Karlsruher Institut für Technologie

Bachelor's Thesis

Evaluating Spectral Partitioning as a Refinement Algorithm

Julian Zenker

Date: 1st July, 2024

Supervisors: Prof. Dr. Peter Sanders
Dr. Lars Gottesbüren
M.Sc. Nikolai Maas

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Spectral approaches to solve hypergraph partitioning problems recently have become subject to research and development as alternative to combinatorical approaches. This is mostly due to the ongoing increase in computing power available, which enables eigensolving in reasonable runtimes. Fed by a hint partition and in combination with traditional multi-level and tree-based techniques, spectral partitioners achieve competitive results on specific datasets from circuit design applications. In this work, we extend the hint-driven approach and implement spectral partitioning as a refinement algorithm for a multi-level framework. We evaluate the different techniques combined and prepare further research and development.
During our experiments, we found that our basic spectral refinement implementation improves the solution for 71% of the instances. Spectral refinement improved the overall cut by 4% in average. We also found that iterated spectral partitioning and spectral weight balancing have no positive effect on the cut on our datasets.

Spektrale Ansätze zur Lösung von Hypergraph-Partitionierungsproblemen sind in letzter Zeit Gegenstand von Forschung und Entwicklung als Alternative zu kombinatorischen Ansätzen geworden. Dies ist vor allem auf die fortwährende Steigerung der verfügbaren Rechenleistung zurückzuführen, die das Lösen von Eigenproblemen in annehmbaren Laufzeiten ermöglicht. Durch eine Hinweis-Partition gespeist und in Kombination mit traditionellen multi-level- und baumbasierten Verfahren erzielen spektrale Partitionierer konkurrenzfähige Ergebnisse auf spezifischen Datensätzen aus Anwendungen im elektronischen Schaltungsdesign. In dieser Arbeit erweitern wir den Hinweis-getriebenen Ansatz und implementieren spektrale Partitionierung als Refinement-Algorithmus für ein multi-level Framework. Wir evaluieren die verschiedenen kombinierten Techniken und bereiten weitere Forschung und Entwicklung vor.
In unseren Experimenten haben wir festgestellt, dass unsere grundlegende Implementierung eines spektralen Refiners die Lösung in 71% der Instanzen verbessert. Das spektrale Refinement verbesserte die Gesamtlösung durchschnittlich um 4%. Wir haben ebenfalls festgestellt, dass die iterative spektrale Partitionierung und spektrales Weight-Balancing keine positiven Auswirkungen auf die Lösung in unseren Datensätzen haben.

Acknowledgments

Many thanks to my supervisors for their much appreciated support and feedback. I also want to thank my supervisors for proposing the subject of this thesis. Working on spectral partitioning includes concepts across the fields of mathematics, algorithmics, software engineering and even physics, and I enjoyed this variety a lot.

Special thanks equally to all my proofreaders and discussion partners (voluntary or not) for their feedback and patience.

Lastly, I would like to mention the people around me for their support and tolerance for experimental working schedules, thank you. Thanks also to Werner the sourdough for nutritional support.

I declare that I have developed and written the enclosed thesis completely by myself.
I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents.
I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 1st July, 2024

Contents

1 Introduction

The task to split a hypergraph into multiple blocks of roughly the same size is referred to as the hypergraph paritioning problem. The partitioning commonly has the objective to cut the least hyperedge weight possible. In practice, this problem occurs from the transformation of concrete problems to a hypergraph representation. Namely, applications e.g. in circuit design, network communication management or logic satisfiability solving use this approach [12]. This leads to a demand on high-performance hypergraph partitioning frameworks that solve large instances in short amounts of time. Different specific applications assign differing priorities to solution quality and runtime. Combinatorical hypergraph partitioning frameworks following the multi-level approach showed to be suitable for either priorization [12].
However, as specialized hardware and increasing computing resources get easily available, alternative performance-intensive approaches recently have rebecome subject to research and development: Some partitioning frameworks following the spectral partitioning approach claim to achieve competitive partitioning results. In particular, the K-SpecPart hypergraph partitioning framework recently outperformed some of the leading partitioning frameworks in terms of quality for small numbers of blocks on specific hypergraph instances from circuit design applications [5]. K-SpecPart relies on a hint partition input and uses a variety of traditional multi-level and tree-based partitioning techniques in combination, however. In consequence, it is difficult to contextualize the results.
This work aims to allow a more differenciated view on spectral partitioning and associated techniques. As K-SpecPart uses an existing solution as input, we embed a spectral partitioning routine as a refinement algorithm into a multi-level framework. This appraoch equally has more benefits: We increase the variety of input instances, since multi-level frameworks call refinement algorithms on multiple differently coarsened versions of the input hypergraph. In addition, we enable the evaluation of the viability of using spectral refinement repeatedly inside a multi-level framework, instead of the consecution of the two approaches. With Mt-KaHyPar as surrounding multi-level framework, we choose a multi-threaded state-of-the-art implementation in terms of both quality and runtime [12].

With this work, we provide a fundamental bipartitioning framework based on a combination of K-SpecPart and Mt-KaHyPar. Our framework focusses on modularity and configurability to enable extensive evaluation and extension.
We also provide a basic parameter configuration as the result of tuning on a variety of instances. This configuration led to improved cuts in comparison to pure Mt-KaHyPar during our experiments.

Finally, we evaluate and discuss the effectiveness of different techniques combined in our framework. We show that weight balancing of the eigenproblem decreases the solution quality and that iterated spectral partitioning has no positive effect on our datasets. In addition, we provide an differenciated outlook to the further research and development on spectral refinement arising from our work.

The remaining chapters now first provide some formal background. Next, we introduce the related concepts we use for our refinement algorithm and then present our implementation. We document our tuning and experimental evaluation and conclude by the discussion of the results and their implications.

2 Preliminaries

The following sections introduce some fundamental definitions and concepts.

Furthermore, this work uses some common mathematical notations. In particular, the characteristic function for sets $S \subseteq X$ is denoted as $\mathbb{1}_S : X \to \{1, 0\}$, whilst $\overrightarrow{1_S} \in \mathbb{R}^n$ with $X = \mathbb{N}$ and $n \in \mathbb{N}$ is defined by $(\overrightarrow{1_S})_i = \mathbb{1}_S(i)$. Vectors only consisting of zeros or ones are noted as $\overrightarrow{0}, \overrightarrow{1} \in \mathbb{R}^n$, respectively. The element-wise product of $u, v \in \mathbb{R}^n$ is written as $u \circ v := ((u)_i \cdot (v)_i)_{1 \leq i \leq n}$. The euclidian length of the vector v is denoted as $|v|$. All subsets of X with a specific cardinality $j \leq |X|$ are included in $\binom{X}{j} := \{S \subseteq X \mid |S| = j\}$. The powerset of S is denoted by $\mathcal{P}(S) =: 2^S$.

2.1 Hypergraph Partioning

A *graph* G is a tuple (V, E) with a finite set of *vertices* $V \subset \mathbb{N}$ and *edges* $E \subseteq \binom{V}{2}$. The vertices incident to an edge are called its *pins*. The vertices to which a vertex $v \in V$ is connected via an edge are called *adjacent* vertices or the *neighbourhood* $N(v) \subset V$. If edges can connect more than two vertices, i.e. $E \subseteq \mathcal{P}(V) \backslash V$, then those edges are called *hyperedges* and the resulting graph is called a *hypergraph*.
Edges and vertices can be *weighted* by weight functions $c : V \to \mathbb{N}$ and $w : E \to \mathbb{N}$. If not specified, the weight functions default to $c = w = 1$. The *degree* $\deg(v)$ of a vertex $v \in V$ is the number of all incident edges. The weighted degree $\deg_w(v)$ is the sum of the weights of all incident edges.
A graph is *connected*, if all vertices are transitively adjacent. If all vertices in a subset of V are adjacent, the subset is called a *clique* or *complete*. A connected graph that loses its connectivity by the removal of any arbitrary edge is called a *tree*.

