universität wien

# MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

## „Engineering NOI-based Coarsening Algorithms for Multilevel Graph Partitioning“


verfasst von / submitted by

Jakob Niedermüller, BSc


angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2020 / Vienna, 2020

| | |
|---|---|
| Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet: | A 066 910 |
| Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet: | Computational Science |
| Betreut von / Supervisor: | Univ.-Prof. Dr. Monika Henzinger |
| Mitbetreut von / Co-Supervisor: | Dipl.-Math. Dipl.-Inform. Dr. Christian Schulz |

## Abstract

The graph partitioning problem asks for dividing a graph into $k$ blocks of vertices while minimizing the size of the edge cut and while maintaining a balance constraint on the block size. Graph partitioning is relevant for many real world applications such as reducing complexity of a traffic network[4, 16] or enabling parallel processing of a certain graph.

Real world graphs, such as social networks, often count millions of vertices and edges and are, thus, too large to compute a direct solution. Moreover, the problem of partitioning a graph exactly is NP complete and no constant factor approximation algorithms exist. Consequently, heuristics like multilevel graph partitioning[11] are used. A state-of-the-art multilevel partitioning framework is provided by *Karlsruhe High Quality Partitioning* (KaHIP)[11, 27]. This multilevel scheme consists of three main phases. In the *coarsening* phase the graph is reduced by iteratively contracting edges while maintaining the overall structure of the graph and without effecting the size of the cut of the final partitioning too much. In the *initial partitioning* phase the graph is small enough to be directly partitioned. In the *uncoarsening* phase the previously contracted edges are uncontracted.

We engineer new coarsening algorithms using KaHIP as a partitioning framework. Our coarsening methods build on the *CAPFOREST* (CF)[21, 22] algorithm developed by *Nagamochi-Ono-Ibaraki* (NOI). CF calculates an edge rating that acts as a heuristic to determine whether the eligible edge can be contracted without effecting the minimum cut size. We incorporate this edge rating into a coarsening procedure, which we, thus, refer to as NOI-Coarsening. We develop four variants of NOI-Coarsening. Three of them directly contract edges based on the calculated CF edge rating. The fourth variant incorporates the CF edge rating into the *Size-Constraint Label Propagation* (SCLaP) algorithm[19].

We evaluate our coarsening algorithms by performing parameter tuning and comparing them with KaHIP's state-of-the-art partitioner. For benchmarking we partition a set containing social network and web graphs as well as a set of more traditional meshlike graphs from technical and physics applications. We evaluate both, quality and running time. In general, NOI-Coarsening methods do not improve over state-of-the-art performance on social networks and web graphs but fall behind to some degree. On high $k$ partitionings ($k = 64$) our best method comes as close as a median 8 percent difference in terms of quality. On the set of rather meshlike graphs we achieve state-of-the-art bipartitioning at only half the running time.

## Zusammenfassung

Das Graphpartitionierungsproblem besteht darin einen Graphen in $k$ Blöcke zu teilen, wobei die Größe des Kantenschnitts minimal sein soll und eine Gleichgewichtsbedingung in Bezug auf die Blockgröße eingehalten werden muss. Graphpartitionierung ist darüber hinaus für viele reale Anwendungen relevant. Beispielsweise um die Komplexität von Verkehrsnetzen[4, 16] zu reduzieren oder um das parallele Prozessieren eines bestimmten Graphen zu ermöglichen.

Graphen aus realen Anwendungen, wie sozialen Netzwerke, bestehen oftmals aus Millionen Knoten und Kanten und sind somit zu groß um eine direkte Lösung zu berechnen. Darüber hinaus ist das Graphpartitionierungsproblem NP-vollständig und es existiert dafür kein Näherungsalgorithmus mit konstantem Faktor. Folglich müssen Heuristiken wie Multilevel-Graphpartitionierung[11] verwendet werden. Ein solches Multilevel-Graphpartitionierungs-Framework nach modernstem Stand wird etwa im *Karlsruhe High Quality Partitionin* (KaHIP) Softwarepaket[11, 27] verwendet. Dieses Multilevel-Schema besteht aus drei Hauptphasen. In der *Coarsening*-Phase wird der Graph durch das iterative Kontrahieren von Kanten geschrumpft, während die grobe Struktur des Graphen beibehalten werden soll und ohne dabei die Größe des Kantenschnitts der letztendlichen Partitionierung zu stark zu beeinflussen. In der sogenannten *Initial Partitioning*-Phase ist der Graph bereits klein genug um direkt partitioniert zu werden. In der *Uncoarsening*-Phase werden die zuvor kontrahierten Kanten wieder dekontrahiert.

Wir entwickeln neue Coarsening-Algorithmen, wobei wir KaHIP als Partitionierungs-Framework verwenden. Unsere Coarsening-Methoden bauen auf *CAPFOREST* (CF) [21, 22] auf - einem Algorithmus von *Nagamochi-Ono-Ibaraki* (NOI). CF berechnet eine Kantenbewertung, die als Heuristik dient um zu entscheiden ob eine Kante kontrahiert werden kann ohne dabei den minimalen Kantenschnitts zu vergrößern. Wir beziehen diese Kantenbewertung in ein Coarsening-Schema ein, das wir daher als NOI-Coarsening bezeichnen. Insgesamt entwickeln wir vier Varianten von NOI-Coarsening, drei davon kontrahieren direkt basierend auf der Kantenbewertung. Die vierte Variante basiert auf dem *Size-Constraint Label Propagation* (SCLaP) Algorithmus[19], inkludiert jedoch die CF-Kantenbewertung.

Wir evaluieren unsere Coarsening-Algorithmen, indem wir die Parameter optimieren und die Algorithmen mit KaHIPs State-of-the-art-Partitionierungsprogramm vergleichen. Als Benchmark partitionieren wir einen Satz an Sozialen Netzwerk- und Webgraphen, sowie einen Satz an traditionelleren mehr gitterartigen Graphen aus technischen und physikalischen Anwendungsbereichen. Wir evaluieren sowohl Qualität als auch Ausführungszeit. Im Allgemeinen kommen die NOI-Methoden bei den sozialen Netzwerk- und Webgraphen nicht über die State-of-the-art-Performance hinaus, sondern fallen eher etwas zurück. Bei Partitionierung mit hohem $k$-Wert ($k = 64$) kommt unsere beste Methode im Median bis auf 8 Prozent Unterschied heran, was die Schnittqualität betrifft. Auf dem Satz an gitterartigen Graphen erreichen wir bei der Bipartitionierung State-of-the-art-Performance in Bezug auf Qualität bei nur halber Ausführungszeit.

## Acknowledgement

I am thankful to Dr. Alexander Noe and Prof. Christian Schulz, who gave me the necessary insight for this thesis and provided valuable feedback in numerous meetings.

My thanks also go to Prof. Monika Henzinger, without whom this work would not have been possible.

I would also like to thank my friends who gave me motivation when it was needed the most.

Ultimately, I thank my parents without whose support I would never have come this far.

# Contents

# 1 Introduction

## 1.1 Motivation

Many real world problems can be modeled as *graphs*. Prominent examples are social networks [23], traffic networks [4, 16] and circuit design [14] but also molecular interactions in biological systems [34] can be represented as graphs. In fact there can be found a plethora of instances in nature, technology and abstract concepts, such as language, that yield graph-like structures. By abstracting physical or conceptual instances to graphs, actual application problems can be solved using computer science and mathematics. Such real world instances very often are too large to compute a solution directly. In order to handle such graphs, *graph partitioning* is performed to reduce complexity or to parallelize the actual problem solving step.

In general, graph partitioning solves the fundamental mathematical problem of dividing a graph into smaller roughly equally sized pieces that are inter-connected as loosely as possible. A typical example where this is needed is route planning [4, 16]. Without partitioning the road network first, route calculation would take significantly longer for more distant destinations. Moreover, from computational science perspective scientific simulations [28, 30] make thorough use of graph partitioning. Probably most importantly, partitioning enables efficient parallel and load balanced processing of graphs. Imagine we want to process a graph on a multiprocessor system with $k$ *processing elements* (PEs). By partitioning the graph, we divide the graph into $k$ blocks of about equal size. Thus, the single blocks fit into the RAM and the CPU load is balanced. The particular PE (blocks) have minimum communication (edges) to other PE, so the parallelization overhead is minimized.

Since graphs involved in real world applications become extremely large (social networks consist of up to millions of nodes and billions of edges [15]) computationally efficient and scalable graph partitioning is highly relevant. Since the problem is NP-complete and there is no constant factor approximation algorithm [1, 2], heuristics like multilevel graph partitioning are used in known software packages such as KaHIP [11, 27], METIS [13, 29] and SCOTCH [3] The overall scheme for multilevel partitioning, as it is used in *Karlsruhe High Quality Partitioning* (KaHIP), is depicted in figure 1.1. This approach consists of three main phases. In the *contraction* (coarsening) phase (which is the phase this thesis will focus on) we take the input graph $G := (V, E)$ and identify a subset $M \subseteq E$ that presumably can be contracted while still maintaining the overall global structure of the graph and without effecting the final cut size by a large margin. The identified edges in $M$ are contracted and the procedure is repeated until the number of nodes $|V|$ falls below a pre-defined

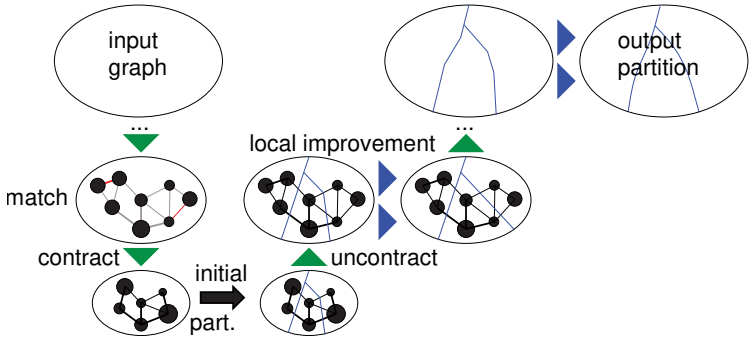



Figure 1.1: Multilevel Graph Partitioning [11]

threshold. Contraction should quickly reduce the size of the input and each computed level should reflect the global structure of the input network. In particular, nodes should represent densely connected subgraphs. In the second phase the graph is small enough to be directly partitioned, which otherwise would be computationally very expensive. After the *initial partitioning* phase, the previously contracted edges are iteratively uncontracted. In this *refinement* (uncoarsening) phase, after each uncontraction iteration, nodes are moved between blocks to improve the cut size or balance of the partitioning. A more detailed outline of how KaHIP works is given in [11].

Coarsening via *graph clustering* algorithms, namely *Size-Constraint Label Propagation* (SCLaP) with *ensemble clustering*, achieves state-of-the-art performance [19] when partitioning social networks. Due to the promising results of this approach we engineer a cluster coarsening algorithm that uses a more complex edge rating incorporating information about the connectedness of two clusters beyond one single edge.

## 1.2 Contribution

Within the scope of this thesis we engineer a cluster coarsening algorithm based on the edge rating provided by *CAPFOREST* (CF)[21, 22] algorithm using KaHIP [11, 27] as framework. We refer to the method as *Nagamochi-Ono-Ibaraki* (NOI)-coarsening. In each level CF does a *breadth-first-search* (BFS) calculating a lower bound $q(e)$ for the connectivity $\lambda(e)$ of each edge in the graph. A high value of $q(e)$ suggests a high probability that this edge can be contracted (i.e the incident nodes were assigned to the same block) safely without effecting the cut size of the partitioning. Both, CF and SCLaP find clusters of strongly connected nodes. With a complexity of $\mathcal{O}(m + n \log n)$ with significantly lower run times in practice, CF is also promising run time performance wise, even though label propagation has a complexity of only $\mathcal{O}(m + n)$[19].

The evaluation shows that NOI-Coarsening does not improve quality over state-of-the-art methods in general. For social network graphs we come close (median

8 percent) to state-of-the-art performance when performing high $k$ partitioning ($k = 64$). On rather traditional meshlike graphs we improve state-of-the-art run time performance while being on par in terms of quality. Also noteworthy is that NOI-Coarsening can achieve very fast coarsening on social network graphs when $k$ is high. Our benchmarks showed a 1.2 to 4 times faster running time for this NOI configuration while finding a 50 percent larger cut in the worst case.

## 1.3 Thesis Structure

In the following chapter (2) we will define preliminary concepts of graph theory and provide the necessary notation. In Chapter 3 we will give an overview over related work and explain relevant concepts. In Chapter 4 we present the theory of the engineered NOI-coarsening algorithms. In Chapter 5 we will present the experimental results. In Chapter 6 we draw conclusions on the findings within this work and give an outlook for future work. At the end of this thesis you find the appendices containing algorithms closer to actual implementations and additional results.

# 2 Fundamentals

In this chapter we will introduce the reader to the basic concepts of graph theory with focus on graph partitioning. Moreover, we provide the notation used throughout this thesis.

## 2.1 Concepts of Graph Theory

A *graph* $G$ is defined by a set of *nodes* or *vertices* $V$ and a set of *edges* $E$, where each edge $e$ is denoted as the pair of nodes $\{u, v\} \subseteq V$ that is connected by the edge. For an *undirected* graph it holds that $e = \{u, v\} = \{v, u\}$, i.e. all edges are bidirectional. A *weighted* graph generalizes the presented definition by the mappings $c : V \to \mathbb{N}$ assigning a weight $c(v)$ to each node $v$ and $\omega : E \to \mathbb{N}^*$ assigning a weight $\omega(e)$ to each edge $e$. We can extend $\omega(\cdot)$ to sets by defining $\omega(E') := \sum_{e \in E'} \omega(e)$, where $E' \subseteq E$. Analogous to edge weights we can extend $c(\cdot)$ to sets by defining $c(V') := \sum_{v \in V'} c(v)$, where $V' \subseteq V$. An unweighted graph can be considered as a special case where each edge and vertex has a weight of one.

We denote a weighted graph as $G := (V, E, c, \omega)$. The *neighborhood* $\Gamma(v)$ of a vertex $v$ is the set of nodes *adjacent* to $v$, i.e. $\Gamma(v) := \{u : \{u, v\} \in E\}$. The number of *incident* edges (or number of neighbors) of a vertex $v$ is called *degree* of $v$, formally $\deg(v) := |\Gamma(v)|$. The weighted degree adds up the weights of the incident edges: $\deg(v) := \omega(\{\{u, v\} \in E\})$. If not stated otherwise, the reader can assume that *deg* denotes the weighted version throughout this text. The smallest resp. largest weighted degree out of all vertices within a graph is called *minimum degree* resp. *maximum degree*. The average degree is calculated by $\frac{2|E|}{|V|}$.

A sequence of edges $(\{v_1, v_2\}, \ldots, \{v_i, v_{i+1}\}, \ldots \{v_{n-1}, v_n\})$ with the corresponding sequence of vertices $(\{v_1, \ldots, v_n\})$ is called a *path* if all vertices (as well as all edges) are pairwise distinct. We refer to a *cyclic* path when we only require pairwise distinctness for edges. We speak of a path in a graph $G := (V, E)$ if all edges (and vertices) of the path lie in $E$ (and $V$). A graph $G := (V, E)$ is called *connected* if every vertex $v \in V$ is reachable from each other vertex $u \in V \setminus \{v\}$ by traversing edges. A *subgraph* of a graph $G := (V, E)$ is a graph $G' := (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. A *connected component* is a subgraph with vertex set $U \subseteq V$ that is connected and where there is no larger connected subgraph $W$ for that holds $U \subset W \subseteq V$. A connected graph has only one component. A *tree* is a connected graph without cyclic paths. Based on this, a *forest* is defined as a graph where each connected component is a tree. A *spanning tree* (or forest) of a graph is a subgraph that is a tree (or forest) and that covers all vertices of the graph. Such a spanning

tree is said to be maximal if it covers all vertices of the graph. *Edge-disjoint* maximum spanning trees (or forests) refers to a set of trees (or forests) that do not share a common edge and cover (when combined) all vertices of the graph.

A *matching* is a set of edges $M \subseteq E$ within a graph $G := (V, E)$ where there is no pair of edges $\{e_i, e_j\}$ in $M$ whose edges $e_i$ and $e_j$ are adjacent to each other. A maximum matching refers to the set $M \subseteq E$ with the largest weight $c(M)$.

*Contracting* an edge $e = \{u, v\}$ means that the vertices of the edge $e$ are replaced by a new vertex $w$, this vertex has the combined weight $c(w) = c(u) + c(v)$ of the original vertices. If by merging such two vertices, there are produced parallel edges $e_1$ and $e_2$, these are also replaced by a new edge $f$ that has their combined weight $c(f) = c(e_1) + c(e_2)$. When speaking of contracting a block of vertices $V_i \subseteq V$, we refer to contracting all edges within that block, resulting in a single super vertex.

A *cut* of a graph $G := (V, E)$ is a bipartition $C = \{S, V \setminus S\}$ of the vertex set $V$. The cut implies a set of edges $F = \{e = \{s, t\} \mid s \in S \wedge t \in V \setminus S\}$ that one has to remove (or "cut through") to render two intra-connected (but not inter-connected) components. We define $\omega : V \to \mathbb{N}$ such that $\omega(S) = \omega(F)$ (combined weight of the cut edges) and refer to this as *capacity* or simply the size of the cut. The cut with the smallest possible capacity is called *minimum cut* $C_{min} = \min_{S \subset V} \omega(S)$. The *edge-connectivity* (or connectivity) $\lambda(G)$ of a graph is the size of the minimum cut whereas the (edge-)connectivity of a pair of vertices $\{s, t\}$ can be defined as the weight of the smallest set of edges whose removal yields a cut $\{S, V \setminus S\}$ with $s \in S$ and $t \in V \setminus S$. Formally: $\lambda(\{u, v\}) = \min_{s \in S \wedge t \in V \setminus S} \omega(S)$.

## 2.2 Graph Partitioning

Similar to the minimum cut problem, where we want to find a bipartition of the graph that is minimal in the sense of cut size, within *graph partitioning* the problem is generalized to a $k$-partition, where $k \in \mathbb{N}^*$. Moreover, balancing of the partition is of interest.

Recall that a $k$-partition $P$ of a set $S$, divides the set into $k$ sets $S_i$ with following properties:

$$S = \bigcup_{i=1}^{k} S_i \quad \forall i \in \{1, \ldots, k\} \tag{2.1}$$

$$S_i \cap S_j \quad \forall i \neq j \in \{1, \ldots, k\} \tag{2.2}$$

$$S_i \neq \emptyset \quad \forall i \in \{1, \ldots, k\} \tag{2.3}$$

The partition then is denoted as

$$P = \{S_1, \ldots, S_k\}. \tag{2.4}$$

Graph partitioning refers to the process of dividing a graph into $k$ subgraphs. Therefore, the set of vertices $V$ is partitioned into $k$ blocks of vertices. Meeting property 2.2, the vertex blocks are mutually exclusive. Edges that run between

vertices of the same block are referred to as *intra-edges* while edges with ends in different blocks are called *inter-edges*. Goal of solving the graph partitioning problem is to find the $k$-partitioning that minimizes the combined weight of all inter-edges while maintaining the balance of the block sizes. I.e. the blocks should have similar or equal combined weight of vertices. For this, we express a *balance constraint* $L$ that determines the allowed deviation of a block weight from the average. The overall scheme can be summed up as follows. We take in a weighted graph $G := (V, E, c, \omega)$, number of blocks $k$ and balance parameter $\epsilon$. We then find the partition

$$P = \{V_1, \ldots, V_i, \ldots, V_k\} \tag{2.5}$$

that minimizes the cut size $\omega(F)$, where $F$ denotes the set of inter-edges

$$F = \{\{v, w\} \in E \mid \exists i \neq j : v \in V_i \wedge w \in V_j\}, \tag{2.6}$$

while maintaining the balance constraint

$$c(V_i) \leq L := (1 + \epsilon)\left\lceil \frac{c(V)}{k} \right\rceil \quad \forall i \in \{1, \ldots, k\}. \tag{2.7}$$

Here, parameter $\epsilon \in \mathbb{R}_{\geq 0}$ is the allowed deviation as proportion of the average block weight.

It is well known that the graph partitioning problem can also be formulated differently like with other objective functions instead of minimizing the cut size. Moreover, the number of blocks as well as balance can be handled dynamically depending on the gain on the objective functions. Within this thesis we will always refer to the version formulated above (Equation 2.5-2.7). More general, in graph partitioning the goal is to identify strongly intra-connected blocks within a graph that are, hopefully, rather independent, i.e. only loosely connected to other identified blocks. This allows, for example, to process a network-like structure on multiple *processing elements* (PE), where each PE processes a block of roughly equal size, having only few dependencies to the other PEs. Graph partitioning is an NP-complete [12, 30] problem, thus for general graphs no constant factor approximation algorithms are available [1, 2, 30] making heuristics for processing large graphs indispensible.

# 3 Related Work

In the following we outline the most relevant research that this thesis is based on. We will give an overview on multilevel partitioning and cluster coarsening. Moreover, we will explain the CAPFOREST algorithm in detail.

## 3.1 Multilevel Graph Partitioning Scheme

An approach that has proven [11] itself to handle partitioning of large real world graphs is multilevel graph partitioning [11]. It is used by partitioning software packages such as *Karlsruhe High Quality Partitioning* (KaHIP) [11, 27], METIS [13, 29] and SCOTCH [3]. The multilevel approach comes down to three levels. As visualized in Figure 3.1, we take in a graph as input *coarsen* it to a smaller graph, that then can be partitioned in the *initial partitioning* phase. In the third stage, the *refinement* phase, the previously coarsened graph is uncoarsened and locally improved towards the objective function. The final output is the partitioned graph.

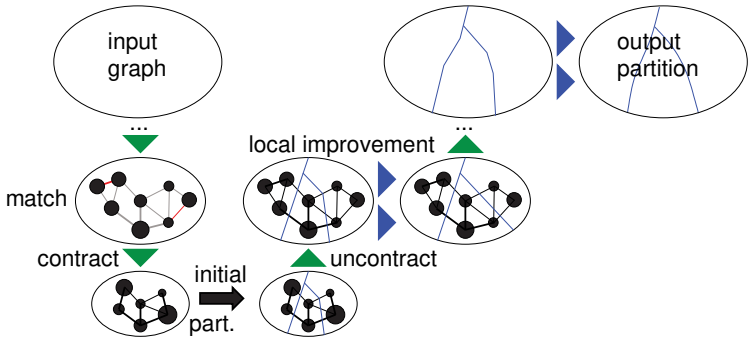



Figure 3.1: Multilevel Graph Partitioning [11]

The algorithm engineered within our work is embedded into the uncoarsening phase of the KaHIP framework. KaHIP's [11, 27] implementation guarantees balance and a fixed number of blocks in the partition. KaHIP comes in different variants focusing on either higher quality ("strong" variant), high run time performance ("fast") or a compromise ("eco"). These variants differ in the choice of the matching algorithm, optimization parameters, refinement strategies and the multilevel iteration scheme. All three variants also come in a "social" version, where cluster coarsening is applied to address social network graphs, i.e. graphs with an extreme high number of edges.

