Karlsruhe Institute of Technology

Master thesis

Acyclic n-Level Hypergraph Partitioning

Daniel Seemaier

Date: March 2, 2020

Supervisors: Prof. Dr. Peter Sanders
Dr. Sebastian Schlag
Priv. Doz. Dr. Christian Schulz

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

Abstract

Directed acyclic graphs are widely used to model the data flow and execution dependencies of streaming applications. Efficient parallelization of such graphs requires acyclic partitioning of the dependency graph. However, normal graphs are not always capable of precisely modelling such dependencies. Thus, we consider *directed acyclic hypergraphs* (DAHs). In this work, we present the first n-level hypergraph partitioning algorithm for directed acyclic hypergraphs. Moreover, we show (i) that our algorithm beats the current state of the art for directed acyclic graph partitioning in terms of solution quality on realistic instances, (ii) that n-level algorithms are superior to single level algorithms, and (iii) that our algorithm improves on the makespan of a parallelized image streaming application.

Zusammenfassung

Gerichtete azyklische Graphen werden häufig zur Modellierung von Datenflüssen und Ausführungsabhängigkeiten von Datenflussapplikationen genutzt. Eine effiziente automatische Parallelisierung solcher Anwendung erfordert eine azyklische Partitionierung der Abhängigkeitsgraphen. Allerdings ist eine präzise Modellierung der Abhängigkeiten mit herkömmlichen Graphen nicht immer möglich. Deswegen betrachten wir in dieser Arbeit gerichtete azyklische Hypergraphen (DAHs) und präsentieren den ersten n-Stufen Algorithmus zur azyklischen Partitionierung solcher Hypergraphen. Unsere Ergebnisse bestehen aus drei Beiträgen: wir zeigen, dass (i) unser Algorithmus häufig bessere Partitionen als der aktuell beste Algorithmus auf praxisnahen gerichteten azyklischen Graphen findet, (ii) unser n-Stufen Algorithmus auf gerichteten azyklischen Hypergraphen besser abschneidet als ein einstufiger Algorithmus und (iii) unser Algorithmus eine effizientere Parallelisierung für eine echte Datenflussapplikation ermöglicht.

Acknowledgments

I would like to thank my supervisors Dr. Sebastian Schlag and Priv. Doz. Christian Schulz, who have introduced me to research in Algorithmics and mentored me since my bachelor thesis. In many meetings over the last three years, they gave me more guidance than I could have ever asked for and supported me not only tremendously with this master thesis, but also made my time while pursuing my master's degree much more exciting. I also want thank Merten Popp, who found time during his busy workweeks to test our algorithm with a real world application, thereby strengthening the result of this research. Lastly, I thank the Steinbruch Centre for Computing for granting me access to the HPC cluster bwUniCluster. Without sufficient computational power, the experimental evaluation of this thesis would not have been possible.

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

Karlsruhe, den 2. März 2020

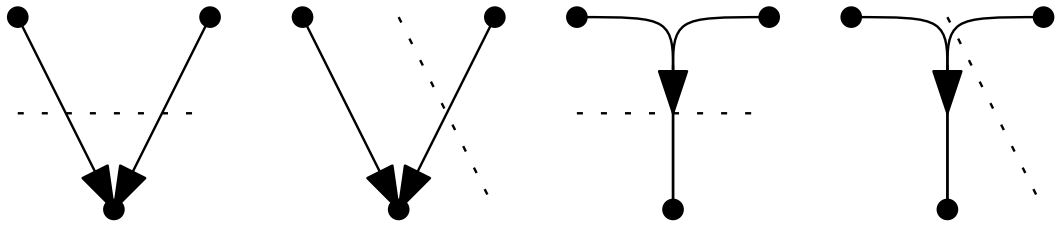
Contents

1 Introduction

Directed acyclic hypergraphs (DAHs) are a generalized concept of directed acyclic graphs (DAGs) where each hyperedge can contain an arbitrary number of tails and heads. The acyclic hypergraph partitioning problem is to partition the hypernodes of a DAH into a fixed number of blocks of roughly equal size such that the corresponding quotient graph is acyclic while minimizing an objective function on the partition.

The problem is motivated by a recent paper on graph partitioning with acyclicity constraint by Moreira et al. [42] who use directed acyclic graphs to model the data flow and execution dependencies of image streaming applications. The imaging applications are executed on embedded processors with limited thermal budget and memory. To cope with these constraints, the application is distributed over multiple processors that process the data one after another. Data dependencies between parts of the application are modeled as a directed dependency graph. To distribute the application, the dependency graph is partitioned into the same number of blocks as there are processors available. Since edges between blocks correspond to interprocessor communication, the goal is to find a partition that minimizes the weighted edge cut of the partition. However, this is merely an imprecise approximation of the real objective function, which is to minimize the number of blocks containing neighbors of a node: say that one could choose between two partitions. One partition places all successors of a node into one other block, whereas the other partition splits them over two other blocks. When using a directed graph to model the data dependencies and the weighted edge cut objective function, both partitions are seen as equal, although the first option requires less interprocessor communication since the image has to be transferred to only one other processor, i.e., only once. This problem is illustrated in Figure 1.1: while the partitions shown in Figure 1.1a and 1.1b have different edge cuts, they behave the same in practice. Directed acyclic hypergraphs allow for a better model, since a single hyperedge can contain an arbitrary number of hypernodes. The *connectivity metric* then counts the number of blocks connected by a hyperedge. Using this model, both partitions are rated the same, as shown in Figure 1.1c and 1.1d.

This application leads to a lot of preexisting work on the directed acyclic graph partitioning problem, but to the best of our knowledge, none concern the directed acyclic hypergraph partitioning problem.



(a) Edge cut: 2 **(b)** Edge cut: 1 **(c)** Connectivity: 1 **(d)** Connectivity: 1

Figure 1.1: Model of an application divided into three subapplications (nodes) executed on two PEs (dashed line). Two partitions are proposed, one shown in Figures 1.1a and 1.1c, the other one shown in Figures 1.1b and 1.1d. Since in both cases, the result of one subapplication has to be sent to one other PE, both partitions perform equally good in practice and should therefore have the same cost. Figures 1.1a and 1.1b model the application using a DAG using the edge cut metric, which rates the partitions differently. In Figure 1.1c and 1.1d, the problem is modeled as a DAH using the connectivity metric, which puts the same cost to both partitions.

1.1 Problem Statement

To the best of our knowledge, there are currently no partitioning algorithms for directed acyclic hypergraphs in existing literature and therefore, it is not possible to use the DAH model in the described application domain. We close this gap by presenting the first algorithm for acyclic HGP in this thesis. Our algorithm can only handle hypergraphs where each hyperedge contains at most one head. This is sufficient to model the data flow and execution dependencies of an image streaming application, and also for a range of other applications [6, 7].

1.2 Contribution

We have three main contributions: first and foremost, and to the best of our knowledge, we present the first algorithm for acyclic hypergraph partitioning by adapting techniques for DAG partitioning recently introduced by Herrmann et al. [25, 26] and Moreira et al. [42, 43]. We compare out algorithm on DAG instances with preexisting DAG partitioning algorithms to show an improvement of 10% on average. Second, we evaluate our algorithm on DAH instances. Since there are no preexisting algorithms that we could include in our benchmark set, we compare our algorithm with simpler heuristics and show that our n-level algorithm produces partitions with 64% lower connectivity on average than a

simple single-level k-way search with topological ordering for initial partitioning. Finally, we evaluate the impact of our improved DAH model on an image streaming application described in Ref. [42]. We show that our approach not only yields an improved model of the transfer costs, but also improves the makespan of such applications by up to 22% over the DAG model.

1.3 Structure of Thesis

The remaining content of this thesis is structured as follows. In Chapter 2, we introduce fundamental definitions from graph theory and give a precise formulation of the acyclic hypergraph partitioning problem. Chapter 3 follows with a broad overview on the current state-of-the-art graph and hypergraph partitioning techniques as well as recent approaches for solving the acyclic graph partitioning problem. This is the work that we build upon when developing our acyclic hypergraph partitioner. The main content of this thesis is described in Chapter 4 and Chapter 5, where we present our approach to the acyclic hypergraph partitioning problem and perform an extensive experimental evaluation and comparison to previous work. Finally, we conclude our work in Chapter 6.

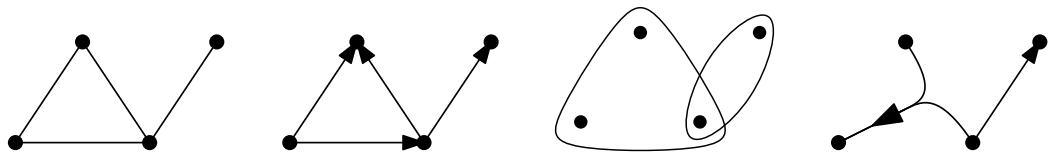
2 Fundamentals

This chapter introduces general definitions that are used throughout this thesis. We start by defining undirected and directed graphs before generalizing them to undirected and directed hypergraphs. After defining those fundamental concepts, we give a precise formulation of the directed acyclic graph and hypergraph partitioning problems. Whenever possible, we use the notation introduced in Ref. [2, 4, 55].

2.1 General Definitions

An *undirected weighted graph* $G = (V, E, c, \omega)$ consists of finite sets V and E and weight functions c and ω. When talking about multiple graphs, we also use $V(G)$ and $E(G)$ to denote the node and edge set of a particular graph. The elements of V are called *nodes* and the elements of E are called *edges*. We define $n \coloneqq |V|$ and $m \coloneqq |E|$. While the elements of V are arbitrary and of no further interest, E may only contain two-subsets of V. In other words, all edges $e \in E$ have the form $e = \{u, v\}$ with $u, v \in V$ and $u \neq v$. Figure 3.3a illustrates a simple undirected graph. We say that two nodes u and v are *adjacent* or *connected* if $\{u, v\} \in E$ and two edges e_1 and e_2 are *adjacent* if $e_1 \cap e_2 \neq \emptyset$. A node u and an edge e are *incident* if $u \in e$. The *neighborhood* $\Gamma(u)$ of a node u is the set of nodes adjacent to it. Its size $d(u) \coloneqq |\Gamma(u)|$ is the *degree* of u. The *maximum degree* $\Delta(G) \coloneqq \max_{u \in V} d(u)$ is the highest degree occurring in G. The *node weight function* $c : V \to \mathbb{R}_{>0}$ assigns non-negative weights to the nodes of G. Analogously, the *edge weight function* $\omega : E \to \mathbb{R}_{>0}$ assigns non-negative weights to the edges of G. The weight functions are extended to sets of nodes and edges by summing over the elements of the set, i.e., $c(V') = \sum_{v \in V'} c(v)$ for $V' \subseteq V$ and $\omega(E') = \sum_{e \in E'} \omega(e)$ for $E' \subseteq E$. If every node or every edge of a graph has the same weight, we say that the graph has *unit node weights* or *unit edge weights* and assume that $c(V) = n$ or $\omega(E) = m$, respectively. A *matching* $M \subseteq E$ in a graph is a set of non-incident edges. *Maximal matchings* are a particular type of matchings with the property that there exists no edge $e \in E$ such that $M \cup \{e\}$ is also a matching.

The concept of *directed graphs* puts an order on the elements of each edge. More precisely, in a directed graph each edge is a pair (u, v) of some nodes $u, v \in V$ with u being the *tail* of the edge and v being its *head*. In this case, e is *directed* from u



(a) Undirected graph. **(b)** Directed graph. **(c)** Undirected hypergraph. **(d)** Directed hypergraph.

Figure 2.1: **(a)** An undirected graph where dots represent nodes and lines represent edges. **(b)** A directed graph. The Arrow of each edge points towards its head. **(c)** An undirected hypergraph. Hyperedges are drawn using closed polygons. **(d)** A directed acyclic hypergraph.

to v. In a directed graph, each node has predecessors $\Gamma^-(u) \coloneqq \{v \mid (v,u) \in E\}$ and successors $\Gamma^+(u) \coloneqq \{v \mid (u,v) \in E\}$. The degree $d(u) \coloneqq d^+(u) + d^-(u)$ of u is the sum of its *out degree* $d^+(u) \coloneqq |\Gamma^+(u)|$ and its *in degree* $d^-(u) \coloneqq |\Gamma^-(u)|$. A *directed cycle* $C = (v_1, \ldots, v_k, v_{k+1} = v_1)$ in a directed graph is a sequence of nodes such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k$ and $v_i \neq v_j$ for $1 \leq i, j \leq k$, $i \neq j$. A directed graph that does not contain any cycles is said to be *acyclic* and referred to as a *directed acyclic graph* or *DAG* for short. Figure 3.3b shows a simple directed acyclic graph. When talking about DAGs, we refer to nodes with in-degree 0 as *sources* and nodes with out-degree 0 as *sinks* of the graph. A *subgraph* $G' \subseteq G$ of a DAG G is a DAG that with $G'(V) \subseteq G(V)$ and $G'(E) \subseteq G'(V) \times G'(V)$. The *induced subgraph* $G[V'] \subseteq G$ for some $V' \subseteq G(V)$ is the subgraph of G with $(G[V'])(V) = V'$ and $(G[V'])(E) = G(E) \cap (V' \times V')$.

A *topological ordering* $\tau : V \to [n]$ of the nodes of a directed graph is an order such that for every edge $(u,v) \in E$, $\tau(u) < \tau(v)$ holds. We use $[n]$ to denote the set $\{1, \ldots, n\}$. As shown in Theorem 2.1.2, the existence of a topological ordering characterizes directed acyclic graphs.

Lemma 2.1.1. *(Source: [8, Proposition 1.4.2].) Let $G = (V, E)$ be a DAG. Then G contains a node v with in degree $d^-(v) = 0$.*

Proof. Let $P = (v_1, \ldots, v_k)$ be a path of maximal length in G. Since G is acyclic, v_1 cannot have a predecessor in P and since P is maximal, it also cannot have a predecessor outside of P. Therefore, v_1 cannot have any predecessors, i.e., $d^-(v_1) = 0$. □

Theorem 2.1.2. *(Source: [8, Proposition 1.4.3].) A directed graph is acyclic if and only if there exists a topological order of the nodes of the graph.*

Proof. Let $G = (V, E)$ be a directed acyclic graph. By Lemma 2.1.1, $G_0 \coloneqq G$ has a node v_0 with in degree zero. Set $\tau(v_0) = 0$ and move on to $G_1 \coloneqq G_0 - v$. Since

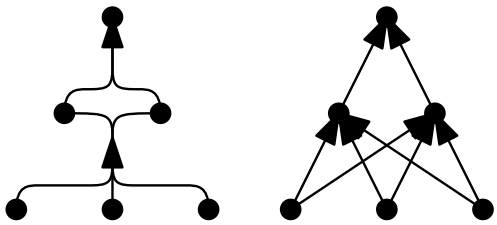
removing a node from an acyclic graph keeps the graph acyclic, G_1 again contains a node v_1 with in degree 0 and we can set $\tau(v_1) = 1$ and so on. By induction, this constructs a topological order τ of G.

For the other direction, let τ be a topological ordering and assume that G contains a cycle $C = (v_1, \ldots, v_k, v_{k+1} = v_1)$. Then $\tau(v_1) < \tau(v_2) < \cdots < \tau(v_k) < \tau(v_1)$, a contradiction. □

We now generalize the concept of graphs to *hypergraphs* by allowing edges to contain an arbitrary number of nodes. More formally, an *undirected weighted hypergraph* $H = (V, E, c, \omega)$ consists of a finite set of hypernodes V and a finite set of hyperedges E, where each hyperedge $e \in E$ is a non-empty subset of V, i.e., $e \subseteq V$. Hyperedges are also referred to as *nets* and the hypernodes contained in a net are its *pins*. Figure 2.1c illustrates a simple hypergraph. For a hypernode u, we define its set of incident hyperedges $I(u) \coloneqq \{e \in E \mid u \in e\}$, and its neighborhood $\Gamma(u) \coloneqq \{v \mid \{u, v\} \subseteq e \text{ for some } e \in E\}$. The *size* of a net is its cardinality $|e|$. A hypergraph where every edge has the same cardinality r is said to be r-uniform. In particular, a 2-uniform undirected hypergraph is an undirected graph. Analogously to undirected graphs, $c : V \to \mathbb{R}_{>0}$ assigns each hypernode a non-negative hypernode weight and $\omega : E \to \mathbb{R}_{>0}$ assigns each hyperedge a non-negative hyperedge weight.

The generalized version of directed graphs are *directed hypergraphs*. A directed hypergraph is an undirected hypergraph where each hyperedge $e \in E$ is divided into a set of tails $e^T \subseteq e$ and heads $e^H \subseteq e$ that fulfill $e^T \cup e^H = e$ and $e^T \cap e^H = \emptyset$. Note that in this thesis, we only consider directed hypergraphs where each hyperedge contains at most one head pin and an arbitrary number of tail pins, i.e., hypergraphs with $|e^H| = 1$ for all $e \in E$. The predecessors of a hypernode u are $\Gamma^-(u) \coloneqq \{v \mid v \in e^T, u \in e^H \text{ for some } e \in E\}$ and its successors are $\Gamma^+(u) \coloneqq \{v \mid u \in e^T, v \in e^H \text{ for some } e \in E\}$. In a directed hypergraph, a cycle C of length k is a sequence of hypernodes, $C = (v_1, \ldots, v_k, v_{k+1} = v_1)$, such that for every $i = 1, \ldots, k$, there exists some hyperedge $e \in E$ with $v_i \in e^T$ and $v_{i+1} \in e^H$. Furthermore, we require that $v_i \neq v_j$ for $i \neq j$, $1 \leq i, j \leq k$. Analogously to directed acyclic graphs, we refer to directed hypergraphs that do not contain any cycles as *directed acyclic hypergraph* or *DAH* for short. This definition of directed acyclic hypergraphs can be seen as an extension of Berge-acyclicity [10] to directed hypergraphs: Consider a bipartite graph G that contains one node for each hypernode and one node for each hyperedge of the hypergraph. For each $e \in E(H)$, add edges (u, e) for $u \in e^T$ and (e, v) for $v \in e^H$ to G. Then G is acyclic if and only if H is acyclic. An example for a directed acyclic hypergraph is shown in Figure 2.1d.

Note that given a directed hypergraph H, another way to construct a directed graph G that is equivalent to the hypergraph in regards to the acyclicity constraint is to replace each hyperedge e with a directed, bipartite graph from e^T to e^H. G is



(a) DAH **(b)** DAG

Figure 2.2: By replacing each directed hyperedge of the DAH (Figure 2.2a) with a bipartite graph from the tails of the hyperedge to its heads, we get a DAG (Figure 2.2b) that is acyclic if and only if the DAH is acyclic. Moreover, a partition of the DAH is acyclic if and only if the corresponding partition of the DAG is acyclic.

acyclic if and only if H is acyclic and an acyclic partition of G is also an acyclic partition of H and vice-versa. Figure 2.2 shows an example of this transformation.

2.2 Problem Formulation

As we did in the previous section, we start this section by defining the graph partitioning problems for undirected graphs. Then, we introduce it for directed acyclic graphs before stating both definitions for undirected and directed acyclic hypergraphs.

Given a graph $G = (V, E)$, the k-way *graph partitioning problem* asks for a partition of V into k *blocks* $\Pi(V) \coloneqq \{V_1, \ldots, V_k\}$ such that $V = \bigcup_{1 \leq i \leq k} V_i$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The weight of each block is limited by the *imbalance parameter* $\epsilon \geq 0$ and might not be zero: we demand that $0 < c(V_i) \leq L_{\max} \coloneqq (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$ for all $1 \leq i \leq k$. This ensures that each block has roughly the same weight for small values of ϵ. The goal is to find a partition that minimizes an *objective function*. In this thesis, we only consider the *edge cut objective* defined by

$$\sum_{e \in \text{cut}} \omega(e),$$

where cut contains all edges with endpoints in two different blocks.

For directed acyclic graphs, we need the concept of the *quotient graph*. Given a directed acyclic graph $G = (V, E)$ with partition $\Pi(V) = \{V_1, \ldots, V_k\}$, the quotient graph is a directed graph with one node v_i for each block V_i and an edge (v_i, v_j) if G contains an edge (v'_i, v'_j) with $v'_i \in V_i$ and $v'_j \in V_j$. The quotient graph can also be seen as the graph resulting from G when contracting all nodes within a

block. With this in mind, the graph partitioning problem for directed acyclic graphs is the same as before, but extended by the *acyclicity constraint*, i.e., the constraint that the partition's quotient graph must be acyclic.

