n-Level Hypergraph Partitioning

Florian Ziegler

July 27, 2012

1457745

Bachelor Thesis

at

Institute of Theoretical Informatics, Algorithmics II
Karlsruhe Institute of Technology

Supervisors:

Prof. Dr. rer. nat. Peter Sanders,

Dipl.-Math. Dipl.-Inform. Christian Schulz,

M.Sc. Vitaly Osipov

Acknowledgments

I would like to express my thanks to all of the people who were involved in my thesis. Their suggestions were invaluable and their help was much appreciated. Without their support this thesis would have had to stay incomplete. Foremost I would like to thank my supervisors Prof. Dr. rer. nat. Sanders, Dipl.-Math. Dipl.-Inform. Christian Schulz and M.Sc. Vitaly Osipov.
My family and friends were of great help to me and a constant source of motivation - I would like to thank them, too.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, 27.07.2012

Florian Ziegler

Abstract

We present a n-Level Hypergraph Partitioner based on the idea of the n-Level Graph Partitioner by Osipov and Sanders. Given a Hypergraph H we compute a 2-partition P that satisfies a given imbalance constraint. To achieve this we coarsen the original hypergraph in multiple levels. At each level we contract a hyperedge, i.e. merging its hypernodes in one to decrease the number of hypernodes and hyperedges. At the lowest level we construct a bisection P and reverse the coarsening. At each level we additionally refine P. We also implement V-Cycles to improve the quality of P further.
We present an extensive experimental evaluation with hypergraph instances widely used in literature, for example parts of the ISPD and MCNC benchmark suites. We compare our system with the state-of-the-art hypergraph partitioners *hMETIS* and *PaToH*. Our tuned system achieves hyperedge cuts which are on average as good as those of *PaToH*. However the runtime of our system is much worse that the one of the competition. Furthermore, our system finds the partition with the smallest hyperedge cut for circa half of the test suite, when compared to the standard presets of *hMETIS* and *PaToH*.

Contents

1 Introduction

1.1 Motivation

Hypergraphs are a generalization of graphs. The main difference is that the edges of hypergraphs can connect more than two nodes.

An important application of hypergraphs is the field of integrated circuit design [3]. Besides that, hypergraphs are used for reducing the communication volume for parallel sparse-matrix vector multiplication [6], for dynamic load balancing for scientific computations [7], etc. Hypergraph Partitioning is an important part of these applications.

There are schemes to model hypergraphs by graphs. However, it has been shown by Ihler et al. [14], that the original cut properties cannot be retained with these models. This causes the necessity to use special hypergraph partitioning algorithms. The problem of partitioning a hypergraph is at least NP-hard [9].

State-of-the-art graph and hypergraph partitioners [24, 20, 5, 18] use the multi-level approach. At each level the graph is coarsened by selecting multiple groups of nodes and contracting each of those into one node of a coarser graph. All nodes of a group are combined into one node carrying the combined weight of its predecessors. This node is then connected to all edges incident to the original nodes. This procedure is repeated until the graph is small enough for a fairly expensive initial partitioning method.

The now much smaller graph is partitioned at the lowest level and then uncoarsened, i.e., the coarsening is reversed and level by level the original graph is restored. The partitioning information is passed on to higher levels. At each level a local refinement algorithm is used to minimize the cut.

A variant of the multi-level method is n-Level Graph Partitioning [20]. It differs from the common multi-level approach as it only contracts one single edge at each level. Sanders and Osipov [20] have shown, that this leads to both improved runtime and better quality of the resulting partitions.

1.2 Our Contribution

In this thesis we explore whether the n-Level partitioning approach is suitable for the hypergraph bisection problem. In essence our method contracts only one single hyperedge at each level of the coarsening phase. This is done by computing a rating function for each hyperedge and contract on each level the hyperedge with the highest rating. As the choice of the hyperedge is essential to the properties of the final contracted graph, multiple rating functions are experimentally evaluated.

As soon as the hypergraph is small enough the coarsening phase ends and we partition the hypergraph with the *PaToH* partitioner.

During uncoarsening the hypergraph is repeatedly refined at each level with a heuristic similar to the Fiduccia-Mattheyses Heuristic.
As in *hMetis* the method is then extended by the V-Cycle approach that performs additional coarsening and refinement phases (with certain modifications) to further improve the quality of the found partition.

1.3 Outline

The thesis is organized as follows: We begin in Chapter 2 with formal definitions of hypergraphs and a proper definition of the hypergraph partitioning problem.
In Chapter 3 we present related work, the concept of multi-level partitioning and refinement heuristics. Additionally, we look at *PaToH*, as we use it in the initial partitioning phase.
We then move forward to describe our n-level hypergraph partitioning approach in Chapter 4. This includes a description of rating functions for the coarsening phase.
In Chapter 5 we perform parameter tuning and thereupon evaluate our algorithm experimentally. We conclude with a summary of the results in Chapter 6.

2 Fundamentals

In this part we give formal definitions of a hypergraph, the partitioning problem and the metrics used to measure the quality of a hypergraph partition. The notation given here will be used in the following chapters.

2.1 General Definitions

A hypergraph $H = (V, \mathcal{E}, c, w)$ is defined as a set of vertices V and a set of hyperedges $\mathcal{E}$. We use the term *hypernode* synonymously for the term *vertex*. Each hyperedge is a non-empty subset of V. The hypergraph has hyperedge weights $w : \mathcal{E} \to \mathbb{R}_{>0}$ and hypernode weights $c : V \to \mathbb{R}_{\geq 0}$. We extend c and w to sets $V' \subseteq V$ and $\mathcal{E}' \subseteq \mathcal{E}$, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $w(\mathcal{E}') := \sum_{e \in \mathcal{E}'} w(e)$.

Definition 1. *Hypergraph Partitioning Problem*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$ and $k \in \mathbb{N}_{>1}$, the hypergraph partitioning problem is to partition V into k subsets $V_1, \ldots, V_k$ such that:

1. $V_1 \cup \cdots \cup V_k = V$
2. $V_i \cap V_j = \emptyset$
3. $\forall i \in \{1 \ldots k\} : c(V_i) \leq L_{max} := (1 + \epsilon) \left\lceil \frac{c(V)}{k} \right\rceil$ for the maximum imbalance ratio ϵ.
4. some cut metric is minimized (see Section 2.2)

A k-partition of H is a mapping $P : V \to \mathbb{N}$ which assigns each hypernode $v \in V$ a partition $P[v] \in \{1, \ldots, k\}$. Having partitioned the hypernodes V into k disjoint parts we can define a k-partition implicitly by $P[v \in V_i] = i$. We do not distinguish between a k-partition and the equivalent partition of hypernodes V and we call V_i a block.

Definition 2. *Independent hyperedges*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$, hyperedges $e_1, \ldots e_m \in \mathcal{E}$ are called *independent* $\Leftrightarrow \forall i, j \in \{1 \ldots m\}, i \neq j : e_i \cap e_j = \emptyset$. Informally speaking they do not share any hypernodes.

Definition 3. *Incidence between hyperedges and hypernodes*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$, a hyperedge $e \in \mathcal{E}$ and a hypernode $v \in V$ are called incident $\Leftrightarrow v \in e$.

Definition 4. *Border Hyperedge*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$ and a k-partition P of H, a hyperedge e is a border hyperedge $\Leftrightarrow \exists v_i, v_j \in e : P[v_i] \neq P[v_j]$. All border hyperedges of H form the set B.

Definition 5. *Parallel Hyperedges*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$, hyperedges $e_1, e_2 \in \mathcal{E}$ are called *parallel* $\Leftrightarrow e_1 = e_2$. Informally speaking two hyperedges are parallel if they are incident to the same hypernodes. *hMETIS* defaults to 5% and *PaToH* to 2% maximum imbalance.

Definition 6. *Nested Hyperedge*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$, hyperedges $e_1, e_2 \in \mathcal{E}$, $e_1 \neq e_2$, e_1 is called *nested* into $e_2 \Leftrightarrow e_1 \subseteq e_2$. Parallel hyperedges are nested hyperedges, too.

2.2 Quality Metrics

We present some cut metrics which are used to measure the quality of a hypergraph partition. These metrics are employed for example in the VLSI domain [17].

Definition 7. *Hyperedge Cut*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$ with the set of border hyperedges B of a k-partition P of H, the hyperedge cut is defined as

$$HEC\left(P\right) = \sum_{e \in B} w(e) = w\left(B\right)$$

Thus the hyperedge cut is the weight of all border hyperedges summed up.

Given a hypergraph $H = (V, \mathcal{E}, c, w)$ and a k-partition P of H, a hyperedge e is said to connect two blocks $i, j \in \{1 \ldots k\}, i \neq j \Leftrightarrow \exists v, w \in e : P[v] = i \wedge P[w] = j$. The number of blocks which is connected by a hyperedge e is defined as $\lambda = |\bigcup_{v \in e} \{P\left[v\right]\}|$

Definition 8. *Sum Of External Degrees*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$ with the set of border hyperedges B of a k-partition P of H, we define the sum of external degrees

$$SOED\left(P\right) = \sum_{e \in B} w(e)\lambda$$

Definition 9. $\lambda - 1$ *Metric*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$ with the set of border hyperedges B of a k-partition P of H, we define the $\lambda - 1$ metric

$$LMO\,(P) = \sum_{e \in B} w(e)(\lambda - 1) = \sum_{e \in \mathcal{E}} w(e)(\lambda - 1)$$

This thesis only deals with hypergraph bisection ($k = 2$), thus for a given hypergraph $H = (V, \mathcal{E}, c, w)$ with the set of border hyperedges B and a 2-partition P of H the following equation holds:

$$HEC(P) = 2 \cdot SOED(P) = LMO(P)$$

Definition 10. *Gain in the 2-partition case with HEC*

Given a hypergraph $H = (V, \mathcal{E}, c, w)$, a hypernode $v \in V$, a 2-partition P and a 2-partition Q with $Q[v'] = P[v']$ for all $v' \in V \setminus \{v\}$ and $Q[v] = (P[v] + 1) \bmod 2$, we define *gain(v)* $:= HEC(Q) - HEC(P)$. Figuratively: *gain(v)* is the change of the Hyperedge Cut when switching the block of v.

In the remainder of this thesis we use the Hyperedge Cut metric exclusively.

