High Quality Graph Partitioning

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie

genehmigte

Dissertation

von

Christian Schulz

aus Berlin

Tag der Einreichung:	23. Mai 2013
Tag der mündlichen Prüfung:	5. Juli 2013
Erster Gutachter:	Prof. Dr. Peter Sanders
Zweiter Gutachter:	Prof. Dr. Burkhard Monien

To my parents.

Acknowledgements

Nine years ago I started my studies of computer science and mathematics here in Karlsruhe. These years have been a blast which leaves me with the feeling that a few lines are not enough to thank everyone who has made my time here so enjoyable.

For the purpose of my diploma thesis, my advisor Prof. Dr. Peter Sanders introduced me five years ago to the incredibly interesting problem of partitioning graphs into blocks of roughly equal size such that some quality metric is minimized. In order to efficiently obtain a partition of a graph one has to use many colorful algorithms and algorithm design patterns. I would like to thank you, Peter, for introducing the problem to me and for all the great opportunities that you gave me during these past years. I am especially grateful for the opportunity to work in your group, for the freedom I could work in and for the possibility of meeting many computer science legends, such as Andrew Goldberg, David Johnson, George Karypis, Kurt Mehlhorn and Robert Tarjan, to name only a few of them. Thank you, Peter!

My thanks also go to my office mates Moritz Kobitzsch and Jochen Speck, as well as my other colleagues Veit Batz, Timo Bingmann, Jonathan Dees, Robert Geisberger, Johannes Fisher, Dennis Luxen, Ingo Müller, Vitaly Osipov, Dennis Schieferdecker, Nodari Sitchinava, and Johannes Singler. We had a lot of interesting work and non-work related discussions. I would like to thank all my co-authors with whom I enjoyed writing joint papers together: David Bader, Marcel Birn, Daniel Delling, Robert Görke, Jonas Fietz, Vincent Heuveline, Manuel Holtgrewe, Andrea Kappes, Matthias Krause, Henning Meyerhenke, Vitaly Osipov, Ilya Safro, Peter Sanders, Nodari Sitchinava, and Dorothea Wagner. Besides that I would like to thank Petra Mutzel for inviting me to Dortmund. I also thank Renato Werneck for the invitation to his session at the ISMP'12 in Berlin, Ilya Safro for the invitation to a workshop in Salt Lake City, and Anand Srivastav for the invitation to Agra. Moreover, I would like to thank Timo Bingmann and Henning Meyerhenke for proofreading this thesis.

I am incredibly grateful to my parents – Manuela and Jochen Grüner – as well as my stunning girlfriend Olga Jochim. Thank you for all your love and support! Without you this work would not have been possible. Of course I would like to thank my three best friends: Fabian Götz, Tobias Flaig, and Mark Hudek – thank you for being there.

I would like to give thanks to the Steinbuch Centre for Computing for the access to various high performance clusters. Last but not least I would like to thank Chris Walshaw for maintaining his Graph Partitioning benchmark and the inventors of vim for creating a wonderful editor which was used to write this thesis and to implement the open source release that comes with this work.

Abstract

In computer science, engineering, and related fields, *graph partitioning* is a common technique. For example, in parallel computing good partitionings of unstructured graphs are very valuable. In this area, graph partitioning is mostly used to partition the underlying graph model of computation and communication. Roughly speaking, nodes in this graph denote computation units, and edges represent communication. This graph needs to be partitioned such that there are few edges between the blocks (pieces). In particular, if we want to use k processors we want to partition the graph into k blocks of about equal size. In this work we focus on a version of the problem that constrains the maximum block size to $(1+\varepsilon)$ times the average block size and tries to minimize the total cut size, i.e. the number of edges that run between blocks.

A successful heuristic for partitioning large graphs is the *multilevel graph partitioning* approach, where the graph is recursively *contracted* to create smaller graphs which should reflect the same basic structure as the input graph. After applying an *initial partitioning* algorithm to the smallest graph, the contraction is undone and, at each level, a *local search* method is used to improve the partitioning induced by the coarser level.

Although several successful multilevel partitioners have been developed in the last two decades, we had the impression that certain aspects of the method are not well understood. We therefore have built our own graph partitioner KaPPa [87] (Karlsruhe Parallel Partitioner) with a focus on scalable parallelization. Somewhat astonishingly, we also obtained improved partitioning quality through rather simple methods. This motivated us to make a fresh start, putting all aspects of multilevel graph partitioning on trial. Our focus in this work is on solution quality and sequential speed for large graphs. We contribute a number of improvements which lead to enhanced partitioning quality. This includes an integration of several improved coarsening schemes, flow methods, improved local search, repeated runs similar to the approaches used in multigrid solvers, a distributed evolutionary algorithm, and a novel algorithm for the highly balanced case.

First we present multilevel graph partitioning algorithms which are bundled in the framework called KaFFPa (Karlsruhe Fast Flow Partitioner). We compare different matching-based and algebraic multigrid-inspired coarsening schemes, experiment with algebraic distance, and demonstrate computational results on several classes of graphs that emphasize the running time and quality advantages of different coarsening schemes. We then look at two novel local improvement schemes, i.e. algorithms that move nodes between the blocks of a partition in order to reduce the cut. The first scheme is, in contrast to previous techniques, very localized and the second scheme is based on iterative *max-flow min-cut* computations in areas around the cut of a partitioned graph. Overall

this leads to a system that for many common benchmarks achieves both high quality solutions and favorable tradeoffs between running time and solution quality.

We then present meta-heuristics for the graph partitioning problem. Here, we start by looking at advanced global search strategies – *iterated multilevel algorithms*. The V-cycle scheme has been introduced by Walshaw [162] and can be traced back to multigrid solvers for sparse systems of linear equations. The main idea is to iterate the coarsening and uncoarsening phase. Once the graph is partitioned, edges that are between two blocks are not contracted. We look at two further strategies and are able to show that iterated multilevel algorithms are superior to multiple restarts of the multilevel algorithm if a weak local search algorithm is used. Further we contribute a novel distributed evolutionary algorithm, KaFFPaE (KaFFPaEvolutionary), in order to tackle the problem. KaFFPaE uses KaFFPa to provide new effective combine and mutation operators. This is combined with a scalable communication protocol. KaFFPaE is able to compute partitions that have quality comparable to or better than previous entries in Walshaw's benchmark archive *within a few minutes* for graphs of moderate size. Previous methods of Soper et al. [157] required running times of up to one week for graphs of that size.

The presented algorithms are able to compute partitions of very high quality in a reasonable amount of time when some imbalance $\varepsilon > 0$ is allowed. However, they are not very good for strict balance constraints such as the case $\varepsilon = 0$. In this case, state-of-the-art local search algorithms are restricted to finding nodes to be exchanged between a pair of blocks in order to decrease the cut *and* to maintain balance. We introduce new techniques that relax the balance constraint for node movements but globally maintain balance by combining multiple local searches. The combination problem is reduced to finding negative cycles in a directed graph, exploiting the existence of efficient algorithms for this problem. This is combined with an algorithm to balance unbalanced solutions and is integrated into our evolutionary algorithm.

The perspective taken in this work is that we develop our graph partitioners KaFFPa, KaFFPaE, and KaBaPE (Karlsruhe Balanced Partitioner Evolutionary) in a benchmark driven way, achieving a system that has been able to improve or reproduce *most* of the entries reported in the Walshaw benchmark. Another equally valid perspective is that we have applied the methodology of algorithm engineering to all aspects of the multilevel graph partitioning approach, achieving improvements in coarsening, local search, parallelization, global search guidance, and embedding into meta-heuristics.

Our partitioners also work very well on the instances of the 10th DIMACS Implementation Challenge on Graph Partitioning and Clustering, achieving the *best marks* both with respect to quality and running time versus quality among all participants. A surprising result was obtained for a part of the challenge, where the objective function was not cut size but a measure of communication volume. This objective function can be expressed as a hypergraph partitioning problem. Interestingly, KaFFPaE outperformed dedicated hypergraph partitioners by just changing the fitness function to prefer solutions with low communication volume – the multilevel algorithm still optimizes cuts. The algorithms developed within this work have been released as an open source project.

Contents

1

Introduction

1.1 Motivation

It is quite fascinating that the problem of dividing a graph into a given number of blocks having roughly equal size, such that some objective function is minimized, literally has applications everywhere. For example, solving the graph partitioning problem can help to balance load and minimize communication in scientific simulations [150, 35, 69], can speed up Dijkstra's Algorithm [108, 120], and in general is an useful technique in the route planning area [111, 100, 48]; it supports VLSI design [7, 8], and also can preserve sparsity in Gaussian elimination on sparse symmetric positive definite matrices [74].

Probably the best known application of graph partitioning is the numerical solution of *partial differential equations* on a highly parallel computer. Here, a continuous simulation space is discretized by a fine mesh. Solving the partial differential equation then becomes an iterative process. In each iteration, all the mesh points are updated using neighboring values in the mesh. Outputs from one iteration serve as inputs for the next. Informally speaking, nodes in the mesh denote computation units and edges represent communication. To achieve high accuracy of the approximation, the number of nodes can become quite large, so that either the time to solve the system is immense or the mesh does not fit into the main memory of a single system. Therefore, to still get an approximation in a reasonable amount of time, parallel computing and graph partitioning comes into play. After we have build a graph model of computation and communication, we can solve the graph partitioning problem to equally distribute the work on k available processors of the supercomputer and minimize the communication overhead. By doing so, we get an efficient parallel computation scheme for approximating the solution of the differential equation.

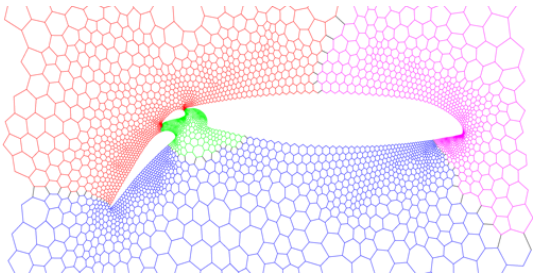


Figure 1.1: An example mesh.

What makes the problem even more appealing is the fact that the problem is NP-complete [90, 73] for most objective functions and that there is no constant factor approximation on general graphs [34], if the objective is to minimize the number of edges that run between blocks. Hence, mostly heuristics are used in practice to partition graphs.

These heuristics usually use an astonishingly large set of "easier" graph algorithms to tackle the problem. For example, algorithms such as weighted matching, spanning trees, edge coloring, breadth-first search, dominating sets, maximum flows, diffusion, negative cycle detection, shortest paths, and strongly connected components.

Probably the most successful heuristic for partitioning large graphs is the *multilevel graph partitioning* approach which was initially introduced to the graph partitioning field in the nineties by Barnard and Simon [18] to speed up spectral partitioning techniques. Later Hendrickson and Leland [85] formulated the multilevel approach as it is known today. The graph is recursively *contracted* to create smaller graphs which should reflect the same basic structure as the input graph. This is achieved by modifying edge and node weights of the coarser graphs. Often the weight of a coarse node is the number of the contracted nodes that it represents and the weight of an edge stands for the number of parallel edges that it replaces. This way, a partitioning of a coarse level creates a partitioning of the finer graphs having the same objective and balance. After applying an *initial partitioning* algorithm to the smallest graph, the contraction is undone and, at each level, a *local search* method is used to improve the partitioning induced by the coarser level. The intuition behind this approach is that a good partition at one level of the hierarchy will also be a good partition on the next finer level so that local search converges quickly, i.e. rapidly finds a good solution. On the other hand, local search has a somewhat global view on the optimization problem on the coarse levels of the multilevel approach, whereas it has very fine view on the fine levels of the hierarchy.

A few years ago, partly during my diploma thesis [152], we started to build our own parallel graph partitioner KaPPa (Karlsruhe Parallel Partitioner) [87] since we had the impression that certain aspects of the multilevel method are not well understood – although several successful multilevel partitioners have been developed in the last two decades. Our focus in KaPPa was on scalable parallelization. Somewhat astonishingly, we also obtained improved partitioning quality through rather simple methods. This motivated us to make a fresh start, putting all aspects of multilevel graph partitioning on trial focusing on solution quality and sequential speed for large graphs in this work.

Solution quality is of major importance in applications such as VLSI Design [7, 8] where even minor improvements in the objective can have a large impact on the production costs and quality of a chip. Indeed, high quality solutions are also favorable in applications where the graph needs to be partitioned only once so that the partition can be used over and over again and the running time of the graph partitioning algorithms is only a minor issue [108, 120, 111, 100, 48, 69]. Thirdly, high quality solutions are even important in areas in which the running time overhead is paramount [157], such as finite element computations [150] or the direct solution of sparse linear systems [74]. Here, high quality graph partitions can be useful for benchmarking purposes. However, due

to the parallelization of our systems, we are able to compute partitions that have quality comparable to or better than previous entries in Walshaw's well-known partitioning benchmark *within a few minutes* for graphs of moderate size. Previous methods of Soper et al. [157] required running times of up to one week for graphs of that size. We therefore believe that in contrast to previous methods, our systems can be very valuable in the area of high performance computing. Moreover, we believe that many of the improvements presented in this work are transferable to other combinatorial graph problems, such as hypergraph partitioning, graph drawing and graph clustering.

1.2 Main Contributions

We start by presenting several advances to the *multilevel graph partitioning scheme*. This includes multiple coarsening schemes to create graph hierarchies, algebraic distance as a measure of connectivity, an initial partitioning algorithm as well as two new local search algorithms: a very localized local search algorithm and an algorithm that is based on multiple iterations of max-flow min-cut computations between pairs of blocks of a given partition. We then contribute several *meta-heuristics* for the graph partitioning problem. We start by looking at advanced global search strategies – iterated multilevel algorithms. The iterated V-cycle scheme has been introduced by Walshaw [162] and can be traced back to multigrid solvers for sparse systems of linear equations. The main idea is to iterate the coarsening and uncoarsening phase multiple times. Once the graph is partitioned, edges that are between two blocks are not contracted. We look at two further strategies, F-cycles and W-cycles, and show that iterated multilevel algorithms are superior to multiple restarts of the multilevel algorithm, if a weak local search algorithm is used. We obtain a system, KaFFPa (Karlsruhe Fast Flow Partitioner), that can be configured to either achieve the best known partitions for many standard benchmark instances, or to be the fastest available system for some large graphs while still improving partitioning quality compared to the previous fastest system. Further, we emphasize the success of the proposed algebraic multigrid coarsening and the algebraic distance connectivity measure between nodes on highly irregular instances. Our experimental evaluations of KaFFPa focus mostly on the number of cut edges. However, we also look at the maximum communication volume and the size of node separators.

To further improve solution quality, we continue by presenting a novel distributed evolutionary algorithm, KaFFPaE (KaFFPaEvolutionary). In KaFFPaE, we have a general combine operator framework, i.e. a partition $\mathscr{P}$ can be combined with another partition or an arbitrary clustering of the graph. Inspired by the V-cycle, this is achieved by running a modified version of KaFFPa that will not contract edges that are cut in one of the input partitions/clusterings. The framework ensures that the resulting partition is at least as good as the input partition $\mathscr{P}$ and in addition, the local search algorithms can effectively exchange good parts of the solution on the coarse levels by moving only a few nodes. We combine this with a scalable communication protocol similar to randomized

rumor spreading and obtain a system that on the one hand scales well to large networks and machines and on the other hand is able to improve the best known partitioning results for many inputs in a short amount of time.

KaFFPa and KaFFPaE also work very well on the instances of the 10th DIMACS Implementation Challenge on Graph Partitioning and Clustering achieving the *best marks* – and thus winning the graph partitioning subchallenges – both with respect to partition quality and running time versus quality among all participants. In this work, we also present the results obtained during the challenge.

The outlined algorithms are able to compute partitions of very high quality in a reasonable amount of time when some imbalance $\varepsilon > 0$ is allowed. However, they are not very good for the highly balanced case. In this case, state-of-the-art local search algorithms are more or less restricted to finding pairs of nodes which have to be exchanged between blocks in order to decrease the cut and to maintain balance. We contribute novel local search techniques for the highly balanced case, including the perfectly balanced case $\varepsilon = 0$, that from a meta-heuristic point of view increase the neighborhood of a perfectly balanced solution in which local search is able to find better solutions. More precisely, these techniques encode local searches that are not restricted to a balance constraint into a small directed graph allowing us to find combinations of these searches that maintain balance. Such combinations are found by applying a negative cycle detection algorithm on the graph. We combine these techniques with an algorithm to balance unbalanced solutions and integrate it into our evolutionary algorithm, KaFFPaE. Our experiments show that the proposed techniques are also helpful if some imbalance is allowed. The obtained system, KaBaPE (Karlsruhe Balanced Partitioner Evolutionary), has been able to improve or reproduce *most* of the best known highly balanced partitioning results reported in the Walshaw benchmark.

We describe multiple algorithmic extensions for our algorithms. For the purpose of partitioning road networks, we present Buffoon. We use natural cuts by Delling et al. [49] as a pre-processing technique to obtain a much smaller version of the graph that has similar cuts as the original road network. Partitions of the smaller graph correspond to partitions of the original graph so that we use KaFFPaE to partition the smaller version of the graph. Buffoon computes partitions of road networks that have comparable to or better quality than those produced by PUNCH [49]. Moreover, we present a preprocessing technique tailored to partition large social networks with high quality. The main idea is to compute a size-constrained clustering of the graph, contract it, and then to apply our previously developed partitioning algorithms on the contracted graph. The presented algorithm is the first algorithm to partition a social network with billions of edges on a single machine with 64 GB main memory. In both cases, the smaller graph is up to orders of magnitude smaller than the input road network which speeds up the computations our partitioners drastically. Our last algorithmic extension can compute a k-way separator from a given k-way partition.