Finally, we define undirected and directed acyclic hypergraphs. Given an undirected hypergraphs $H = (V, E)$, a k-way partition $\Pi(V) = \{V_1, \ldots, V_k\}$ of H is a partition of V fulfilling the same conditions as before. For $u \in V$, we set $b[u]$ to the block containing u, i.e., $b[u] = i$ if and only if $u \in V_i$. Moreover, we define the *connectivity set* of a hyperedge e with $\Lambda(e) \coloneqq \{V_i \mid V_i \cap e \neq \emptyset\}$ and the *connectivity* of a hyperedge e with $\lambda(e) \coloneqq |\Lambda(e)|$. Two blocks $V_i, V_j \in \Pi$ are adjacent if there is a hyperedge $e \in E$ with $V_i \cap E \neq \emptyset$ and $V_j \cap E \neq \emptyset$. A hypernode u and a block V_i are adjacent if $u \notin V_i$ but there exists a hyperedge e with $u \in e$ and $e \cap V_i \neq \emptyset$. The most prominent objective functions in hypergraph partitioning are the *cut* and *connectivity* objectives. Given a partition $\Pi(V) = \{V_1, \ldots, V_k\}$, the former one is defined as

$$\sum_{e \in \text{cut}} \omega(e),$$

where cut contains all hyperedges that are cut, i.e., have $\lambda(e) > 0$. The connectivity metric is defined as the sum

$$\sum_{e \in \text{cut}} (\lambda(e) - 1)\omega(e).$$

In other words, a hyperedge that only contains hypernodes from one block does not contribute to the connectivity metric whereas an edge containing hypernodes from two different blocks contributes one and so on. Note that both metrics are equal to the ordinary edge cut metric used in graph partitioning for 2-uniform hypergraphs. This allows us to compare our algorithm to the current state-of-the-art for DAG partitioning later on.

The quotient graph Q of a partitioned directed acyclic hypergraph H again contains a node v_i for each block V_i and an edge (v_i, v_j) if H contains an hyperedge e with tail pins in V_i and head pins in V_j. More formally, $V(Q) \coloneqq \Pi$ and $E(Q) \coloneqq \{(V_i, V_j) \mid \exists e \in E(H) : e^T \cap V_i \neq \emptyset \text{ and } e^H \cap V_j \neq \emptyset\}$. The hypergraph partitioning problem for directed acyclic hypergraphs is the same as before, but with the further restriction that the resulting quotient graph must also be acyclic.

Both graph partitioning problems are NP-complete [30, 42] and there are no constant factor approximation algorithms [5, 42]. Since the corresponding hypergraph partitioning problems are generalized versions of them, they are also NP-complete. In practice, we must therefore focus on good heuristics to find high-quality partitions of large graphs.

3 Related Work

The general graph partitioning problem has been studied for a long time and subsequently, there is an enormous amount of preexisting literature. For a broad overview, we refer to existing literature [14, 58]. More specialized literature focuses on hypergraph partitioning and acyclic graph partitioning. In this chapter, we give a brief introduction to each topic. We start by giving a general introduction to graph partitioning. Afterwards, we introduce hypergraph partitioning by briefly presenting the core components of KaHyPar [2, 4, 27, 54, 55] and PaToH [15]. Finally, we discuss recent approaches for acyclic graph partitioning introduced by Moreira et al. [42, 43] and Herrmann et al. [25, 26].

3.1 Multilevel Graph Partitioning

Most recent graph partitioning algorithms employ the multilevel paradigm first introduced by Hendrickson and Leland [24]. Note that this scheme isn't limited to ordinary node-based graph partitioning, but has been applied successfully to a wide range of problems, such as sequential and distributed edge partitioning [38, 57], graph drawing [39] and even support-vector machines [56]. We also make heavy use of this scheme in this thesis. Therefore, we give a brief introduction to multilevel graph partitioning algorithms in this section. We start by outlining the multilevel partitioning scheme. Afterwards, we present the refinement algorithm by Fiduccia and Mattheyses (FM algorithm) [23] since we use this algorithm as a basis for our own refinement algorithm.

Outline. Multilevel graph partitioning typically consists of three phases, namely *coarsening*, *initial partitioning* and *uncoarsening* or *refinement*. The whole process is depicted in Figure 3.1. During coarsening, the algorithm constructs a hierarchy of roughly $\log(n)$ coarser graphs by contracting matchings or clusters. Both variants can be implemented efficiently. For instance, Birn et al. [12] present an efficient 2-approximative parallel matching algorithm with linear running time (on a single core) and low I/O complexity. Meyerhenke et al. [40] present a clustering algorithm using size-constrained label propagation which also has linear running time. Contracting a set $S \subseteq V$ of nodes works as follows: remove all nodes in S from

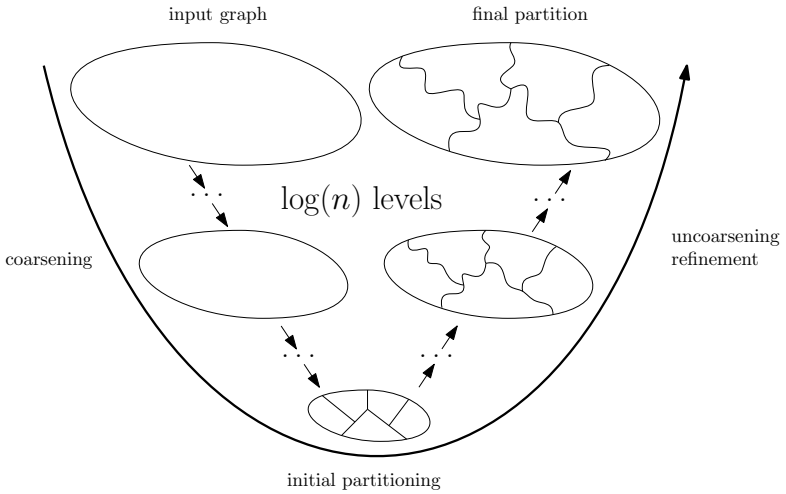


Figure 3.1: Multilevel partitioning scheme: coarsening, initial partitioning and uncoarsening with refinement.

the graph and insert a new node s with node weight $c(s) = c(S)$ and neighborhood $\Gamma(s) = \bigcup_{u \in S} \Gamma(u)$. If this process creates parallel edges $e_1, \ldots, e_m$, they are merged into one edge e with accumulated edge weight $\omega(e) = \omega(\{e_1, \ldots, e_m\})$. After roughly $\log(n)$ levels of coarser graphs, the coarsening algorithm decides that the graph is small enough and terminates. We use the following terminology when referring to different graphs of the hierarchy: the *finest graph* is the input graph, i.e., the graph without any contractions. When contracting the finest graph, we get a series of *coarser graphs*. The smallest graph is referred to as the *coarsest graph*. The next phase computes a partition for the coarsest graph. This partition is referred to as *initial partition*. Since the coarsest graph is small compared to the input graph, the initial partitioning can be relatively slow without affecting the overall running time much. During uncoarsening, the partition of a coarser graph is projected onto the finer graph. This is done by assigning all nodes that were contracted into one node to that nodes partition in the coarser graph. After each projection, a *refinement algorithm* improves the the current partition. The most prominent refinement algorithms are variants of the algorithm by Fiduccia and Mattheyses (FM) [23] or the one by Kernighan and Lin (KL) [35]. Since we use the FM algorithm as basis of our own refinement algorithm, we elaborate on it in the following paragraph. Once refinement on the finest level was executed, the algorithm terminates.

The intuition of multilevel graph partitioning is based on the following observations: first, projecting a partition from a coarser level onto a finer level preserves the edge cut. Thus, the final edge cut is *at least* as low as the edge cut of the initial partition, assuming that the refinement algorithm guarantees no worsening. The other observation is that the multilevel scheme allows a more global view on the partition

landscape compared to a single-level algorithm. During the coarser levels of the graph hierarchy, the refinement algorithm considers to move chunks of nodes (since a single node on the coarsest graph corresponds to many nodes on the finest graph), allowing a global view. During finer levels, refinement performs fine-grained moves.

Fiduccia–Mattheyses Algorithm. The FM algorithm [23] is a linear-time local search heuristic originally proposed as a 2-way refinement algorithm. The algorithm alternates between both blocks and moves the node with the highest gain (i.e., the change in the objective caused by the move) from the current block to the other one. To do that, the algorithm maintains two priority queues, one for each block. The queues are initialized with all nodes belonging to the corresponding block, with priority equal to their gain value. This allows an implementation with amortized linear running time when using bucket queues as priority queues; for details, see Ref. [23]. Once a node was moved, it is excluded from the rest of the current pass of the algorithm. The pass terminates as soon as one priority queue runs empty. Then, the algorithm undos moves in reverse order until the best partition observed during refinement is restored.

While Fiduccia and Mattheyses move all nodes during a pass of one iteration, Karypis and Kumar [34] reduce its running time significantly by only considering boundary nodes for movement and stopping the search after a constant number of fruitless moves, i.e., moves that did not yield an improved partition. Moreover, they extended the original 2-way FM algorithm to a k-way refinement algorithm using only a single global priority queue while maintaining the linear running time. Osipov and Sanders [47] present an adaptive stopping criterion (instead of a constant number of fruitless moves) based on a random walk model. The criterion terminates the current pass as soon as further improvements become *unlikely*. Moreover, Osipov et al. [52] introduce a highly localized version of the FM algorithm called multi-try FM. Instead of initializing a pass of the algorithm with all border nodes, they choose to repeatedly initialize it with only *a single node*. The intuition behind this improvement is that it allows the algorithm to find localized improvements before the partition gets trashed at other places.

For hypergraph partitioning, Akhremtsev et al. [2] implement a variant of the FM algorithm that maintains k priority queues, one for each block, and uses an extension of the adaptive stopping criterion introduced by Osipov and Sanders [47]. Each priority queue contains all hypernodes that can be moved to the corresponding block, i.e., that are in another block but adjacent to it, with the gain value as priority. Similar to multi-try FM [52], their FM implementation is highly localized and gets repeatedly initialized with only a couple of hypernodes. Since calculating and updating gain values on hypergraphs is expensive, they implement several novel techniques to improve the performance of their FM algorithm. First and foremost,

they cache gain values across multiple passes to prevent having to calculate the same gain value multiple times. Secondly, they exclude large hyperedges since those bottleneck the gain value calculation and are unlikely to change their connectivity.

3.2 Undirected Hypergraph Partitioning

In this section, we briefly introduce hypergraph partitioning by presenting two algorithms for undirected hypergraph partitioning, namely KaHyPar [2,4,27,28,55] and PaToH [15]. We present KaHyPar because it computes the best partitions out of all publicly available HGP algorithms on a large collection of hypergraph instances [54] and we use it as a framework to implement our own algorithms. PaToH is generally one of the fastest HGP algorithm [54] and we use it during initial partitioning in our of our experiments. For an in-depth introduction to high-quality hypergraph partitioning and KaHyPar, we refer to the doctoral thesis of Sebastian Schlag [54].

3.2.1 KaHyPar

Many recent hypergraph partitioners [15, 21, 32, 59] and graph partitioners [29, 33, 50, 52] use the multilevel paradigm first introduced by Hendrickson and Leland [24] with roughly $\log(n)$ levels to partition an undirected hypergraph or undirected graph. KaHyPar uses an extreme version of this paradigm known as the n-level multilevel graph partitioning paradigm that was first introduced by Osipov and Sanders with KaSPar [47].

Like multilevel algorithms for graph partitioning, KaHyPar constructs a hierarchy of coarser graphs by contracting pairs of hypernodes during coarsening. Contracting a pair (u, v) of hypernodes works as follows: remove both hypernodes from the hypergraph and replace them by a new hypernode w with hypernode weight $c(w) = c(u) + c(v)$. Add w to every hyperedge that contained u or v. If this process yields parallel hyperedges $e_1, \ldots, e_k$, the parallel hyperedges are replaced by a single hyperedge e with hyperedge weight $\omega(e) = \omega(\{e_1, \ldots, e_k\})$. In contrast to the normal multilevel paradigm, KaHyPar stores the resulting hypergraph *after every single* pair contraction in a hierarchy of coarser hypergraphs, giving the paradigm its name. At some point, the coarsening algorithm decides that the hypergraph is small enough and terminates. KaHyPar then computes an initial partition for the coarsest hypergraph. During uncoarsening, the hierarchy is unrolled and the hypernodes get uncontracted in reverse order. After each uncontraction operation, a localized refinement algorithm tries to improve the connectivity of the partition around the uncontracted hypernodes. Once all hypernodes were uncontracted, the partitioning process is complete.

KaHyPar implements various algorithms for each phase. For coarsening, KaHyPar rates pairs of adjacent hypernodes according to the *heavy-edge* rating function [55] and contracts the pair with the highest rating next. For initial partitioning, it implements several algorithms including random hypernode assignment, breath-first search and greedy hypergraph growing to obtain an initial k-way partitioning using recursive bisection. Finally, KaHyPar implements several refinement algorithms, namely localized 2-way FM refinement [55], localized k-way FM refinement [2] and 2-way flow based refinement algorithms [27]. Further techniques outside the n-level paradigm include the detection of community structures to guide the coarsening algorithm [28] and a memetic algorithm [4]. For more details on the various component as well as a general introduction to hypergraph partitioning, we refer to Ref [54].

3.2.2 PaToH

PaToH [15] is a multilevel hypergraph partitioning algorithm with roughly $\log(n)$ levels by Çatalyürek and Aykanat that originated from sparse-matrix vector multiplication. It optimizes the connectivity objective and uses recursive bisection to obtain a k-way partition. During coarsening, PaToH constructs a hierarchy of coarser hypergraphs using either a matching based hierarchical clustering algorithm or a hierarchic-agglomerative clustering algorithm. Then, it obtains an initial partition on the coarsest hypergraph using greedy hypergraph growing. Refinement is done using a variant of the 2-way FM algorithm that only keeps border hypernodes in its priority queues. After every mode, the algorithm updates the gain values of its adjacent hypernodes, removes hypernodes that are no longer boundary hypernodes from its priority queue and adds those that became border nodes due to the move.

3.3 Acyclic Graph Partitioning

Research on acyclic graph partitioning dates back multiple decades, although early work only considers simple heuristics to partition the graph. Two approaches make use of a maximum-fanout-free cone clustering [16] of the graph, a technique commonly used to make circuits sparser. Kocan et al. [37] cluster the graph and greedily join the relatively small clusters into blocks to obtain a final partition. Cong et al. [19] obtain an initial partition based on a topological ordering of the graph, then cluster it and use a variant of the FM algorithm with cycle detection to improve the partition on the clustered graph. In a subsequent work, Cong et al. [17] improve on their initial research by presenting `FLARE`, a 2-level partitioning algorithm using edge separability-based circuit clustering [18] and scheduled 2-way FM refinement. These approaches are superseded in terms of solution quality by recent multilevel

algorithms from Herrmann et al. [25, 26] and Moreira et al. [42, 43]. The former presents new coarsening algorithms to produce acyclic coarser graphs and shows how to use a pre-existing undirected graph partitioner to obtain an initial partition of a DAG. The latter introduces more sophisticated k-way refinement algorithms and an evolutionary algorithm. To the best of our knowledge, the latest work of Herrmann et al. [25] is the current state-of-the-art in terms of solution quality. Since we make use of these techniques, we present them in detail in the following sections.

3.3.1 Initial Partitioning

In contrast to the classic multilevel partitioning scheme, both Herrmann et al. and Moreira et al. compute the initial partition of the DAG on the finest level of the graph hierarchy, i.e., before computing the coarser graphs. Moreira et al. do this because their coarsening algorithm might create cycles in the graph, making it potentially impossible to obtain a balanced initial partition otherwise. Herrmann et al. tried both variants but report better results for computing the initial partition on the input graph.

Moreira et al. [42] use a simple greedy heuristic to obtain an initial k-way partition of the input DAG that fulfills the acyclic constraint. To be more precise, they compute a topological order τ of the DAG and assign nodes $\tau^{-1}(i \cdot \lceil \frac{n}{k} \rceil), \ldots, \tau^{-1}((i+1) \cdot \lceil \frac{n}{k} \rceil - 1)$ to block i, $i = 0, \ldots, k-1$. While the resulting partition is acyclic and balanced, the heuristic ignores its edge-cut.

Herrmann et al. [26] propose a more sophisticated heuristic for computing initial bipartitions that makes use of pre-existing undirected graph partitioners such as METIS [33]. The algorithm is outlined in Figure 3.2: first, they treat the DAG as undirected graph and use the pre-existing undirected graph partitioner to obtain an initial bipartition. The resulting bipartition is balanced and optimizes the correct objective, but does generally violate the acyclic constraint. Therefore, additional work is required to restore the constraint.

3.3.2 Coarsening

Moreira et al. [43] use sized-constrained label propagation [40] to identify clusters and contract them. In regards to the acyclic property of the directed graph, they contract arbitrary sets of nodes which can cause the coarser graph to become cyclic, although this is unproblematic when calculating the initial partition before coarsening.

Herrmann et al. [26] present a novel coarsening algorithm that computes an acyclic coarser graph. On a high level, the algorithm computes a clustering of the graph while avoiding *forbidden edges*, i.e., edges that might

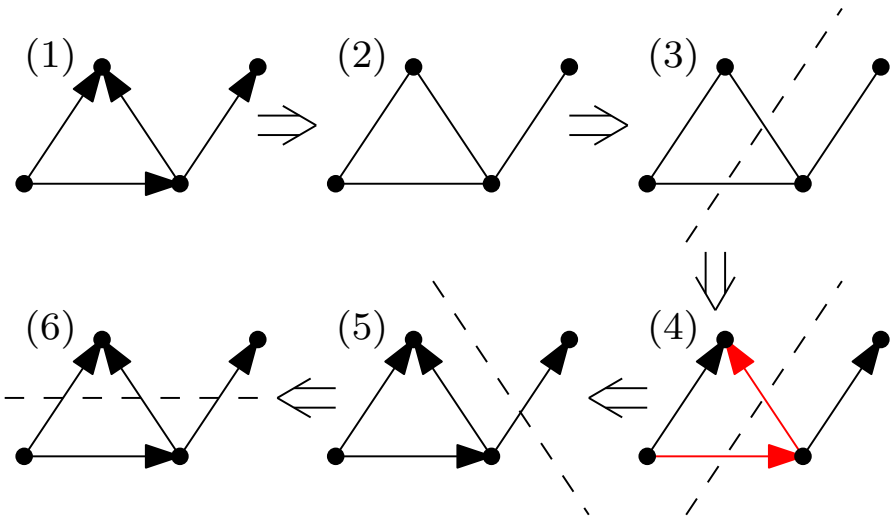


Figure 3.2: Obtaining an initial partition using a pre-existing initial partitioner: Treat the **(1)** directed input graph as **(2)** undirected graph and **(3)** use a pre-existing graph partitioner to partition the undirected graph. The **(4)** resulting partition might violate the acyclic constraint, which **(5)** can be restored by moving nodes. Those movements might imbalance the partition, requiring **(6)** a final re-balance step.

induce a cycle when contracted. Afterwards, it simultaneously contracts all clusters to obtain the coarser graph.

To identify forbidden edges, they introduce the concept of *toplevel values* as outlined in Definition 3.3.1.

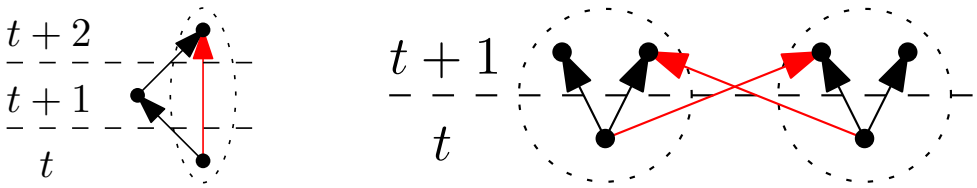
Definition 3.3.1. *(Toplevel of a node. Source: [26].) Let $G = (V, E)$ be a DAG. The* toplevel *of a node $v \in V$, denoted by $\texttt{top}[v]$, is the length of a longest path from any source in G to v. In particular, sources s of the graph have toplevel $\texttt{top}[s] = 0$.*

Observe that the contraction of an edge $e = (u, v)$ induces a cycle in the coarser graph if and only if the graph contains an u-v-path avoiding e. Theorem 3.3.2 states conditions that are sufficient to identify such edges.

Theorem 3.3.2. *(Source: [26].) Let $G = (V, E)$ be a DAG and $C = \{C_1, \ldots, C_k\}$ be a clustering of V. If C is such that*

- *for any clustering C_i and for all $u, v \in C_i$, $|\texttt{top}[u] - \texttt{top}[v]| \leq 1$, and*
- *for two different clusters C_i and C_j and for all $u \in C_i$ and $v \in C_j$, either $(u, v) \notin E$, or $\texttt{top}[u] \neq \texttt{top}[v] - 1$,*

then contracting all clusters in C yields an acyclic coarser graph.



(a) Contracting the cluster induces a cycle of length 2. The cluster is forbidden by the first condition since the head of the red edge has toplevel $> t + 1$.

(b) Contracting both clusters induces a cycle of length 2. The clustering is forbidden by the second condition since the red edges connect nodes with toplevel difference ≤ 1 from different clusters.