A *k-way partition* Π of a (hyper-)graph H is an ordered set of k disjoint blocks $P_1, ..., P_k \subseteq V$, i.e. $\dot{\bigcup}_{P_i \in \Pi} P_i = V$. The *cut* of Π is the sum of the weights of all (hyper-)edges that span between multiple blocks, i.e. $\text{cut}(\Pi) := \sum_{e \in E_\Pi} w(e)$ with "cut" (hyper-)edges $E_P = E \backslash \bigcup_{P_i \in / Pi} \{e \in E \mid e \subseteq P_i\}$. For the maximum ratio $(1 + \varepsilon)$ between the weight of a block and the ideal block weight, ε is called *imbalance* of Π. This results in $c(V_i) \leq (1 + \varepsilon)\lceil \frac{c(P_i)}{k} \rceil, 1 \leq i \leq k$.

Finally, the partitioning problem for hypergraphs is to map a tuple (H, k, ε_0) to k balanced blocks P_i, i.e. $\varepsilon \leq \varepsilon_0$, with minimal cut.

2.2 Graph Laplacians

Let $G = (V, E)$ be a graph with n vertices $v_i \in V$.

The weighted *degree matrix* $D_G \in \mathbb{R}^{n \times n}$ of G is defined by the equations $(D_G)_{ii} = \deg_w(v_i)$ and $(D_G)_{ij} = 0, i \neq j$. Moreover, the weighted *adjacency matrix* $A_G \in \mathbb{R}^{n \times n}$ of G is defined by $(A_G)_{ij} = w(\{v_i, v_j\})$, assuming that $w(\{u, v\}) := 0$ if $\{u, v\} \notin E$.
Finally, the weighted *Laplacian* $L_G \in \mathbb{R}^{n \times n}$ of G is defined as $L_G = D_G - A_G$.

As the weighted degree of a vertex is the sum of all incident edge weights, and those edge weights also appear in the respective row and column in the adjaciency matrix, the row and column sums of graph Laplacians equal to zero. In consequence, $L_G \vec{1} = 0$.

2.3 Eigenproblems

An operator $A : \mathbb{R}^n \to \mathbb{R}^n$ is *linear*, if it can be represented by a matrix similarly noted as $A \in \mathbb{R}^{n \times n}$. So $A(x) = Ax$, where $(A)_{ij} := (A(e_j))_i$ with *unit* vectors e_k, i.e. $(e_k)_l := 1$ if $k = l$ and otherwise $(e_k)_l := 0$. Algebraically, $\mathbb{R}^{n \times n} \cong \{A \mid A : \mathbb{R}^n \to \mathbb{R}^n \text{ linear}\} \subset (\mathbb{R}^n)^{\mathbb{R}^n}$. The *identity* operator $\mathbb{1} : x \mapsto x$ corresponds to the identity matrix $\mathbb{1}$ with $(\mathbb{1})_{ii} := 1$ and $(\mathbb{1})_{ij} := 0$ for $1 \leq i, j \leq n; j \neq i$. The sum of the diagonal entries $(A)_{ii}, 1 \leq i \leq n$ of a matrix $A \in \mathbb{R}^{n \times n}$ is called its *trace*.

If a vector $x \in \mathbb{R}^n \backslash \{0\}$ matches the condition $A(x) = \lambda x$ for a $\lambda \in \mathbb{R}$ and a linear operator $A : \mathbb{R}^n \to \mathbb{R}^n$, then x is called *eigenvector* of A with corresponding *eigenvalue* λ.

The problem to find such *eigenpairs* (x, λ) is called the *eigenvalue problem* (or abbreviated as *eigenproblem*). The set of all eigenvalues of a matrix is called its *spectrum*. Eigenproblems can be generalized such that (x, λ) shall fulfill the equation $Ax = \lambda Bx$ for operators $A, B \in \mathbb{R}^{n \times n}$.

A matrix $A \in \mathbb{R}^{n \times n}$ and its represented operator are *symmetric*, if A is equal to its *transpose* A^T, where $(A^T)_{ij} := (A)_{ji}$ for all $1 \leq i, j \leq n$. Two vectors $u, v \in \mathbb{R}^n$ are called *orthogonal*, if $u^T v = 0$. If operators $A, B \in \mathbb{R}^{n \times n}$ are symmetric, there are n eigenpairs with mutually orthogonal eigenvectors v_i. This follows from the spectral theorem [3]. A symmetric matrix is called *positive definite* if all of its eigenvalues are positive, and positive *semi-definite* if they are at least non-negative. If a symmetric matrix A is *diagonally dominant*, i.e. $(A)_{ii} \geq \sum_{j \neq i} (A)_{ij}$ for all i, it is also positive semi-definite [10]. Strict diagonal dominance equally implies strict positive definity.

2.4 Numerical Problem Solving

The problem to find $x \in \mathbb{R}^n$ such that $Ax = b$ for $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$ can be classified i.a. by its *condition*. This is a measure of how much small changes on the parameters A and b affect the result x. Algorithms to solve a problem $Ax = b$ can be classified by their

stability. The stability of an algorithm measures how much small errors on the input result in magnified errors on the output of the algorithm. Hence, a high condition for a problem leads to even higher output error ranges when combined with unstable algorithms.

To reduce the impact of ill-conditioned problems, a problem $Ax = b$ can be *preconditioned*. In its simplest form, preconditioning is a transformation of the problem with matrices $P, T \in \mathbb{R}^{n \times n}$, where $PT = \mathbb{1}$. Consequently, effective preconditioning requires preconditioners that result in problems $ATy = b, Px = y$ with lower condition than the original.

3 Related Work

In order to prepare the discussion of the spectral refinement algorithm 4, this chapter firstly introduces the spectral partitioning approach 3.1 we use for the refiner, then the surrounding multi-level architecture 3.2 and eigensolving 3.3 core concepts.

3.1 Spectral Hypergraph Partitioning Basics

In order to solve a hypergraph partitioning problem with spectral methods, the problem must be first transformed to an eigenproblem. The following section describes how this transformation can be achieved 3.1.1 and showcases some relevant spectral partitioning frameworks 3.1.2.

3.1.1 The Spectral Partitioning Approach

Intuitively, the spectral approach to the partitioning problem can be illustrated as follows: An unweighted chain graph, i.e. $(V, \{\{v_i, v_i + 1\} | 1 \leq i \leq |V| - 1\})$, can be interpreted as a simplified model for a fixed uniform string, e.g. of a guitar. The string can vibrate or *oscillate* in different *modes*, i.e. it forms $l + 1$ *nodes* and l *antinodes* for $l \in \mathbb{N}$. Nodes are points where the string does not move and antinodes are areas between two nodes where the string oscillates. As all antinodes have the same length, they can be interpreted as a balanced partition of the string. A way to calculate the modes of an oscillator is to compute the eigenvectors of its Laplacian.
This analogy between partitioning and mode calculation can be equally generalized to more complex graphs: The k-way partitioning problem can be solved by calculating the k^{th} mode of an oscillator counterpart of a graph and interpreting the antinodes as partitions. The oscillator consists of the vertices of the graph as identical masses connected by ideal springs according to the edges of the graph. The partitioning result of the interpreted mode calculation is equivalent to the interpreted k^{th} eigenvector of the graph's Laplacian [7].
More formally, for a connected graph $G = (V, E)$ with Laplacian $L_G \in \mathbb{R}^{n \times n}$ and a vector $x \in \{0, 1\}^n$, the expression $x^T L_G x$ equals to the cut of a bipartition $\Pi = (P_0, P_1)$ of G induced by x with $P_p = \{v_i \in V \mid (x)_i = p\}$. This is because by definition $(L_G x)_i = \sum_{u \in N(v_i) \cap P_0} w(\{u, v\})$ for $(x)_i = 1$ and therefore $v_i \in P_1$, thus:

$$x^T L_G x = x^T (L_G x) = \sum_{v \in P_1} \sum_{u \in N(v) \cap P_0} w(\{u, v\}) = \text{cut}(\Pi) \tag{3.1.1}$$