In the following we will outline all three main phases of the used multilevel scheme. For a more detailed overview we refer to [11].

### 3.1.1 Coarsening

In order to make the partitioning task less complex, a relatively large instance $G := (V, E)$ is reduced to a smaller graph $G' := (V', E')$. Coarsening describes the process of iteratively reducing the graph by identifying sets of edges $M \subseteq E$, which are then contracted to reduce the size of the graph. At the end of each iteration, the graph should reflect the global structure of the graph. I.e. contraction should take place in regions of the graph that are densely connected while edges between those dense clusters of vertices should be left untouched. Such, initial partitioning of the coarse graph can assign strongly connected nodes to the same partition.

One way to establish a quantitative measure of whether an edge is eligible for contraction are edge ratings[11]. The simplest way to rate an edge $e$, is using the weight of an edge $\omega(e)$. By contracting edges with large weight, the cut size tends to be reduced. However, more sophisticated ratings are applicable. For instance, such that punish contraction of heavy nodes, for better maintaining of balance, or contraction of edges with many out going edges, whose contraction tends to increase the cut size. Traditionally [11], matchings are then calculated, maximizing the sum over the edge rating $r(e)$. Applicable algorithms are, for instance, *Sorted Heavy Edge Matching* (SHEM) [29], *Greedy Matching* and *Global Path Algorithm* (GPA) [18]. A parallel matching algorithm used by KaHIP is *bisection matching* [11, 17]. The calculated maximum matchings $M$ are then contracted until the graph is small enough for initial partitioning, i.e. till $|V|$ falls below some threshold.[11]

An alternative approach that has been applied within KaHIP is to perform a clustering, such as *Size-Constraint Label Propagation* (SCLaP) [19] (cf. Section 3.2.1). The clustering algorithm identifies blocks of vertices $V_i \subseteq V$ consisting of densely connected nodes. These blocks $V_i$ are then contracted to super vertices. In a similar fashion as with matchings, this is iterated till the graph is shrunk to a manageable size.

### 3.1.2 Initial Partitioning

Within the initial partitioning phase, the coarsened graph is small enough to be quickly partitioned. The partitioning therefore can be repeated with different seeds to further improve the result.[11]

### 3.1.3 Refinement

During the refinement (or uncoarsening) phase, the edges contracted in the coarsening phase are iteratively uncontracted. After each iteration, local search algorithms are applied to further improve the cut while maintaining the balance. I.e. nodes are moved between the boundaries of the partition blocks such that the cut size is decreased or balance is increased.[11]

At the end of the multi-level process, the original graph is restored with vertex set $V$ partitioned as stated in Equations 2.1-2.3.

### 3.1.4 Multilevel Iterations

To further increase the quality, the whole multilevel partitioning can be repeated [26, 32, 33]. Such iterations are called *V-cycles* in reference to the shape of the multilevel scheme depicted in figure 3.1. For more global search strategies used in multi-level partitioning, we refer to [26].

### 3.1.5 KaHIP

## 3.2 Cluster-based Coarsening

For the coarsening phase within multilevel graph partitioning (cf. Section 3.1.1) clustering algorithms can be used to achieve a fast reduction of the graph.[19] Within KaHIP, SCLaP is used in combination with *Ensemble Clustering*.

### 3.2.1 Size-Constrained Label Propagation

SCLaP [19] extends the *Label Propagation Algorithm* (LPA) proposed in [25] by a constraint on block sizes. Basically, Label Propagation iterated over the graph in a random fashion while at each step the currently visited vertex $v$ is assigned the label of the cluster its most strongly connected to. In the size-constraint variant, the label is only assigned if the corresponding cluster is not exceeding a defined upper bound. This is done iteratively till convergence or till some stopping criterion is met (convergence is not guaranteed [19]).

Algorithm 1 shows more thoroughly how SCLaP works. We take in a graph, upper block bound $b$ and a fixed number of runs $m$ (instead of a stopping criterion). First, we initialize blocks (or clusters) as singletons. I.e. every cluster consists of a single vertex. We then perform the label iterations (lines 2-9). Within each iteration, we mark all vertices as *unvisited*. We then traverse the whole graph (lines 4-9) by iteratively picking a random *unvisited* vertex $u$. The vertex is then moved (line 8) to the cluster its most strongly connected to, but only if the size-constraint in line 7 is met. The strength of the connection to a vertex $V_i$ is measured by

$$\omega\big(\{\{u, w\} \mid w \in \Gamma(u) \cap V_i)\}\big). \tag{3.1}$$

Expression 3.1 sums up the weight over all incident edges between the currently visited vertex $u$ and adjacent vertices of a specific block $V_i$. Vertex $u$ is then marked as *visited*. When all vertices where visited and all label iterations are performed, each one of the blocks of the produced clustering are contracted to a super vertex in the coarsened graph $G' := (V', E')$.

By using a size-constraint the balance is maintained throughout the coarsening process. The value of the upper bound $b$ can be defined by $\frac{L}{l}$, where $L$ is the upper

**Algorithm 1:** SCLaP

**Input:** undirected weighted graph $G = (V, E, c, \omega)$, block upper bound $b$, number of runs $m$
**Output:** clustering $\{V_1, \ldots, V_k\}$

```
initialize blocks V_i as singletons {v}   ∀v ∈ V, i ∈ I, where I = {1, ..., |V|}
foreach label iteration 1 to m do
    label all vertices v ∈ V as unvisited
    while there is an unvisited vertex do
        u ← pick a random unvisited vertex v ∈ V
        V_i ← pick a block that maximizes ω({{u, w} | w ∈ Γ(u) ∩ V_i})
        if c(V_i) + c(u) ≤ b then
            move u to the block V_i
        mark u as visited
remove empty blocks from clustering and update index set I accordingly
```

bound for the balance constraint (Equation 2.7) and $l$ is the *coarsening factor*. This coarsening factor determines by which factor the size of the graph shall be reduces at most in one coarsening iteration.

### 3.2.2 Ensemble Clustering

Ensemble Clustering [19, 24] combines multiple *weak* clusterings to one *strong* clustering. Within multi level graph partitioning, Ensemble Clustering is used [19] to combine multiple SCLaP clusterings to one *overlay clustering* of higher quality. Basically, Ensemble Clustering assigns two nodes to the same block in the overlay clustering $O$ only if these two nodes are found within the same cluster in all input clusterings $\mathcal{C}_i$. This principle is illustrated in figure 3.2.

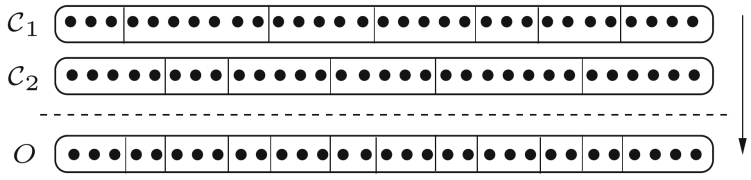



Figure 3.2: Ensemble Clustering [19]

## 3.3 CAPFOREST

An algorithm, crucial for this work, was proposed by Nagamochi et al. in [21] and later optimized in [22]. This algorithm called *CAPFOREST* (CF)[1] was originally developed with the goal to provide an efficient exact minimum cut algorithm. Its capability of finding edges that do not affect the minimum cut makes the algorithm also a powerful tool for graph partitioning.

To give some intuition [7], imagine an (for sake of simplicity) unweighted and connected undirected graph $G$ whose minimum cut has capacity, say, $\lambda = 2$. We then compute a maximum spanning forest on $G$. Since this spanning forest is maximal and the graph is connected, it is also a tree. This spanning tree, by definition, contains at least one edge of the minimum cut. Removing this tree from the graph $G$, the remaining graph $G'$ contains still up to one edge of the minimum cut. Moreover, $G'$ is possibly disconnected. Now calculating a maximum spanning forest on $G'$, this spanning forest contains the remaining edge of the minimum cut if it was not already removed with the first maximum spanning forest. Removing the second spanning forest, we now can be sure that the remainder of the graph $G''$ does not contain the minimum cut edges and, thus, these edges can be safely contracted without affecting the minimum cut. Similarly, if we know that threre is a cut with capacity $\bar{\lambda} = 3$, but we do not know if it is a minimal cut, then we can use the described procedure as a mean for contracting edges that do not affect cuts strictly smaller than $\lambda$.

Formulated more generally, given some known cut capacity $\bar{\lambda}$, computing the first $\bar{\lambda} - 1$ edge-disjoint maximum spanning forests, edges that are not within these spanning forests can be contracted without affecting any cut-edge of a strictly smaller cut. What is left, is to find an initial cut and its capacity. A trivial approach would be to remove some vertex $v$ which yields the cut $\{G \setminus \{v\}\}$ with capacity $\bar{\lambda} = deg(v)$.

CF extends this idea to weighted graphs $G := (V, E, c, \omega)$ by not directly computing all spanning forests but calculating lower bounds $q(e)$ on the connectivity $\lambda(e)$ of each edge $e \in E$. The set of edges with the same $q$-value forms a forest $E_{q_i} = \{e \in E \mid q(e) = q_i\}$. The partition of $E$ into such forests $\{E_{q_1}, \ldots, E_{q_i}, \ldots, E_{q_k}\}$ forms an edge-disjoint maximum spanning forest. More precisely, the set $E_{q_i}$ corresponds to a maximum spanning forest after removing all forests with smaller $q$ values from the original graph $G$. I.e. $E_{q_i}$ induces a maximum spanning forest on $G'(V, E')$, where $E' = E \setminus (E_{q_1} \cup \ldots \cup E_{q_{i-1}})$.

In Algorithm 2 we see the concrete procedure. The algorithm takes in an undirected weighted graph $G := (V, E, c, \omega)$. At the beginning every vertex is labeled as *unvisited* and assigned an $r$-value of zero. We then perform a *breadth-first-search* (BFS) (line 4-10) that traverses the whole graph in a particular order. Within each iteration we *visit* a vertex with largest $r$-value. We then scan (line 4-10) all incident

[1] The name stems from *capacitated forest* because of computing forests within capacitated (i.e. weighted) graphs. It is also the extended version of Nagamochi et al.'s unweighted pendent *FOREST* [21].

edges that lead to an unvisited vertex[2] $w$. The $r$-value of the regarded vertex $w$ is then increased by the weight of the scanned edge $e$. Moreover the current $r$-value is assigned to $e$. This $q(e)$ represents a lower bound on the connectivity $\lambda(e)$ of that edge (proven in [21]). After all incident edges are scanned, the currently *visited* vertex $u$ is marked as such. CF's output are the lower bounds $q(e)$. CF has a run time complexity of $\mathcal{O}(m + n \log n)$ [21].

**Algorithm 2:** CAPFOREST

**Input:** undirected weighted graph $G = (V, E, c, \omega)$
**Output:** lower bounds $q(e)$ on $\lambda(e)$, where $e \in E$

```
1  label all vertices v ∈ V as unvisited
2  r(v) := 0   ∀v ∈ V
3  q(e) := 0   ∀e ∈ E
4  while there is an unvisited vertex do
5      u ← pick unvisited vertex with largest r
6      foreach incident edge e = {u, w}, where w ∈ V do
7          if w is unvisited then
8              r(w) := r(w) + ω(e)
9              q(e) := r(w)
10     mark u as visited
```

After running Algorithm 2, given a known cut with capacity $\bar{\lambda}$, edges with $q(e) \geq \bar{\lambda}$ can again be safely contracted without affecting any strictly smaller cut. To make this more clear, recall that the connectivity of an edge $e = \{u, w\}$ is the size of the smallest possible cut that places $u$ and $v$ on different sides of the cut. With $q(e)$ being a lower bound for this connectivity, we know that an edge with $q(e) \geq \bar{\lambda}$ cannot be part of a cut with a size strictly smaller than $\bar{\lambda}$.

Nagamochi et al. provide an algorithm [21] to compute the minimum cut by performing CF multiple times. The idea is to start off of a known cut and to iteratively improve the cut while contracting the graph. As initial cut size $\bar{\lambda}$ serves the minimum degree $\min_{v \in V} \deg(v)$, which is the capacity of the trivial cut $\{\{v\}, V \setminus \{v\}\}$. After calling CF, all edges with $q(e) \geq \bar{\lambda}$ are contracted and the smallest known capacity $\bar{\lambda}$ is then updated by setting it to $\bar{\lambda} = \min(\min_{v \in V'} \deg(v), \bar{\lambda})$, where $V'$ is the set of vertices of the contracted graph. This process is iterated until size of the graph is reduced to $|V| < 3$. In each at least one edge is contracted, the algorithm thus terminates and the final $\bar{\lambda}$ is the size of a minimum cut of the original graph $G$.[21, 22]

In [22] Nagamochi et al. modify this algorithm such that within each CF BFS they compute a forest of contractible edges, more precisely edges where $q(e) > \hat{\lambda}$.

[2] Note that this is equivalent to scanning all *unscanned* edges, like it is stated in the original papers [21, 22] by Nagamochi et al.

This forest is then contracted in each iteration of the minimum cut algorithm. A state of the art minimum cut solver that uses this optimized version of the algorithm is *Vienna Minimum Cuts* (VieCut) [6, 7, 8, 9, 10].

In context of coarsening we can think of CF's lower bounds $q$ resp. its $r$-values as edge ratings that can be used as a heuristic for the connectivity of an edge (cf. Section 3.1.1). In the following we will refer to methods based on CF as *Nagamochi-Ono-Ibaraki* (NOI)-methods.[3]

[3] In reference to the authors of [22].

# 4 NOI-based Coarsening

## 4.1 Overview

The *Nagamochi-Ono-Ibaraki* (NOI) coarsening algorithm engineered within this work aims to make the multilevel graph partitioning, implemented in *Karlsruhe High Quality Partitioning* (KaHIP)[11, 19, 27], of large graphs and social networks more efficient. KaHIP, basically, takes a graph and parameters specifying number of partition blocks and balancing of the block sizes as input. The program provides a number of partitioning algorithms that iteratively solves the partitioning problem while aiming for the lowest possible cut size, or an approximation to that, under the given constraints. The output is the partitioned graph. For efficiency reasons the partitioning is achieved in a multilevel approach for which the overall scheme is depicted in Section 3.1.

This approach consists of three main phases. In the *contraction* (coarsening) phase (which is the phase, this thesis focuses on) we identify a subset $M \subseteq E$ containing edges that presumably can be contracted while still maintaining the overall global structure of the graph and without effecting the final cut size by a large margin. The edges in $M$ are contracted and the procedure is repeated until the number of nodes $|V|$ falls below a pre-defined threshold. Contraction should quickly reduce the size of the input and each computed level should reflect the global structure of the input network. In particular, nodes should represent densely connected subgraphs. In the second phase the graph is small enough to be directly partitioned, which otherwise would be computationally very expensive. After the *initial partitioning* phase, the previously contracted edges are iteratively uncontracted. In this *refinement* (uncoarsening) phase, after each uncontraction iteration, nodes are moved between blocks to improve the cut size or balance of the partitioning. A more detailed outline of how KaHIP works is given in [11].

The scope of this thesis involves implementing the *CAPFOREST* (CF) algorithm by Nagamochi et al. [21][22] into the *Size-Constraint Label Propagation* (SCLaP)[19] coarsening of KaHIP. SCLaP is a clustering algorithm that iterates, in random order, over all nodes in the graph. For each node the neighbors are scanned and the label of the most strongly connected cluster is assigned to the current node, given that the cluster size does not exceed a pre-defined size constraint. Each cluster is then joint to a super vertex. To build the multilevel hierarchy, we repeat the computing and contracting of the clustering recursively until the number of nodes falls below an empirically found threshold criterion. In the scope of the thesis we use this procedure as framework but substitute the label propagation routine with the CF algorithm.

In each level this algorithm does a *breadth-first-search* (BFS) calculating a lower bound $q(e)$ for the connectivity $\lambda(e)$ of each edge in the graph. A high value of $q(e)$ suggests a high probability that this edge can be contracted safely without effecting the cut size of the partitioning (i.e the incident nodes were assigned to the same block). We present multiple variants using CF's lower bounds on the connectivity as a heuristic to decide whether an edge shall be contracted or not.

The implementation of CF itself within this work is based off the exact minimum cut framework *Vienna Minimum Cuts* (VieCut)[6, 7, 8, 9, 10].

## 4.2 NOI-Coarsening

Hiroshi Nagamochi et al. originally proposed[21, 22] CF within the context of the exact minimum cut problem. Within a minimum cut solver, CF is used as a mean to find edges that can be contracted while maintaining at least one minimum cut. Within coarsening of multilevel graph partitioning, we try to achieve a similar goal: preserving low-connectivity edges that have a higher chance of being inter-block edges in the final partitioning. This is utterly important since when contracting such edges, the cut size is already increased before the initial partitioning even takes place. In order to find such edges, NOI's algorithm[22] runs CF iteratively (cf. Section 3.3). It starts with the *minimum degree* as the smallest known cut $\bar{\lambda}$. In each iteration CF finds edges that can be safely contracted. After the contraction $\bar{\lambda}$ is updated if a smaller cut is found. Since in each execution of CF at least one[22] edge is contracted without affecting the minimum cut, the algorithm finds the minimum cut within $O(n)$ runs of CF. Overall, the implementation presented by NOI comes with a run time complexity of $O(mn + n^2 \log n)$[22].

Due to the nature of graph partitioning, however, the contraction scheme looks entirely different for coarsening. First of all we have to take the balance constraint into account so we have to consider the block sizes before contracting any edge. This also makes the order of contractions relevant. Since once a block is full, incident edges cannot be contracted any more. Furthermore, a highly scalable multilevel approach comes with other demands performance wise and making approximations is a necessity instead of solving the problem exactly. To be competitive we have to have a runtime complexity similar to SCLaP ($O(m + n)$)[19, 25]. Apart from that, instances we are interested in, are usually very large graphs that can have a very low minimum degree and minimum cut, both even can be zero. Thus, the central idea of the minimum cut contraction scheme of iteratively decreasing the lower bound for the minimum cut is not applicable to coarsening.

Instead, we are interested in a greedy approach where we run CF only one time (or a constant number of times) using the lower bound $q(e)$ as a heuristic for connectivity in order to decide whether an edge is eligible for contraction and/or in which order edges shall be contracted. By this, we follow the idea of a cluster coarsening scheme that aims to combine strongly connected groups of vertices. By traversing the graph

only once via the BFS of CF (cf. algorithm 2), the run time complexity is $O(m + n \log n)$[21].

Algorithm 3 shows how the NOI-coarsening routine works. The input of the algorithm is an undirected weighted graph together with an upper bound $b$ for the block size. We start by labeling all vertices as *unvisited* and all edges as *uncontractible*. Moreover, we set the rating function $r(\cdot)$ to zero for all vertices and initialize our blocks (or clusters) as singletons. I.e each block consists of only one single vertex at the beginning. We then iterate over the whole graph analogous to the original CF routine (alg. 2) by Nagamochi et al.[21, 22]. In each iteration of the BFS (line 5 to 13), we pick the unvisited vertex $u$ with the largest rating $r(u)$ (ties are randomly broken). For this vertex, we scan all incident edges that lead to an unvisited vertex and decide whether this edge shall be contracted or not. The predicate used here is left for optimization (cf. Section 4.3). Most importantly, however, we have to guarantee that the contraction does not exceed the block sizes implied by the balance constraint and the coarsening factor (line 9). Therefore we only mark an edge $\{u, w\}$ for later contraction if the combined node weight of the blocks, containing $u$ resp. $w$, does not exceed the upper bound $b$ if the eligible edge is eventually contracted. The blocks containing the nodes of the regarded edge are then merged together. As in algorithm 2, in every BFS iteration, the $r$-values for the unvisited adjacent vertices are updated by adding the weight of the respective incident edge (line 12).