Figure 3.3: Cyclic formations prevented by the **(a)** first and **(b)** second condition of Theorem 3.3.2.

The first condition prevents the formation of cycles that only contain one of the contracted clusters, while the second condition prevents the formation of cycles containing multiple contracted clusters. An example for each case is given in Figure 3.3.

Initially, all nodes are in their own cluster. The algorithm iterates over all nodes still in a singleton cluster and tries to add it to one of the neighboring clusters. A node can only be added to a cluster if it does not violate the criteria formulated in Theorem 3.3.2. The algorithm then selects the best neighboring cluster based on some rating function such as the edge weight and adds the node to the cluster. If no neighboring clusters are viable, the node stays in its singleton cluster.

3.3.3 Refinement

Moreira et al. [42] describe simple k-way refinement algorithms that move nodes with the highest *gain values*, where the gain value of a move is the change of the objective function on the partition caused by it. Since arbitrary node movements can cause the partition to become cyclic, they propose several restrictions on the set of potential movements and suggest the use of an online cycle detection algorithm. For the following paragraphs, let $\Pi = \{V_1, \ldots, V_k\}$ be the acyclic k-way partition with quotient graph Q and let $\tau : Q(V) \to [k]$ be a topological order of the quotient graph. They propose the following restrictions on possible target blocks of a node.

The *simple move* heuristic only considers to move a node in block $\tau^{-1}(i)$ to blocks $\tau^{-1}(i-1)$ (if the node does not have predecessors in its current block) and $\tau^{-1}(i+1)$ (if the node does not have successors in its current block). Observe that a simple move does not change the topological order of the quotient graph. The *advanced move* heuristics determines the highest block $\tau^{-1}(a)$ (in the topological order) containing a predecessor of a node and the lowest block $\tau^{-1}(b)$ containing

a successor. Then, the viable target blocks are $\tau^{-1}(a), \tau^{-1}(a+1), \ldots, \tau^{-1}(b-1), \tau^{-1}(b)$. An advanced move might add a new edge to the quotient graph, but the topological order is preserved. *Global moves* consider all target blocks. Whenever a global move would introduce a new edge in the quotient graph, they use Kahn's algorithm [31] to determine whether the move creates a cycle in the quotient graph. Although Kahn's Algorithm has linear running time, this approach is viable since the size of the quotient graph is usually small.

Both Moreira et al. [42] and Herrmann et al. [26] suggest the use of the 2-way FM algorithm. Moreira et al. schedule the FM algorithm on pairs on blocks to refine a k-way partition, whereas Herrmann et al. use recursive bisection and only consider bipartitions. In 2-way refinement, there is no need for cycle detection since all moves are simple moves. Therefore, it is easy to identify movable nodes.

3.4 Memetic Algorithms

Memetic algorithms are inspired by the theory of evolution and were first introduced by Pablo Moscato [44]. For a broad introduction to memetic algorithms, we refer to the work by Moscato and Cotta [45]. Here, we only summarize the metaheuristic briefly. The basic building blocks of memetic algorithms are *mutation*, *recombination*, *local search*, a *fitness function*, and a population of individuals (i.e., solutions to the optimization problem). Initially, the algorithm produces individuals that form the initial population. Then, mutation and recombination operations are repeated until the population converges or the time limit is exceeded. Both operations produce offsprings that form the next generation of individuals. If the population is full, old individuals have to be evicted to make room for the new ones. To chose which individuals to evict, the fitness function is used. In general, individuals with low fitness are most likely to evicted from the population. The operations to produce offsprings work as follows. Mutation selects on individual and changes it, for example by applying local search to it. Recombination selects two individuals and combines them, forming the offspring.

KaHyPar also includes a memetic algorithm [4], which uses natural combine operations provided by the multilevel partitioning scheme. Since we use this implementation as starting point for our own memetic algorithm, we briefly describe its core components. The algorithm first produces a population of high-quality hypergraph partitions obtained using KaHyPar-C [28]. For mutation, it selects a random individual and iterates coarsening and local search using random seeds. For recombination, it selects two individuals using tournament selection [41] and combines them using the following operation: Coarsen the graph, but only contract pairs of hypernodes that are in the same blocks in both individuals. This ensures

that either individual can be used as initial partition on the coarsest hypergraph. They use the better one and proceed with unrolling the hypergraph hierarchy while improving the partition using local search. When inserting a new offspring into the population, it evicts the individual most similar to the new offspring to ensure a diverse population. Furthermore, KaHyPar implements more sophisticated mutation and recombination operations to produce a more diverse population.

Besides undirected hypergraph partitioning, memetic algorithms have been applied successfully to a broad field of problems, including graph partitioning [53] and clustering [11], node separators [53], and the territory design problem [1]. Recently, Moreira et al. [43] proposed a memetic algorithm for the DAG partitioning problem. For more applications and trends, we refer to recent surveys [36,46] on the subject.

4 Acyclic Hypergraph Partitioning

Our hypergraph partitioner with acyclicity constraint is based on KaHyPar [2,4,55], a state-of-the-art hypergraph partitioner. To cope with the acyclicity constraint, we extend KaHyPar with directed hypergraphs and implement new algorithms for coarsening, initial partitioning, and the refinement of directed hypergraphs and acyclic partitions. We start this chapter with an outline that summarizes the interaction between each component of our DAH partitioner. The following subchapters then explore each component in detail.

Outline. Just like Moreira et al. [42, 43] and Herrmann et al. [25, 26], we start the partitioning process by computing an initial partition of the hypergraph on the input hypergraph. In this phase, we consider two alternatives: partitioning via topological order and partitioning via a pre-existing undirected hypergraph partitioner. Once the initial partition has been obtained, we run a 2-way refinement algorithm on the initial partition, followed by a V-cycle structured as follows. For coarsening, we use our acyclic coarsening algorithm described in Chapter 4.2.2. Since the hypergraph is already partitioned, the coarsening algorithm does not contract pairs of hypernodes in different blocks. This ensures that the initial partition can be projected onto the coarsest hypergraph of the n-level hypergraph hierarchy. The contracted hypernodes are then uncontracted. After each uncontraction, we run our 2-way refinement algorithm initialized with the uncontracted hypernodes. Once we obtained a k-way partition, we run a V-cycle with our k-way refinement algorithm presented in Chapter 4.2.3. The whole algorithm is also outlined in Figure 4.1. To further improve the result of our multilevel algorithm, we introduce an evolutionary algorithm in Chapter 4.4.

4.1 Data Structure for Directed Hypergraphs

In memory, KaHyPar represents hypergraphs as undirected bipartite graphs stored in an adjacency array [55]. The bipartite graph contains one node for each hypernode and one for each hyperedge. The nodes representing hypernodes have outgoing edges to the nodes representing incident hyperedges and analogously, nodes representing hyperedges are connected to nodes representing their pins.

We extend this format by introducing a *head counter* h for each hypernode and each hyperedge of the bipartite graph. For hyperedges, h stores the number of head pins and for hypernodes u, it stores the number of hyperedges that contain u as a head pin. During hypergraph construction, we ensure that head pins are placed first in the adjacency array of a hyperedge and likewise, we ensure that hyperedges containing a head pin u are placed first in the adjacency array of u. This allows us to effectively iterate over only the heads or tails of a hyperedge.

4.2 n-Level Acyclic Hypergraph Partitioning

Recall that KaHyPar uses the n-level hypergraph partitioning scheme [47] described in Chapter 3.2.1 to obtain hypergraph partitions of high quality. We use this implementation and exchange the algorithms used for initial partitioning, coarsening and refinement with our own. These algorithms are presented in the following sections.

4.2.1 Initial Partitioning

This section describes our approaches for obtaining an initial partition of the directed acyclic hypergraph. Each algorithm starts with an unpartitioned directed acyclic hypergraph $H = (V, E)$ and produces a partition of V into blocks $V_1, \ldots, V_k$ for a fixed number of blocks k. Recall that depending on the configuration, H could refer to the input hypergraph or the coarsened hypergraph, i.e., the output of the coarsening algorithm described in Chapter 4.2.2.

Initial Partitioning via Topological Ordering. Recall that Moreira et al. [42] compute their initial partition based on a topological ordering of the graph. We implement the same approach for directed acyclic hypergraphs to obtain an initial k-way partition. First, we calculate a topological ordering of the nodes of the hypergraph using Kahn's algorithm [31] adapted for directed hypergraphs (Algorithm 1). Using the topological order, our algorithm greedily assigns nodes to blocks until they are full. More precisely, it assigns the first nodes to the first block until its weight exceeds $\lceil \frac{c(V)}{k} \rceil$ before it assigns the next nodes to block two and so on. Note that this approach always produces a balanced initial partition for hypergraphs with unit node weight. For weighted hypergraphs, it might produce an initial partition violating the balance constraint due to the greedy assignment of nodes to blocks. In this case, the refinement step must balance the partition.

Initial Partitioning via Undirected Partitioning. This algorithm is based on the initial partitioning algorithm for DAG partitioning presented by Herrmann et al. [26]

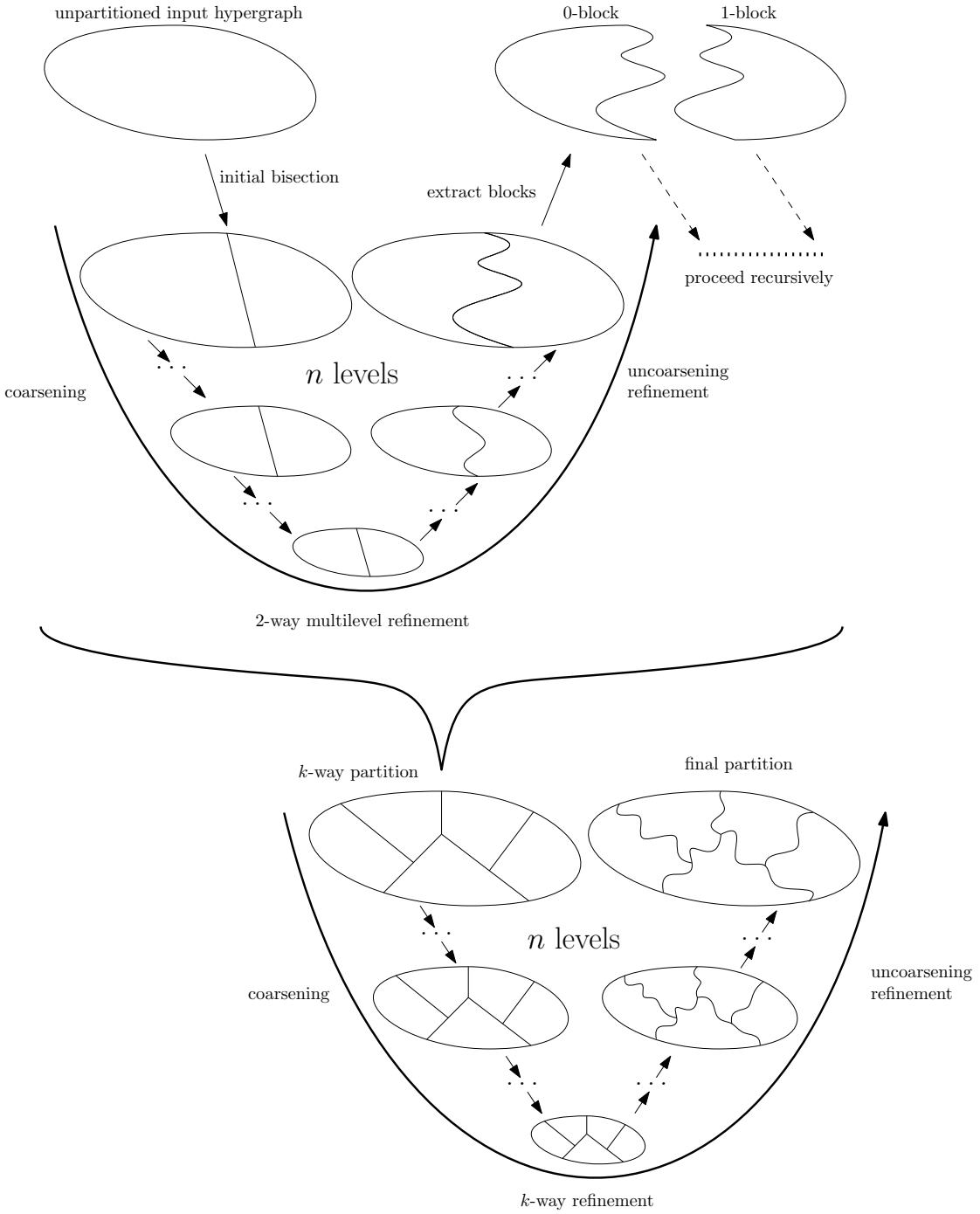


Figure 4.1: High-level overview over our algorithm with all components. First, we obtain an initial partition using recursive bisection. For each bisection step, we use one of the initial partitioning algorithms. Then, we improve the partition using our 2-way refinement algorithm. Once the hypergraph is partitioned into k blocks, we run one V-cycle with our k-way refinement algorithm.

Algorithm 1: Kahn's algorithm [31] for DAHs.

```
Data: DAH H = (V, E).
Result: Topological ordering τ of H.
1 i ← 0
2 while exists v ∈ V with d⁺(v) = 0 do
3 |   τ(i) = v
4 |   H ← H − v
5 |   i ← i + 1
6 return τ
```

described in Chapter 3.3.1. Given a DAH, we compute an initial bipartition of the hypergraph as follows. First, we turn the DAH into an undirected hypergraph by merging the tails and heads of each hyperedge. Note that due to our data structure, this is merely a conceptual step and does not require any actual work. Next, we pass the undirected hypergraph to a preexisting hypergraph partitioner that minimizes the connectivity metric to obtain an initial bipartition. We use KaHyPar-MF [27] and PaToH [15]. To the best of our knowledge, KaHyPar-MF regularly finds partition with the lowest connectivity metric out of all hypergraph partitions while PaToH is the fastest one. This bipartition is the projected onto the original DAH. In general, the resulting bipartition violates the acyclicity constraint and therefore, we must perform further steps to make it acyclic.

The algorithm to make the bipartition acyclic is shown in Algorithm 2. On a high level, we select one edge in the quotient graph that we want to remove and move hypernodes from one block to the other one accordingly. Denote the two blocks by V_1 and V_2 and assume that we want to remove the quotient graph edge from V_1 to V_2. We start a breadth-first search at every hypernode in V_1 that has successors in V_2. The search only scans successors in V_2 and moves every node from V_2 to V_1. Once the search has completed, no hypernode in V_1 has successors in V_2 and therefore, the quotient graph edge from V_1 to V_2 is removed.

The resulting acyclic partition might become imbalanced due to the movements from one block to the other one. To cope with this problem, we run an additional balancing step afterwards. This step simply moves hypernodes from the overloaded block to the underloaded block. Note that cannot move arbitrary hypernodes while keeping the bipartition acyclic. More precisely, if we have an acyclic bipartition with blocks V_1 and V_2 and a quotient graph edge from V_1 to V_2, we can only moves hypernodes in V_1 that have no successors in V_1. In an effort the keep the connectivity of the bipartition low, we sort the movable hypernodes in the overloaded block by their gain value using a priority queue. The whole process in depicted in Algorithm 4.

Algorithm 3 puts it all together: first, it obtains a bipartition violating the acyclic constraint. Then, it runs Algorithm 2 and Algorithm 4 twice, once removing the quotient graph edge from V_1 to V_2 and once removing the reverse edge. Finally, it selects the bipartition with the lower connectivity and returns it. We run this algorithm twice; once as depicted, and then a second time using predecessors instead of successors in Algorithm 2. We then select the best partition out of all options.

We also considered to skip the balancing step by using an asymmetric number of further bisections to split the resulting two blocks: after bisecting a block B (that should ultimatly be split into k blocks) of weight $c(B)$ into smaller blocks B_1 and B_2, we set $k_1 \coloneqq \lceil k \cdot \frac{c(B_1)}{c(B)} \rceil - 1$ and $k_2 \coloneqq \lfloor k \cdot \frac{c(B_2)}{c(B)} \rfloor - 1$ and continue by splitting B_i into k_i blocks, $i = 1, 2$. Once the algorithm obtained all k blocks, we use the hard rebalancing algorithm described in Chapter 4.3.2 to ensure a balanced k-way partition. However, since initial experiments did not show an improvement over Algorithm 3, we did not pursue this approach any further.

Algorithm 2: Subroutine `FixCyclic`$(\cdot)$ referenced in Algorithm 3: moves nodes to make a bipartition acyclic.

```
Data: Cyclic bipartition (V_1, V_2) of DAH H = (V, E).
Result: Acyclic bipartition.
S ≔ new Stack()
for u ∈ V_1 with Γ⁺(u) ∩ V_2 ≠ ∅ do  S ≔ S ∪ {u}
while S ≠ ∅ do
    u ≔ S.pop()
    for v ∈ Γ⁺(u) ∩ V_2 do
        S ≔ S ∪ {v}
        V_1 ≔ V_1 ∪ {v}
        V_2 ≔ V_2 \ {v}
return (V_1, V_2)
```

Algorithm 3: Initial partitioning algorithms that makes use of a preexisting hypergraph partitioner `HG`$(\cdot,\cdot)$ for undirected hypergraphs.

```
Data: DAH H = (V, E).
Result: Bipartition V = V_1 ∪̇ V_2.
(V_1, V_2) ≔ HG(H, k)                          // partition as undirected hypergraph
(V'_1, V'_2) ≔ Balance(FixCyclic(V_1, V_2))    // break (V_1, V_2)
(V''_1, V''_2) ≔ Balance(FixCyclic(V_2, V_1))  // break (V_2, V_1)
return min{(V'_1, V'_2), (V''_1, V''_2)}  // select bipartition with lower connectivity
```

Algorithm 4: Subroutine `Balance`$(\cdot)$ referenced in Algorithm 3: moves nodes from the overloaded block to the underloaded one to balance the bipartition.

```
Data: Imbalanced acyclic bipartition (V_1, V_2) of DAH H = (V, E). Let V_1
      be the overloaded and V_2 be the underloaded block and assume that
      the quotient graph edge goes from V_1 to V_2.
Result: Acyclic balanced bipartition.
Q := new PriorityQueue()
for u ∈ V_1 do
    if Γ^+(u) ∩ V_1 = ∅ then
        Q.insert(Gain(u), u)
while c(V_1) > (1 + ε)⌈c(V)/2⌉ do
    u := Q.deleteMax()
    V_1 := V_1 \ {u}
    V_2 := V_2 ∪ {u}
    UpdateGainValues(Q)      // update gain values of neighbors of u in Q
    UpdateMovableHypernodes(Q)   // remove/add hypernodes from/to Q
return (V_1, V_2)
```

4.2.2 Coarsening

The coarsening phase iteratively selects a set of hypernodes and contracts them, yielding a hierarchy of n levels of coarser hypergraphs. When contracting pairs of hypernodes, the hypergraph can become cyclic as illustrated in Figure 4.2. We explore two types of coarsening algorithms: the first one is the coarsening algorithm already implemented in KaHyPar-K. It selects pairs of hypernodes to be contracted without constraints in regards to keeping the directed hypergraph acyclic. The second approach restricts the algorithm implemented in KaHyPar-K to pairs of hypernodes that can be contracted safely while keeping the hypergraph acyclic. This approach is based on the acyclic clustering by Herrmann et al. [26] presented in Chapter 3.3.2 and described in the rest of this chapter.

When contracting a pair of hypernodes that are both pins of the same hyperedge, but one is one of the hyperedge's tails and the other one is one of the hyperedge's heads, it is not obvious whether the resulting hypernode should be a tail or a head of the hyperedge. Recall that in this thesis, we focus on directed hypergraphs where each hyperedge contains *at most* one head. For those instances, whenever one of the contraction partners is a head of a hyperedge e, the resulting hypernode is also a head of e. For the more general case, we propose two differ-

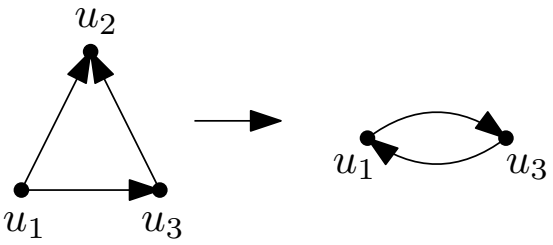


Figure 4.2: Acyclic directed hypergraph with three hypernodes u_1, u_2, u_3 on the left. Contracting the hyperedge $\{u_1, u_2\}$ yields the cyclic hypergraph on the right.

ent solutions. First, one could exclude such pairs altogether, i.e., only contract pairs of hypernodes that have the same role in all shared hyperedges. We ran some initial experiments with this setting and observed that it seemed to decrease partition quality. Hence, we did not pursue it further. As an alternative solution (that we did not implement), we propose to introduce *mixed pins* and leave it to the refinement algorithms to treat them appropriately.