Note that $\{\{u, v\} | u \in N(v) \cap P_0\}$ is distinct for each $v \in P_1$ and that 3.1.1 also holds for $x \in \{-1, 0\}^n$.
Relaxing the habitat space of x to $\mathbb{R}^n$ with $|x| = 1$, it can be shown that $x^T L_G x$ is minimized by the eigenvectors v_i of L_G and their corresponding eigenvalues λ_i scaled by $v_i^T v_i$ [7]. By definition, $L_G \vec{1} = \vec{0}$, so $\min_{x \in \mathbb{R}^n} x^T L_G x = v_1^T L_G v_1 = 0$ and the first non-trivial minimizer is the "Fiedler" vector v_2 with $v_2^T L_G v_2 = v_2^T \lambda_2 v_2$, i.e. $x^T L_G x \geq v_2^T v_2 \lambda_2$ for $x \in \mathbb{R}^n \backslash \{\vec{0}\}, x^T \vec{1} = 0$.
The naive spectral partitioning approach now consists of approximating the optimum v_2 by defining $(x)_i = \text{sign}((v_2)_i) \in \{-1, 1\}$ [7], justified i.a. by the oscillator analogy.
The lower bound for this approximation states $\text{cut}(\Pi) \geq \frac{1}{4} |V| \lambda_2$. This follows since $x^T x = n = |V|$ for $x \in \{-1, 1\}^n$. Also, interpreting $x \in \{-1, 1\}^n$ as $x = x_0 + x_1$ with $x_0 \in \{-1, 0\}^n$ and $x_1 \in \{0, 1\}^n$ results in $x^T L_G x = 4 \cdot \text{cut}(\Pi)$, now defining $P_p = \{v_i \in V \mid (x_p)_i \neq 0\}$. This is because by definition, $L_G \vec{1} = \vec{0}$ and $x_0 = x_1 - \vec{1}$, so $x = 2x_1 - \vec{1}$.

3.1.2 Spectral Partitioning Frameworks

Partitioning frameworks using the spectral approach like Sphynx and K-SpecPart have recently shown competitive performances both in speed and quality [5] [1]. Sphynx focusses on multi-GPU usage, whilst K-SpecPart produces particularly high-quality solutions for small k's.

Sphynx directly generates a partitioning solution from a set of eigenvectors. Due to its GPU approach, it shows particularly competitive runtimes, but does not achieve the same quality as leading partitioners.

K-SpecPart in turn consists of an iterative architecture fed by an externally computed initial solution. In each iteration, a solution candidate is generated from a newly reformulated eigenproblem. The solution candidates are resused in the next iteration and then overlayed to create the overall output solution. This approach showed higher qualitys than some leading partitioners on the specific benchmark sets in [5], but cannot compare in terms of runtime and was only tested for small k's.

3.2 Multi-Level Hypergraph Partitioning

The *multi-level paradigm* is one of the most successful approaches to hypergraph partitioning in terms of quality and runtime [12]. This section presents its architecture, common refinement algorithms and frameworks implementing the multi-level paradigm. It consists of three phases:
At first, the input hypergraph is *coarsened*. This phase aims to iteratively approximate the original hypergraph with significantly smaller hypergraphs creating a sequence of hypergraph instances. The resulting coarsened version of the original hypergraph allows an

initial partition without significant quality losses but reduced running time of the actual partitioning algorithms like *recursive bipartitioning* or *direct k-way partitioning*. After the initial partitioning as second phase, its resulting initial solution is iteratively projected back to the original hypergraph instance during the third *uncoarsening* phase. Each intermediate solution is *refined* before being projected up to the next level, i.e. the hypergraph next in size calculated in the coarsening phase. The refinement of a partition consists of the consecutive execution of multiple algorithms, each trying to improve the current partitioning solution. Some common refinement algorithms are introduced in 3.2.1.

More details about a state-of-the-art implementation of the multi-level paradigm can be found in [12].

3.2.1 Refinement Algorithms

Three state-of-the-art refinement algorithms are currently used in Mt-KaHyPar, all implemented in a parallel manner: Label propagation, *Fiduccia-Mattheyses* (FM) and flow-based refinement [12].

One of the simplest and fastest refinement algorithms is *label propagation*. It works by moving each vertex to the block that results in the smallest cut and has enough weight capacity left. This procedure is repeated multiple times. Label propagation is not able to perform actions that increase the cut, even if that action would enable smaller cuts afterwards. Still, label propagation can reduce the runtime of later refinement by performing a first simple refinement in low runtime.

Another family of common refinement algorithms is the FM algorithm. Here, a number of possible moves is ordered by its effect on the cut and then applied in that order if allowed by the balance constraint. The applied sequence is then reverted back to the index with the smallest cut. Those two steps are repeated a number of times. FM is able to tolerate actions increasing the cut and hence escaping local minima due to the backpropagation step.

A refinement technique with significantly higher running time is *flow*-based refinement: The refinement problem is transformed to a *maximum flow* problem by creating a directed graph with *source* and *sink* vertices. All edges are attributed with *capacities*. Then, more and more vertices are identified as source or sink after subsequently solving the maximum flow problem for the current graph. Those sources and sinks are then used to create a bipartition. If the bipartition is balanced, the algorithm stops. This approach is able to find complex refinement options due to the global nature of maximum flow problems.

Another parallelizable and fast global approach, currently not used by Mt-KaHyPar, is *hill-scanning* refinement [20]. It is conceptually based on FM, but only performs unfavourable moves when a later amelioration can be predicted in advance. This makes the backpropagation in FM obsolete.

3.2.2 Multi-Level Frameworks

There exist numerous multi-level hypergraph partitioning frameworks. This work aims to evaluate the spectral partitioning approach as a refinement algorithm on Mt-KaHyPar, which is one of the leading multi-level solvers in particular in terms of refinement and distributed computing [12]. Mt-KaHyPar follows the traditional multi-level scheme whilst heavily applying distributed computing techniques like multi-threading with shared memory on each step.

As this work follows the spectral partitioning approach of the K-SpecPart framework [5], it equally takes into account K-SpecPart dependencies (h)METIS and TritonPart to better compare performances. A discussion about this can be found in 6.1.

The METIS graph partitioning framework family also follows the multi-level scheme [15] and stays a recent subject to research and development with high performing results [9] [21]. There exist i.a. the variant hMETIS for hypergraph partitioning.

TritonPart is a partitioning tool developed for integrated circuit design. Its FM-refinement algorithm showed to be particularly effective for the use in spectral partitioning [4].

3.3 Eigenproblem Solvers

As the eigenvectors computed for spectral partitioning are only used as indicator (see 3.1.1), a method approaching the eigenvectors iteratively is best-suited as eigensolver. Like this, the solver can be limited to a number of iterations. Even without having definitely converged, the partial result can still be used for further processing.

Matrix-free solving methods lend themselves to avoid issues storing the graph Laplacian's of large instances. Such methods only require some functional effects of the matrices. Those effects typically include the matrix-vector-multiplication or getting the diagonal of the matrix. For graph Laplacians, those effects can be algorithmically expressed: The diagonal is a vector consisting of the degrees of all vertices, the matrix-vector-multiplication is described in section 4.1.1.

To accelerate convergence, some eigensolvers allow preconditioning via matrices or likewise via their effects.

To avoid the trivial first eigenpair $(\vec{1}, 0)$ of Laplacians, it is useful to exploit the functionality of eigensolvers that can take into account a *deflation space*, i.e. a set of vectors to which computed solutions shall be mutually orthognal.

3.3.1 Eigensolver Algorithms

A widely-used eigensolving algorithm that meets all features mentioned above is called *Locally Optimal Block Preconditioned Conjugate Gradient* (LOBPCG) [8]. Sphynx and K-SpecPart i.a. show its suitability for spectral partitioning applications [1] [5].