**Algorithm 3:** NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size

**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  label all vertices v ∈ V as unvisited
2  label all edges e ∈ E as uncontractible
3  r(v) := 0   ∀v ∈ V
4  initialize blocks as singletons {v}   ∀v ∈ V
5  while there is an unvisited vertex do
6      u ← pick unvisited vertex with largest r
7      foreach incident edge e = {u, w}, where w ∈ V do
8          if w is unvisited then
9              if combined block weight ≤ b then
10                 mark e as contractible
11                 merge block containing u and block containing w
12             r(w) = r(w) + ω(e)
13     mark u as visited
14 contract all contractible edges
```

So far, executing the algorithm without any optimization will just add single vertices to a block (line 11) till the block is full, i.e. until the upper bound is hit.

The "intelligence" of the algorithm only comes from the order of how the vertices are visited, which is determined by the edge[1] rating $r$. The procedure propagates contractions along the path of highest $r$ values, which in theory should tend towards the pursued goal of contracting primarily high connectivity edges. However, along those paths any adjacent edge is also contracted regardless of its $r$-value. This comes as a major caveat since this is likely to negatively influence the final cut of the partitioning. Nevertheless, this bare version of NOI-Coarsening will serve as a basis for engineering a more mature coarsening algorithm in the following sections.

## 4.3 Optimizations

Starting from algorithm 3, an effective way of changing how edges are contracted is to add a predicate, which has to be fulfilled before an edge is marked as *contractible*. Nagamochi et al., in their papers ([21, 22]) introduce the predicate $r(w) + \omega(e) \geq \min_{v \in V} \deg(v)$ in the context of their contraction scheme for finding the minimum cut. They show[21, 22] that contracting such edges does not effect the minimum cut (cf. Section 3.3 for intuition). Generally speaking, within coarsening this is not as powerful as within the exact minimum cut problem as there is no strict guarantee that excluding edges not fulfilling the predicate will improve the solution (in terms of the quality of the partitioning). However, by not contracting edges that are of relatively low weight, it should increase the likelihood of a good partitioning due to more high connectivity edges within the partition blocks. Another restriction in comparison to the minimum cut problem is that we also have to handle graphs with a minimum degree of zero in which case the predicate has no effect at all. Nevertheless, it is an optimization that comes almost for free, run time performance wise, and that, theoretically, should have a significant impact when the minimum degree is relatively large.

In other words, one can imagine this approach as one strategy to set a threshold on the edge rating $r$ that determines if an edge is contracted. Another way would be to simply set an empirical threshold or to define a rule. As an example by establishing the rule that in each BFS iteration of algorithm 3 only the edge with the highest $r$-value is contracted, we eliminate the problem of contracting low connectivity edges along a path of high connectivity edges (as described in previous section).

Analyzing early experiments, we see that a major caveat of our NOI coarsening implementation is that it tends to produce many blocks with size close to the upper bound at first, yielding small isolated blocks primarily consisting of low degree vertices afterwards.

To tackle this problem, we have to get rid of the low degree vertices. One way is to preprocess the graph before performing the CF procedure. For this we set a fixed empirical threshold $d$ for the degree of a vertex. As formulated in algorithm 4, we iterate over the whole graph and mark a random edge incident to a low degree vertex

[1] Recall that although $r(\cdot)$ is a function on the set of vertices here, within the specific iteration, $r(w)$ rates the currently *scanned* edge $\{u, w\}$.

as *contractible*. That way, the low degree vertex is guaranteed to be contracted in the coarsening process.

**Algorithm 4:** ContractLowDegreeVertices()

**Input:** $G := (V, E, c, \omega)$, *blocks*, block upper bound $b$, degree threshold $d$
**Output:** partially coarsened graph $G' := (V', E', c', \omega')$

```
1 forall vertices v ∈ V do
2     if deg(v) ≤ d then
3         Pick a random edge e = {v, w} incident to v
4         if combined block weight ≤ b then
5             mark e as contractible
6             merge block containing u and block containing w
7 contract contractible edges
```

Another way is to resolve the issue of islands of low degree blocks after performing CF. Since SCLaP seems not to suffer from this problem, it makes sense to combine the two approaches. For this purpose we simply perform CF first and perform SCLaP on the result.

These amongst other considerations that are explained below, lead to the final variants described in the following section.

## 4.4 Variants

In order to engineer a NOI-based coarsening algorithm that aims for either state of the art quality or run time performance or both, different strategies are applied to optimize and extend the core algorithm outlined in Section 4.2.

### 4.4.1 Basic-NOI-Coarsening

Algorithm 5 describes the most bare bone variant with only small changes compared to algorithm 3. We combine the NOI-Coarsening routine as in algorithm 3 with the optimization in form of the predicate (line 10) mentioned in the previous section. Additionally, we also incorporate algorithm 4 as preprocessing step in line 5. Thus, our algorithm takes an empiric threshold $d$ for the degree cut off, until to which we want to preprocess vertices, as an additional input.

In contrast to algorithm 3, Basic-NOI-Coarsening does not necessarily contract all edges that are scanned. This comes with the aforementioned desired effect of potentially preserving low connectivity edges as long as the minimum degree is not close to zero. However, as soon as we do not immediately mark an edge as *contractible* when scanning it, line 13 gets somewhat more complex implementation wise. Reason for this is that executing algorithm 5 does not simply add single vertices to a block till the block is full. Imagine that within the CF BFS the currently visited node has only incident edges that do not fulfill the predicate. With the next visited node starting to grow a different block, it can eventually happen that an edge

**Algorithm 5:** Basic-NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size, degree threshold $d$
**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  label all vertices v ∈ V as unvisited
2  label all edges e ∈ E as uncontractible
3  r(v) := 0   ∀v ∈ V
4  initialize blocks as singletons {v}   ∀v ∈ V
5  ContractLowDegreeVertices(G, T, b, d)          // Execute algorithm 4
6  while there is an unvisited vertex do
7      u ← pick unvisited vertex with largest r
8      for each edge e = {u, w} incident to u do
9          if vertex w is unvisited then
10             if r(w) + ω(e) ≥ min_{v∈V} deg(v) then
11                 if combined block weight ≤ b then
12                     mark e as contractible
13                     merge block containing u and block containing w
14                 r(w) = r(w) + ω(e)
15         mark w as visited
16 contract all contractible edges
```

between two large blocks is marked for contraction. When merging the two blocks, where each can consist of multiple nodes, all the affected vertices must be remapped to the new combined block. A data structure making such merging more efficient is *Union Find*, where we extend the data structure to also track the node weight of the blocks (necessary due to line 11). For a more detailed version of the algorithm making use of abstract data structures we refer to algorithm 12 in the appendices.

### 4.4.2 Pre-Sort-NOI-Coarsening

The goal of cluster coarsening algorithms is to contract edges of high connectivity. In NOI coarsening every contraction is finite - once the blocks are merged (especially with Union Find), the decision to contract this edge cannot be undone. Applying algorithm 5 it happens that a high connectivity edge is not contracted since combining the respective blocks would result into exceeding the defined upper bound block size. The likelihood for this to happen can be reduced by contracting edges of high connectivity first. Since the edge rating $r(v)$ is a heuristic measure for the connectivity, it makes sense to sort the edges according to their $r$-value first and contract them starting with the highest $r$-value afterwards. This results in blocks growing from the locally highest $r$-value edges. In other words, the contractions are propagated along the path of the highest $r$-values until the respective blocks are full.

Algorithm 6 is built in a similar way as the Basic-NOI variant (algorithm 5). Here, however, the CF routine functions as a pre-sorting of the edges. For this, we define $r : E \to \mathbb{N}^*$ such that it is a direct relation between an edge and its edge rating and initialize it to zero for all edges (line 4). Within the CF BFS, in line 11, we just assign the edge rating $r(w)$ directly to the edge.

In the second part (from line 13 on) we iteratively pick the edge with highest rating $r$ and then check if its eligible for contraction. Eventually the selected edges are contracted.

### 4.4.3 Multi-Run-NOI-Coarsening

Another caveat of CF, which pre-sorting alone does not resolve, is that the edge rating heuristic depends on the order of how the graph is traversed. I.e. runs with different starting nodes yield different $r$-values on the same edges. In order to mitigate this effect, we perform multiple runs and calculate the average edge rating.

The Multi-Run variant shown in algorithm 7 extends the Pre-Sort variant by an outer for loop that runs the CF routine $n$ times, where $n$ is an additional empiric input parameter. The edge rating is added up (line 12) for the respective edge in each run (line 3 to 13). Note that calculating the sum of the $r$-values is equivalent to averaging the values since we are only interested in the priority of the edges to each other. As in algorithm 6, we end up with a sorting that determines the order of the subsequent contractions.

**Algorithm 6:** Pre-Sort-NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size, degree threshold $d$
**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  label all vertices v ∈ V as unvisited
2  label all edges e ∈ E as uncontractible
3  r(v) := 0   ∀v ∈ V
4  r(e) := 0   ∀e ∈ E
5  ContractLowDegreeVertices(G, T, b, d)          // Execute algorithm 4
6  while there is an unvisited vertex do
7      u ← pick unvisited vertex u with largest r(u)
8      for each edge e = {u, w} incident to u do
9          if vertex w is unvisited then
10             r(w) = r(w) + ω(e)
11             r(e) = r(w)
12     mark w as visited
13 while there is an uncontractible edge do
14     e ← pick uncontractible edge e = {u, w} with largest r(e)
15     if combined block weight ≤ b then
16         mark e as contractible
17         merge block containing u and block containing w
18 contract all contractible edges
```

**Algorithm 7:** Multi-Run-NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size, degree threshold $d$, number of runs $n$

**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
ContractLowDegreeVertices(G, T, b, d)          // Execute algorithm 4
R(e) := 0   ∀e ∈ E
for iteration 1 to n do
    label all vertices v ∈ V as unvisited
    label all edges e ∈ E as uncontractible
    r(v) := 0   ∀v ∈ V
    while there is an unvisited vertex do
        u ← pick unvisited vertex u with largest r(u)
        for each edge e = {u, w} incident to u do
            if vertex w is unvisited then
                r(w) = r(w) + ω(e)
                R(e) = R(e) + r(w)
        mark w as visited
while there is an uncontractible edge do
    e ← pick uncontractible edge e = {u, w} with largest r(e)
    if combined block weight ≤ b then
        mark e as contractible
        merge block containing u and block containing w
contract all contractible edges
```

### 4.4.4 SCLaP-NOI-Coarsening

SCLaP[19, 25] (cf. Section 3.2.1) finds clusters of vertices of high connectivity simply by assigning random vertices iteratively to the cluster the respective vertex is most strongly connected to. How strong a connection is, is determined by the sum of the edge weights connecting the regarded vertex and the respective cluster. Since CF's edge rating $r(\cdot)$ is also a heuristic measure for the connectivity and thereby takes into account the edge weight along multiple edges, it seems promising to combine the propagation scheme of SCLaP with the more sophisticated heuristic of CF.

Starting off of algorithm 7, in algorithm 8, we perform multiple CF-iterations for the same reason as stated in the Multi-Run variant (Section 4.4.3). The ratings $R(e)$ then are all we need for the label propagation scheme that follows in line 14. We initialize our blocks (or clusters) as singletons of $V$. I.e., at the start each block consists of only one vertex. We perform multiple label iterations (line 15) which yields more stable results. Within the while loop in line 17 the actual label propagation takes place. Starting at some random node, we move the regarded node $u$ to the block its most strongly connected to by means of the CF-rating, more formally we move it to the block $V_i$ that maximizes

$$\sum_{e \in \{\{u,w\} | w \in \Gamma(u) \cap V_i)\}} R(e). \tag{4.1}$$

The sum in Expression 4.1, adds up the $R$-values of all edges that are running between the current node $u$ and a particular block $V_i$. The block that has the highest sum value, is the one $u$ is moved[2] to. Eventually, each cluster is contracted to a single super vertex in the coarsened graph.

For a more detailed procedure, that elaborates on how to find the block that maximizes the sum in Expression 4.1, we refer to algorithm 15 in the appendices.

[2]Sticking to the terms orginally used in [25] and [19], node $u$ is assigned the *label* of that block.

**Algorithm 8:** SCLaP-NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size, degree threshold $d$, number of CAPFOREST iterations $n$, number of SCLaP iterations $m$