1.3 Outline

We begin in Chapter 2 by presenting preliminaries and basic concepts that are used throughout this work. We continue by elaborating related work in Chapter 3. Multilevel algorithms and global search strategies are examined in Chapter 4. Chapter 5 covers evolutionary graph partitioning techniques. We contribute specialized algorithms for the highly balanced case of the problem in Chapter 6. Chapter 7 compares our partitioners against state-of-the-art partitioning libraries, while Chapter 8 covers easy algorithmic extensions, e.g. for the partitioning of road networks, the partitioning of large social networks, or a post-processing technique to obtain node separators from a given partitioning. Conclusions are given in the respective chapters and in the last chapter of this work – Chapter 9. Appendix A evaluates the performance of the algebraic multigrid inspired coarsening scheme.

2

Preliminaries

In this chapter, we present the basic concepts that are used in this work.

2.1 Graphs and Related Problems

A weighted (directed) *graph* G consists of a set of nodes V and a set of edges $E \subset V \times V$ to represent relations between the nodes, as well as two cost functions. One function assigns weights to the nodes $c : V \to \mathbb{R}_{>0}$ and a second function $\omega : E \to \mathbb{R}$ assigns costs to the edges. In general, we write n for the number of nodes and m for the number of edges. In an undirected graph an edge $(u,v) \in E$ implies an edge $(v,u) \in E$ and that both edge weights are equal. We use the set notation $\{u,v\} \in E$ in the undirected case. We extend c and ω to sets, i.e. $c(V')$:= $\sum_{v \in V'} c(v)$ and $\omega(E')$:= $\sum_{e \in E'} \omega(e)$. The set $\Gamma(u)$:= $\{v : \{u,v\} \in E\}$ denotes the neighbors of a node u. The *degree* of a node is the number of its neighbors. With Δ we denote the *maximum degree* of a graph. The weighted degree of a node is the sum of the weights of its incident edges. A graph is *bipartite* if its node set can be divided into two disjoint sets U and V such that $\{u,v\} \in E$ implies $u \in U$ and $v \in V$ or vice versa. A *subgraph* is a graph whose node and edge set are subsets of another graph. We call a subgraph induced if it has every possible edge.

A *matching* $M \subseteq E$ is a set of edges that do not share any common nodes, i.e. the graph (V,M) has maximum degree one. The weight of a matching is defined as the weight induced by its edges $\omega(M)$. A matching is said to be *maximal* if there is no edge that can be added to the matching, and a matching that has maximum weight among all matchings is called a *maximum weight matching*. For a graph a subset $C \subseteq V$ is a *closed node set* if and only if for all nodes $u,v \in V$, the conditions $u \in C$ and $(u,v) \in E$ imply $v \in C$. In other words, a subset C is a *closed node set* if there is no edge starting in C and ending in its complement $V \setminus C$. A subset $D \subseteq V$ is a *dominating set* if for each $v \in V$ either v itself or one of its neighbors is contained in D. D is called an *independent set* if the nodes of D don't share an edge.

A sequence of nodes $s \to \cdots \to t$ such that each pair of consecutive nodes is connected by an edge, is called an *s-t path*. We say that s is the source and t is the target.

The length of a path is defined by the sum of its edge weights. A shortest *s-t* path is a path with the smallest weight among all *s-t* paths. A path with equal source and target is called a *cycle*. It is *simple* if no node is contained twice. A cycle with negative weight is also called *negative cycle*. A directed graph is *strongly connected* if there is a *u-v* path and a *v-u* path for each pair of nodes u, v. A maximal strongly connected induced subgraph is called *strongly connected component*. These concepts are transferred straightforwardly to undirected graphs. However, the term *connected component* is used instead. In a directed graph, a linear ordering $\prec$ of the nodes such that an edge $(u,v) \in E$ implies that $u \prec v$ in the ordering is called a *topological order*.

In a directed graph with two designated nodes, a source s and a sink t, that has *non-negative* edge weights (serving as capacities), a *flow* is a function $f : V \times V \to \mathbb{R}$ that satisfies a capacity constraint, a flow conservation constraint and a skew-symmetry constraint. The *capacity constraint* demands that the flow value associated is lower or equal to the capacity ($f(u,v) \leq \omega(u,v)$) and the *flow conservation constraint* requires that each node emits the same amount of flow as it receives – except the two designated nodes source s and sink t. The *skew symmetry constraint* requests $f(u,v) = -f(v,u)\ \forall (u,v) \in V \times V$. $f(u,v)$ denotes the amount of flow on an edge (u,v). The value of a flow val(f) is defined as the total amount of flow that is transferred from the source to the sink. The *residual capacity* is defined as $r_f(u,v) = \omega(u,v) - f(u,v)$. The *residual graph* for a directed graph $G = (V,E)$ and a flow f is given as $G_f = (V,E_f)$ where $E_f = \{(u,v) \in V \times V \mid r_f(u,v) > 0 \text{ and } (u,v) \in E \text{ or } (v,u) \in E\}$.

An *s-t cut* is defined as tuple $(S, V \setminus S)$ with $s \in S \subset V$ and $t \in V \setminus S$. The weight of an *s-t* cut is defined as $\sum_{(u,v) \in E \cap S \times V \setminus S} \omega(u,v)$, i.e. the weight of the edges starting in S and ending in $V \setminus S$. A minimum *s-t* cut has the smallest weight among all *s-t* cuts. It is well-known that the value of a maximum *s-t* flow corresponds to the value of a minimum *s-t* cut, and if a maximum *s-t* flow is given, then a minimum cut is easily computed.

2.2 Partitions and Clusterings

Given a number $k \in \mathbb{N}_{>1}$ and an undirected graph with *non-negative* edge weights, the *graph partitioning problem* asks for *blocks* of nodes $V_1,\ldots,V_k$ that partition the node set V, i.e.

$$1.\ V_1 \cup \cdots \cup V_k = V$$
$$2.\ V_i \cap V_j = \emptyset\ \forall i \neq j.$$

A *balance constraint* demands that all blocks have about equal size. More precisely, it requires that, $\forall i \in \{1..k\} : |V_i| \leq L_{\max} := (1+\varepsilon)\lceil |V|/k \rceil$ for some imbalance parameter $\varepsilon \in \mathbb{R}_{\geq 0}$ in the case that the cost function of the nodes is identical to one. In the case of $\varepsilon = 0$, we also use the term *perfectly balanced*. A block V_i is *underloaded* if $|V_i| < L_{\max}$ and *overloaded* if $|V_i| > L_{\max}$. A *clustering* is also a partition of the nodes, however k is usually not given in advance and the balance constraint is removed. Note that a partition

is also a clustering of a graph. In both cases, the *goal* is to minimize or maximize a particular objective function. We introduce two well-known objective functions for the partitioning problem in the next section. A node $v \in V_i$ that has a neighbor $w \in V_j$, $i \neq j$, is a *boundary node*. An edge that runs between blocks is also called *cut edge*. The set $E_{ij} := \left\{\{u,v\} \in E : u \in V_i, v \in V_j\right\}$ is the set of cut edges between two blocks V_i and V_j. An abstract view of the partitioned graph is the so called *quotient graph*, where nodes represent blocks and edges are induced by connectivity between blocks, i.e. there is an edge in the quotient graph if there is an edge that runs between the blocks in the original, partitioned graph. An example is given in Figure 2.1. Given two clusterings $\mathscr{C}_1$ and $\mathscr{C}_2$, the *overlay clustering* is the clustering where each block corresponds to a connected component of the graph $G_{\mathscr{E}} = (V, E \backslash \mathscr{E})$ where $\mathscr{E}$ is the union of the cut edges of $\mathscr{C}_1$ and $\mathscr{C}_2$, i.e. all edges that run between blocks in either $\mathscr{C}_1$ or $\mathscr{C}_2$.

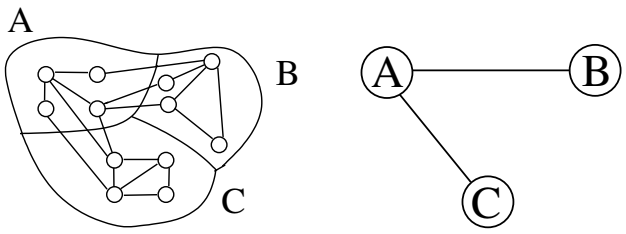


Figure 2.1: A graph that is partitioned into three blocks of size four on the left and its corresponding quotient graph on the right. There is an edge in the quotient graph if there is an edge between the corresponding blocks in the original graph.

2.3 Objective Functions

In practice, we often seek to find a partition that minimizes (or maximizes) an objective. Probably the most prominent objective function is to minimize the *total cut*

$$\sum_{i<j} \omega(E_{ij}).$$

It is well-known that there are more realistic (and more complicated) objective functions involving also the block that is worst and the number of its neighboring nodes [84], but minimizing the cut size has been adopted as a kind of standard, since it is usually highly correlated with the other formulations. We believe that the results presented in this work are adaptable to other objective functions and also to other settings such as graph clustering. Hence, by default we *minimize the total cut*.

The *second objective* that we investigate, has a closer look at the communication volume and was used in a subchallenge of the 10th DIMACS Challenge on Graph Partitioning and Graph Clustering [16]. For a block V_i, the communication volume is defined as $\text{comm}(V_i) := \sum_{v \in V_i} c(v)D(v)$, where $D(v)$ denotes the number of different blocks in

which v has a neighbor node, excluding V_i. The *maximum communication volume* is then defined as $\max_i \text{comm}(V_i)$, which should be minimized.

Thirdly, we look at the size of node separators. The *node separator problem* asks to find three subsets A, B and S, such that there are no edges between A and B. The objective is to minimize the size of the separator S or depending on the application the weight of its nodes $c(S)$ while A and B are balanced. Note that removing the set S from the graph results in at least two connected components.

The *expansion* of a not necessarily balanced, non-trivial cut (V_1, V_2) is defined as

$$\frac{\omega(E_{12})}{\min(c(V_1), c(V_2))}.$$

Similarly, the *conductance* of such a cut is defined as

$$\frac{\omega(E_{12})}{\min(\text{vol}(V_1), \text{vol}(V_2))},$$

where $\text{vol}(S) := \sum_{v \in S} d(v)$ denotes the volume of the set S. Note that the problem that asks to find a non-trivial cut with minimum conductance or expansion, does not directly enforce a balanced cut as in the balanced graph partitioning problem.

2.4 Instances

Throughout this work we present experiments on various kinds of graphs. In this section, we summarize the main properties, the source, and the area of application of the graphs. The graphs stem from different applications: finite element methods (FEM), street networks, geometric graphs, sparse matrices, social networks and web graphs. Table 2.2 presents the properties of the graphs. All of the graphs presented here have unit edge and node weights. Our default value for the allowed imbalance is 3%, since this is one of the values used in the Walshaw benchmark [157] and the default value in Metis [95]. When reporting average values of different algorithm configurations, we report the geometric mean of the average results on the instances (graph, k) under consideration.

2.4.1 Graph Families

rgg$X$ is a *random geometric graph* with 2^X nodes where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost connected. The largest graph of this class is rgg24. The graphs are taken from [87] and are available for download at the 10th DIMACS Implementation Challenge [16].

Delaunay$X$ is the Delaunay triangulation of 2^X random points in the unit square. The largest graph of this class is del25. The graphs are taken from [87] and are available for download at the 10th DIMACS Implementation Challenge [16].

graph	n	m	Rf.
Random Geometric Graphs			
rgg15	2^{15}	160 240	[17]
rgg16	2^{16}	342 127	[17]
rgg17	2^{17}	728 753	[17]
rgg18	2^{18}	≈1.5M	[17]
rgg20	2^{20}	≈6.9M	[17]
Graphs from Numeric Simulations			
3elt	4 720	13 722	[161]
4elt	15 606	45 878	[161]
fe_sphere	16 386	49 152	[161]
cti	16 840	48 232	[161]
cs4	33 499	43 858	[161]
fe_pwt	36 519	144 794	[161]
fe_body	45 087	163 734	[161]
t60k	60 005	89 440	[161]
wing	62 032	121 544	[161]
finan512	74 752	261 120	[161]
fe_tooth	78 136	452 591	[161]
fe_rotor	99 617	662 431	[161]
598a	110 971	741 934	[161]
fe_ocean	143 437	409 593	[161]
144	144 649	≈1.1M	[161]
wave	156 317	≈1.1M	[161]
m14b	214 765	≈1.7M	[161]
auto	448 695	≈3.3M	[161]
m6	≈3.5M	≈10.5M	[17]
as365	≈3.8M	≈11.3M	[17]
nlr	≈4.2M	≈12.5M	[17]
htric00	≈6.6M	≈9.9M	[17]
hbubbl10	≈18.3M	≈27.5M	[17]
VLSI Graphs			
memplus	17 758	108 384	[161]
g3circuit	≈1.6M	≈3.0M	[46]
Sparse Matricies			
bcsstk29	13 992	605 496	[161]
bcsstk32	44 609	≈2.0M	[161]
af_shell9	504 855	≈8.5M	[46]
af_shell10	≈1.5M	≈25.6M	[46]
kktpower	≈2.1M	≈6.5M	[46]
nlpkkt160	≈8.4M	≈110M	[46]

graph	n	m	Rf.
Delaunay Triangulations			
delaunay15	2^{15}	98 274	[17]
delaunay16	2^{16}	196 575	[17]
delaunay17	2^{17}	393 176	[17]
delaunay18	2^{18}	786 396	[17]
delaunay20	2^{20}	≈3.1M	[17]
Social Networks			
p2p-gnu04	6 405	29 215	[109]
PGPgcomp	10 680	24 316	[109]
email-EuAll	16 805	60 260	[109]
as-22july06	22 963	48 436	[46]
slashdot0902	28 550	379 445	[109]
loc-brightkite	56 739	212 945	[109]
loc-gowalla	196 591	950 327	[109]
coAutCiteseer	227 320	814 134	[17]
wiki-talk	232 314	≈1.5M	[109]
citCiteseer	268 495	≈1.2M	[17]
coAutDBLP	299 067	977 676	[17]
coPapCiteseer	434 102	≈16.0M	[17]
coPapDBLP	540 486	≈15.2M	[17]
as-skitter	554 930	≈5.8M	[109]
Road Networks			
uk	4 824	6 837	[161]
luxemburg	114 599	119 666	[17]
bel	463 514	591 882	[51]
nld	893 041	≈1.1M	[51]
deu	≈4.4M	≈10.9M	[51]
great-britain	≈7.7M	≈8.2M	[17]
asia	≈12.0M	≈12.7M	[17]
eur	≈18.0M	≈44.4M	[51]
Web Graphs			
web-google	356 648	≈2.1M	[109]
uk-2002	≈18.5M	≈262M	[104]
uk-2007-05	≈106M	≈3.3BN	[104]
Kronecker Graphs			
k500-s-log17	131 072	≈5.1M	[17]
k500-s-log21	≈2.1M	≈91.0M	[17]
Erdös Rényi Graph			
er-f1.5-s23	≈8.4M	≈100M	[17]

Table 2.2: Basic properties of the graphs from our benchmark set (m number of undirected edges). The instances are grouped roughly by their application area. Within their groups, the graphs are sorted by the number of nodes.

2.4.2 Sources

The graphs in Table 2.2 stem from different sources: the 10th DIMACS Implementation Challenge [17, 16], the Florida Sparse Matrix Collection [46], the Laboratory of Web Algorithms [104], the Stanford Large Network Dataset Collection [109] and the Walshaw Benchmark [161]. Most of the graphs are available for download at the website of the 10th DIMACS Implementation Challenge [17, 16] in the same graph format that is used by Chaco [83], Metis [95] and Scotch [127].

2.5 Machines

We now describe the machines that are used in the following chapters.

Machine A is a cluster with 200 nodes where each node is equipped with two Intel Xeon X5355 Quad-Core processors which run at a clock speed of 2.667 GHz. Each node has 16 GB local memory and 2x4 MB of L3-Cache. All nodes are attached to an InfiniBand 4X DDR interconnect which is characterized by its very low latency of below 2 microseconds and a point to point bandwidth between two nodes of more than 1300 MB/s. This machine is used in Chapter 4, Chapter 5 and Chapter 6. This machine has been replaced at the end of 2012 by machine B.

Machine B is a cluster with 400 nodes where each node is equipped with two Intel Xeon E5-2670 Octa-Core processors (Sandy Bridge) which run at a clock speed of 2.6 GHz. Each node has 64 GB local memory, 20 MB L3-Cache and 8x256 KB L2-Cache. All nodes have local disks and are connected by an InfiniBand 4X QDR interconnect which is characterized by its very low latency of about 1 microsecond and a point to point bandwidth between two nodes of more than 3700 MB/s. This machine is used in Chapter 7 and Chapter 8.

Machine C has two Quad-Core Intel Xeon X5550 processors which run at a clock speed of 2.67 GHz. It has 48 GB local memory 2x4 MB L3-Cache and 4x256 KB L2-Cache. This machine is used in Chapter 6 and Chapter 8.

3

Related Work

In this chapter we give a brief overview of the work that has been done on graph partitioning. There has been a *huge* amount of research on graph partitioning so that we refer the reader to [70, 150, 23] for most of the material. Hence, in this chapter we try to focus on selected related work and recent coordinate free techniques that have not yet been covered by these papers. There are further methods like space-filling curves [130, 131, 76, 149], methods that handle graphs having geometrical information [22, 76, 82, 119, 134], simulated annealing [92] and ant-colony optimization [103, 44] or parallel approaches to graph partitioning [94, 167, 164, 41, 42, 87, 136] which are not discussed here. Moreover, at this place we restrict ourselves to the techniques that we did not integrate into our systems or only present an abstract view. Techniques that we integrated into our systems are explained in detail in the corresponding chapters.