LOBPCG iteratively minimizes the trace of a block version of the posed eigenproblem, i.e. for a query for d eigenpairs of $A, B \in \mathbb{R}^{n \times n}$, LOBPCG solves $\min_{X^T BX = \mathbb{1}} \mathrm{trace}(X^T AX)$ for $X \in \mathbb{R}^{n \times d}$. Most LOBPCG implementations require the parameter matrices including the preconditioner to be positive definite.

The basic LOBPCG algorithm has been subject to multiple enhancements and variations i.a. in terms of stability and parallelism [8] [22].

Alternatives to LOBPCG such as the Lanczos method come with drawbacks such as e.g. no preconditioning functionality and do not provide significant advantages in terms of runtime and memory in large scale applications [2].

3.3.2 LOBPCG Implementations

A scalable, native and well-documented implementation of the original LOBPCG is provided by the SLEPc eigensolving programming interface for the C and C++ programming languages [17]. An alternative implementing a recently enhanced variant of LOBPCG, also for C++, can be found in the Anasazi package of the Trilinos software collection [8].

Another native implementation is included in SciPy [6], referenced by the Julia language's LOBPCG interface [19]. This version was successfully used for K-SpecPart [5].

4 Spectral Refinement Algorithm

This chapter describes our spectral refinement algorithm adapted from K-SpecPart [5]. Algorithm 1 shows the overall framework.

Algorithm 1: Spectral Refinement Framework

`Data`: $H = (V, E), \Pi = \{P_0, P_1\}$
`Result`: Π'
remove degree-zero vertices from H
$H^* \leftarrow$largest connected component of H
$P \leftarrow$preconditioner for H^*
$L_{H^*} \leftarrow$Laplacian operator for H^*
$L_{G_W} \leftarrow$weight balance graph Laplacian operator for H^*
$S \leftarrow$solution candidate list
push processed version of Π to S
`repeat`
    $L_{G_H} \leftarrow$hint graph Laplacian operator for end of S
    $B \leftarrow$combine L_{G_W} and L_{G_H}
    $x_1, ... \leftarrow$non-trivial eigenvectors for (L_{H^*}, B) using P
    generate solution for H^* and push to S
retrieve solution Π^* from S
include Π^* in Π'
set Π in Π' for $H \backslash H^*$
distribute degree-zero vertices to Π'

For the solution generation, we implement different variations: The first variation is *tree distilling*, another variation is to perform *Fiedler distilling* on X_1. To retrieve the final solution Π^* if S contains more than one partition in algorithm 1, we overlay all solutions in S.

4.1 Eigenproblem

The following section explains the composition of the generalized eigenproblem used for spectral partitioning in [5] and its implementation. For this, let $H = (V, E)$ be a hypergraph instance with *instance size* $I = \sum_{e \in E} |e|$.

To avoid computation issues due to graph Laplacian matrices being only positive semi-definite, we add a small positive offset (also called *definity shift*) to the diagonal entries, what makes the matrices diagonally dominant and thus strictly positive definite.

We use an external LOBPCG implementation to obtain the requested number of non-trivial eigenvectors by passing $\vec{1} \in \mathbb{R}^{|V|}$ as deflation space. To obtain a preconditioner, we use the *Combinatorical Multigrid* (CMG) algorithm implementation shipped with K-SpecPart.

4.1.1 Hypergraph Representation

Naively defining Laplacians for hypergraphs would result i.a. in loosing the characteristic property of row and column sums being zero, i.e. $L_H \vec{1} \neq \vec{0}$. To avoid this, we use a *clique expansion* graph $G = (V, E')$ of H as proposed by [5]. G is constructed by transforming each hyperedge $e \in E$ into a clique in G, i.e. $u, v \in e \Rightarrow e' := \{u, v\} \in E'$. The weight $w(e')$ is set to $\frac{w(e)}{|e|-1}$, which restores the zero row and column sum property of the Laplacian L_G. This leads to a rather dense Laplacian, so we only use its matrix-vector-multiplication as functional effect for the eigenproblem.

We express the clique expansion graph Laplacian's effect as described in the following: Multiplying a vector $x \in \mathbb{R}^n$ with the Laplacian L_G of a clique expansion graph G of a hypergraph H component-wisely results in $(L_G x)_i = x_i \sum_{u \neq v_i} w(\{v_i, u\}) - \sum_{j \neq i} x_j w(\{v_i, v_j\})$. Rewriting this in function of the individual edges, revectorizing and simplifying gives:

$$
\begin{aligned}
L_G x &= \left(x_i \sum_{e \in E} (|e|-1) \frac{w(e)}{|e|-1} \mathbb{1}_{\{i \in e\}} - \sum_{e \in E} \left(x^T \vec{1}_e \frac{w(e)}{|e|-1} - x_i \frac{w(e)}{|e|-1} \right) \mathbb{1}_{\{i \in e\}} \right)_{1 \leq i \leq n} \\
&= \sum_{e \in E} \left((|e|-1) \frac{w(e)}{|e|-1} x \circ \vec{1}_e - \left(x^T \vec{1}_e \frac{w(e)}{|e|-1} \vec{1}_e - \frac{w(e)}{|e|-1} x \circ \vec{1}_e \right) \right) \\
&= \sum_{e \in E} \frac{w(e)}{|e|-1} \left(|e| (x \circ \vec{1}_e) - (x^T \vec{1}_e) \vec{1}_e \right)
\end{aligned}
$$

This can be reformulated to a more "algorithmic" notation as follows:

$$
L_G x = x \circ \sum_{e \in E} |e| \frac{w(e)}{|e|-1} \vec{1}_e - \sum_{e \in E} x^T \vec{1}_e \frac{w(e)}{|e|-1} \vec{1}_e \tag{4.1.1}
$$

To implement this, we first accumulate the sums in 4.1.1 component-wisely. Finally, we compute the resulting main term vectorially. Using parallel-safe vector data structures, both the accumulation and the final computing can forthrightly be parallelized.

K-SpecPart implements a slightly different variant that leads to different weights in the graph represented by the resulting Laplacian. We test their variant in 5.

To use an explicitly computed Laplacian for preconditioning, we approximate G with the number of edges reduced to $2|I|$: Instead of replacing a hyperedge with a complete clique,

this clique is approximated with the sum of multiple cycles through the randomly permuted pins. Using two cycles per hyperedge produces "an essentially optimal sparse spectral approximation" [5, p. 6].
We use this approximation of the combinatorical Laplacian as input for the CMG algorithm to compute the preconditioner.

For the approximated clique expansion graph Laplacian, we use the implementation of [5] to obtain an adjaciency matrix of the approximated clique expansion graph and then "laplacianize" it. We implement this laplacianization by calculating a row's sum, then negating it and finally setting the diagonal entry to the previously computed row sum. Again, using parallel-safe data structures enables a straight-forward parallelization of the laplacianization.

4.1.2 Vertex Weight Balancing

To take into account the vertex weights, K-SpecPart generalizes the eigenproblem with the Laplacian of a weight balance graph $G_W = (V, E_W := \binom{V}{2})$ as second operator parameter, i.e. $L_G x = \lambda L_{G_W} x$. As shown in 3.1.1, a Laplacian's matrix-vector-product is closely related to the cut induced by the vector. In result, λ is minimized with vectors that induce a partition Π with maximal cut on the second operator parameter's graph. Thus, the weight of $e \in E_W$ set to the product of the weights of its pins enforces a maximization of the product of the total weights of $P_b \in \Pi$, i.e. $c(P_0) \cdot c(P_1)$. Balanced partitioning solutions equally maximize this product. As the weight balance graph is complete, its Laplacian is fully dense and we again use an implicit representation as matrix-vector-product:

Multiplying a vector $x \in \mathbb{R}^n$ with the weight balance graph Laplacian component-wisely results in $(L_{G_W})_i = (x)_i c(v_i) \sum_{j \neq i} c(v_j) - c(v_i) \sum_{j \neq i} (x)_j c(v_j), v_i \in V$. This can be reformulated with $(c)_i := c(v_i)$ as follows:

$$L_B x = (c \circ x)\, c^T \vec{1} - \left(c^T x\right) c = c \circ \left(\left(c^T \vec{1}\right) x - \left(c^T x\right) \vec{1}\right)$$

We implement this by precomputing the scalars resulting from the two vector-vector products and then assembling the final result vectorially, what can be parallelized using adequate datastructures.