**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  R(e) := 0   ∀e ∈ E
2  for CF iteration 1 to n do
3      label all vertices v ∈ V as unvisited
4      label all edges e ∈ E as uncontractible
5      r(v) := 0   ∀v ∈ V
6      ContractLowDegreeVertices(G, T, b, d)      // Execute algorithm 4
7      while there is an unvisited vertex do
8          u ← pick unvisited vertex u with largest r(u)
9          for each edge e = {u, w} incident to u do
10             if vertex w is unvisited then
11                 r(w) = r(w) + ω(e)
12                 R(e) = R(e) + r(w)
13             Mark w as visited
14 initialize blocks V_i as singletons {v}   ∀v ∈ V, i ∈ I, where I is a suitable
     index set
15 for SCLaP iteration 1 to m do
16     label all vertices v ∈ V as unvisited
17     while there is an unvisited vertex do
18         u ← pick a random vertex v ∈ V
19         if c(V_i) + c(u) ≤ b then
20             move u to the block V_i that maximizes  Σ_{e∈{{u,w}|w∈Γ(u)∩V_i)}} R(e)
21         mark u as visited
22 contract block V_i to a single super vertex v ∈ V'   ∀i ∈ I
```

# 5 Experimental Evaluation

In this chapter we evaluate our algorithms which we described in the previous chapter. In 5.1 we describe the experimental setup including environment and methodology used for the experiments. In Section 5.2 we explain the choice of parameters. Section 5.3 compares quality and run time performance of our algorithms with existing algorithms.

## 5.1 Experimental Setup

### 5.1.1 Implementation Details

The algorithms within this thesis were implemented within the graph partitioning framework of *Karlsruhe High Quality Partitioning* (KaHIP)[11, 27] version 2.10 called KaFFPa. All algorithms discussed in Section 4.4 have been implemented with C++11. The Algorithms Section in the appendices, gives some more insight on how the implementation looks like.

### 5.1.2 Environment

The experiments discussed in the following sections are performed on a machine with four `AMD Opteron 6174` CPUs with 12 cores each. The machine comes with 48 cores in total, each one of it running at a clock speed of 2.2 GHz, and 252 GB of RAM. The machine runs on `Ubuntu 18.04` with Linux kernel version 4.15. The source code is compiled with `gcc` version 7.5.0 with `g++` compile optimization level `-O3` and run sequentially on a single core.

### 5.1.3 Methodology

The experiments are performed on the graphs from Walshaw's benchmark archive [31] listed in table 5.2 and mostly larger social network graphs shown in table 5.1 ranging up to approximately 1.4 million nodes and 14 million edges.

For the social network benchmarks KaFFPa offers three social configurations, one prioritizes running time (fastsocial), one quality (strongsocial) and one a compromise between the two (ecosocial). These configurations use cluster based coarsening, more precisely *Size-Constraint Label Propagation* (SCLaP) with Ensemble-Clustering, by default. This makes it possible to shrink highly irregular graphs more effectively than the matching based approach used in other configurations. Moreover, the multi-level framework is tuned particularly towards complex social networks and web graphs

| Graph | $\|V\|$ | $\|E\|$ | Max. deg. | Avg. deg. | Type |
|---|---|---|---|---|---|
| p2p-Gnutella04 | 6 405 | 29 215 | 103 | 9 | Peer to peer network |
| wordassociation-2011 | 10 617 | 63 788 | 332 | 12 | Word associations |
| PGPgiantcompo | 10 680 | 24 316 | 205 | 5 | PGP users network |
| email-EuAll | 16 805 | 60 260 | 3 282 | 7 | Email connections |
| as-22july06 | 22 963 | 48 436 | 2 390 | 4 | Router connections |
| soc-Slashdot0902 | 28 550 | 379 445 | 2 272 | 27 | News social network |
| loc-brightkite | 56 739 | 212 945 | 1 134 | 8 | Social network |
| enron | 69 244 | 254 449 | 1 634 | 7 | Email connections |
| loc-gowalla | 196 591 | 950 327 | 14 730 | 10 | Social network |
| coAuthorsCiteseer | 227 320 | 814 134 | 1 372 | 7 | Citation network |
| wiki-Talk | 232 314 | 1 458 806 | 100 029 | 13 | User interactions |
| citationCiteseer | 268 495 | 1 156 647 | 1 318 | 9 | Citation network |
| coAuthorsDBLP | 299 067 | 977 676 | 336 | 7 | Citation network |
| cnr-2000 | 325 557 | 2 738 969 | 18 236 | 17 | Web graph |
| web-Google | 356 648 | 2 093 324 | 5 235 | 12 | Web graph |
| coPapersCiteseer | 434 102 | 16 036 720 | 1 188 | 74 | Citation network |
| coPapersDBLP | 540 486 | 15 245 729 | 3 299 | 56 | Citation network |
| as-skitter | 554 930 | 5 797 663 | 29 874 | 21 | Internet topology graph |
| amazon-2008 | 735 323 | 3 523 472 | 1 077 | 10 | Product similarity graph |
| eu-2005 | 862 664 | 16 138 468 | 68 963 | 37 | Web graph |
| in-2004 | 1 382 908 | 13 591 473 | 21 869 | 20 | Web graph |

Table 5.1: Large social network graphs used for experimental evaluation sorted w.r.t. their size $|V|$.[20][19]

(cf. [19]). We use the less CPU-intensive fastsocial and ecosocial frameworks for the evaluation of the graphs in table 5.1.

For the Walshaw benchmark configurations we use fast, eco and strong. These framework configurations target traditional meshlike graphs, which are typically more regular in terms of degree distribution. Here, the default coarsening scheme is matching-based (random matchings resp. *Global Path Algorithm* (GPA) computed matchings) (cf. [26]). Within the parameter tuning section we present only results performed with configuration ecosocial for social network graphs resp. eco for Walshaw's benchmark archive graph collection. We perform partitionings aiming for a bipartition, $k = 16$ resp. $k = 64$ blocks and a balance constraint of three percent allowed imbalance. For parameter tuning we present the case of $k = 16$. Each computation is performed 5 times with different random seeds per instance using the same configuration.

### 5.1.4 Performance profiles

In order to compare different methods and to assess their quality and running time, we use performance profiles[5] (cf. Figure 5.1 to 5.14). Sticking to the notation originally used by Dolan et al.[5], we use performance profiles to represent the performance of a partitioning method $s \in \mathcal{S}$ applied onto a set of graph instances $\mathcal{P}$ with respect to the running time $t$ (resp. cut size $\lambda$). The performance ratio

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in \mathcal{S}\}} \tag{5.1}$$

is the ratio of the performance achieved by a specific method on a particular instance to the best performance of any method on this particular instance. The lower the ratio, the better is the partition of graph instance $p$ found by method $s$. The best possible performance ratio is 1 being equally good as the best method. For a particular method we then calculate the cumulative distribution function

$$\rho_s(\tau) = \frac{1}{n_p}|\{p \in \mathcal{P} : r_{p,s} \leq \tau\}|, \tag{5.2}$$

where for any given threshold $\tau \in \mathbb{R}$ the number of graph instances for which the performance ratio lies below (or equals) the threshold are counted and divided by the total number of graph instances $n_p$ within the test set $\mathcal{P}$. The resulting number $\rho_s(\tau)$ is the probability that the performance ratio of method $s$ lies within a factor $\tau$ of the ratio of the best method. We plot the probabilities

$$\rho_s(\tau) \colon [1, x] \to [0, 1] \tag{5.3}$$

for each benchmarked method with respect to threshold $\tau$ capping its range at a certain cutoff $x$.

By comparing the monotonously increasing line plots for different methods, usually one quickly sees which method performs best. Higher lines indicate higher performance. There are a few practical ways to look at these plots that help interpreting performance profiles. Firstly, the highest probability at $\tau = 1$ shows the method that has the most wins based on all instances. Note that intuitively, one could think that the probabilities at $\tau = 1$ must add up to 1, however, this is not the case if ties occur, i.e. if there is more than one winner per instance (cf. 5.12 (a) for an extreme example). Secondly, if we require the method to solve the most instances within a factor of, say 5, to the best method, we pick at the method that has the highest probability at $\tau = 5$. In the following sections we highlight the respective $\tau$ values discussed throughout the text by a blue vertical line in the corresponding plot. Conversely, we can ask for a method that solves, say, 75 percent of the instances efficiently, where *efficient* means lying within the respective factor of $\tau$. In this case we look at the y-axis and pick the method which reaches 0.75 first. Such would be highlighted by a blue horizontal line at $P(\tau) = 0.75$. Last but not least, as long as a line is highest for some $\tau$, we can derive that the method is the most

optimal one in some sense. Put even more simply, plotted lines in the top left show the best results.

## 5.2 Parameter Tuning

The engineered algorithms (Algorithm 5 to 8) discussed within Section 4.4 contain iteration numbers and a threshold $d$ for the ContractLowDegreeVertices optimization (cf. Algorithm 4) as parameters.

All variants contain threshold $d$ as parameter. We only present threshold tuning for Basic-NOI-Coarsening and SCLaP-NOI-Coarsening since these two variants are working in an entirely different manner while Pre-Sort-NOI-Coarsening and Multi-Run-NOI-Coarsening behave similar to Basic-NOI-Coarsing with respect to how the algorithm traverses the graph. The tuning is performed with KaFFPa's ecosocial resp. eco configuration and $k = 16$ since these settings act as a compromise between run time performance and quality and representing a more general partitioning than $k = 2$ or $k = 64$. Preliminary testing with other configurations show no surprising differences with respect to the impact of the tuned parameters.

Multi-Run-NOI-Coarsening and SCLaP-NOI-Coarsening contain iteration numbers. Both take in the number of *CAPFOREST* (CF) runs $n$, the SCLaP-NOI variant additionally takes the number of label propagation runs $m$ as input. The latter parameter is not tuned however since KaFFPa's default method for social networks, SCLaP, sets this parameter to 10 by default. For sake of comparability, we choose $m = 10$ in our SCLaP-NOI method.

Both parameters to be tuned, threshold $d$ and CF-iterations $n$, are largely independent from each other, we therefore treat them individually by setting $d = 0$ for the iteration tuning and $n = 10$ for the threshold tuning.

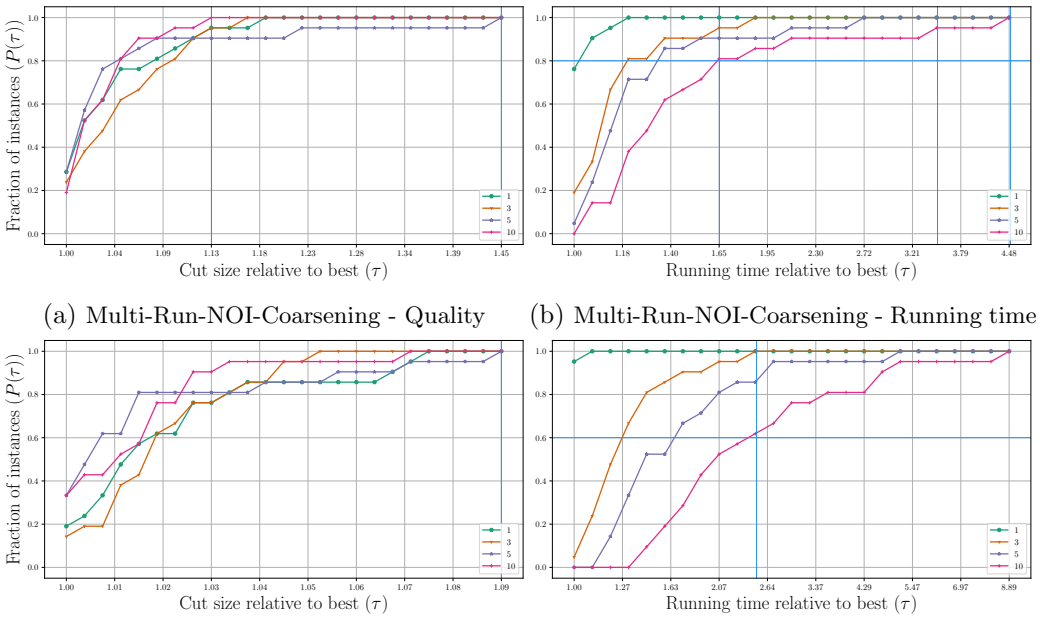
### 5.2.1 Social networks

#### Number of CF-iterations

Performing partitionings with the Multi-Run-NOI-Coarsening method shows (cf. Figure 5.1 (a)) that increasing iteration number $n$ tends to increase the quality of the cut slightly. This can be seen by observing that the lines for 10 (pink) and 5 (purple) iterations are found in the top left. The pink line shows consistent performance over all instances, the worst instance only measuring a 13 percent larger cut than the best result. Whereas the purple line only tops at $\tau = 1.45$, which means that even with 5 iterations the "unlucky" case of the CF-*breadth-first-search* (BFS) traversing the graph in an unideal order (cf. Subsection 4.4.3 for the theoretical background) occurs from time to time[1] yielding a 45 percent worse cut size than the best result.

From Figure 5.1 (b), we can see that the impact of performing multiple runs on the overall running time is significant. While for 80 percent of instances, the achieved running time of the algorithm performing 10 iterations is still within 60 percent

[1] Remember, we use 5 random seeds for each configuration.




(a) Multi-Run-NOI-Coarsening - Quality (b) Multi-Run-NOI-Coarsening - Running time

(c) SCLaP-NOI-Coarsening - Quality (d) SCLaP-NOI-Coarsening - Running time

Figure 5.1: Performance profiles comparing different CF-iteration numbers $n$ on social network graphs with KaFFPa's ecosocial configuration, k=16 and allowed imbalance of 3%.

of the algorithm performing only one iteration, the partitioning of a few instances takes 3.5 to 4.5 times as long. Since the Multi-Run variant aims for high quality by distinguishing itself from the Basic variant by performing multiple iterations and because we want to diminish the risk of suboptimal graph traversal, we choose $n = 10$ for this method.

As for the SCLaP-NOI-Coarsening method, Subfigure 5.1 (c) shows a slightly smaller impact of increasing the iteration number than in the Multi-Run case (Subfigure (a)). A quality performance decrease of about 10 percent has to be expected due to suboptimal graph traversal. Furthermore, the influence on the running time is more drastic in the SCLaP-NOI case. More precisely Subfigure (d) shows that only 60 percent of instances lie within a running time increase of factor 2.5 for the 10 iteration configuration and at least all instances are solved within this factor for the 3 iteration configuration (orange). Due to the minor cut size decrease but steep increase in running time, we settle for $n = 3$ mitigating the occurrence of "unlucky" graph traversals while the running time increase is still within reasonable bounds.

### ContractLowDegreeVertices threshold

Within all variants (cf. Algorithm 5 to 8) contracting low degree vertices is performed at a certain threshold $d$ for the vertex degree. Subfigure 5.2 (a) and (c) show that contracting low degree vertices does not significantly improve cut performance.

Looking at the raw numbers (not shown) for each instance reveals that a significant cut performance gain is only seen with the loc-brightkite instance (cf. table 5.1). However, these results are not surprising since the optimization is based on analyzing this particular graph showing loosely connected areas in the graph structure.

It stands out that the optimization yields a large increase in run time performance for the Basic-NOI-Coarsening variant. Contracting the nodes being within the lowest 0.1% percentile of node degree results in the best run time performance within a factor 1.5 for all instances and outperforms the method without this optimization ($d = 0$) by a factor of up to 4.2 on some instances as can be seen in Subfigure 5.2 (b). The more than significant running time decrease can be observed on several actual social network graphs and email connection networks. Apparently, several of these social network graphs contain low degree structure elements, where our Basic-NOI routine (and the related Pre-Sort and Muti-Run variant) hold up longer before finding a similar good cut than without the optimization. The lines in Subfigure (a) are mostly coinciding from $\tau = 1.1$ onwards. Thus, we expect performance decrease to be bounded within a 10 percent margin.

As the Basic variant aims for maximum speed, a threshold of 0.1 percent (i.e $d = 0.001$) is chosen for Basic-NOI-Coarsening. However, since some variants do not benefit from this optimization and 60 percent of graph instances display a cut performance decrease (cf. values at $\tau = 1.0$ in Subfigure (a) and (c)), we set the threshold to zero for the remaining variants.

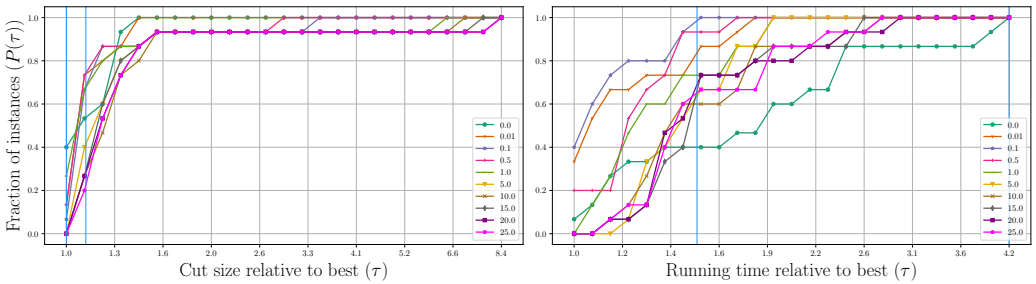
### 5.2.2 Walshaw benchmark graphs

#### Number of CF-iterations

Comparing different iteration numbers $n$ on Walshaw instances shows similar results like on the social network graphs. The cut performance differences are mostly within a 10 percent margin. However, the increase is more clearly visible than in the social network case (compare Subfigures 5.3 (a) and (c) with Subfigures 5.1 (a) and (c)). The running time increase is very similar as can be seen comparing Subfigures 5.3 (b) and (d) with Subfigures 5.1 (b) and (d). We, thus, come to same conclusions as before and set $n = 10$ for the Multi-Run variant and $n = 3$ for the SCLaP method.

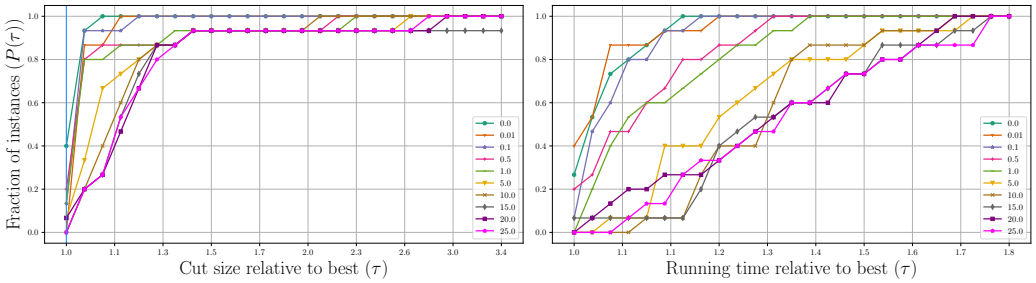
#### ContractLowDegreeVertices threshold

In contrast to the observations regarding threshold tuning for the collection of social network instances in the previous subsection, running the Walshaw experiments with a high threshold $d$ consistently decreases the cut found by Basic-NOI-Coarsening (Subfigure 5.4 (a)). Not only do partitionings with higher threshold $d$ show better cut performance on most instances (see $\tau$=1) but also their performance is consistent - e.g. the $d = 10$ configuration is always best within a 10 percent margin. Runs with low contraction threshold appear to have problems to find a good cut on some instances. On one particular graph, finan512, low threshold methods yield a 50 to 100 percent worse cut (not shown in Subfigure (a) since we cut off at $\tau = 1.25$). This is




(a) Basic-NOI-Coarsening - Quality

(b) Basic-NOI-Coarsening - Running time



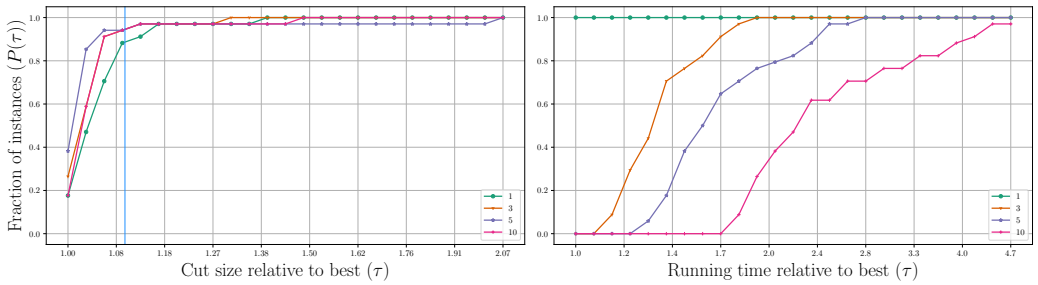

(c) SCLaP-NOI-Coarsening - Quality

(d) SCLaP-NOI-Coarsening - Running time

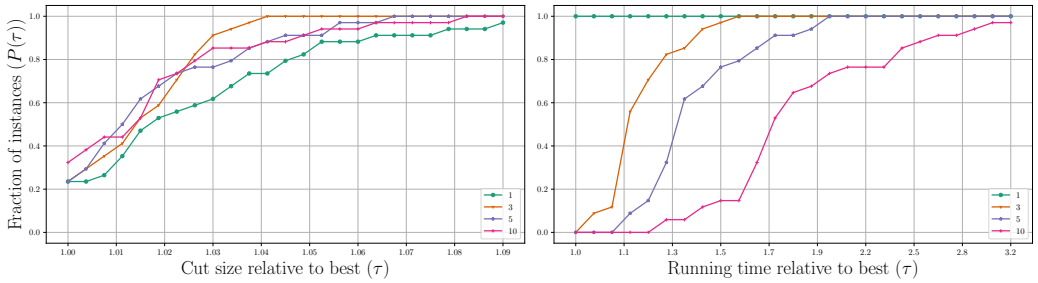
Figure 5.2: Performance profiles comparing different thresholds for the ContractLowDegreeVertices optimization on social network graphs with KaFFPa's ecosocial configuration, k=16 and allowed imbalance of 3%.

caused rather by the high variance the Basic-NOI method yields on this graph than setting a specific threshold as raw data imply. The graph is further discussed in the next section. Looking at Subfigure (b), we see that the higher threshold methods tend to be faster. Although this is not as clear as in the social network case, it can be observed at $\tau = 1$ and the point where they top out ($P(\tau) = 1$). Again, lower threshold methods have problems finding the cut being almost 3 times slower (not shown due to the cut off). We believe that there is a different reason for the described improvements using the ContractLowDegreeVertices optimization than the ones seen in the social network case. The LowDegreeVertexOptimization is originally designed for graphs with dense areas and few low degree vertex areas that we can contract without distorting the larger clusterings. We find this structure rather in social network graphs than in Walshaw's collection that contains mostly graphs from technical and physics applications, which are far less organized in clusters but show meshlike structures. However, some Walshaw graphs appear to be coarsened more effectively by aggressively contracting portions of the graph.

As can be seen in Subfigure 5.4 (c), increasing the threshold for the SCLaP-NOI method tends to decrease quality of the cut noticeably (cf. $\tau = 1$) even for smaller thresholds with barely significant differences in running time (Subfigure (d)). Only the improvement on the worst case instances discussed in the previous paragraph can also be observed for the SCLaP-NOI variant.




(a) Multi-Run-NOI-Coarsening - Quality (b) Multi-Run-NOI-Coarsening - Running time



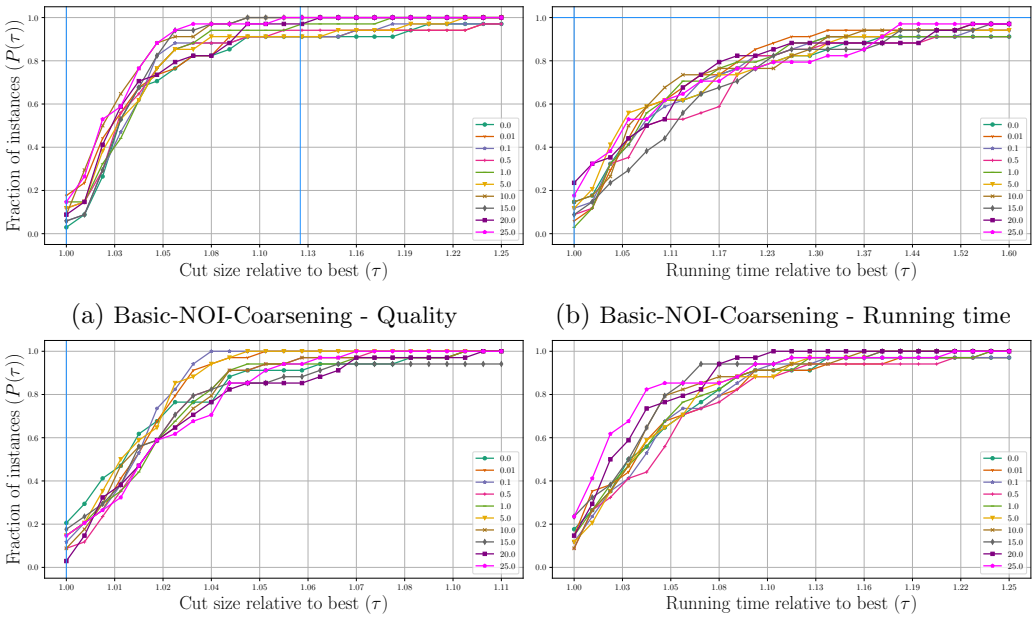

(c) SCLaP-NOI-Coarsening - Quality (d) SCLaP-NOI-Coarsening - Running time

Figure 5.3: Performance profiles comparing different CF-iteration numbers $n$ on Walshaw graphs with KaFFPa's eco configuration, k=16 and allowed imbalance of 3%.

Concluding, on Walshaw instances we set $d = 0.1$ (we contract the lowest 10 percent percentile of nodes with respect to their degree) for the Basic-NOI variant and its related variants (Pre-Sort-NOI and Multi-Run-NOI). For the SCLaP-NOI method we do not use the optimization, i.e. we set $d = 0$.

## 5.3 Comparison with Existing Algorithms

We assess quality and run time performance by comparing our coarsening variants presented in Section 4.4 and letting them compete with KaFFPa's default coarsening method on both social networks and smaller meshlike graphs from Walshaw's benchmark archive. Since our coarsening methods are implemented within the KaFFPa framework, we can compare cut size and running time of the coarsening methods within each of the respective framework configurations. The results are presented as performance profiles, the raw cut size and running time data can be found in the appendices in Section B. There we include all cut sizes and running time measurements for the social network instances - due to the vast amount of data on the Walshaw archive graphs, we show the raw data only for the high quality configuration strong.




(a) Basic-NOI-Coarsening - Quality

(b) Basic-NOI-Coarsening - Running time

(c) SCLaP-NOI-Coarsening - Quality

(d) SCLaP-NOI-Coarsening - Running time

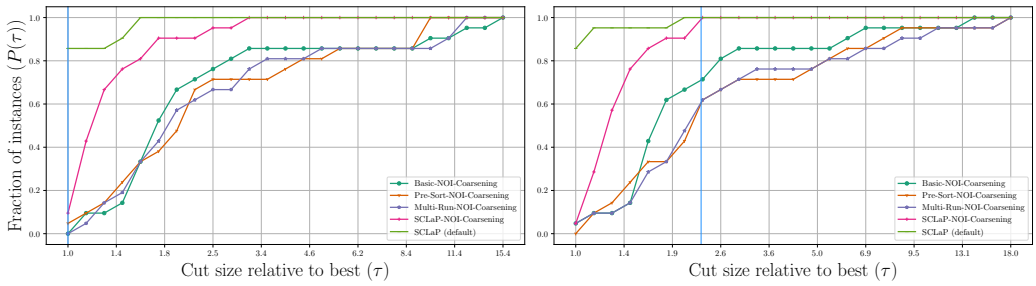
Figure 5.4: Performance profiles comparing different thresholds for the ContractLowDegreeVertices optimization on Walshaw graphs with KaFFPa's eco configuration, k=16 and allowed imbalance of 3%.

### 5.3.1 Social networks

$k = 2$

As we can see from Figure 5.5, SCLaP-NOI-Coarsening clearly shows the best bipartitioning quality out of our own methods but overall, the algorithm does not reach the cut performance of the state-of-the-art coarsening (default). Looking at $\tau = 1$ in the fastsocial benchmark (Subfigure (a)), we see that for around 83 percent of instances KaFFPa's default method finds the smallest cut whereas for around 10 percent of instances our SCLaP-NOI-Coarsening method finds the smallest cut. Pre-Sort-NOI-Coarsening is best on a few instances while Basic-NOI-Coarsening and Multi-Run-Coarsening never find the smallest cut.

Using the ecosocial framework, Subfigure (b) shows a similar picture but SCLaP-NOI-Coarsening method comes closer to state-of-the-art by achieving always the best cut within a factor of 2.3. For both configurations, fastsocial and ecosocial, it stands out that the Pre-Sort and Multi-Run variant do not show any significant cut performance increase over the simpler Basic-NOI-Coarsening method. It appears that the order in which CF-BFS traverses the graph does not matter. This is already indicated to some extent by the observations in the parameter tuning section, where varying the number of CF-iterations shows little impact on the cut size (cf. Subsection 5.2.1). SCLaP-NOI has a somewhat larger cut than KaFFPa's default



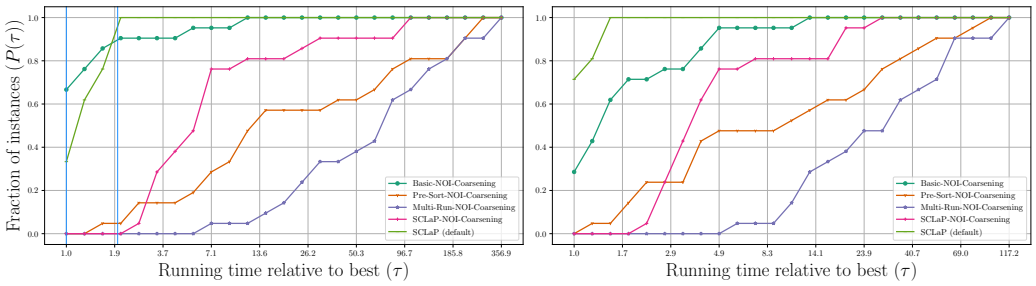

(a) fastsocial (b) ecosocial

Figure 5.5: Performance profiles comparing cut size of engineered variants and KaFFPa state-of-the-art coarsening (default) on social network graphs with k=2 and allowed imbalance of 3%.

method most of the times. On a few graphs found in table 5.1 the two methods are on par. There can be seen no particular pattern as to why, except that all of those are rather smaller less dense graphs. One instance stands out - on the Email connection network graph *enron* the SCLaP-NOI variant finds a 33 percent smaller cut using the fsocial configuration (see table 2). Otherwise the graph shows no extreme irregularities.

It strikes that in general the more complex SCLaP-NOI coarsening, using the CF rating, does not beat the more straight forward SCLaP (KaFFPa default) approach. However, from theory perspective the CF rating heuristic has no strict advantage in the general decision process of the SCLaP routine (cf. Sections 3.2.1, 3.3 and Chapter 4 for more theoretical background). The strength of the CF heuristic is rather that the rating contains information on the connectedness over a distance greater than one. A possible explanation for the underwhelming performance on the exclusively unweighted benchmark graphs could, thus, be that CF cannot play to his strength in the unweighted case since the impact of the far distance connectedness can be much higher in (edge-)weighted structures. It is plausible that in the unweighted case less complex approaches like adding the processed vertex to the largest block, as it is done by SCLaP, yield better results. We did some initial testings on a very small weighted example graph. In this instance the test results (not shown) are in favor of the SCLaP-NOI method, which finds a smaller cut than SCLaP in the default configuration. The result is very consistent with a standard deviation of zero for the performed NOI runs. For any conclusive statement proper evaluation of weighted graphs is needed. We leave this for future work and focus on the presented benchmark graphs within the scope of this thesis.

Figure 5.6 shows mixed results for NOI-Coarsening with respect to running time. On the one hand Basic-NOI-Coarsening is faster than KaFFPa's state-of-the-art coarsening on about 70 percent (cf. $\tau = 1$ in (a)) of instances using the fastsocial framework, on the other hand KaFFPa's default method is more consistent never exceeding double the running time of Basic-NOI-Coarsening while the more sophis-




(a) fastsocial (b) ecosocial

Figure 5.6: Performance profiles comparing running time of engineered variants and KaFFPa state-of-the-art coarsening (default) on social network graphs with k=2 and allowed imbalance of 3%.

ticated NOI variants run much slower overall. Moreover, all NOI methods perform vastly slower on certain social network graphs, which in extreme cases results in a running time difference of factor 100 or more. But only for methods without the LowDegreeVertexOptimization (cf. discussion in Subsection 5.2.1). Out of our methods Basic-NOI-Coarsening is the fastest most of the time. This is expected since it is the only variant with the said optimization. Further testing shows that the LowDegreeVertexOptimization yields a run time performance increase as large by a factor of almost 90 for the social network instance *loc-gowalla*.

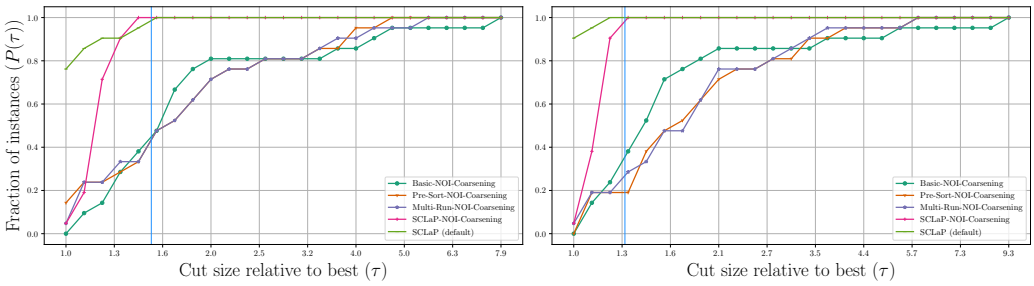
### $k = 16$

The quality benchmarking results of solving the partitioning problem with $k = 16$ in Figure 5.7 show the same tendencies as with $k = 2$ (compare Figure 5.5), the performance throughout the different methods is much more similar though. SCLaP-NOI-Coarsening finds the smallest cut for all instances within a margin of 50 percent using the fastsocial configuration and only a margin of 30 percent for the ecosocial configuration.

Similar can be said for comparing the run time performance of the $k = 16$ partitioning in Figure 5.8 with the analogous $k = 2$ experiments (Figure 5.6). This means worst case performance is bounded by a somewhat smaller factor than in the bipartitioning case. However, KaFFPa's default method is even a bit more dominant. While for the fast social configuration Basic-NOI-Coarsening is still competitive, using the ecosocial configuration, SCLaP is fastest on most instances. Interestingly, for the ecosocial configuration the SCLaP-NOI variant tends to be the fastest out of the NOI-Coarsening methods.

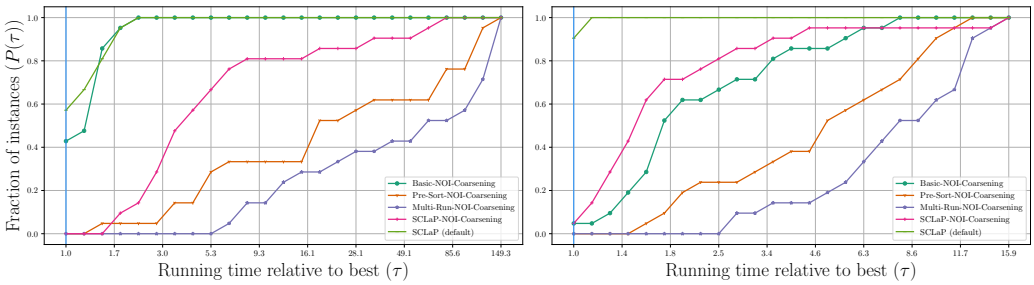
### $k = 64$

In Subfigure 5.9 we see that cut performance differences between the default method and SCLaP-NOI-Coarsening are within an 18 percent margin at maximum. Still,




(a) fastsocial (b) ecosocial

Figure 5.7: Performance profiles comparing cut size of engineered variants and KaFFPa state-of-the-art coarsening (default) on social network graphs with k=16 and allowed imbalance of 3%.




(a) fastsocial (b) ecosocial

Figure 5.8: Performance profiles comparing running time of engineered variants and KaFFPa state-of-the-art coarsening (default) on social network graphs with k=16 and allowed imbalance of 3%.

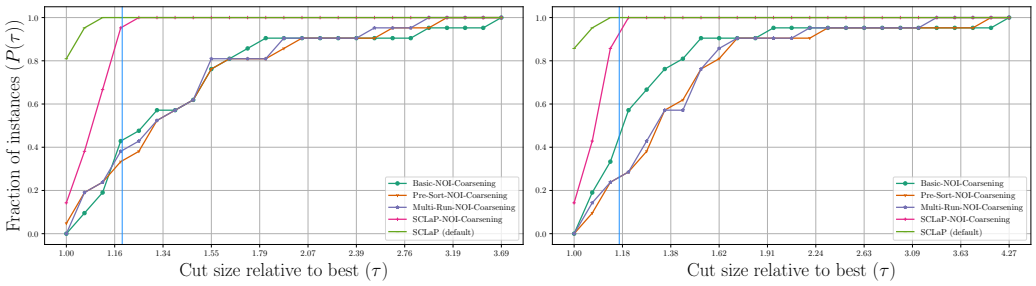
KaFFPa's default yields the better cut on most instances. Furthermore, the other three methods show less extreme cut sizes but the order of which solver is best and which is worst remains the same. Overall, we conclude that with increasing $k$, results get more similar quality wise.

Also in terms of running time differences are less extreme. However, Basic-NOI-Coarsening falls behind the SCLaP-based methods also with the fastsocial configuration.

### 5.3.2 Walshaw benchmark graphs

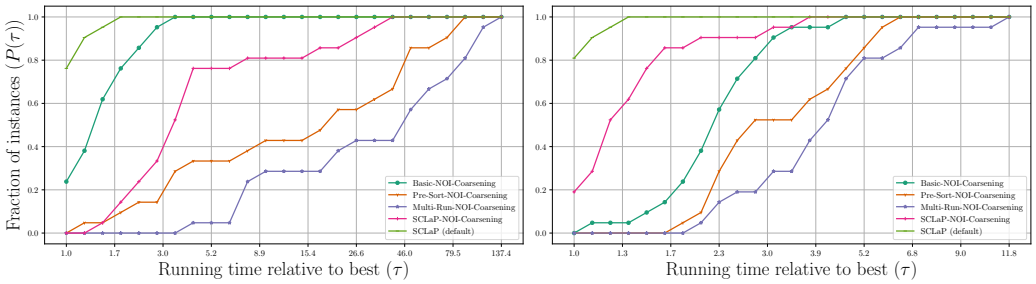
$k = 2$