For historic reasons the chapter is organized as follows: we start by outlining local search algorithms such as the Kernighan-Lin and the Fiduccia-Mattheyses algorithm in Section 3.1. We continue in Section 3.2 with the illustration of methods to obtain an initial partition and explain the multilevel method in Section 3.3. Evolutionary algorithms are covered in Section 3.4, and we elaborate on more advanced concepts such as flow-based methods for improving conductance or expansion cuts in Section 3.5. We continue with an overview of hardness and approximation results in Section 3.6 and exact methods in Section 3.7. Section 3.8 reports the rules of the well-known Walshaw Benchmark. We finish this chapter with a description of available software packages in Section 3.9.

3.1 Local Search

In general, given a partition of a graph, a *local search algorithm* aims to improve an objective function (such as the number of edges that run between blocks) by moving nodes between the blocks. With the application of improving the paging properties of computer programs in mind, Kernighan and Lin [98] were probably the first that defined the graph partitioning problem and worked on local improvement methods for this problem. In his PhD thesis [99] from 1969, Kernighan states *"A program [...] can be*

thought of as [...] a set of connected entities. The entities might be subroutines, [...]. The connections between the entities might represent [...] references by one entity to another. The problem is to assign the objects to "pages" (of a given size) to minimize the number of references between objects which lie on different pages." and defines methods to find and improve a partition of a graph. The main idea of Kernighan and Lin was that, given a balanced partition of a graph into two blocks V_1 and V_2, there are subsets $A \subset V_1$ and $B \subset V_2$ such that the partition that is created when moving the nodes in A to V_2 and the nodes in B to V_1, is globally optimal. Kernighan and Lin then contributed a method to find "good" sets A and B to reduce the cost of a partition. We explain the method in the next section. One major drawback of the method is that it is expensive in terms of asymptotic running time. The worst case running time for one iteration of the Kernighan-Lin algorithm is $O(n^2 \log n)$. Dutt [59] has shown how to improve this to $O(m \max(\log n, \Delta))$ time. A major breakthrough has been achieved by Fiduccia and Mattheyses [66] in 1982. Fiduccia and Mattheyses modified the algorithm and present data structures such that the asymptotic running time of their local search algorithm was reduced to linear time $O(m)$. The modifications made and the data structures used are explained in Section 3.1.2. Karypis and Kumar [96] further improved the running time by stopping the algorithm of Fiduccia and Mattheyses, when it did not decrease the edge cut for x node moves. x is usually much smaller than the number of nodes.

All three algorithms allow a node to be moved at most once during one iteration of the algorithm. More expensive local search algorithms such as Tabu Search weaken this restriction, i.e. a node can be moved multiple times during one iteration. We explain Tabu Search at the end of Section 3.1.3. However, today most of the methods for improving a given partition are variations of the Fiduccia-Mattheyses algorithm.

3.1.1 Kernighan-Lin Algorithm

As mentioned earlier, the Kernighan-Lin algorithm [98] tries to improve a given partition by finding subsets $A \subset V_1$, $B \subset V_2$ and then moving the nodes in A and B to the respective opposite block. Indeed, there are optimal choices for A and B, but finding them is also NP-hard [98]. Hence, Kernighan and Lin developed a heuristic approach to find "good" sets. One *pass* of the Kernighan-Lin algorithm consists of finding these sets and exchanging the corresponding nodes. The Kernighan-Lin algorithm repeatedly finds such sets A, B to be exchanged until it reaches a local optimum, i.e. exchanging the sets does not decrease the number of edges cut.

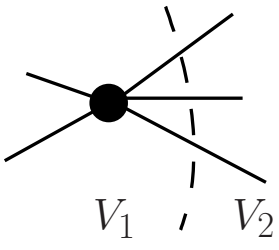


Figure 3.1: A node with gain one.

We now explain how Kernighan and Lin perform a single pass. Before we start, we introduce the definition of a node's gain. Let the input graph G be partitioned into two blocks V_1 and V_2. The *gain* of a node $v \in V_1$ is defined as $g(v) = \omega(\{(v,w) \mid w \in \Gamma(v) \cap V_2\}) - \omega(\{(v,w) \mid w \in$

$\Gamma(v) \cap V_1$), i.e. the reduction in the cut when v is moved from block V_1 to block V_2. The notion of gain is used analogously for nodes in block V_2. Thus, when $g(v) > 0$ we can decrease the cut by $g(v)$ by moving v to the opposite block. Figure 3.1 gives an example. Since the partition should stay balanced, the Kernighan-Lin algorithm searches for pairs of nodes to be exchanged. For $v \in V_1$ and $w \in V_2$ let $g(v,w)$ denote the gain of exchanging v and w between V_1 and V_2. Analogously, if v and w are not adjacent, then the gain is $g(v,w) = g(v) + g(w)$. If v and w are adjacent, then $g(v,w) = g(v) + g(w) - 2\omega(v,w)$ since the edge between v and w will still be a cut edge after the nodes are exchanged. Now a pass works as follows. First of all, a node can have two states: marked and unmarked. At the beginning of a pass each node is unmarked. Then, the following procedure is repeated p times where $p = \min(|V_1|, |V_2|)$. Find an unmarked pair $v \in V_1$ and $w \in V_2$ for which $g(v,w)$ is maximum. Note that $g(v,w)$ is not necessarily positive. Mark v and w and update the gain values of all the remaining unmarked nodes as if we had exchanged v and w. Only the gain values of the neighbors of v and w must be updated, since only these values could have changed. The step of finding a pair such that $g(v,w)$ is maximum can be implemented in $O(n \log n)$ time.

After this procedure is done, we have an ordered list L of node pairs (v_i, w_i), $i = 1, \ldots, p$. To output the sets A and B, we first find the smallest index $k \in \{0, \ldots, p\}$ such that $\sum_{i=1}^{k} g(v_i, w_i)$ is maximum. The sets are then defined as $A := \cup_{i=1}^{k} \{v_i\}$ and $B := \cup_{i=1}^{k} \{w_i\}$. If k is not zero, then the cut will be reduced if A and B are exchanged. In this case, the exchange is done and a new pass is started. Note that the algorithm has the ability to climb out of local minima to a certain extent due to the way in which the sets A and B are created. This is one of the key features of the algorithm.

3.1.2 Fiduccia and Mattheyses

Over time there have been many improvements made to the Kernighan-Lin algorithm. The most important improvement is a slight modification of the algorithm and the reduction in running time that was provided by Fiduccia and Mattheyses [66] in 1982. They reduce the complexity for a single pass to $O(m)$ by using novel data structures. Like the Kernighan-Lin method, the Fiduccia-Mattheyses method performs passes in which each node is moved at most once, and the best bisection observed during an iteration (if the corresponding reduction in the number of edges cut is positive) is used as input for the next iteration. However, instead of selecting pairs of nodes, the Fiduccia-Mattheyses method selects single nodes for movement.

In the perfectly balanced case, a pass of the Fiduccia-Mattheyses method works as follows. The algorithm starts by setting the state of every node to unmarked. In each step an unmarked node v with maximum gain value is *alternately* selected from the blocks V_1 and V_2. The node is then marked and the gain values of its unmarked neighbors are updated. This leads to two ordered sequences $(v_1, \ldots, v_p)$ and $(w_1, \ldots, w_p)$ with $v_i \in V_1$ and $w_i \in V_2$. The algorithm then searches for the smallest index $k \in \{0, \ldots, p\}$ such that $\sum_{i=1}^{k} g(v_i) + g(w_i)$ is maximized. If the resulting sum is positive, the algorithm performs

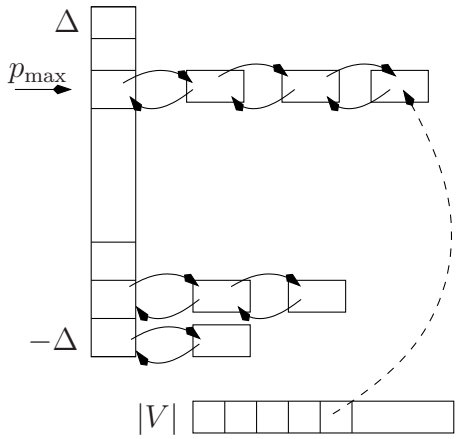


Figure 3.2: Bucket priority queue used in the Fiduccia-Mattheyses algorithm.

the movements and starts the next pass. Otherwise, the algorithm stops. If a reasonable amount of imbalance is allowed, then instead of alternately selecting the blocks, the balance criterion is used for the selection of the block. Note that both, the Kernighan-Lin and the Fiduccia-Mattheyses algorithm, do not stop when the corresponding sums are negative and thus are able to climb out of local minima to a certain extent.

In comparison to the Kernighan-Lin algorithm, there are two major modifications. First, nodes are selected independently for exchange so that computation of the gain values can be done efficiently. On the other hand, Fiduccia and Mattheyses provide a data structure such that the node with the best gain and the update of the gain values of the neighbors of a moved node can be done in constant time if the edge weights are non-negative integers. The data structure that is used to achieve this is a bucket priority queue. We now briefly outline this data structure since we also use it in our local search algorithms. Figure 3.2 gives an illustration. Let us assume that the graph has unit edge weights (integer weights are introduced straightforwardly). First, observe that the largest gain that a node can have is smaller than or equal to the maximum degree of a node (a similar argument holds for the smallest gain value that a node can have). Hence, one needs at most $2\Delta + 1$ buckets to order and to maintain all nodes sorted by their gain. Now, one needs two such specialized bucket queues, one for each block. Let the buckets be numbered/ordered in the following way: $[-\Delta, \ldots, \Delta]$. In the ith bucket a doubly linked list[1] stores all nodes that have gain i. Furthermore, for each node the position within the linked list of its bucket and a pointer $p_{\max}$ to the largest/maximum non-empty bucket is stored separately. Clearly, removing a node from and inserting a node into its bucket can be done in constant time. Hence, if the gain of a node changes, then the position of the node in the bucket queue is also updated in constant time. The pointer to the maximum element is updated in constant time if a node is inserted into the queue and the gain is larger than the current maximum. When trying to remove a node

[1] we use arrays in our implementation

v with maximum gain, one might have to update the pointer to the maximum non-empty bucket by decreasing the pointer until one has found the largest non-empty bucket since the pointer is not updated when a node is removed. This can also be done in amortized constant time. To see this, let v and w be two subsequent max gain nodes selected for movement by the Fiduccia-Mattheyses algorithm that are initially in the same block. The cost for updating the pointer $p_{\max}$ can be at most $O(d(v)+d(w))$ (this is the case when $g(v)=d(v)$ and $g(w)=-d(w)$). However, this is amortized, since the gain values of all neighbors of both maximum elements have to be updated. Hence, the total running time of one pass of the Fiduccia-Mattheyses algorithm is in $O(m)$. It is worth mentioning that Bob Darrow was the first who implemented the Fiduccia-Mattheyses algorithm (he is mentioned only in the acknowledgements of the paper of Fiduccia and Mattheyses [66]).

On small randomly generated graphs, the quality of the partitions produced by the Fiduccia-Mattheyses algorithm is slightly worse than the quality produced by the Kernighan-Lin algorithm [92]. The graphs used had about five hundred nodes and a random initial partition of the graph was used as starting point of the algorithms.

3.1.3 Further Improvements

Karypis and Kumar [96] successfully improved the algorithm further by introducing the following modifications. First of all, only boundary nodes are kept in the priority queues and the moves are done right away. After local search has stopped, they undo the node movements until they are at the best partition found during the iteration. On the other hand, their variant of the algorithm terminates when the edge cut does not decrease after x node moves. Terminating the Fiduccia-Mattheyses pass in this fashion significantly reduces the running time. However, note that *all* boundary nodes are used for initialization. In Chapter 4, we will introduce a variant of the algorithm that is highly localized, i.e. the priority queue is initialized with a single boundary node. This version of the algorithm is able to find partitions with improved quality compared to the Fiduccia-Mattheyses algorithm being initialized with all boundary nodes, and it can still be implemented in linear time.

The third modification that has been done by Karypis and Kumar is the following. First note that the Fiduccia-Mattheyses algorithm is dependent on the order in which the gain values are inserted into the priority queue, and a random tie-breaking mechanism is used if there are multiple nodes with equal gain. Hence, an additional pass of the algorithm can yield an improved cut even if a previous pass did not yield an improvement. Karypis and Kumar used multiple restarts of the algorithm to improve solution quality.

The Fiduccia-Mattheyses algorithm leaves the algorithm designer with the freedom to choose the block from which a node shall be moved to the opposite block. Indeed, one has to take the balance constraint into account. However, if the imbalance parameter is not too small, there are a number of possibilities. In Sanders et al. [87], we implemented three different block selection strategies which improved the quality of the output partition. The first strategy selects nodes with maximum gain from the blocks alternately,

the second strategy always selects the block with maximum size, and the third strategy always uses the block where the node has the larger gain value (with respect to the balance constraint). Note that the first strategy does not alter the balance of a partition, the second strategy may improves the balance and the third strategy may worsen the balance of the input partition but stays within the balance constraint.

Helpful Sets by Diekmann et al. [53, 121], introduce a more sophisticated neighborhood relation for the exchange of nodes in the bipartitioning case. These algorithms are inspired by a proof technique of Hromkovič and Monien [89] for proving upper bounds on the bisection width of a graph. Instead of migrating single nodes, whole sets of nodes are exchanged between the blocks to improve the cut. Selecting such node sets is based on the notion of helpfulness. The helpfulness of a node set is the reduction in the cut if the node set is moved to the opposite block. More precisely, a node set is called ℓ-helpful if it reduces the cut by ℓ. One round of the algorithm works as follows. First, the algorithms tries to find an ℓ-helpful set in *one* block of the partition. Then the algorithm tries to find a balancing set in the opposite block. A balancing set has the same cardinality as the found ℓ-helpful set and is at least $-\ell + 1$ helpful. If such sets can be found, the movements are performed, and the algorithm proceeds with the next round. Otherwise, the algorithm stops. The running time of the algorithm is comparable to the Kernighan-Lin algorithm while solution quality is often better than other methods [121].

Extension to k-way Local Search

A common method to create a k-partition is recursive bisection [99]. The graph is recursively divided into two blocks (this includes two-way local search) until the number of blocks is reached, i.e. a bisection algorithm is used to split the graph into two blocks. The same is done on the graphs induced by the two created blocks until the desired number of blocks is reached. More algorithms to obtain a k-partition are described in Section 3.2. It has been shown by Simon and Teng [156] that, due to the lack of global knowledge, recursive bisection can create partitions that are very far away from the optimal partition so that there is a need for k-way local search algorithms.

There are multiple ways of extending the Fiduccia-Mattheyses algorithm to get a local search algorithm that can improve a k-partition. Given a k-partition of a graph, the first obvious idea is to use a two-way local search algorithm between all pairs of blocks that share a non-empty boundary, i.e. blocks that are connected by at least one edge. This is a general concept that enables us to extend flow-based methods that are presented in Chapter 4 to improve a k-partition.

One early extension of the Fiduccia-Mattheyses algorithm to k-way local search was described by Sanchis [140] as well as Hendrickson and Leland [85]. The algorithm makes use of $k(k-1)$ priority queues, one for each type of move (source block, target block). A single node movement is done as follows. First, all queues maximizing the gain are found. Then the movement with the highest gain that preserves or improves the balance is performed. Roughly speaking, a node is moved to a block A which maximizes

the reduction in the cut when the node is moved. However, the running time of this algorithm is significantly higher than the running time of the two-way Fiduccia-Mattheyses algorithm.

Karypis and Kumar [96] present a k-way version of the Fiduccia-Mattheyses algorithm that runs in linear time $O(m)$. Instead of $k(k-1)$ priority queues, Karypis and Kumar use one global priority queue for all types of moves. The priority used is the maximum local gain, i.e. the maximum reduction in the cut when the node is moved to one of its neighboring blocks. The node that is selected for movement yields the maximum improvement for the objective and maintains or improves upon the balance constraint. To improve the running time, the priority queue only contains boundary nodes, i.e. nodes that have an external degree greater than zero, and local search is stopped after x movements that did not decrease the overall cut.

A more expensive k-way local search algorithm is based on tabu search [77, 78], which has been applied to graph partitioning by [137, 21, 19, 20, 72]. We briefly outline the method reported by Galinier et al. [72]. Instead of moving a node only once per pass, as in the traditional versions of the Kernighan-Lin/Fiduccia-Mattheyses algorithms, specific types of moves are excluded only for a number of iterations. The number of iterations that a move (v, block) is excluded depends on an aperiodic function f and the current iteration i. The algorithm always moves a non-excluded node with the highest gain. If the node is in block A, then the move (v,A) is excluded for $f(i)$ iterations after the node is moved to the block yielding the highest gain, i.e. the node cannot be put back to block A for $f(i)$ iterations.

3.2 Obtaining Partitions

The quality of the presented local search algorithms highly depends on the quality of the input partition. Indeed there are multiple ways to obtain initial partitions of a graph. For example, Kernighan and Lin [99, 98] used random bipartitions of a graph as starting point for their Kernighan-Lin local search.

Keeping in mind that recursive bisection [99, 98] can be used to obtain a k-partition of a graph, we start to elaborate on two methods for bisecting a graph: namely, spectral partitioning and greedy graph growing. Then we look at frameworks that can be used to directly derive a k-partition. It is worth mentioning that the techniques presented in this section are nowadays most often combined with the multilevel approach, i.e. they are only used as algorithms to obtain a partition of the coarsest graph in the multilevel hierarchy.

3.2.1 Spectral Partitioning

The first method to split a graph into two blocks, spectral bisection, is still used today by many researchers. Spectral techniques were first used by Donath et al.