4.1.3 Hint Graph

K-SpecPart finally adds the Laplacian of a hint graph $G_H = (V, E_H)$ to the second operator parameter to emphasize similarities to an existing partition solution $\hat{\Pi} = \{P_0, P_1\}$, i.e. $L_G x = \lambda(L_{G_W} + L_{G_H})x$: Again, minimal cuts on G correspond to maximal cuts on the second parameter's graph. So for each vertex in $P_b \in \Pi$ we add an edge to all other vertices in $V \backslash P_b$ to E_H. In consequence, the hint graph Laplacian consists of approximately $\frac{k-1}{k}$ non-zero entries proportionately. We express the matrix-vector-product as follows:

Multiplying a vector $x \in \mathbb{R}^n$ with the hint balance graph Laplacian component-wisely results in $(L_{G_H}x)_i = (x_i|N(v_i)| - \sum_{v_j \in N(v_i)} x_j)_i$. This can be rewritten vector-wisely:

$$L_{G_H}x = x \circ \left(\vec{1}_{P_0}|P_1| + \vec{1}_{P_1}|P_0|\right) - \vec{1}_{P_0}\left(x^T\vec{1}_{P_1}\right) - \vec{1}_{P_1}\left(x^T\vec{1}_{P_0}\right)$$

Once again, we implement this by precomputing the vector-vector products and then assembling the main result vectorially, what can equally be parallelized.

4.2 Solution Generation

This section wraps up techniques required to generate a partitioning solution for a connected hypergraph instance $H = (V, E)$ from a set of the first m non-trivial eigenvectors $x_i, 1 \leq i \leq m$ of the eigenproblem described in the previous section.

Solution generation by tree distilling consists of first generating tree partitions and then overlaying them to get a solution. Fiedler distilling is described in the following.

4.2.1 Fiedler Distilling

For the direct evaluation of the eigenproblem computation, we generate a partitioning solution $\hat{\Pi}$ from the Fiedler eigenvector x_1 by splitting it at the optimal index. To implement this, we first sort V by the corresponding entries in x_1 and put all vertices in P_0. We then add vertices to P_1 in order of x_1 until P_1 reaches a valid block weight. After this, we search for the best cut with valid balance: we continue adding vertices to P_1 in order of x_1 until P_1 reaches the maximum allowed block weight, whilst updating the maximum cut improvement. To compute the current cut improvement, we use the local gain computation implementation of [12]. Finally, we reset $\hat{\Pi}$ to the state of the step with the highest cut improvement.

4.2.2 Tree Partitioning

The K-SpecPart approach to generate a partitioning solution consists of partitioning different trees whose count is exponentially related to m. Those trees have the same vertices V and have edge weights that "'summarize' the underlying cuts" [5, p. 6]. The goal is to create easy-to-partition approximations of H that still give high quality solutions due to an incorporation of the x_i.

The first tree type is called *path graph*. The vertices of V are chained in order of their value of x_i for $1 \leq i \leq m$, what gives m separate trees.

The remaining two types of trees generated are *spanning* trees, i.e. consisting of a subset of E. This requires H to be connected. Both spanning tree types are derived from

an approximation $\hat{G}$ of the clique expansion graph of H as described in 4.1.1, with edges reweighted to a distance metric between its pins. The distance metric is calculated by interpreting $((x_i)_v)_{1\leq i\leq d}, v \in V$ as vector in $\mathbb{R}^d$ and calculating the euclidian distance between the two vectors representing the pins of an edge. The eigenvectors x_i are combinatorically combined to all $2^m - 1$ non-trivial combinations, hence d being the number of eigenvectors in the current combination. We generate two trees for each of those $2^m - 1$ combinations.

The first spanning tree type is a so called *low stretch* spanning tree (LSST) that preserves the edge weights in $\hat{G}$: the sum of the weights of the edges in the LSST between two pins of an edge in $\hat{G}$ equals approximately the edge weight in $\hat{G}$. We then reweigh the LSST such that all edge weights equal to the cut induced by the removal of this edge. In result, finding the minimal balanced partition of the LSST induces a partitioning solution on H with equal properties.

The second spanning tree type is a *minimal* spanning tree (MST). MSTs minimize their total edge weight. We reweigh the MSST in the same way as the LSST. The goal of generating a MST is to compensate for suboptimal LSSTs, as LSSTs and MSTs may perform "complementary" according to [5].

To partition the trees, [5] use their *linear tree sweep* algorithm implementation and METIS. The generated partitions are then refined using the TritonPart FM-algorithm [4].

We use the implementation of [5] for the tree partitioning component used in algorithm 1.

4.2.3 Solution Overlay

For a set $\mathcal{S}$ of partitioning solutions of H, [5] create a single *solution overlay* in the following way:

At first, we remove all hyperedges $E_{\hat{\mathcal{S}}} \subseteq E$ cut by any $\Pi \in \hat{\mathcal{S}}$ for a subset $\hat{\mathcal{S}} \subseteq \mathcal{S}$ of the partitions with the highest qualities. The remaining connected components are then *contracted* with $E_{\hat{\mathcal{S}}}$, i.e. we replace all components by single vertices with accumulated weight. The hyperedges between two components are equally accumulated and set between the representing vertices. As the resulting hypergraph enables all partitioning solutions from $\hat{\mathcal{S}}$, its optimal cut is less or equal to the smallest cut induced by a $\Pi \in \hat{\mathcal{S}}$ and - if balanced - induces a valid partitioning solution on H.

To partition the contracted hypergraph, [5] show that *integer linear programming* (ILP) solvers like Google's OR-Tools [11] can solve the optimization problem with adequate speed and quality for the use on small constracted instances. The resulting partition is then refined using the FM-algorithm. As proposed by [5], we use the TritonPart FM implementation [4]. We express the hypergraph partitioning problem $(H = (V, E), k, \varepsilon_0)$ as optimization problem with the implementation shipped with K-SpecPart [25]:

For each block number $i \leq k$, [5] create boolean variables for all hyperedges $e \in E$ and vertices $v \in V$ that represent the exclusive affiliation of the respective entity to a block. We ensure balance constraints such that the weights of the verticex enabled for a block number

do not surpass the maximum allowed block weight in respect to ε_0. To get a valid partition, each vertex is only allowed to have exactly one variable set to $\mathtt{true}$ for each block number. To prepare the objective function, we add constraints for all hyperedges, such that they can only belong to a block if all pins also belong to that block. The objective function is set to be conceptually dominated by the sum of the hyperedge weights whose pins belong only to one partition. Hence, maximizing the objective function results in a small cut.

We take the maximum quality solution from all input partitions and the newly generated partitioning solution as the result of the overlay algorithm.

4.3 Preprocessing and Postprocessing

As our refinement algorithm requires a connected hypergraph instance, we perform additional steps to assure the connectedness.

We remove all vertices that do not belong to any hyperedge, i.e. having degree zero, before the start of the algorithm. After the algorithm has terminated, we then reinsert them, whilst greedily distributing them to the blocks. For this, we use the implementation of [12].

To assure the connectedness having all degree-zero-vertices removed, we perform *island isolation*, i.e. partition only the largest connected component of the input instance. As final result, we reuse the input hint partition for vertices not belonging to the partitioned component. We use the island isolation implementation of [5].

5 Experimental Evaluation

To evaluate spectral partitioning as a refinement algorithm, we firstly test and tune our refiner implementation, before evaluating specific scenarios and the overall performance of the enclosing partitioning solver including our refiner. The raw results can be found on our KIT gitlab repository [1].

5.1 Experimental Setup

This section documents the environment we used for the experiments in terms of hardware, software and datasets.