Comparing the cut performance on Walshaw's benchmark instances of the different coarsening methods (cf. Figure 5.11) shows that as in the social network case SCLaP-NOI-Coarsening is clearly the best out of our own variants. SCLaP-NOI-Coarsening is competing with KaFFPa's state-of-the-art solver with each of the three tested framework configurations. Both find the best cut about 50 percent of the time for the




(a) fastsocial (b) ecosocial

Figure 5.9: Performance profiles comparing cut size of engineered variants and KaFFPa state-of-the-art coarsening (default) on social network graphs with k=64 and allowed imbalance of 3%.

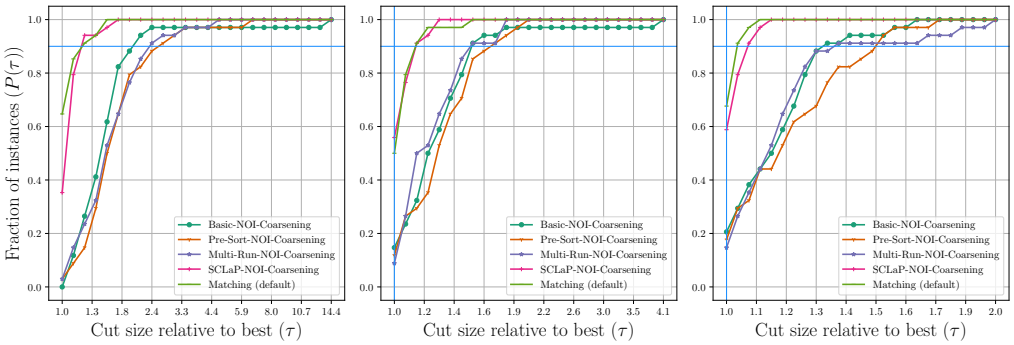



(a) fastsocial (b) ecosocial

Figure 5.10: Performance profiles comparing running time of engineered variants and KaFFPa state-of-the-art coarsening (default) on social network graphs with k=64 and allowed imbalance of 3%.

eco configuration while for fast SCLaP has a slight advantage. With configuration strong, both competing methods find the best cut more often than not. At $\tau = 1$ (Subfigure (a)), we see that the fractions do not add up to 1 but exceed 1. This is due to both methods finding the same best (possibly minimum) cut on the same instances. For 90 percent of instances the other three variants solve the partitioning with maximally double the cut size independent of configuration.

A particular instance, however, seems to be solved very poorly quality wise by the Basic, Pre-Sort and Multi-Run method but only with fast configuration. This instance, finan512, is a financial portfolio optimization graph that shows no extreme irregularities. Oddly, while with fast configuration Basic-NOI-Coarsening is 14 times slower than SCLaP-NOI and default, with eco and strong configuration Basic-NOI catches up to factor 2 and Multi-Run-NOI and Pre-Sort-NOI even find the best cut (cf. Table 1 in the appendices). Comparing the different methods in the raw data (not shown), it strikes that the standard deviation is extremely high for the Basic variant. Pre-Sort and Multi-Run are much more stable and achieve an average standard deviation of zero in eco and strong. This trend can be seen, even though to



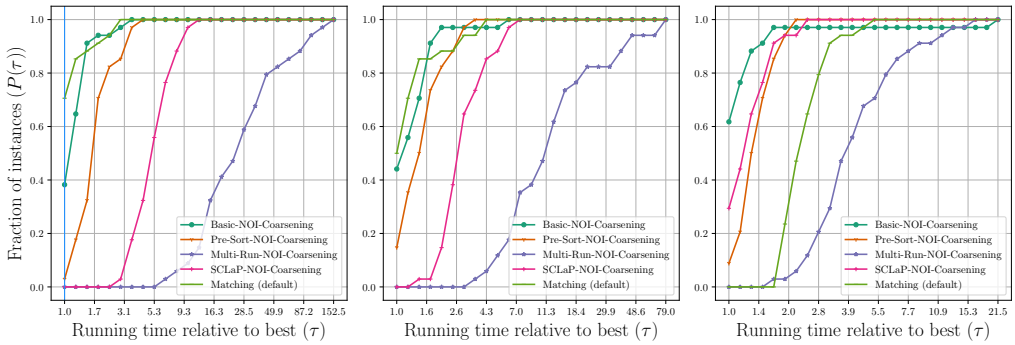

(a) fast (b) eco (c) strong

Figure 5.11: Performance profiles comparing cut size of engineered variants and KaFFPa state-of-the-art coarsening (default) on Walshaw graphs with k=2 and allowed imbalance of 3%.

a lesser extent, on several other Walshaw instances. Apparently, in contrast to the social network experiments, order of graph traversal during the CF-BFS matters a lot with these smaller meshlike graphs. More general, highly regular graphs like cti and fe_square show good results for all methods. Only with increased irregularity, SCLaP-NOI coarsening and the matching based default coarsening stand out.

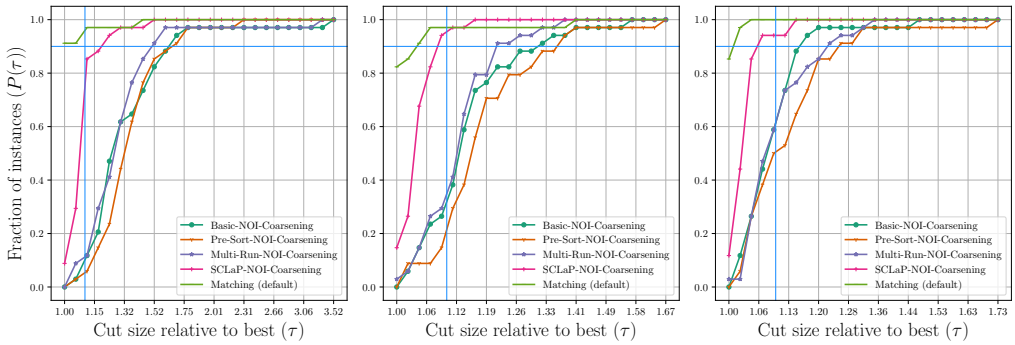
In terms of running time, the three benchmarked framework configurations show slightly different results. Subfigure 5.12 (a) shows that KaFFPa's default method is the fastest most of the times, followed by Basic-NOI-Coarsening and Pre-Sort-NOI-Coarsening - SCLaP-NOI-Coarsening and Multi-Run-Coarsening are much slower when using the fast configuration. Since with the fast configuration smaller instances are solved within a very short time close to measuring accuracy, values at $\tau = 1$ are less meaningful. Moreover, many ties occur at the smallest measured running times, thus values at $\tau = 1$ add up to more than 1. Looking at Subfigure (b), we see that Basic-NOI-Coarsenings run time performance is on par with the default method, otherwise the order remains the same. Using the strong variant (Subfigure (c)), all NOI variants except Multi-Run excel. It stands out that for the strong configuration SCLaP-NOI-Coarsening performs neck-at-neck with state-of-the-art quality wise while solving the partitioning consistently in half the time.

Unexpectedly, the outlier instance that can be seen in 5.12 (c) for Basic-NOI-Coarsening is the highly regular graph fe_sphere. Although Basic-NOI finds the best cut (ex aequo with SCLaP-NOI and default), it takes 20 times faster than the fastest method, which is SCLaP-NOI. While with social network graphs poor run time performance often goes hand in hand with a poor cut, for the Walshaw experiments, this is not the case. More general, we have to note that Basic-NOI-Coarsening is rather unstable.




(a) fast (b) eco (c) strong

Figure 5.12: Performance profiles comparing running time of engineered variants and KaFFPa state-of-the-art coarsening (default) on Walshaw graphs with k=2 and allowed imbalance of 3%.



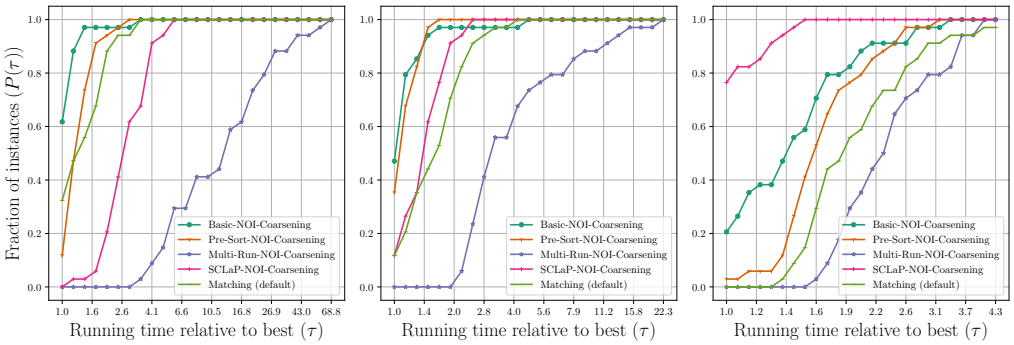

(a) fast (b) eco (c) strong

Figure 5.13: Performance profiles comparing cut size of engineered variants and KaFFPa state-of-the-art coarsening (default) on Walshaw graphs with k=16 and allowed imbalance of 3%.

$k = 16$

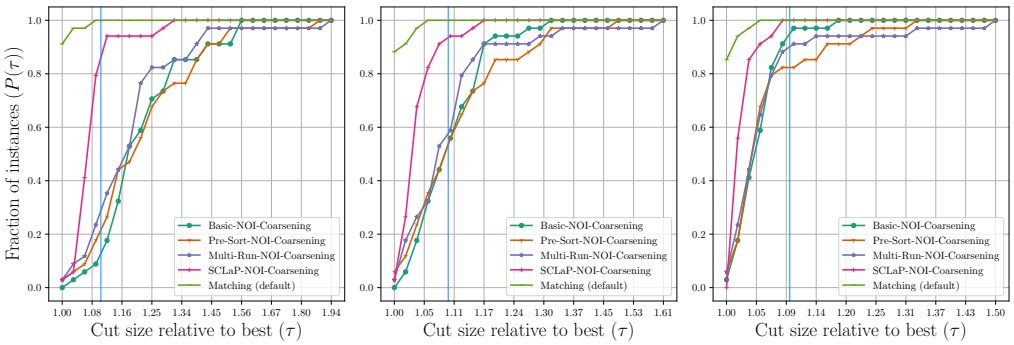
Partitioning into 16 blocks shows a similar picture (cf. 5.13) as with bipartitioning, with the difference being however that KaFFPa's default method performs marginally better quality wise than SCLaP-NOI. For all three configurations this difference amounts to about 10 percent at maximum in 90 percent of instances.

Run-time-wise (cf. 5.14), Basic-NOI-Coarsening and Pre-Sort-NOI-Coarsening are even more dominant than in the $k = 2$ case being always faster than the default method. Interestingly, SCLaP-NOI variant performs very well with the strong framework (Subfigure (c)) while the Basic and the Pre-Sort variant fall somewhat behind.




(a) fast (b) eco (c) strong

Figure 5.14: Performance profiles comparing running time of engineered variants and KaFFPa state-of-the-art coarsening (default) on Walshaw graphs with k=16 and allowed imbalance of 3%.



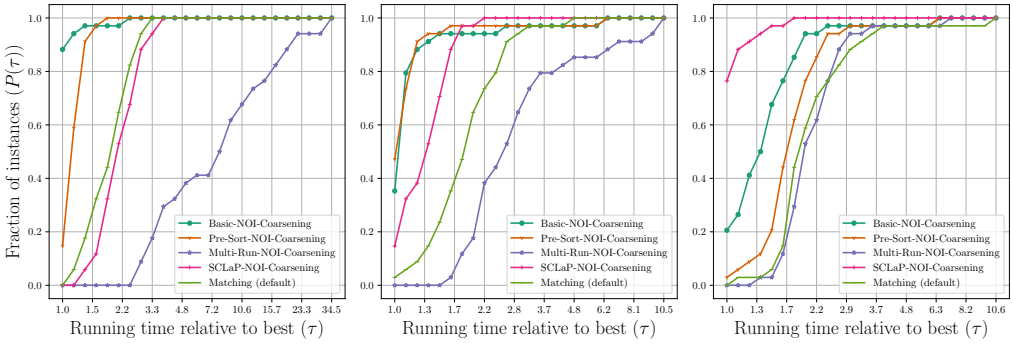

(a) fast (b) eco (c) strong

Figure 5.15: Performance profiles comparing cut size of engineered variants and KaFFPa state-of-the-art coarsening (default) on Walshaw graphs with k=64 and allowed imbalance of 3%.

Again, as with $k = 2$, Basic-NOI-Coarsening underperforms on the same graphs as in the $k = 2$ graph, which shows that it is not a random outlier, even though the performance difference is not as extreme as in the bipartitioning case.

### $k = 64$

As within the social network experiments, we also see a trend here towards more similar results when further increasing $k$. For $k = 64$, the cut performance difference between SCLaP-NOI and default method is less than 10 percent on most instances (cf. Figure 5.15). The overall order of the methods in terms of quality is very similar to the $k = 16$ case. Looking at the running time in Figure 5.16 we find almost the same image as with $k = 16$.




(a) fast (b) eco (c) strong

Figure 5.16: Performance profiles comparing running time of engineered variants and KaFFPa state-of-the-art coarsening (default) on Walshaw graphs with k=64 and allowed imbalance of 3%.

| Graph | Number of Nodes $\|V\|$ | Number of Edges $\|E\|$ | Max. degree | Avg. degree |
|---|---|---|---|---|
| add20 | 2 395 | 7 462 | 123 | 6 |
| data | 2 851 | 15 093 | 17 | 11 |
| 3elt | 4 720 | 13 722 | 9 | 6 |
| uk | 4 824 | 6 837 | 3 | 3 |
| add32 | 4 960 | 9 462 | 31 | 4 |
| bcsstk33 | 8 738 | 291 583 | 140 | 67 |
| whitaker3 | 9 800 | 28 989 | 8 | 6 |
| crack | 10 240 | 30 380 | 9 | 6 |
| wing_nodal | 10 937 | 75 488 | 28 | 14 |
| fe_4elt2 | 11 143 | 32 818 | 12 | 6 |
| vibrobox | 12 328 | 165 250 | 120 | 27 |
| bcsstk29 | 13 992 | 302 748 | 70 | 43 |
| 4elt | 15 606 | 45 878 | 10 | 6 |
| fe_sphere | 16 386 | 49 152 | 6 | 6 |
| cti | 16 840 | 48 232 | 6 | 6 |
| memplus | 17 758 | 54 196 | 573 | 6 |
| cs4 | 22 499 | 43 858 | 4 | 4 |
| bcsstk30 | 28 924 | 1 007 284 | 218 | 70 |
| bcsstk31 | 35 588 | 572 914 | 188 | 32 |
| fe_pwt | 36 519 | 144 794 | 15 | 8 |
| bcsstk32 | 44 609 | 985 046 | 215 | 44 |
| fe_body | 45 087 | 163 734 | 28 | 7 |
| t60k | 60 005 | 89 440 | 3 | 3 |
| wing | 62 032 | 121 544 | 4 | 4 |
| brack2 | 62 631 | 366 559 | 32 | 12 |
| finan512 | 74 752 | 261 120 | 54 | 7 |
| fe_tooth | 78 136 | 452 591 | 39 | 12 |
| fe_rotor | 99 617 | 662 431 | 125 | 13 |
| 598a | 110 971 | 741 934 | 26 | 13 |
| fe_ocean | 143 437 | 409 593 | 6 | 6 |
| 144 | 144 649 | 1 074 393 | 26 | 15 |
| wave | 156 317 | 1 059 331 | 44 | 14 |
| m14b | 214 765 | 1 679 018 | 40 | 16 |
| auto | 448 695 | 3 314 611 | 37 | 15 |

Table 5.2: Walshaw benchmark graphs sorted w.r.t to their size $|V|$.[31]

# 6 Discussion

## 6.1 Conclusion

During this thesis we engineered four *Nagamochi-Ono-Ibaraki* (NOI) based coarsening algorithms following two different approaches. The first one in its most basic version directly contracts edges during the NOI routine. The second approach followed the idea of providing a NOI based edge rating for *Size-Constraint Label Propagation* (SCLaP) algorithm.

Out of the methods following the first approach (Basic-NOI, Pre-Sort-NOI and Multi-Run-NOI), the Basic-NOI-Coarsening variant is the most promising. The enhancements in the other two variants did not improve the size of the cut significantly enough to justify the running time increase. Basic-NOI-Coarsening showed promising results in terms of running time in faster configurations of KaFFPa while SCLaP-NOI-Coarsening showed much higher cut performance than our other methods.

On social network and web graphs Basic-NOI-Coarsening cannot compete with state-of-the-art in terms of quality though finding cuts in the same amount of time and being much faster than our other methods. On the more traditional graphs of Walshaw's benchmark archive, Basic-NOI-Coarsening finds a faster cut than state-of-the-art on all instances for high $k$ partitioning ($k = 64$) when using the fast KaFFPa framework configuration. The running time advantage ranges from 20 percent to a factor of 4. The cut size lies always within a 50 percent margin of the state-of-the-art cut. On lower $k$ partitionings performance for both, quality and running time, decreases relatively.

SCLaP-NOI-Coarsening routine could not improve state-of-the-art results on social network and web graphs. On high $k$ partitionings ($k = 64$) our algorithm yields a cut that is 7 to 8 percent larger at the median and 16 to 18 percent larger in the worst case depending on the configuration. The algorithm takes significantly longer to find the cut. On lower $k$ partitionings worst case performance is further away from state-of-the-art performance.

For traditional rather meshlike graphs our SCLaP-NOI algorithm is competitive. Bipartitioning with the high quality configuration of KaFFPa's framework is neck-at-neck with KaFFPa's state-of-the-art algorithm while our method achieves a median performance improvement of approximately 100 percent in terms of running time. For higher $k$ partitionings our algorithm falls behind a few percent (approx. 10 percent at maximum) quality wise while still partitioning instances two times faster.

## 6.2 Future Work

Our most promising coarsening algorithms Basic-NOI-Coarsening and SCLaP-NOI-Coarsening still leave room for improvement. The former is interesting in terms of finding cuts in a short amount of CPU time. Further optimizations and analysis should, thus, focus towards fast execution time. SCLaP-NOI-Coarsening needs proper evaluation on weighted graphs in order to verify if the NOI rating can improve conventional SCLaP-Coarsening. Initial testing showed promising first results. Moreover, both, NOI and SCLaP-routine, can be parallelized to further improve run time performance.

Furthermore, during the course of this thesis we tried to optimize our algorithms. Our LowDegreeVertexOptimization improved Basic-NOI-Coarsening performance for both, more irregular mostly larger social network graphs as well as for smaller meshlike graphs. Thus, incorporating this optimization within state-of-the-art algorithms could potentially further improve partitioning performance. For SCLaP-NOI-Coarsening the optimization improved the performance only on graphs with few low density areas that get isolated during the coarsening process. Further analysis is needed as to how to avoid isolating these structure elements without impacting performance on other graphs.

# Acronyms

| | |
|---|---|
| BFS | breadth-first-search |
| CF | CAPFOREST |
| GPA | Global Path Algorithm |
| KaHIP | Karlsruhe High Quality Partitioning |
| LPA | Label Propagation Algorithm |
| NOI | Nagamochi-Ono-Ibaraki |
| PE | processing elements |
| SCLaP | Size-Constraint Label Propagation |
| SHEM | Sorted Heavy Edge Matching |
| VieCut | Vienna Minimum Cuts |

# Glossary

| | |
|---|---|
| adjacent | Two nodes $v$ and $u$ are adjacent to each other if there is an edge $e = \{u, v\}$ connecting them. Two edges $e$ and $f$ are called adjacent if they share the same node, i.e. $e \cap f \neq \emptyset$. |
| balance constraint | Within graph partitioning, a balance constraint $L$ defines the allowed deviation of the weight of a block $V_i$ from the average block weight. Formally, $c(V_i) \leq L := (1+\epsilon)\frac{c(V)}{k} \quad \forall i \in \{1, \ldots, k\}$, where $\epsilon \in \mathbb{R}_{\geq 0}$. |
| capacity | The capacity $\omega(S)$ is the combined weight of the cut edges determined by a cut $C = \{S, V \setminus S\}$. It is the size of the cut. Formally, $\omega(S) = \sum_{e \in \{\{u,v\} \mid u \in S, v \in V \setminus S\}} \omega(e)$. |
| coarsening | Coarsening a graph means reducing a graph to an instance of smaller size. This is achieved by finding edges that can be contracted by calculating a matching. Another approach is to perform a clustering, where clusters are eventually contracted to super vertices. In context of multi level partitioning this is repeated till the graph is small enough for initial partitioning. |
| connected | A connected graph is a graph where all vertices are connected with each other by paths (any amount of edges). |
| connected component | A (connected) component is a sub graph where all vertices are connected with each other by paths. |
| contraction | The contraction of an edge $e = \{u, v\}$ can be defined as replacing the vertices of the edge $e$ by a new vertex $w$, this vertex has the combined weight $c(w) = c(u) + c(v)$ of the original vertices. Expressed more intuitively, the vertices are *merged*. If by merging such two vertices, there are produced parallel edges $e_1$ and $e_2$, these are merged too, i.e. replaced by a new edge $f$ that has their combined weight $c(f) = c(e_1) + c(e_2)$. |
| cut | A cut of a graph $G := (V, E)$ is a bipartition $C = \{S, V \setminus S\}$ of the vertex set $V$. The cut implies a set of edges $F = \{e = s, t \mid s \in S \wedge t \in V \setminus S\}$ that connect the two sets. The name "cut" refers to "cutting through" these edges. |
| cyclic | A cyclic path is a path that allows equal nodes in the sequence of vertices. If this is the case, we say the path (and its graph) contains *cycles*. |

| | |
|---|---|
| degree | The degree of a vertex $v$ is the number of incident edges. Equivalently, it is also the number of neighbors, formally $deg(v) := \|\{u : \{u, v\} \in E\}\|$. |
| edge | An edge connects two (except in hyper graphs) nodes in a graph $G := (E, V)$ and can be represented as a pair of nodes $e = \{u, v\}$, where $e \in E$ and $u, v \in V$. |
| edge-connectivity | The edge-connectivity or just connectivity $\lambda(G)$ of a graph $G$ is the cut size of the mincut, i.e. $\lambda(G) = \min_{S \subset V} \omega(S)$. The edge-connectivity (or just connectivity) of two vertices $s, t$ refers to the the $\{S, V \setminus S\}$ with $s \in S$ and $t \in T$. Formally: $\lambda(\{u, v\}) = \min_{s \in S \wedge t \in V \setminus S} \omega(S)$. |
| edge-disjoint | Edge-disjoint sub graphs do not share a common edge. Edge disjoint maximum spanning trees cover all vertices of the graph while each pair of trees does not have a shared edge. |
| forest | A forest is a graph where each connected component is a tree. |
| graph | A graph $G$ consists of a set of nodes $V$ and a set of edges $E$, the nodes can be connected by edges $e = \{u, v\}$, where $e \in E$ and $u, v \in V$. |
| graph partitioning | A graph partitioning devides a graph into $k$ blocks where each block should be of similar or equal size. |
| incident | A vertex $v$ and an edge $e = \{u, w\}$ are considered incident if $e$ is connected to $v$, i.e $v = u \vee v = w$. |
| initial partitioning | Initial partitioning refers to the phase in multilevel partitioning, where the actual partitioning takes place. |
| inter-edge | In context of coarsening, clustering or partitioning, we deal with multiple vertex blocks $V_i$. An edge $\{v, w\}$ running between two different blocks is called inter-edge. I.e. for such edges holds $v \in V_i \implies w \notin V_i$. |
| intra-edge | In context of coarsening, clustering or partitioning, we deal with multiple vertex blocks $V_i$. An edge $\{v, w\}$ running within one and the same block is called intra-edge. I.e. for such edges holds $v \in V_i \implies w \in V_i$. |
| matching | A matching within a graph $G := (V, E)$ is a set of edges $M \subseteq E$ for that holds that any two edges within $M$ are not adjacent to each other, i.e do not share a vertex. |
| maximum degree | Largest degree in the graph, i.e. $\max_{v \in V} deg(v)$. |
| minimum cut | The minimum cut refers to (one of) the smallest possible cut(s), i.e $C_{min} = \min_{S \subset V} \sum_{e \in \{\{u,v\} \mid u \in S \wedge v \in V \setminus S\}} \omega(e)$. |
| minimum degree | Smallest degree in the graph, i.e. $\min_{v \in V} deg(v)$. |

| | |
|---|---|
| neighborhood | The neighborhood of a vertex $v$ is defined as $\Gamma(v) = \{u : \{u, v\} \in E\}$ which denotes the set of nodes adjacent to the vertex $v$. |
| node | A node or vertex $v$ is a unit in a graph $G := (V, E)$ that can have multiple edges $e = \{u, v\}$ (where $e \in E$ and $u, v \in V$) connecting it to other nodes. |
| path | A path is a sequence of edges ($e_1 = \{v_1, v_2\}$, ..., $e_i = \{v_i, v_{i+1}\}$, ... $e_{n-1} = \{v_{n-1}, v_n\}$) that determines a corresponding sequence of vertices ($\{v_1, \ldots, v_i, \ldots, v_n\}$) where any $v_i$ and $v_j$ and in consequence also any $e_i$ and $e_j$ within the sequences are pairwise distinct. We speak of a path *in* a graph $G := (V, E)$ if for all edges holds $e_i \in E$ with $i \in \{1 \ldots n-1\}$. |
| refinement | Within multilevel graph partitioning, during refinement, the previously contracted edges are iteratively uncontracted and quality of the partitioning is improved (refined) in each iteration. |
| spanning tree | A spanning tree of a graph $G := (V, E)$ is a subgraph $G' : (V, E')$ that is a tree and contains every vertex of $V$. |
| subgraph | A subgraph of a graph $G := (V, E)$ is a graph $G' := (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. |
| tree | A tree is a connected graph without cyclic paths. |
| undirected | An undirected graph is a graph in which the edges have no direction (or both directions). I.e. formally it then holds for an edge $e = \{u, v\} = \{v, u\}$. |
| Union Find | An abstract data structure for managing set partitions. |
| V-cycle | Performing multiple iterations of multilevel graph partitioning, one iteration is called a V-cycle. |
| vertex | Vertex is used interchangeably with *node*. |
| weighted | A weighted graph $G := (V, E, c, \omega)$ has a weight $c(v)$ assigned to each vertex $v \in V$, where $c : V \to \mathbb{N}$. Each edge $e \in E$ is assigned a weight $\omega(e)$, where $\omega : E \to \mathbb{N}^*$. |

# Bibliography


1. Y. Akhremtsev, P. Sanders, and C. Schulz. "High-Quality Shared-Memory Graph Partitioning". *CoRR* abs/1710.08231, 2017. arXiv: 1710.08231. URL: http://arxiv.org/abs/1710.08231.
2. T. N. Bui and C. Jones. "Finding Good Approximate Vertex and Edge Partitions is NP-Hard". *Inf. Process. Lett.* 42, 1992, pp. 153–159.
3. C. Chevalier and F. Pellegrini. "PT-Scotch: A tool for efficient parallel graph ordering". *Parallel Computing* 34:6-8, 2008, pp. 318–331. ISSN: 0167-8191. DOI: 10.1016/j.parco.2007.12.001. URL: http://dx.doi.org/10.1016/j.parco.2007.12.001.
4. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. "Customizable Route Planning". In: *Experimental Algorithms*. Ed. by P. M. Pardalos and S. Rebennack. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 376–387. ISBN: 978-3-642-20662-7.
5. E. D. Dolan and J. J. Moré. "Benchmarking optimization software with performance profiles". *Mathematical programming* 91:2, 2002, pp. 201–213.
6. M. Henzinger, A. Noe, and C. Schulz. "Faster Parallel Minimum Cuts", 2020.
7. M. Henzinger, A. Noe, and C. Schulz. "Shared-Memory Branch-and-Reduce for Multiterminal Cuts". *2020 Proceedings of the 22nd Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2020.
8. M. Henzinger, A. Noe, and C. Schulz. "Shared-memory Exact Minimum Cuts". In: *Proceedings of the 33rd International Parallel and Distributed Processing Symposium (IPDPS)*. 2019.
9. M. Henzinger, A. Noe, C. Schulz, and D. Strash. "Finding All Global Minimum Cuts in Practice", 2020.
10. M. Henzinger, A. Noe, C. Schulz, and D. Strash. "Practical Minimum Cut Algorithms". In: *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2018, pp. 48–61.
11. M. Holtgrewe, P. Sanders, and C. Schulz. "Engineering a scalable high quality graph partitioner". In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–12.
12. L. Hyafil and R. L. Rivest. *Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems*. Technical report Rapport de Recherche no. 33. IRIA – Laboratoire de Recherche en Informatique et Automatique, 1973.

13. G. Karypis and V. Kumar. “A fast and high quality multilevel scheme for partitioning irregular graphs”. *SIAM Journal on scientific Computing* 20:1, 1998, pp. 359–392.
14. G. Kirchhoff. “Über die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Verteilung galvanischer Ströme geführt wird”. In: *Annalen der Physik und Chemie*. 72nd ed. Johann Poggendorff, 1847, pp. 497–508.
15. J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. 2014.
16. D. Luxen and D. Schieferdecker. “Candidate Sets for Alternative Routes in Road Networks”. In: *Experimental Algorithms*. Ed. by R. Klasing. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 260–270. ISBN: 978-3-642-30850-5.
17. F. Manne and R. H. Bisseling. “A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem”. In: *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 708–717. ISBN: 978-3-540-68111-3.
18. J. Maue and P. Sanders. “Engineering Algorithms for Approximate Weighted Matching”. In: *Experimental Algorithms*. Ed. by C. Demetrescu. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 242–255. ISBN: 978-3-540-72845-0.
19. H. Meyerhenke, P. Sanders, and C. Schulz. “Partitioning (hierarchically clustered) complex networks via size-constrained graph clustering”. *Journal of Heuristics* 22:5, 2016, pp. 759–782. ISSN: 1572-9397. DOI: 10.1007/s10732-016-9315-8. URL: https://doi.org/10.1007/s10732-016-9315-8.
20. H. Meyerhenke, P. Sanders, and C. Schulz. “Partitioning Complex Networks via Size-Constrained Clustering”. In: *Experimental Algorithms*. Ed. by J. Gudmundsson and J. Katajainen. Springer International Publishing, Cham, 2014, pp. 351–363. ISBN: 978-3-319-07959-2.
21. H. Nagamochi and T. Ibaraki. “Computing edge-connectivity in multigraphs and capacitated graphs”. *SIAM Journal on Discrete Mathematics* 5:1, 1992, pp. 54–66.
22. H. Nagamochi, T. Ono, and T. Ibaraki. “Implementing an efficient minimum capacity cut algorithm”. *Mathematical Programming* 67:1-3, 1994, pp. 325–341.
23. W. de Nooy. “Social Network Analysis, Graph Theoretical Approaches to”. In: *Encyclopedia of Complexity and Systems Science*. Ed. by R. A. Meyers. Springer New York, New York, NY, 2009, pp. 8231–8245. ISBN: 978-0-387-30440-3. DOI: 10.1007/978-0-387-30440-3_488. URL: https://doi.org/10.1007/978-0-387-30440-3_488.
24. M. Ovelgönne and A. Geyer-Schulz. “An ensemble learning strategy for graph clustering”. 588, 2013. DOI: 10.1090/conm/588/11701.

25. U. N. Raghavan, R. Albert, and S. Kumara. "Near linear time algorithm to detect community structures in large-scale networks". *Physical Review E* 76:3, 2007. ISSN: 1550-2376. DOI: 10.1103/physreve.76.036106. URL: http://dx.doi.org/10.1103/PhysRevE.76.036106.

26. P. Sanders and C. Schulz. "Engineering Multilevel Graph Partitioning Algorithms". In: *Algorithms – ESA 2011*. Ed. by C. Demetrescu and M. M. Halldórsson. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 469–480. ISBN: 978-3-642-23719-5.

27. P. Sanders and C. Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. LNCS. Springer, 2013, pp. 164–175.

28. K. Schloegel, G. Karypis, and V. Kumar. "Graph Partitioning for High-Performance Scientific Simulations". In: *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003, pp. 491–541. ISBN: 1558608710.

29. K. Schloegel, G. Karypis, V. Kumar, J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, and M. Kaufmann. "Graph Partitioning for High Performance Scientific Simulations", 2000.

30. C. Schulz. "High Quality Graph Partitioning". PhD thesis. Karlsruher Instituts für Technologie, 2013.

31. A. J. Soper, C. Walshaw, and M. Cross. "A combined evolutionary search and multilevel optimisation approach to graph-partitioning". *Journal of Global Optimization* 29:2, 2004, pp. 225–241.

32. M. Toulouse, K. Thulasiraman, and F. Glover. "Multi-level Cooperative Search: A New Paradigm for Combinatorial Optimization and an Application to Graph Partitioning". In: *Euro-Par'99 Parallel Processing*. Ed. by P. Amestoy, P. Berger, M. Daydé, D. Ruiz, I. Duff, V. Frayssé, and L. Giraud. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 533–542. ISBN: 978-3-540-48311-3.

33. C. Walshaw. "Multilevel refinement for combinatorial optimisation problems". *Annals of Operations Research* 131:1-4, 2004, pp. 325–372.

34. W. Winterbach, P. Van Mieghem, M. Reinders, H. Wang, and D. de Ridder. "Topology of molecular interaction networks". *BMC systems biology* 7:1, 2013, p. 90.

# Appendices

## A Algorithms

**Procedure** ContractLowDegreeVertices

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, block upper bound $b$, Union Find partition $S$, degree threshold $d$
**Output:** Union Find data set $S$

**forall** vertices $v \in V$ **do**
  **if** $deg(v) \leq d$ **then**
    Pick a random edge $e = \{v, w\}$ adjacent to $v$
    **if** $UnionFind.Size(S, v) + UnionFind.Size(S, w) \leq b$ **then**
      $unionFind.Union(e)$
**return** $S$

**Algorithm 9:** CAPFOREST

**Input:** undirected weighted graph $G := (V, E, c, \omega)$
**Output:** lower bounds $q(e)$ on $\lambda(e)$, where $e \in E$

$V_{unv} := V$ `// label all vertices as` ***unvisited***
$r(v) := 0 \quad \forall v \in V$
**while** $V_{unv} \neq \emptyset$ **do**
  $u := \operatorname{argmax}_{v \in V_{unv}} r(v)$ `// pick` ***unvisited*** `vertex with largest` $r$
  **foreach** $e = \{u, w\}$, where $w \in V$ **do**
    **if** $w \in V_{unv}$ **then**
      $r(w) := r(w) + \omega(e)$
      $q(e) := r(w)$
  $V_{unv} = V_{unv} \setminus \{u\}$ `// mark` $u$ `as` ***visited***

**Algorithm 10:** NOI-based coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size

**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  V_unv := V                                   // label all vertices as unvisited
2  E_con := ∅
3  r(v) := 0   ∀v ∈ V
4  Block(v) := {v}   ∀v ∈ V                     // initialize blocks as singletons
5  while V_unv ≠ ∅ do
6      u := argmax_{v ∈ V_unv} r(v)             // pick unvisited vertex with largest r
7      foreach e = {u, w}, where w ∈ V do
8          if w ∈ V_unv then
9              if ω(Block(u)) + ω(Block(w)) ≤ b then
10                 E_con = E_con ∪ {e}          // mark e as contractible
                   // combine blocks of u and w
11                 Block(v) = Block(u) ∪ Block(w)   ∀v ∈ Block(u) ∪ Block(w)
12             r(w) = r(w) + ω(e)
13     V_unv = V_unv \ {u}                      // mark u as visited
14 Contract(E_con)                              // contract all contractible edges
```