[56, 57] and Fiedler [67], and have been improved subsequently by several researchers [29, 132, 155, 86, 18]. We try to outline the basic idea of spectral bisection[2]. The spectral bisection method infers global information of the connectivity of a graph by computing the eigenvector corresponding to the second lowest eigenvalue of the Laplacian matrix L of the graph. This eigenvector is also known as Fiedler vector, and the associated eigenvalue is called algebraic connectivity [67] due to the relation to the connectivity in the graph. The Laplacian matrix is defined as $L = D - A$ where D is the diagonal matrix expressing node degrees and A is the adjacency matrix. It is well-known that the smallest eigenvalue of L is 0 and that $\mathbb{1} = (1, \ldots, 1)$ is the corresponding eigenvector. The differences of the entries in the Fiedler vector provide information about the distance between the associated nodes. Hence, the spectral bisection method sorts nodes of the graph with respect to entries in the second eigenvector, and then divides the sorted set into two halves. An interesting observation of Fiedler [67] is that at least one of the blocks created by this method is connected if the input graph is connected.

The *main idea* of the spectral bisection algorithms used today is the following [67]. First of all, notice that $x^T Lx = \sum_{(u,v)\in E}(x_u - x_v)^2$ holds. Let us assume that we have a partition of the graph into two blocks V_1 and V_2. Furthermore, if $v \in V_1$, then let x_v be -1, otherwise let x_v be 1. If C is the set of cut edges, then clearly $x^T Lx = \sum_{(u,v)\in C}(x_u - x_v)^2$. Hence, $x^T Lx = 4|C|$ and it is useful to define the following optimization problem.

$$\min\left\{x^T Lx \;\middle|\; x^T\mathbb{1} = 0, x^T x = n, x \in \{-1,1\}^n\right\}$$

The first constraint ensures that the nodes are balanced between the blocks. The objective is to minimize the cut. When the integrality constraint is dropped, one can use Lagrange Multipliers to solve the relaxed problem and then infer a perhaps suboptimal solution for the original problem. Lagrange's optimality conditions give us $Lx - \lambda_1\mathbb{1} - \lambda_2 x = 0$, $x^T x = n$ and $x^T\mathbb{1} = 0$. By the definition of the Laplacian matrix, we have $\mathbb{1}^T L = 0$ and hence $\lambda_1 = 0$. Thus $Lx = \lambda_2 x$ and x must be an eigenvector. Furthermore, $\frac{x^T Lx}{n} = \lambda_2$ and $x \neq \mathbb{1}$ so that x must be the eigenvector corresponding to the second smallest eigenvalue. Hence, we have a connection of the cut of a partition and an eigenvalue problem involving the graph Laplacian of a graph.

The second eigenvector can be computed using a modified Lanczos algorithm [105]. However, this method is expensive in terms of running time. Barnard and Simon [18] use a multilevel method to obtain a fast approximation of the Fiedler vector. The structure is similar to the multilevel method which is explained in full detail in Section 3.3. However, the graph is coarsened using maximal independent sets instead of edge contractions. The nodes of the maximal independent set form the nodes of the next coarser graph. We do this analogously in the AMG-inspired coarsening scheme that we introduce in Chapter 4. On the coarsest level the Fiedler vector is computed using the Lanczos algorithm. The vector is then projected to the next finer level by setting the entries to their associated coarse nodes, if they exist, and by averaging over their neighbors in the current fine

[2]Parts of the formulations have been taken from [88]

graph, otherwise. On each level the current vector is refined using a Rayleigh quotient iteration [125, 126], since it can take advantage of a good initial approximation.

Hendrickson and Leland [86] extend the spectral method to partition a graph into more than two blocks by using multiple eigenvectors which are computationally inexpensive to obtain. The method produces better partitions than recursive bisection, but is only useful for the partitioning of a graph into four or eight blocks. The authors also extended the method to graphs with node and edge weights.

3.2.2 Graph Growing

The second method for obtaining a bisection of a graph is called graph growing [95]. Its simplest version works as follows. Starting from a random node v, the blocks are assigned using a breadth-first search starting at v. All nodes touched during the breadth-first search are assigned to block V_1. The search is stopped after half of the original node weights are assigned to this block and V_2 is set to $V \setminus V_1$. The computed partitions have a rather large cut because breadth-first search does not care about cut edges at all. Hence, the authors use a local search algorithm to improve the partition. Moreover, the method depends heavily on the chosen start node v, so that the method must be repeated several times to get a good solution.

There are two variations of this algorithm. An algorithm called greedy graph growing [95] takes the resulting cut into account. Instead of performing a simple breadth-first search, the algorithm always adds the node to the block that results in the smallest increase in the cut. This can be implemented similar to the Fiduccia-Mattheyses algorithm, i.e. using the same data structures.

A variant of the algorithm by George et al. [75] first searches for two nodes that are "far" away from each other. Such nodes are called pseudo peripheral nodes. To find such nodes one first chooses a random node v. Starting at v one performs a breadth-first search that explores the whole graph. The last node w touched by the breadth-first search then serves as new start node for the next round. This is repeated a few times. When the process is stopped, one hopes to have two nodes that have a large distance in the graph. These nodes serve as seed nodes for the assignment of blocks. To obtain a partition

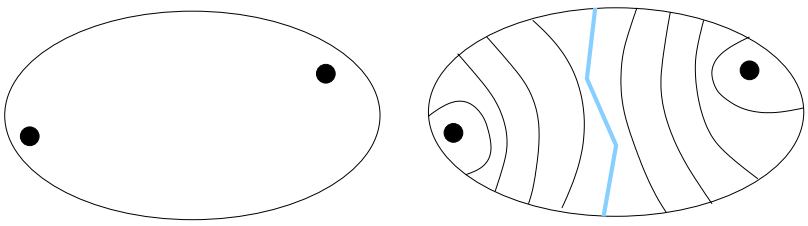


Figure 3.3: Visualisation of graph bisection. First two pseudoperipheral nodes are found and then two breadth-first search are performed alternately to assign nodes to blocks.

of the graph, two breadth-first searches, one for each node, are started and performed alternately. The nodes touched by the first breadth-first search are assigned to block one and the nodes touched by the second breadth-first search are assigned to block two. An example is shown in Figure 3.3.

3.2.3 Bubble Framework

The spectral and the graph growing method are able to bisect a graph into two parts. With the idea in mind to optimize the shape of a partition, a center-based method to directly compute a k-partition has been proposed by Diekmann et al. [54]. A similar idea has been used before by Walshaw et al. [166] for the selection of nodes for migration in a dynamic graph partitioning setting, i.e. the graph is modified over time using node and edge insertions, and the partition has to be updated.

The bubble framework by Diekmann et al. [54] can be seen as an extension of graph growing and is related to Lloyd's k-means algorithm [110]. Instead of growing blocks using one or two breadth-first searches around the seed nodes, one uses k seed/center nodes and k simultaneous breadth-first searches to find the blocks. To find k initial seed nodes that are fairly distributed over the graph, Diekmann et al. [54] do the following. The initial seed is found by performing a breadth-first search from a node v with minimum degree which in case of finite element meshes is usually a corner of the mesh. The first seed node v_1 is then the node that is farthest away from v, e.g. the last node touched by the breadth-first search. The second seed node is the node that is farthest away from v_1, the third seed node the node that is farthest away from v_1 and v_2 and so on. Hence, an additional seed node is always found by performing a breadth-first search that is initialized with all previously found seed nodes. Overall, k breadth-first searches are performed to find the seed nodes.

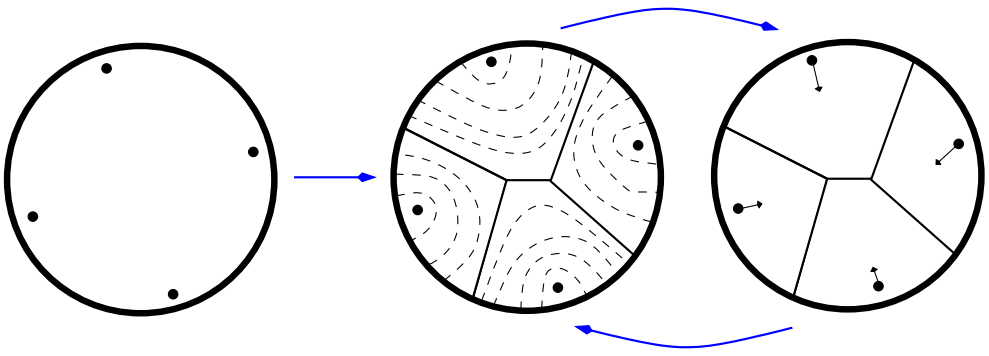


Figure 3.4: The three steps of the bubble framework. Black nodes indicate the seed nodes. On the left hand side, seed nodes are found. In the middle, a partition is found by performing breadth-first searches around the seed nodes and on the right hand side new seed nodes are found.

Once the seed nodes are found, the blocks are grown around the nodes – again using breadth-first searches. The breadth-first searches are scheduled such that always the smallest block receives the next node. Hence, the algorithm ensures that the resulting blocks of the partition are connected but not necessarily balanced. The authors use local search algorithms to balance the load of the blocks and to improve the cut of the resulting partition, but this can result in unconnected blocks.

The third step of the algorithm is to find new seed nodes for the next round. The new center of a block is defined as the node that minimizes the sum of the distances to all other nodes within its block. To avoid the expensive computation, the authors compute this value only for the initial seed of a block and all its neighboring nodes. Among those, the node with the smallest value is used as a new seed node. The second and the third step of the algorithm are iterated until either the seed nodes stop changing or no improved partition was found for more than 10 iterations. One drawback of the algorithm is its computational complexity $O(km)$. Figure 3.4 illustrates the three steps of the algorithm.

Subsequently, this approach has been used and improved by [148, 117, 116, 115, 114]. For example, Schamberger [148] introduced the usage of diffusion as a growing mechanism around the initial seeds and extended the method to weighted graphs. This enables the authors to use the method within the multilevel framework. The main idea of diffusion in the graph partitioning area is that the load that is distributed using the diffusion process spreads faster in areas of the graphs that are densely connected than in areas that are only sparsely connected. A modification of the diffusion scheme enhances the balance of the partition computed in the second step of the framework.

Because these algorithms still have a large execution time, approaches using algebraic multigrid techniques were used to improve the running time of the algorithm by Meyerhenke et al. [116]. To do so the diffusion process was modified to a disturbed diffusion process. The disturbed diffusion process has the advantage that it does not result in a fully balanced load situation upon convergence and that it can be solved by solving a system of linear equations. Hence, algebraic multigrid (AMG) is used to solve the sparse linear system, and the hierarchy created is reused by the partitioning algorithm.

This approach has been further improved by Meyerhenke et al. [115] by combining it with a faster diffusion process that is restricted to local areas of the graph. The corresponding graph partitioner is called DibaP. The AMG scheme is used on the coarse levels of the multilevel scheme, and the faster scheme is used on the finer levels of the hierarchy. The approach computes partitions of very high quality and has been able to obtain many entries in the Walshaw Benchmark [157]. While the approach is much faster than the initial version of the algorithm, it still has worst-case complexity $O(km)$. It remains an open problem to have a fast and high quality direct k-partitioning scheme.

Pellegrini [128] uses diffusion only between pairs of blocks during recursive bisection and only in an area around the initial cut of a partition to speed up the computation.

3.3 Multilevel Approach

The basic idea of multilevel graph partitioning can be traced back to multigrid solvers for solving systems of linear equations [158], but more recent practical methods are based on mostly graph theoretic aspects, in particular edge contraction and local search. As already mentioned, the method was initially introduced to the graph partitioning area by Barnard and Simon [18] to speed up spectral partitioning techniques. Hendrickson and Leland [85] formulated the multilevel approach as it is known today. Already a decade earlier, Bui et al. [33] remarked that a two level approach, i.e. randomly contracting edges, improves the result of a partitioning algorithm if it is applied on the coarse graph.

Before we outline the multilevel approach, we need to define the notion of edge contractions. *Contracting* an edge $\{u,v\}$ means to replace the nodes u and v by a new node x connected to the former neighbors of u and v. We set $c(x) = c(u) + c(v)$ so that the weight of a node at each level is the number of nodes it is representing in the original graph. If replacing edges of the form $\{u,w\}$, $\{v,w\}$ would generate two parallel edges $\{x,w\}$, a single edge with $\omega(\{x,w\}) = \omega(\{u,w\}) + \omega(\{v,w\})$ is inserted. *Uncontracting* an edge e undoes its contraction. In order to avoid tedious notation, G will denote the current state of the graph before and after a (un)contraction unless we explicitly want to refer to different states of the graph.

The *multilevel approach* to graph partitioning consists of three main phases. It is outlined in Figure 3.5. In the *contraction* (coarsening) phase, a hierarchy of graphs is created. There are multiple ways to do that. The most common way is to iteratively identify matchings $M \subseteq E$ and contract the edges in M. Contraction should quickly reduce the size of the input and each computed level should reflect the global structure of the input network. An example matching that is contracted is shown in Figure 3.6. Contraction is stopped when the graph is sufficiently small to be directly partitioned using some expensive other algorithm which were described in the previous sections such as spectral partitioning, graph growing or bubbling. In the *local improvement* (or uncoarsening) phase, the matchings are iteratively uncontracted. Note that due to the way that the contraction is defined, a partitioning of the coarse level creates a partitioning of the finer

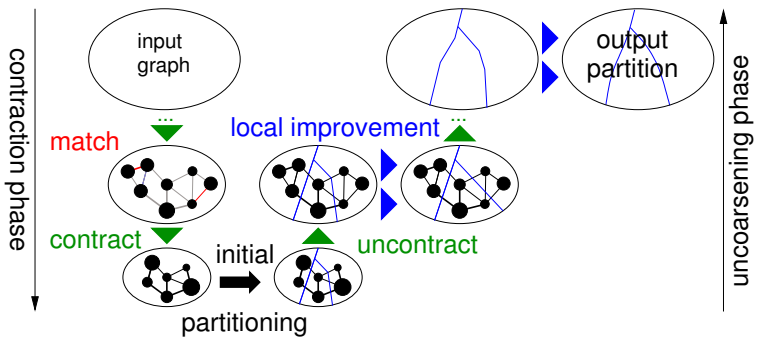


Figure 3.5: The multilevel approach to graph partitioning. Source: [143][2].

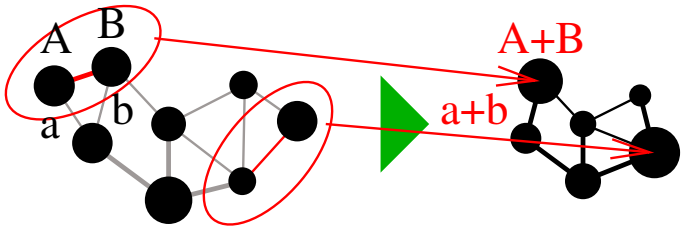


Figure 3.6: A example matching highlighted in red and contraction of matched edges. Source: [123].

graph having the same objective and balance. After uncontracting a matching, a local improvement algorithm moves nodes between blocks in order to improve the cut size or balance. Usually variants of the Fiduccia-Mattheyses algorithm are used. The intuition behind this approach is that a good partition at one level will also be a good partition on the next finer level, so that local search will quickly find a good solution. Moving a node on a coarse level hierarchy usually corresponds to the movement of a whole set of node movements of the finest level of the hierarchy. Intuitively, the multilevel scheme has a global view on the optimization problem on the coarse levels of the hierarchy and a very local view on the finest levels with respect to the original graph.

It is worth mentioning that there are recursive partitioning approaches that use the multilevel approach for the bisection of a graph by Karypis et al. [95], and that they were the first who had a linear time $O(m)$ implementation of this scheme to obtain a k-partition [96] (using recursive multilevel bisection only on the coarsest level and a direct k-way local search algorithm).

An interesting variant of the multilevel algorithm has been proposed by Sanders and Osipov [122]. Their n-level approach is based on the extreme idea of contracting only one single edge between two consecutive levels of the multilevel hierarchy. During uncoarsening, local search is done highly localized around the uncontracted edge. Using sophisticated data structures their algorithm requires sublinear time on real-world graphs.

3.4 Evolutionary Algorithms

For a general overview of genetic/evolutionary algorithms tackling the graph partitioning problem, we refer the reader to the overview paper by Kim et al. [101]. In this section we focus on the description of hybrid evolutionary approaches that combine evolutionary ideas with the multilevel graph partitioning framework [157, 19, 20]. Other approaches such as Probe by Chardaire et al. [37], which can be viewed as a genetic algorithm without selection, and Fusion Fission by Bichot et al. [24], which is inspired by nuclear processes, are not covered here. Hybrid algorithms are usually able to compute partitions

[2]Sources of images in this work correspond to our papers or the associated conference talks.

with considerably better quality than those that can be found by using a single execution of a multilevel algorithm.

Evolutionary approaches that are combined with local search heuristics are called memetic algorithms. The first approach that combined evolutionary ideas with a multilevel partitioner was by Soper et al. [157]. The authors define two main operations, a combine and a mutation operation. Both operations modify the edge weights of the graph depending on the input partitions and then use the multilevel partitioner Jostle, which uses the modified edge weights to obtain a new partition of the original graph. The combine operation works as follows (the mutation operation is done in a similar way). The algorithm first computes node biases and then uses those to compute perturbations of the edge weights. However, node biases are not an input to the multilevel graph partitioner. Given two partitions of the graph, a node is assigned a random value in $[0,0.01]$ if the node is a boundary node in both input partitions and a *larger* bias of 0.1 plus a random value in the same range, otherwise. For an edge, the perturbed weight is then defined as one plus the biases of its incident nodes. Note that the perturbed edge weights are chosen such that the local search is guided to mimic the input partitions. The algorithm uses a fixed population size of fifty and, to obtain a new generation, creates fifty new individuals using a ratio of 7:3 of combine and mutation operations. A new generation is then defined as the best fifty partitions out of the current generation and the fifty newly created ones. While producing partitions of very high quality, the authors report running times of up to one week. In their paper the authors introduce the well-known Walshaw benchmark, which is presented in the next section. A similar approach based on edge weight perturbations is used by Delling et al. [49].

