n-Level Hypergraph Partitioning

Vitali Henne, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
{meyerhenke,sanders,sebastian.schlag,christian.schulz}@kit.edu
vitali.henne@gmail.com

Abstract. We develop a multilevel algorithm for hypergraph partitioning that contracts the vertices one at a time and thus allows very high quality. This includes a rating function that avoids nonuniform vertex weights, an efficient “semi-dynamic” hypergraph data structure, a very fast coarsening algorithm, and two new local search algorithms. One is a k-way hypergraph adaptation of Fiduccia-Mattheyses local search and gives high quality at reasonable cost. The other is an adaptation of size-constrained label propagation to hypergraphs. Comparisons with hMetis and PaToH indicate that the new algorithm yields better quality over several benchmark sets and has a running time that is comparable to hMetis. Using label propagation local search is several times faster than hMetis and gives better quality than PaToH for a VLSI benchmark set.

Keywords: hypergraph partitioning, local search, label propagation

1 Introduction

Context. Hypergraph partitioning (HGP) is an important problem with many application areas. Two prominent areas are VLSI design and scientific computing (e.g. the acceleration of sparse matrix-vector multiplications) [20]. While the former is an example of a field where small optimizations can lead to significant savings, the latter is an example where hypergraph-based modeling better captures the objectives of the application domain [8] than graph-based approaches.

We focus on a version of the problem that partitions the vertices of a given hypergraph into k blocks of roughly equal size (in our case 1 + ε times the average block size) while optimizing an objective function. In this paper, we minimize the total cut size, i.e., the number of hyperedges that span multiple blocks.

Since the 1990s HGP has evolved into a broad research area [3,6,20]. The two most widely used general-purpose tools are PaToH [8] (originating from scientific computing) and hMetis [14,15] (originating from VLSI design). Other tools with certain distinguishing characteristics are known, in particular Mondriaan [26] (matrix partitioning), MLPart [2] (circuit partitioning), Zoltan [10] and Parkway [24] (parallel), and UMPa [25] (multi-objective). All these tools use the multilevel paradigm, which has three phases. The first of which recursively coarsens the hypergraph to obtain a hierarchy of smaller hypergraphs that reflect
the basic structure of the input. After applying an initial partitioning algorithm to the smallest hypergraph in the second phase, coarsening is undone and, at each level, a local search method is used to improve the partition induced by the coarser level.

The two most popular local search approaches are greedy algorithms [15,25] or variations of the Fiduccia-Mattheyses (FM) heuristic [11]. FM-type algorithms move vertices to other blocks in the order of improvements in the objective. Since it allows to worsen the objective temporarily, FM can escape local optima to some extent — as opposed to simple greedy methods. However, currently only partitioners based on recursive bisection use FM-based local search algorithms [2,8,10,14,26]. On the other hand, direct $k$-way hypergraph partitioners [4,15,24,25] always employ greedy methods, although generalizations of 2-way FM to $k$-way partitioning have been proposed by Sanchis [21].

When improving a $k$-way partition directly, each vertex can potentially be moved to $k - 1$ other blocks. Sanchis’s algorithm maintains these moves in $k - 1$ priority queues (PQs) for each block, resulting in $k(k - 1)$ PQs in total. Hence, previous work on multilevel HGP notes two reasons for resorting to greedy methods: (i) Working with $k(k - 1)$ PQs limits the practicality to small values of $k$ and (ii) the Sanchis algorithm has been observed to be trapped early in local minima when used without the multilevel framework [12,15].

Motivation and Contribution. To our knowledge, the reasons above have kept other partitioners from evaluating direct $k$-way local search algorithms based on FM in the multilevel context. The present paper closes this gap with the following contributions, described in detail in Section 3: (i) We present the first direct $k$-way $n$-level hypergraph partitioner. It is motivated by the success of $n$-level graph partitioning [19] that performs a very fine-grained coarsening by only contracting a single edge on each level of the multilevel hierarchy. (ii) Generalizing a greedy local search method based on size-constrained label propagation (SCLaP) [18], we provide indication that greedy algorithms may work well in some cases, but cannot escape from relatively poor local optima in others. (iii) We therefore propose a localized FM-based $k$-way local search algorithm along the lines of Sanchis [21] that is started with a single pair of vertices only.

On 164 out of 252 experiments on well established benchmark sets, our FM-based $k$-way local search computes better partitions than both hMetis and PaToH and produces partitions of equal quality in 17 out of the 88 remaining cases (see Section 4). Moreover, our algorithm is about as fast as hMetis. The speed of our algorithm is mainly due to (i) a semi-dynamic hypergraph data structure, (ii) engineering the coarsening phase with the aim of uniform coarsening, and (iii) employing various speed-up techniques [7,11,16] to accelerate the gain update step, which is the main bottleneck of most FM implementations [20].
2 Preliminaries

An undirected hypergraph $H = (V, E, c, \omega)$ is defined as a set of vertices $V$ and a set of hyperedges $E$ with vertex weights $c : V \rightarrow \mathbb{R}_{\geq 0}$ and hyperedge weights $\omega : E \rightarrow \mathbb{R}_{\geq 0}$, where each hyperedge is a subset of the vertex set $V$ (i.e., $e \subseteq V$). We use $n$ to denote the number of hypernodes and $m$ to denote the number of hyperedges. In HGP literature, hyperedges are also called nets and the vertices of a net are called pins [8]. We extend $c$ and $\omega$ to sets, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex $v$ is incident to a net $e$ if $\{v\} \subseteq e$. We use $I(v)$ to denote the set of all nets incident to a vertex $v$. The degree $d(v)$ of a vertex $v$ is the number of its incident nets: $d(v) := |I(v)|$. Two vertices are adjacent if there exists a net $e$ that contains both vertices. The set $\Gamma(v) := \{u \mid \exists e \in E : \{v, u\} \subseteq e\}$ denotes neighbors of $v$. The size $|e|$ of a net $e$ is the number of its pins. Nets of size one are called single-node nets.

A $k$-way partition of a hypergraph $H$ is a partition of its vertex set into $k$ blocks $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. We use $b[v]$ to refer to the block id of vertex $v$. We call a $k$-way partition $\Pi$ $\varepsilon$-balanced if each block $V_i \in \Pi$ satisfies a balance constraint: $\forall i \in \{1..k\} : |V_i| \leq L_{\text{max}} := (1 + \varepsilon)|V|/k$ for some parameter $\varepsilon$. We call a block $V_i$ overloaded if $|V_i| > L_{\text{max}}$ and underloaded if $|V_i| < L_{\text{max}}$.

Given a $k$-way partition $\Pi$, the number of pins of a net $e$ in block $V_i$ is defined as $\Phi(e, V_i) := |\{v \in V_i \mid e \in I(v)\}|$. If $\Phi(e, V_i) > 0$, we say that net $e$ is connected to block $V_i$. Similarly, we say that a block $V_i$ is adjacent to a vertex $v \notin V_i$ if $\exists e \in I(v) : \Phi(e, V_i) > 0$. $R(v)$ denotes the set of all blocks adjacent to $v$. For each net $e$, $\Lambda(e) := \{V_i \mid \Phi(e, V_i) > 0\}$ denotes the connectivity set of $e$. We define the connectivity of a net $e$ as the cardinality of its connectivity set: $\lambda(e) := |\Lambda(e)|$ [8]. We call a net internal if $\lambda(e) = 1$ and cut net otherwise (i.e., $\lambda(e) > 1$). Analogously a vertex that is contained in at least one cut net is called border vertex.

The $k$-way hypergraph partitioning problem is to find an $\varepsilon$-balanced $k$-way partition of a hypergraph $H$ that minimizes the total cut $\omega(E')$ where $E' := \{e \in E : \lambda(e) > 1\}$ for some $\varepsilon$. This problem is known to be NP-hard [17].

Contracting a pair of vertices $(u, v)$ means merging $v$ into $u$. The weight of $u$ becomes $c(u) := c(u) + c(v)$ and we connect $u$ to the former neighbors $\Gamma(v)$ of $v$. We refer to $u$ as the representative and $v$ as the contraction partner. This process can lead to parallel nets (i.e., $\exists e_i, e_j \in E : e_i \triangle e_j \neq \emptyset$, where $\triangle$ is the symmetric difference). In this case, we choose net $e_i$ as representative, update its weight to $\omega(e_i) = \omega(e_i) + \omega(e_j)$ and remove $e_j$ from the hypergraph. If a contraction creates single-node nets we remove them from the hypergraph, since such nets can never become part of the cut. Uncontracting a vertex $u$ undoes the contraction and restores removed parallel and single-node nets. The uncontracted vertex $v$ is put in the same block as $u$ and the weight of $u$ is set back to $c(u) := c(u) - c(v)$.
3 $n$-Level $k$-way Hypergraph Partitioning

We now present our main contributions. A high-level overview of our $n$-level hypergraph partitioning framework is provided in Algorithm 1. As other multilevel algorithms our algorithm has a coarsening, initial partitioning and an uncoarsening phase. During the coarsening phase, we successively shrink the hypergraph by contracting only a single pair of vertices at each level, until it is small enough to be initially partitioned by some other partitioning algorithm. We describe the details of our coarsening algorithm in Section 3.1 and briefly discuss initial partitioning in Section 3.2. The initial solution is transferred to the next finer level by performing a single uncontraction step. Afterwards, one of our localized local search algorithms described in Section 3.3 and Section 3.4 is used to further improve the solution quality.