We ran the experiments on machines with 16 CPUs (11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz with 8 cores) and 64GB memory, or 64 CPUs (Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz with 16 cores) and 512GB memory.

For the implementation of the refinement algorithm, we extend the Mt-KaHyPar framework written in C++ by a custom refiner class extending the IRefiner interface. The refiner class then natively calls Julia code at runtime via the C interface provided by Julia [18]. In turn, Julia code calls external dependency binaries (h)METIS, ilp_part and TritonPart via the shell execution interface provided by Julia.
We provide installation instructions for the dependencies at our Mt-KaHyPar branch [2].

We use three sets of instances: (1) a set of regular graphs i.e. with roughly homogenous vertex degrees, (2) a set of irregular graphs and (3) a set of hypergraphs from the dissertation of Schlag [24]. Datasets (1) and (2) come from the compression of texts [14][23], from the 10th DIMACS implementation challenge and from the medium benchmark set from the dissertation of Heuer[13].
Dataset (1) includes 33 instances with 15 thousand to 18 million vertices, dataset (2) consists of 38 instances ranging from 5000 to 4 million vertices and dataset (3) includes 100 instances with 29 thousand to 500 thousand vertices.

[1] `https://gitlab.kit.edu/unkjx/bt/-/tree/main/experiments/export`
[2] `https://github.com/kahypar/mt-kahypar/tree/julian/spectral-refinement`

5.2 Experiments

We first led experiments to tune instance-independent parameters of the eigenproblem and then evaluated the solution generation from the eigenvectors. Finally, we evaluated specific configurations and compared the performance of our tuned framework with pure MtKaHyPar calls.

For the tuning of the eigenproblem, we executed the main loop in algorithm 1 once and used Fiedler-distilling as solution generation. We found the following results:
Firstly, we tuned the number of LOBPCG iterarations. Figure 5.1 shows that more than 40 iterations do not result in significant quality benefits, so we chose 42 iterations for the remaining experiments.
Then, we tested if approximating graphs with the random cycle technique used for hyper-

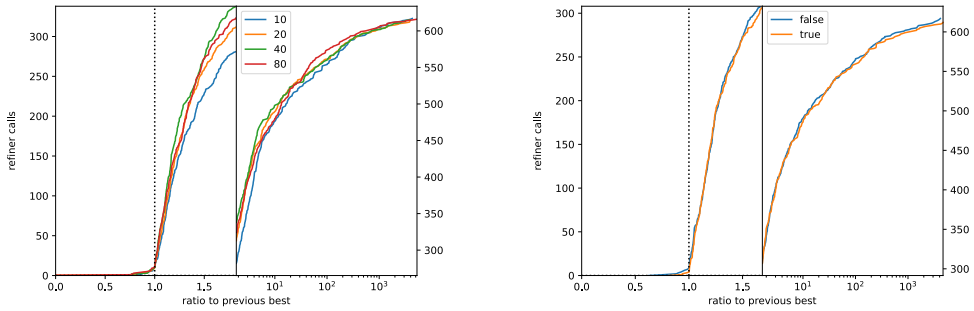

Figure 5.1: Performance profiles of refiner calls for different numbers of maximum LOBPCG iterations (left) and for the option of approximating graph Laplacians with random cycles (right) on datasets (1) and (2).

graphs would affect the quality or robustness of the algorithm and found it to make no difference, as shown in 5.1. This proves the correctness of the approximation algorithm, as the approximation is exact for hyperedges with two pins.
Moreover, we tested by how much the Laplacian matrix diagonals should be shifted for the matrices to be positive definite. The definity shift should lead to less failures of LOBPCG, but in turn should not decrease the quality. We tested five positive values for the definity shift of the implicit eigenproblem Laplacian operators and the explicit preconditioner Laplacian matrix separately. Figure 5.2 shows the results: Shifting the Laplacian operators by 0.1 leads to the most successful refiner calls on datasets (1) and (2) and does not reduce the quality. Slight quality increases may result from more successful LOBPCG runs, so we do not conclude a significant improvement. No definity shift at all meanwhile led to LOBPCG failure on every single call. For the explicit Laplacian, we found no clear result in terms of quality, even though we ran the experiment on all datasets. In terms of robustness, any shift on the preconditioner led to roughly 10% less LOBPCG failures.
Next, we evaluated different ratios between the weight balance operator and the hint opera-

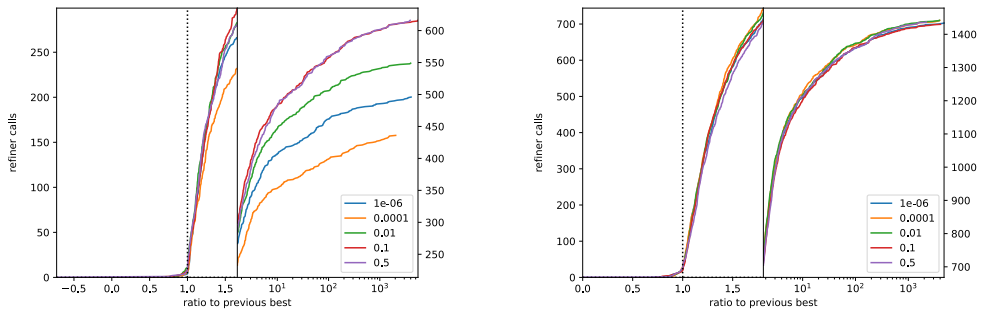

Figure 5.2: Performance profiles of refiner calls on datasets (1) and (2) for different definity shifts on the eigenproblem parameter operators (left) and on the preconditioner (right).

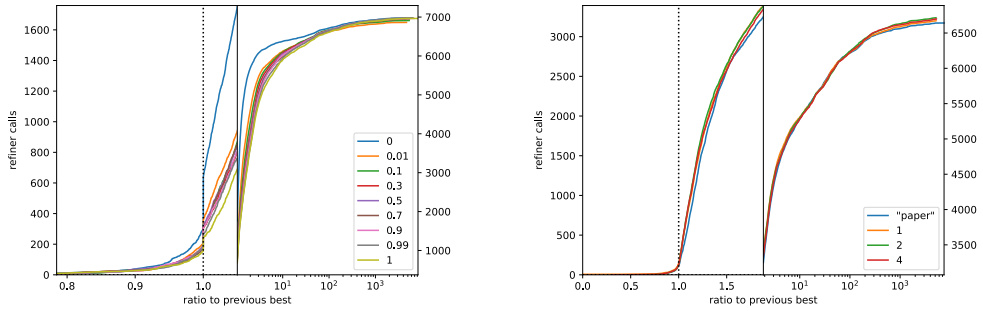

Figure 5.3: Performance profiles of refiner calls on all datasets for different ratios between weight balance and hint operator (left) and for different variants of the hypergraph Laplacian operator (right). The operator resulting from a ratio r is $rL_{G_W} + (1-r)L_{G_H}$. The "paper" variant represents the naive implementation of the combinatorical hypergraph clique expansion graph, the numerical variants are treated as parameter for the hypergraph operator computation implemented in K-SpecPart.

tor. The results in figure 5.3 show that any fraction of the weight balance operator decreases the quality. Only using the hint operator even results in more runs that find a solution superior to the hint.
Another tuning experiment was to compare different variants of the hypergraph Laplacian operator. We compared our naive implementation with different parameterizations of the K-SpecPart implementation in figure 5.3. In result, the K-SpecPart variants show minimal benefits in robustness and quality, but not related to the input parameter.

To evaluate the solution generation, we used algorithm 1 with tree distilling and five random seeds per instance. This led to the following results:
For the number of eigenvectors used, we found more than two eigenvectors to show a decrease in robustness and no benefit in quality (see 5.4). This equally aligns with the default parameter configuration of K-SpecPart [5].