**Algorithm 11:** NOI-based coarsening using Priority Queue and Union Find

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size

**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  Label all vertices v ∈ V as unvisited
2  Label all vertices v ∈ V as unseen
3  r(v) := 0   ∀v ∈ V
4  UnionFind.Init(S, V)  // Initialize Union Find structure with each
     vertex v ∈ V representing its own class in set S
5  while there is an unvisited vertex do
6      PriorityQueue.Init(S, r)      // Initialize a Priority Queue using
         rating function r(·) as priority key
7      Pick random unvisited vertex v as starting node
8      PriorityQueue.Insert(S, v, 0)  // Insert vertex v with key r(v) = 0
         into the priority queue
9      while S is not empty do
10         u := PriorityQueue.ExtractMax(S)  // Get the vertex with the
             largest r and remove it from the priority queue
11         foreach e = {u, w}, w ∈ V do
12             if vertex w is unvisited then
13                 if UnionFind.Size(S, u) + UnionFind.Size(S, w) ≤ b then
14                     UnionFind.Union(S, u, w)      // Union the partition
                         sets containing u and v
15                 r(w) = r(w) + ω(e)
16                 if vertex w is seen then
17                     PriorityQueue.IncreaseKey(S, w, r(w))    // Update the
                         key of vertex w to the current value of r(w)
18                 else
19                     mark vertex w as seen
20                     PriorityQueue.Push(S, w, r(w))  // Put vertex w onto
                         the priority queue
21         Mark w as visited
22 Contract(S)
```

**Algorithm 12:** Basic-NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size, degree threshold $d$
**Output:** coarsened graph $G' := (V', E', c', \omega')$

Label all vertices $v \in V$ as *unvisited*
Label all vertices $v \in V$ as *unseen*
$r(v) := 0 \quad \forall v \in V$
$UnionFind.Init(T, V)$ `// Initialize Union Find structure with each vertex` $v \in V$ `representing its own class`
$UnionLowDegreeVertices(G, T, b, d)$ `// Call procedure ContractLowDegreeVertices`
**while** there is an *unvisited* vertex **do**
    $PriorityQueue.Init(S, r)$ `// Initialize a Priority Queue using rating function` $r(\cdot)$ `as priority key`
    Pick random *unvisited* vertex $v$ as starting node
    $PriorityQueue.Insert(S, v, 0)$ `// Insert vertex` $v$ `with priority` $r(v) = 0$ `into the priority queue`
    **while** $S$ is not empty **do**
        $u := PriorityQueue.ExtractMax()$ `// Get the vertex with the largest` $r$ `and remove it from the priority queue`
        **for** each edge $e = \{u, w\}$ adjacent to $u$ **do**
            **if** vertex $w$ is *unvisited* **then**
                **if** $r(w) + \omega(e) \geq \min_{v \in V} deg(v)$ **then**
                    **if** $UnionFind.Size(T, u) + UnionFind.Size(T, w) \leq b$ **then**
                        $UnionFind.Union(T, u, w)$ `// Union the partition sets containing` $u$ `and` $v$
                    $r(w) = r(w) + \omega(e)$
                    **if** vertex $w$ is *seen* **then**
                        $PriorityQueue.IncreaseKey(S, w, r(w))$ `// Update the key of vertex` $w$ `to the current value of` $r(w)$
                    **else**
                        mark vertex $w$ as *seen*
                        $PriorityQueue.Insert(S, w, r(w))$ `// Put vertex` $w$ `onto the priority queue`
        Mark $w$ as *visited*
$Contract(S)$ `// contract Union Find partition`

**Algorithm 13:** Pre-Sort-NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size, degree threshold $d$
**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  Label all vertices v ∈ V as unvisited
2  Label all vertices v ∈ V as unseen
3  r(v) := 0   ∀v ∈ V
4  PriorityQueue.Init(R, r)        // Initialize a Priority Queue using
     rating function r(·) as priority key
5  UnionFind.Init(T, V)  // Initialize Union Find structure with each
     vertex v ∈ V representing its own class
6  UnionLowDegreeVertices(G, T, b, d)                  // Call procedure
     ContractLowDegreeVertices
7  while there is an unvisited vertex do
8      PriorityQueue.Init(S, r)      // Initialize a Priority Queue using
         rating function r(·) as priority key
9      Pick random unvisited vertex v as starting node
10     PriorityQueue.Insert(S, v, 0)      // Insert vertex v with priority
         r(v) = 0 into the heap
11     while heap is not empty do
12         u := PriorityQueue.ExtractMax(S)  // Get the vertex with the
             largest r and remove it from the heap
13         for each edge e = {u, w} adjacent to u do
14             if vertex w is unvisited then
15                 r(w) = r(w) + ω(e)
16                 PriorityQueue.Insert(S, e, r(w))     // Push edge onto the
                     priority queue
17                 if vertex w is seen then
18                     PriorityQueue.IncreaseKey(S, w, r(w))    // Update the
                         key of vertex w to the current value of r(w)
19                 else
20                     mark vertex w as seen
21                     PriorityQueue.Insert(S, w, r(w))        // Put vertex w
                         onto the heap
22         Mark w as visited
23 while R is not empty do
24     e := PriorityQueue.ExtractMax(R) UnionFind.Union(T, e)
25 Contract(T)
```