**Hypergraph Data Structure.** Traditional multilevel algorithms create a new hypergraph for each level of the hierarchy. This is not feasible in the $n$-level context, since storing each level explicitly would lead to quadratic space consumption. We therefore designed a semi-dynamic hypergraph data structure that supports efficient contraction and uncontraction operations. Conceptually, we represent the hypergraph $H$ as an undirected bipartite graph $G = (W, F)$. The vertices and nets of $H$ form the vertex set $W$. For each net $e$ incident to a vertex $v$, we add an edge $(e, v)$ to the graph. The edge set $F$ is thus defined as $F := \{(e, v) \mid e \in E : \{v\} \subseteq e\}$. When contracting a vertex pair $(u, v)$, we mark $v$ as deleted. The edges $(v, w)$ incident to $v$ are treated as follows: If $G$ already contains an edge $(u, w)$, then net $w$ contained both $u$ and $v$ before the contraction. In this case, we simply delete the edge $(v, w)$ from the graph. Otherwise, net $w$ only contained $v$. We therefore have to relink the edge $(v, w)$ to $u$. Representing this graph using an adjacency array allows us to implement deletion and relink operations with very little space overhead.

After initial partitioning, we initialize the connectivity set $\Lambda(e)$ as well as the pin counts $\Phi(e, V_i)$ for each cut net $e$. These data structures are then maintained and updated during the local search phase.

**Algorithm 1: Multilevel Hypergraph Partitioning Framework**

**Input:** Hypergraph $H$, number of desired blocks $k$, balance parameter $\varepsilon$.

1. **while** $H$ is not small enough **do** // coarsening phase
   
2. \[(u, v) := \text{argmax}_{u \in V} \text{score}(u)\]  // choose vertex pair with highest rating
   \[H := \text{contract}(H, u, v)\]  // $H := H \setminus \{v\}$
   
3. $\Pi := \text{partition}(H, k, \varepsilon)$  // initial partitioning phase
   
4. **while** $H$ is not completely uncoarsened **do** // uncoarsening phase
   
5. $(H, \Pi, u, v) := \text{uncontract}(H, \Pi)$
   
6. $(H, \Pi) := \text{refine}(H, \Pi, u, v, k, \varepsilon)$
   
**Output:** $\varepsilon$-balanced $k$-way partition $\Pi = \{V_1, \ldots, V_k\}$
3.1 Coarsening

The vertex pairs \((u, v)\) to be contracted are chosen according to a rating function. The goal of the coarsening phase is to contract highly connected vertices such that the number of nets remaining in the hypergraph and their size is successively reduced [14]. Removing nets leads to simpler instances for initial partitioning, while small net sizes allow FM-based local search algorithms to identify moves that improve the solution quality. Our coarsening algorithm therefore prefers vertex pairs that have a large number of heavy nets with small size in common. This score is then inversely scaled with the product of the vertex weights \(c(v)\) and \(c(u)\) to keep the vertex weights of the coarse hypergraph reasonably uniform:

\[
r(u, v) := \frac{1}{c(v) \cdot c(u)} \sum_{e \in \{I(v) \cap I(u)\}} \frac{\omega(e)}{|e| - 1}.
\]

(1)

This scaling factor was already effective in \(n\)-level graph partitioning [19]. At the beginning of the coarsening algorithm, all vertices are rated in random order, i.e., for each vertex \(u\) we compute the ratings of all neighbors \(\Gamma(u)\) and choose the vertex \(v\) with the highest rating as contraction partner for \(u\). Ties are broken randomly. For each vertex, we insert the vertex pair with the highest score into an addressable PQ using the rating score as key. This allows us to efficiently choose the next vertex pair that should be contracted. After contraction, we remove \(v\) from the PQ. We then remove all parallel- and single-node nets in \(I(u)\). The latter are easily identified, because \(|e| = 1\). For parallel hyperedge detection we use an efficient algorithm similar to the one in [13], which is used to identify vertices with identical structure in a graph: We create a fingerprint for each net \(e : f_i := \bigoplus_{v \in e} v \oplus x\), for some seed \(x\). These fingerprints are then sorted, which brings potentially parallel nets together. A final scan over the fingerprints then identifies parallel nets: Only if two consecutive fingerprints \(f_i, f_j\) are identical, we have to check whether \(e_i \triangle e_j = \emptyset\) by comparing their pins.

Since each contraction potentially influences the rating scores of all neighbors \(\Gamma(u)\), we have to recalculate their ratings and update the priority queue accordingly. To avoid unbalanced inputs for the initial partitioning phase, vertices \(v\) with \(c(v) > c_{\text{max}} := s \cdot c(V)\) are never allowed to participate in a contraction step and are thus removed from the priority queue. The parameter \(s\) will be chosen in Section 4 and \(t\) is the maximum size of the coarsest graph, which we set to 160k. We refer to this algorithm as full.

As in the graph partitioning case, the \(n\)-level approach has the advantage that it obviates the need to employ a matching or clustering algorithm to determine the vertices to be contracted. However, this comes at the expense of continuously re-rating the neighbors \(\Gamma(u)\) adjacent to the representative. In hypergraph partitioning, this is the most expensive part of the algorithm, because after each contraction, we have to look at all pins of all incident nets \(I(v)\). The re-rating can therefore easily become the bottleneck – especially if \(H\) contains large nets. To improve the running time of the coarsening phase in these cases, we developed two variations of the full algorithm. Both variations only differ in the way the
re-rating of adjacent vertices is handled. After contracting the vertex pair \((u, v)\),
the first version only updates the rating of those neighbors, which had chosen either
the representative \(u\) or the contracted vertex \(v\) as contraction partner.
This can be done efficiently by maintaining the set \(L_w := \{u \mid (u, w) \in PQ\}\)
of all representatives that choose \(w\) as contraction partner. The re-rating step
then only reevaluates the rating function for each vertex in \(L_u \cup L_v\). All other
ratings are left untouched. We refer to this version as \textit{partial}. The second vari-
ation does not re-rate any vertices immediately after the contraction. Instead,
all adjacent vertices \(\Gamma(u)\) are marked as invalid. If the priority queue returns an
invalid vertex, we recalculate its rating and update the priority queue accord-
ingly. In case the queue returns a valid rating, we normally continue with the
coarsening process. This version is referred to as \textit{lazy}.

### 3.2 Initial Partitioning

The coarsening process is repeated until the number of remaining vertices is
below 160\(k\) or the priority queue becomes empty. The latter can happen if no
valid contraction step remains, e.g., a step that would not lead to a representative
\(u\) having weight \(c(u) > c_{\text{max}}\). The hypergraph is then small enough to be initially
partitioned by an initial partitioning algorithm. Our framework allows using
hMetis or PaToH as initial partitioner. Because hMetis produces better initial
partitions than PaToH, we use the recursive bisection variant of hMetis for initial
partitioning. In this variant of hMetis, the maximum allowed imbalance of a
partition is defined differently [14]: An imbalance value of 5, for example, allows
each block to weigh between 0.45 \cdot c(V) and 0.55 \cdot c(V) \textit{at each bisection step}. We
therefore translate our maximum allowed block weight to match this definition,
i.e., we use imbalance parameter

\[ \varepsilon' := 100 \cdot \left( \frac{1 + \varepsilon}{k} + \frac{\max_{v \in V} c(v)}{c(V)} \right)^{\frac{1}{\log_2(k) - 0.5}} - 0.5 \]

(2)

for initial partitioning with hMetis. We call the initial partitioner multiple times
with different random seeds and use the best partition as initial partition of the
coarsest graph.

### 3.3 Localized direct \(k\)-way FM Local Search

Our local search algorithm follows ideas similar to the \(k\)-way FM-algorithm
proposed by Sanchis [21] and is further inspired by the local search algorithm
used by Sanders and Osipov [19]. Sanchis uses \(k(k - 1)\) priority queues to be
able to maintain all possible moves for all vertices. We reduce the number of
priority queues to \(k - 1\) queue \(P_i\) for each block \(V_i\). In contrast to Sanchis,
we only consider to move a vertex to \textit{adjacent} blocks rather than calculating
and maintaining gains for moves to \textit{all} blocks. This simultaneously reduces the
memory requirements and restricts the search space of the algorithm to moves
that are more likely to improve the solution. Another key difference is the way
a local search pass is started: Instead of initializing the priority queues with all
vertices or all border vertices, we perform a highly localized local search starting
only with the representative and the just uncontracted vertex. The search then
gradually expands around this vertex pair by successively inserting moves for
neighboring vertices into the queues.