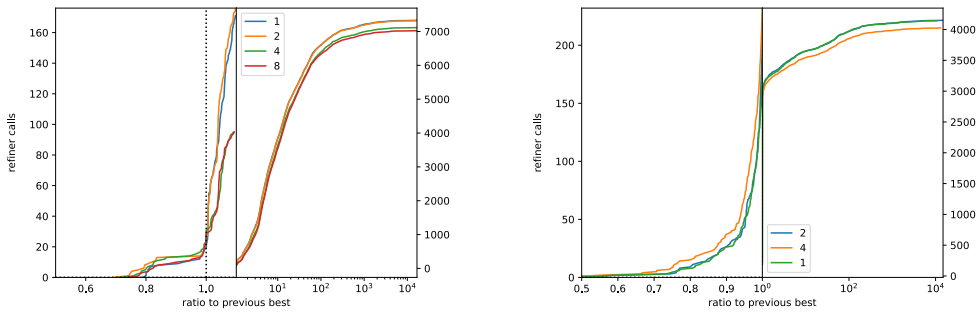

Figure 5.4: Performance profiles of refiner calls on all datasets for different numbers of eigenvectors generated (left) and for different numbers of solutions used for overlays (right).

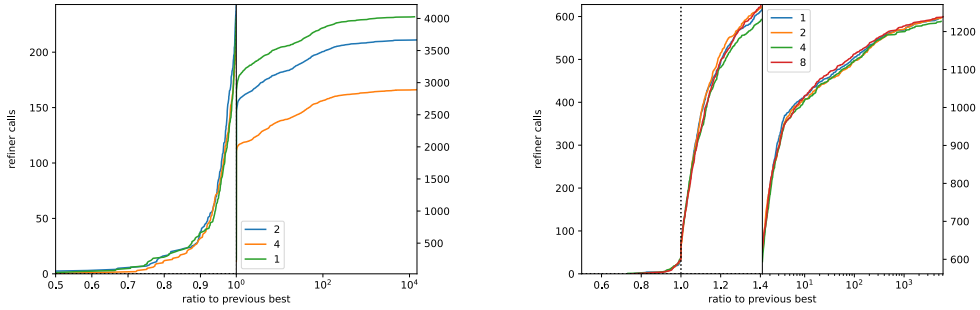

Figure 5.5: Performance profiles of refiner calls for different numbers of iterations of the main loop: with tree distilling as solution generation on all datasets (left) and with simple fiedler distilling on datasets (1) and (2) (right).

For the number of solutions to be used for the overlay algorithm, we found picking the four best solutions to lead to the best quality, but to decrease the robustness due to timeout limits in our experiments.
The number of iterations in algorithm 1 did not show significant effects on the solution quality in our experiments displayed in 5.5. To further evaluate the iterated computation of eigenvalues, we tested different numbers of iterations with Fiedler distilling as solution generation in algorithm 1. For the final result of algorithm 1, we took the solution of the last iteration instead of an overlay. We again found no significant improvement of the solution quality with increasing numbers of iterations.

Combining the above results, we achieved an average improvement of 4% for 71% of all instances. Figure 5.6 shows the differenciated results for the different instance categories. Our tuned refinement algorithm led to cut average improvements of 5.5% in 6.1% of the refiner calls. Those improvements appear on high coarsening levels, as the distribution in figure 5.6 displays.

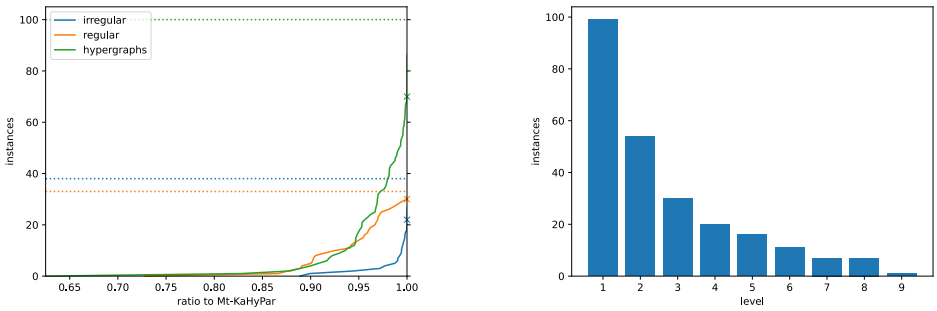

Figure 5.6: Performance profiles of the the tuned partitioner for the different datasets (left) and the number of improvements found on each coarsening level (right).

6 Discussion

After the experimental evaluation, we now discuss the results and their implications before outlining future work on spectral refinement.

Our tuning experiments led to a viable configuration although we did not extensively evaluate the available configuration space. During the experiments, we found weight balancing and multiple iterations in algorithm 1 not to have significant effects on the solution quality in our implementation. This may be caused by our interpretation of the K-SpecPart dependencies, however, as K-SpecPart used the CPLEX ILP-solver instead of Google's OR-tools and the TritonPart FM-algorithm could not be reproduced identically from the resources delivered by K-SpecPart. Another cause may be the limited number of configurations we tested.

6.1 Future Work

As this work only provides the starting point for the research on spectral refinement embedded inside a multi-level partitioning framework, this section aims to present the leads and approaches for further research and implementation collected over the course of this work.

Some direct improvements to the refinement algorithm 1 include i.a. better handling of unconnected hypergraphs. This is because the current implementation only partitions the largest component due to i.a. the tree generation requiring connectedness. A recursive approach could be easily implemented, an approach exploiting the unconnectedness possibly leads to much smaller cuts, however. This could be realized by only partitioning some components in a possibly imbalanced way and affiliating the parts to the weight of sufficiently small unpartitioned components. A more lightweight approach to tackle unconnected hypergraphs could be to run a traditional refinement algorithm on the final partitioning solution including the removed components of the hypergraph.

One of the main limitations of the current implementation is its constraint to bipartitioning. Extending the implementation to a multi-way variant following the approach of K-SpecPart [5] requires additional steps and distinction for a partitioning problem $(H, k, \varepsilon_0), k > 2$: At first, the initial k-way hint solution needs to be processed to create k bipartitions. For each of those bipartitions, a separate bipartitioning eigenproblem is solved. The concatenated eigenvector solutions are then reduced to the equivalent of one set of eigenvectors by a dimensionality reduction algorithm. During the tree distillation, the tree bipartitioning

needs to be extended to k-way partitioning. Moreover, k-way partitioning offers objective metrics differing from simple cut minimization. The suitability of the K-SpecPart approach for those metrics is still to be evaluated.

Another obvious approach to improve the current implementation is performance optimization. Due to this work's focus on solution quality, the current implementation by far not fully exploits efficient and optimized programming techniques. In particular, the inter-process communication between Mt-KaHyPar and its Julia descendant could be optimized by using further speed-up methods like the precompilation of Julia code or the usage of fast TLS [18]. Another approach would be to implement the whole refinement algorithm natively in C++, what the current implementation partially prepares. However, the high effort of performance optimizations and particularly of a C++-reimplementation needs to be justified by further extensive experimental evaluation at first (see 6.1.3). We furthermore discuss performance optimization in terms of runtime in 6.1.2.

6.1.1 Variation of External Tools

The external components introduced by K-SpecPart, namely eigensolver, refiner, ILP-solver and full partitioning solver can be variated in terms of implementation or type.

Using a different implementation for external tools can lead to improvements in terms of computational performance or quality. Using e.g. the respective highly optimized FM and ILP components already implemented for Mt-KaHyPar [16] and recursive Mt-KaHyPar calls instead of (h)METIS possibly leads to relevant computational performance improvements. As [5] suggests, quality improvements are harder to predict as we need to take into account the suitability of an implementation for the specific problem.

The usage of other types of tools, e.g. to replace the FM refinement used in our implementation by other refinement approaches such as flow-based refinement or hill-climbing refinement, also could affect the quality, as the refinement plays a relevant role for restoring the balance after an overlay.