2.3 Visual Representation of a Hypergraph

Figure 1 shows a sample hypergraph $H = (V, \mathcal{E}, c, w)$ with $\mathcal{E} = \{e_1, e_2, e_3\}$, $V = \{v_1, v_2, \ldots v_{12}\}$, $e_1 = \{v_5, v_8, v_9, \ldots v_{12}\}$, $e_2 = \{v_1, v_2, \ldots v_8\}$ and $e_3 = \{v_6, v_7, v_8, v_{11}, v_{12}\}$. Hyperedges are represented by ellipses and the black circles depict the hypernodes. Colors are used only as a visual aid. The weight of the hyperedges and hypernodes is right of the vertical bar. In this example the functions w and c are defined as follows:

$$w(e) = \begin{cases} 1, & \text{if } e \in \{e_2, e_3\} \\ 2, & \text{if } e = e_1 \end{cases} \qquad c(v) = \begin{cases} 1, & \text{if } v \in \{v_2, v_3, \ldots, v_6, v_8, v_9, v_{11}\} \\ 2, & \text{if } v \in \{v_1, v_7\} \\ 3, & \text{if } v \in \{v_{10}, v_{12}\} \end{cases}$$

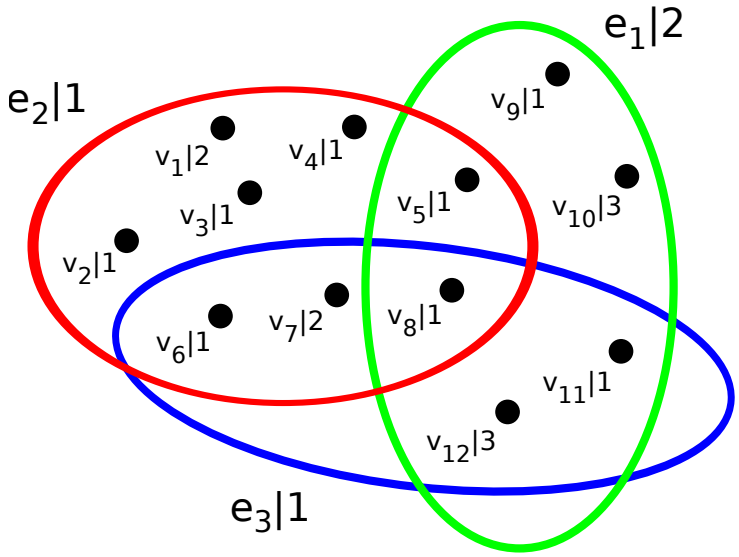


Figure 1: Sample Hypergraph

3 Related Work

In this Section we introduce the Kernighan-Lin and the Fiduccia-Mattheyses Heuristic and an overview of Multi-Level Hypergraph Partitioning with two real world implementations (*hMETIS* and *PaToH*) which build upon it.

3.1 Kernighan-Lin Heuristic

Originally the Kernighan-Lin (KL) Heuristic [19] was designed for graph partitioning but it is easily adapted to hypergraphs [25]. KL is a local search method which improves a given perfectly balanced ($\epsilon = 0$) 2-partition P in multiple *passes*. A gain function is defined which returns for a hypernode v by how much the hyperedge cut would decrease if v was moved to the opposite block. Gains may be negative. Then two hypernodes of different blocks are picked which together maximize the gain when switched. If these two hypernodes are incident to the same hyperedges the algorithm takes this into account when computing the total gain of the switch. If a hypernode was moved it is not eligible for moving again in this pass. Pairs of hypernodes switch blocks until all hypernodes have been moved once. The algorithm now rolls back to the switch which yielded the best global gain. As long as a better or equal 2-partition resulted from a pass, another pass takes place.

3.2 Fiduccia-Mattheyses Heuristic

The Fiduccia-Mattheyses (FM) Heuristic [8] or derived heuristics are building blocks of many state of the art hypergraph partitioners. FM is a local search method which improves a given (balanced) 2-partition P in multiple *passes*. Many of its ideas are taken from the KL heuristic. A gain function is defined which returns for a hypernode v by how much the hyperedge cut would decrease if v was moved to the opposite block. Gains may be negative.

At the beginning of a pass all hypernodes are marked as **free**. As long as free hypernodes remain the following procedure is repeated: The free hypernode with the highest gain that does not lead to a violation of the balance constraint when moved, is picked and moved to its opposite block and marked as **locked**. This leads to a 2-partition P'. Remark: After the move the gains of adjacent hypernodes change. As the gains may be negative, the overall cut of the resulting 2-partition may be greater than previous cuts.

If no more hypernodes can be moved, the 2-partition P'' with the lowest cut encountered in the pass is chosen as the basis of the next pass.

The FM heuristic usually converges after a couple of passes and the worst case running time is linear to the size of the hypergraph, which is a significant improvement over the KL heuristic. The other important difference between the two heuristics is that KL *swaps* two hypernodes whereas FM moves only one hypernode at a time.

3.3 Multi-level Hypergraph Partitioning

State-of-the-art hypergraph partitioners like hMetis [13] and PaToH [22] use a multi-level approach. This scheme divides the partitioning process into three phases (see Figure 2).

1. Coarsening Phase
2. Initial Partitioning
3. Uncoarsening and Refinement Phase

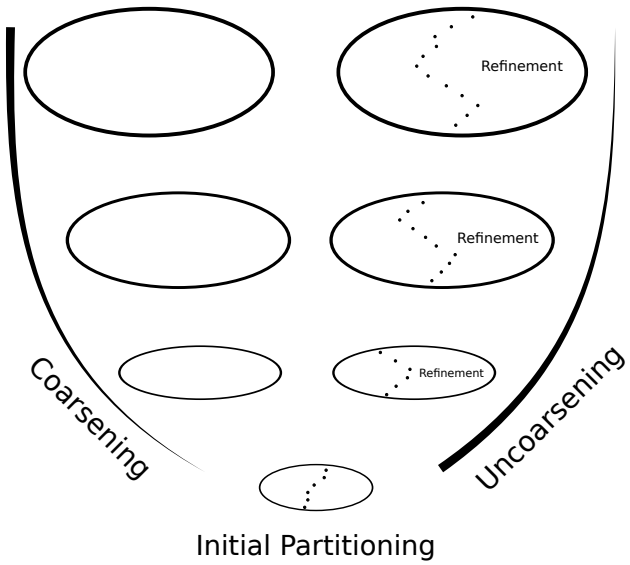


Figure 2: Multi-Level Hypergraph Partitioning

The idea behind this division is to reduce the number of hypernodes and hyperedges in the coarsening phase significantly in order to decrease the cost of the initial partitioning. The coarsening is then reversed to obtain a 2-partition of the initial hypergraph.

The coarsening phase is itself divided into multiple steps, the so called levels, thus the name "multi-level". At each level groups of hypernodes are each merged into only one hypernode. Hyperedges, which only contain one hypernode or hyperedges that are parallel to another hyperedge after this merging are removed from the hypergraph. The hypernode which represents a merged group carries the weight of all hypernodes of the group.

As one hypernode at a coarse level may represent very many hypernodes of a finer level, we have a rather global view on the optimization problem at the coarse levels. Whereas, on the finer levels the view is confined rather locally.

The process of compiling and merging groups is repeated until the hypergraph is small enough for an efficient computation of an initial partition. In the second phase the initial partitioning takes place. The coarsened hypergraph is often partitioned by a heuristic as the loss of topological information in the coarsening phase can not be compensated by an optimal initial partition. For more information on initial partitioning we refer the reader to Bichot and Siarry [4].

The third phase consists of uncoarsening the hypergraph back to its initial state. At each level the 2-partition is refined locally by moving hypernodes between the blocks in order to decrease the hyperedge cut.

3.4 hMETIS

hMETIS is a multilevel hypergraph partitioner that originates form the University of Minnesota. It was implemented and improved multiple times by George Karypis et al. [17][18]. *hMETIS* operates directly on the hypergraph without first converting it into a graph.

For the coarsening of a hypergraph *hMETIS* provides multiple ways to choose the hypernodes which form one group (which are then merged together). We outline three of them: **Hyperedge Coarsening** (HEC), **Modified Hyperedge Coarsening** (MHEC) and **Edge Coarsening** (EC). We follow closely the description of Karypis, Aggarwal and Kumar [18].

3.4.1 Hyperedge Coarsening

At each level, the **Hyperedge Coarsening** scheme finds a set of hyperedges such that all of them are independent and no additional hyperedge can be added without two hyperedges of the set becoming dependent. All hypernodes incident to one hyperedge of the set form one group and are merged together. This is also called contraction of a hyperedge. As the hyperedges are independent there is no overlap between the groups.

3.4.2 Modified Hyperedge Coarsening

The **Modified Hyperedge Coarsening** scheme is derived from the **Hyperedge Coarsening** scheme. MHEC adds another step for each level to HEC. It takes place after the hyperedges selected by HEC have been contracted. All hyperedges which were not contracted are examined. All hypernodes of a hyperedge which are not the result of the HEC part of this phase are merged.

3.4.3 Edge Coarsening

In the **Edge Coarsening** scheme initially the weight function w is defined as follows:

$$w(e) = \frac{1}{|e| - 1} \qquad e \text{ is a hyperedge}$$

At the beginning of each level all hypernodes are *unmatched*. The unmatched hypernodes are visited in a random order. For a unmatched adjacent hypernode u of a hypernode v the *rating* is the sum of all weights of all hyperedges connecting u and v is computed. The hypernode which is connected by the highest rating constitutes a group with v. Both hypernodes are marked as matched. Each group thus consists of two hypernodes and no hypernode is in more than one group. This process is repeated until no more pair of hypernodes can form a group. It is possible that unmatched hypernodes remain.

3.4.4 Initial Partitioning

hMETIS provides two algorithms for initial partitioning. The first one creates a random 2-partition which is then refined by the FM refinement algorithm.

The second initial partitioning algorithm randomly picks a hypernode v. A breadth-first-search is performed from v and the found hypernodes are added to the block of v until half of the hypernode weight of the initial hypergraph is in this block. The so composed 2-partition is also refined by the FM algorithm.

3.4.5 Uncoarsening and Refinement

In the uncoarsening phase *hMETIS* reverses the coarsening phase level by level. It restores the groups of hypernodes which were contracted and refines the result. *hMETIS* provides two refinement algorithms. The first is a heuristic very similar to FM with two exceptions. The number of passes is restricted to 2 and a pass stops early when $l = 0.01N_V$ moves did not improve the cut. N_V is the current number of hypernodes.

The second refinement algorithm is called *Hyperedge Refinement* (HER). HER visits all border hyperedges in a random order. For a border hyperedgee the gains by moving all its hypernodes to one block are computed. The hyperedge is moved to the block with the greatest cut improvement given that the balance constraint is not violated.

3.4.6 Multiphase Refinement

hMETIS supports the so called *multiphase refinement* technique. The idea behind this technique is to restart coarsening in or after the uncoarsening phase. Although one modification is applied: only hypernodes of the same block are merged together. If another coarsening phase is started after the hypergraph was completely uncoarsened, it is called a *V-Cycle*. If the uncoarsening is restarted in the middle of uncoarsening it is called a *v-Cycle*. In these additional coarsening phases other coarsening schemes may be used. v-Cycles are cheaper as the hypergraph is smaller at the time of invocation. We implement the V-Cycle technique, too (see Section 4.6).

3.5 PaToH

PaToH is a hypergraph partitioner developed by Catalyurek and Aykanat. The program provides a multitude of coarsening, initial partitioning and refinement algorithms. For an exhaustive description of the algorithms used therein see the PaToH manual [21]. We restrict ourselves to the description of the coarsening and refinement algorithms which are used by default. For a description of the initial partitioning algorithms see Catalyurek and Aykanat [5].

3.5.1 Coarsening

PaToH employs a coarsening method similar to edge coarsening of *hMETIS*. With edge coarsening in *hMETIS* a group consists of maximal two hypernodes, *PaToH* permits more than two hypernodes in a group – *PaToH* calls them **cluster**. All hypernodes are visited in a random order and are matched with one of their adjacent hypernodes and these two hypernodes form a cluster. If the currently visited hypernode v is matched with a hypernode which is already part of a cluster, v is simply added to this cluster. Thus clusters of more than two hypernodes may be contracted together.