**Algorithm Outline.** At the beginning of a local search pass, all queues are empty
and disabled. A disabled priority queue will not be considered when searching
for the next move with the highest gain. All vertices are labeled inactive and
unmarked. Only unmarked vertices are allowed to become active. To start the
local search phase after each uncontraction, we activate the representative and
the just uncontracted vertex if they are border vertices. Otherwise, no local
search phase is started. **Activating** a vertex $v$ means that we calculate the

$$g_i(v) := \sum_{e \in I(v)} \{\omega(e) : \Phi(e, i) = |e| - 1\} - \sum_{e \in I(v)} \{\omega(e) : \lambda(e) = 1\}$$

Thus, instead of considering all $k - 1$ possible moves of a vertex $v$, we only
examine moves to those blocks that are in the union of the connectivity sets of
its incident nets: $\bigcup_{e \in I(v)} \{\Lambda(e) \setminus b[v]\}$. After insertion, all PQs corresponding to
underloaded blocks become enabled. Since a move to an overloaded block will
never be feasible, all queues corresponding to overloaded blocks are left disabled.
The algorithm then repeatedly queries only the non-empty, enabled queues to
find the move with the highest gain $g_i(v)$, breaking ties randomly. Vertex $v$ is then
moved to block $V_i$ and labeled inactive and marked. Since each vertex is allowed
to move at most once during each pass, we remove all other moves of $v$ from the
PQs. We then update all neighbors of $v$ and continue local search until either
no non-empty, enabled PQ remains or a constant number of $c$ moves neither
decreased the cut nor improved the current imbalance. The latter criterion is
necessary, because otherwise the $n$-level approach could lead to $|V|^2$ local search
steps in total. After local search is stopped, we undo all moves until we arrive at
the lowest cut state reached during the search that fulfills the balance constraint.
All vertices become unmarked and inactive and the algorithm is then repeated
until no further improvement is achieved.

**Activation and Gain Computation.** Our gain computation algorithm is detailed
in Algorithm 2. For each vertex $v$, we are only interested in gain values for
moves to adjacent blocks $R(v)$. Calculating these gains can be done efficiently
by looking at all incident nets $I(v)$ and all adjacent blocks exactly once. While
iterating over $I(v)$, we generate the set $R(v)$ of all adjacent blocks (lines 11 and
16). To calculate the gain, we distinguish three cases for each net $e$: If $\lambda(e) = 1$
and $|e| > 1$, net $e$ is internal in block $b[v]$ and will become a cut net when $v$ is
moved to another block. We maintain the sum of the weights of all internal nets
in $\omega_{int}$. If $\lambda(e) = 2$ and one block $V_i$ of the two blocks in the connectivity set
Algorithm 2: Activation

1 Function Activate(v, G)

Input: Vertex v to be activated, gain array Ω[1..k], ∀1 ≤ i ≤ k : Ω[i] = 0
2 if v is not a border vertex then return // only activate border vertices
3 \( \omega_{\text{int}} := 0 \) // weight of internal nets in I(v)
4 \( R := \{ \} \) // discovered blocks adjacent to v
5 foreach \( e \in I(v) \) do // visit incident nets
6     switch \( \lambda(e) \) do // and look at connectivity
7         case 1 : // e is internal
8             if |e| > 1 then \( \omega_{\text{int}} := \omega_{\text{int}} + \omega(e) \)
9         case 2 : // e might be removable from the cut
10             foreach \( V_i \in \Lambda(e) \) do // visit all connected blocks
11                 \( R := R \cup \{ V_i \} \)
12                 if \( \Phi(e, V_i) = |e| - 1 \) then // move removes e from the cut
13                     \( \Omega[V_i] := \Omega[V_i] + \omega(e) \)
14             else // e will not be removable from the cut
15                 foreach \( V_i \in \Lambda(e) \) do // to find moves with negative or zero gain
16                     \( R := R \cup \{ V_i \} \)
17     R := R \{ b[v] \} // remove current block of v
18     \( \Omega[b[v]] := 0 \) // and reset the gain value
19     foreach \( V_i \in R \) do // R now contains all adjacent blocks
20         \( P_i, \text{insert}(v, \Omega[V_i] - \omega_{\text{int}}) \) // \( g_i(v) = \Omega[V_i] - \omega_{\text{int}} \)
21         \( \Omega[V_i] := 0 \) // reset slot to initial state
22     if \( c(V_i) < L_{\text{max}} \) then \( P_i,\text{enable() \} // enable eligible PQs \)

Output: ∀ adjacent blocks \( V_i \) of v : \( P_i \) contains v with priority \( g_i(v) \). If block \( V_i \) is underloaded, priority queue \( P_i \) is enabled.

\( \Lambda(e) \) contains all but one pin, net e can be removed from the cut by moving v to block \( V_i \). The weight of these nets is stored in \( \Omega[V_i] \). All other nets cannot be removed from the cut by moving v to a different block. We therefore just update \( R \) accordingly. Finally, by iterating over the set of all adjacent blocks, we can compute the gain values for moving v to all connected blocks \( V_i \) by substracting the internal weight \( \omega_{\text{int}} \) from the weight stored in \( \Omega[V_i] \) (line 20).

Update of Neighbors. After moving a vertex v from block \( V_{\text{from}} \) to a different block \( V_{\text{to}} \), we have to update all of its neighbors \( \Gamma(v) \). All previously inactive neighbors are activated using Algorithm 2. All neighbors that became internal are labeled inactive and all corresponding moves are deleted from the priority queue. Finally, we update the gains for all moves of the remaining neighbors that are already active and remain border vertices. We reuse the gain values that are already calculated and only perform delta-gain-updates: If the move changed the
Algorithm 3: Delta-Gain-Update

Function DeltaGainUpdate(v, V_{from}, V_{to})

1. **Input**: Vertex $v$ that was moved from block $V_{from}$ to $V_{to}$
2. **foreach** $e \in I(v)$ do // walk all incident nets
3.    **foreach** $u \in e \setminus \{v\}$ do // and consider each pin
4.        if $\Phi(e, V_{from}) = |e| - 1$ then // move made $e$ a cut net
5.            **foreach** $V_i \in \Pi \setminus \{V_{from}\}$ do $P_i.update(u, \omega(e))$
6.        if $\Phi(e, V_{to}) = |e|$ then // move removed $e$ from the cut
7.            **foreach** $V_i \in \Pi \setminus \{V_{to}\}$ do $P_i.update(u, -\omega(e))$
8.        if $\Phi(e, V_{to}) = |e| - 1 \land b[u] \neq V_{to}$ then // only $v$ still outside $V_{to}$
9.            // moving it to $V_{to}$ would remove $e$ from the cut
10.           $P_{to}.update(u, \omega(e))$
11.    if $\Phi(e, V_{from}) = |e| - 2 \land b[u] \neq V_{from}$ then // 2 pins outside $V_{from}$
12.        // moving $v$ to $V_{from}$ could have removed $e$ from the cut
13.           $P_{from}.update(u, -\omega(e))$

**Output**: The gains for all moves of all neighbors $\Gamma(v)$ are updated.

For each net $e$, we have to consider four cases:

1. **Before the move**, net $e$ was completely internal in block $V_{from}$. Now, after the move, $e$ has become a cut net with $\Lambda(e) = 2$ and $\Lambda(e) = \{V_{from}, V_{to}\}$, if all but one pin of net $e$ are in block $V_{from}$. In this case, before the move, net $e$ contributed $-\omega(e)$ to the gain of all its pins for moving to another block. Since the move of vertex $v$ now made $e$ a cut net, all other pins can be moved to another block without incurring a further increase in cut. We therefore change the contribution of net $e$ from $-\omega(e)$ to zero by increasing the corresponding gains by $\omega(e)$ (lines 4 to 5).

2. **Vertex $v$ was the only pin** of net $e$ that was outside of block $V_{to}$ before the move and the movement therefore removed $e$ from the cut. Now that $e$ is internal, it contributes $-\omega(e)$ to the gain of all its remaining pins for moving to another part, since each move would again make it a cut net (lines 6 to 7).

3. **After the move of $v$ only one pin** of net $e$ remains outside of $V_{to}$. If that pin is also moved to $V_{to}$, we remove $e$ from the cut. The corresponding move of this pin therefore receives a delta-gain-update of $\omega(e)$. For all other pins, the contribution of $e$ to their gain values did not change (lines 8 to 10).

4. **Before the move of $v$, there was only one pin** left that was outside of $V_{from}$. Moving this pin to $V_{from}$ would have removed $e$ from the cut. However, now that $v$ is also outside of $V_{from}$, the move of this pin cannot decrease the cut...
any more. The contribution of net \( e \) to the gain of moving this pin to \( V_{\text{from}} \) therefore changes from \( \omega(e) \) to zero (lines 11 to 13).

