.54	326658	328697.5	3.55	324522	330068.9	1.97	324129	329013.00	3.38
	16	525727	545376.8	1.70	514111	526928.9	8.57	521218	530292.2	4.75	520877	528143.60	10.60
	32	748372	760636.4	5.15	738114	745897.7	20.44	736915	748235.3	10.36	734200	744422.90	32.07
	64	968937	991546.9	15.39	955482	962698.6	49.52	954638	970354.4	23.57	958454	965764.11	90.54
	128	1338487	1345748.6	45.80	1320797	1326574.4	123.87	1321435	1329646.3	59.27	1320443	1328933.20	250.45
thermal2	2	930	1499.8	0.01	929	936.7	0.13	930	936.8	0.05	930	936.80	0.07
	4	2801	3796.2	0.02	2793	2806.7	0.32	2795	2807.1	0.07	2794	2809.90	0.14
	8	6641	7160.7	0.03	6536	6568.8	0.64	6557	6575.2	0.12	6541	6569.10	0.32
	16	11368	11938.9	0.07	10794	10932.5	1.17	10863	10934.4	0.25	10880	10935.80	0.92
	32	18197	18955.7	0.22	17893	18147.8	2.56	18019	18189.2	0.53	18067	18202.40	2.99
	64	28380	28817.7	0.63	27030	27226.1	5.78	26957	27146.0	1.22	26980	27113.20	8.08
	128	42605	43015.0	1.64	40817	40991.4	11.85	40676	40830.8	2.55	40783	40974.90	18.50
vibrobox	2	2491	2497.7	0.02	2491	2497.5	0.59	2489	2493.5	0.15	2489	2498.20	0.55
	4	3858	4306.2	0.21	4030	4176.8	1.27	4123	4343.4	0.27	4061	4202.90	1.39
	8	5108	5468.3	0.64	4699	4849.4	2.37	4968	5155.4	0.39	4746	4853.30	2.88
	16	5757	6270.8	1.62	5667	5936.4	4.09	5780	5979.4	0.65	5188	5412.80	5.56
	32	7041	7283.4	2.76	6849	6927.8	6.86	6862	7013.4	0.95	6818	6907.90	10.62
	64	8071	8201.0	5.27	7658	7745.4	11.17	7818	7916.4	1.39	7756	7820.40	19.84
	128	9433	9522.9	7.27	8633	8728.4	15.69	9011	9011.0	1.75	8751	8811.50	27.30