3.5.2 Refinement

PaToH uses its custom refinement scheme: One pass of a modified FM (Boundary FM) heuristic is followed by one pass of a modified KL (Boundary KL) heuristic. Boundary FM only moves hypernodes which are incident to a border hyperedge and then only from the heavier block to the other. Boundary KL has the same constraints as Boundary FM. This means that both Boundary FM and Boundary KL are local refinement heuristics in contrast to the original FM and KL heuristics which operate globally.

4 n-Level Hypergraph Partitioning

We start this Section with an overview of n-level hypergraph partitioning in Section 4.1 and carry on with a detailed description of the coarsening phase, the initial partitioning phase and the refinement phase in Sections 4.2, 4.3 and 4.4 respectively. Section 4.5 introduces various rating functions which are used.

n-level hypergraph partitioning closely resembles the multi-level approach. The difference is in the number of hyperedges which are contracted at each level. While a classic multi-level scheme contracts multiple independent hyperedges (or groups of hypernodes) at a level, our algorithm contracts only one single hyperedge at each level.

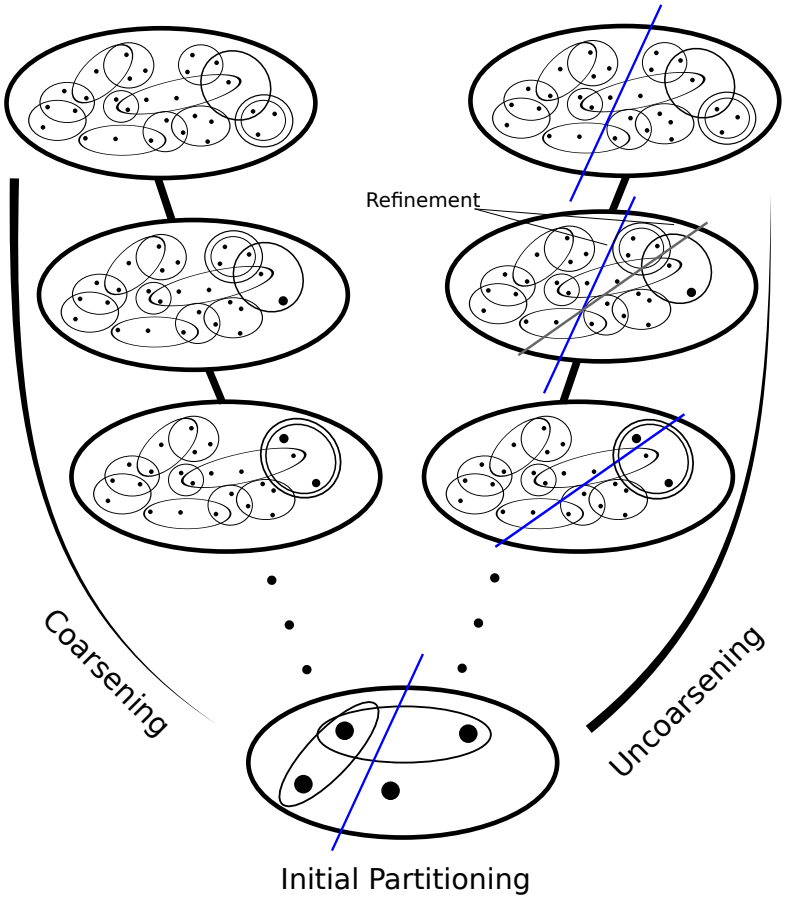


Figure 3: n-Level Hypergraph Partitioning. Double encircled hyperedges are either contracted (left hand side) or have been uncontracted in the previous step (right hand side).

4.1 General Description

Given a hypergraph $H = (V, \mathcal{E}, c, w)$ without any parallel edges, we decrease the size of H, i. e. the size of the sets $\mathcal{E}$ and V by contracting **one single** hyperedge at a time. The necessary information to reverse this process is stored in an appropriate form on a stack S for later use.

This coarsening phase is stopped when number of hypernodes falls below a certain threshold. H is then partitioned initially which yields a 2-partition P.

The steps to coarsen H are reversed with the help of the information stored on the stack S. The block information is preserved and at each level the projected 2-partition is refined by a modified FM heuristic. An outline of the algorithm is shown in Algorithm 1 and illustrated in Figure 3.

Algorithm 1 PARTITION

Require: Hypergraph $H = (V, \mathcal{E}, c, w)$, rating function *rate*

```
COARSEN(H, rate) // coarsen H by contracting hyperedges
Initially Partition H into two blocks
UNCOARSEN(H) // refine at each level
```

4.2 Coarsening Phase

The hypergraph H is coarsened by iteratively contracting single hyperedges. To decide which hyperedge to contract next, we use a rating that is associated with each hyperedge. These ratings are computed by a so called **rating function**. In Section 4.5 we explain the importance of rating functions for the hypergraph partitioning problem and describe various rating functions.

Algorithm 2 COARSEN(H, *rate*)

Require: Hypergraph $H = (V, \mathcal{E}, c, w)$, rating function *rate*

```
initialize addressable priority queue PQ
if tie-breaking then
    randomly shuffle E
for all e ∈ E do
    PQ.insert(e, rate(e))
while PQ is not empty and |V| > t do
    e := PQ.deleteMax()
    incident_node_weight := Σ_{v∈e} c(v)
    if incident_node_weight ≤ s then
        CONTRACT(H, e, PQ, rate)
```

As depicted in Algorithm 2 coarsening starts with randomly shuffling the hyperedges if tie-breaking is used. Tie-breaking changes the order in which equally rated hyperedges are inserted into the priority queue PQ. Then all hyperedges are rated according to a chosen rating function and inserted into a addressable priority queue. This is necessary as the contraction of one hyperedge may change the rating of the adjacent hyperedges.

There are two parameters of importance in this phase. First of all the threshold t which defines the number of hypernodes that should remain in the coarsest hypergraph. To avoid to heavy hypernodes on the coarse levels, we only allow contractions of hyperedges with a combined hypernode weight smaller than or equal to s times the initial number of hypernodes of H. t and s are tuning parameters (see Section 5.1.3).

While the priority queue is not empty and the number of hypernodes of H is greater than the threshold t the hyperedge e with the maximum rating is removed from the priority queue and contracted, if the sum of the weight of the incident hypernodes of e is less than or equal to s.

4.2.1 Contracting a Single Hyperedge

Algorithm 3 shows how a single hyperedge e is contracted. A sample contraction is depicted in Figure 4. The first step of the contraction of e is to remove all hyperedges nested into e from H and PQ. This is reflected in Figure 4b where e_4 is removed.

After that, one hypernode $r \in e$ is picked as the representative of the contracted hyperedge in later levels. The weight of r is set to the accumulated weight of all hypernodes incident to e. Additionally all adjacent hyperedges to e are connected to r if they were not already connected to r. In our example (Figure 4c) v_3 is picked as the representative and its weight is changed to $c(v_1) + c(v_2) + c(v_3) = 3$. The hyperedges e_5 and e_3 are connected to v_3.

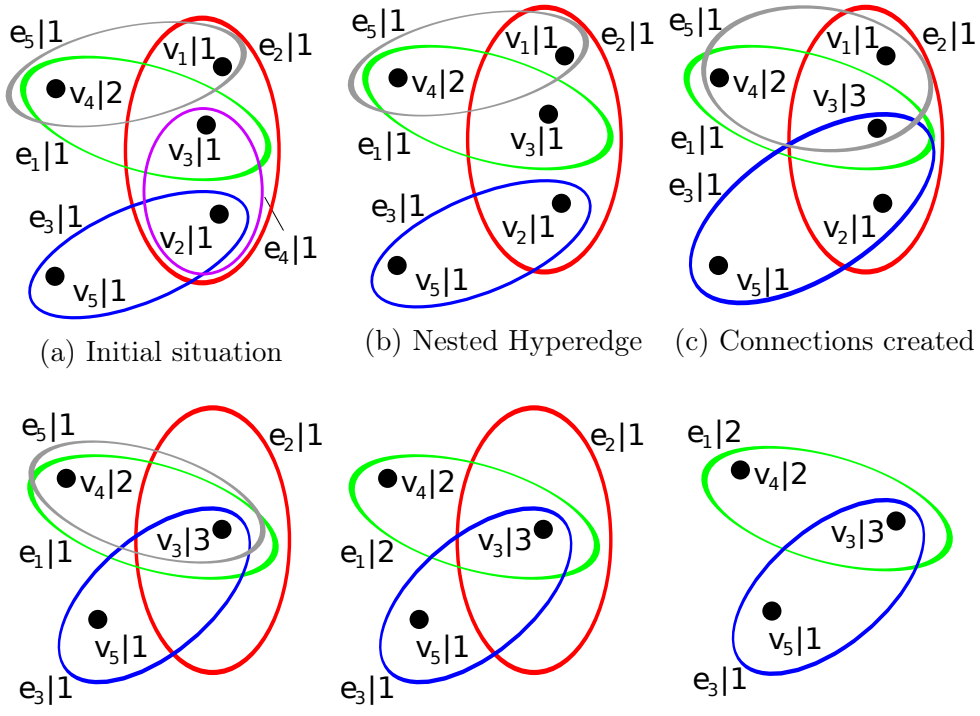
The hypernodes incident to e with the exception of r are removed from the hypergraph. Figure 4d visualizes this step. It is clear that the choice of the representative r may be arbitrary as the changes that would be made to H if another hypernode was picked would lead to a similar change in topology.

The removal of these hypernodes may lead to **parallel hyperedges** as it is the case in our example (Figure 4d with the hyperedges e_5 and e_1). Removing these parallel hyperedges takes several steps. First we find for all adjacent hyperedges those which are parallel to each other and put them into a set PES (Parallel hyperedge set). As there can be more than of these sets, we create a container $PESS$ (Set of parallel hyperedge sets) to held them. After that the following procedure is applied to each $PES \in PESS$: Remove the first hyperedge *per* and add the weight of all other hyperedges to the weight of *per*. Then remove all hyperedges but *per* from H and PQ.

Algorithm 3 CONTRACT(H, e, PQ, $rate$)

Require: hypergraph $H = (V, \mathcal{E}, c, w)$, hyperedge $e \in \mathcal{E}$, addressable priority queue PQ, rating function $rate$

```
for all hyperedges ne nested into e do
    PQ.remove(ne)
    H.remove(ne)

r ← pick one hypernode of e
w ← 0

for all hypernodes n ∈ e do
    w ← w + c(n)
c(r) := w

for all hyperedges ae adjacent to e do
    if ae is not incident to r then
        H.connect(ae, r)

for all hypernodes n ∈ e do
    if n ≠ r then
        H.remove(n)

initialize set of sets PESS
insert sets of parallel hyperedges adjacent to e into PESS

for all PES ∈ PESS do
    per ← remove first hyperedge from PES
    for all pe ∈ PES do
        w(per) ← w(per) + w(pe)
        PQ.remove(pe)

H.remove(e)

for all hyperedges ae adjacent to e do
    PQ.update(ae, rate(ae))
```



(a) Initial situation (b) Nested Hyperedge (c) Connections created

(d) Hypernodes removed (e) Parallel Hyperedge (f) Hyperedge removed

Figure 4: Contraction of hyperedge e_2 in a sample hypergraph with v_3 as the representative.