For each active vertex, the priority queues maintain the gains to all adjacent blocks. The set of adjacent blocks, however, is subject to change during local search, because vertex movements can increase as well as decrease the connectivity of incident nets. The update process therefore has to take these changes into account. Otherwise we would either miss potential moves or perform stale moves, i.e., move a vertex to a block that was adjacent to \( v \) at some point of the local search but is not adjacent any more at the time it is returned by the priority queue. The movement of \( v \) increased the set of adjacent blocks \( R(u) \) for one of its neighbors \( u \) if \( V_{\text{to}} \not\in R(u) \) before the move. In this case, we calculate the gain \( g_{\omega}(u) \) and insert \( u \) into the priority queue \( P_{\text{to}} \). Similarly, if the movement decreased the set of adjacent blocks (i.e., \( V_{\text{from}} \not\in R(u) \) after the move), we remove the vertex from \( P_{\text{from}} \).

**Critical Nets.** Having identified the cases in which a move changes the gain contribution of one of its incident nets, we generalize the notion of critical nets introduced by Fiduccia and Mattheyses [11] for bipartitioning to \( k \)-way partitioning. A net is said to be critical if there exists at least one move for one of its pins that affects the gain contribution. Notice that the conditions concerning \( \Phi(e, \cdot) \) in lines 4, 6, 8 and 11 of Algorithm 3 can be evaluated without considering a pin of the corresponding net. Thus, we can determine whether or not a net is critical by evaluating these conditions once before iterating over all pins (line 3). Only if one of the conditions evaluates to true, we actually have to consider each pin of the net.

**Locked Nets.** Performing delta-gain-updates rather than re-calculating the gains for each neighbor of a moved vertex from scratch considerably reduces the complexity of the update step. The complexity can be reduced even further by noticing that the contribution of a net does not change any more once two of its pins have been moved to two different blocks. The net is then locked in those two blocks, because neither of the two vertices is allowed to be moved again during the current local search pass. It is therefore not possible to remove such a net from the cut by moving any of the remaining movable pins to another block. Thus it is not necessary to perform any further delta-gain-updates for locked nets. This observation was first described by Krishnamurthy [16] for bipartitioning and transferred to \( k \)-way partitioning by Sanchis [21]. We integrate locking of nets into our algorithm by labeling each net during a local search pass. Initially, all nets are labeled free. Once the first pin of a net is moved, the net becomes loose. It now has a pin in one block that cannot be moved again. Further moves to this block do not change the label of the net. As soon as another pin is moved to a different block, the net is labeled locked and is excluded from future delta-gain-updates. However, we still have to account for changes in connectivity as described above.
3.4 Local Search with Size-Constrained Label Propagation (SCLaP)

The label propagation algorithm for graph clustering was recently equipped with a size constraint to work as a coarsening and a local search algorithm for graph partitioning [18]. We briefly outline the previous local search algorithm before describing our adaptation to hypergraphs. Initially, each vertex \( v \) is in its current block \( b[v] \). The algorithm then works in rounds. In each round, the vertices are visited in random order and each vertex \( v \) is moved to the eligible (i.e. not overloaded after the move) block \( V_i \) that has the strongest connection to \( v \). Ties are broken randomly. After all vertices are visited, the process is repeated until the labels have converged or a maximum number of \( \ell \) rounds is reached.

We modify this local search algorithm as follows in order to be applicable in our \( n \)-level hypergraph partitioning context (see Algorithm 4). Instead of iterating over all vertices, we start the first iteration only with the vertex pair \((u, v)\) that has just been uncontracted. If one of these two vertices changes its block, all of its neighbors are allowed to change their block in the next iteration. This can be done efficiently by maintaining two queues \( Q_1 \) and \( Q_2 \). \( Q_1 \) contains the vertices for the current iteration and \( Q_2 \) those for the next iteration. After each round, we clear \( Q_1 \) and swap it with \( Q_2 \). In order to reflect our partitioning objective, we move a vertex to the eligible block that maximizes the gain as

\[
\text{Algorithm 4: Label Propagation Local Search}
\]

\begin{verbatim}
Input: Uncontracted vertex pair \((u, v)\)
1 \( Q_1 := \{u, v\} \quad \quad \quad \text{// set of vertices for current iteration} \)
2 \( Q_2 := \{\} \quad \quad \quad \text{// set of vertices for next iteration} \)
3 while \( Q_1 \neq \{\} \land \text{num_iterations} \leq \text{max_iterations} \) do
4     \( \text{foreach } v \in Q_1 \text{ in random order do} \)
5         \( \Omega[1..k] := \bot \)
6         \( \text{foreach } V_i \in R(v) \text{ do} \quad \text{// for all adjacent blocks} \)
7             \( \text{foreach } e \in I(v) \text{ do} \quad \text{// calculate gains for moves} \)
8                 \( \text{if } c(v) + c(V_i) \leq L_{\text{max}} \text{ then} \quad \text{// enforce balance constraint} \)
9                     \( \text{if } \Omega[V_i] = \bot \text{ then } \Omega[V_i] := 0 \)
10                    \( \Omega[V_i] := \Omega[V_i] + \text{gain}(v, e, V_i) \)
11            \( V_{\text{max}} := \text{argmax}_{1..k} \Omega[V_i] \quad \text{// choose max-gain move with max. } \Lambda\text{-decrease} \)
12            \( \text{if } b[v] \neq V_{\text{max}} \text{ then} \)
13                \( \text{move}(v, V_{\text{max}}) \)
14                \( \text{foreach } w \in \Gamma(v) \text{ do} \quad \text{// all neighbors may change their block in next round} \)
15                   \( Q_2 := Q_2 \cup \{w\} \)
16         \( Q_1 := \{\} \)
17     \( \text{swap}(Q_1, Q_2) \)
4 \text{end while} \)
Output: Refined partition \( \Pi = \{V_1, \ldots, V_k\} \)
\end{verbatim}
defined in Equation 3. For each incident net \( e \), we calculate its contribution to the gain for moving it to each adjacent block \( V_i \) (line 6 – line 10):

\[
\text{gain}(v, e, V_i) := \begin{cases} 
-\omega(e) & \text{if } \lambda(e) = 1 \land V_i \not\in \Lambda(e) \\
\omega(e) & \text{if } \lambda(e) = 2 \land \Phi(e, V_i) = |e| - 1 \\
0 & \text{else}
\end{cases}
\] (4)

Finally, we adapt the tie-breaking scheme: If multiple blocks have the same maximum gain, we choose to move the vertex to the block that leads to the highest connectivity decrease for all incident nets. A move of vertex \( v \) to block \( V_i \) decreases the connectivity of a net \( e \), if \( \Phi(e, b[v]) = 1 \) and \( \Phi(e, V_i) \neq 0 \). Analogously, a move increases the connectivity if \( \Phi(e, V_i) = 0 \). The total connectivity decrease for moving a vertex \( v \) to block \( V_i \) can therefore be computed during the gain calculation. The intention behind this tie-breaking scheme is to successively reduce the number of blocks a net is connected to. Only in case multiple blocks also have the same connectivity decrease value, we resort to random tie breaking.

3.5 Iterated Multilevel Algorithms

\( V \)-cycles are a common technique to further improve a solution [14,22,27]. The idea is to reuse an already computed partition as input for the multilevel approach. During coarsening, the quality of the solution is maintained by only contracting vertices belonging to the same block. The current partition of the coarsest graph is then used as initial partition. During uncoarsening, local search algorithms can then further improve solution quality. We also adopt this technique for \( n \)-level hypergraph partitioning by modifying the rating algorithm such that we only allow the contraction of vertex pairs that belong to the same block.

4 Experiments

\textit{Instances.} We evaluate our algorithms on hypergraphs derived from two well established benchmark sets: The ISPD98 Circuit Benchmark Suite [1] and the University of Florida Sparse Matrix Collection [9]. From the latter, we use the instances that are part of the 10th DIMACS implementation challenge dataset [5]. The matrices are translated into hypergraphs using the row-net model, i.e. each row of the matrix is treated as a net. All hypergraph have unit net and vertex weights. We exclude the two largest instances \( nlpkkt200 \) and \( nlpkkt240 \), because they could not be partitioned using hMetis. In both cases the amount of memory needed by hMetis exceeded the amount of memory available on our machine.

We divided the benchmark set into medium-sized and large instances and use \( k \in \{2, 4, 8, 16, 32, 64, 128\} \) for the number of blocks and an allowed imbalance of \( \varepsilon = 0.03 \). The properties of the hypergraphs are summarized in Table 5.