Acyclic Hypergraph Coarsening. The algorithm is based on Theorem 4.2.2, which identifies pairs of hypernodes that may not be in the same cluster.

Definition 4.2.1. *(Mixed-level and single-level clusters.) Let $H = (V, E)$ be a DAH and $C = \{C_1, \ldots, C_k\}$ be a clustering of V such that for each $C_i \in C$ and for all $u, v \in C_i$, $|\texttt{top}[u] - \texttt{top}[v]| \leq 1$. We refer to clusters C_i where all $u, v \in C_i$ have $\texttt{top}[u] = \texttt{top}[v]$ as* single-level *clusters and clusters C_j that contain at least one pair of nodes $u, v \in C_j$ with $|\texttt{top}[u] - \texttt{top}[v]| = 1$ as* mixed-level *clusters.*

Theorem 4.2.2. *(Based on Theorem 3.3.2.) Let $H = (V, E)$ be a DAH and $C = \{C_1, \ldots, C_k\}$ be a clustering of V, such that*

- *for any clustering C_i and for all $u, v \in C_i$, $|\texttt{top}[u] - \texttt{top}[v]| \leq 1$, and*
- *for two different **mixed-level clusters** C_i and C_j and for all $u \in C_i$ and $v \in C_j$, either $(u, v) \notin E$ or $|\texttt{top}[u] - \texttt{top}[v]| > 1$.*

Then the coarser hypergraph $H \circ C$ is acyclic.

Proof. First, recall that a DAH can be seen as a DAG by replacing each hyperedge e with a directed bipartite graph from e^T to e^H. This transformed graph is acyclic if and only if the DAH is acyclic. Hence, we can proof the theorem based on graphs rather than hypergraphs.

Let G be the corresponding DAG. Assume that $G \circ C$ contains a cycle. By Theorem 3.3.2, the cycle must contain at least one single-level cluster C_i. Moreover, since the nodes of C_i have the same toplevel, the cycle must have length at least 2.

Let C_{i-1} be the predecessor and C_{i+1} be the successor of C_i in the cycle. Let t be the lowest toplevel of nodes in C_{i-1}. Then the toplevel of nodes in C_i and C_{i+1} is at least $t+1$, which forbids a path from C_{i+1} to C_{i-1}, a contradiction. □

Note that the difference between Theorem 3.3.2 and Theorem 4.2.2 lies in the distinction between single-level and mixed-level clusters: the second condition must only hold for pairs of mixed-level clusters. Since adjacent nodes in DAGs always have different toplevels, the clustering algorithm by Herrmann et al. [25] only produces mixed-level clusters. In contrast, DAHs might contain adjacent hypernodes with the same toplevel, justifying this distinction.

Based on this theorem, our clustering algorithm works as follows. We start on the input hypergraph and compute all toplevels, then start the clustering process. Generally, we only consider putting hypernodes in the same cluster whose toplevel differs by at most one, i.e., we only build clusters that do not violate the first condition of Theorem 4.2.2. Since the second condition is a fairly strong restriction, we follow Herrmann et al. [25] and opt to allow clusters violating the condition. Instead, whenever we add a hypernode that a cluster that would violate the second condition of the theorem, we run a cycle detection subroutine to ensure that the operation does not induce a cycle in the coarser hypergraph. Once the clustering has been computed, we contract pairs of hypernodes placed in the same cluster pair by pair, yielding one coarser graph after each contraction. If the coarser hypergraph after the last contraction is still too large, we repeat the algorithm. We stop it once one the hypergraph is small enough (i.e., has less than $180 \cdot k$ hypernodes, the same stopping criterion used in KaHyPar [2]) or one repetition of the algorithm could not find any more non-singleton clusters.

More precisely, our algorithm works as follows. At the beginning of the algorithm, all hypernodes are in their own singleton cluster. For each hypernode u that is still in a singleton cluster, we rate each neighbor using the heavy-edge metric already implemented in KaHyPar. We select the highest rated neighbor v whose cluster can include u without violating the first condition from Theorem 4.2.2. If all hypernodes in v's cluster have the same toplevel as u, we know that we can safely add u to v's cluster without inducing a cycle in the coarser hypergraph. Otherwise, we temporarily add u to v's cluster and search the hypergraph for edges violating the second condition in Theorem 4.2.2, then check whether they induce a cycle in the contracted hypergraph. To be more precise, let the toplevel of hypernodes in v's cluster be t and $t+1$. We maintain a queue of hypernodes that are to be processed. Initially, the queue contains all hypernodes in v's cluster with toplevel t. For each hypernode x in the queue, we explore its successors. Whenever we scan a successor y that is in another cluster, we add all hypernodes from y's cluster with toplevel t to the queue. If y is in v's cluster, but x is not, the search found a cycle in the coarsened hypergraph. At this point, we abort the search, remove u from

v's cluster and move on to the next hypernode. If the search does not find a cycle, we leave u in v's cluster and move on. Once the clusters are found, we contract hypernodes in the same cluster pair by pair to obtain a n-level clustering.

4.2.3 Refinement

2-way FM Refinement. On a high level, the 2-way FM refinement algorithm moves hypernodes with the highest gain values between the blocks of a bipartition while making sure to only consider movements that keep the partition acyclic. Over the course of the algorithm, it keeps track of the best bipartition. Once a stopping criterion decides that the refinement is unlikely to find a further improvement of the bipartition, it rollbacks to the best partition found.

More precisely, the algorithm uses two priority queues (one for each block) to keep track of hypernodes and their gain values. Each priority queue contains movable hypernodes in the corresponding block and their gain value, i.e., the change in the connectivity metric when moving the hypernode to the other block. A hypernode is movable if and only if it can be moved to the other block without causing the partition to become cyclic. During 2-way refinement, this is easy to decide: denote the blocks of the bipartition by V_1 and V_2 and assume that V_2 is the successor of V_1 in the quotient graph. Then a hypernode in V_1 can be moved to V_2 if and only if it does not have any successors in V_1. Analogously, a hypernode in V_2 can be moved to V_1 if and only if it does not have any predecessors in V_2. Therefore, it is sufficient to keep track on the number of successors or predecessors that a hypernode has in the same block. We implement this using a simple array that we compute once at the start of the uncoarsening phase and then update it appropriately after every uncontraction operation or movement. In particular, we can use this counter to decide whether new hypernodes become movable (counter becomes zero) or unmovable (counter becomes nonzero) after a movement. We then insert those hypernodes into the appropriate priority queue or remove them.

Initially, the priority queues are empty. After uncontracting a hypernode, the resulting hypernodes and their partners are inserted into the priority queues if they are movable. If none of those are movable, the refinement step is skipped and the next hypernode is uncontracted. Otherwise, the algorithm pulls the hypernode with the highest gain value from the priority queue. The hypernode is only moved to the other block if that does not violate the balance constraint. Nonetheless, the hypernode is marked and therefore excluded from the rest of this refinement round. If the node was moved, all unmarked movable neighbors are inserted into their corresponding priority queue and the gain values of all affected hypernodes are updated.

Once the stopping criteria decides that any further improvement of

the partition if unlikely or both priority queues are empty, the algorithm reverts to the best partition found.

k-way FM Refinement. The k-way FM refinement aims to improve a given k-way partition and is based on the k-way FM refinement algorithm implemented in KaHyPar [2]. The algorithm maintains k priority queues, one queue for each block. Each queue holds hypernodes that can be moved to the block, with the priority being the gain value of the respective move. Viable target blocks for a hypernode depends on the configuration of the refinement algorithm; we consider two alternatives, namely the use of *advanced moves* and *global modes*, described in the following paragraph. We limit the set of movable hypernodes to border hypernodes. The algorithm always performs the best move across all priority queues and after the stopping criterion is reached, the best partition found during the process is restored. Similar to the refinement algorithm of Moreira et al. [42], we consider the following alternatives for viable moves.

Advanced Moves. In this configuration, we only consider hypernode movements that can never create a cycle in the quotient graph. Let $\tau : Q(V) \to [k]$ be a topological order on the quotient graph Q. For a hypernode $u \in V$, let a be the maximum index such that $\tau^{-1}(a)$ contains a predecessor of u and let b be the minimum index such that $\tau^{-1}(b)$ contains a successor of u, i.e., $a = \max\{\tau(b[v]) \mid v \in \Gamma^-(u)\}$ and $\tau(b) = \min\{\tau(b[v]) \mid v \in \Gamma^+(u)\}$. Note that $a \leq \tau(b[u]) \leq b$ since τ is a topological ordering. We then only consider blocks $\tau^{-1}(a), \ldots, \tau^{-1}(b)$ as viable target blocks for u. This ensures that moving u can never create a cycle in the quotient graph. We therefore do not need any cycle detection algorithm. Since we only need to quotient graph to compute the initial topological ordering, we do not need to keep it up-to-date during refinement.

Global Moves. This configuration considers all adjacent blocks as viable target blocks for a hypernode movement. In particular, it computes gain values for movements that might create a cycle in the quotient graph. Therefore, we must use a cycle detection algorithm to prevent those movements. Initially, we considered the use of advanced online cycle detection algorithms [9, 49], but eventually, we settled on Kahn's algorithm [31] since cycle detection did not prove to be a bottleneck in our final implementation. After executing a move, we use the cycle detection algorithm to scan the quotient graph for any cycles. If the move created one, we reverse it, mark the hypernode and remove it from all priority queues. Since the quotient graph might change after every movement, we must keep it up-to-date.

4.3 Larger Imbalances on Coarse Levels

In graph partitioning, temporarily allowing larger imbalances on coarse levels is a well-known technique to improve the resulting partition quality. For instance, Meyerhenke et al. [40] relax the balance constraint on the coarsest level by a constant offset, then narrow it down linearly while unrolling the graph hierarchy until the desired balance constraint is enforced on the finest level of the hierarchy. Previously, Walshaw and Cross [61] formalized this idea and adjusted the allowed imbalance on a given level more carefully, trying to prevent the balance constraint from shrinking too rapidly and therefore losing partition quality. In general, this technique is based on the intuition that a larger imbalance on the coarse levels of the graph hierarchy gives the refinement algorithm more freedom to build partitions that would otherwise be unreachable due to the strict balance constraint. In acyclic DAH partitioning, the partition landscape is even more fractioned than in ordinary graph partitioning due to the additional acyclicity constraints. Hence, we try this technique to further improve the quality of our partitioning algorithm.

Outline. We run the k-way FM refinement algorithm on the coarsest level of the hypergraph hierarchy using a looser imbalance factor $\epsilon' > \epsilon$, producing a partition that violates the ϵ-balance constraint. During the uncoarsening and refinement phase, we then gradually improve the balance of the partition until we end with an ϵ-balanced partition on the finest level. To be more precise, we linearly lower ϵ' to ϵ, i.e., use $\epsilon_i = (\epsilon' - \epsilon)/(n - n_0 + i)$ after the i-th uncontraction operation, where n_0 denotes the number of hierarchies. To improve the balance of the partition, we use a soft rebalancing algorithm that moves hypernodes based on their gain value to improve the balance while not increasing the connectivity metric. If this step is unable to sufficiently improve the balance of the partition, we follow up with a hard rebalancing algorithm that moves hypernodes from overloaded blocks at the cost of increasing the connectivity metric. After balance is restored, we run the k-way FM refinement algorithm from Chapter 4.2.3 initialized with all hypernodes that were touched during the balancing step and the uncontracted hypernodes.

4.3.1 Soft Rebalance

The soft rebalancing algorithm is a version of the FM algorithm that only performs *Advanced Moves*. It accepts a partition if it lowers the imbalance of the partition while not increasing its connectivity.

The priority queues are organized as suggested by Träff [60]. We use one priority queue for each block as well as one priority queue for each hypernode. The queue

corresponding to a hypernode contains an entry for each block that the hypernode can be moved to with its corresponding gain value as priority. The queue corresponding to a block contains all movable hypernodes in that block indexed by their highest possible gain value, i.e., the maximal key in their hypernode priority queue.

The queues are initialized with all movable hypernodes at the start of the uncoarsening and refinement phase. After each uncontraction operation, one round of the algorithm is executed. A round of the soft rebalancing algorithm works as follows. It enables all priority queues that belong to overloaded blocks and pulls the hypernode with the maximal gain across all queues. The target of the move is determined by the maximal element in the hypernode priority queue. If the move reduces the imbalance of the partition, the hypernode is moved to the target block, the priority queues and gains are updated accordingly and the hypernode is inserted into the priority queue of the target block. If the hypernode was not moved, the inviable target block is removed from the hypernode's priority queue and is re-inserted into the block's priority queue.

The round terminates if the improved balance of the partition meets the current imbalance factor or a certain number of moves was performed without producing a viable partition. The algorithm then reverts hypernode movements until the last accepted partition.

4.3.2 Hard Rebalance

Since the algorithm described in the last section is not guaranteed to sufficiently improve the balance of the partition, we also present an algorithm that always succeeds at the cost of a higher connectivity metric. The hard rebalancing algorithm selects an overloaded block and an underloaded block. It then moves nodes along subsequent blocks in the topologically ordered quotient graph from the overloaded block to the underloaded block. The blocks are selected such that the sum of the gain values of all movements is maximal.

More precisely, the algorithm keeps two priority queues for each block, for a total of $2k$ priority queues. One queue contains all hypernodes that can be moved to the previous block while the other one contains all hypernodes movable to the next block. The priorities are the corresponding gain values. Let $\tau : Q(V) \to [k]$ be a topological ordering of the quotient graph Q. Moving hypernodes from one block $\tau^{-1}(i)$ to block $\tau^{-1}(j)$ with $i < j$ involves moving one hypernode from block $\tau^{-1}(i)$ to block $\tau^{-1}(i+1)$, one from $\tau^{-1}(i+1)$ to $\tau^{-1}(i+2)$ and so on. We therefore select a pair of an overloaded and an underloaded block such that the sum of the gain values of all of those moves is maximal among all possible (overloaded, underloaded) pairs of blocks. Note that this is not a precise approach to find the best pair of blocks, since moves might change the gain values of subsequent moves, although

our experiments indicate that it is a good approximation in practice. After moving a hypernode, we insert it into the priority queue of its new block (if it is still movable) and update gain values of adjacent hypernodes.

This algorithm can always restore the balance of a DAH partition. To see this, we need Lemma 4.3.1.

Lemma 4.3.1. *Let H be a DAH with an acyclic k-way partition Π, and topological ordering $\tau : \Pi \to [k]$ of the quotient graph. Then every block $\tau^{-1}(i)$, $1 < i \leq k$ has at least one hypernode movable to block $\tau^{-1}(i-1)$ and every block $\tau^{-1}(j)$, $1 \leq j < k$ has at least one hypernode movable to block $\tau^{-1}(j+1)$.*

Proof. Recall that a H can be seen as a DAG G by replacing each hyperedge e with a directed bipartite graph from e^T to e^H. This transformed graph is acyclic if and only if the DAH is acyclic. Hence, we can prove the statement for DAGs rather than DAHs. Let $1 < i \leq k$. Consider the block-induced subgraph $G' \coloneqq G[\tau^{-1}(i)]$. Since G is a DAG, so is G' and we can use Lemma 2.1.1 to see that G' contains a node $u \in \tau^{-1}(i)$ with indegree zero in G'. Moving this node to block $\tau^{-1}(i-1)$ does not create a backward edge in the quotient graph with respect to τ, proving the first statement of the lemma. The second statement follows analogously. □

Moreover, observe that the topological ordering of the k-way partition does not change during the coarse of the algorithms: Since we only move hypernodes between subsequent blocks, the moves can only create new quotient graph edges between subsequent blocks. Hence, since the topological ordering is static during the course of the algorithm, and since we always have movable hypernodes, the algorithm can always perform movements until the desired balance is reached.

However, this is only true when working on a DAH. If the hypergraph contains cycles, it might fail when a block has no movable hypernodes. While our input hypergraphs are always acyclic, one of the coarsening algorithms described in Chapter 4.2.2 might produce cyclic coarser hypergraphs. In this case, we simply stop the rebalancing step and try again after uncontracting the next hypernode. Since the finest hypergraph is acyclic, the algorithm succeeds eventually.

4.4 Memetic Acyclic Hypergraph Partitioning

In Chapter 3.4, we referenced memetic algorithms as a successful metaheuristic for high-quality graph and hypergraph partitioning. Now, we present a memetic algorithm for the DAH partitioning problem. We use the memetic algorithm already implemented in KaHyPar [4] as framework and exchange its building blocks for mutation, recombination and generating the initial partition with new

algorithms. From a metaheuristic point of view, the algorithm remains unchanged. Hence, we follow the description from Ref. [4] closely.

Population. The memetic algorithm starts by generating the initial population $\mathcal{P}$. The population consists of individuals, which are always ϵ-balanced k-way partitions of the input hypergraphs. We generate the initial individuals using our multilevel algorithm introduced in previous chapters. The size of $\mathcal{P}$ is choosing dynamically by measuring the time t_I it takes to generate one individuals, i.e., the running time of our multilevel algorithm: $|\mathcal{P}| \coloneqq \max(3, \min(50, \delta \cdot (t/t_I)))$, where δ is a configuration parameter that we set to 0.15.

The fitness of an individual is its connectivity, since that is the objective that we want to optimize. An individual with lower connectivity is fitter than one with higher connectivity. The initial population is evolved over several generational cycles using the *steady-state* paradigm [20]: We generate only a single offspring per generation. When inserting a new individual I_1 into the population, we evict an old one I_2 based on similarity: the difference between both individuals is defined as $d(I_1, I_2) \coloneqq |D(I_1) \ominus D(I_2)|$, where $D(I)$ is a multi-set that contains each hyperedge $e \in E$ exactly $\lambda(e) - 1$ times and $\ominus$ is the symmetric difference. In other words, we consider two individuals to be somewhat similar if all hyperedges have roughly the same connectivity in both partitions.

Recombination Operator. For recombination, we select parents using binary tournament selection [41]: First, we select two individuals at random and choose the fitter one as first parent P_1. Then, we repeat the process to select the second parent P_2. We then run a modified V-cycle to combine both parents. During coarsening, we only allow the contraction of two hypernodes u and v if they are in the same block in both P_1 and P_2, i.e., if $b_1[u] = b_1[v]$ and $b_2[u] = b_2[v]$. This allows us to use P_1 or P_2 as initial partition once coarsening has terminated. We apply the fitter one of both parents as initial partition. Finally, we unroll the graph hierarchy and improve the partition using our k-way refinement algorithm. Note that this recombination operator produces offsprings that are at least as good as the better of both parents.

Mutation Operations. We implement two mutation operations. The first one starts by selecting a random individual I, then perform a modified V-cycle that works as follows. During coarsening, it only contracts pairs of hypernodes u and v that are in the same block in I, i.e., $b[u] = b[v]$. On the coarsest level of the graph hierarchy, I is used as initial partition and the graph hierarchy is unrolled and improved using our k-way refinement algorithm. This operation produces an offspring that is at least as good as I. The second mutation operation also selects a random individual I_1, then generates a new individual I_2

as described earlier. Next, both individuals get recombined, but we always use I_2 as initial partition. This operator can therefore produce offsprings that are worse than the individual selected from the population.

5 Experimental Evaluation

In this chapter, we evaluate the performance of our algorithm and the influence of various components on solution quality. We start by presenting our methodology, the systems used for evaluation and the benchmark setup. We then present our main results by comparing our algorithm to previous DAG partitioning approaches in Chapter 5.1. For these experiments, we only use DAG instances, since the other algorithm cannot cope with hypergraphs. Afterwards, we compare our memetic multilevel algorithm to simpler approaches for DAH partitioning in Chapter 5.2 and evaluate the impact of DAH partitioning on an image streaming application in Chapter 5.3. Finally, we evaluate the influence of the acyclic coarsening algorithm and present partitioning approaches that did not prove to be beneficial in Chapter 5.2.2 and Chapter 5.2.1, respectively.

Methodology and Setup. We implement all algorithms described in Chapter 4 in the KaHyPar hypergraph partitioning framework [2, 4, 27, 28, 54, 55]. We use `mlDHGP` to refer to our multilevel algorithm and `memDHGP` for our memetic multilevel algorithm. The code is written in C++ and compiled using g++ 9.1 using `-O3 -march=native` as compile flags. Our implementation is based on the KaHyPar version from march 2019[1]. The source code of our preliminary version is available at `github.com/danielseemaier/kahypar/tree/HyperDAG`, and we plan to integrate our algorithm into the next release of KaHyPar available at `kahypar.org`.

We ran our experiments on two different machines. Machine A is a single node from the HPC cluster bwUniCluster equipped with two Intel Xeon E5-2670 Octa-Core (Sandy Bridge) processor clocked at 2.6 GHz, 64 GB main memory, 20 MB L3-Cache and 8x256 KB L2-Cache. Machine B has two Intel Xeon E5-2650 v2 Octa-Core (Sandy Bridge) clocked at 2.6 GHz, 128 GB main memory, 20 MB L3-Cache and 8x256 KB L2-Cache.