We now describe an efficient way to compute the set $PESS$. A *fingerprint* is a quadruple consisting of a hash value, a counter value, a boolean value *matched* and an unique hyperedge identifier. We set up an array of fingerprints whose length is the number of adjacent hyperedges of the representative hypernode r. The array entries are initialized by performing the following procedure for all adjacent hyperedges of r: Let the hyperedge be e. A counter c is set to the number of adjacent hypernodes of e and the identifiers of the adjacent hypernodes are XORed into a hash value. An arbitrary constant value is used as seed. The counter and this hash value form together with the identifier of e and the initial value `false` for *matched* the fingerprint for e.

After all hyperedges have been processed, the now fully initialized fingerprint array is sorted by the hash values of the entries. Parallel hyperedges are in neighboring slots of the fingerprint array. A loop iterates over all fingerprints from the start of the array. For each unmatched fingerprint f all unmatched fingerprints with the same hash and the same counter are considered as a candidate for parallel edges to the hyperedge represented by f. A check is performed which involves comparing the hypernodes of f and each candidate to filter out the false positives. All fingerprints representing

parallel hyperedges are marked as matched thereby prohibiting that they are later considered again.

Although sorting could be avoided by using a hash table instead of an array, the time used for allocation, clearing/deallocation of hash tables is in practice greater than the time used for sorting.

4.3 Initial Partitioning Phase

We use *PaToH* as our initial partitioner. For a rough description of the *PaToH* algorithm see Section 3.5.

PaToH is called z times with varying random seeds to generate different 2-partitions of H. We pick the 2-partition P with the smallest hyperedge cut as the initial partition of H. We do not check P for imbalance as we rely on the initial partitioner and our refinement phase for keeping the partitions balanced. z is a tuning parameter (see Section 5.1.3). The exact arguments used for PaToH are documented in Section 5.3.1.

4.4 Uncoarsening and Refinement Phase

As depicted in Algorithm 4 the hypergraph is uncoarsened to its original state by uncontracting the hyperedges which were contracted previously in reverse order and projecting the 2-partition up to the higher levels. The stack S holds the necessary information for this step. After each uncontraction the projected 2-partition is refined heuristically.

Algorithm 4 Uncoarsening and Refinement phase

Require: H is a hypergraph with $H = (V, E, c, w)$, S a stack with sufficient information to reverse the coarsening phase

```
function UNCOARSEN(H, S)
    while S is not empty do
        s := S.pop() // S holds a datastructure to reverse
                     // the contraction of a hyperedge
        e := s.contracted_hyperedge
        UNCONTRACT(H, s, e)
        REFINE(H, e)
```

4.4.1 Uncontracting a Hyperedge

In essence the steps taken in Algorithm 3 are reversed to uncontract a hyperedge. In the following description we assume, that all information that is needed to restore hyperedges, hypernodes, weights etc. is saved on stack s. We use the term "restore" a hypernode or hyperedge to indicate that the hypernode/hyperedge is readded to H and connected to all hyperedges/hypernodes which it ws connected to at the time of its removal.

First, the removed hyperedge e is restored (and connected to its representative hypernode r). After that, all hyperedges which were removed because they were parallel hyperedges are restored. The weight of each representative of a parallel hyperedge group is reset to its original value. Then the remaining hypernodes of e are restored and their block information set to the value of the representative hypernode. This ensures that the 2-partition from coarser levels is projected to higher levels of the uncoarsening phase. The weight of the representative hypernode r is reset to its original value. When the hyperedge e was contracted, the representative hypernode was connected to additional hyperedges. These connections are now removed again. Finally, the nested hyperedges of e are restored. After this, the 2-partition is locally refined around the hyperedge e, see the next Section.

4.4.2 Refinement

Our algorithm implements a local refinement heuristic which refines a 2-partition P around a (just uncontracted) hyperedge e. Algorithm 5 shows a high-level view of the refinement heuristic. In case that the 2-partition P violates the balance constraint the refinement phase is used to rectify this. The exact method used is explained later. Unless otherwise noted, we assume in the following that P is legal. Each hypernode has two status flags: One for **active/inactive** and one for **marked/unmarked**. We need those two flags as we later have to determine if an inactive hypernode had already been in one of the priority queues. Initially the flags of all hypernodes of H are set to unmarked and inactive. Two addressable priority queues PQ_0 and PQ_1 are initialized. They will hold hypernodes and their respective gain value. PQ_0 holds hypernodes which are in block 0, PQ_1 hypernodes which are in block 1.

A priority queue is called *eligible* if is is non-empty and the hypernode with the highest gain of this queue does not violate the balance constraint if switched to the other block. We pick the hypernode v with the highest gain out of all eligible priority queues and switch its block. v is set inactive and marked and removed from its priority queue. The gain of all adjacent hypernodes is recalculated and updated in their priority queues. The block-switch of v can turn incident hyperedges of v into border hyperedges. All their incident hypernodes are inserted into the respective priority queues as long as they are inactive and unmarked.

Algorithm 5 Refinement

Require: H is a hypergraph with $H = (V, E, c, w)$, P is a 2-partition of H which does not violate the balance constraint and $e \in E$ a hyperedge

```
function REFINE(H, P, e)
    for all hypernodes v ∈ V do
        set v to inactive
        set v to unmarked
    initialize empty addressable priority queues PQ_0 and PQ_1
    for all adjacent border hyperedges be of e do
        for all adjacent inactive and unmarked hypernodes v of be do
            PQ_{P[v]}.insert(v, gain(v))
            set v active
    // perform local search
    // rollback to minimum cut
```

This process is repeated until the stopping criteria is fulfilled. We use a very simple stopping criteria. Initially a counter c is set to 0 and each iteration increases the counter by 1. When c reaches a predefined threshold i the local search is aborted. Whenever a block-switch of a hypernode leads to a hyperedge cut less or equal to the previously achieved minimum that occured in this local search c is set to 0. i is a tuning parameter (see Section 5.1.3).

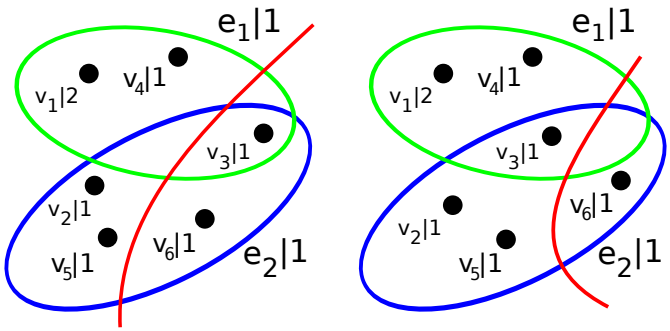
After the local search is aborted our algorithm rolls back to a random 2-partition with lowest cut. For example: The local search found two 2-partitions with hyperedge cut *hec* and none with a lower cut then one of these 2-partitions is picked by random. If the local search did not yield a cut less than the initial one all switches are undone.

Corner Cases There are two corner cases which are worth mentioning. If no eligible priority queue is found the hypernode with maximum gain out of each non-empty queue is removed and the counter c increased by one. This is done as these hypernodes violate the balance constraint and would lead to an too early abort of the local search.

Additionally the precondition that the initial 2-partition P does not violate the balancing constraint is in practice not always achievable. In this case it is possible that the local search cannot find another 2-partition Q which improves the cut and meets the balance constraint. If the initial 2-partition P violates the balance constraint our algorithm aborts the local search as soon as a block switch leads to a valid 2-partition and does not roll back to the minimum cut.

Efficient Computation of Gain Changes The move of a hypernode may cause changes of the gain of its adjacent hypernodes. We employ a mechanism similar to the one employed by Fiduccia and Mattheyses [8] which is more efficient as it avoids the recomputation of the gains. The gain of a hypernode move is the change in the global cut if the hypernode switches blocks. Obviously, only the incident hyperedges of a hypernode contribute to its gain value. The core of this method is that the contribution of each hyperedge to the gain can be computed independently.

For example in Figure 5a the gain of the move of v_3 is 1. The of e_2 is 0 as the e_2 is and stays a border hyperedge. The contribution of e_1 is $w(e_1) = 1$ as the move of v_3 would make e_1 a non-border hyperedge. All other hypernodes have a gain of 0.



(a) Nested Hyperedge (b) Connections created

Figure 5: Gain computation. The red line is the block boundary.

Now in Figure 5b v_3 was moved. This leads to v_6 having gain 1 and v_1, v_3 and v_4 having gain -1. This can be computed by looking at the incident hyperedges of the moved hypernode v_3 and adapting the gain of their respective hypernodes. e_1 had previously two hypernodes in the left block and one in the right. Now all its incident hypernodes are one block. This means that v_1 and v_2 would increase the cut if moved, thus their gain reduces by 1. The gain of v_3 does not need to be updated as it is no more in the priority queues.

For a hyperedge the following observations hold: If the number of hypernodes in one block drops to 0 the gain of the hypernodes in the other block changes by the negative weight of the hyperedge. If the number of hypernodes in one block drops to 1 the lone hypernode changes its gain by the weight of the hyperedge. Note that the moved hypernode is not eligible for moving in this pass again at it is inactive and marked after the switch and thus it is not relevant whether the gain change is computed correctly or at all for this hypernode. Similar rules apply for the other way round. Special care has to be taken if a hyperedge contains only two or three hypernodes. It is necessary for an efficient implementation that the number of hypernodes and

their block information is known and kept up to date. Our implementation saves this information in the hyperedges itself and updates it on the block switch of a hypernode.

To sum it up: A hypernode v switches blocks. The incident hyperedges of v are visited. For each hyperedge e the following procedure is executed: The gain change for all incident hypernode of e is computed and applied to each hypernode.

4.5 Hyperedge Rating Functions

The coarsening phase reduces the amount of topological information in the hypergraph. Obviously the order of contraction impacts the coarsening of H and the rating function used is responsible for this order. Thus different rating functions generate different hypergraphs at the coarsest level. As these hypergraphs are the input of the initial partitioning phase, the rating functions have an influence on how the initial 2-partition will look like and more importantly how good the cut will be. The rating function is selected by a command-line parameter (see Section B).

We define two classes of rating functions. A rating function of the first class does not differentiate whether it rates a hyperedge for the first time in a coarsening phase or later in the coarsening phase. A function of the second class does this, i.e. the first rating of an hyperedge includes some computation based on nested hyperedges. This leads to a different initial ordering in the priority queue from where the hyperedges are picked for contraction and thus a different contraction sequence.