A multilevel memetic algorithm for the perfectly balanced graph partition problem, i.e. $\varepsilon = 0$, was proposed by Benlic et al. [19, 20]. The main idea of their algorithm is that among high quality solutions a large number of nodes will always be grouped together. In their work the partitions represent the individuals. We briefly sketch the combination operator for the case that two partitions are combined. First the algorithm selects two individuals/partitions from the population using a λ-tournament selection rule, i.e. choose λ random individuals from the population and select the best among those if it has not been selected previously. Let the selected partitions be $P_1 = (V_1, \ldots, V_k)$ and $P_2 = (W_1, \ldots, W_k)$. Then sets of nodes that are grouped together within the partitions are computed:

$$B := \left\{ \bigcup_{j=1}^{k} \{V_j \cap W_{\sigma(j)}\} \right\}$$

such that the number of nodes that are grouped together, $\sum_{j=1}^{k} |V_j \cap W_{\sigma(j)}|$, is maximum among all permutations σ of $\{1, \ldots, k\}$. An offspring is created as follows. Nodes in B will be grouped within a block of the offspring. That means if a node is in the set B, then it is assigned to the same block to which it was assigned to in P_1. Otherwise, it is assigned to a random block, such that the balance constraint remains fulfilled. Local search is then used to improve the computed offspring before it is inserted into the population. In

Benlic et al. [20] the authors combine their approach with tabu search. Their algorithms produce partitions of very high quality, but cannot guarantee that the output partition fulfills the desired balance constraint.

3.5 Flow-Based Approaches

Ford and Fulkerson [71] presented their well-known max-flow min-cut theorem in 1956. While it can be used to separate two nodes in a graph by computing a maximum flow and hence a minimum cut between them, it completely ignores balance, and it is unclear how it could directly be applied to the balanced graph partitioning problem. However, the algorithm is often used as a subroutine to solve related max-flow problems, e.g. to bisect regular graphs by Bui et al. [33], to improve a given partition when quality is measured by expansion or conductance by Lang and Rao [107] and Andersen and Lang [9], or as a pre-processing technique for road network partitioning by Delling et al. [49]. Note that the problem that asks to find a non-trivial cut (V_1, V_2) with minimum conductance or expansion, does not necessarily yield a balanced cut as in the balanced graph partitioning problem. In Chapter 4 we define a flow-based algorithm that improves a given balanced partition when quality is measured using the edge-cut metric.

Bui et al. [33] define an algorithm for bisecting r-regular graphs. To do so, the authors define a larger neighborhood $N_d(v)$ of a node v, which consists of all nodes within distance d of v. For two nodes u, v a flow problem is constructed by replacing $N_d(v)$ by an infinite capacity source and $N_d(u)$ by an infinite capacity sink where the parameter d depends on the input graph. Edges connecting nodes in $N_d(u)$ with nodes in $N_d(v)$ are replaced by edges that directly connect the source and the sink. This flow problem is solved for all pairs of nodes u, v in the graph and, for r-regular graphs. If the smallest cut happens to be a bisection, the authors are able to show that the algorithm has found an optimal partition. The algorithm solves n^2 flow problems on graphs that have rn edges (for r-regular graphs) and gives guarantees only for r-regular graphs. Hence the algorithm will not scale to large graphs and it is unclear whether one could create balanced bisections of real-world graphs. The algorithm can be seen as a rather theoretical approach.

Max-flow Quotient-cut Improvement (MQI) by Lang and Rao [107] and Improve by Andersen and Lang [9] are flow-based methods for improving graph bipartitions when cut quality is measured by quotient-style metrics such as expansion or conductance. In this case, the balance constraint is dropped, since the measures directly optimize cut vs. balance of the partition. Given a partition V_1, V_2 of the graph, MQI constructs a flow problem such that the output partition is the best improvement among all partitions, where V_1' is a strict subset of V_1 w.r.t. the quotient-style metric. The flow problem is constructed as follows. Let V_1 be the smaller block, c be the number of cut edges and a the number of nodes in V_1. First, the construction completely discards all nodes in V_2. Each undirected edge in V_1 is replaced by two directed edges with capacity a, one in each

direction. Then a source and a sink are inserted. Each node in V_1 is connected to the sink by a directed edge with capacity c and the source is connected to all boundary nodes in V_1 via a directed edge with capacity a. Solving this flow problem, the authors can construct a new partition having best quotient cut score among all partitions (V_1', V_2'), where V_1' is a strict subset of V_1 if one exists. Improve [9] solves a slightly different flow problem that takes both blocks into account and is able to always outperform or tie MQI. Since both approaches need a pre-partition of the graph, the authors combine their algorithms with Metis [95] and have been able to improve some of the bipartitions in the Walshaw Benchmark.

Delling et al. [49] use a flow-based method as a pre-processing technique for the partitioning of road networks. Here, a clustering of the graph is computed by multiple max-flow min-cut computations to identify natural cuts and dense regions of the graph. The resulting clustering is then contracted to obtain a smaller representation of the graph, which still contains small cuts. Often this graph is up to two orders of magnitude smaller than the input graph which drastically reduces the running time of a partitioning algorithm. Moreover, the cuts found are much better than when applying a partitioning algorithm to the original graph. Since we adopt this technique for the partitioning of road networks, we describe it in more detail in Chapter 5.

3.6 Hardness Results

That the problem of partitioning a graph into k blocks of roughly equal size, such that the cut metric is minimized, is NP-complete has been shown by Hyafil and Rivest [90] and Garey et al. [73]. Andreev and Räcke [10] have shown that there is no constant factor approximation for the perfectly balanced version ($\varepsilon = 0$) of this problem on general graphs. If $\varepsilon \in (0,1]$, then an $O\left(\log^2 n\right)$ factor approximation can be achieved. If an even larger imbalance $\varepsilon > 1$ is allowed, then an approximation ratio of $O(\log n)$ is possible [60]. The minimum weight k-cut problem, asks for a partition of the nodes into k non-empty blocks without enforcing a balance constraint. Goldschmidt et al. [80] proved that, for a fixed k, this problem can be solved optimally in $O\left(n^{k^2}\right)$. Moreover, the problem is NP-complete [80] if k is not part of the input.

For the unweighted minimum bisection problem, Feige et al. [62] have shown that there is an $O\left(\log^{1.5} n\right)$ approximation algorithm and an $O(\log n)$ approximation minimum bisection on planar graphs. The bisection problem is efficiently solvable if the balance constraint is dropped – in this case it is the min cut problem. Wagner et al. [160] have shown that the minimum bisection problem becomes harder the more the balance constraint is tightened towards the perfectly balanced case. More precisely, if the block weights are bounded from below by a constant, i.e. $|V_i| \geq C$, then the problem is solvable in polynomial time. The problem is NP-hard if the block weights are constrained by $|V_i| \geq \alpha n^{\delta}$ for some $\alpha, \delta > 0$ or if $|V_i| = \frac{n}{2}$. The case $|V_i| \geq \alpha \log n$ for some $\alpha > 0$

is open. Note that the case $|V_i| \geq \alpha n^{\delta}$ also implies that the general graph partitioning problem with similar lower bounds on the block weights is NP-hard.

If the balance constraint of the problem is dropped and one uses a different objective function such as sparsest cut, then there are better approximation algorithms. The sparsest cut objective combines cut and balance into a single objective function. For general graphs and the sparsest cut metric, Arora et al. [14, 13] achieve an approximation ratio of $O\left(\sqrt{\log n}\right)$ in $\tilde{O}(n^2)$ time. Using multiple max-flow min-cut computations, Leighton et al. achieve an approximation ratio of $O(\log n)$.

Being of high theoretical importance, most of the approximation algorithms are not implemented, and the approaches that implement approximation algorithms are too slow to be used for large graphs or are not able to compete with state-of-the-art graph partitioners. Hence, mostly heuristics are used in practice.

3.7 Exact Methods

There is a large amount of literature on methods that solve the graph partitioning problem optimally. This includes methods dedicated to the bipartitioning case [32, 93, 64, 153, 11, 12, 50, 52, 63] and some methods that solve the general graph partitioning problem [65, 154]. Most of the methods rely on the branch-and-bound framework [106].

Bounds are derived using various approaches: Karisch et al. [93] and Armbruster et al. [11] use semi-definite programming, and Sellman et al. [153] and Sensen [154] employ multi-commodity flows. Linear programming is used by Brunetta et al. [32], Ferreira et al. [65] and by Armbruster et al. [12], and quadratic programming is adopted by Hager et al. [81]. Felner et al. [64] and Delling et al. [50, 52] utilize combinatorial bounds. Depending on the method used, the bounds derived can be very good and yield small branch-and-bound trees, but are hard to compute or the bounds are somewhat weaker and yield larger trees but are faster to compute. The latter is the case when using combinatorial bounds. On finite connected subgraphs of the two dimensional grid without holes, the bipartitioning problem can be solved optimally in $O\left(n^4\right)$ time [63].

All of these methods typically can only solve very small problems while having very large running times, or if they can solve large bipartitioning instances using a moderate amount of time [50, 52], highly depend on the bisection width of the graph. Methods that solve the general graph partitioning problem [65, 154] have immense running times for graphs with up to a few hundred nodes. Moreover, the experimental evaluation of these methods only considers small block numbers $k \leq 4$.

3.8 Walshaw Benchmark

The Walshaw benchmark[3] was created in 2000 by Soper, Walshaw and Cross [157] as part of a research project into very high quality partitions. This public domain archive is maintained by Chris Walshaw and contains 34 real-world graphs stemming from applications such as finite element computations, matrix computations, VLSI Design and shortest path computations as well as the best partitions of these graphs found so far. Everyone can submit partitions applying the rules used there, i.e. running time is not an issue, but one wants to achieve minimal cut values for $k \in \{2,4,8,16,32,64\}$ and balance parameters $\varepsilon \in \{0,0.01,0.03,0.05\}$. Currently, the solutions of over forty different algorithms are submitted to the archive. It is the most popular graph partitioning benchmark in the literature.

3.9 Software Packages

There are a number of software packages that implement the described algorithms. One of the first publicly available software packages called Chaco is due to Hendrickson [83]. Like most of the publicly available software packages, Chaco implements the multilevel approach outlined in Chapter 2 and basic local search algorithms. Moreover, they implement spectral partitioning techniques. Probably the fastest and best known system is Metis [95, 96] by Karypis and Kumar. ParMetis is a widely used parallel implementation of their algorithm [94]. Scotch [127, 41, 42] is a graph partitioning framework by Pellegrini. It uses recursive multilevel bisection and includes sequential as well as parallel partitioning techniques. Jostle [165, 163] is a well-known sequential and parallel graph partitioner developed by Chris Walshaw. The commercialised version of this partitioner is known as NetWorks. It has been able to hold most of the records in the Walshaw Benchmark for a long period of time. If a model of the communication network is available, then Jostle and Scotch are able to take this model into account for the partitioning process. Party [54, 121] implements the bubble/shape-optimized framework and the helpful sets algorithm described within this chapter. The software package DibaP by Meyerhenke [115] builds upon Party and implements the bubble framework using diffusion and AMG-based techniques. Zoltan [27] is a project for parallel partitioning, load balancing and data-management services. It provides interfaces to various graph partitioning packages such as ParMetis and Scotch.

[3] `http://staffweb.cms.gre.ac.uk/~wc06/partition/`

4

Multilevel Graph Partitioning

When we started to build our own parallel multilevel graph partitioner, KaPPa [87], with focus on scalable parallelization we also obtained improved partitioning quality through rather simple methods. This motivated us to make a fresh start by putting all aspects of multilevel graph partitioning in a *sequential* setting on trial. Hence, in this chapter our focus is on the different components of the multilevel graph partitioning scheme. We start by describing different coarsening schemes, such as matching and AMG-inspired coarsening techniques in Section 4.1. After shortly outlining the initial partitioning algorithm that we use in Section 4.2, we explain novel local improvement techniques in Section 4.3. Here, we look at two novel local search algorithms. The first algorithm is based on iterated max-flow min-cut computations and the second algorithm is a highly localized version of the Fiduccia-Mattheyses algorithm. We then look at global search strategies in Section 4.4 and conclude this chapter with an extensive experimental evaluation in Section 4.5.

References. This chapter is based on the publications [143, 146] which were published together with Peter Sanders. The parts on AMG-inspired coarsening and algebraic distance stem from the conference paper [139] which was published together with Ilya Safro and Peter Sanders. Our own initial partitioning has not been published before.

4.1 Coarsening

We start with the presentation of two different coarsening schemes. First we explain matching based coarsening schemes, the matching algorithm that we use and the concept of edge ratings that was introduced in KaPPa [87]. Today, the matching based scheme is used in most graph partitioning algorithms due to its simplicity and speed. We also look at a new edge rating that uses algebraic distance as a measure of connection strength between nodes. This is followed by an AMG-inspired coarsening scheme that works in particular very well on highly unstructured networks, e.g. social networks.

4.1.1 Matching Based Coarsening

The general matching based coarsening scheme has already been explained in Chapter 3. Using this scheme a coarse graph is constructed by contracting edges of a matching that has been computed by a matching algorithm. Although a maximum matching can be computed in polynomial time, optimal algorithms are too slow to be used in a multilevel framework. In practice heuristic matching algorithms are fast, produce good matchings and also can give an approximation guarantee.

Before we explain the matching algorithms used in our system, we present the two-phase approach which we already used in KaPPa [87]. The two-phase approach makes contraction more systematic by separating two issues: A *rating function* and a *matching* algorithm. A rating function indicates how much sense it makes to contract an edge based on *local* information. A matching algorithm tries to maximize the sum of the ratings of the contracted edges looking at the *global* structure of the graph. While the rating function allows a flexible characterization of what a "good" contracted graph is, the simple, standard definition of the matching problem allows to reuse previously developed algorithms for weighted matching.

Edge Ratings

To explain the concept of edge ratings, we have to characterize the properties of "good" matchings for the purpose of contraction in a multilevel algorithm for graph partitioning. Intuitively, a matching should contain *large edge weights* since we want to solve the problem on the coarsest level and our main objective is to find a small cut. On the other hand a matching should contain a *large number of edges*, e.g. being maximal, so that there are only few levels in the hierarchy and the algorithm can converge quickly. In order to *represent the input* on the coarser levels, we want to find matchings such that the graph after contraction has somewhat *uniform node weights* and *small node degrees*. Uniform node weights are also helpful to achieve a balanced partition on the coarser levels and makes local search algorithms more effective. Using the *edge weight* as a rating function has been done before and takes care of the first two requirements. However, the edge weight completely ignores the two latter properties of a good matching.

The perspective taken in KaPPa [87] is to encode these properties into a single scalar edge rating function. Then an approximate weight matching algorithm is applied that tries to find a matching which maximizes the sum of the ratings. The matching algorithm used is the Global Paths Algorithm (GPA) by Maue et al. [112] which is also used in KaFFPa and explained below. The default configurations of KaFFPa employ combinations of the ratings

$$\begin{aligned} \text{expansion}^{*2}(\{u,v\}) &:= \omega(\{u,v\})^2/c(u)c(v), \text{ and} \\ \text{innerOuter}(\{u,v\}) &:= \omega(\{u,v\})/(\text{Out}(v)+\text{Out}(u)-2\omega(u,v)), \end{aligned}$$

where $\text{Out}(v)$ is set to $\text{Out}(v) := \sum_{x\in\Gamma(v)} \omega(\{v,x\})$, since they yielded the best results in

KaPPa [87]. For the purpose of partitioning unstructured networks, we also look at a rating based on algebraic distance as a measure of connectivity between the nodes.

The notion of *algebraic distance* has been introduced by Safro et al. [138, 38] and is based on the principle of obtaining low-residual error components [30]. This principle was used for linear ordering problems to distinguish between *local* and *global* edges [138]. A local edge e is characterized by having a small distance between its end nodes after e is removed. In contrast to a local edge, a non-local or global edge is defined via a large distance of its end nodes in the graph after e is removed. The main difference between the graph partitioning problem and linear ordering problems is the balance constraint. Thus, we introduce a node weight *normalized algebraic distance* to ensure that the node weights do not get too non-uniform during coarsening.