\textit{System.} All experiments are performed on a single core of a machine consisting of two Intel Xeon E5-2670 Octa-Core processors (Sandy Bridge) clocked at 2.6 GHz. The machine has 64 GB main memory, 20 MB L3-Cache and 8x256 KB L2-Cache and is running Red Hat Enterprise Linux (RHEL) 6.4.
Methodology. The algorithms are implemented in the $n$-level hypergraph partitioning framework *KaHyPar* (Karlsruhe Hypergraph Partitioning). The code is written in C++ and compiled using gcc-4.9.1 with flags `-O3 -mtune=native -march=native`. Unless otherwise mentioned, we perform ten repetitions with different seeds for each experiment and report the arithmetic mean of the computed cut and running time as well as the best cut found. When averaging over different instances, we use the geometric mean in order to give every instance a comparable influence on the final result. We compare our algorithms to both the $k$-way (hMetis-K) and the recursive-bisection variant (hMetis-R) of hMetis 2.0 (p1) [14,15] and to PaToH 3.2 [8]. As noted in Section 3.2, hMetis-R employs a different balance constraint. We therefore translate our imbalance parameter $\varepsilon = 0.03$ to $\varepsilon'$ as described in Equation 2 such that it matches our balance constraint after $\log_2(k)$ bisections. Since PaToH ignores the random seed if configured to use the quality preset, we report both the result of the quality preset (PaToH-Q) as well as the average over ten repetitions using PaToH in default configuration (PaToH-D). In both cases, we configured PaToH to use a final imbalance ratio of $\varepsilon = 0.03$ to match our balance constraint.

Algorithm Configuration. We performed a large amount of experiments to tune the parameters of our algorithms using the medium-sized instances and $k \in \{2, 4, 8, 16, 32, 64\}$. A full description of these experiments is omitted due to space constraints. We use two sets of parameter settings *fast* and *strong* that turned out to work well. Both configurations use the same parameters for coarsening and initial partitioning: We use the lazy variant as coarsening algorithm, because it is the fastest algorithm and provides comparable quality to both other variants (see Table 1). The coarsening process is stopped as soon as the number of vertices drops below $t = 160k$ or no eligible vertex is left. The scaling factor $s$ for the maximum allowed vertex weight during coarsening is set to 2.5. We use the default configuration of hMetis as initial partitioner and perform only one initial partitioning trial, since more iterations seldomly produced different cuts. The strong configuration uses the $k$-way FM algorithm described in Section 3.3 and stops each pass as soon as $c = 200$ moves did not yield any improvement. The fast configuration employs the SCLaP-based refinement algorithm presented in Section 3.4 and performs $\ell = 5$ rounds on each level. Both configurations can be augmented using V-cycles (referred to as *FastV* and *StrongV*). The maximum number of V-cycle iterations is set to 3 for FastV and to 10 for StrongV.

| Variant | avg. cut | best cut | coarsening time |
|---------|----------|----------|-----------------|
| full    | 2 990.95 | 2 910.30 | 1.00            |
| partial | 2 996.93 | 2 914.21 | 0.10            |
| lazy    | 2 988.74 | 2 916.62 | 0.06            |

Table 1. Test results for the three different variants of our coarsening algorithm on medium-sized instances. Running time of the coarsening phase is relative to *full*. 
4.1 Main Results

Evaluation of Local Search Algorithms. In the following, we report the final results of the parameter tuning on the medium-sized benchmark set and evaluate the performance of our local search algorithms described in Section 3.3 and Section 3.4 in the n-level context. The results are summarized in Table 2. Using an FM-based heuristic pays off: The localized k-way FM algorithm consistently outperforms the greedy SCLaP-based algorithm. The cuts of label propagation using V-cycles are on average 4% larger than those of StrongV. Its advantage decreases for larger values of k. This can be explained by the fact that as k increases, cut nets are more likely to connect more than two blocks. The FM algorithm then has to deal with an increasing number of zero-gain moves, which effectively weakens its ability to climb out of local minima.

V-cycles improve the solution quality of both algorithms by around 1% on average. The impact of global search iterations is larger for the SCLaP-based algorithm than for localized k-way local search, especially if the number of blocks is small. In these cases, it is more likely that a vertex can switch its label and thereby improve the solution quality. With increasing k, this becomes more difficult. The effect of V-Cycles is more stable for k-way FM algorithm, since it is already a strong heuristic.

Considering running times, we note that the SCLaP-based algorithm is an order of magnitude faster on average than the localized FM algorithm. However, best cuts found by the former are still larger than the average cuts of the latter – even for instances of medium size.

| k  | StrongV | Strong | FastV | Fast  |
|----|---------|--------|-------|-------|
|    | best    | avg    | best  | best  |
|    | t[s]    | avg[%] | t[s]  | t[s]  |
| 2  | 792     | 815    | 13.7  | +0.56 | +0.52  |
|    | +4.50   | +5.70  | 1.3   | +6.15 | +8.39  |
| 4  | 1 662   | 1 744  | 37.4  | +1.54 | +1.02  |
|    | +5.17   | +5.31  | 1.9   | +7.95 | +7.77  |
| 8  | 2 823   | 2 920  | 76.3  | +1.66 | +1.18  |
|    | +5.43   | +5.52  | 2.8   | +7.58 | +7.68  |
| 16 | 4 090   | 4 190  | 155.7 | +1.54 | +1.45  |
|    | +4.57   | +4.50  | 4.6   | +5.95 | +5.94  |
| 32 | 5 454   | 5 529  | 229.1 | +1.17 | +1.06  |
|    | +3.32   | +3.18  | 7.8   | +4.19 | +3.98  |
| 64 | 6 843   | 6 921  | 228.8 | +0.92 | +0.85  |
|    | +2.42   | +2.32  | 14.1  | +2.84 | +2.76  |
| avg| 2 877   | 2 955  | 82.7  | +1.23 | +1.01  |
|    | +4.23   | +4.41  | 3.9   | +5.76 | +6.07  |

Comparison to other Hypergraph Partitioners. We now switch to our benchmark set containing large instances to avoid the effect of overtuning our algorithms to the instances used for parameter tuning. We exclude cage15 from the following results, because hMetis took more than 18 hours to compute a single bipartition. hMetis-K is the only algorithm that often produces imbalanced partitions. Out of 1610 cases, 209 partitions are imbalanced (up to 12% imbalance). It therefore
Table 3. Comparison of our algorithms and state-of-the-art hypergraph partitioners on large benchmark instances.

| Algorithm | avg. cut | best cut | t[s] |
|-----------|----------|----------|------|
| StrongV   | 12 968   | 12 706   | 979.0|
| Strong    | 13 088   | 12 815   | 211.3|
| FastV     | 13 955   | 13 581   | 61.5 |
| Fast      | 14 269   | 13 861   | 30.7 |
| hMetis-R  | 13 155   | 12 977   | 260.7|
| hMetis-K  | 13 548   | 13 341   | 134.3|
| PaToH-Q   | 13 805   | 13 805   | 12.6 |
| PaToH-D   | 14 560   | 13 912   | 3.1  |

has slight advantages in the following comparisons because we do not disqualify imbalanced partitions. Table 3 and Table 4 summarize the results. Detailed per-instance results can be found in the Appendix in Table 6.

The strong variants of KaHyPar produce the smallest average and minimum cuts. The results of KaHyPar-FastV are comparable to the cuts produced by PaToH. On average, the cuts produced by PaToH-D, PaToH-Q, hMetis-K, hMetis-R are 12%, 7%, 5% and 2% larger than those of KaHyPar-StrongV, respectively. hMetis-R performs surprisingly well, considering the fact that we had to tighten the balancing constraint in order to ensure balanced solutions. However, out of 161 instances, KaHyPar-StrongV computed 112 partitions that were better than those produced by hMetis-R and reproduced the cuts of hMetis-R in 14 of the remaining 49 cases. Also note that KaHyPar-Strong dominates the previously best solver hMetis-R with respect to both quality and running time.

As can be seen in Table 4, the greedy label propagation algorithm outperforms PaToH on VLSI instances, while still being on average around four times faster than hMetis. On sparse matrix instances however, it is not able to escape from local minima and thus cannot improve the quality above PaToH’s level.