Performance Profiles. We use performance profiles [22] to compare the solution quality of different algorithms. The plots show one curve for each algorithm included in the comparison. The x-axis shows $\tau \in [1, 100]$ and the y-axis shows the fraction

[1] Commit hash `84c7e7c523701efb0e51752053656e34d206cf4c`, repository `github.com/SebastianSchlag/kahypar`.

of instances for which each algorithm computed a partition that is within a factor of τ of the best partition computed by any algorithm for that instance. In particular, $\tau = 1$ shows the fraction of instances for which each algorithm computed the best partition (i.e., partition with the lowest connectivity). A value of 0.8 on the y-axis for $\tau = 1$ shows that the algorithm computed the best result on 80% of all instances and a value of 1 for $\tau = 1.1$ reveals that the algorithm computes partitions within a factor of 1.1 of the partition of the best algorithm on *every* instance.

Instances. We use three sets of graphs for our experimental evaluation. The first set of graphs are deduced from the PolyBench Benchmark Suite [51]. Those graphs were kindly provided to us in DAG format by Herrmann et al. and were also used for evaluation in previous papers on DAG partitioning [25, 26, 43]. Next, we use graphs from the ISPD98 Circuit Benchmark Suite [3]. These instances contain one node for each cell and a directed edge from the source of a net to each of its sinks. In case the resulting instance does not form a DAG, i.e., contains cycles, we do the following: We gradually add directed edges and skip those that would create a cycle. We use these instances for our main experiments. Some of our initial experiments use DAGs based on the circuits from the ISCAS85 Combinational Benchmark Circuits [13]. Those DAGs contain one node for each logic gate and a directed edge from the output of a logic gate to the input of another logic gate. Basic properties of these instances are provided in Table 5.1.

To conduct experiments on hypergraphs, we transform all DAGs into DAHs using the row-net model on their adjacency matrices: A hypergraph contains one hypernode for each node in the DAG and a hyperedge for each node u with outgoing edges. The head of the hyperedge is u and the tails are the successors of u.

When evaluating the impact of DAH partitioning on an image streaming application, we use DAGs modeling the data flow of an advanced imaging algorithm [48] and transform them into DAHs. This transformation works differently from the previous description and is described in Chapter 5.3.

5.1 DAG Model

We start our experimental evaluation by comparing our algorithm to the current state-of-the art on DAG partitioning. Note that the comparison is possible even though previous algorithms optimize the edge-cut on DAGs, whereas our algorithm optimizes the connectivity metric on DAHs since both objectives are equal for 2-uniform hypergraphs. The comparisons includes the algorithm by Herrmann et al. [25] and the one by Moreira et al. [43]. We name the former algorithm `HOUKC` (first letters of the authors last names), the latter one `Moreira` (last name of the first

Graph	n	m	Ref.	Graph	n	m	Ref.
PolyBench				ISPD98			
2mm	36 500	62 200	[51]	ibm01	13 865	42 767	[3]
3mm	111 900	214 600	[51]	ibm02	19 325	61 756	[3]
adi	596 695	1 059 590	[51]	ibm03	27 118	96 152	[3]
atax	241 730	385 960	[51]	ibm04	31 683	108 311	[3]
covariance	191 600	368 775	[51]	ibm05	27 777	91 478	[3]
doitgen	123 400	237 000	[51]	ibm06	34 660	97 180	[3]
durbin	126 246	250 993	[51]	ibm07	47 830	146 513	[3]
fdtd-2d	256 479	436 580	[51]	ibm08	50 227	265 392	[3]
gemm	1 026 800	1 684 200	[51]	ibm09	60 617	206 291	[3]
gemver	159 480	259 440	[51]	ibm10	74 452	299 396	[3]
gesummv	376 000	500 500	[51]	ibm11	81 048	258 875	[3]
heat-3d	308 480	491 520	[51]	ibm12	76 603	392 451	[3]
jacobi-1d	239 202	398 000	[51]	ibm13	99 176	390 710	[3]
jacobi-2d	157 808	282 240	[51]	ibm14	152 255	480 274	[3]
lu	344 520	676 240	[51]	ibm15	186 225	724 485	[3]
ludcmp	357 320	701 680	[51]	ibm16	189 544	648 331	[3]
mvt	200 800	320 000	[51]	ibm17	188 838	660 960	[3]
seidel-2d	261 520	490 960	[51]	ibm18	201 648	597 983	[3]
symm	254 020	440 400	[51]				
syr2k	111 000	180 900	[51]				
syrk	594 480	975 240	[51]				
trisolv	240 600	320 000	[51]				
trmm	294 570	571 200	[51]				
ISCAS85							
c432	196	336	[13]	c2670	1 426	2 075	[13]
c499	243	408	[13]	c3540	1 719	2 936	[13]
c880	443	729	[13]	c5315	2 485	4 386	[13]
c1355	587	1 064	[13]	c6288	2 448	4 800	[13]
c1908	913	1 497	[13]	c7552	3 719	6 144	[13]

Table 5.1: Basic properties of DAG instances.

author), our own multilevel algorithm `mlDHGP+X`, and our own memetic algorithm `memDHGP+X`, where `X` $\in \{$`KaHyPar`, `PaToH`$\}$ is the undirected hypergraph partitioner used during initial partitioning. Whenever we omit `+X`, we default to `KaHyPar` since it showed the best results. When using `+KaHyPar`, we invoke KaHyPar using the newest configuration available[2] and when using `+PaToH`, we invoke it using the default configuration. To run the `HOUKC` algorithm on our benchmark set, we use the implementation publicly available on the first author's website[3]. Since the author's implementation of `Moreira` belongs to Intel and is therefore not publicly available, we refrain from running it on our benchmark set and opt to use the numbers provided in Ref. [43]. We have therefore no data for this algorithm on the ISPD98 instances, but can still compare it to the other algorithms on the PolyBench instances.

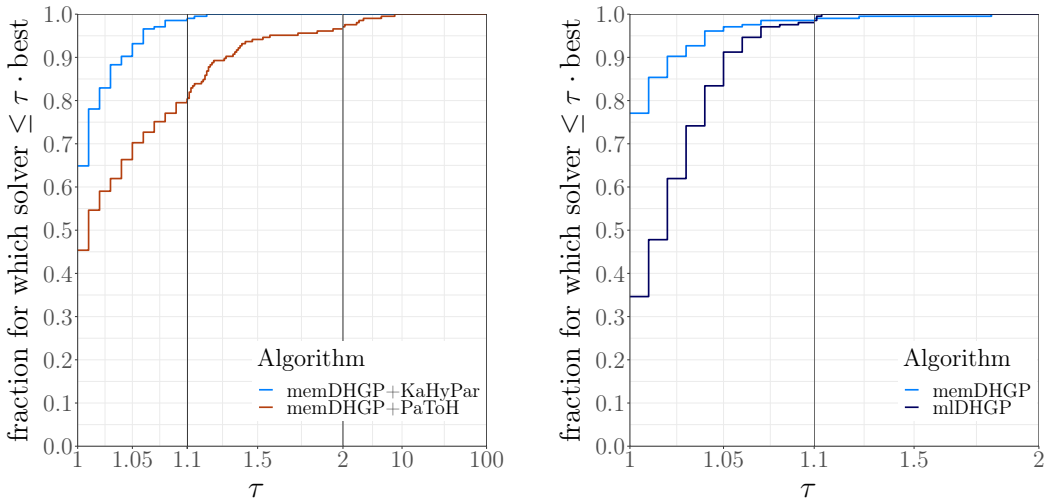
We partition each graph in our benchmark set 5 times for different values of $k \in \{2, 4, 8, 16, 32\}$ and use $\epsilon = 3\%$ as imbalance parameter. These values were chosen because they were also used in previous work on DAG partitioning [25, 43]. Each algorithm gets allocated 8 hours (except for `adi`, for which we give all solvers 24 hours) for each instance of our benchmark set. We run them on a single core of Machine A. While the memetic algorithms take the time limit as an input parameter, non-memetic algorithms do not. We therefore run them repeatedly using random seeds until the time limit exceeds.

The effect of the undirected hypergraph partitioner used during initial partitioning and of our memetic component is summarized in Figure 5.1. As we can see in Figure 5.1a, using a state-of-the undirected hypergraph partitioner pays off: With KaHyPar, our algorithm computes the same or a better solution on 65% out of all instances (PolyBench and ISPD98), whereas the same is only true for 45% out of all instances when using it with PaToH. Hence, we will only consider `memDHGP+KaHyPar` and `mlDHGP+KaHyPar` in the rest of the experiments. Moreover, Figure 5.1b summarizes the influence of our memetic component. We observe the `memDHGP` computes a strictly better partition on 65% out of all instances compared to repeated runs using random seeds (i.e., `mlDHGP`) with the largest improvement made on `2mm` with $k = 8$ (10%) and `ibm01` with $k = 2$ (8%). Based on these observations, we conclude that our memetic algorithm is more effective than repeated restarts of our multilevel algorithm.

Next, we compare our algorithm to `HOUKC` and `Moreira`. The best edge cuts found by each algorithm are compared in Figure 5.2. Detailed results with per- instance edge cuts are available in Tables A.1–A.4. We observe that `memDHGP+KaHyPar` outperforms the other algorithms on both the PolyBench and ISPD98 instances. Looking at Figure 5.2a, we see that it computes the best partition on over 82% of

[2] `github.com/SebastianSchlag/kahypar/blob/master/config/km1_kahypar_mf_jea19.ini`

[3] `people.bordeaux.inria.fr/julien.herrmann`



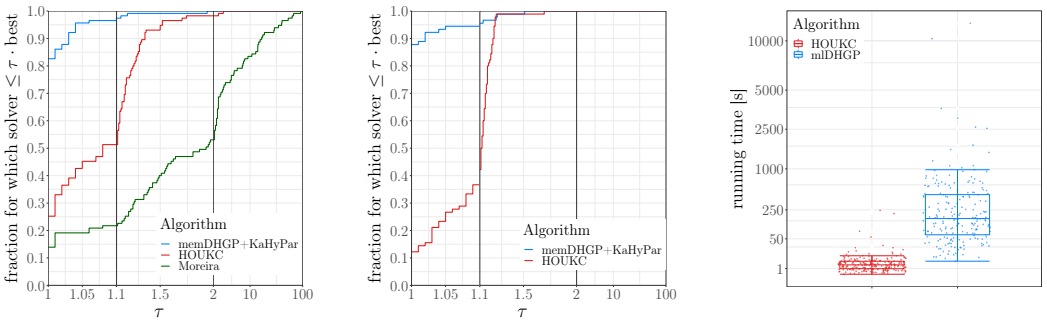
(a) Influence of initial partitioning algorithm on PolyBench and ISPD98 DAGs.

(b) Influence of memetic algorithm on PolyBench and ISPD98 DAGs.

Figure 5.1: Influence of the algorithm used for undirected initial partitioning and of our memetic algorithm on PolyBench and ISPD98 DAGs.

all PolyBench instances, while `HOUKC` and `Moreira` only compute the best partition on 25% and 14%, respectively. Moreover, `memDHGP+KaHyPar` is within a factor of 1.1 of the best algorithm on over 95%, whereas the other algorithms are only within a factor of 1.1 of the best algorithm on 51% and 22%, respectively. The ISPD98 instances reveal a similar observation: Here, `memDHGP+KaHyPar` computes the best solution on over 87% instances compared to `HOUKC`. The average improved to the previous state-of-the art `HOUKC` is 11.1% (PolyBench instances) and 9.7% (ISPD98 instances). The best improved is observed on the graph `covariance` from the PolyBench instances with $k = 2$: While `HOUKC` computed an edge cut of 34 307, our algorithm finds a partition with an edge cut of 11 281.

We finish this chapter by taking a look at the running times of `mlDHGP+KaHyPar` and `HOUKC` in Figure 5.2c. Note that we exclude `Moreira` from this comparison since we did not run their code ourself and the authors did not report per-instance running times for their algorithm. As can be seen in the figure, `mlDHGP+KaHyPar` is slower than `HOUKC` by several orders of magnitude. This has several reasons: `HOUKC` uses METIS [33] during initial partition rather than `KaHyPar`, only uses 2-way refinement, uses a multilevel scheme with only $\log(n)$ levels rather than n levels, and only supports DAGs and no DAHs. Computing and updating gain values is faster when optimizing the edge cut objective on graphs than the connectivity objective on hypergraph. In particular, due to the acyclicity constraint, edge cut gain values



(a) `memDHGP`, `HOUKC` and `Moreira` on PolyBench DAGs.

(b) `memDHGP` and `HOUKC` partitioner on ISPD98 DAGs.

(c) Running time of `HOUKC` and `mlDHGP` on PolyBench and ISPD98 instances.

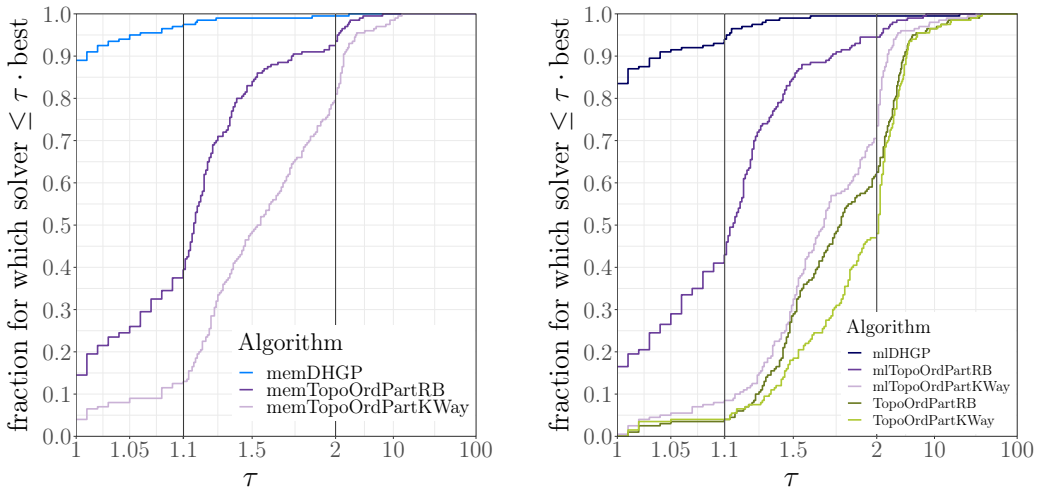
Figure 5.2: Performance and running time of DAG partitioners on PolyBench and ISPD98 DAGs.

never change when moving nodes between the blocks of a bipartition of a normal graph. Hence, `HOUKC` does not need to update gain values during 2-way refinement.

5.2 DAH Model

After establishing the state-of-the art for DAG partitioning, we move on the DAH instances. For these experiments, we use the PolyBench and ISPD98 instances transformed into DAHs as described above, but exclude the `adi` graph from the PolyBench instances since it is too large to be partitioned by our algorithm within a reasonable time frame. We use the same values for k and ϵ as before. Since we are not aware of any other algorithms for DAH partitioning, we compare the performance of our algorithm to simpler heuristics, namely `TopoOrdPartRB` and `TopoOrdPartKWay`. The former one partitions the graph using recursive bisection. For each bisection, it uses Kahn's algorithm to compute a topological order of the graph, splits the graph based on that, and then uses our 2-way refinement algorithm to improve the bisection. The latter one again uses Kahn's algorithm to compute a topological order, but then directly splits the graph into k blocks and improves it using our k-way refinement algorithm. We also test these heuristics with multilevel refinement and name those algorithms `mlTopoOrdPartRB` and `mlTopoOrdPartKWay`. Finally, we use those partitions as input to our memetic algorithm and name the result `memTopoOrdPartRB` and `memTopoOrdPartKWay`. As before, we give the memetic algorithm 8 hours time and use Machine A to perform these experiments.

The results are shown in Figure 5.3 and Table 5.2 with detailed per-instance



(a) Memetic algorithms on PolyBench and ISPD98 DAHs.

(b) Multilevel and singlelevel algorithms on PolyBench and ISPD98 DAHs.

Figure 5.3: Comparison of `memDHGP` and `mlDHGP` to simpler approaches.

results available in Tables B.1–B.4. Looking at the performance profile shown in Figure 5.3a, we observe that `memDHGP` outperforms the simpler heuristics, computing the best result on almost 90% out of all instances, whereas the second best approach, `memTopoOrdPartRB`, only computes the best result on less than 15% out of all instances. On average, `memDHGP` computes partitions with 20% lower connectivity than `memTopoOrderPartRB`. We can therefore conclude that using a high quality undirected hypergraph partitioner during initial partitioning improves the overall result significantly.

Focusing on Figure 5.3b and Table 5.2, we see that using recursive bisection gives significantly better results than using direct k-way partitioning. Looking at the single-level algorithms `TopoOrderPartRB` and `TopoOrderPartKWay`, the approach using recursive bisection is 10% better on average than direct k-way. This advantage increases when looking at the multilevel algorithms: Here, recursive bisection improves partitions by 49%. We believe that this is due to the fact that the k-way search space is much more fractured than the 2-way search space due to the acyclicity constraint. Moreover, we observe that the multilevel algorithms are better than the single-level algorithms; this is expected as the multilevel component adds a more global view to the optimization landscape.

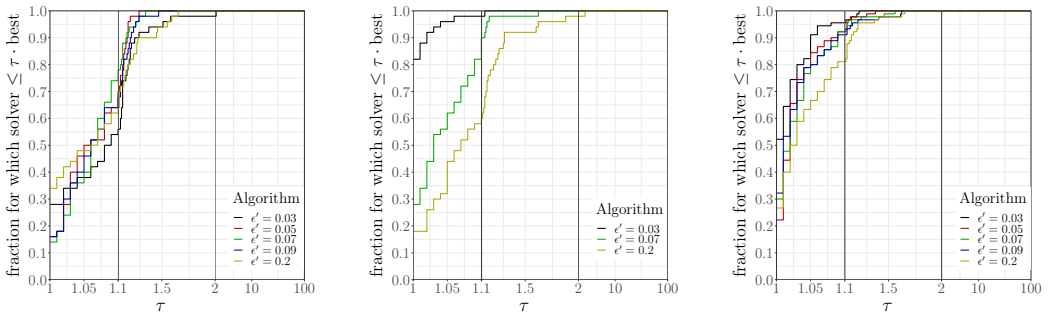
Algorithm	gmean
`TopoOrderPartRB`	16 571
`mlTopoOrderPartRB`	9 128
`memTopoOrderPartRB` (8h)	8 071
`TopoOrderPartKWay`	18 161
`mlTopoOrderPartKWay`	13 605
`memTopoOrderPartKWay` (8h)	10 643
`mlDHGP`	7 244
`memDHGP` (8h)	6 526

Table 5.2: Geometric mean solution quality of the best results out of multiple repetitions for different algorithms on PolyBench and ISPD98 instances (as DAHs).

5.2.1 Influence of Larger Imbalances on Coarse Levels

Next, we evaluate the influence of allowing a larger imbalance on coarse levels. This algorithm is described in Chapter 4.3. We executed the experiment on Machine B and used the ISCAS85 and PolyBench DAHs with $k = 2, 4, 8, 16, 32$, $\epsilon = 3\% = 0.03$ (maximum imbalance of the resulting partition) and various values for ϵ' (imbalance on the coarsest level of the hypergraph hierarchy).

The result of this experiments is summarized in Figure 5.4. In our initial experiment, we obtained an initial k-way partition using a topological ordering of the DAH, i.e., using `mlTopoOrdKWay` from the previous chapter. We then ran a V-cycle with larger imbalance on coarse levels as described in Chapter 4.3. The result of this run can be seen in Figure 5.4a. We observe that using $\epsilon' = 20\%$ computed the best partition on more than on third out of all instances, whereas the configuration that does not allow a larger imbalance on coarse levels ($\epsilon' = \epsilon = 3\%$) only computes the best partition on 28% out of all instances. Moreover, when using $\epsilon' = 7\%$, the computed partition is within a factor of 1.1 of the best partition on 74% out of all instances, whereas the same can only be said for 54% out of all instances when not allowing a larger imbalance on coarse levels. Based on this, it seems that this techniques can improve a partition computed using `mlTopoOrdKWay` on ISCAS85 instances. However, as we have seen in the previous chapter, `mlTopoOrdKWay` computes partitions of low quality while `mlDHGP` computes much better partitions. Hence, we repeated the same experiment with `mlDHGP` instead of `mlTopoOrdKWay`. With this configuration, we see in Figure 5.4b that $\epsilon' \in \{7\%, 20\%\}$ no longer compute better partitions, but worsen the overall result. The same observation applies when partitioning the PolyBench instances, as can be seen in Figure 5.4c. Here, we see that not allowing larger imbalances on coarse



(a) ISCAS85 partitioned using `mlTopoOrdKWay`.

(b) ISCAS85 partitioned using `mlDHGP`.

(c) PolyBench instances partitioned using `mlDHGP`.