Given the Laplacian of a graph $L = D - W$, where W is a weighted adjacency matrix of a graph and D is the diagonal matrix with entries $D_{vv} = d(v)$, we define its node weight normalized version by $\tilde{L} = \tilde{D} - \tilde{W}$ based on the normalized edge weights $\tilde{\omega}(e = \{v,w\}) := \omega(e)/\sqrt{c(v)c(w)}$. We then define an iteration matrix $\tilde{H}$ for Jacobi over-relaxation as

$$\tilde{H} = (1-\alpha)I + \alpha\tilde{D}^{-1}\tilde{W},$$

where $0 \leq \alpha \leq 1$. Note that the original matrix $H_{\text{JOR}} = (1-\alpha)I + \alpha D^{-1}W$ is basically the JOR-matrix to solve the system $Lx = 0$. For $\alpha = 1/2$ the matrix is also known as a lazy random-walk matrix. However, instead of solving the system (which has the trivial solutions $x = 0$ and $x = \mathbb{1}$) one uses a few iterations of a process used to solve the linear system

$$x^k = \tilde{H}x^{k-1} = \tilde{H}^k x^0,$$

where x^0 is a random vector sampled over $[-1/2, 1/2]$. Intuitively, the process *assimilates* the random values in a neighborhood of a node. This is repeated R times using different random start vectors. The algebraic distance is then defined by averaging over

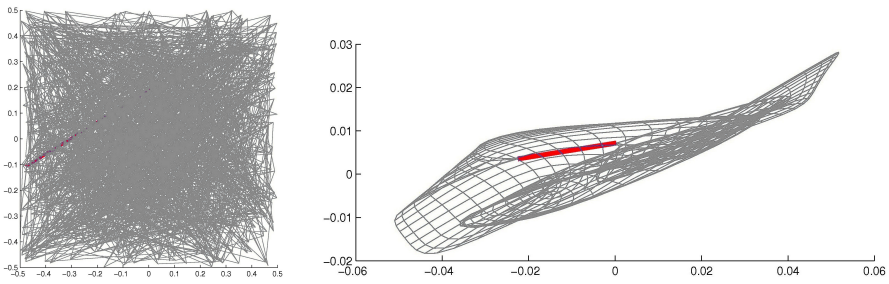


Figure 4.1: Algebraic distance can be used as a measure of connectivity strength. Left: a 12x40 grid with random initialized coordinates. A non-local edge is highlighted in red. Right: the same grid. Coordinates of the nodes have been updated using 15 JOR iterations on each coordinate. The length/distance of the red edge is larger than the length of the local edges. Source: [139].

the result vectors. More precisely, let $x^{(k,r)}$ be the result of the process of the rth try. In [38] it is conjectured that if $|x_i^k - x_j^k| > |x_u^k - x_v^k|$, then the connectivity between u and v is larger than the connectivity between i and j. Hence, for an edge $e \in E$ the 2-normed algebraic distance coupling ρ_e is then defined as

$$\rho_{\{u,v\}} = \sqrt{\sum_{r=1}^{R} |x_u^{(k,r)} - x_v^{(k,r)}|^2}.$$

If $e \notin E$, then the algebraic distance coupling is set to zero. In our experimental settings we use $\alpha = 0.5$, $R = 5$, and $k = 20$. Figure 4.1 gives an intuition on why ρ can be used as a measure of connectivity strength. Recall that if ρ_e is small then its end nodes are strongly coupled and otherwise loosely coupled. Having this in mind, we define an advanced edge rating function

$$\text{ex\_alg}(e) := \text{expansion}^{*2}(e)/\rho_e.$$

This rating function prefers edges for contraction that are strongly coupled with respect to algebraic distance. Note that the iterative process, usually a sparse matrix vector multiplication, can be parallelized easily. We will see in Appendix A that this particular edge rating function is especially helpful for partitioning unstructured graphs.

Matching Algorithms

We use two different matching algorithms within our system. The first strategy aims for partitioning quality, whereas the latter is used in situations where running time is paramount. We start by explaining the Global Paths Algorithm and then a combination of this algorithm with the most simple random matching algorithm.

Global Paths Algorithm. The Global Paths Algorithm (GPA), was proposed by Maue et al. [112] as a synthesis of Greedy and Path Growing algorithms by Drake et al. [58]. The greedy algorithm sorts the edges by descending weight (or rating) and then scans them. If an edge $\{u,v\}$ and its end points are not matched yet, it is put into the matching.

Similar to the Greedy approach, GPA scans the edges in order of decreasing weight (or rating); but rather than immediately building a matching, it first constructs a collection of paths and even length cycles. To be more precise, these paths initially contain no edges. While scanning the edges, the set is extended by successively adding applicable edges. An edge is called applicable if it connects two endpoints of different paths or the two endpoints of an odd length path. Afterwards, optimal solutions/matchings are computed for each of these paths and cycles using dynamic programming. Both algorithms achieve a half-approximation in the worst case, but empirically, GPA gives considerably better results [112]. The first algorithm that achieved a half-approximation was given by Preis [133]. In the context of graph partitioning it has been shown that the GPA algorithm is a good choice when focusing on partitioning quality [87], so that we also use this algorithm in our system.

RandomGPA Algorithm. The most simple random matching algorithm traverses the nodes in a random order and if the current node is not already matched, it chooses a random unmatched neighbor for the matching. RandomGPA is a synthesis of the most simple random matching algorithm and the GPA algorithm. Depending on the level of the multilevel scheme either the random algorithm is used (on the large graphs of the hierarchy) or the GPA algorithm is used (on the small graphs of the hierarchy). The choice of the matching algorithm further depends on the number of blocks that a graph has be partitioned in. We go into more detail when we configure our algorithms. If the random matching algorithm is used, then no edge rating will be computed.

4.1.2 AMG-inspired Coarsening

When one wants to solve a system of linear equations, one of the most traditional approaches to creating hierarchies in Algebraic Multigrid (AMG) is the Galerkin operator [159], which projects a fine system of equations to a system of coarser scale. In the context of graphs this projection is defined as

$$L_c = PL_fP^T,$$

where L_f and L_c are the Laplacians of fine and coarse graphs $G_f = (V_f, E_f)$ and $G_c = (V_c, E_c)$, respectively and P is the projection matrix. The (u,J)th entry of the projection matrix P represents the strength of the connection between a fine node u and a coarse node J. The entries of P are called interpolation weights. They describe both the coarse-to-fine and fine-to-coarse relations between nodes.

The coarsening begins by selecting a dominating set of coarse (or seed) nodes $C \subset V_f$ such that all other fine nodes in $F = V_f \setminus C$ are strongly coupled to C. These nodes will serve as nodes of the next coarser level. The remaining nodes will be split among those nodes using the interpolation matrix P. The whole process is illustrated in Figure 4.2. The *selection* of the seed nodes can be done by traversing all nodes, moving nodes from F to C until the following equation is satisfied for all nodes u in F (initially $F = V_f$, and $C = \emptyset$)

$$\sum_{v \in C \cap N(u)} 1/\rho_{uv} \geq \Theta \cdot \sum_{v \in N(u)} 1/\rho_{uv},$$

where $\Theta \in (0,1)$ is a parameter of coupling strength which is usually set to 0.5. Note that initially the constraint is not fulfilled. Moreover, C is a dominating set if the constraint is fulfilled and the graph does not contain singletons. To see this assume that there is a node in F that is only adjacent to nodes in the same set. In this case, the left hand side of the equation is zero which means that the constraint cannot be fulfilled.

We now have found the seed nodes for the next coarser level and start to explain how the *projection matrix P* is *constructed*. The projection matrix controls how fractions of a node from F are assigned to the seed nodes of the coarser level. The construction differs from other AMG-based approaches for combinatorial optimization problems since the

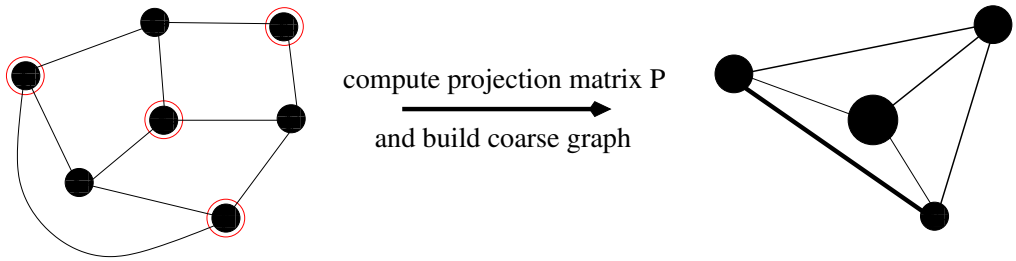


Figure 4.2: (a) Left: begin by selecting a dominating set C (circled nodes), these nodes will serve as nodes of the coarse graph. Compute a projection matrix to estimate the weight of the edges connecting the coarse nodes and to split fine nodes between incident seed nodes. Right: the resulting coarse graph using the dominating set as coarse nodes. The weights of the nodes are also updated.

graph partitioning problem demands *balanced* blocks and graphs should stay sparse on coarse levels of the hierarchy. To achieve this, we avoid too heavy coarse nodes and limit the number of fractions that a node from F can be divided to. In our algorithm the number of fractions is limited to at most two (the two strongest connections).

More precisely, the construction works as follows. The entries P_{vv} of seed nodes v are set to one. For a node $v \in F$ that is not a seed node for the coarse level, we try to find two neighboring seed nodes that maximize the connection strength $(1/\rho_{e_1} + 1/\rho_{e_2})$ to v and that do not become overloaded when v is split among them. Here, we only look at the κ strongest connections. If we do not succeed to find such a pair, we find the strongest connection, w.r.t $1/\rho_e$, to a neighboring seed node such that it does not become overloaded if v is assigned to this seed node. It might still happen that we do not find such a node. In this case, we make v a seed node, i.e. we put it into C. The different stages of the algorithm are sketched in Figure 4.3.

Having found the nodes $S \subset C$ that v is split among, we set the entry in the projection matrix P_{vc} depending on the connection strength to $\frac{1/\rho_{vc}}{\sum_{k\in S} 1/\rho_{vk}}$ for $c \in S$. The weight of a coarse node is set to $c(q \in C) := \sum_v c(v)P_{v,q}$ and the weight of an edge between two coarse nodes is (according to the projection equation) $\omega(c_1,c_2) := \sum_{u\neq v} P_{u,c_1}\omega(u,v)P_{v,c_2}$.

The algorithm can be viewed as a simplified version of the algorithm presented in [30] with the additional restriction on the size of coarse nodes and the possibility to adaptively control the number of fractions that a node can be split in. P_{uq} thus represents the likelihood of i belonging to the qth aggregate. We emphasize the adaptivity of the seed set C, which is updated if seed nodes would become too heavy.

We now discuss the differences between our scheme and the weighted aggregation (WAG) scheme by Chevalier et al. [43]. Both schemes assign fine F-nodes to their coarse C-neighbors. However, algebraic distance is missing which is responsible for the seed sets and the number of fractions a fine node is split into. The projection matrix P is constructed as in classical AMG schemes. Moreover, the WAG scheme produces dense coarse graphs, which we avoid by splitting a node into at most two coarse nodes.

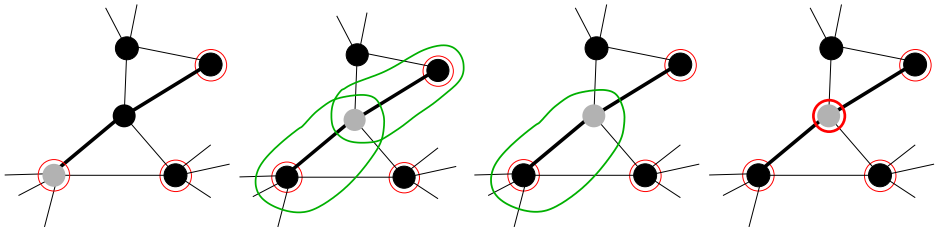


Figure 4.3: Different cases for the construction of interpolation weights P. Seed nodes are surrounded by a circle. (a) For seed nodes v, we set $P_{vv} = 1$. (b) The algorithm first tries to split a node between two seed nodes such that the seed nodes will not be overloaded when the grey node is split among them. (c) If this is not successful, the algorithm tries to find a seed node that is not overloaded when the grey node is aggregated with it. (d) If (b) and (c) are not successful, then the grey node is added to the set of seed nodes. Source: [139].

4.2 Initial Partitioning

The contraction is *stopped* when the number of remaining nodes is below $\max(60k, n/(60k))$. The graph is then small enough to be initially partitioned, i.e. assigning the coarsest nodes to blocks. For historic reasons, some of the experiments in our experimental sections have been obtained using the graph partitioning packages Scotch and Metis as initial partitioning algorithms (we clearly mark these experiments).

To gain independence from external software packages, we also implemented our own initial partitioning algorithms. We use the multilevel recursive bisection scheme to perform this task. More precisely, we perform recursive bisection to obtain a k-partition of the graph. Each bisection step itself uses a multilevel algorithm that stops as soon as the number of nodes is below 120. Greedy graph growing is used on the coarsest level to obtain a bipartition. The multilevel recursive bipartitioning technique has already been described in Chapter 3. We combine it with the local search techniques described within this chapter. If k is not even, we split the graphs into two blocks, V_1 and V_2, such that $c(V_1) \leq (1+\varepsilon)\lfloor \frac{k}{2} \rfloor \lceil \frac{c(V)}{k} \rceil$ and $c(V_2) \leq (1+\varepsilon)\lceil \frac{k}{2} \rceil \lceil \frac{c(V)}{k} \rceil$. Block V_1 will be recursively partitioned in $\lfloor \frac{k}{2} \rfloor$ blocks and block V_2 will be recursively partitioned in $\lceil \frac{k}{2} \rceil$ blocks. Note that allowing ε imbalance in each bipartition step can result in k-partitions having a larger imbalance. If the desired number of blocks is greater than two, we allow an imbalance of 1% in each bipartitioning step to ensure that the imbalance of the k-partition is not too far away from the default value of 3% imbalance. Since initial partitioning is very fast it is repeated several times using different random seeds. The exact number of repetitions depends on the specific algorithm configuration.

4.3 Uncoarsening

We now look at local improvement methods. We present two novel local improvement methods. The first method is based on max-flow min-cut computations between pairs of blocks, i.e. an algorithm to improve a given bipartition. Roughly speaking, this method is applied between all pairs of blocks that share a non-empty boundary. In contrast to previous flow-based methods our objective is the edge cut, whereas previous systems improve conductance or expansion. The second method is called multi-try FM. It is a highly localized k-way search that is repeatedly initialized with a *single* boundary node. Previous methods use *all* boundary nodes for the initialization of the algorithm. At the end of the section, we show how pairwise local searches can be scheduled and how multi-try FM local search can be incorporated. Before we explain these algorithms, we look at different methods for the transfer of a partition from a coarse level to the next finer level in a multilevel algorithm. There are two different schemes, one for each coarsening algorithm that we have seen so far.

4.3.1 Projection

Recall that once a partition of a coarse graph is transfered to a partition of the next finer graph in the multilevel hierarchy, local improvement methods try to reduce the cut size. In the matching case the projection is easily done by assigning the node to the block of its coarse representative. For the AMG-inspired coarsening scheme the projection is more sophisticated. In particular, we have to choose a block to which a fine node is assigned to. This is due to the fact that fractions of it can be assigned to different coarse nodes and hence to different blocks. We assign a fine node v to the block that minimizes $\text{cut}_B \cdot p_B(v)$, where cut_B is the cut after v would be assigned to block B and $p_B(v)$ is a penalty function to avoid blocks that are heavily overloaded. To be more precise, after some experiments we fixed the penalty function to

$$p_B(v) = 2^{\max(0,100\frac{c(B)+c(v)}{L_{\max}})},$$

where $L_{\max}$ is the upper bound for the block weight. Note that slight imbalances (e.g. overloaded blocks), can usually be fixed by our local search algorithms. Also observe that when using the AMG-inspired coarsening scheme, the cuts and balance of partition of a coarse level are *not* the same as the cut and balance of the partition on the next finer level after projection. This is in contrast to the matching based scheme.

4.3.2 Max-Flow Min-Cut Based Search

We now explain how maximum flows can be employed to improve a balanced partition of *two blocks* V_1, V_2 without violating the balance constraint. Informally speaking, we want to find an area around the initial cut of the partition such that each s-t cut in this area

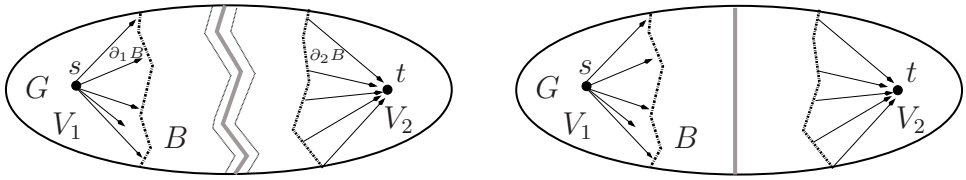


Figure 4.4: The construction of a flow problem $\mathscr{F}$ is shown on the left hand side. Each cut within B induces a cut within the balance constraint in the original graph. A min cut in this area is shown on the right hand side. Source: [143].

corresponds to a balanced cut of the input graph. That yields a local improvement algorithm which can be used in a multilevel framework. First we introduce a few notations. Given a set of nodes $B \subset V$, we define its *border* $\partial B := \{u \in B \mid \exists (u,v) \in E : v \notin B\}$. The set $\partial_1 B := \partial B \cap V_1$ is called *left border* of B and the set $\partial_2 B := \partial B \cap V_2$ is called *right border* of B. A B *induced flow problem* $\mathscr{F}$ is the node induced subgraph $G[B]$ using the edge weights of G as capacities, plus two nodes s,t that are connected to the border of B. More precisely, s is connected to all left border nodes $\partial_1 B$ and all right border nodes $\partial_2 B$ are connected to t. These new edges get the capacity ∞. Note that the additional edges are directed. $\mathscr{F}$ has the *cut property* if each (s,t)-cut induces a feasible cut in G.

The basic idea is to construct a flow problem $\mathscr{F}$ having the cut property. Each min-cut will then yield a feasible improved cut within the balance constraint in G. By performing two breadth-first searches we can find such a set B. Each node touched during these searches belongs to B. The first breadth-first search is initialized with the boundary nodes in V_1 and is only expanded into V_1. As soon as the weight of the area found by this breadth-first search would exceed $(1+\varepsilon)c(V)/2 - c(V_2)$, we stop the search. The second breadth-first search is done analogously for V_2. The constructed flow problem has the cut property since the worst case new weight of V_2 is lower or equal to $c(V_2) + (1+\varepsilon)c(V)/2 - c(V_2) = (1+\varepsilon)c(V)/2$. The same holds for the worst case new weight of V_1. Figure 4.4 explains the construction, an example is shown in Figure 4.5.