Looking at the more detailed comparison of KaHyPar-StrongV to the other partitioning packages in Table 4, we see that the localized $k$-way FM algorithm is only beaten by hMetis-R for $k = 2$ and $k = 4$. The improvement in solution quality increases with an increasing number of blocks. For $k = 128$ our algorithm produces 7% better cuts than hMetis-K and 3% better cuts than hMetis-R. For large values of $k$, it becomes increasingly difficult for the greedy refinement algorithm used in hMetis-K to find moves with a positive gain. The same problem holds true for our SCLaP-based algorithm, which actually performs worse than its counterpart in hMetis-K. This could be explained by the fact that our algorithm only tries to optimize around the just uncontracted vertex pair and is likely to be trapped in a local minimum, while the greedy refinement of hMetis-K in each iteration visits all vertices and moves them to an eligible block if the move has a positive gain.
Table 4. Detailed comparison of our algorithms and other partitioners on large benchmark instances. The first table summarizes the performance of the \( k \)-way local search algorithm on all large instances. The second and third table show the results for the greedy algorithm on large VLSI (second) and large sparse matrix instances (third). The average cuts are shown as increases in cut (%) relative to the values obtained by our algorithm shown in the first column.

| \( k \) | \textbf{StrongV} | \textbf{hMetis-K} \text{cut [%]} | \textbf{hMetis-R} \text{cut [%]} | \textbf{PaToH-Q} \text{cut [%]} | \textbf{PaToH-D} \text{cut [%]} |
|---|---|---|---|---|---|
| 2 | 2 | 563.6 | 141.0 | +2.02 76.0 | −1.84 82.0 | +2.94 4.0 | +8.97 1.1 |
| 4 | 5 | 471.3 | 313.9 | +3.02 89.7 | −0.07 151.4 | +8.06 7.7 | +13.60 2.0 |
| 8 | 9 | 382.7 | 684.0 | +3.19 103.6 | +1.22 211.7 | +7.08 11.3 | +14.01 2.8 |
| 16 | 14 | 1760.4 | 1100.0 | +4.56 124.7 | +2.27 267.5 | +6.98 11.3 | +13.46 3.5 |
| 32 | 21 | 980.8 | 1884.3 | +5.25 158.5 | +2.44 319.9 | +6.85 18.4 | +12.01 4.3 |
| 64 | 32 | 190.5 | 3115.7 | +6.36 208.7 | +2.75 369.6 | +6.71 21.4 | +12.50 5.0 |
| 128 | 44 | 865.1 | 4407.8 | +7.04 271.1 | +3.46 418.8 | +6.63 25.1 | +11.56 4.7 |
| avg | 12 | 967.7 | 979.0 | +4.48 134.3 | +1.45 230.7 | +6.45 12.6 | +12.28 3.1 |

| \( k \) | \textbf{FastV} | \textbf{hMetis-K} \text{cut [%]} | \textbf{hMetis-R} \text{cut [%]} | \textbf{PaToH-Q} \text{cut [%]} | \textbf{PaToH-D} \text{cut [%]} |
|---|---|---|---|---|---|
| 2 | 4 | 578.0 | 4.1 | +5.47 15.2 | −5.64 16.8 | −0.16 0.9 | +9.33 0.2 |
| 4 | 10 | 012.8 | 5.0 | +6.82 19.0 | −6.16 30.9 | +2.25 1.7 | +12.09 0.4 |
| 8 | 5 | 215.2 | 6.8 | +5.92 23.5 | −4.37 42.9 | +3.25 2.5 | +11.75 0.5 |
| 16 | 10 | 655.8 | 10.3 | +4.26 31.6 | −2.60 54.8 | +2.25 3.2 | +9.62 0.6 |
| 32 | 10 | 649.4 | 16.7 | +2.69 46.8 | −1.73 66.4 | +3.25 4.3 | +8.27 0.8 |
| 64 | 14 | 322.2 | 28.0 | +0.76 72.9 | −0.44 77.2 | +3.57 4.8 | +8.29 0.9 |
| 128 | 18 | 316.5 | 42.5 | +2.48 106.6 | +0.72 88.9 | +3.21 5.6 | +7.22 1.0 |
| avg | 6 | 673.5 | 11.6 | −3.22 36.0 | −2.92 47.6 | +2.49 2.8 | +9.50 0.6 |

| \( k \) | \textbf{FastV} | \textbf{hMetis-K} \text{cut [%]} | \textbf{hMetis-R} \text{cut [%]} | \textbf{PaToH-Q} \text{cut [%]} | \textbf{PaToH-D} \text{cut [%]} |
|---|---|---|---|---|---|
| 2 | 4 | 573.2 | 225.8 | −5.42 313.7 | −10.98 350.5 | −7.42 15.4 | −4.98 5.0 |
| 4 | 10 | 012.8 | 233.9 | −3.21 371.5 | −9.29 650.1 | −2.61 30.0 | −1.45 9.2 |
| 8 | 18 | 887.7 | 247.0 | −4.41 403.1 | −9.26 914.2 | −5.78 44.3 | −1.19 13.1 |
| 16 | 31 | 280.3 | 269.6 | −2.37 438.5 | −7.89 1143.5 | −4.26 58.3 | +0.86 16.7 |
| 32 | 49 | 073.0 | 300.6 | −1.56 484.7 | −7.39 1352.9 | −3.62 72.7 | +0.57 20.3 |
| 64 | 77 | 031.8 | 343.0 | −0.50 547.7 | −7.14 1552.8 | −3.71 83.5 | +2.28 23.8 |
| 128 | 114 | 556.2 | 398.6 | −0.83 638.0 | −5.59 1733.5 | −2.18 99.1 | +2.83 27.3 |
| avg | 27 | 440.7 | 282.8 | −2.63 449.2 | −8.23 979.9 | −4.24 49.4 | −0.19 14.5 |
5 Conclusions and Future Work

We presented the \( n \)-level direct \( k \)-way hypergraph partitioning framework KaHyPar. Using a highly localized version of \( k \)-way FM, our algorithm produces better partitions than hMetis and PaToH on 73\% of the VLSI instances and 71\% of the sparse matrix instances. Our greedy algorithm based on size-constrained label propagation gives better quality than PaToH on VLSI instances while still being several times faster than hMetis.

Motivated by the effectiveness of hMetis-R, evaluating recursive bisection in the context of \( n \)-level partitioning seems to be a promising area of future research. Having both a direct \( k \)-way and a recursive bisection \( n \)-level partitioner, KaHyPar could then be embedded into an evolutionary framework which combines both approaches to find better solutions [23].

Throughout local search, a lot of moves have gain zero. Integrating the concept of higher-level gains as described by [16,21] therefore would be a promising approach to give these moves more meaning. The running time of our \( k \)-way local search could be improved by developing an adaptive stopping rule as in [19] that is able to model zero-gain moves and stops local search if further improvement becomes unlikely. Having shown that our localized direct \( k \)-way local search algorithm is able to optimize the total cut size, future work could also look at different partitioning objectives that rely on a global view of the problem, like the \((\lambda-1)\) or sum-of-external-degrees metric [15].

References

1. C. J. Alpert. The ISPD98 Circuit Benchmark Suite. In Proc. of the 1998 Int. Symp. on Physical Design, ISPD ’98, pages 80–85, New York, 1998. ACM.
2. C. J. Alpert, J.-H. Huang, and A. B. Kahng. Multilevel Circuit Partitioning. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 17(8):655–667, 1998.
3. C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: a Survey. Integration, the VLSI Journal, 19(1-2):1 – 81, 1995.
4. C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level Direct K-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices. J. Parallel Distrib. Comput., 68(5):609–625, 2008.
5. D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In Encyclopedia of Social Network Analysis and Mining, Springer, 2014.
6. D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. Proc. Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, volume 588 of Contemporary Mathematics. AMS, 2013.
7. A. E. Caldwell, A. B. Kahng, and I. L. Markov. Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning. In ALENEX’99, volume 1619 of LNCS, pages 182–198. Springer, 1999.
8. U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans. on Parallel and Distributed Systems, 10(7):673–693, Jul 1999.
9. T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw., 38(1):1–1:25, 2011.
10. K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel Hypergraph Partitioning for Scientific Computing. IEEE, 2006.
11. C. Fiduccia and R. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In 19th ACM/IEEE Design Automation Conf., pages 175–181, 1982.
12. J. Gong and S. K. Lim. Multiway Partitioning with Pairwise Movement. In IEEE/ACM Int. Conf. on Computer-Aided Design, pages 512–516, 1998.
13. B. Hendrickson and E. Rothberg. Improving the Run Time and Quality of Nested Dissection Ordering. SIAM J. on Scientific Computing, 20(2):468–489, 1998.
14. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. IEEE Trans. on Very Large Scale Integration VLSI Systems, 7(1):69–79, 1999.
15. G. Karypis and V. Kumar. Multilevel K-way Hypergraph Partitioning. In Proc. 36th ACM/IEEE Design Automation Conference, pages 343–348. ACM, 1999.
16. B. Krishnamurthy. An Improved Min-Cut Algorithm for Partitioning VLSI Networks. IEEE Transactions on Computers, C-33(5):438–446, 1984.
17. T. Lengauer. Combinatorial Algorithms for Integrated Circuit Layout. John Wiley & Sons, Inc., 1990.
18. H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-Constrained Clustering. In Experimental Algorithms, volume 8504 of LNCS, pages 351–363. Springer, 2014.
19. V. Osipov and P. Sanders. \(n\)-Level Graph Partitioning. In Algorithms ESA 2010, volume 6346 of LNCS, pages 278–289. Springer, 2010.
20. D. A. Papa and I. L. Markov. chapter Hypergraph Partitioning and Clustering, pages 61–1–61–19. Chapman and Hall/CRC, 2007.
21. L. A. Sanchis. Multiple-way Network Partitioning. IEEE Trans. on Computers, 38(1):62–81, 1989.
22. P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In Algorithms ESA 2011, volume 6942 of LNCS, pages 469–480. Springer, 2011.
23. P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In Meeting on Algorithm Engineering & Experiments (ALENEX’12). SIAM, 2012.
24. A. Trifunovi and W. J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. J. of Parallel and Distributed Computing, 68(5):563 – 581, 2008.
25. U. V. Çatalyürek and M. Deveci and K. Kaya and B. Uçar. UMPa: A multi-objective, multi-level partitioner for communication minimization. In Bader et al. [6], pages 53–66.
26. B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. SIAM Rev., 47(1):67–95, 2005.
27. C. Walshaw. Multilevel Refinement for Combinatorial Optimisation Problems. Annals of Operations Research, 131(1-4):325–372, 2004.
## A Benchmark Instances

Table 5. Properties of our benchmark set containing medium-sized instances (top) and large instances (bottom). Tables are split into two groups: VLSI instances and sparse matrix instances. Within each group, the hypergraphs are sorted by size.