4.5.1 Rating functions of the first class

$$rater_1(e) = \frac{w(e)}{\sqrt[|e|]{\prod_{v \in e} c(v)}} \qquad rater_2(e) = \frac{w(e)^{|e|}}{\sum_{v \in e} c(v)}$$

$$rater_3(e) = \frac{w(e)}{|e|} \qquad rater_4(e) = \frac{w(e)^{|e|}}{\prod_{v \in e} c(v)}$$

$$rater_5(e) = \frac{w(e)}{\sum_{v \in e} c(v)} \qquad rater_6(e) = w(e) \left(\sum_{v \in e} \frac{1}{c(v)} \right)$$

$$rater_7(e) = w(e) \left(\sum_{v \in e} \frac{1}{\sqrt{c(v)}} \right) \qquad rater_8(e) = w(e) \sqrt[|e|]{\left(\sum_{v \in e} \frac{1}{c(v)} \right)}$$

$$rater_9(e) = w(e) \sqrt[|e|]{\left(\sum_{v \in e} \frac{1}{\sqrt{c(v)}}\right)} \qquad rater_{10}(e) = \frac{w(e)^{|e|}}{\sqrt[|e|]{\sum_{v \in e} c(v)}}$$

$$rater_{11}(e) = \frac{w(e)}{\sqrt[|e|]{\sum_{v \in e} c(v)}} \qquad rater_{12}(e) = \frac{w(e)^{|e|}}{\sqrt[|e|]{\prod_{v \in e} c(v)}}$$

$$rater_{17}(e) = \frac{w(e)}{\prod_{v \in e} c(v)}$$

4.5.2 Rating functions of the second class

$$rater_{13}(e) = \begin{cases} \dfrac{w(e) + \sum_{n \text{ nested in } e} w(n)}{\sqrt[|e|]{\prod_{v \in e} c(v)}}, & \text{first rating in a coarsening phase} \\ \dfrac{w(e)}{\sqrt[|e|]{\prod_{v \in e} c(v)}}, & \text{otherwise} \end{cases}$$

$$rater_{14}(e) = \begin{cases} \dfrac{w(e)^{|e|} + (\sum_{n \text{ nested in } e} w(n))^{|e|}}{\sqrt[|e|]{\prod_{v \in e} c(v)}}, & \text{first rating in a coarsening phase} \\ \dfrac{w(e)^{|e|}}{\sqrt[|e|]{\prod_{v \in e} c(v)}}, & \text{otherwise} \end{cases}$$

$$rater_{15}(e) = \begin{cases} \dfrac{w(e) + \sum_{n \text{ nested in } e} 1}{\sqrt[|e|]{\prod_{v \in e} c(v)}}, & \text{first rating in a coarsening phase} \\ \dfrac{w(e)}{\sqrt[|e|]{\prod_{v \in e} c(v)}}, & \text{otherwise} \end{cases}$$

$$rater_{16}(e) = \begin{cases} \dfrac{w(e)^{|e|} + (\sum_{n \text{ nested in } e} 1)^{|e|}}{\sqrt[|e|]{\prod_{v \in e} c(v)}}, & \text{first rating in a coarsening phase} \\ \dfrac{w(e)^{|e|}}{\sqrt[|e|]{\prod_{v \in e} c(v)}}, & \text{otherwise} \end{cases}$$

$rater_{13}$ and $rater_{15}$ are derived from $rater_1$. $rater_{14}$ and $rater_{16}$ are derived from $rater_{12}$.

4.6 V-Cycles

Like *hMETIS*, we implement the "V-Cycles" technique. This means that after the first complete run we coarsen the hypergraph again, possible with another rating function. Unlike the first initial coarsening phase only hyperedges which are not border hyperedges, i.e. all its incident hypernodes are in the same block, may be contracted. The initial partitioning phase is omitted as we already have a 2-partition from the first run. Thus only the uncoarsening and refinement phase are performed. The number of these additional coarsening and refinement phases is specified by a tuning parameter v (see Section 5.1.3). Another tuning parameter q (see Section 5.1.3) determines whether only one rating function is used in the V-Cycles or the rating function is randomly picked for each cycle from a suitable set. The advantage of this method over multiple tries with different seeds is that V-Cycles allow us to improve the 2-partition P further, i. e. get closer to the local minimum which P is neighboring.

Figure 6 shows a single V-Cycle. At the coarsest level some hyperedges were not contracted as they are border hyperedges of P. In the subsequent uncoarsening and refinement phase P is refined further by applying the introduced algorithms.

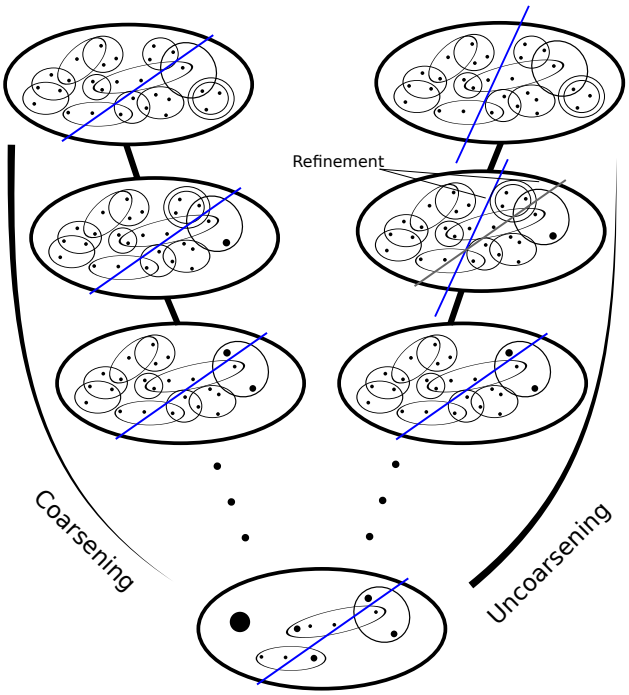


Figure 6: One V-Cycle of the n-Level Hypergraph Partitioning. Double encircled hyperedges are either contracted (left hand side) or have been uncontracted in the previous step (right hand side).

5 Experimental Evaluation

In this Section we present an experimental evaluation of our n-Level hypergraph partitioner.

First of all, we present our test set and give the data sources. Then we evaluate the rating functions. A description of the various tuning parameters follows. These parameters are then tuned extensively and we give a summary of our tuning efforts. We introduce three parameter sets *Fast*, *Strong* and *Best* which are then compared to *hMETIS* and *PaToH*. We also compare our algorithm with the results from the original n-Level paper.

5.1 Experimental Setting

5.1.1 Environment

All experiments were performed on our **Evaluation System** with 252 GiB of main memory and four AMD Opteron 6168 processors running at 1.90 GHz. Each of these processors has twelve cores which share 12 MiB of L3-Cache and each core has 512 kiB of L2-Cache. Both machines run Ubuntu 11.04 (x86_64). Our program was compiled with g++ version 4.5.2 of the GCC and optimization level 3.

5.1.2 Instances

15 graphs out of 5 benchmark sets were picked to evaluate our algorithm. The graphs are listed in Table 1. The first three hypergraphs are from Chris Walshaw's benchmark suite [27]. The next three hypergraphs are from the benchmark suite published by the MCNC[1]. These graphs stem from the circuit placement domain. The files were retrieved from the website of the "Binghampton Laboratory for Algorithms, Circuits, and Computer Aided design" [10]. The subsequent three hypergraphs are taken from the ISPD98 benchmark suite released by IBM [2]. The files were retrieved from the website of "UCSD VLSI CAD Laboratory" [16]. The next three hypergraphs are representations of sparse matrices taken from the The University of "Florida Sparse Matrix Collection" [26]. The last three hypergraphs are from the ISCAS89 benchmark suite also published by the MCNC. These graphs model digital circuits. The files were retrieved from [15].

A program (HGConv) from [11], a perl script (hmetis.perl) from [16], two perl scripts (grscgen.pl, grsc2ispd.pl) from [23] and custom scripts were used to convert the diverse file formats to the *hMETIS* and *PaToH* input formats.

[1] Microelectronics Center of North Carolina

Graph	$\|V\|$	$\|\mathcal{E}\|$	HN Deg	HE Deg	$\|\mathcal{E}\|/\|V\|$
Walshaw Suite					
cs4	22499	43858	3.90	2	1.95
bcsstk32	44609	985046	44.16	2	22.08
memplus	17758	54196	6.10	2	3.05
MCNC					
avqlarge	25178	25384	3.29	3.26	1.01
avqsmall	21918	22124	3.48	3.45	1.01
industry2	12637	13419	3.81	3.59	1.06
ISPD					
ibm03	23136	27401	4.04	3.41	1.18
ibm04	27507	31970	3.85	3.31	1.16
ibm05	29347	28446	4.30	4.44	0.97
Sparse Matrices					
crystk01	4875	4875	64.80	64.80	1
s3rmq4m1	5489	5489	51.21	51.21	1
vibrobox	12328	12328	27.81	27.81	1
ISCAS89					
s15850	10533	10383	2.35	2.38	0.99
s35932	18148	17828	2.65	2.70	0.98
s38584	21021	20717	2.63	2.66	0.99

Table 1: Test set for the experimental evaluation. HN Deg is the average number of hyperedges a hypernode is connected to. HE Deg is analogously defined.

5.1.3 Tuning Parameters

This Section explains the tuning parameters of our algorithm. These parameters effect both the quality, i.e. Hyperedge Cut, of the final 2-partition as well as the run time of our program.

Initial Partitioning Iterations z. The parameter z defines the number of runs of the initial partitioner in the initial partitioning phase. Each run is made with a different seed. This leads to multiple initial 2-partitions. The one with the smallest Hyperedge Cut is chosen as the starting point of the uncoarsening and refinement phase.

Hypernode Threshold for Contraction t. As long as there are more than t hypernodes in the hypergraph, the coarsening phase continues, i.e. new hyperedges are picked from the queue and contracted. We chose a value which is similar to the ones of *hMETIS*, *PaToH* and the original n-Level graph partitioner: $t = 100$.

Maximal Weight Of Representative Hypernode s. When the partitioner picks a hyperedge to contract in the coarsening phase, it is first checked whether the resulting representative hypernode would exceed the threshold s. s is defined as a fraction of the weight of all hypernodes in the original hypergraph. This parameter is used to prevent the forming of heavy clusters in which many light weight hypernodes are connected to a heavy hypernode by a hyperedge of size two. For tuning parameter s we adopted the same value as the original n-Level graph partitioner: $s = 0.0375$.

Maximal number of local search iterations without improvement i. After the uncontraction of a hyperedge e, refinement kicks in and tries to optimize the cut if e was near the cut. When i hypernodes were switched to another partition and no new best global cut was found, the refinement is aborted and uncoarsening continues.

Number of additional coarsening and refinement phases v. The parameter v determines the number of additional coarsening and refinement phases (V-Cycles) which are performed after the first run in order to coarsen the hypergraph in another way. This is based on the assumption that there are multiple different ways to coarsen a hypergraph which yield different possibly better cuts.

Mode of picking the rating function for a V-Cycles q. The parameter q determines whether the rating function used in a V-Cycle is the one specified by r or randomly picked from a suitable set.

5.2 Parameter Tuning

In this Section we show the results of various experiments to fine tune our algorithm by examining the influence of the tuning parameters on the final Hyperedge Cut. For each parameter we pick the value which leads to the best final Hyperedge Cut and keep this value for the rest of the parameter tuning stage. This leads us to a parameter set which we call **Best**. For the comparison of our algorithm with *hMETIS* and *PaToH* we will additionally define the parameter sets **Strong** and **Fast** (see Section 5.2.5).

5.2.1 Evaluation of Rating Functions

In Table 2 we show an evaluation of different rating functions (see Section 4.5). We ran our algorithm 25 times for each hypergraph with different seeds and computed the arithmetic mean of the results of these runs. Then we used the geometric to combine the arithmetic means to get a single value per rating function.