There are multiple ways to *improve* this method. This includes *repeated usage* of the method, larger areas B in which the flow problem is solved and the search for better balanced cuts using the information already provided by the max-flow computation. First, if we found an improved cut, we can apply the method again since the initial boundary has changed, i.e. the set B will also have changed. Secondly, we can *adaptively control the size* of the flow problem by dropping the feasibility constraint. This enables us to search for cuts that fulfill the balance constraint in a *larger subgraph*. If the found min-cut in $\mathscr{F}'$ (using $\varepsilon' = \alpha\varepsilon$ with $\alpha > 1$ for construction) fulfills the balance constraint in G, we accept it and increase α to $\min(2\alpha, \alpha')$ where α' is an upper bound for α. Otherwise, the cut is not accepted and we decrease α to $\max(\frac{\alpha}{2}, 1)$. This method is iterated until a maximal number of iterations is reached or if the computed cut yields a feasible partition without a decreased cut. We call this method *adaptive flow iterations*.

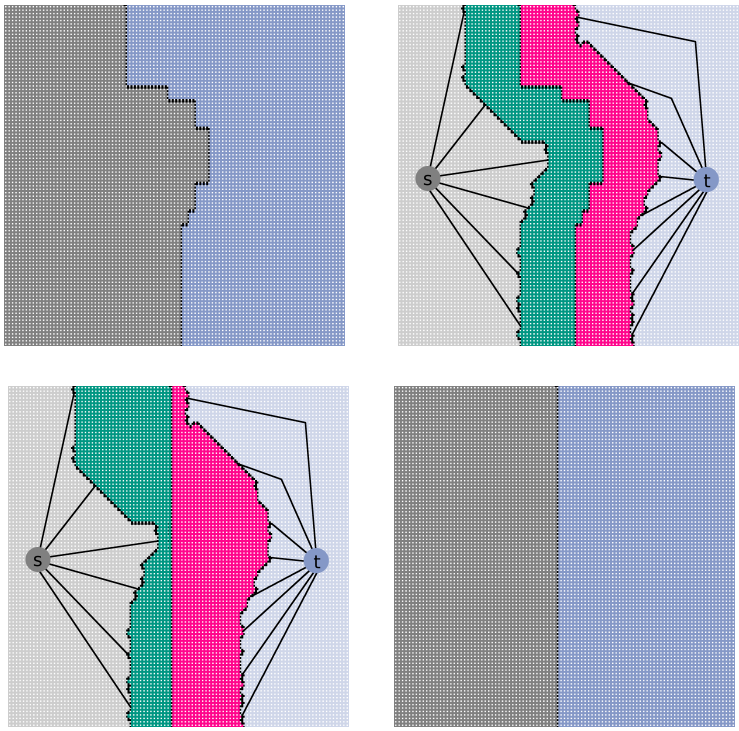


Figure 4.5: Top left: An example grid that is partitioned into two blocks. Top right: The constructed flow problem. Bottom left: The minimum cut in this area. Bottom right: The final partition corresponding to the minimum cut of the flow problem. Source: [143].

Most Balanced Minimum Cuts

The third idea to improve this method is to use the information already provided by the max-flow computation to extract a cut that has better balance. Picard and Queyranne [129] have shown that *one* (s,t)-max-flow contains information about *all* minimum (s,t)-cuts in the graph. We present a heuristic that, given a max-flow, aims to output a better balanced minimum cut with respect to the balance of the induced bipartition in G. Having an algorithm at hand that not only outputs one minimum cut but also a cut with good balance makes the idea to search for feasible cuts in larger subgraphs even more attractive. Recall, for a graph $G = (V,E)$ a set $C \subseteq V$ is a *closed node set* iff for all nodes $u,v \in V$, the conditions $u \in C$ and $(u,v) \in E$ imply $v \in C$. In other words, there is no edge starting in C and ending in $V \backslash C$. An example can be found in Figure 4.6. We now formulate the Lemma of Picard and Queyranne:

Lemma 4.1 (Picard and Queyranne [129]) *Each closed node set containing s in the residual graph of a maximum (s,t)-flow yields a minimum (s,t)-cut.*

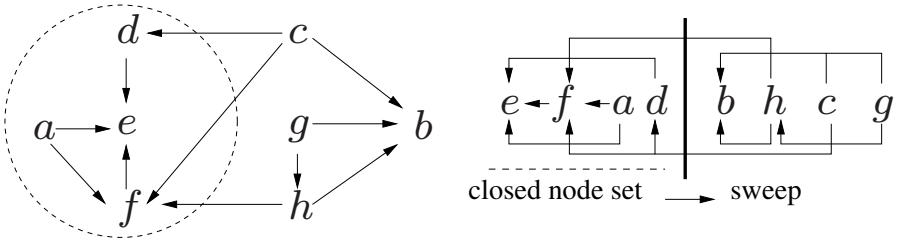


Figure 4.6: Left: the set $C = \{a,d,e,f\}$ is a closed node set since no edge is starting in C and ending in $V \backslash C$. Right: using a reverse topological ordering of a DAG one can output multiple closed node sets. Source: [143].

Specifically, given a closed node set C of the residual graph containing s, the corresponding min-cut is $(C, V \backslash C)$. Note that distinct maximum flows may produce different residual graphs but the closed node sets remain the same. We now show how the residual graph can be compactified and explain our heuristic to compute closed node sets inducing better balanced min-cuts with respect to balance of the bipartition in the original graph.

Observe that a cycle in the residual graph cannot contain a node of both, a closed node set and its complement. Hence, to enumerate all minimum cuts of a graph, Picard and Queyranne [129] contract the strongly connected components of the residual graph which results in a weighted, directed acyclic graph (DAG). We refer to this DAG as *minimum cut representation*. That the problem of finding the most balanced minimum cut is still NP-hard has been shown by Bonsma [28]. Note that each closed node set of the minimum cut representation induces a minimum cut and that we still can compute all minimum cuts using the compactified representation. Hence, we can search for closed node sets containing the component S that contains the source s in the minimum cut representation. We do that by using the following heuristic which is repeated a few times using different random seeds for initialization.

The basic idea is that using a topological order yields closed node sets quite easily. Therefore, we first compute a random topological order by using a randomized depth first search[1]. We sweep through the reverse topological order and sequentially add the components to the closed node set. By doing so we compute closed node sets each inducing a min-cut having a different balance. We use the best balanced minimum cut with respect to the original bipartition found. The closed node set with the best balance occurred using different random topological orders is returned by the algorithm. Note that this procedure can still find cuts that are not feasible in oversized subgraphs, e.g. if there is no feasible minimum cut. Therefore the algorithm is combined with the adaptive strategy from above. We call this method *balanced adaptive flow iterations*.

[1]We also tried an algorithm that iteratively removes nodes having outdegree zero to compute a topological order. Improvements obtained by using this algorithm were negligible.

4.3.3 FM Local Search

We will see in the experimental evaluation of this section that the combinations of local search with the flow-based method is beneficial. We implemented two kinds of FM local improvement schemes within our framework. The first scheme is so called *quotient graph style refinement* [87]. On each pair of blocks that share a non-empty boundary we can apply a two-way local improvement method which only moves nodes between the current two blocks. The second scheme is so called *global k-way local search*. This method has a more global view since it is not restricted to moving nodes between two blocks only. Concretely, we look at two methods – classical k-way search and multi-try k-way search.

Two-way Local Search. The two-way local search algorithm used within our framework is similar to the method used in KaPPa [87]. We present it here for completeness. It is basically the FM-algorithm [66] already presented in Chapter 3: for each of the two blocks A, B under consideration, a priority queue of nodes eligible to move is kept. The priority is based on the *gain*, i.e. the decrease in edge cut when the node is moved to the other side. Each node is moved at most once within a single local search. The queues are initialized randomly with the partition boundary. After a node is moved, its unmoved neighbors become eligible.

There are different possibilities to select a block from which a node shall be moved. The classical FM-algorithm [66] alternates between both blocks. We employ the *TopGain* strategy from KaPPa which selects the block with the largest gain and breaks ties randomly if the gain values are equal. In order to achieve a good balance, TopGain adopts the exception that the block with larger weight is used when one of the blocks is overloaded. After a stopping criterion is applied, we roll back to the best feasible cut found (among ties we choose the partition that has better balance).

To *balance imbalanced blocks* we have two strategies. A soft balancing algorithm always selects a node for movement from the block having larger weight and uses the same roll back mechanism as before, i.e. the cut of a partition cannot increase. A hard rebalancing allows increased cuts. It also selects nodes for movement from the block having larger weight, but the roll back mechanism recreates the partition having the best balance (among ties we choose the partition having the smaller cut).

The two-way method is applied between all adjacent pairs of blocks. A sophisticated algorithm to schedule two-way searches is used to save running time in areas where local search has been repeatedly unsuccessful. We explain it in Section 4.3.4.

k-way Local Search. Our variant of k-way local search uses only one priority queue P which is initialized with the *complete* partition boundary in a random order. The priority is based on the max gain $g(v) = \max_P g_P(v)$ where $g_P(v)$ is the decrease in edge cut when moving v to block P. Again each node is moved at most once and after a node is moved, its unmoved neighbors become eligible. Ties are broken randomly if there is more than

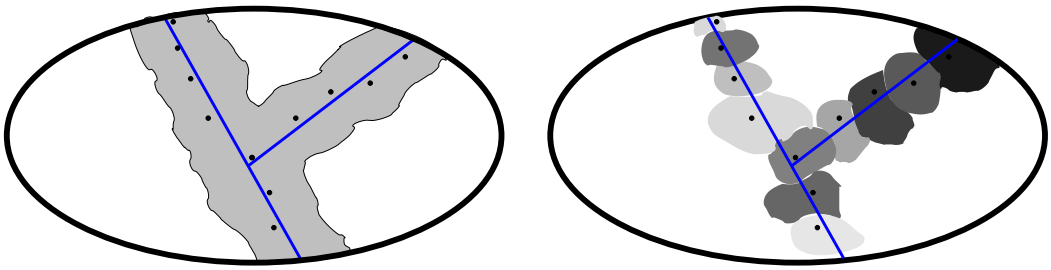


Figure 4.7: On the left hand side a typical k-way local search that is initialized with all boundary nodes is illustrated. On the right hand side Multi-try FM is shown. Multiple k-way searches that are initialized with single boundary nodes are started. In both cases, each node is moved at most once during a round. Source: [143].

one block yielding the same maximum gain. Local search then repeatedly looks for the highest gain node v. However, a node v is not moved if the movement would lead to an unbalanced partition. The k-way local search is stopped if the priority queue P is empty (i.e. each node was moved once) or a stopping criterion described below applies. Afterwards local search is rolled back to the lowest cut fulfilling the balance criterion that occurred. This procedure is then repeated until no improvement is found or a maximum number of iterations is reached.

Multi-try k-way Local Search. Recent results in KaSPar [122] and KaPPa [87] indicate that localization of local search yields increased partition quality. Hence, we introduce a localized variant of the k-way local search algorithm described above. Previous k-way methods were initialized with *all* boundary nodes, i.e. all boundary nodes are eligible for movement at the beginning. In contrast, our method is repeatedly initialized with a *single* boundary node. Intuitively, this introduces a larger amount of diversification compared to the classical method that is restricted to move the boundary node having the largest gain.

Multi-try FM is organized in rounds. A round works as follows. Instead of putting *all* boundary nodes directly into a priority queue, we put the boundary nodes of the current block pair under consideration into a todo list T. Subsequently, we begin a k-way local search starting with a *single* random node v of T. However, local search is only started if v was not touched by a previous localized k-way search in this round. Either way, v is removed from the todo list. We stress that a localized k-way search is restricted to the movement of nodes that have not been touched by a previous local search during the round. This assures that at most n nodes are touched during a round of the algorithm and that the algorithm can be implemented in linear time. Figure 4.7 illustrates the differences to the global k-local search algorithm.

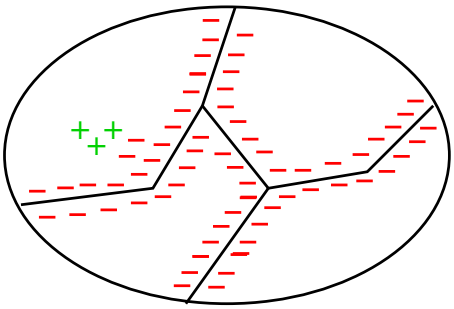


Figure 4.8: An example illustrating why localization helps to find better partitions. – stands for a node with negative gain, + stands for a node with positive gain. Source: [123].

To understand why localization of local search is helpful to create partitions of higher quality, consider the example depicted in Figure 4.8. We see a 4-partition of a graph that is a local optimum in the sense that at least two node movements are necessary until moving a node with positive gain is possible. Recall that classical k-way local search is initialized with all boundary nodes (in this case all of them have negative gain). It then starts to move nodes with negative gain at multiple places of the graph. When it finally moves nodes with positive gain (+) the partition is already much worse than the input partition. Hence, the movement of the positive gain nodes does not yield an improvement with respect to the given input partition. On the other hand, a localized local search that starts close to the nodes with positive gain, can find the positive gain nodes by moving only a small number of nodes with negative gain. Since it did not move as many negative gain nodes as the classical k-way search, it may still finds an improvement with respect to the input partition.

Stopping Criterion. We adopt the stopping criterion proposed in KaSPar by Osipov and Sanders [122] for both k-way local search algorithms. This stopping rule is derived using a random walk model. Gain values in each step are modelled as identically distributed, independent random variables whose expectation μ and variance σ^2 is obtained from the previously observed p steps since the last improvement. Osipov and Sanders [122] derived that it is unlikely for the local search to produce a better cut if

$$p\mu^2 > \alpha\sigma^2 + \beta$$

for some tuning parameters α and β. The parameter β is a base value that avoids stopping just after a small constant number of steps that happen to have small variance. As in KaSPar, we set it to $\beta := \ln n$.

4.3.4 Scheduling Pair-Wise Local Search

Our algorithm to schedule two-way local searches on pairs of blocks is called *active block scheduling*. The main idea is that local search should be done in areas in which change still happens. The algorithm begins by setting every block of a partition *active*. The scheduling algorithm is organized in rounds. In each round, the algorithm tries to improve all adjacent pairs of blocks that have *at least* one active block in a random order. If changes occur during this search, both blocks are marked active for the next round of the algorithm. In this case, local improvement between pairs of blocks can be both, two-way local search and/or flow-based improvement, depending on the configuration. After each pair-wise improvement a multi-try k-way local search is started. Note that the multi-try search is not restricted to moving nodes only between the current block pair (A,B) under consideration. More precisely, if a node is incident to another block than A or B, it can potentially be moved to this block. The todo list T is initialized with the boundary nodes of the current pair of blocks. Each block that changed during this search is also marked active for the next round. The algorithm stops if no active block is left. Pseudocode for the algorithm can be found in Algorithm 1.

Algorithm 1 Active Block Scheduling

```
procedure activeBlockScheduling
  set all blocks active
  while there are active blocks
    L := ⟨edge (A,B) in quotient graph : A active or B active⟩
    set all blocks inactive
    permute L randomly
    for each (A,B) ∈ L do
      pairwiseLocalImprovement(A, B)
      multi-try FM search starting with boundary of A and B
      if anything changed during local search then
        activate blocks that have changed
```

4.4 Global Search

A common approach to obtain high quality partitions is to use a multilevel algorithm multiple times using different random seeds for initialization of the coarsening and local search algorithms. One can then simply use the best partition that has been found.

When using a matching based coarsening scheme, an improvement of this method are Iterated Multilevel Algorithms (V-cycles) which were introduced by Walshaw [162]. The main idea is to iterate coarsening and local search several times, again using different seeds for random tie breaking. However, instead of performing a full restart, one can use the information/partition that has already been obtained. More precisely, after the first

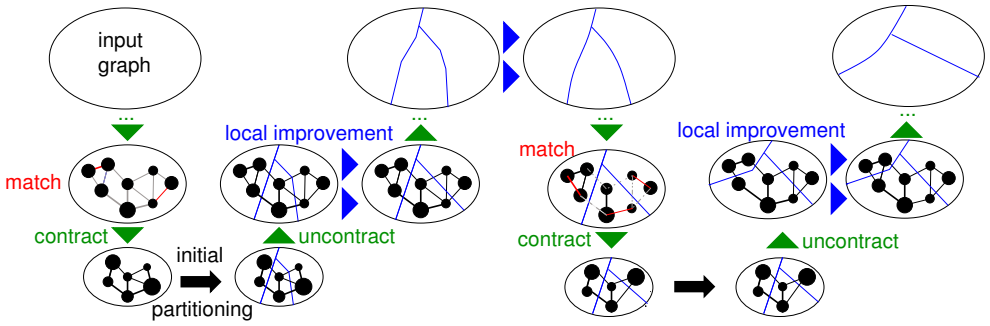


Figure 4.9: An illustration of iterated V-cycles. In this case, the multilevel scheme is iterated two times. During the coarsening of the second iteration, cut edges are not contracted and the input partition is used as initial partition of the coarsest graph. Source: [143].

iteration of the multilevel scheme is done, one performs additional iterations such that cut edges are not contracted. Thus a given partition can be used as initial partition of the coarsest graph (having the same balance and cut as the partition of the finest graph). This ensures non-decreasing partition quality, if the local search algorithm guarantees no worsening. The concept of iterated V-cycles is illustrated in Figure 4.9. Note that our local search algorithms have, to some extent, the ability to climb out of local minima and due to the randomization of the coarsening, the hierarchies created in later cycles are different. That introduces further diversification for local search.