### Properties of our benchmark set containing medium-sized instances (top)

| Hypergraph | \(V\) | \(E\) | \#pins | \(d(v)\) | \(e\) |
|------------|------|------|-------|------|------|
| ibm01      | 12   | 752  | 50 566 | 1    | 3.97 |
| ibm02      | 19   | 601  | 81 199 | 1    | 4.14 |
| ibm03      | 23   | 136  | 93 573 | 1    | 4.04 |
| ibm04      | 27   | 507  | 105 859 | 1    | 3.85 |
| ibm05      | 29   | 347  | 126 308 | 1    | 4.30 |
| ibm06      | 32   | 498  | 128 182 | 1    | 3.94 |
| ibm07      | 45   | 926  | 175 639 | 1    | 3.82 |
| vibrobox   | 12   | 328  | 342 828 | 9    | 27.81 |
| bcsstk29   | 13   | 992  | 619 488 | 5    | 44.27 |
| memplus    | 17   | 758  | 126 150 | 2    | 7.10 |
| bcsstk30   | 28   | 924  | 204 492 | 4    | 70.65 |
| bcsstk31   | 35   | 588  | 181 416 | 2    | 33.20 |
| bcsstk32   | 44   | 609  | 204 701 | 2    | 45.16 |

### Properties of our benchmark set containing large instances (bottom)

| Hypergraph | \(V\) | \(E\) | \#pins | \(d(v)\) | \(e\) |
|------------|------|------|-------|------|------|
| ibm08      | 51   | 309  | 204 890 | 1    | 3.99 |
| ibm09      | 53   | 395  | 222 088 | 1    | 4.16 |
| ibm10      | 69   | 429  | 297 567 | 1    | 4.29 |
| ibm11      | 70   | 558  | 280 786 | 1    | 3.98 |
| ibm12      | 71   | 076  | 317 760 | 1    | 4.47 |
| ibm13      | 84   | 199  | 357 075 | 1    | 4.24 |
| ibm14      | 147  | 605  | 546 816 | 1    | 3.70 |
| ibm15      | 161  | 570  | 715 823 | 1    | 4.43 |
| ibm16      | 183  | 484  | 778 823 | 1    | 4.24 |
| ibm17      | 185  | 495  | 860 036 | 1    | 4.64 |
| ibm18      | 210  | 613  | 819 697 | 1    | 3.89 |
| finan512   | 74   | 752  | 596 992 | 3    | 7.99 |
| ashell10   | 504  | 855  | 17 588 875 | 20    | 34.84 |
| audikw1    | 943  | 695  | 77 651 847 | 21    | 82.28 |
| ldoor      | 952  | 203  | 46 522 475 | 28    | 48.86 |
| ecology2   | 999  | 999  | 4 995 991 | 3    | 5.00 |
| ecology1   | 1 000 | 000  | 4 996 000 | 3    | 5.00 |
| thermal2   | 1 228 | 045  | 8 580 313 | 1    | 6.99 |
| ashell10   | 1 508 | 065  | 52 672 325 | 15    | 34.84 |
| G3circuit  | 1 355 | 478  | 7 660 826 | 2    | 4.83 |
| kkt_power  | 2 063 | 494  | 14 612 663 | 1    | 7.08 |
| nlpkkt120  | 3 542 | 400  | 96 845 792 | 5 27.34 |
| cage15     | 5 154 | 859  | 99 199 551 | 3    | 19.24 |
| nlpkkt160  | 8 345 | 600  | 229 518 112 | 5 27.50 |

### Hypergraph Partitioning

\(n\)-Level Hypergraph Partitioning 19
| N  | Best Avg. t[s] | Best Avg. t[s] | Best Avg. t[s] | Best Avg. t[s] | Best Avg. t[s] | Best Avg. t[s] | Best Avg. t[s] | Best Avg. t[s] | Best Avg. t[s] |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 203 | 243.3 | 203 | 243.4 | 203 | 243.4 | 203 | 250.6 | 203 | 260.3 |
| 2 | 568 | 174.8 | 568 | 174.8 | 568 | 174.8 | 568 | 174.8 | 568 | 174.8 |
| 3 | 864 | 882.9 | 864 | 882.9 | 864 | 882.9 | 864 | 882.9 | 864 | 882.9 |
| 4 | 1571 | 1687.4 | 1571 | 1687.4 | 1571 | 1687.4 | 1571 | 1687.4 | 1571 | 1687.4 |
| 5 | 2220 | 2356.4 | 2220 | 2356.4 | 2220 | 2356.4 | 2220 | 2356.4 | 2220 | 2356.4 |
| 6 | 2570 | 2797.5 | 2570 | 2797.5 | 2570 | 2797.5 | 2570 | 2797.5 | 2570 | 2797.5 |
| 7 | 356 | 367.3 | 356 | 367.3 | 356 | 367.3 | 356 | 367.3 | 356 | 367.3 |
| 8 | 707 | 721.9 | 707 | 721.9 | 707 | 721.9 | 707 | 721.9 | 707 | 721.9 |
| 9 | 2016 | 2056.0 | 2016 | 2056.0 | 2016 | 2056.0 | 2016 | 2056.0 | 2016 | 2056.0 |
| 10 | 3306 | 3296.5 | 3306 | 3296.5 | 3306 | 3296.5 | 3306 | 3296.5 | 3306 | 3296.5 |
| 11 | 4716 | 4720.3 | 4716 | 4720.3 | 4716 | 4720.3 | 4716 | 4720.3 | 4716 | 4720.3 |
| 12 | 761 | 761.7 | 761 | 761.7 | 761 | 761.7 | 761 | 761.7 | 761 | 761.7 |
| 13 | 942 | 953.2 | 942 | 953.2 | 942 | 953.2 | 942 | 953.2 | 942 | 953.2 |
| 14 | 3137 | 3288.6 | 3137 | 3288.6 | 3137 | 3288.6 | 3137 | 3288.6 | 3137 | 3288.6 |
| 15 | 10419 | 10619.7 | 10419 | 10619.7 | 10419 | 10619.7 | 10419 | 10619.7 | 10419 | 10619.7 |
| 16 | 1786 | 1774.9 | 1786 | 1774.9 | 1786 | 1774.9 | 1786 | 1774.9 | 1786 | 1774.9 |
| 17 | 6590 | 6587.8 | 6590 | 6587.8 | 6590 | 6587.8 | 6590 | 6587.8 | 6590 | 6587.8 |
| 18 | 1276 | 1279.5 | 1276 | 1279.5 | 1276 | 1279.5 | 1276 | 1279.5 | 1276 | 1279.5 |
| 19 | 2306 | 2322.7 | 2306 | 2322.7 | 2306 | 2322.7 | 2306 | 2322.7 | 2306 | 2322.7 |
| 20 | 603 | 608.7 | 603 | 608.7 | 603 | 608.7 | 603 | 608.7 | 603 | 608.7 |
| 21 | 3210 | 3253.9 | 3210 | 3253.9 | 3210 | 3253.9 | 3210 | 3253.9 | 3210 | 3253.9 |
| 22 | 5124 | 5171.5 | 5124 | 5171.5 | 5124 | 5171.5 | 5124 | 5171.5 | 5124 | 5171.5 |
| 23 | 9350 | 9350.5 | 9350 | 9350.5 | 9350 | 9350.5 | 9350 | 9350.5 | 9350 | 9350.5 |
| 24 | 1151 | 1168.5 | 1151 | 1168.5 | 1151 | 1168.5 | 1151 | 1168.5 | 1151 | 1168.5 |
| 25 | 1696 | 1724.2 | 1696 | 1724.2 | 1696 | 1724.2 | 1696 | 1724.2 | 1696 | 1724.2 |
| 26 | 2709 | 2792.7 | 2709 | 2792.7 | 2709 | 2792.7 | 2709 | 2792.7 | 2709 | 2792.7 |
| 27 | 3839 | 3948.3 | 3839 | 3948.3 | 3839 | 3948.3 | 3839 | 3948.3 | 3839 | 3948.3 |
| 28 | 4931 | 4908.4 | 4931 | 4908.4 | 4931 | 4908.4 | 4931 | 4908.4 | 4931 | 4908.4 |
| 29 | 7390 | 7392.5 | 7390 | 7392.5 | 7390 | 7392.5 | 7390 | 7392.5 | 7390 | 7392.5 |
| 30 | 9847 | 9901.0 | 9847 | 9901.0 | 9847 | 9901.0 | 9847 | 9901.0 | 9847 | 9901.0 |
| 31 | 2239 | 2319.4 | 2239 | 2319.4 | 2239 | 2319.4 | 2239 | 2319.4 | 2239 | 2319.4 |
| 32 | 3419 | 3451.0 | 3419 | 3451.0 | 3419 | 3451.0 | 3419 | 3451.0 | 3419 | 3451.0 |
| 33 | 5447 | 5489.7 | 5447 | 5489.7 | 5447 | 5489.7 | 5447 | 5489.7 | 5447 | 5489.7 |
| 34 | 8039 | 8051.4 | 8039 | 8051.4 | 8039 | 8051.4 | 8039 | 8051.4 | 8039 | 8051.4 |
| 35 | 11869 | 11993.8 | 11869 | 11993.8 | 11869 | 11993.8 | 11869 | 11993.8 | 11869 | 11993.8 |