Rating Function	Min cut	Max cut	Avg cut	time [s]
1	429.06	**593.05**	**483.65**	13.61
2	-6.07%	-5.52%	-8.22%	19.01%
3	-11.91%	-18.97%	-19.72%	41.03%
4	**0.24%**	-2.92%	-2.63%	9.33%
5	-5.56%	-8.22%	-8.25%	12.54%
6	-48.95%	-49.61%	-54.67%	**-9.26%**
7	-50.77%	-57.32%	-58.44%	-1.85%
8	-7.09%	-25.47%	-18.04%	11.56%
9	-5.92%	-12.84%	-11.85%	17.59%
10	-35.27%	-42.47%	-44.35%	4.98%
11	-40.19%	-53.90%	-52.08%	-12.71%
12	-1.00%	-9.77%	-4.36%	5.07%
13	-1.82%	-2.60%	-1.47%	27.79%
14	-4.57%	-12.04%	-6.94%	22.72%
15	-1.87%	-1.73%	-1.16%	21.33%
16	-5.08%	-12.39%	-7.27%	23.98%
17	1.98%	1.05%	-1.29%	-5.34%

Table 2: Cut and time of different rating functions with $s = 0.0375$, $t = 100$, $max\_imbal = 2\%$, $z = 25$, $i = 10$, $v = 0$. The columns show the improvement in final cut and the increase in runtime.

For the rest of this evaluation we pick the rating function $rater_1$ as the default rating function, as it gives the best final cut on average.

5.2.2 Initial Partitioning Trials

Table 3 shows the impact of the number of initial partitioning trials. Each trial is run with a different random seed.

Obviously the improvement is small and with $z \geq 5$ the results stabilize at an improvement of the final cut of about 1% - 1.4%. Thus we choose $z = 50$ for the following experiments to err on the right side. The variable results (i. e. $z = 20$ and $z = 30$) indicate that the best initial cut must not necessarily lead to the best final cut or may be imbalanced. As the initial cut defines the topology of the partition, the subsequent refinement stage is greatly influence. For example refinement passes may abort sooner or later. This may explain the variability in runtime, as the highest variability is in the $z \geq 20$ range, where regularly new better cuts are found with a slight increase of z.

IP Rounds	Min cut	Max cut	Avg cut	time [s]
1	428.30	608.85	489.34	11.91
2	-0.37%	-0.05%	0.55%	2.06%
3	-0.23%	1.55%	0.83%	-10.22%
4	-0.36%	2.92%	0.99%	2.43%
5	-0.16%	2.40%	1.11%	6.34%
6	-0.12%	0.80%	1.24%	-4.19%
7	-0.07%	1.76%	1.19%	2.36%
8	-0.26%	3.55%	**1.41%**	8.16%
9	-0.24%	**4.05%**	1.24%	-2.47%
10	-0.39%	2.96%	1.18%	2.84%
20	-0.38%	0.66%	0.73%	12.93%
30	-0.15%	3.29%	0.96%	14.99%
40	-0.47%	3.19%	1.13%	23.39%
50	**0.01%**	2.88%	1.25%	30.36%
100	-0.53%	3.99%	1.18%	62.31%

Table 3: Initial Partitioning with $s = 0.0375$, $t = 100$, $max\_imbal = 2\%$, $i = 10$, $v = 0$, $rater_1$. The columns show the improvement in final cut and the increase in runtime.

5.2.3 Termination parameter for a refinement pass

In Table 4 we list the results of tuning parameter i, i. e. the maximal acceptable number of successive moves in one refinement pass which do not yield an improvement in cut size.

Setting with $i > 100$ lead to variable results does not result in a improved cut. The improvement stabilizes at approximately 11.3% - 11.8%. Therefore we pick $i = 100$ for our further experiments. A possible explanation for the variability in runtime may be cache effects, but we did not investigate this matter.

Parameter	Min cut	Max cut	Avg cut	time [s]
1	450.86	642.65	521.49	13.94
2	1.37%	1.98%	1.78%	5.63%
3	2.54%	3.82%	3.25%	-3.00%
4	2.73%	7.50%	4.57%	4.69%
5	3.15%	7.57%	4.77%	6.90%
6	3.88%	7.44%	5.70%	-0.42%
7	3.80%	6.97%	6.14%	9.96%
8	3.77%	7.99%	6.37%	6.43%
9	3.69%	6.09%	6.55%	7.09%
10	5.03%	10.14%	7.62%	10.82%
20	5.26%	12.02%	9.08%	18.47%
30	6.43%	11.82%	10.06%	28.57%
40	6.54%	9.57%	9.90%	32.27%
50	6.60%	10.24%	10.23%	38.53%
100	6.91%	13.45%	11.44%	61.36%
130	7.22%	12.01%	11.52%	96.66%
160	**7.28%**	11.95%	11.32%	98.31%
190	7.11%	13.62%	11.74%	110.68%
220	6.94%	**14.24%**	**11.79%**	119.35%
250	6.92%	13.15%	11.53%	115.86%

Table 4: Varying the refinement parameter with $s = 0.0375$, $t = 100$, $max\_imbal = 2\%$, $z = 50$, $v = 0$, $rater_1$. The columns show the improvement in final cut and the increase in runtime.

5.2.4 V-Cycles

In Table 5 and Table 6 we list the results of the experimental evaluation of our V-Cycle implementation. The experiments of Table 5 were employed with the setting that the chosen rating function ($rater_1$ in this case) is employed for coarsening in the additional V-Cycles, too. The experiments of Table 6 were employed with the setting that the rating function for each V-Cycle is picked randomly out of the set $rater_1$, $rater_4$, $rater_{15}$ and $rater_{17}$. We have chosen these rating functions as they yielded good cuts (see Table 2).

V-Cycles	Min cut	Max cut	Avg cut	time [s]
0	418.17	559.12	460.29	23.37
1	1.95%	2.67%	2.91%	73.56%
2	2.25%	5.08%	3.80%	145.69%
3	2.31%	7.73%	4.41%	217.32%
4	**2.56%**	8.52%	4.73%	271.69%
5	2.37%	11.65%	5.01%	332.86%
6	2.28%	11.20%	5.21%	410.56%
7	2.54%	13.08%	5.51%	491.40%
8	2.52%	11.03%	5.29%	509.91%
9	2.47%	9.90%	5.51%	628.15%
10	2.48%	**12.62%**	**5.79%**	697.23%

Table 5: $s = 0.0375$, $t = 100$, $max\_imbal = 2\%$, $z = 50$, $i = 100$, $rater_1$, q not set. The columns show the improvement in final cut and the increase in runtime.

We decided to stop the testing at 10 additional V-Cycles as the time penalty grows too big. Instead, we consider it more applicable to start the program with an appropriate configuration multiple times with different seeds and pick the best.

Interestingly using different rating functions for different V-Cycles yields a significant improvement as seen in Table 6. This is why we set q per default. Additionally we choose $v = 9$ as it gives the best final cut.

The improvement when using different rating functions is due to a larger amount of diversification in the coarsening process. That this diversification does not necessarily mean an improvement is shown by the results for $v = 9$ and $v = 10$.

V-Cycles	Min cut	Max cut	Avg cut	time [s]
0	419.35	557.13	461.56	22.53
1	3.52%	5.28%	3.96%	80.48%
2	4.13%	6.43%	4.99%	169.72%
3	4.40%	11.08%	5.95%	249.32%
4	4.37%	9.75%	6.27%	338.18%
5	4.47%	8.11%	6.45%	430.40%
6	4.53%	8.70%	6.50%	503.61%
7	4.77%	10.13%	7.04%	584.24%
8	4.67%	11.77%	7.23%	688.99%
9	4.63%	**12.79%**	**7.63%**	732.59%
10	**4.68%**	9.77%	7.32%	806.74%

Table 6: $s = 0.0375$, $t = 100$, $max\_imbal = 2\%$, $z = 50$, $i = 100$, $rater_1$, q set. The columns show the improvement in final cut and the increase in runtime.

5.2.5 Parameter Sets: Best, Strong and Fast

In this Section we define parameter sets for the final evaluation.

Fast $z = 10$, $i = 10$, $rater_{17}$, $v = 0$, (q set)

This parameter set is optimized for runtime. The most costly part, V-Cycles, is completely left out. The parameters for the refinement phase and the initial partitioning phase are kept low. We did not pick yet lower values for i and z as the time saving does not justify the loss in quality. Additionally this set uses the fastest rating function $rater_{17}$.

Strong $z = 25$, $i = 50$, $rater_1$, $v = 3$, q set

This parameter set aims to be a trade-off between runtime and quality. V-Cycles are used but only three of them. Of course q is set, thus different rating functions are used for each V-Cycle. The parameters for the refinement phase and initial partitioning phase are moderate. This parameter set employs the rating function with the best final cut, $rater_1$.

Best $z = 50$, $i = 100$, $rater_1$, $v = 9$, q set

This parameter set is optimized for quality. Nine V-Cycles are performed with q set. The parameters for the refinement phase and the initial partitioning phase are high and this set also employs the rating function with the best final cut, $rater_1$.

5.2.6 Parameter Tuning: Summary

In Table 7 we show the improvement of final cut after each tuning step.

	Avg Cut		Time
Tuning step	absolute	improvement [%]	absolute [s]
Original	483.57	–	12.25
$z = 50$	483.22	0.07	15.53
$i = 100$	461.83	4.50	22.49
$v = 10$	433.63	10.33	186.31
$v = 9, q$ set	427.77	11.54	204.29

Table 7: Comparison of the tuning steps. Percentages are relative to the original parameter choice. *Original*: $z = 10$, $i = 10$, $rater_1$, $v = 0$. Numbers taken from Table 3 to Table 6.

In Table 8 we show the improvement in cut and increase in time of the three parameter sets compared to the original parameter choice.

	Avg Cut		Time	
Preset	absolute	improvement [%]	absolute [s]	increase [%]
Original	483.57	–	12.25	–
Fast	490.10	-1.35	12.22	-0.24
Strong	438.27	9.37	65.70	436.33
Best	428.37	11.41	198.07	1,516.90

Table 8: Comparison of the parameter sets with the original parameter choice $z = 10$, $i = 10$, $rater_1$, $v = 0$. Percentages are relative to *Original*. Numbers taken from Table 3 and Table 13.

5.3 Final Evaluation

In this Section we present the final evaluation of our algorithm. First we list the used parameters for the third party partitioners. Secondly, we compare our algorithm to the original n-Level graph partitioner. At last we present a comparison between our n-Level hypergraph partitioner and *PaToH* and *hMETIS*.

5.3.1 PaToH Parameters

PaToH version 3.2 was used. The program was invoked with the following command line for the comparison and also for initial partitioning:

```
patoh ${graph} 2 FI=0.02 OD=3 PQ=Q UM=U SD=${seed}
```

Parameter description for the above command line:

`2`
: Create a 2-partition.

`FI=0.02`
: Final Imbalance of the 2-partition is maximal 2%.

`OD=3`
: Output Detail level 3 (very verbose output).

`PQ=Q`
: Use the Quality preset.

`UM=U`
: Use the Hyperedge Cut metric.

`SD=$seed`
: Use $seed as the seed.

5.3.2 hMETIS Parameters

hMETIS version 2.0pre1 was used. The program was invoked with the following command line:

```
hmetis2.0pre1 ${graph} 2 -ptype=kway -ufactor=2 -seed=${seed}
```

Parameter description for the above command line:

`2`
: Create a 2-partition.