The choice and tuning of the LOBPCG implementation (see 3.3.2) highly influences the robustness and quality of the algorithm and thus needs further evaluation. This work's first approach was to use the native SLEPc implementation, that turned out to be hard to tune and lacking the functionality of easiliy accessing the current intermediate solution when early stopping without convergence. We then used the Julia implementation because it showed to be suitable in K-SpecPart, but it still has some stability issues. Those currently present the main cause for the refinement algorithm to fail. Using a more robust and tuned version like the one provided by the Anasazi package would possibly improve the robustness of the whole implementation. Further tuning of LOBPCG to the special application as has been done e.g. for [1] could equally influence the choice of the LOBPCG implementation. Finally, a LOBPCG variant that allows preconditioning via matrix effects rather than explixit matrices could reduce both memory consumption and runtime.

6.1.2 Runtime

As this work focusses on solution quality, runtime is a substantial aspect to be investigated in future works. Runtime considerations are necessary in multiple ways:

To be able to use spectral partitioning as a refinement algorithm in runtime-sensitive applications, it needs to be much faster. This is because even much more extensive studies (see 3.2.2 and 3.1.2) on spectral partitioning including runtime considerations as the one led for the K-SpecPart framework report long runtimes compared to multi-level partitioners such as Mt-KaHyPar. The GPU-approach of frameworks like Sphynx tackles the runtime issue, but does not achieve the same quality as K-SpecPart. With a differentiated evaluation of the runtime potential of the different techniques, the use of some of them in an isolated manner could be considered. For example, the overlay algorithm is independent of the spectral components and could be applied to the results of different refinement solutions in a traditional multi-level environment.

To improve the overall runtime and fit spectral partitioning in the Mt-KaHyPar framework, parallelization techniques should be applied to each step of the refinement algorithm 1 as extensively as possible. We repeatedly included parallelization approaches in this work, but not further implemented and evaluated them due to comparability, stability issues and the focus on solution quality.

The usage of GPU-based computation techniques like in Sphynx could also lead to viable runtimes for the use inside leading traditional paritioning frameworks.

6.1.3 Further Experimental Evaluation

As already mentioned, extensive experimental evaluation plays an important role for strategic decisions on how to further develop the spectral refinement approach - this work only provides some first results (see 5). Aspects to be evaluated include i.a. the following:

The configuration space outlined in 5.2 can be further explored and evaluated onto full extent to possibly achieve higher quality, robustness and speed. This includes direct parameter variations such as the number of cycles for the clique expansion approximation and equally more extensive studies on the relation between the parameters, as the parameters were mostly assumed to be independent for our tuning experiments. A particularly promising experiment would be to further increase the number of solutions overlayed, as we only tested up to four solutions.

Further experimental evaluation is equally needed to test the different components in terms of their runtime and contribution to the final solution quality. We provided first results on the contributions, but did not extensively evaluate i.a. the refinement steps or the ILP-solving.

More sophisticated evaluation of the behaviour of different instance categories could equally lead to heuristic variation of the refinement algorithm with potentially improved results.

Moreover, this could lead to heuristics when to apply spectral partitioning inside a multi-level partitioning call.

Another approach to improve the quality of our results is to lead experiments on larger datasets with larger numbers of seeds.

Finally, the data we collected during our experiments can be further evaluated, particularly for relations between the performance of different components.

6.2 Conclusion

With this work, we further proved the potential of combining spectral partitioning approaches with traditional multi-level partitoning techniques. The productive use of spectral refinement inside a state-of-the-art multi-level partitioner still requires more evaluation and runtime-optimizing development.

Nevertheless, we provided a theoretical overview on spectral partitioning, documented our implementation and its experimental evaluation and found our basic spectral refinement implementation to improve the results of the Mt-KaHyPar framework for 71% of our dataset instances. Our combined partitioning framework enables further extension and evaluation without extensive implementation effort. Hence, this work provides justification and an environment for future research and development on spectral partitioning in the context of a multi-level partioning framework.

Bibliography

[1] Seher Acer, Erik G Boman, Christian A Glusa, and Sivasankaran Rajamanickam. Sphynx: A parallel multi-gpu graph partitioner for distributed-memory systems. Parallel Computing, 106:102769, 2021.

[2] Peter Arbenz, Ulrich L Hetmaniuk, Richard B Lehoucq, and Raymond S Tuminaro. A comparison of eigensolvers for large-scale 3d modal analysis using amg-preconditioned iterative methods. International Journal for Numerical Methods in Engineering, 64(2):204–236, 2005.

[3] Sheldon Axler. Linear algebra done right. Springer Nature, 2024.

[4] Ismail Bustany, Grigor Gasparyan, Andrew B. Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. An open-source constraints-driven general partitioning multi-tool for vlsi physical design. In 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), pages 1–9, 2023.

[5] Ismail Bustany, Andrew B. Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. K-specpart: Supervised embedding algorithms and cut overlay for improved hypergraph partitioning, 2023.

[6] The SciPy community. Scipy lobpcg. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.lobpcg.html`.

[7] James Demmel. Spectral partitioning, 1999. `https://people.eecs.berkeley.edu/~demmel/cs267/lecture20/lecture20.html`.

[8] Jed A. Duersch, Meiyue Shao, Chao Yang, and Ming Gu. A robust and efficient implementation of lobpcg. SIAM Journal on Scientific Computing, 40(5):C655–C676, 2018.

[9] Ghizlane Echbarthi and Hamamache Kheddouci. Streaming metis partitioning. In 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pages 17–24, 2016.

[10] Semyon Aranovich Gershgorin. Uber die abgrenzung der eigenwerte einer matrix. Bulletin de l'Académie des Sciences de l'URSS., 6:749–754, 1931.

[11] Google. Or-tools | google for developers. `https://developers.google.com/optimization?hl=en`.

[12] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. ACM Trans. Algorithms, 20(1), jan 2024.

[13] Tobias Heuer. Scalable High-Quality Graph and Hypergraph Partitioning. PhD thesis, Karlsruher Institut für Technologie (KIT), 2022.

[14] Artur Jez. "Faster Fully Compressed Pattern Matching by Recompression". In: ACM Trans. Algorithms 11.3 (2015), 20:1–20:43. doi: 10.1145/2631920.

[15] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Computer Science & Engineering (CS&E) Technical Reports, 1997.

[16] ITI KIT. Mt-kahypar. `https://github.com/kahypar/mt-kahypar/`.

[17] A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov. Block locally optimal preconditioned eigenvalue xolvers (blopex) in hypre and petsc. SIAM Journal on Scientific Computing, 29(5):2224–2239, 2007.

[18] The Julia Programming Language. Embedding julia. `https://docs.julialang.org/en/v1/manual/embedding`.

[19] The Julia Programming Language. Julia linear algebra: Lobpcg. `https://iterativesolvers.julialinearalgebra.org/stable/eigenproblems/lobpcg/`.

[20] Dominique LaSalle and George Karypis. A parallel hill-climbing refinement algorithm for graph partitioning. In 2016 45th International Conference on Parallel Processing (ICPP), pages 236–241. IEEE, 2016.

[21] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. G-kway: Multilevel gpu-accelerated k-way graph partitioner. 61st ACM/IEEE Design Automation Conference (DAC '24), 2024.

[22] Yu Li, Hehu Xie, Ran Xu, Chun'guang You, and Ning Zhang. A parallel generalized conjugate gradient method for large scale eigenvalue problems. CCF Transactions on High Performance Computing, 2:111–122, 2020.

[23] Pizza&Chili Corpus (Compressed Indexes and their Testbeds). http://pizzachili.dcc.uchile.cl/index.html. Accessed: 2024-07-01.

[24] Sebastian Schlag. High-Quality Hypergraph Partitioning. PhD thesis, Karlsruher Institut für Technologie (KIT), 2020. 46.12.02; LK 01.

[25] TILOS-AI-Institute. Tilos-ai-institute/hypergraphpartitioning: Hypergraph partitioning: Benchmarks, evaluators, best known solutions and codes. `https://github.com/TILOS-AI-Institute/HypergraphPartitioning/tree/main/K_SpecPart/ilp_partitioner`.