**Detailed results per instance. The best cut is highlighted.**
| Method | Best | Average | Best | Average |
|--------|------|---------|------|---------|
| KaHyPar-StrongV | 8 377.0 | 8 406.0 | 8 820.5 | 0.61 |
| KaHyPar-Fast | 2 001.0 | 2 001.0 | 1.78 |
| ibm15 | 22 439.3 | 22 820.0 | 1 241.50 |
| ibm16 | 5 287.0 | 5 608.0 | 6.71 |
| ibm18 | 65 227.0 | 65 704.0 | 421.42 |
| ibm20 | 24 477.0 | 26 056.0 | 1 179.00 |
| ibm21 | 11 006.0 | 11 286.0 | 13.40 |
| ibm22 | 22 222.0 | 22 439.3 | 410.37 |
| ibm23 | 2 045.0 | 2 057.9 | 2.21 |
| ibm24 | 23 180.0 | 23 387.7 | 47.28 |
| ibm25 | 36 713.0 | 38 199.1 | 1 050.94 |
| ibm26 | 5 057.0 | 5 064.1 | 7.93 |
| ibm27 | 11 530.4 | 1 048.91 |
| ibm28 | 14 793.0 | 15 035.7 | 2.42 |
| ibm29 | 8 309.0 | 8 350.6 | 12.61 |
| ibm30 | 16 124.0 | 16 409.2 | 19.69 |
| ibm31 | 14 793.0 | 15 035.7 | 2.42 |
| ibm32 | 14 793.0 | 15 035.7 | 2.42 |
| ibm33 | 14 793.0 | 15 035.7 | 2.42 |
| ibm34 | 14 793.0 | 15 035.7 | 2.42 |
| ibm35 | 14 793.0 | 15 035.7 | 2.42 |
| ibm36 | 14 793.0 | 15 035.7 | 2.42 |
| ibm37 | 14 793.0 | 15 035.7 | 2.42 |
| ibm38 | 14 793.0 | 15 035.7 | 2.42 |
| ibm39 | 14 793.0 | 15 035.7 | 2.42 |
| ibm40 | 14 793.0 | 15 035.7 | 2.42 |
| ibm41 | 14 793.0 | 15 035.7 | 2.42 |
| ibm42 | 14 793.0 | 15 035.7 | 2.42 |
| ibm43 | 14 793.0 | 15 035.7 | 2.42 |
| ibm44 | 14 793.0 | 15 035.7 | 2.42 |
| ibm45 | 14 793.0 | 15 035.7 | 2.42 |
| ibm46 | 14 793.0 | 15 035.7 | 2.42 |
| ibm47 | 14 793.0 | 15 035.7 | 2.42 |
| ibm48 | 14 793.0 | 15 035.7 | 2.42 |
| ibm49 | 14 793.0 | 15 035.7 | 2.42 |
| ibm50 | 14 793.0 | 15 035.7 | 2.42 |
| ibm51 | 14 793.0 | 15 035.7 | 2.42 |
| ibm52 | 14 793.0 | 15 035.7 | 2.42 |
| ibm53 | 14 793.0 | 15 035.7 | 2.42 |
| ibm54 | 14 793.0 | 15 035.7 | 2.42 |
| ibm55 | 14 793.0 | 15 035.7 | 2.42 |
| ibm56 | 14 793.0 | 15 035.7 | 2.42 |
| ibm57 | 14 793.0 | 15 035.7 | 2.42 |
| ibm58 | 14 793.0 | 15 035.7 | 2.42 |
| ibm59 | 14 793.0 | 15 035.7 | 2.42 |
| ibm60 | 14 793.0 | 15 035.7 | 2.42 |
| ibm61 | 14 793.0 | 15 035.7 | 2.42 |
| ibm62 | 14 793.0 | 15 035.7 | 2.42 |
| ibm63 | 14 793.0 | 15 035.7 | 2.42 |
| ibm64 | 14 793.0 | 15 035.7 | 2.42 |
| ibm65 | 14 793.0 | 15 035.7 | 2.42 |
| ibm66 | 14 793.0 | 15 035.7 | 2.42 |
| ibm67 | 14 793.0 | 15 035.7 | 2.42 |
| ibm68 | 14 793.0 | 15 035.7 | 2.42 |
| ibm69 | 14 793.0 | 15 035.7 | 2.42 |
| ibm70 | 14 793.0 | 15 035.7 | 2.42 |
| ibm71 | 14 793.0 | 15 035.7 | 2.42 |
| ibm72 | 14 793.0 | 15 035.7 | 2.42 |
| ibm73 | 14 793.0 | 15 035.7 | 2.42 |
| ibm74 | 14 793.0 | 15 035.7 | 2.42 |
| ibm75 | 14 793.0 | 15 035.7 | 2.42 |
| ibm76 | 14 793.0 | 15 035.7 | 2.42 |
| ibm77 | 14 793.0 | 15 035.7 | 2.42 |
| ibm78 | 14 793.0 | 15 035.7 | 2.42 |
| ibm79 | 14 793.0 | 15 035.7 | 2.42 |
| ibm80 | 14 793.0 | 15 035.7 | 2.42 |
| ibm81 | 14 793.0 | 15 035.7 | 2.42 |
| ibm82 | 14 793.0 | 15 035.7 | 2.42 |
| ibm83 | 14 793.0 | 15 035.7 | 2.42 |
| ibm84 | 14 793.0 | 15 035.7 | 2.42 |
| ibm85 | 14 793.0 | 15 035.7 | 2.42 |
| ibm86 | 14 793.0 | 15 035.7 | 2.42 |
| ibm87 | 14 793.0 | 15 035.7 | 2.42 |
| ibm88 | 14 793.0 | 15 035.7 | 2.42 |
| ibm89 | 14 793.0 | 15 035.7 | 2.42 |
| ibm90 | 14 793.0 | 15 035.7 | 2.42 |
| ibm91 | 14 793.0 | 15 035.7 | 2.42 |
| ibm92 | 14 793.0 | 15 035.7 | 2.42 |
| ibm93 | 14 793.0 | 15 035.7 | 2.42 |
| ibm94 | 14 793.0 | 15 035.7 | 2.42 |
| ibm95 | 14 793.0 | 15 035.7 | 2.42 |
| ibm96 | 14 793.0 | 15 035.7 | 2.42 |
| ibm97 | 14 793.0 | 15 035.7 | 2.42 |
| ibm98 | 14 793.0 | 15 035.7 | 2.42 |
| ibm99 | 14 793.0 | 15 035.7 | 2.42 |
| ibm100 | 14 793.0 | 15 035.7 | 2.42 |
| Algorithm | Strong | Fast | Vibrobox | Finan512 |
|-----------|--------|------|----------|---------|
| KaHyPar-StrongV | 1975.0 | 1461.4 | 718.4 | 2274.0 |
| KaHyPar-FastV | 1984.0 | 1461.4 | 718.4 | 2274.0 |
| PaToH-D | 1975.0 | 1461.4 | 718.4 | 2274.0 |
| PaToH-Q | 1975.0 | 1461.4 | 718.4 | 2274.0 |

**Table values:**
- **Strong**
- **Fast**
- **Vibrobox**
- **Finan512**

**Columns:**
- **Algorithm**
- **Strong**
- **Fast**
- **Vibrobox**
- **Finan512**