Figure 5.4: Influence of larger imbalance on coarse levels for ISCAS85 and PolyBench DAHs and various values for ϵ'.

levels computes the best partition on most instances and generally outperforms all other configurations, and partition quality decreases with increasing values of ϵ'.

In our experiments, we observed that almost all balance improvements stem from our hard rebalancing algorithm that decreases partition quality. This decrease in partition quality outweighs the improvement found during the initial k-way refinement with larger imbalance on the coarsest level and all improvements found during uncoarsening. We think that this is due to the observation that k-way refinement and our soft rebalancing algorithm are too restricted by the acyclicity constraint.

5.2.2 Influence of Acyclic Coarsening

In this experiment, we evaluate the influence of our acyclic coarsening algorithm presented in Chapter 4.2.2 by comparing it to the coarsening algorithm that is already implemented in KaHyPar [2]. We used Machine B to execute this experiment. The instances are ISPD98 DAHs that were partitioned into $k = 2, 4, 8, 16, 32$ blocks with maximum imbalance $\epsilon = 3\%$.

The average connectivity using the acyclic coarsening is 16 086, whereas the coarsening algorithm already implemented in KaHyPar yields an average connectivity of only 20 915. Hence, the acyclic coarsening algorithm produces partitions with 23% lower connectivity on average. This improvement can also be seen in Figure 5.5: With acyclic coarser hypergraphs, we compute strictly better partitions on almost all instances. Using the coarsening algorithm that is already implemented in KaHyPar, we only get within a factor of 1.1 of the partition computed using the acyclic coarsening algorithm on 10% out of all instances. We believe that this is due to the fact that hypernodes in a cycle cannot be moved to another block

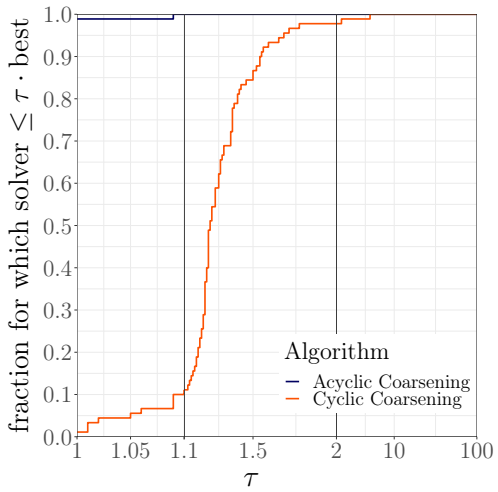


Figure 5.5: ISPD98 instances partitioned using the acyclic coarsening algorithm and the one that is already implemented in KaHyPar and does not keep the hypergraph acyclic.

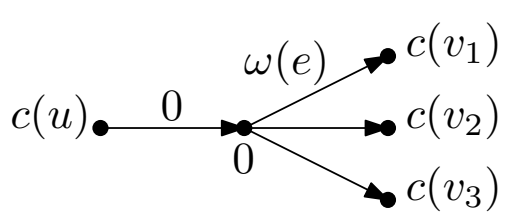
without violating the acyclicity constraint, therefore restricting the refinement algorithm. Judging from this experiment, we conclude that acyclic coarsening algorithms are necessary to obtain high-quality acyclic DAH partitions. This was already observed by Herrmann et al. [25] for the DAG case.

5.3 Impact on Streaming Application

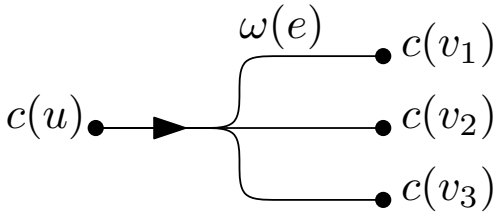
The experiment described in this chapter was conducted by Merten Popp. The author of this thesis only implemented the transformation of DAGs into DAHs.

To evaluate the impact of our DAH partitioning algorithm on image streaming applications, we integrate the algorithm into a toolchain implementing the *Local Laplacian filter* that was also used to evaluate earlier work on DAG partitioning [42, 43]. The filter is an edge-aware image processing filter using concepts of *Gaussian pyramids* and *Laplacian pyramids* as well as a point-wise remapping function to enhance image details without creating artefacts. For more details on the image processing algorithm see Ref. [48].

The toolchain implementing the algorithm invokes our algorithm on a DAG structured as depicted in Figure 5.6a. The DAG consists of *compute nodes* and *output nodes*. Compute nodes correspond to parts of the filter application and have the size of the program as node weight. Output nodes are merely conceptual and have node weight zero. Each compute node can have an arbitrary number of output nodes as successors; the edge from compute node to output node has



(a) DAG model: Nodes with nonzero weight are *compute nodes*, nodes with zero weight are *output nodes*. One compute node can have multiple output nodes. The weight of edges from output nodes to compute nodes correspond to the amount of data that has to be transferred. Note that all outgoing edges from an output node have the same weight.

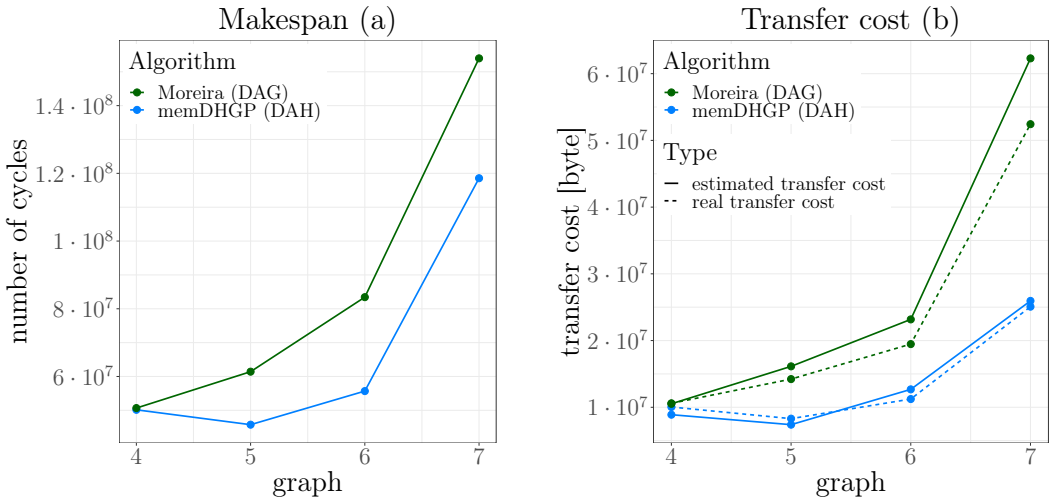


(b) DAH model: Each output node is replaced by a single hyperedge that models the data transfer.

Figure 5.6: Structure of the data dependency and execution flow graphs, once as DAG (Figure 5.6a) and once as DAH (Figure 5.6b). The DAGs are kindly provided to us by Moreira et al. and we perform the transformation to the corresponding DAHs ourself.

edge weight zero. The successors of output nodes are other compute nodes, which depend on the output of the preceding compute node. All outgoing edges from an output node have the same edge weight, namely the size of the compute node's output. To benefit from DAH partitioning, we transform the supplied DAG into a DAH as depicted in Figure 5.6b. Each output node is replaced with a single hyperedge that has the same hyperedge weight as its outgoing edges. The head of the output hyperedge is the preceding compute node and its tails are the succeeding compute nodes. This DAH is then partitioned using `memDHGP` with a time limit of 10 minutes and the final partition is projected back onto the original DAG by placing output nodes into the same block as its preceding compute node.

The filtering algorithm is configurable by parameter K, the number of levels that the image pyramids have. A higher value for K improves the filters result, but also increases the size of the DAG (DAH). To evaluate our algorithm, we use values $K \in \{4, 5, 6, 7\}$, resulting in a DAG with at most $n = 752$ and $m = 862$ for $K = 7$. We compare the impact of DAH partitioning to `Moreira`, the algorithm previously used in the toolchain. The result is illustrated by Figure 5.7. By looking at Figure 5.7a, we observe that DAH partitioning generally results in a lower makespan compared to DAG partitioning. The best improvement is a reduction in makespan by 22%, observed on the largest filtering algorithm, i.e., $K = 7$. Figure 5.7b hints that this improvement is in part due to a more accurate modeling of transfer costs. The figure compares the edge cuts reported by the respective



(a) Makespan of each filtering algorithm. The values are obtained using a cycle-true compiled simulator of the target platform. Lower is better.

(b) Edge cut as reported by the respective algorithm (i.e., estimated data transfer) versus actual data transfer. Closer gap is better.

Figure 5.7: Impact of the DAH model with improved partitioning results on image streaming applications: Lower makespan and more accurate modeling of the transfer cost.

algorithms (i.e., the estimated transfer cost) to the actual transfer costs. We observe that the DAH model indeed estimates the transfer cost much better than the DAG model, confirming our initial hypothesis. Based on this, we conclude that the DAH model is better suited than the DAG model in this application domain.

6 Conclusion

Motivated by the shortcomings of the DAG model for representing dataflow and execution dependencies in image streaming applications, we presented the first hypergraph partitioning algorithm for directed acyclic hypergraphs that can cope with the acyclicity constraint. We implemented our novel algorithm using the hypergraph partitioning framework KaHyPar and ran extensive experiments to benchmark its components and to compare it to the previous state-of-the art on DAG partitioning.

Indicated by our experimental evaluation, we observed that our n-level partitioning algorithm improved on the current state-of-the-art algorithm for DAG partitioning: Compared to previous algorithms, we computed the best partitions on 82% out of all instances in our benchmark set and improved the edge cut by 10% on average. Based on this, we concluded that our algorithm can be seen as the new state-of-the art for DAG partitioning in terms of partitioning quality. Since there are no previous algorithms for DAH partitioning, we compared our algorithms to more simple heuristics and showed significant improvements. Compared to direct k-way partitioning based on a topological ordering with single-level FM refinement, we showed that our memetic algorithm could lower the connectivity of the resulting partition by 64% on average. Getting back to the problem that motivated our work, we showed that acyclic DAH partitioning allows a significantly more efficient parallelization of image streaming applications on embedded devices, improving the makespan of an advanced image filter by 22%. We could therefore conclude that our algorithm outperforms previous DAG partitioner that were used in the application domain.

6.1 Future Work

We have left several paths for future extensions and improvements. First, our current implementation can only handle DAHs where every hyperedge has at most one head. This limitation could be lifted with changes to the coarsening or refinement step. Secondly, since k-way local search is a bottleneck of our algorithm, future work could work towards parallelized refinement algorithms. Lastly, we plan to integrate our algorithm into the next KaHyPar release.

Bibliography

[1] Nitin Ahuja, Matthias Bender, Peter Sanders, Christian Schulz, and Andreas Wagner. Incorporating road networks into territory design. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '15, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct k-way hypergraph partitioning algorithm. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 28–42.

[3] Charles J. Alpert. The ispd98 circuit benchmark suite. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, pages 80–85, New York, NY, USA, 1998. ACM.

[4] Robin Andre, Sebastian Schlag, and Christian Schulz. Memetic multilevel hypergraph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 347–354, New York, NY, USA, 2018. ACM.

[5] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 120–124, New York, NY, USA, 2004. Association for Computing Machinery.

[6] Giorgio Ausiello, Paolo G. Franciosa, and Daniele Frigioni. Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. In *Theoretical Computer Science*, pages 312–328, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[7] Giorgio Ausiello and Luigi Laura. Directed hypergraphs: Introduction and fundamental algorithms—a survey. *Theoretical Computer Science*, 658:293 – 306, 2017. Horn formulas, directed hypergraphs, lattices and closure systems: related formalism and application.

[8] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag London, 2nd edition, 2009.

[9] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan.

A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2), December 2015.

[10] Claude Berge and Edward Minieka. *Graphs and Hypergraphs*. North-Holland mathematical library. North-Holland Publishing Company, 1976.

[11] Sonja Biedermann, Monika Henzinger, Christian Schulz, and Bernhard Schuster. Memetic Graph Clustering. In *17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:15, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In *Euro-Par 2013 Parallel Processing*, pages 659–670, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[13] Franc Brgles and Hideo Fujiwara. A neutral netlist of 10 combinational circuits and a target translator in fortran. *IEEE International Symposium on Circuits and Systems*, 1985.

[14] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering - Selected Results and Surveys*, pages 117–158. Springer, 2016.

[15] Umit V. Catalyürek and Cevdet Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.

[16] Jason Cong and Yuzheng Ding. On area/depth trade-off in lut-based fpga technology mapping. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, page 213–218, New York, NY, USA, 1993. Association for Computing Machinery.

[17] Jason Cong and Sung Kyu Lim. Performance driven multiway partitioning. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ASP-DAC '00, page 441–446, New York, NY, USA, 2000. Association for Computing Machinery.

[18] Jason Cong and Sung Kyu Lim. Edge separability based circuit clustering with application to circuit partitioning. In *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pages 429–434, Jan 2000.

[19] Jason Cong, Zheng Li, and Rajive Bagrodia. Acyclic multi-way partitioning of boolean networks. In *31st Design Automation Conference*, pages 670–675, June 1994.

[20] Kenneth A. De Jong. *Evolutionary computation - a unified approach*. MIT Press, 2006.

[21] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit. V. Catalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Conference on Parallel and Distributed Processing (IPDPS)*, pages 124–124. IEEE, 2006.

[22] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[23] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[24] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[25] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Umit Çatalyürek. Acyclic partitioning of large directed acyclic graphs. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 371–380, May 2017.

[26] Julien Herrmann, Yusuf Özkaya, Bora Uçar, Kamer Kaya, and Umit Çatalyürek. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM Journal on Scientific Computing*, 41(4):A2117–A2145, 2019.

[27] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network flow-based refinement for multilevel hypergraph partitioning. *ACM Journal of Experimental Algorithmics (JEA)*.

[28] Tobias Heuer and Sebastian Schlag. Improving coarsening schemes for hypergraph partitioning by exploiting community structure. In *16th International Symposium on Experimental Algorithms (SEA 2017), London, UK, 21th - 23rd June 2017. Ed.: C. Iliopoulos*, volume 75 of *LIPIcs - Leibniz International Proceedings in Informatics*, page Art. Nr. 21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Wadern, 2017. 46.12.02; LK 01.

[29] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a Scalable High Quality Graph Partitioner. *Proceedings of the 24th IEEE International Parallal and Distributed Processing Symposium*, pages 1–12, 2010.

[30] Laurent Hyafil and Ronald Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. IRIA-Laboria, Rocquencourt, France, 1973.

[31] Arthur B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.

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

[33] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[34] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, January 1998.

[35] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, January 1998.

[36] Hye-Jin Kim and Yong-Hyuk Kim. Recent progress on graph partitioning problems using evolutionary computation. *CoRR*, abs/1805.01623, 2018.

[37] Fatih Kocan and Mehmet H. Gunes. Acyclic circuit partitioning for path delay fault emulation. In *The 3rd ACS/IEEE International Conference on Computer Systems and Applications, 2005.*, page 22, Jan 2005.

[38] Lingda Li, Robel Geda, Ari B. Hayes, Yanhao Chen, Pranav Chaudhari, Eddy Z. Zhang, and Mario Szegedy. A simple yet effective balanced edge partition model for parallel computing. *SIGMETRICS Perform. Eval. Rev.*, 45(1):6, June 2017.

[39] Henning Meyerhenke, Martin Nollenburg, and Christian Schulz. Drawing large graphs by multilevel maxent-stress optimization. *IEEE transactions on visualization and computer graphics*, 24(5):1814–1827, 2018.

[40] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering. In *Experimental Algorithms*, pages 351–363, Cham, 2014. Springer International Publishing.

[41] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9, 1995.

[42] Orlando Moreira, Merten Popp, and Christian Schulz. Graph partitioning with acyclicity constraints. In *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, pages 30:1–30:15, 2017.

[43] Orlando Moreira, Merten Popp, and Christian Schulz. Evolutionary multi-level acyclic graph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018*, pages 332–339, 2018.

[44] Pablo Moscato. On evolution, search, optimization, genetic algorithms and martial arts - towards memetic algorithms. *Caltech Concurrent Computation Program*, 10 2000.

[45] Pablo Moscato and Carlos Cotta. *A Modern Introduction to Memetic Algorithms*, pages 141–183. Springer US, 2010.

[46] Pablo Moscato and Luke Mathieson. Memetic algorithms for business analytics

and data science: A brief survey. *Moscato P., de Vries N. (eds) Business and Consumer Analytics: New Ideas. Springer, Cham*, pages 545–608, 05 2019.

[47] Vitaly Osipov and Peter Sanders. n-Level Graph Partitioning. In *Proceedings of the 18th European Conference on Algorithms: Part I*, volume 6346 of *LNCS*, pages 278–289. Springer, 2010.

[48] Sylvain Paris, Samuel W Hasinoff, and Jan Kautz. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. *ACM Trans. Graph.*, 30(4):68, 2011.

[49] David J. Pearce and Paul H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithms*, pages 1–7, 2006.

[50] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[51] Louis-Noël Pouchet. Polybench/c: The polyhedral benchmark suite. `https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`, 2012.

[52] Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th European Symposium on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

[53] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In *Proceedings of the Meeting on Algorithm Engineering and Expermiments*, ALENEX '12, page 16–29, USA, 2012. Society for Industrial and Applied Mathematics.

[54] Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD thesis, 2019.

[55] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 53–67.

[56] Sebastian Schlag, Matthias Schmitt, and Christian Schulz. Faster support vector machines. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 199–210.

[57] Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Darren Strash. Scalable edge partitioning. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 211–225.

[58] Christian Schulz and Darren Strash. Graph partitioning: Formulations and applications to big data. In *Encyclopedia of Big Data Technologies*, pages 1–7. Springer, 2018.

[59] Aleksandar Trifunovic and William J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563 – 581, 2008.

[60] Jesper Larsson Träff. Direct graph k-partitioning with a kernighan–lin like heuristic. *Operations Research Letters*, 34(6):621 – 629, 2006.

[61] Chris Walshaw and Marg Cross. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, January 2000.

A Detailed DAG Results

Table A.1: Detailed per instance results on the ISPD98 benchmark set [3]. `HOUKC` refers to the algorithm developed by Herrmann et. al. [25]. `mlDHGP + X` refers to our multilevel algorithm and `memDHGP + X` refers to our memetic algorithm with `X` as undirected hypergraph partitioner for initial partitioning. The *Best* column reports the best edge cut found during 8 hours of individual runs. For `mlDHGP + X`, the *Average* column reports the average edge cut of 5 individual runs and the *Best* column reports the best edge cut found during 8 hours. For `memDHGP + X`, the *Best* column reports the best result found after running for 8 hours. The *Overall Best* column shows the best cut found by any tool with the following identifiers: H: `HOUKC`, N: one of the new approaches. In general, lower is better.