`-ptype=kway -ufactor=2`
: Final Imbalance of the 2-partition is maximal 2%.

`-seed=$seed`
: Use $seed as the seed.

5.3.3 Comparison with the original n-Level algorithm

As we borrow heavily from the original n-Level Graph Partitioning algorithm designed by Osipov and Sanders [20] it suggests itself that we compare our hypergraph partitioner to their implementation. For that purpose we choose a subset of Walshaw's benchmark suite [27] converted the graphs into the *hMETIS* hypergraph format and run experiments with the same parameters as in the original paper as far as this is possible.

We use an identical rating function $f = \frac{w(e)}{\prod_{v \in e} c(v)} = rater_{17}$ and the somewhat relaxed parameters $s = 0.0375$, $t = 50$, $z = 25$. As the original implementation does not use V-Cycles, we do not use them here, either. Note that the original graph algorithm uses a more complex termination condition for the refinement phase. Additionally the weight of the heaviest node of an edge decides whether the edge may be contracted and not the combined weight of all incident nodes.

In Table 9 we show our results next to the results of Osipov and Sanders (**orig**). The cut is the smallest cut that was found in all runs. We made 100 runs per graph with a different seed.

Graphs				$\epsilon = 0.03$		$\epsilon = 0.05$	
Name	$\|V\|$	$\|\mathcal{E}\|$	HN Deg	orig	our	orig	our
bcsstk29	13,992	302,748	43.27	2818	2818	2818	2818
4elt	15606	45878	5.88	137	137	137	137
fesphere	16386	49152	6.00	384	384	384	384
cti	16840	48232	5.73	318	318	318	318
memplus	17758	54196	6.10	5626	**5618**	**5516**	5529
cs4	22499	43858	3.90	**366**	372	363	**362**
feocean	143437	409593	5.71	311	311	**311**	314
Geom Mean	-	-	-	606.51	607.80	604.09	604.88

Table 9: Comparison of our algorithm with the original n-Level Graph Partitioner of Osipov and Sanders. The found minimal (hyper)edge cut is shown with ϵ being the maximum imbalance. Best results are in bold face.

5.3.4 Comparison with PaToH and hMETIS

In this Section we compare out algorithm with the state-of-the-art hypergraph partitioners *PaToH* and *hMETIS*. Table 10 shows the results. A table with much more information in provided is Appendix C. Each program was run 100 times per hypergraph of the test set with different seeds. We average the results per graph and computed the geometric mean of these averages.

	Min Cut	Avg Cut		Time
Partitioner	abs.	abs.	impr. [%]	abs. [s]
hMETIS	405.08	**412.77**	4.33	8.03
Best	**399.68**	428.37	0.72	198.07
PaToH	405.67	431.46	–	**1.05**
Strong	400.26	438.27	-1.58	65.70
Fast	412.04	490.10	-13.60	12.22

Table 10: Comparison of our different parameter sets with *hMETIS* and *PaToH*. Percentages are relative to *PaToH*. Best results are are in bold face.

As we do not use a custom hypergraph data structure (see Appendix A) and did not stress performance optimization in the development of out algorithm it is not surprising that we lag far behind in terms of runtime. We are approximately one order of magnitude slower than the other two partitioners.

In terms of quality both **Best** and **Strong** achieve a better minimum cut than the two third party partitioners. **Best** yields a better average cut than *PaToH* with **Strong** being the runner-up. Nonetheless *hMETIS* computes the best average cuts by a significant margin.

6 Discussion

6.1 Conclusion

In this thesis we presented a n-Level hypergraph partitioner based on the ideas of the of Osipov and Sanders, using well known techniques like a FM refinement heuristic and V-Cycles, for example. We have shown that our implementation finds equally good partitions for graphs as the implementation of the original n-Level graph partitioner.

We have shown that our algorithm yields comparable results for benchmark instances from benchmarks suites widely used in the literature, like the ISPD or the MCNC benchmark suites.

It must be noted, that our implementation has a far worse runtime than the other two partitioners.

6.2 Future Work

As the experimental evaluation of the V-Cycles (see Section 5.2.4) has shown, the application of different rating functions has great potential. An extension of our algorithm to use different rating functions at different depths of the coarsening phase is desirable to evaluate if fundamental different topological characteristics exists at different levels and whether they can be exhibited to achieve better results.

Additionally the runtime of our implementation must be improved. We did not stress runtime as we wanted to evaluate the design. However, it is essential for larger hypergraph instances that our implementation improves its performance, for example by designing a custom hypergraph data structure.

Furthermore, the refinement phase can be enhanced, too. As long as the imbalance constraint is not violated, our implementation does not take imbalance into account when selecting the hypernodes which switch their blocks. A more sophisticated termination condition for the refinement phase is also a sensible extension.

A Hypergraph Data Structure

We build our hypergraph data structure on top of an undirected graph. The underlying implementation is the class *adjacency_list* taken from *Boost Graph Library* [1].

Hypernodes and Hyperedges are both represented by nodes in the underlying graph. An edge between a hypernode-node and a hyperedge-node means incidence. There are no edges between two hyperedge-nodes and there are no edges between two hypernode-nodes.

Two arrays hold handles to all hyperedges/hypernodes in the underlying graph. Additionally two hash tables hold the information which hyperedges/hypernodes are currently visible. If a hypernode is removed in the coarsening phase for example, it is removed from the graph and its id from the hash table. When it is readded to the hypergraph, the hash table is updated and the handle to the node in the graph is updated in the array, too.

The rest of the program solely works with the abstraction of hypernodes and hyperedges. For this purpose our Hypergraph data structure provides several helper routines.

B Command-Line Arguments

In this Section we list the possible command line arguments and their meaning.

`-h, -help`
: Print help.

`-i hypergraph.hgr`
: The input graph file has to be in the *hMETIS* format for unweighted hypergraphs [12].

`-p partition.hgr`
: Partition file which can be used to compute some metrics for the given hypergraph.hgr file. On each line there must be either a 0 or a 1 (partition info) and the number of lines needs to be identical to the number of hypernodes.

`-r repetitions of initial partitioning`
: The number of trials to initially partition the hypergraph at the coarsest level. Each of this iterations will be run with a different seed.

`-l contraction limit`
: Contraction of hyperedges is stopped when the number of remaining hypernodes is equal to or smaller than l.

`-s seed`
: Specify the seed as an integer number.

`-u`
: Disable tie-breaking .

`-e name of experiment`
: Name of the experiment which is currently performed. May be left out.

`-b maximal imbalance`
: Specify the maximal final imbalance in percent. For example: `-b 5` means 5% maximal imbalance.

`-w`
: Writes out the found 2-partition. The format is the same as at `-p`.

`-t refinement moves without improvement`
: Specify the number of refinement moves after which the refinement pass is aborted if no improvement to the global cut was achieved.

`-k rating function`
: Specify the rating function which shall be used.

`-v additional V-Cycles`
: Specify the number of additional V-Cycles.

`-f factor`
: Specify the maximum weight of a hypernode in the hypergraph as a fraction of the number of hypernodes in the original hypergraph: $factor \cdot |V|$

`-q`
: Picks a random rating function for each additional V-Cycle from the set 1, 4, 15, 17.

C Further Results

In this Chapter we present further results without comments.

Rating Function	crystk01	s3rmq4m1	s15850	vibrobox	industry2	memplus	s35932	s38584	avqsmall	cs4	ibm03	avqlarge	ibm04	ibm05	bcsstk32
1	489.12	468.12	61.12	2761.84	217.28	8047.20	49.96	60.24	176.04	389.24	1101.88	178.88	677.28	1857.64	5361.96
2	546.36	490.56	65.00	3016.20	248.24	8129.24	73.00	53.60	173.36	400.08	1204.40	178.76	698.24	1790.20	7823.40
3	790.32	481.92	77.60	2717.96	321.36	8273.28	79.72	64.56	241.76	451.88	1072.96	229.08	725.92	1884.28	6826.32
4	502.32	473.04	63.12	2665.36	242.52	7989.08	68.36	59.32	168.76	385.72	1050.40	170.80	685.52	1950.44	5573.44
5	682.56	465.60	63.44	2633.72	318.76	7918.08	73.00	60.00	169.40	395.68	1191.68	176.44	644.84	1816.12	6285.80
6	510.48	468.96	96.48	4197.56	372.60	8131.40	159.32	187.64	404.64	385.68	1507.96	336.08	1111.12	2307.88	7215.36
7	501.12	468.96	92.04	4243.92	328.88	8158.80	158.60	190.36	420.20	400.64	1572.60	491.96	1116.92	2264.20	7529.28
8	775.68	490.56	84.76	2700.92	217.84	8144.04	71.72	69.40	250.24	414.80	1179.44	231.56	767.88	1866.72	7135.72
9	786.84	488.16	67.56	2675.04	226.52	8141.28	59.12	65.64	200.88	422.52	1097.48	204.64	807.36	1839.16	7071.80
10	490.20	458.16	93.12	3910.64	331.40	8190.96	121.64	126.52	420.72	431.08	1461.68	471.52	689.92	2157.92	7177.16
11	500.40	468.48	92.64	3984.24	353.84	8098.80	158.32	175.48	392.36	411.48	1458.36	482.64	754.88	2270.60	7433.24
12	493.44	473.40	61.60	2927.48	225.84	8242.04	50.04	63.24	181.56	404.16	1108.68	181.44	755.20	1847.68	6877.96
13	497.76	425.04	61.04	3298.36	210.52	8047.20	49.96	60.96	178.36	389.24	1203.76	175.44	719.64	1855.00	5361.96
14	502.56	474.72	61.84	3478.96	219.92	8242.04	50.04	62.68	192.16	404.16	1243.56	186.28	760.00	1846.68	6877.96
15	491.04	423.60	61.16	3300.68	210.36	8047.20	49.96	59.20	177.96	389.24	1219.60	174.84	713.88	1853.48	5361.96
16	494.64	471.84	61.88	3458.92	222.56	8242.04	50.04	64.76	185.00	406.12	1263.52	189.40	778.92	1848.84	6906.44
17	502.20	418.08	61.88	2610.84	269.76	6137.92	73.00	58.00	165.08	387.84	1063.52	175.80	656.36	2506.84	4794.96

Table 11: Average Cut (arithmetic mean) of different rating functions with $s = 0.0375$, $t = 100$, $max\_imbal = 2\%$, $z = 25$, $i = 10$, $v = 0$