**Algorithm 14:** Multi-Run-NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size, degree threshold $d$, number of runs $n$
**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  PriorityQueue.Init(R, r)        // Initialize a Priority Queue using rating function r(·) as priority key
2  UnionFind.Init(T, V)            // Initialize Union Find structure with each vertex v ∈ V representing its own class
3  UnionLowDegreeVertices(G, T, b, d)                    // Call procedure ContractLowDegreeVertices
4  for iteration 1 to n do
5      Label all vertices v ∈ V as unvisited
6      Label all vertices v ∈ V as unseen
7      r(v) := 0   ∀v ∈ V
8      while there is an unvisited vertex do
9          PriorityQueue.Init(S, r)        // Initialize a Priority Queue using rating function r(·) as priority key
10         Pick random unvisited vertex v as starting node
11         PriorityQueue.Insert(S, v, 0)        // Insert vertex v with priority r(v) = 0 into the heap
12         while heap is not empty do
13             u := PriorityQueue.ExtractMax(S)     // Get the vertex with the largest r and remove it from the heap
14             for each edge e = {u, w} adjacent to u do
15                 if vertex w is unvisited then
16                     r(w) = r(w) + ω(e)
17                     if m == 1 then
18                         PriorityQueue.Insert(R, e, r(w))   // Push edge onto the priority queue
19                     else
20                         PriorityQueue.IncreaseKeyBy(R, e, r(w))  // Add r(w) to the value of the key
21                     if vertex w is seen then
22                         PriorityQueue.IncreaseKey(S, w, r(w))     // Update the key of vertex w to the current value of r(w)
23                     else
24                         mark vertex w as seen
25                         PriorityQueue.Insert(S, w, r(w))   // Put vertex w onto the heap
26             Mark w as visited
27 while R is not empty do
28     e := PriorityQueue.ExtractMax(R) UnionFind.Union(T, e)
29 Contract(T)                          // contract unioned partition sets of V
```

**Algorithm 15:** SCLaP-NOI-Coarsening

**Input:** undirected weighted graph $G := (V, E, c, \omega)$, upper bound $b$ for block size, degree threshold $d$, number of CAPFOREST iterations $n$, number of SCLaP iterations $m$

**Output:** coarsened graph $G' := (V', E', c', \omega')$