		HOUKC		mlDHGP with KaHyPar		memDHGP	mlDHGP with PaToH		memDHGP	Overall Best	
Graph	K	Average	Best (8h)	Average	Best (8h)	Best (8h)	Average	Best (8h)		Result	Solver
ibm01	2	3 175	2 752	3 235	2 428	**2 255**	2 730	2 290		2 255	N
	4	6 092	5 099	5 434	5 028	4 848	5 325	**4 841**		4 841	N
	8	7 449	6 880	8 026	7 240	6 958	8 268	**6 639**		6 639	N
	16	10 555	8 603	9 131	8 135	8 028	8 870	**7 627**		7 627	N
	32	12 652	11 119	10 909	10 086	9 572	11 107	**9 404**		9 404	N
ibm02	2	8 540	4 708	8 772	**3 262**	5 873	8 806	8 599		3 262	N
	4	13 264	11 375	12 290	**11 374**	11 497	12 317	11 400		11 374	N
	8	17 832	16 591	17 557	16 522	**16 253**	17 520	16 387		16 253	N
	16	24 856	23 002	21 708	20 209	**19 727**	22 128	20 455		19 727	N
	32	30 407	29 082	26 379	25 263	**24 264**	26 659	25 393		24 264	N
ibm03	2	14 601	13 687	15 278	12 584	**11 870**	14 265	12 051		11 870	N
	4	21 802	20 077	20 652	18 622	**17 757**	18 840	17 835		17 757	N
	8	26 051	24 361	25 370	21 494	**20 579**	22 975	20 699		20 579	N
	16	30 776	27 238	29 885	24 637	24 006	28 097	**23 837**		23 837	N
	32	33 439	31 034	32 134	27 309	27 093	30 035	**27 085**		27 085	N
ibm04	2	9 518	9 108	9 727	8 508	**8 237**	9 727	**8 237**		8 237	N
	4	14 226	13 190	12 668	11 512	10 970	12 358	**10 944**		10 944	N
	8	18 508	16 683	18 677	16 983	16 298	18 811	**15 878**		15 878	N
	16	25 885	22 874	24 363	22 800	21 812	24 298	**21 373**		21 373	N
	32	30 512	27 107	27 882	26 486	**25 078**	28 127	25 680		25 078	N
ibm05	2	8 360	5 882	7 494	**5 478**	5 830	7 285	6 979		5 478	N
	4	17 040	13 278	14 932	10 740	**10 710**	15 035	11 885		10 710	N
	8	23 170	19 480	19 618	16 076	15 980	19 803	**15 934**		15 934	N
	16	29 747	25 590	25 512	22 049	**20 771**	24 914	21 604		20 771	N
	32	34 495	30 721	29 437	27 465	27 582	30 155	**26 899**		26 899	N
ibm06	2	14 049	12 736	12 664	11 804	11 341	12 832	**11 285**		11 285	N
	4	23 206	20 317	21 641	19 097	**18 197**	21 705	18 374		18 197	N
	8	30 875	26 980	25 402	23 202	22 455	25 155	**22 263**		22 263	N
	16	34 069	30 848	29 421	27 435	**26 384**	29 793	27 263		26 384	N
	32	38 243	36 197	32 781	31 310	30 839	32 826	**30 597**		30 597	N
ibm07	2	16 341	15 855	15 738	15 356	13 681	16 003	**12 965**		12 965	N
	4	26 842	23 522	22 608	21 583	20 499	22 273	**20 348**		20 348	N
	8	29 702	27 069	26 935	25 655	**24 464**	27 186	24 586		24 464	N
	16	36 633	33 606	31 746	30 788	29 808	32 195	**29 797**		29 797	N
	32	43 083	40 205	36 959	35 901	**34 648**	37 017	34 665		34 648	N
ibm08	2	25 139	24 481	24 418	22 381	22 079	24 384	**21 925**		21 925	N
	4	52 118	38 711	41 350	38 644	38 495	41 402	**38 330**		38 330	N
	8	84 639	81 587	50 063	49 238	48 429	50 043	**47 124**		47 124	N
	16	96 107	88 135	88 727	87 323	**85 996**	89 513	86 083		85 996	N
	32	109 264	96 746	93 556	92 591	90 779	94 172	**90 660**		90 660	N
ibm09	2	19 509	15 084	17 233	12 661	12 305	16 307	**12 127**		12 127	N
	4	28 408	25 120	26 143	23 342	22 557	26 184	**20 892**		20 892	N
	8	36 168	31 734	33 276	30 411	**29 654**	34 341	30 168		29 654	N
	16	42 373	39 154	39 712	37 301	35 902	39 529	**34 707**		34 707	N
	32	50 041	45 987	45 226	41 007	40 701	45 131	**39 887**		39 887	N

Table A.2: Detailed per instance results on the ISPD98 benchmark set [3]. `HOUKC` refers to the algorithm developed by Herrmann et. al. [25]. `mlDHGP + X` refers to our multilevel algorithm and `memDHGP + X` refers to our memetic algorithm with `X` as undirected hypergraph partitioner for initial partitioning. The *Best* column reports the best edge cut found during 8 hours of individual runs. For `mlDHGP + X`, the *Average* column reports the average edge cut of 5 individual runs and the *Best* column reports the best edge cut found during 8 hours. For `memDHGP + X`, the *Best* column reports the best result found after running for 8 hours. The *Overall Best* column shows the best cut found by any tool with the following identifiers: H: `HOUKC`, N: one of the new approaches. In general, lower is better.

		HOUKC		mlDHGP with KaHyPar		memDHGP	mlDHGP with PaToH	memDHGP	Overall Best	
Graph	K	Average	Best (8h)	Average	Best (8h)	Best (8h)	Average	Best (8h)	Result	Solver
ibm10	2	24 983	24 073	24 310	21 575	21 328	22 560	**21 310**	21 310	N
	4	38 620	35 083	39 383	33 217	36 352	39 288	**32 101**	32 101	N
	8	49 646	44 820	47 827	40 423	39 202	46 082	**38 238**	38 238	N
	16	63 960	54 164	55 610	50 854	**49 632**	56 129	49 892	49 632	N
	32	69 990	65 302	64 229	61 838	59 914	64 105	**59 180**	59 180	N
ibm11	2	19 224	16 926	21 879	14 374	13 578	15 748	**13 318**	13 318	N
	4	36 346	26 539	26 919	22 750	21 623	24 724	**21 310**	21 310	N
	8	39 755	32 812	32 816	30 401	28 563	33 247	**28 477**	28 477	N
	16	52 698	45 779	40 706	38 055	39 294	43 773	**37 257**	37 257	N
	32	63 925	57 699	50 612	47 999	**47 331**	52 963	47 930	47 331	N
ibm12	2	29 359	**27 238**	30 315	27 860	27 365	29 620	27 688	27 238	H
	4	50 457	47 922	49 225	44 108	**42 728**	49 591	46 107	42 728	N
	8	60 024	53 785	57 394	52 487	51 425	57 046	**50 955**	50 955	N
	16	72 429	65 979	66 486	62 965	**61 186**	67 160	61 484	61 186	N
	32	84 328	76 066	73 872	70 503	68 739	73 252	**68 712**	68 712	N
ibm13	2	30 698	19 008	21 700	**17 161**	17 484	22 151	17 659	17 161	N
	4	39 781	29 198	39 288	31 700	32 060	38 609	**26 500**	26 500	N
	8	54 061	**39 453**	55 253	42 881	44 535	47 765	41 596	39 453	H
	16	71 208	60 006	65 263	55 070	**49 820**	65 962	49 993	49 820	N
	32	89 053	76 762	81 831	72 262	74 997	81 416	**70 987**	70 987	N
ibm14	2	33 205	**31 988**	51 511	48 338	48 140	52 065	49 670	31 988	H
	4	55 342	**49 972**	69 320	64 838	62 888	70 364	66 680	49 972	H
	8	76 297	68 992	68 051	62 718	60 929	67 598	**56 972**	56 972	N
	16	96 638	80 591	79 801	74 705	**73 224**	80 029	73 861	73 224	N
	32	104 543	96 677	91 692	89 688	87 904	92 823	**86 504**	86 504	N
ibm15	2	74 713	71 593	66 301	63 603	**63 136**	82 679	67 804	63 136	N
	4	105 577	95 911	97 786	**87 849**	92 812	96 479	88 349	87 849	N
	8	146 984	123 993	123 403	112 014	113 564	124 884	**108 619**	108 619	N
	16	169 587	153 693	136 151	135 061	**124 709**	143 941	133 614	124 709	N
	32	191 476	174 057	158 765	154 660	149 558	160 815	**148 763**	148 763	N
ibm16	2	55 871	52 980	51 699	48 222	48 063	50 167	**45 371**	45 371	N
	4	108 576	93 874	98 471	93 941	91 481	99 729	**89 976**	89 976	N
	8	130 302	117 375	129 900	115 437	119 439	126 431	**114 458**	114 458	N
	16	162 743	148 626	147 987	136 916	134 387	142 235	**132 412**	132 412	N
	32	181 924	172 909	166 347	158 854	157 879	164 966	**153 490**	153 490	N
ibm17	2	75 860	57 177	70 331	59 100	59 470	61 401	**56 895**	56 895	N
	4	100 287	89 849	121 023	78 692	**77 889**	121 175	107 211	77 889	N
	8	151 126	141 679	152 455	**124 639**	126 610	147 848	130 307	124 639	N
	16	182 272	166 847	171 507	153 812	155 789	165 498	**150 026**	150 026	N
	32	211 541	198 404	188 792	**167 274**	173 762	194 056	182 853	167 274	N
ibm18	2	37 123	34 949	35 907	33 434	33 394	36 651	**33 277**	33 277	N
	4	63 000	53 948	64 540	53 190	53 237	58 432	**48 482**	48 482	N
	8	92 636	78 164	86 580	76 686	75 728	81 435	**70 558**	70 558	N
	16	121 219	108 744	107 824	93 018	**88 959**	113 181	98 976	88 959	N
	32	144 219	132 289	124 788	111 650	**110 816**	128 875	119 170	110 816	N
Mean		41189	36205	37 828	33 459	33 007	37 382	33 088		

Table A.3: Detailed per instance results on the PolyBench benchmark set [51]. `HOUKC` refers to the algorithm developed by Herrmann et. al. [25]. `Moreira` refers to the algorithm developed by Moreira et al. [43]. `mlDHGP + X` refers to our multilevel algorithm and `memDHGP + X` refers to our memetic algorithm with `X` as undirected hypergraph partitioner for initial partitioning. For `HOUKC`, the *Average* column reports the better average from Table A.1 and Table A.2 in [25] and the *Best* column reports the best edge cut found during 8 hours of individual runs or the best edge cut reported in [25], if that is lower (marked with a star). For `mlDHGP + X`, the *Average* column reports the average edge cut of 5 individual runs and the *Best* column reports the best edge cut found during 8 hours. For `memDHGP + X`, the *Best* column reports the best result found after running for 8 hours. The *Overall Best* column shows the best cut found by any tool with the following identifiers: H: `HOUKC`, N: one of the new approaches, M: `Moreira` et. al. In general, lower is better.

		HOUKC		Moreira et. al.		mlDHGP memDHGP with KaHyPar			mlDHGP memDHGP with PaToH		Overall Best	
Graph	K	Average	Best (8h) or [25]	Average	Best	Average	Best (8h)	Best (8h)	Average	Best (8h)	Result	Solver
2mm	2	**200**	**200**	**200**	**200**	**200**	**200**	**200**	**200**	**200**	200	H,M,N
	4	2 160	946	947	**930**	1 065	**930**	**930**	1 006	**930**	930	M,N
	8	5 361	2 910	7 181	6 604	2 819	2 576	**2 465**	5 563	5 110	2 465	N
	16	11 196	8 103	13 330	13 092	7 090	5 963	**5 435**	7 881	6 632	5 435	N
	32	15 911	12 708	14 583	14 321	11 397	10 635	**10 398**	12 228	11 012	10 398	N
3mm	2	1 000	**800**	1 000	1 000	**800**	**800**	**800**	1 000	1 000	800	H,N
	4	9 264	**2 600**	38 722	37 899	2 647	**2 600**	**2 600**	**2 600**	**2 600**	2 600	H,N
	8	24 330	7 735	58 129	49 559	8 596	6 967	**6 861**	14 871	9 560	6 861	N
	16	37 041	21 903	64 384	60 127	23 513	**19 625**	19 675	28 021	23 967	19 675	N
	32	46 437	36 718	62 279	58 190	34 721	**30 908**	31 423	38 879	34 353	31 423	N
adi	2	142 719	*__134 675__	134 945	**134 675**	138 433	138 057	138 279	138 520	138 329	134 675	H,M
	4	212 938	**210 979**	284 666	283 892	213 255	212 709	212 851	213 390	212 564	210 979	H
	8	256 302	**229 563**	290 823	290 672	253 885	252 271	253 206	254 282	252 376	229 563	H
	16	282 485	**271 374**	326 963	326 923	281 068	277 337	280 437	281 751	276 958	271 374	H
	32	306 075	305 091	370 876	370 413	309 930	303 078	**299 387**	309 757	302 157	302 157	N
atax	2	39 876	32 451	47 826	47 424	39 695	24 150	**23 690**	45 130	43 450	23 690	N
	4	48 645	43 511	82 397	76 245	50 725	42 028	**39 316**	50 144	47 486	39 316	N
	8	51 243	48 702	113 410	111 051	54 891	48 824	**47 741**	52 163	49 450	47 741	N
	16	59 208	52 127	127 687	125 146	68 153	**50 962**	51 256	53 256	51 191	51 191	N
	32	69 556	57 930	132 092	130 854	66 267	54 613	56 051	56 773	**54 536**	54 536	N
covariance	2	27 269	**4 775**	66 520	66 445	**4 775**	**4 775**	**4 775**	5 893	5 641	4 775	H,N
	4	61 991	*34 307	84 626	84 213	12 362	11 724	**11 281**	13 339	12 344	11 281	N
	8	74 325	*50 680	103 710	102 425	24 429	21 460	**21 106**	51 984	41 807	21 106	N
	16	119 284	99 629	125 816	123 276	62 011	60 143	**58 875**	65 302	59 153	58 875	N
	32	121 155	94 247	142 214	137 905	76 977	73 758	**72 090**	80 464	74 770	72 090	N
doitgen	2	5 035	**3 000**	43 807	42 208	**3 000**	**3 000**	**3 000**	**3 000**	**3 000**	3 000	H,N
	4	37 051	**9 000**	72 115	71 082	11 029	**9 000**	**9 000**	28 317	27 852	9 000	H,N
	8	51 283	36 790	76 977	75 114	36 326	34 912	**34 682**	42 185	38 491	34 682	N
	16	62 296	50 481	84 203	77 436	51 064	48 992	50 486	50 993	**48 193**	48 193	N
	32	68 350	59 632	94 135	92 739	59 159	58 184	57 408	57 208	**55 721**	55 721	N
durbin	2	**12 997**	**12 997**	**12 997**	**12 997**	**12 997**	**12 997**	**12 997**	**12 997**	**12 997**	12 997	H,M,N
	4	21 566	*21 566	21 641	21 641	21 556	21 557	**21 541**	21 556	**21 541**	21 541	N
	8	27 519	27 518	27 571	27 571	27 511	**27 508**	27 509	27 511	27 509	27 509	N
	16	32 852	32 841	32 865	32 865	32 869	**32 824**	32 825	32 852	32 825	32 825	N
	32	39 738	39 732	39 726	39 725	39 753	39 717	**39 701**	39 753	**39 701**	39 701	N
fdtd-2d	2	6 024	**4 381**	5 494	5 494	5 233	4 756	4 604	6 318	6 285	4 381	H
	4	15 294	11 551	15 100	15 099	11 670	9 325	**9 240**	11 572	10 232	9 240	N
	8	23 699	19 527	33 087	32 355	17 704	15 906	**15 653**	17 990	15 758	15 653	N
	16	32 917	28 065	35 714	35 239	25 170	22 866	22 041	24 582	**22 003**	22 003	N
	32	42 515	39 063	43 961	42 507	32 658	30 872	29 868	32 682	**29 772**	29 772	N
gemm	2	**4 200**	**4 200**	383 084	382 433	**4 200**	**4 200**	**4 200**	4 768	4 690	4 200	H,N
	4	59 854	**12 600**	507 250	500 526	**12 600**	**12 600**	**12 600**	13 300	**12 600**	12 600	H,N
	8	116 990	33 382	578 951	575 004	70 827	31 413	**30 912**	188 172	175 495	30 912	N
	16	263 050	224 173	615 342	613 373	185 872	164 235	**148 040**	202 920	194 017	148 040	N
	32	330 937	277 879	626 472	623 271	270 909	265 771	**258 607**	280 849	275 188	258 607	N
gemver	2	20 913	*20 913	29 349	29 270	22 725	19 485	19 390	20 317	**18 930**	18 930	N
	4	40 299	35 431	49 361	49 229	38 600	35 021	**33 324**	37 632	34 328	33 324	N
	8	55 266	43 716	68 163	67 094	50 440	44 253	43 276	47 799	**42 548**	42 548	N
	16	59 072	54 012	78 115	75 596	53 819	48 618	48 182	53 775	**46 563**	46 563	N
	32	73 131	63 012	85 331	84 865	58 898	53 581	54 953	59 210	**52 404**	52 404	N
gesummv	2	**500**	**500**	1 666	**500**	**500**	**500**	**500**	**500**	**500**	500	H,M,N
	4	10 316	**1 500**	98 542	94 493	5 096	**1 500**	**1 500**	1 548	**1 500**	1 500	N
	8	9 618	4 021	101 533	98 982	25 535	**3 500**	**3 500**	3 640	**3 500**	3 500	N
	16	35 686	11 388	112 064	104 866	30 215	**7 500**	**7 500**	7 883	**7 500**	7 500	N
	32	45 050	28 295	117 752	114 812	31 740	15 620	16 339	16 144	**15 500**	15 500	N
heat-3d	2	9 378	8 936	8 695	8 684	8 930	**8 640**	**8 640**	9 242	8 936	8 640	N
	4	16 700	15 755	**14 592**	**14 592**	15 355	14 642	**14 592**	16 304	14 865	14 592	M,N
	8	25 883	24 326	**20 608**	**20 608**	23 307	21 190	21 300	25 462	23 074	20 608	M
	16	42 137	*41 261	31 615	**31 500**	38 909	38 053	35 909	40 148	37 659	31 500	M
	32	64 614	60 215	51 963	**50 758**	55 360	53 525	51 682	54 621	50 848	50 758	M

Table A.4: Detailed per instance results on the PolyBench benchmark set [51]. `HOUKC` refers to the algorithm developed by Herrmann et. al. [25]. `Moreira` refers to the algorithm developed by Moreira et al. [43]. `mlDHGP + X` refers to our multilevel algorithm and `memDHGP + X` refers to our memetic algorithm with `X` as undirected hypergraph partitioner for initial partitioning. For `HOUKC`, the *Average* column reports the better average from Table A.1 and Table A.2 in [25] and the *Best* column reports the best edge cut found during 8 hours of individual runs or the best edge cut reported in [25], if that is lower (marked with a star). For `mlDHGP + X`, the *Average* column reports the average edge cut of 5 individual runs and the *Best* column reports the best edge cut found during 8 hours. For `memDHGP + X`, the *Best* column reports the best result found after running for 8 hours. The *Overall Best* column shows the best cut found by any tool with the following identifiers: H: `HOUKC`, N: one of the new approaches, M: `Moreira` et. al. In general, lower is better.

		HOUKC		Moreira et. al.		mlDHGP with KaHyPar		memDHGP with KaHyPar	mlDHGP with PaToH	memDHGP with PaToH	Overall Best	
Graph	K	Average	Best (8h) or [25]	Average	Best	Average	Best (8h)	Best (8h)	Average	Best (8h)	Result	Solver
jacobi-1d	2	646	**400**	596	596	440	**400**	**400**	491	423	400	H,N
	4	1 617	1 123	1 493	1 492	1 188	1 046	**1 044**	1 250	1 128	1 044	N
	8	2 845	2 052	3 136	3 136	2 028	1 754	**1 750**	2 170	1 855	1 750	N
	16	4 519	3 517	6 340	6 338	3 140	2 912	**2 869**	3 355	2 982	2 869	N
	32	6 742	5 545	8 923	8 750	4 776	4 565	**4 498**	4 910	4 587	4 498	N
jacobi-2d	2	3 445	*3 342	2 994	2 991	3 878	3 000	**2 986**	3 942	3 129	2 986	N
	4	7 370	7 243	5 701	**5 700**	7 591	5 979	5 881	7 528	6 245	5 700	M
	8	13 168	12 134	9 417	9 416	10 872	9 295	**8 935**	11 753	10 492	8 935	N
	16	21 565	18 394	16 274	16 231	15 605	14 746	**13 867**	15 889	14 736	13 867	N
	32	29 558	25 740	22 181	21 758	20 597	19 647	**18 979**	21 653	19 530	18 979	N
lu	2	5 351	**4 160**	5 210	5 162	**4 160**	**4 160**	**4 160**	**4 160**	**4 160**	4 160	H,N
	4	21 258	**12 214**	13 528	13 510	12 720	**12 214**	**12 214**	16 091	15 992	12 214	H,N
	8	53 643	34 074	33 307	**33 211**	42 963	33 873	33 954	41 113	38 318	33 211	M
	16	105 289	81 713	74 543	**74 006**	81 224	74 400	74 448	83 980	75 150	74 006	M
	32	156 187	141 868	130 674	129 954	125 932	122 977	**121 451**	131 850	127 904	121 451	N
ludcmp	2	5 731	**5 337**	5 380	**5 337**	**5 337**	**5 337**	**5 337**	**5 337**	**5 337**	5 337	H,N
	4	22 368	15 170	**14 744**	**14 744**	18 114	16 889	17 560	26 606	17 113	14 744	N
	8	60 255	41 086	37 228	**37 069**	46 268	37 688	37 790	52 980	39 362	37 069	N
	16	106 223	86 959	78 646	78 467	89 958	**76 074**	80 706	96 275	85 572	78 467	N
	32	158 619	144 224	134 758	134 288	130 552	**125 957**	127 454	136 218	131 161	127 454	N
mvt	2	21 281	16 768	24 528	23 091	23 798	**16 584**	16 596	32 856	20 016	16 596	N
	4	38 215	**29 229**	74 386	73 035	41 156	29 318	30 070	52 353	42 870	29 229	H,N
	8	46 776	39 295	86 525	82 221	50 853	36 531	**35 471**	60 021	55 460	35 471	N
	16	54 925	48 036	99 144	97 941	58 258	**41 727**	42 890	65 738	59 194	42 890	N
	32	62 584	54 293	105 066	104 917	58 413	**45 958**	46 122	69 221	64 611	46 122	N
seidel-2d	2	4 374	**3 401**	4 991	4 969	4 036	3 578	3 504	4 206	3 786	3 401	H,N
	4	11 784	10 872	12 197	12 169	11 352	10 645	**10 404**	11 480	10 604	10 404	N
	8	21 937	20 711	21 419	21 400	19 954	18 528	**17 770**	20 309	18 482	17 770	N
	16	38 065	33 647	38 222	38 110	29 930	27 644	**27 583**	30 329	28 348	27 583	N
	32	58 319	51 745	52 246	51 531	41 256	38 949	**38 175**	42 291	39 058	38 175	N
symm	2	26 374	21 963	94 357	94 214	22 000	21 840	**21 836**	29 871	26 134	21 836	N
	4	59 815	42 442	127 497	126 207	41 486	38 290	**37 854**	65 111	57 620	37 854	N
	8	91 892	69 554	152 984	151 168	69 569	**58 084**	60 644	82 865	75 151	60 644	N
	16	105 418	89 320	167 822	167 512	90 978	**83 703**	85 508	96 932	89 445	85 508	N
	32	108 950	**97 174**	174 938	174 843	110 495	104 376	100 337	108 814	104 592	97 174	H
syr2k	2	4 343	**900**	11 098	3 894	**900**	**900**	**900**	**900**	**900**	900	H,N
	4	12 192	3 048	49 662	48 021	3 150	2 978	**2 909**	16 589	9 991	2 909	N
	8	28 787	12 833	57 584	57 408	12 504	**9 969**	10 154	21 427	19 507	10 154	N
	16	29 519	24 457	59 780	59 594	25 054	**21 626**	21 828	26 120	23 588	21 828	N
	32	36 111	31 138	60 502	60 085	33 424	31 236	29 984	31 358	**29 340**	29 340	N
syrk	2	11 740	**3 240**	219 263	218 019	**3 240**	**3 240**	**3 240**	3 439	**3 240**	3 240	H,N
	4	56 832	**9 960**	289 509	289 088	10 417	10 119	9 970	89 457	80 801	9 960	H
	8	112 236	**30 602**	329 466	327 712	83 000	46 130	58 876	107 220	101 516	30 602	H
	16	179 042	147 058	354 223	351 824	117 357	113 122	**111 635**	150 363	135 615	111 635	N
	32	196 173	173 550	362 016	359 544	158 590	**154 818**	154 921	182 222	175 999	154 921	N
trisolv	2	336	280	6 788	3 549	280	**279**	**279**	308	**279**	279	N
	4	828	827	43 927	43 549	823	**821**	**821**	865	823	821	N
	8	2 156	1 907	66 148	65 662	2 112	**1 893**	1 895	2 035	1 897	1 895	N
	16	6 240	5 285	71 838	71 447	8 719	4 125	**4 108**	4 358	4 240	4 108	N
	32	13 431	*13 172	79 125	79 071	16 027	8 942	8 784	9 210	**8 716**	8 716	N
trmm	2	13 659	**3 440**	138 937	138 725	**3 440**	**3 440**	**3 440**	**3 440**	**3 440**	3 440	H,N
	4	58 477	14 543	192 752	191 492	14 942	12 622	**12 389**	35 964	35 824	12 389	N
	8	92 185	49 830	225 192	223 529	65 303	46 059	**45 053**	67 011	61 045	45 053	N
	16	128 838	103 975	240 788	238 159	92 172	**79 507**	80 186	96 421	87 275	80 186	N
	32	153 644	131 899	246 407	245 173	120 839	115 460	**112 267**	120 753	113 205	112 267	N
Mean		25777	17897	44 923	43 200	18 887	15 988	16 095	20 308	18 642		