Rating Function	crystk01	s3rmq4m1	s15850	vibrobox	industry2	memplus	s35932	s38584	avqsmall	cs4	ibm03	avqlarge	ibm04	ibm05	bcsstk32
1	1252.48	781.84	100.00	561.96	171.60	1517.88	100.00	130.28	108.04	100.00	491.92	108.48	437.80	953.64	100.00
2	1922.36	1203.96	182.20	1319.80	413.20	1704.56	211.40	286.36	278.20	168.00	1945.28	245.12	3012.80	1760.80	191.72
3	1019.36	520.52	218.36	1069.64	718.60	2133.88	275.92	589.20	696.44	120.16	2733.16	662.20	4343.52	2518.64	116.60
4	1507.20	1016.76	153.44	756.88	389.28	1517.48	191.60	241.00	183.44	100.00	929.52	181.08	1092.84	2466.92	100.00
5	713.48	451.68	148.28	730.72	302.20	1533.88	219.84	238.16	183.92	100.00	1375.60	176.92	1565.40	1504.20	232.12
6	1100.00	969.88	194.08	2537.80	443.92	2136.92	134.56	715.96	397.08	100.00	3878.52	167.52	2867.92	3353.56	116.52
7	1059.96	1028.12	347.88	2541.40	895.32	2136.92	641.04	800.68	668.88	100.00	4078.24	427.40	3660.84	4734.84	116.68
8	1003.96	528.68	307.08	1015.84	474.28	2136.92	288.48	388.76	116.08	100.00	1737.04	117.08	1816.68	2484.44	116.92
9	1005.68	527.96	234.56	1032.04	625.08	2136.92	165.64	397.96	246.12	100.00	2368.76	208.28	3026.36	2764.60	117.76
10	1086.16	1035.60	406.24	1786.32	1319.52	2032.76	1035.96	782.76	1216.68	114.16	4785.80	688.92	5063.40	6561.48	116.28
11	965.00	768.52	319.00	1609.12	996.88	1701.36	800.92	639.64	912.40	174.24	4455.64	501.72	4501.28	5220.32	197.04
12	1510.48	1004.72	107.44	1693.80	367.60	1914.72	100.52	215.84	145.48	101.20	1642.56	139.36	2041.92	1999.68	119.64
13	101.12	97.12	100.00	556.44	171.52	1517.88	100.00	130.08	108.32	100.00	574.20	109.24	507.12	789.88	100.00
14	204.04	552.08	107.44	1919.84	347.04	1914.72	100.52	220.16	143.92	101.20	1731.40	141.12	2294.76	1838.96	119.64
15	101.12	97.12	100.00	556.44	171.52	1517.88	100.00	130.08	108.32	100.00	574.20	109.24	507.12	789.88	100.00
16	204.04	552.08	107.44	1919.84	347.04	1914.72	100.52	220.16	143.92	101.20	1731.40	141.12	2294.76	1838.96	119.64
17	197.04	106.68	148.24	561.12	382.24	149.36	190.84	245.96	188.08	100.00	870.80	186.80	911.28	1872.44	100.00

Table 12: Average contraction depth (hypernodes, arithmetic mean) of different rating functions with $s = 0.0375$, $t = 10$, $max\_imbal = 2\%$, $z = 25$, $i = 10$, $v = 0$

Graph	hMETIS			PaToH			Best			Strong			Fast		
	best	avg	time[s]	best	avg	time[s]	best	avg	time[s]	best	avg	time[s]	best	avg	time[s]
crystk01	420	420.00	5.68	420	420.60	0.91	420.00	420.00	39.12	420.00	420.18	15.92	420.00	512.55	2.33
s3rmq4m1	360	**362.76**	3.65	360	365.34	0.52	360.00	368.16	21.94	360.00	374.01	7.68	360.00	419.61	1.29
s15850	54	57.78	1.09	53	58.39	0.15	**52.00**	**55.86**	9.87	52.00	56.33	3.16	55.00	62.48	0.62
vibrobox	1990	**1990.00**	27.15	1990	2070.59	1.96	1990.00	2379.50	556.61	1990.00	2502.40	197.85	2085.00	2640.67	54.84
industry2	179	209	190.37	183	223.99	0.37	**177.00**	**198.68**	108.99	177.00	202.93	33.69	183.00	254.75	6.97
memplus	5983	6052.41	6.68	5640	6556.39	1.47	**5450.00**	**5883.37**	1603.80	5488.00	6809.45	594.97	5831.00	6146.68	33.78
s35932	43	**43.00**	2.21	43	43.80	0.34	43.00	43.61	207.33	43.00	44.76	65.27	44.00	70.76	14.48
s38584	49	49.08	4.39	51	52.25	0.36	49.00	55.28	25.64	49.00	55.25	7.75	50.00	59.50	1.64
avqsmall	142	**144.18**	3.82	143	155.55	1.52	142.00	156.17	2437.62	142.00	158.28	921.44	148.00	168.49	230.35
cs4	383	398.61	9.47	381	397.66	1.14	**364.00**	**370.23**	30.45	366.00	372.81	9.94	372.00	389.29	2.24
ibm03	959	**962.96**	16.00	973	1009.13	1.07	**958.00**	1020.72	270.97	965.00	1039.10	76.12	975.00	1054.78	12.02
avqlarge	**142**	**144.58**	4.67	144	159.40	1.68	143.00	159.64	2711.02	143.00	161.92	1077.41	153.00	173.47	235.38
ibm04	594	**601.53**	18.74	596	618.67	1.31.	**586.00**	631.72	169.10	587.00	638.90	50.31	607.00	665.44	8.56
ibm05	1724	**1730.06**	38.93	1725	1742.65	1.75	**1723.00**	1761.78	716.15	1723.00	1770.23	198.77	1769.00	2482.75	58.11
bcsstk32	4667	**4851.07**	107.47	4862	4991.40	33.19	4667.00	4941.47	979.32	4667.00	4973.69	314.65	4667.00	4874.08	48.00
Geom Mean	405.08	412.77	8.03	405.67	431.46	1.05	399.68	428.37	198.07	400.26	438.27	65.70	412.04	490.10	12.22

Table 13: Comparison of *hMETIS*, *PaToH* and our algorithm with the three parameter sets. Best results are in bold face.

Zusammenfassung

Wir stellen in dieser Arbeit einen n-Level Hypergraph Partitionierer vor, der auf den Ideen des n-Level Graph Partitionierers von Osipov und Sanders gründet. Wir berechnen für einen gegebenene Hypergraphen H eine Bisektion P, die ein vorher definiertes Imbalance-Kriterium erfüllt. Um dies zu erreichen, wird der ursprüngliche Hypergraph in mehreren Stufen (sogenannte Level) vergröbert. Auf jedem Level wird genau eine Hyperkante kontrahiert, d. h. ihre Hyperknoten werden zu einem zusammengefasst, um die Anzahl der Hyperkanten und -knoten zu verringern. Auf dem gröbsten Level berechnen wir eine Bisektion und kehren die Vergröberung wieder um. Auf jedem Level verfeinern wir die gefundenen Partition zusätzlich. Wir implementieren ebenfalls V-Zyklen (V-Cycles) um die Qualität der Partition weiter zu verfeinern.

Wir führen eine umfangreiche experimentelle Evaluation durch, die auf in der Literatur weit verbreitete Benchmark-Instanzen, beispielsweise ISPD oder MCNC, zurückgreift. Wir vergleichen unser System mit aktuellen Hypergraph Partitionierern wie *hMETIS* und *PaToH*. Unser optimiertes System erzielt Hyperedge Cuts, die im Mittel genauso gut sind, wie die von *PaToH*. Allerdings ist die Laufzeit unseres Systems wesentlich schlechter, als die der Vergleichssysteme. Darüberhinaus findet unser System den kleinsten Hyperedge Cut für ungefähr die Hälfte der Testsuite im Vergleich zu den mit Standardeinstellungen laufenden Partitionnierern *hMETIS* und *PaToH*.

References

[1] URL: http://www.boost.org/doc/libs/1_42_0/libs/graph/doc/index.html.

[2] Charles J. Alpert. "The Ispd98 Circuit Benchmark Suite". In: *Proc. ACM/IEEE International Symposium on Physical Design, April 98*. 1998, pp. 80–85.

[3] Charles J. Alpert and Andrew B. Khang. "Recent directions in netlist partitioning: a survey". In: *Integration, the VLSI Journal* 19 (1995), pp. 1–81.

[4] Charles-Edmond Bichot and Patrick Siarry. *Graph Partitioning*. Wiley-ISTE, 2011. ISBN: 978-1-84821-233-6.

[5] U. Catalyurek and C. Aykanat. "Hypergraph-Partitioning Based Decomposition for Parallel Sparse-Matrix Vector Multiplication". In: *IEEE Transaction on Parallel and Distributed Systems* 10.7 (1999), pp. 673–693.

[6] Umit V. Catalyurek and Cevdet Aykanat. "Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication". In: *IEEE Transactions on Parallel and Distributed Systems* 10 (1999), pp. 673–693. ISSN: 1045-9219.

[7] U.V. Catalyurek et al. "Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations". In: *Parallel and Distributed Processing Symposium, 2007. IEEE International*. Mar. 2007, pp. 1 –11.

[8] C.M. Fiduccia and R.M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In: *Design Automation, 1982. 19th Conference on*. June 1982, pp. 175 –181.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. ISBN: 978-0716710455.

[10] *Gigascale Systems Research Center Bookshelf Benchmarks*. URL: http://vlsicad.cs.binghamton.edu/benchgsrc.html.

[11] *HGConv tool to convert netD/areM to and from nodes/nets/wts*. URL: http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/Fundamental/HGraph/.

[12] *hMETIS A Hypergraph Partitioning Package Manual*. URL: http://glaros.dtc.umn.edu/gkhome/fetch/sw/hmetis/manual.pdf.

[13] *hMETIS website*. URL: http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview.

[14] Edmund Ihler, Dorothea Wagner, and Frank Wagner. “Modeling Hypergraphs by Graphs with the Same Mincut Properties”. In: *Information Processing Letters* 45.4 (1993), pp. 171–175.

[15] *ISCAS89 Benchmark Suite Files*. URL: `http://www.pld.ttu.ee/~maksim/benchmarks/iscas89/bench/`.

[16] *ISPD98 Benchmark Files*. URL: `http://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html`.

[17] George Karypis and Vipin Kumar. “Multilevel k-way Hypergraph Partitioning”. In: *In Proceedings of the Design and Automation Conference*. 1998, pp. 343–348.

[18] George Karypis et al. “Multilevel hypergraph partitioning: Application in VLSI domain”. In: *IEEE TRANS. VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*. 1999, pp. 69–529.

[19] B. W. Kernighan and S. Lin. “An Efficient Heuristic Procedure for Partitioning Graphs”. In: *The Bell system technical journal* 49.1 (1970), pp. 291–307.

[20] Vitaly Osipov and Peter Sanders. “n-Level Graph Partitioning”. In: *ESA 2010, 18th European Symposium, Proceedings*. 2010, pp. 278–289.

[21] *PaToH manual*. URL: `http://bmi.osu.edu/~umit/PaToH/patoh-matlab.pdf`.

[22] *PaToH website*. URL: `http://bmi.osu.edu/~umit/software.html`.

[23] *Perl Scripts For Converting .bench to .net*. URL: `http://www.ece.uic.edu/~masud/iscas2spice.htm`.

[24] Christian Schulz and Peter Sanders. “Engineering Multilevel Graph Partitioning Algorithms”. In: *ESA 2011, Lecture Notes in Computer Science*. Vol. 6942/2011. 2011, pp. 469–480.

[25] D. G. Schweikert and B. W. Kernighan. “A proper model for the partitioning of electrical circuits”. In: *Proceedings of the 9th Design Automation Workshop*. DAC ’72. 1972, pp. 57–62.

[26] *The University of Florida Sparse Matrix Collection*. URL: `http://www.cise.ufl.edu/research/sparse/matrices/index.html`.

[27] *Walshaw’s Benchmark Suite*. URL: `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`.