```
1  R(e) := 0   ∀e ∈ E
2  for run 1 to n do
3      Label all vertices v ∈ V as unvisited
4      Label all vertices v ∈ V as unseen
5      r(v) := 0   ∀v ∈ V
6      UnionLowDegreeVertices(G, T, b, d)                    // Call procedure
         ContractLowDegreeVertices
7      Contract(T)
8      while there is an unvisited vertex do
9          PriorityQueue.Init(S, r)        // Initialize a Priority Queue using
             rating function r(·) as priority key
10         Pick random unvisited vertex v as starting node
11         PriorityQueue.Insert(S, v, 0)        // Insert vertex v with priority
             r(v) = 0 into the heap
12         while S is not empty do
13             u := PriorityQueue.ExtractMax(S)     // Get the vertex with the
                 largest r and remove it from the heap
14             for each edge e = {u, w} adjacent to u do
15                 if vertex w is unvisited then
16                     if r(w) + ω(e) ≥ min_{v∈V} deg(v) then
17                         r(w) = r(w) + ω(e)
18                         R(e) = R(e) + r(w)
19                         if vertex w is seen then
20                             PriorityQueue.IncreaseKey(S, w, r(w)) // Update the
                                 key of vertex w to the current value of r(w)
21                         else
22                             mark vertex w as seen
23                             PriorityQueue.Insert(S, w, r(w))     // Put vertex w
                                 onto the heap
24             Mark w as visited
25 Cluster(v) := v   ∀v ∈ V
26 Ω(v) := ω(v)   ∀v ∈ V
27 L(v) := 0   ∀v ∈ V
28 for SCLaP iteration 1 to m do
29     Label all vertices as unpicked while there is an unpicked vertex v ∈ V do
30         pick a random vertex v from V
31         mark v as picked block := Cluster(v)
32         l := 0
33         foreach adjacent edge e = u, w of vertex v do
34             L(Cluster(w)) = L(Cluster(w)) + R(e)
35             if L(Cluster(w)) > l then
36                 if Ω(v) + ω(v) ≤ b then
37                     block = Cluster(w)
38                     l = L(w)
39             L(v) = 0
40         Cluster(v) = Cluster(w)
41         Ω(block) = Ω(block) + ω(v)
42         Ω(v) = Ω(v) − ω(w)
43 Contract(Cluster)
```

# B Results

| KaFFPa: | | **strong** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Coarsening: | | Basic-NOI | | Pre-Sort-NOI | | Multi-Run-NOI | | SCLaP-NOI | | Default | |
| Graph | $k$ | Cut | $t$ | Cut | $t$ | Cut | $t$ | Cut | $t$ | Cut | $t$ |
| crack | 2 | 192 | **0,170** | 186 | 0,201 | 190 | 0,467 | **184** | 0,201 | 185 | 0,359 |
| crack | 16 | 1 251 | 1,63 | 1 280 | 2,01 | 1 236 | 2,19 | 1 169 | **0,855** | **1 134** | 1,83 |
| crack | 64 | 2 709 | 3,15 | 2 743 | 4,21 | 2 751 | 4,91 | 2 685 | **2,68** | **2 654** | 4,10 |
| uk | 2 | 21 | **0,074** | 24 | 0,089 | 23 | 0,158 | 21 | 0,094 | **19** | 0,165 |
| uk | 16 | 164 | **0,307** | 175 | 0,394 | 172 | 0,547 | 163 | 0,395 | **156** | 0,817 |
| uk | 64 | 455 | **0,832** | 466 | 0,985 | 462 | 1,36 | 449 | 1,10 | **439** | 2,24 |
| whitaker3 | 2 | **126** | **0,128** | **126** | 0,151 | **126** | 0,405 | **126** | 0,189 | **126** | 0,334 |
| whitaker3 | 16 | 1 168 | 1,24 | 1 149 | 1,69 | 1 188 | 2,64 | 1 141 | **1,11** | **1 119** | 1,75 |
| whitaker3 | 64 | 2 655 | 3,23 | 2 644 | 4,72 | 2 631 | 5,37 | 2 605 | **2,71** | **2 589** | 5,68 |
| fe_ocean | 2 | **311** | **1,25** | **311** | 1,71 | 617 | 7,20 | **311** | 2,86 | **311** | 6,54 |
| fe_ocean | 16 | 8 829 | **9,95** | 10 665 | 17,9 | 10 556 | 27,6 | 8 415 | 12,0 | **8 123** | 28,3 |
| fe_ocean | 64 | 21 889 | **27,5** | 23 951 | 65,5 | 22 787 | 60,3 | 21 691 | 29,2 | **20 705** | 99,2 |
| fe_4elt2 | 2 | **130** | **0,117** | **130** | 0,132 | **130** | 0,387 | **130** | 0,183 | **130** | 0,332 |
| fe_4elt2 | 16 | 1 127 | 1,23 | 1 152 | 1,48 | 1 096 | 1,80 | 1 036 | **0,914** | **1 014** | 1,78 |
| fe_4elt2 | 64 | 2 635 | 2,98 | 2 651 | 3,73 | 2 646 | 4,68 | 2 601 | **2,57** | **2 566** | 4,32 |
| t60k | 2 | 85 | **0,867** | 79 | 1,17 | 86 | 2,08 | 76 | 0,952 | **74** | 1,53 |
| t60k | 16 | 879 | 4,79 | 861 | 4,18 | 859 | 5,51 | 860 | **2,98** | **837** | 5,03 |
| t60k | 64 | 2 200 | 12,0 | 2 243 | 12,7 | 2 262 | 14,5 | 2 202 | **8,23** | **2 184** | 14,3 |
| m14b | 2 | 4 494 | 12,1 | 4 576 | 13,5 | 4 467 | 49,8 | **3 823** | **11,7** | **3 823** | 25,2 |
| m14b | 16 | 49 724 | 134 | 49 559 | 165 | 48 406 | 190 | 43 909 | **79,8** | **43 064** | 135 |
| m14b | 64 | 106 270 | 321 | 103 839 | 393 | 101 927 | 408 | 100 200 | **180** | **98 939** | 309 |
| bcsstk30 | 2 | **6 251** | **2,37** | 9 508 | 4,11 | 10 385 | 34,9 | **6 251** | 4,14 | **6 251** | 7,34 |
| bcsstk30 | 16 | 79 124 | 14,8 | 79 807 | 17,9 | 82 319 | 44,7 | 74 298 | **11,3** | **73 648** | 18,7 |
| bcsstk30 | 64 | 185 539 | 29,6 | 186 796 | 32,4 | 188 947 | 62,6 | 180 522 | **23,0** | **178 422** | 47,0 |
| fe_rotor | 2 | 2 437 | 5,26 | 2 108 | 6,50 | 2 227 | 18,5 | **1 959** | **4,28** | **1 959** | 8,74 |
| fe_rotor | 16 | 23 298 | 48,1 | 22 440 | 45,8 | 22 155 | 60,0 | 21 521 | **28,2** | **21 126** | 44,3 |
| fe_rotor | 64 | 50 364 | 141 | 48 948 | 123 | 49 185 | 162 | 48 857 | **60,5** | **47 246** | 126 |
| bcsstk29 | 2 | **2 818** | **0,734** | **2 818** | 1,28 | **2 818** | 7,58 | **2 818** | 1,15 | **2 818** | 1,81 |
| bcsstk29 | 16 | 24 912 | 4,31 | 25 218 | 4,67 | 24 801 | 11,0 | 23 673 | **3,31** | **23 247** | 6,67 |
| bcsstk29 | 64 | 59 945 | **7,13** | 59 629 | 9,02 | 58 536 | 14,6 | 58 477 | 7,18 | **57 727** | 16,9 |
| bcsstk32 | 2 | 5 694 | 3,20 | 6 661 | **3,11** | 5 651 | 22,9 | **4 667** | 3,18 | 4 820 | 6,52 |
| bcsstk32 | 16 | 42 959 | 18,1 | 44 668 | 18,4 | 45 596 | 36,8 | 39 008 | **12,8** | **37 602** | 19,8 |
| bcsstk32 | 64 | 104 870 | 32,5 | 105 845 | 35,7 | 103 634 | 56,2 | 98 371 | **23,2** | **95 432** | 40,0 |
| vibrobox | 2 | 16 348 | 1,18 | 17 612 | 1,37 | 13 601 | 4,98 | **11 936** | **0,915** | 11 958 | 1,87 |
| vibrobox | 16 | 36 557 | 16,3 | 37 341 | 18,3 | 36 442 | 20,1 | 34 479 | **8,11** | **34 170** | 18,1 |
| vibrobox | 64 | 49 915 | 52,0 | **49 007** | 56,2 | 49 526 | 63,1 | 49 913 | **32,9** | 49 925 | 59,5 |
| data | 2 | 209 | **0,072** | 220 | 0,083 | 209 | 0,205 | **194** | 0,097 | **194** | 0,185 |
| data | 16 | 1 239 | **0,369** | 1 468 | 0,520 | 1 405 | 0,753 | 1 236 | 0,433 | **1 174** | 0,972 |
| data | 64 | 3 107 | 1,24 | 3 121 | **1,21** | 3 105 | 1,95 | 3 045 | 1,41 | **3 016** | 3,20 |
| fe_sphere | 2 | **384** | 6,12 | 396 | 0,337 | 410 | 0,782 | **384** | **0,285** | **384** | 0,492 |
| fe_sphere | 16 | 1 793 | 10,7 | 3 052 | 5,47 | 1 852 | 6,07 | **1 778** | **3,90** | 1 782 | 5,04 |
| fe_sphere | 64 | **3 671** | 12,7 | 3 768 | 12,3 | 3 837 | 13,3 | 3 700 | **6,74** | 3 717 | 11,1 |
| memplus | 2 | 6 874 | 2,47 | 7 003 | 2,74 | 7 085 | 6,71 | 6 417 | **1,83** | **5 949** | 5,56 |
| memplus | 16 | 13 819 | 19,6 | 13 889 | 18,5 | **13 681** | 24,6 | 15 636 | **6,11** | 14 247 | 24,5 |
| memplus | 64 | 16 885 | 86,8 | **16 884** | 90,2 | 17 053 | 96,1 | 18 295 | **14,0** | 17 668 | 148 |
| cs4 | 2 | 391 | **0,486** | 421 | 0,774 | 400 | 1,31 | 387 | 0,573 | **369** | 0,916 |
| cs4 | 16 | 2 222 | 4,76 | 2 199 | 6,32 | 2 228 | 7,20 | 2 196 | **4,33** | **2 150** | 7,64 |
| cs4 | 64 | 4 212 | 17,0 | 4 210 | 20,2 | 4 192 | 21,0 | 4 192 | **12,9** | **4 143** | 21,2 |
| cti | 2 | 351 | **0,202** | 449 | 0,266 | 339 | 0,580 | **338** | 0,323 | 342 | 0,697 |
| cti | 16 | 3 480 | 3,38 | 3 579 | 3,09 | 3 541 | 3,58 | 3 295 | **2,12** | **2 905** | 5,41 |
| cti | 64 | 6 568 | 10,3 | 7 262 | 13,0 | 6 746 | 12,4 | 6 368 | **7,60** | **5 941** | 16,9 |
| fe_body | 2 | 357 | **0,944** | 370 | 1,03 | 380 | 3,62 | **285** | 1,22 | **285** | 1,72 |
| fe_body | 16 | 2 062 | **3,08** | 2 221 | 5,11 | 2 316 | 7,62 | 1 984 | 3,74 | **1 865** | 5,30 |
| fe_body | 64 | 5 303 | 6,65 | 5 827 | 12,1 | 5 467 | 12,4 | 5 166 | **6,41** | **5 028** | 11,9 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| auto | 2 | 11 943 | 62,4 | 14 482 | 78,3 | 11 229 | 170 | 9 812 | **40,5** | **9 766** | 89,8 |
| auto | 16 | 89 096 | 600 | 92 379 | 785 | 85 159 | 738 | 79 532 | **301** | **77 169** | 432 |
| auto | 64 | 184 461 | 1 030 | 183 148 | 1 100 | 177 549 | 1 300 | 176 163 | **529** | **173 658** | 784 |
| bcsstk31 | 2 | 3 294 | **2,10** | 2 995 | 2,78 | 3 198 | 14,0 | 2 881 | 2,56 | **2 754** | 4,46 |
| bcsstk31 | 16 | 27 955 | 14,0 | 28 201 | 15,7 | 28 976 | 30,0 | 24 928 | **8,66** | **24 762** | 14,2 |
| bcsstk31 | 64 | 64 723 | 28,8 | 63 580 | 32,7 | 64 320 | 43,7 | 60 795 | **19,7** | **60 365** | 38,2 |
| 3elt | 2 | 110 | **0,089** | 111 | 0,103 | 92 | 0,201 | 91 | 0,114 | **87** | 0,212 |
| 3elt | 16 | 658 | **0,555** | 695 | 0,733 | 678 | 0,971 | 608 | 0,561 | **581** | 1,20 |
| 3elt | 64 | 1 657 | **1,42** | 1 668 | 1,95 | 1 656 | 2,45 | 1 610 | 1,50 | **1 599** | 3,66 |
| add20 | 2 | 772 | 0,277 | 810 | **0,225** | 718 | 0,357 | **656** | 0,365 | 699 | 0,640 |
| add20 | 16 | 2 362 | 1,70 | 2 373 | 1,59 | 2 345 | 2,14 | **2 255** | **0,627** | 2 281 | 2,70 |
| add20 | 64 | 3 365 | 7,14 | 3 317 | 7,77 | 3 363 | 8,21 | 3 374 | **5,13** | **3 158** | 5,23 |
| fe_tooth | 2 | 4 282 | 6,70 | 4 194 | 6,96 | 4 259 | 15,1 | **3 897** | **4,47** | 3 898 | 7,62 |
| fe_tooth | 16 | 19 051 | 38,9 | 18 475 | 38,8 | 18 616 | 40,0 | 18 543 | **18,0** | **17 965** | 38,2 |
| fe_tooth | 64 | 36 459 | 91,5 | 36 612 | 81,5 | 36 616 | 99,1 | 36 182 | **52,6** | **35 431** | 81,9 |
| bcsstk33 | 2 | 12 266 | **0,797** | 14 261 | 0,953 | 12 447 | 9,80 | **10 160** | 1,12 | 10 749 | 2,14 |
| bcsstk33 | 16 | 56 871 | **4,53** | 59 119 | 7,85 | 58 301 | 16,2 | **56 618** | 4,76 | 56 799 | 11,5 |
| bcsstk33 | 64 | 111 967 | **13,2** | 111 416 | 19,6 | **109 794** | 29,9 | 110 071 | 14,1 | 110 396 | 41,7 |
| 4elt | 2 | 144 | **0,200** | 159 | 0,229 | 156 | 0,635 | 146 | 0,262 | **141** | 0,463 |
| 4elt | 16 | 1 067 | 1,92 | 1 091 | 1,90 | 1 046 | 2,65 | 1 001 | **1,27** | **951** | 2,36 |
| 4elt | 64 | 2 801 | 3,75 | 2 827 | 5,37 | 2 848 | 6,30 | 2 701 | **3,23** | **2 653** | 6,02 |
| brack2 | 2 | 698 | 2,81 | **692** | **2,51** | 785 | 8,60 | 696 | 2,69 | 702 | 5,12 |
| brack2 | 16 | 12 964 | 14,7 | 13 250 | 16,8 | 12 712 | 22,0 | 12 260 | **10,5** | **11 835** | 16,2 |
| brack2 | 64 | 28 062 | 40,3 | 27 923 | 48,6 | 27 746 | 51,6 | 27 093 | **24,4** | **26 572** | 46,6 |
| add32 | 2 | 16 | **0,083** | 17 | 0,092 | 18 | 0,687 | **10** | 0,086 | **10** | 0,151 |
| add32 | 16 | 172 | 0,277 | 152 | **0,273** | 154 | 0,938 | 120 | 0,375 | **117** | 0,896 |
| add32 | 64 | 563 | **0,748** | 534 | 0,763 | 554 | 1,82 | 545 | 1,04 | **523** | 2,44 |
| wing_nodal | 2 | 1 946 | **0,561** | 1 761 | 0,752 | 1 798 | 1,73 | 1 822 | 0,567 | **1 714** | 1,02 |
| wing_nodal | 16 | 8 753 | 3,65 | 8 909 | 5,58 | 8 973 | 6,30 | 8 608 | **3,38** | **8 525** | 6,25 |
| wing_nodal | 64 | 16 612 | 9,45 | 16 450 | 14,8 | **16 360** | 14,8 | 16 424 | **8,37** | 16 397 | 14,8 |
| 144 | 2 | 7 928 | 9,50 | 7 582 | 16,8 | 8 071 | 43,6 | 6 489 | **8,55** | **6 471** | 17,6 |
| 144 | 16 | 43 413 | 83,4 | 41 570 | 121 | 40 878 | 132 | **39 002** | **73,2** | 39 307 | 110 |
| 144 | 64 | 83 086 | 184 | 83 233 | 307 | 82 170 | 309 | 80 794 | **131** | **80 281** | 217 |
| wave | 2 | 10 014 | 12,4 | 10 272 | 16,7 | 8 717 | 38,3 | **8 691** | **11,5** | 8 723 | 20,7 |
| wave | 16 | 49 346 | 146 | 50 245 | 180 | 47 890 | 171 | 44 906 | **92,7** | **43 340** | 140 |
| wave | 64 | 91 185 | 309 | 93 131 | 469 | 89 991 | 405 | 87 528 | **164** | **85 801** | 288 |
| 598a | 2 | **2 368** | **5,52** | 3 005 | 7,99 | 2 372 | 29,9 | 2 371 | 6,38 | 2 369 | 12,0 |
| 598a | 16 | 27 823 | 65,6 | 27 439 | 65,6 | 27 226 | 88,9 | 27 217 | **38,2** | **26 585** | 55,0 |
| 598a | 64 | 59 677 | 138 | 59 016 | 180 | 59 894 | 213 | 58 249 | **92,0** | **58 208** | 149 |
| finan512 | 2 | 248 | 1,77 | **162** | 1,56 | **162** | 5,79 | **162** | **1,50** | **162** | 3,12 |
| finan512 | 16 | 1 426 | 2,54 | 1 409 | 2,73 | 1 393 | 7,16 | 1 377 | **2,49** | **1 328** | 5,57 |
| finan512 | 64 | 11 337 | 12,3 | 14 292 | 19,9 | 16 178 | 28,3 | 11 049 | **8,30** | **10 799** | 18,1 |
| wing | 2 | 809 | **1,10** | 803 | 2,01 | 805 | 4,65 | 795 | 1,52 | **788** | 2,89 |
| wing | 16 | 4 150 | 20,5 | 4 081 | 30,7 | 4 107 | 27,7 | 4 082 | **14,3** | **3 973** | 23,2 |
| wing | 64 | 7 975 | 61,9 | 7 978 | 88,7 | 7 925 | 78,3 | 7 948 | **47,1** | **7 842** | 79,4 |
| fe_pwt | 2 | 357 | **0,292** | 345 | 0,358 | **343** | 1,81 | 345 | 0,637 | 346 | 1,20 |
| fe_pwt | 16 | 2 916 | **1,05** | 3 091 | 1,58 | 3 117 | 2,67 | 2 879 | 1,59 | **2 872** | 3,14 |
| fe_pwt | 64 | 9 848 | **6,34** | 10 376 | 9,41 | 11 054 | 8,48 | 8 434 | 11,4 | **8 368** | 17,0 |

Table 1: Results of the evaluation of our coarsening algorithms using KaFFPa's strong configuration and a balance constraint of 3% compared to KaFFPa's state-of-the-art coarsening (default) with equal configuration. Size of the found cut and execution time $t$ are averaged over 5 seeds and best result is highlighted for each instance. The used instances stem from Walshaws benchmark archive[31] (cf. Table 5.2).

| KaFFPa configuration: | | **fastsocial** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Coarsening algorithm: | | Basic-NOI | | Pre-Sort-NOI | | Multi-Run-NOI | | SCLaP-NOI | | Default | |
| Graph | $k$ | Cut | $t$ | Cut | $t$ | Cut | $t$ | Cut | $t$ | Cut | $t$ |
| loc-brightkite_edges | 2 | 29 284 | **0,277** | 30 068 | 21,1 | 29 788 | 24,1 | 26 005 | 1,05 | **20 851** | 0,380 |
| loc-brightkite_edges | 16 | 77 709 | 1,20 | 86 113 | 24,9 | 87 262 | 31,3 | 66 481 | 2,07 | **57 670** | **0,892** |
| loc-brightkite_edges | 64 | 99 990 | 2,10 | 104 178 | 20,7 | 103 236 | 29,1 | 87 407 | 2,65 | **79 572** | **1,28** |
| soc-Slashdot0902 | 2 | 137 158 | 0,410 | 111 781 | 1,81 | 118 666 | 8,53 | **95 841** | 2,31 | 124 593 | **0,368** |
| soc-Slashdot0902 | 16 | 285 327 | 2,29 | **264 915** | 6,64 | 268 013 | 16,0 | 274 195 | 5,50 | 279 216 | **1,46** |
| soc-Slashdot0902 | 64 | 317 014 | 5,21 | 311 484 | 7,18 | 310 957 | 32,4 | 309 985 | 8,17 | **309 848** | **4,03** |
| p2p-Gnutella04 | 2 | 8 146 | **0,035** | 8 265 | 0,045 | 8 318 | 0,234 | 8 309 | 0,209 | **8 051** | 0,061 |
| p2p-Gnutella04 | 16 | 18 401 | 0,386 | 18 130 | 0,407 | 18 203 | 1,46 | 17 799 | 0,505 | **17 740** | **0,272** |
| p2p-Gnutella04 | 64 | 20 430 | 0,483 | 20 394 | 0,584 | 20 282 | 1,67 | 20 016 | 0,660 | **19 982** | **0,476** |
| amazon-2008 | 2 | 134 076 | **3,11** | 166 835 | 28,3 | 222 259 | 89,8 | 89 514 | 13,3 | **77 394** | 5,28 |
| amazon-2008 | 16 | 457 901 | **5,22** | 495 314 | 414 | 471 009 | 617 | 288 065 | 31,6 | **256 779** | 10,3 |
| amazon-2008 | 64 | 612 238 | **10,1** | 597 439 | 450 | 573 636 | 682 | 398 524 | 31,2 | **374 556** | 11,1 |
| loc-gowalla_edges | 2 | 99 294 | **0,944** | 124 236 | 206 | 121 082 | 267 | 69 857 | 6,44 | **69 475** | 1,01 |
| loc-gowalla_edges | 16 | 314 283 | **3,25** | 365 881 | 337 | 366 070 | 408 | 279 256 | 15,0 | **250 868** | 3,62 |
| loc-gowalla_edges | 64 | 409 045 | 6,45 | 452 638 | 226 | 450 380 | 252 | 374 761 | 17,7 | **353 315** | **5,87** |
| PGPgiantcompo | 2 | 755 | 0,062 | 651 | 0,199 | 637 | 0,363 | 428 | 0,074 | **383** | **0,031** |
| PGPgiantcompo | 16 | 2 254 | 0,142 | 2 327 | 0,567 | 2 169 | 1,07 | 1 734 | 0,191 | **1 732** | **0,099** |
| PGPgiantcompo | 64 | 3 840 | 0,219 | 3 863 | 0,611 | 3 665 | 1,26 | 3 371 | 0,307 | **3 273** | **0,193** |
| email-EuAll | 2 | 1 175 | **0,281** | 2 780 | 3,29 | 2 236 | 4,80 | 2 191 | 0,884 | **743** | 0,325 |
| email-EuAll | 16 | 32 066 | 1,40 | 23 458 | 4,09 | **22 609** | 6,19 | 24 932 | 2,79 | 25 121 | **0,933** |
| email-EuAll | 64 | 41 272 | 1,45 | 38 790 | 3,99 | 38 595 | 7,29 | 37 716 | 4,02 | **36 467** | **1,06** |
| enron | 2 | 19 125 | 0,845 | 15 827 | 51,8 | 14 129 | 66,3 | **7 892** | 1,04 | 11 850 | **0,186** |
| enron | 16 | 100 479 | 1,27 | **77 705** | 72,4 | 78 283 | 87,8 | 88 504 | 1,98 | 79 086 | **0,590** |
| enron | 64 | 123 217 | 3,89 | 120 790 | 71,7 | 119 034 | 89,4 | 117 076 | 4,11 | **104 972** | **1,52** |
| web-Google | 2 | 30 654 | **1,69** | 59 287 | 19,0 | 53 924 | 48,1 | 15 026 | 8,60 | **11 182** | 2,87 |
| web-Google | 16 | 91 595 | **2,58** | 99 725 | 39,7 | 90 796 | 71,0 | 33 924 | 13,3 | **25 290** | 3,68 |
| web-Google | 64 | 121 890 | **3,87** | 110 922 | 34,7 | 102 404 | 69,6 | 49 991 | 13,9 | **42 853** | 4,13 |
| in-2004 | 2 | 37 033 | **5,63** | 35 884 | 81,7 | 41 372 | 632 | 6 224 | 162 | **3 829** | 6,46 |
| in-2004 | 16 | 89 075 | **5,23** | 75 956 | 170 | 80 231 | 745 | 25 634 | 172 | **19 979** | 7,96 |
| in-2004 | 64 | 183 414 | **6,55** | 148 498 | 204 | 140 538 | 896 | **49 824** | 158 | 51 749 | 8,97 |
| coPapersCiteseer | 2 | 466 057 | **3,51** | 403 157 | 8,68 | 383 914 | 540 | 315 782 | 33,4 | **288 939** | 6,34 |
| coPapersCiteseer | 16 | 1 644 421 | **4,57** | 1 451 814 | 13,8 | 1 412 508 | 548 | 984 822 | 34,6 | **830 879** | 7,34 |
| coPapersCiteseer | 64 | 1 966 165 | **5,10** | 1 655 847 | 14,3 | 1 620 734 | 554 | 1 228 207 | 37,7 | **1 097 656** | 7,75 |
| coAuthorsCiteseer | 2 | 38 089 | 1,56 | 43 500 | 65,1 | 40 858 | 86,7 | 26 169 | 3,05 | **22 248** | **1,06** |
| coAuthorsCiteseer | 16 | 98 021 | **2,09** | 97 762 | 138 | 97 408 | 179 | 73 504 | 7,19 | **58 526** | 2,27 |
| coAuthorsCiteseer | 64 | 115 374 | 4,67 | 112 377 | 116 | 112 506 | 150 | 86 700 | 7,31 | **73 890** | **2,54** |
| wordassociation-2011 | 2 | 12 119 | **0,055** | 15 623 | 0,454 | 16 420 | 1,05 | 11 990 | 0,171 | **11 350** | 0,067 |
| wordassociation-2011 | 16 | 38 417 | 0,577 | 39 905 | 0,986 | 39 276 | 2,49 | 35 041 | 0,530 | **32 121** | **0,314** |
| wordassociation-2011 | 64 | 43 466 | 0,709 | 44 730 | 1,22 | 44 507 | 3,88 | 42 277 | 0,932 | **39 147** | **0,575** |
| wiki-Talk | 2 | 294 185 | 61,0 | **99 593** | 2 980 | 122 461 | 3 020 | 144 493 | 242 | 145 592 | **42,4** |
| wiki-Talk | 16 | 1 018 775 | 164 | **637 376** | 2 660 | 637 612 | 2 920 | 817 536 | 362 | 863 243 | **151** |
| wiki-Talk | 64 | 1 203 653 | 219 | **986 513** | 2 430 | 1 005 938 | 2 120 | 1 007 000 | 425 | 1 095 698 | **116** |
| eu-2005 | 2 | 328 212 | **5,00** | 195 753 | 28,8 | 244 924 | 343 | 52 416 | 430 | **21 408** | 7,18 |
| eu-2005 | 16 | 1 470 410 | **6,59** | 1 196 232 | 501 | 1 104 709 | 823 | 471 028 | 493 | **359 489** | 9,38 |
| eu-2005 | 64 | 3 229 735 | 16,3 | 2 659 893 | 561 | 2 597 620 | 953 | **2 121 652** | 502 | 2 161 365 | **12,6** |
| cnr-2000 | 2 | 3 525 | **0,880** | 2 570 | 9,93 | 3 459 | 47,0 | 291 | 74,7 | **288** | 1,36 |
| cnr-2000 | 16 | 53 103 | **1,22** | 29 721 | 21,0 | 37 342 | 68,2 | **6 724** | 78,8 | 9 593 | 1,80 |
| cnr-2000 | 64 | 791 458 | 4,54 | 741 991 | 50,8 | 743 305 | 122 | **705 196** | 82,9 | 718 523 | **2,70** |
| citationCiteseer | 2 | 69 925 | 1,92 | 137 731 | 64,1 | 107 492 | 78,9 | 36 094 | 5,72 | **33 987** | **1,78** |
| citationCiteseer | 16 | 241 769 | 3,99 | 389 374 | 322 | 390 312 | 371 | 174 383 | 10,9 | **152 886** | **2,81** |
| citationCiteseer | 64 | 362 083 | 7,37 | 477 096 | 259 | 461 147 | 322 | 281 915 | 12,0 | **243 858** | **3,48** |
| coPapersDBLP | 2 | 783 865 | **4,54** | 767 697 | 10,7 | 778 390 | 344 | 643 531 | 30,0 | **531 097** | 8,18 |
| coPapersDBLP | 16 | 2 550 105 | **5,97** | 2 528 345 | 27,8 | 2 504 902 | 370 | 1 746 393 | 37,9 | **1 480 202** | 10,6 |
| coPapersDBLP | 64 | 2 954 971 | **9,05** | 2 862 281 | 29,0 | 2 861 508 | 388 | 2 238 065 | 37,0 | **1 949 482** | 10,9 |
| as-skitter | 2 | 476 014 | **4,92** | 647 145 | 494 | 637 228 | 638 | 328 999 | 112 | **264 582** | 5,50 |
| as-skitter | 16 | 1 447 872 | 9,33 | 2 077 814 | 854 | 1 979 407 | 1 070 | 1 193 238 | 121 | **1 015 514** | **7,52** |
| as-skitter | 64 | 2 217 021 | 30,0 | 2 519 487 | 785 | 2 511 361 | 960 | 1 974 508 | 137 | **1 771 772** | **9,31** |
| coAuthorsDBLP | 2 | 88 056 | 18,6 | 101 213 | 347 | 101 622 | 344 | 67 074 | 5,26 | **49 626** | **1,89** |
| coAuthorsDBLP | 16 | 203 224 | 4,94 | 253 474 | 403 | 264 649 | 435 | 166 424 | 13,0 | **129 569** | **3,95** |
| coAuthorsDBLP | 64 | 235 127 | 10,5 | 298 102 | 288 | 294 215 | 317 | 190 914 | 12,8 | **161 453** | **4,24** |
| as-22july06 | 2 | 5 433 | **0,061** | 7 417 | 15,4 | 6 938 | 11,0 | 6 015 | 0,337 | **3 579** | 0,068 |
| as-22july06 | 16 | 16 726 | 1,14 | 21 703 | 14,3 | 22 622 | 11,9 | 17 573 | 2,40 | **14 836** | **0,887** |
| as-22july06 | 64 | 21 852 | 2,16 | 27 553 | 9,83 | 27 725 | 11,5 | 22 744 | 3,82 | **20 410** | **1,51** |

Table 2: Results of the evaluation of our coarsening algorithms using KaFFPa's fastsocial configuration and a balance constraint of 3% compared to KaFFPa's state-of-the-art coarsening (default) with equal configuration. Size of the found cut and execution time $t$ are averaged over 5 seeds and best result is highlighted for each instance. The used instances[20] comprise mostly social network graphs (cf. Table 5.1).

| KaFFPa configuration: | | ecosocial | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Coarsening algorithm: | | Basic-NOI | | Pre-Sort-NOI | | Multi-Run-NOI | | SCLaP-NOI | | Default | |
| Graph | $k$ | Cut | $t$ | Cut | $t$ | Cut | $t$ | Cut | $t$ | Cut | $t$ |
| loc-brightkite_edges | 2 | 29 100 | 0,960 | 30 443 | 22,6 | 30 297 | 30,1 | 25 406 | 2,49 | **20 217** | **0,891** |
| loc-brightkite_edges | 16 | 68 372 | 9,54 | 85 514 | 34,5 | 85 578 | 45,8 | 63 373 | 8,25 | **54 976** | **6,81** |
| loc-brightkite_edges | 64 | 82 970 | 55,1 | 90 299 | 82,6 | 89 811 | 96,8 | 80 736 | 36,2 | **74 016** | **34,2** |
| soc-Slashdot0902 | 2 | 137 334 | 1,83 | 77 544 | 2,87 | **64 055** | 21,3 | 89 969 | 5,35 | 123 832 | **1,42** |
| soc-Slashdot0902 | 16 | 291 782 | 24,9 | 283 324 | 22,1 | 280 335 | 40,3 | 265 911 | 20,9 | **265 523** | **14,4** |
| soc-Slashdot0902 | 64 | 311 134 | 205 | 311 946 | 198 | 310 858 | 234 | 305 961 | **108** | **304 787** | 131 |
| p2p-Gnutella04 | 2 | **7 729** | **0,112** | 8 374 | 0,133 | 8 271 | 0,608 | 8 237 | 0,350 | 7 878 | 0,158 |
| p2p-Gnutella04 | 16 | 17 751 | **1,71** | 17 954 | 3,10 | 17 595 | 4,63 | 17 442 | 1,89 | **17 350** | 1,84 |
| p2p-Gnutella04 | 64 | 19 576 | 12,3 | 20 141 | 24,5 | 20 095 | 23,7 | 19 409 | **11,8** | **19 379** | 11,8 |
| amazon-2008 | 2 | 119 154 | 19,0 | 210 814 | 64,0 | 241 330 | 217 | 86 935 | 41,6 | **75 973** | **17,4** |
| amazon-2008 | 16 | 370 241 | 347 | 530 112 | 751 | 503 176 | 1 050 | 273 249 | 212 | **243 069** | **154** |
| amazon-2008 | 64 | 470 278 | 1 040 | 589 492 | 1 810 | 557 004 | 2 060 | 375 773 | 524 | **357 177** | **462** |
| loc-gowalla_edges | 2 | 115 198 | 8,98 | 128 884 | 217 | 123 880 | 308 | 67 361 | 21,5 | **67 262** | **5,44** |
| loc-gowalla_edges | 16 | 330 122 | 104 | 356 148 | 391 | 355 899 | 487 | 266 102 | 70,1 | **243 948** | **62,6** |
| loc-gowalla_edges | 64 | 388 248 | 428 | 411 351 | 612 | 414 668 | 674 | 361 449 | 182 | **338 615** | **172** |
| PGPgiantcompo | 2 | 537 | 0,095 | 780 | 0,247 | 693 | 0,619 | 425 | 0,166 | **372** | **0,066** |
| PGPgiantcompo | 16 | 1 955 | 0,477 | 2 310 | 1,01 | 2 472 | 1,82 | **1 640** | 0,452 | 1 653 | **0,334** |
| PGPgiantcompo | 64 | 3 541 | 0,986 | 4 331 | 1,64 | 4 371 | 2,44 | 3 048 | 0,834 | **3 040** | **0,700** |
| email-EuAll | 2 | 1 283 | **0,373** | 3 532 | 3,80 | 2 108 | 6,77 | 1 685 | 1,29 | **733** | 0,457 |
| email-EuAll | 16 | 30 210 | 3,34 | 23 874 | 6,23 | 24 186 | 8,81 | 24 343 | 3,88 | **23 002** | **1,81** |
| email-EuAll | 64 | 38 222 | 12,6 | 40 591 | 12,7 | 39 648 | 17,9 | 37 008 | 8,28 | **34 576** | **6,10** |
| enron | 2 | 17 913 | 1,54 | 16 057 | 55,1 | 15 151 | 76,0 | **7 300** | 3,07 | 7 577 | **0,652** |
| enron | 16 | 92 331 | 10,4 | 93 164 | 82,8 | 87 516 | 102 | 80 363 | **8,40** | **70 426** | 8,81 |
| enron | 64 | 111 590 | 45,7 | 128 396 | 109 | 124 657 | 139 | 111 177 | **23,0** | **98 835** | 31,0 |
| web-Google | 2 | 33 169 | **8,38** | 65 358 | 29,4 | 69 086 | 110 | 15 334 | 26,2 | **11 249** | 10,1 |
| web-Google | 16 | 49 466 | 229 | 82 776 | 311 | 71 230 | 401 | 29 839 | 76,9 | **24 960** | **31,7** |
| web-Google | 64 | 74 985 | 348 | 91 207 | 453 | 81 341 | 529 | 45 583 | 178 | **39 264** | **127** |
| in-2004 | 2 | 22 582 | 346 | 28 084 | 322 | 29 448 | 1 830 | 5 157 | 466 | **3 524** | **28,8** |
| in-2004 | 16 | 103 052 | 1 400 | 78 813 | 1 110 | 68 525 | 2 700 | 23 824 | 718 | **19 627** | **223** |
| in-2004 | 64 | 196 907 | 1 230 | 180 841 | 1 520 | 154 107 | 3 340 | 48 982 | 810 | **46 137** | **285** |
| coPapersCiteseer | 2 | 410 772 | **14,8** | 401 777 | 28,7 | 442 127 | 1 620 | 320 510 | 105 | **283 863** | 21,5 |
| coPapersCiteseer | 16 | 1 468 874 | 301 | 1 455 448 | 193 | 1 442 222 | 1 790 | 935 175 | 173 | **813 353** | **112** |
| coPapersCiteseer | 64 | 1 587 476 | 1 370 | 1 623 337 | 1 270 | 1 631 819 | 2 940 | 1 184 233 | 736 | **1 066 286** | **630** |
| coAuthorsCiteseer | 2 | 31 639 | 4,16 | 45 418 | 69,8 | 45 518 | 112 | 24 057 | 8,83 | **20 904** | **3,45** |
| coAuthorsCiteseer | 16 | 81 347 | 77,6 | 104 359 | 201 | 98 427 | 259 | 65 814 | 33,3 | **56 042** | **23,2** |
| coAuthorsCiteseer | 64 | 90 490 | 147 | 103 609 | 299 | 102 777 | 377 | 80 126 | 84,3 | **70 687** | **53,8** |
| wordassociation-2011 | 2 | 11 461 | **0,181** | 12 455 | 0,652 | 14 472 | 1,94 | 11 286 | 0,425 | **11 079** | 0,183 |
| wordassociation-2011 | 16 | 33 677 | 2,59 | 41 954 | 3,54 | 41 088 | 5,53 | 33 689 | 2,00 | **31 319** | **1,71** |
| wordassociation-2011 | 64 | 38 955 | 32,0 | 40 013 | 41,5 | 39 758 | 37,6 | 39 961 | 19,9 | **37 642** | **17,3** |
| wiki-Talk | 2 | 158 903 | 162 | 117 752 | 3 490 | 152 375 | 3 530 | 162 547 | 485 | **90 865** | **101** |
| wiki-Talk | 16 | 1 101 769 | 620 | 724 493 | 3 660 | **723 275** | 3 560 | 793 724 | 764 | 813 466 | **449** |
| wiki-Talk | 64 | 1 173 412 | 1 930 | 1 063 649 | 3 230 | 1 056 336 | 3 220 | **982 834** | 928 | 1 071 756 | **639** |
| eu-2005 | 2 | 271 690 | 233 | 370 374 | 86,5 | 344 267 | 1 010 | 43 038 | 1 310 | **20 705** | **50,6** |
| eu-2005 | 16 | 1 214 824 | 1 440 | 1 103 418 | 1 590 | 1 074 643 | 2 540 | 390 826 | 1 810 | **333 862** | **442** |
| eu-2005 | 64 | 2 995 324 | 3 840 | 2 692 259 | 3 240 | 2 637 076 | 4 500 | **2 001 034** | 2 360 | 2 026 402 | **1 250** |
| cnr-2000 | 2 | 1 709 | 50,4 | 2 244 | 61,9 | 2 999 | 170 | 317 | 225 | **291** | **13,2** |
| cnr-2000 | 16 | 59 105 | 94,1 | 35 576 | 84,3 | 34 774 | 201 | 6 711 | 250 | **6 373** | **17,1** |
| cnr-2000 | 64 | 750 833 | 183 | 755 537 | 223 | 750 560 | 366 | **690 489** | 289 | 697 059 | **80,1** |
| citationCiteseer | 2 | 61 965 | 8,44 | 151 994 | 84,4 | 159 660 | 133 | 35 589 | 18,2 | **33 249** | **6,37** |
| citationCiteseer | 16 | 211 957 | 94,5 | 373 897 | 404 | 390 729 | 491 | 156 759 | 67,7 | **143 879** | **52,5** |
| citationCiteseer | 64 | 283 233 | 349 | 386 312 | 773 | 391 417 | 793 | 252 209 | **175** | **229 131** | 177 |
| coPapersDBLP | 2 | 813 098 | **23,5** | 815 794 | 37,2 | 814 895 | 1 030 | 645 090 | 95,8 | **520 158** | 30,4 |
| coPapersDBLP | 16 | 2 491 240 | 201 | 2 523 143 | 226 | 2 546 747 | 1 230 | 1 625 213 | 186 | **1 457 767** | **124** |
| coPapersDBLP | 64 | 2 655 144 | 1 400 | 2 847 545 | 1 420 | 2 810 537 | 2 450 | 2 124 552 | 749 | **1 908 572** | **653** |
| as-skitter | 2 | 599 433 | 99,7 | 624 201 | 562 | 642 487 | 913 | 325 066 | 344 | **252 091** | **21,1** |
| as-skitter | 16 | 1 398 298 | 860 | 1 877 706 | 2 730 | 1 879 191 | 2 800 | 1 093 905 | 629 | **964 291** | **239** |
| as-skitter | 64 | 1 999 135 | 2 340 | 2 234 653 | 3 470 | 2 241 030 | 3 810 | 1 870 227 | 1 210 | **1 671 851** | **749** |
| coAuthorsDBLP | 2 | 79 970 | 24,6 | 105 859 | 363 | 103 661 | 387 | 66 531 | 15,3 | **47 385** | **7,13** |
| coAuthorsDBLP | 16 | 172 685 | 79,9 | 240 183 | 493 | 237 889 | 565 | 147 717 | 51,7 | **120 480** | **45,2** |
| coAuthorsDBLP | 64 | 194 736 | 266 | 235 965 | 665 | 233 478 | 713 | 179 207 | 150 | **154 008** | **110** |
| as-22july06 | 2 | 5 023 | 0,306 | 7 011 | 16,4 | 6 563 | 12,3 | 5 586 | 0,816 | **3 516** | **0,212** |
| as-22july06 | 16 | 15 633 | 4,32 | 20 278 | 16,9 | 19 931 | 14,8 | 16 619 | 4,07 | **14 409** | **2,53** |
| as-22july06 | 64 | 20 889 | 13,7 | 25 641 | 16,1 | 25 587 | 18,6 | 21 693 | 9,40 | **19 821** | **7,46** |

Table 3: Results of the evaluation of our coarsening algorithms using KaFFPa's ecosocial configuration and a balance constraint of 3% compared to KaFFPa's state-of-the-art coarsening (default) with equal configuration. Size of the found cut and execution time $t$ are averaged over 5 seeds and best result is highlighted for each instance. The used instances[20] comprise mostly social network graphs (cf. Table 5.1).