B Detailed DAH Results

Table B.1: Detailed per instance results on the ISPD98 benchmark suite [3]. `mlDHGP` refers to our algorithm with `KaHyPar` as undirected hypergraph partitioner for initial partitioning. `memDHGP` refers to our memetic algorithm that uses `mlDHGP` equipped with `KaHyPar` as undirected hypergraph partitioner for initial partitioning to build an initial population. The *Best* column reports the best edge cut found during 8 hours of individual runs. For `mlDHGP`, the *Average* column reports the average edge cut of 5 individual runs. For `memDHGP`, the *Best* column reports the best result found after running for 8 hours. In general, lower is better.

		mlDHGP	memDHGP	TopoOrderPartRB		TopoOrderPartKWay	
Hypergraph	K	Average	Best (8h)	Average	Best (8h)	Average	Best (8h)
ibm01	2	838	**629**	877	659	1 267	660
	4	1 835	**1 427**	2 035	1 684	4 921	2 615
	8	2 923	**2 136**	3 512	2 670	6 513	4 153
	16	3 764	**3 049**	4 584	3 710	8 271	6 032
	32	4 774	**4 013**	5 626	4 706	9 894	6 652
ibm02	2	2 222	**1 869**	2 629	1 990	3 048	2 319
	4	4 391	**3 247**	5 296	4 017	7 520	5 185
	8	6 898	**5 674**	8 561	6 677	11 208	9 485
	16	9 787	**8 481**	10 678	9 300	14 195	12 709
	32	12 545	**11 448**	13 773	12 362	18 141	14 596
ibm03	2	3 782	**2 242**	3 772	2 862	4 306	2 932
	4	5 955	**4 231**	6 335	4 748	8 661	6 746
	8	7 679	**5 911**	8 478	6 771	12 510	10 131
	16	9 179	**7 386**	10 278	8 601	15 725	12 304
	32	10 051	**8 496**	12 271	10 116	18 507	14 162
ibm04	2	3 080	**717**	4 448	3 044	5 252	3 204
	4	5 232	**2 467**	6 175	3 707	9 871	6 086
	8	7 239	**5 339**	9 919	7 304	13 859	9 917
	16	9 415	**7 343**	11 868	10 029	17 680	12 584
	32	11 129	**9 259**	13 795	11 947	21 342	16 273
ibm05	2	4 630	**3 954**	4 799	4 232	4 952	4 248
	4	7 629	**5 930**	9 574	7 222	11 693	7 933
	8	10 434	**8 612**	13 292	10 339	17 575	11 821
	16	13 095	**11 285**	16 394	13 566	21 884	16 613
	32	15 371	**13 837**	18 577	15 938	25 750	20 079
ibm06	2	4 486	**2 730**	5 624	3 839	7 027	4 279
	4	8 189	**5 858**	8 789	6 648	14 557	11 971
	8	10 203	**8 281**	11 483	9 590	19 122	15 012
	16	12 720	**10 157**	14 123	11 751	23 880	20 591
	32	15 155	**12 179**	17 588	14 851	30 019	24 375
ibm07	2	5 355	**3 680**	8 273	3 871	8 831	4 602
	4	10 343	**6 250**	11 463	7 130	16 176	11 691
	8	12 386	**8 993**	13 861	9 482	21 580	16 862
	16	13 927	**11 870**	17 289	14 109	27 718	23 060
	32	16 880	**14 264**	20 418	17 595	34 122	26 761
ibm08	2	12 865	9 344	11 639	**9 331**	12 536	11 281
	4	18 373	16 860	18 385	**15 454**	22 071	20 504
	8	22 238	20 526	21 703	**20 013**	28 396	24 835
	16	25 572	**23 100**	27 093	25 072	34 302	30 639
	32	29 667	**27 425**	31 907	29 150	40 661	34 423
ibm09	2	5 593	**3 357**	11 804	9 159	13 130	9 651
	4	10 610	**6 416**	18 274	13 852	21 028	17 693
	8	12 053	**8 726**	20 346	16 942	26 819	21 107
	16	14 987	**11 588**	22 256	20 090	33 138	28 843
	32	17 802	**14 449**	25 313	22 858	39 478	34 857

Table B.2: Detailed per instance results on the ISPD98 benchmark suite [3]. `mlDHGP` refers to our algorithm with `KaHyPar` as undirected hypergraph partitioner for initial partitioning. `memDHGP` refers to our memetic algorithm that uses `mlDHGP` equipped with `KaHyPar` as undirected hypergraph partitioner for initial partitioning to build an initial population. The *Best* column reports the best edge cut found during 8 hours of individual runs. For `mlDHGP`, the *Average* column reports the average edge cut of 5 individual runs. For `memDHGP`, the *Best* column reports the best result found after running for 8 hours. In general, lower is better.

		mlDHGP	memDHGP	TopoOrderPartRB		TopoOrderPartKWay	
Hypergraph	K	Average	Best (8h)	Average	Best (8h)	Average	Best (8h)
ibm10	2	12 129	**8 288**	11 717	8 315	17 445	11 372
	4	18 728	**11 809**	15 691	12 922	24 176	18 224
	8	22 141	**16 401**	22 146	18 837	36 282	28 691
	16	24 929	**20 199**	27 168	24 613	49 440	39 218
	32	30 100	**26 238**	33 799	30 462	59 161	51 361
ibm11	2	10 669	**6 550**	10 943	7 618	13 320	8 190
	4	16 257	**10 447**	18 761	15 073	25 457	20 894
	8	17 992	**12 341**	23 071	18 430	36 747	29 617
	16	20 197	**16 198**	27 404	23 673	42 915	35 907
	32	23 409	**19 767**	30 592	27 495	50 761	43 688
ibm12	2	15 449	11 349	14 881	**10 588**	14 858	12 725
	4	20 307	**15 652**	20 215	16 538	23 398	18 847
	8	23 036	**18 126**	24 916	21 053	36 501	31 499
	16	28 437	**23 367**	30 176	26 434	48 537	35 157
	32	34 536	**27 911**	38 183	33 930	62 718	53 703
ibm13	2	11 893	8 695	12 790	**8 284**	19 262	10 593
	4	14 791	**10 285**	21 166	12 883	43 564	34 438
	8	21 405	**14 330**	32 543	21 452	58 298	48 381
	16	25 313	**16 761**	35 524	28 945	70 447	58 881
	32	29 676	**26 017**	43 126	37 405	92 014	81 315
ibm14	2	24 379	14 713	15 305	**14 219**	21 228	18 308
	4	30 912	21 613	24 657	**21 539**	38 520	33 361
	8	36 370	**30 710**	36 889	32 478	49 762	44 724
	16	42 321	**35 598**	44 721	40 298	63 893	58 170
	32	48 741	**43 979**	53 609	48 580	78 351	72 065
ibm15	2	27 810	**19 804**	28 396	24 432	38 193	29 247
	4	44 069	**33 151**	52 804	46 517	79 810	74 511
	8	51 886	**38 306**	65 971	57 918	102 738	93 440
	16	58 961	**49 687**	74 815	68 546	119 898	105 971
	32	66 287	**56 374**	82 252	75 762	141 076	129 211
ibm16	2	25 941	**11 494**	26 062	19 608	34 882	30 835
	4	46 933	**36 124**	50 521	45 089	80 648	61 926
	8	57 761	**45 328**	61 132	55 221	97 002	89 035
	16	67 904	**58 471**	72 549	64 470	114 337	105 134
	32	80 591	**69 325**	83 631	77 873	135 371	127 061
ibm17	2	36 934	32 507	27 629	**25 282**	36 665	29 655
	4	47 186	39 301	42 638	**37 519**	64 506	53 945
	8	62 896	**54 230**	61 043	55 706	86 830	78 413
	16	74 427	**65 672**	77 867	70 777	109 982	101 743
	32	86 597	**80 152**	93 864	87 711	129 758	121 273
ibm18	2	21 296	**16 338**	20 830	18 825	24 096	18 409
	4	36 235	**28 131**	33 642	31 101	50 926	39 056
	8	49 742	**38 947**	45 337	41 406	73 198	65 903
	16	57 312	**47 532**	60 672	55 461	94 165	84 746
	32	67 770	**58 061**	76 324	69 222	114 532	104 020
Mean		16 151	12 245	18 344	15 030	26 839	21 246

Table B.3: Detailed per instance results on the PolyBench benchmark set [51]. `mlDHGP` refers to our algorithm with `KaHyPar` as undirected hypergraph partitioner for initial partitioning. `memDHGP` refers to our memetic algorithm that uses `mlDHGP` equipped with `KaHyPar` as undirected hypergraph partitioner for initial partitioning to build an initial population. The *Best* column reports the best edge cut found during 8 hours of individual runs. For `mlDHGP`, the *Average* column reports the average edge cut of 5 individual runs. For `memDHGP`, the *Best* column reports the best result found after running for 8 hours. In general, lower is better.

		mlDHGP	memDHGP	TopoOrderPartRB		TopoOrderPartKWay	
Hypergraph	K	Average	Best (8h)	Average	Best (8h)	Average	Best (8h)
2mm	2	212	**200**	224	**200**	344	210
	4	633	**608**	905	840	1 618	750
	8	1 376	**1 320**	1 608	1 440	3 169	1 433
	16	2 239	**2 153**	2 695	2 248	4 691	2 630
	32	3 796	**3 624**	4 562	3 934	7 015	4 229
3mm	2	**800**	**800**	1 112	805	1 564	1 090
	4	2 419	**2 000**	3 155	2 480	5 036	3 566
	8	3 950	**3 540**	5 940	4 689	9 374	5 653
	16	6 264	**5 999**	9 099	7 537	12 996	8 123
	32	9 234	**8 861**	12 719	11 483	19 224	12 516
atax	2	9 206	**460**	**460**	**460**	14 644	5 829
	4	9 438	4 943	7 162	**1 719**	24 248	19 462
	8	22 036	17 127	20 110	**9 291**	27 736	20 983
	16	30 917	28 378	29 675	**24 167**	46 152	29 036
	32	43 936	41 981	40 637	**39 098**	52 265	46 790
covariance	2	2 930	**2 590**	3 343	3 160	3 190	3 059
	4	6 058	5 705	5 361	**5 265**	7 029	5 681
	8	8 834	**8 238**	9 660	9 092	12 815	10 472
	16	13 406	**12 758**	13 917	13 480	19 825	16 529
	32	17 605	**17 210**	20 211	19 833	29 596	24 640
doitgen	2	**400**	**400**	3 134	2 927	3 444	2 283
	4	**1 200**	**1 200**	3 652	3 600	6 760	3 114
	8	2 892	**2 800**	5 301	4 613	11 254	5 405
	16	6 001	**5 800**	7 263	6 949	15 725	8 243
	32	9 566	**9 192**	11 405	11 221	20 172	14 876
durbin	2	**349**	**349**	**349**	**349**	352	**349**
	4	1 024	**1 020**	1 023	**1 020**	1 033	**1 020**
	8	2 361	**2 339**	2 362	2 344	2 375	2 344
	16	5 030	**4 996**	5 027	5 000	5 047	5 018
	32	10 374	10 364	10 366	**10 358**	10 396	10 378
fdtd-2d	2	2 650	**1 756**	3 491	3 490	3 491	3 490
	4	5 549	**3 960**	10 473	4 294	10 474	4 269
	8	7 755	**6 351**	13 745	8 673	24 366	8 120
	16	10 971	**8 959**	19 112	13 681	34 855	15 108
	32	14 110	**12 759**	24 248	18 726	42 703	22 036
gemm	2	**4 200**	**4 200**	6 179	4 758	5 989	4 506
	4	**12 600**	**12 600**	18 908	14 781	18 579	14 581
	8	20 931	**19 714**	39 528	39 290	41 055	35 135
	16	33 978	**31 355**	63 139	63 139	77 882	76 501
	32	52 721	**50 300**	89 660	89 660	117 319	115 717
gemver	2	2 577	**480**	1 824	**480**	4 800	2 947
	4	5 341	**2 070**	6 705	4 576	8 081	5 851
	8	10 615	**8 305**	10 522	8 357	15 511	9 673
	16	13 432	**12 474**	13 263	12 618	20 260	14 005
	32	17 250	16 576	16 823	**16 362**	25 050	21 086
gesummv	2	350	**250**	518	501	523	500
	4	975	**750**	927	760	1 191	1 051
	8	1 394	**1 250**	1 539	1 515	2 128	2 053
	16	2 247	**2 246**	2 600	2 582	5 403	2 971
	32	3 526	**3 428**	3 644	3 454	4 689	4 295
heat-3d	2	**1 280**	**1 280**	1 347	**1 280**	1 358	**1 280**
	4	3 843	**3 840**	3 947	**3 840**	4 190	**3 840**
	8	9 427	**8 777**	9 222	8 960	9 776	8 960
	16	15 406	14 509	16 496	14 325	19 799	**14 313**
	32	21 102	**19 382**	22 727	20 483	28 957	21 080

Table B.4: Detailed per instance results on the PolyBench benchmark set [51]. `mlDHGP` refers to our algorithm with `KaHyPar` as undirected hypergraph partitioner for initial partitioning. `memDHGP` refers to our memetic algorithm that uses `mlDHGP` equipped with `KaHyPar` as undirected hypergraph partitioner for initial partitioning to build an initial population. The *Best* column reports the best edge cut found during 8 hours of individual runs. For `mlDHGP`, the *Average* column reports the average edge cut of 5 individual runs. For `memDHGP`, the *Best* column reports the best result found after running for 8 hours. In general, lower is better.

		mlDHGP	memDHGP	TopoOrderPartRB		TopoOrderPartKWay	
Hypergraph	K	Average	Best (8h)	Average	Best (8h)	Average	Best (8h)
jacobi-1d	2	401	**400**	412	402	411	402
	4	926	**793**	1 245	1 206	1 279	1 206
	8	1 587	**1 467**	2 900	2 814	3 053	2 793
	16	2 634	**2 423**	6 213	3 900	6 676	3 349
	32	3 992	**3 786**	8 788	5 540	13 680	4 753
jacobi-2d	2	**1 008**	**1 008**	1 053	**1 008**	1 049	**1 008**
	4	3 524	**2 981**	3 093	3 024	3 129	3 024
	8	5 786	**4 995**	7 184	6 978	7 419	6 837
	16	8 198	**7 215**	13 070	9 282	15 715	8 992
	32	11 312	**10 326**	16 921	14 002	24 070	11 587
lu	2	3 327	3 221	2 966	**2 776**	3 644	3 190
	4	5 922	**5 735**	6 219	5 898	9 635	9 181
	8	10 218	**9 831**	10 971	10 837	20 432	18 592
	16	15 319	**15 145**	15 735	15 152	27 899	27 673
	32	22 034	**21 652**	23 252	22 984	36 568	36 178
ludcmp	2	2 952	**2 887**	3 020	2 917	4 364	3 878
	4	7 546	**7 468**	7 631	7 479	11 193	10 758
	8	12 568	12 494	12 557	**12 322**	22 516	22 189
	16	18 211	**17 933**	20 093	19 412	31 115	30 422
	32	25 273	**24 491**	26 447	26 164	42 154	42 154
mvt	2	446	**404**	3 247	558	11 174	468
	4	1 069	**818**	2 988	1 664	14 887	8 545
	8	2 425	**1 648**	6 860	4 187	20 852	14 909
	16	2 851	**2 586**	13 203	10 041	32 053	23 345
	32	6 288	**4 295**	14 809	10 009	40 295	34 690
seidel-2d	2	**838**	**838**	996	935	1 275	938
	4	2 582	**2 473**	2 775	2 672	3 349	2 784
	8	4 668	**4 274**	6 020	5 403	7 265	4 905
	16	7 247	**6 580**	10 166	9 045	14 873	9 157
	32	10 869	**9 966**	15 649	13 240	26 383	15 662
symm	2	836	**820**	2 915	2 346	2 946	2 808
	4	2 630	**2 540**	4 963	4 370	7 034	6 031
	8	6 257	**6 107**	9 023	8 862	11 819	9 618
	16	10 721	**10 445**	13 520	13 251	20 199	19 794
	32	15 672	**15 282**	18 851	18 594	33 848	32 173
syr2k	2	900	**880**	1 139	900	1 356	900
	4	1 938	**1 820**	2 327	1 978	3 062	1 994
	8	3 834	3 372	3 913	**3 198**	6 010	3 763
	16	5 579	**4 967**	5 868	5 294	10 551	5 354
	32	7 912	7 590	8 621	**7 520**	16 163	9 684
syrk	2	**3 240**	**3 240**	3 393	3 376	3 854	**3 240**
	4	7 390	**7 320**	10 083	10 079	10 431	9 482
	8	13 566	**13 202**	13 924	13 924	19 379	17 118
	16	20 121	**19 674**	31 052	30 851	30 102	28 704
	32	28 222	**27 446**	42 805	42 805	47 622	46 759
trisolv	2	**279**	**279**	280	**279**	283	280
	4	620	595	777	600	643	**581**
	8	1 088	**1 054**	1 260	1 133	1 366	1 289
	16	1 788	**1 742**	2 008	1 808	2 622	2 420
	32	2 783	**2 683**	3 347	3 020	4 420	3 984
trmm	2	2 704	**1 844**	3 755	3 579	4 113	3 440
	4	6 226	**5 673**	7 311	7 167	11 452	8 793
	8	10 082	**9 914**	13 669	13 484	19 559	12 482
	16	16 173	**15 472**	20 933	20 933	30 026	20 348
	32	22 126	**21 437**	27 168	27 168	42 503	41 895
Mean		4 447	3 900	5 698	4 853	8 247	6 045