In multigrid linear solvers, Full-Multigrid methods are preferable to simple V-cycles [31]. Therefore, we now introduce two novel global search strategies namely *W-cycles* and *F-cycles* for graph partitioning. A W-cycle works as follows: on *each* level we perform *two recursive calls* using different random seeds for contraction and local search. As soon as the graph is partitioned, edges that are between blocks are not contracted. An F-cycle works similar to a W-cycle with the difference that further recursive calls are only made the second time that the algorithm reaches a particular level. Figure 4.10 gives an abstract view of W- and F-cycles.

In most cases, the partitions of the coarsest graph obtained in a second iteration of the V-cycle by an initial partitioner are much worse than the partition that is used on the coarsest graph (the partition of the finest graph that is given). In other words, in a second iteration of a V-cycle the initial partitioner is not able find better partitions of the coarsest graph than the one that is applied. Therefore no further initial partitioning is used as soon as the graph is partitioned. Experiments in Section 6.4 show that all cycle variants are more efficient than simple restarts of the algorithm.

In order to bound the execution time of the more sophisticated schemes, we introduce a level split parameter d such that further recursive calls are only performed every dth level. We go into more detail after we have analyzed the running time of the global search strategies. Our default value of d is set to two.

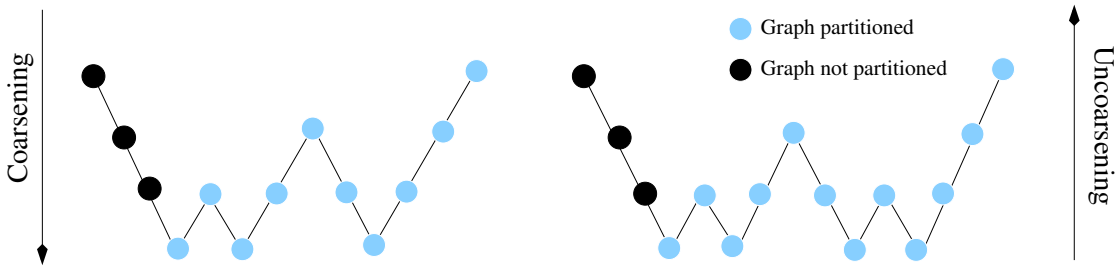


Figure 4.10: An abstract view of F-cycle (left) and W-cycle (right). Edges that run between blocks are not contracted as soon as the graph is partitioned. Source: [143].

Analysis. We now roughly analyse the running time of the different global search strategies under a few assumptions. In the following the shrink factor a names the factor that the graph shrinks (nodes and edges uniformly) during one coarsening step.

Theorem 4.2 *If the time for coarsening and uncoarsening is $T_{cr}(n) := bn$ and a constant shrink factor $a \in [1/2, 1)$ is given, then:*

$$T_W(n) \begin{cases} \leq \frac{1}{1-2a^d} T_V(n) & \text{if } 2a^d < 1 \\ \in \Theta(n \log n) & \text{if } 2a^d = 1 \\ \in \Theta(n^{\log_{d \log 1/a} 2}) & \text{if } 2a^d > 1 \end{cases}$$

$$T_F(n) \leq \frac{1}{1-a^d} T_V(n),$$

where T_V is the time for a single V-cycle and T_W,T_F are the time for a W-cycle and F-cycle with level split parameter d.

Proof. The running time of a single V-cycle is given by

$$T_V^\ell(n) = \sum_{i=0}^{\ell} T_{\text{cr}}(a^i n) = bn \sum_{i=0}^{\ell} a^i = bn \frac{1-a^{\ell+1}}{1-a},$$

where ℓ is the number of levels of the cycle. The running time of a W-cycle with level split parameter d is given by the time for d coarsening and uncoarsening steps plus the time for the two recursive calls on the coarsest of the just created graphs. For the case $2a^d < 1$, we get

$$\begin{aligned} T_W^\ell(n) &= bn \sum_{i=0}^{d-1} a^i + 2T_W(a^d n) = bn \sum_{k=0}^{\ell/d} 2^k \sum_{i=0}^{d-1} a^{k \cdot d} a^i \\ &= bn \sum_{i=0}^{d-1} a^i \sum_{k=0}^{\ell/d} (2a^d)^k \leq T_V^\ell(n) \sum_{k=0}^{\infty} (2a^d)^k \\ &= \frac{1}{1-2a^d} T_V^\ell(n). \end{aligned}$$

The other two cases for the W-cycle follow directly from the master theorem for analyzing divide-and-conquer recurrences. To analyse the running time of an F-cycle, we observe that

$$\begin{aligned} T_F^\ell(n) &\leq \sum_{i=0}^{\ell} T_{\mathrm{V}}^i(a^{i\cdot d}n) = \sum_{i=0}^{\ell} ba^{i\cdot d}n\frac{1-a^{i+1}}{1-a} = \frac{bn}{1-a}\sum_{i=0}^{\ell} a^{i\cdot d}(1-a^{i+1}) \\ &\leq bn\frac{1-a^{\ell+1}}{1-a}\sum_{i=0}^{\infty}(a^d)^i = \frac{1}{1-a^d}T_V^\ell(n). \end{aligned}$$

This completes the proof of the theorem.

4.5 Experimental Evaluation

4.5.1 Preliminaries

Structure. This section is organized as follows. First we define three different configurations of KaFFPa – KaFFPaStrong, Eco and Fast. We then use a matching based scheme to evaluate the local and global search techniques presented within this chapter. We compare ourselves against state of the art partitioning tools and have a closer look at the size of node separators in Chapter 7. Algebraic distance and the algebraic multigrid schemes presented in this chapter are evaluated in Appendix A.

Methodology. In our experimental evaluation we perform two types of experiments, normal tests and tests for effectiveness. Using normal tests we can compare average cut values. To obtain fair comparisons of the different methods, we also use tests for effectiveness of the algorithms. Effectiveness tests create a setting in which the algorithms that are compared have approximately the same amount of time to create a partition. This is necessary because different configurations and components of the algorithm can consume different amounts of running time. Both test schemes are described below.
Normal Tests: We perform ten repetitions on the networks under consideration and report the average of computed cut size, running time and the best cut found. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*. [2]
Effectiveness Tests: Each algorithm is allotted the same amount of time to compute a partition in order to create fair comparisons of the methods. First, for each graph and k each configuration is executed once. Let t be the largest running time that occurred. Now each algorithm gets time $3t$ to compute a good partition, i.e. we take the best partition out of repeated runs. If a configuration can perform a next run depends on the remaining

[2] Because we have multiple repetitions for each instance (graph, k), we compute the geometric mean of the average (**Avg.**) edge cut values for each instance or the geometric mean of the best (**Best.**) edge cut value occurred. The same is done for the running time **t**.

time, i.e. we flip a coin with corresponding probabilities such that the expected time over multiple runs is $3t$. This is repeated five times. The final score is computed as above using these values.

Instances. In this section, we use the following instances for the experimental evaluation of the proposed algorithms: rgg17, rgg18, delaunay17, delaunay18, bcsstk29, 4elt, cti, memplus, cs4, bcsstk32, body, pwt, sphere, t60k, wing, finan512, rotor, bel, nld and af_shell9. Basic properties of these graphs can be found in Chapter 2.4.

Implementation. We have implemented the algorithms described above using C++. Priority queues for the local search are based on binary heaps and bucket queues. Hash tables use the library (extended STL) provided with the GCC compiler. The flow problems are solved using Andrew Goldbergs Network Optimization Library HIPR [40] which is integrated into our code. The AMG coarsening was implemented in a separate framework (using the LEDA library) by Ilya Safro based on [138] and was used to create the multilevel hierarchies for KaFFPa in the AMG case. Experiments that are presented in this chapter have been performed on machine A.

4.5.2 Configuring the Algorithm

We define three configurations of our algorithm: Strong, Eco and Fast. The Strong configuration has been able to achieve the best known partitions for many standard benchmark instances, the Eco version is a good tradeoff between quality and speed and the Fast version of KaFFPa is the fastest system for some large graphs while still improving partitioning quality to the previous fastest system. All configurations use the FM algorithm. The Strong configuration further employs flows, k-way multi-try FM and global search. The Eco configuration also employs flows. We now describe the different configurations of the algorithm.

KaFFPaStrong: The aim of KaFFPaStrong is to achieve the best known partitions for many standard benchmark instances. It uses the GPA algorithm as a matching algorithm combined with the rating function expansion^{*2}. We employ innerOuter on the first level to infer structural information of the graph since the rating function expansion^{*2} has the disadvantage that it evaluates to one on the first level of an unweighted graph. We perform $64/\log k$ initial partitioning attempts using our initial partitioning algorithm. The *local search phase* first employs k-way FM local search (since it converges very fast). It uses the adaptive search strategy from KaSPar [122] with $\alpha = 10$ and is stopped as soon as one round/iteration of the algorithm did not find an improvement or if the number of performed iterations exceeds ten. We continue by performing quotient-graph style refinement. We use the active block scheduling algorithm combined with the *multi-try* k-way local search as described in Section 4.3.4 (using $\alpha = 10$). A pair of blocks is refined as follows: We start with a pairwise FM search which is followed by the *max-flow min-cut* algorithm (including the most balancing cut heuristic). The FM search is stopped if more than 5% of the number of nodes in the current block pair have been moved without yielding an improvement. The upper bound factor for the flow region size is set to $\alpha' = 8$. We use two F-cycles as *global search strategy*.

KaFFPaEco: The aim of KaFFPaEco is to obtain a graph partitioner that is fairly fast and is able to compute partitions of high quality. This configuration uses the RandomGPA matching strategy, i.e. it matches the first $\max(2, 7 - \log k)$ levels using a random matching algorithm. The remaining levels are matched using the GPA algorithm employing the edge rating function expansion^{*2}. It then performs $\min(4, 16/\log k)$ initial partitioning using our initial partitioning algorithm. Local search is configured as follows: again we start with k-way local search as in KaFFPaStrong. In KaFFPaEco the number of k-way rounds/iterations is bounded by $\min(5, \log k)$. We then apply quotient-graph style refinements as in KaFFPaStrong using slightly different parameters. The two-way FM search is stopped if 1% of the number of nodes in the current block pair have been unsuccessfully moved. The flow region upper bound factor is set to $\alpha' = 2$. We use a single V-cycle in order to be competitive regarding running time.

KaFFPaFast: The aim of KaFFPaFast is to get a very fast system for large graphs while still improving partitioning quality to the fastest system. KaFFPaFast matches the first four levels using a random matching algorithm. It then continues by using the GPA algorithm equipped with expansion*^{2} as a rating function. We perform exactly one initial partitioning attempt using our initial partitioning algorithm. Local search works as follows: for $k \leq 8$ we only perform quotient-graph refinement. Each pair of blocks is refined exactly once using the pair-wise FM algorithm. Pairs of blocks are scheduled randomly. For $k > 8$ we only perform one k-way local search round. In both cases, local search is stopped as soon as 15 steps have been performed without yielding an improvement. Again, we do not apply flow-based local search or a more sophisticated global search strategy in order to be competitive regarding running time.

4.5.3 Insights about Flows

In this section we evaluate max-flow min-cut based improvement algorithms. We define a basic FM configuration to compare with as follows. The basic configuration uses the GPA algorithm as a matching algorithm and performs five initial partitioning attempts using Scotch as initial partitioner. It further employs the active block scheduling algorithm equipped with the two-way FM algorithm. The FM algorithm stops as soon as 5% of the number of nodes in the current block pair have been moved without yielding an improvement. Edge rating functions are used as in KaFFPaStrong. During this test our main focus is the evaluation of flows. Hence, we *don't* use k-way local search or multi-try FM search.

The basic configuration is extended by specific components to evaluate the performance of flow-based local search algorithms. A configuration that uses FM (L)ocal search, the (M)ost balanced cut heuristics and (F)lows will be indicated by (+L, +M, +F). Table 4.11 indicates that a multilevel algorithm that only uses flow-based local search techniques obtains cuts and running times which are on average worse than those of the basic FM configuration. Partitioning quality improves if the areas in which min-

Variant	(-L, -M, +F)		(-L, +M, +F)		(+L, -M, +F)		(+L, +M, +F)	
α'	Avg.	$t[s]$	Avg.	$t[s]$	Avg.	$t[s]$	Avg.	$t[s]$
1	$-$**19.58**	0.76	-17.09	0.80	1.64	1.19	1.74	1.22
2	-11.86	0.90	-9.16	0.96	3.66	1.31	4.17	1.39
4	-4.86	1.24	-2.20	1.29	5.27	1.62	6.21	1.76
8	-2.30	2.11	0.41	2.07	5.99	2.41	7.06	2.72
16	-1.88	4.17	0.81	3.92	6.14	4.30	**7.21**	5.01
Ref.	(+L, -M, -F)		2 974	1.13				

Table 4.11: The final score of different algorithm configurations. α' is the flow region upper bound factor. Shown are improvements relative to the basic configuration in %.

Effectiveness	(-L, +M,+F) Avg.	(+L,-M, +F) Avg.	(+L,+M,+F) Avg.
$\alpha' = 1$	−16.41	1.62	1.65
2	−8.26	3.02	3.36
4	−3.05	4.04	4.63
8	**−1.12**	**4.16**	**4.74**
16	−1.29	3.70	4.28
(+L, -M, -F)	2 833	2 831	2 827

Table 4.12: Three effectiveness tests each one with six different algorithm configurations. Shown are improvements relative to the basic configuration in %.

imum cuts are found are enlarged, i.e. using larger factors α', and if the most balanced minimum cut heuristic is enabled. When using the most balanced minimum cut heuristic and a factor $\alpha' = 8$, partitioning quality is comparable to the basic configuration. The running time, however, is still worse compared to the basic configuration that only employs FM local search. Intuitively, the lack of partition quality by flows on its own and the combination of flows with the most balanced cut heuristic is due to the inability of the method to accept suboptimal cuts. This steers the algorithm to solve only small flow problems and hence yields bad cuts.

As a consequence, we also combined both methods with FM local search. Concretely, we use the same FM local search algorithm as in the basic configuration before we apply the flow-based improvement algorithm. This way our algorithm can get close to good cuts and we can use flow-based methods to get the best cut close to this cut. Table 4.11 shows that the combination produces up to 6.14% better cuts on average than the basic configuration. If the most balancing cut heuristic is enabled, cuts are on average 7.21% lower than the cuts produced by the basic configuration.

In general, one hopes that a combination of different local search algorithms improves solution quality. Such combinations come with increased running time which makes comparisons more difficult. To obtain a fair comparison of the methods, we now turn to effectiveness tests. This test gives both algorithm configurations roughly the same amount of time to compute a solution. Table 4.12 shows that the combinations are more effective than repeated executions of the basic FM configuration. The most effective configuration is the basic FM configuration using flows with $\alpha' = 8$ combined with the most balanced cut heuristic. It yields 4.74% lower cuts than the basic configuration in the effectiveness test. We conclude that the combination of flow-based methods with classical local search is a very useful method to obtain high quality graph partitions which cannot be found as effectively by classical local search algorithms on its own.

4.5.4 Insights about Global Search Strategies

We now compare different global search strategies. Our baseline configuration is a relatively fast configuration of our algorithms since we want to evaluate the performance of the different global search strategies. The configuration uses the GPA matching algorithm and performs one initial partitioning attempt using Scotch. Local search employs k-way local search followed by quotient graph style refinements without flow-based local search algorithms. In this test we use this basic configuration and vary the global search strategy. All of our global search algorithms use the level split parameter $d = 2$.

In Table 4.13 we summarize the results of this experiment. Clearly, more sophisticated global search strategies decrease the cut since different levels of the multilevel hierarchy are visited multiple times using different random seeds for coarsening and local search. But they also increase the running time of the algorithm. The effectiveness results in Table 4.13 indicate that repeated executions of more sophisticated global search strategies are always superior to repeated executions of a single V-cycle. We also used Wilcoxon's signed paired rank test (using a 5% significance level) to estimate whether the difference in the performance of the algorithms is statistically significant or not. It turns out that the three strongest configurations yield significantly better cuts than the four weaker configurations.[3] Moreover, all more sophisticated global search strategies yield significantly better cuts than the single V-cycle. The increased effectiveness is due to different reasons. By using a given partition of the finest graph in later iterations, we obtain a very good initial partition of the coarsest graph. Consider the case where two consecutive V-cycles are performed. After the first iteration of the V-cycle, cut edges are not contracted and we use the input partition of the finest level as initial partition of the coarsest graph. This partition is usually not found by an initial partitioning algorithm even though cut edges haven't been contracted. On the other hand, time is saved by using the active block strategy which converges very quickly in later cycles.

Variant	Avg.	t[s]	Eff. Avg.
2 F-cycle	2.69	2.31	**2 806**
3 V-cycle	2.69	2.49	**2 810**
2 W-cycle	2.91	2.77	**2 810**
1 W-cycle	1.33	1.38	2 815
1 F-cycle	1.09	1.18	2 816
2 V-cycle	1.88	1.67	2 817
1 V-cycle	2 973	0.85	2 834

Table 4.13: Results for different global search strategies. Improvements are shown in % relative to the basic configuration.

[3] It is worth mentioning that there are graph classes on which two F-cycles are preferable to three V-cycles. For example, when partitioning the largest random geometric graph (rgg24) into 64 blocks, two F-cycles are about 20% percent faster than three V-cycles and yield smaller cuts on average.

4.5.5 Removal / Knockout Tests

We use two different experiments to evaluate interactions and relative importance of our algorithmic improvements. *Removal tests* use KaFFPaStrong and remove algorithmic components step by step yielding weaker and weaker variants of the algorithm. During *knockout tests* only one component is removed at a time, i.e. each variant is exactly the same as KaFFPaStrong minus the specified algorithmic component. In both cases we use Scotch as initial partitioning algorithm. Table 4.14 summarizes the results.

We shortly summarize the main results. First, in order to achieve high quality partitions we don't need to perform classical global k-way local search. The changes in solution quality are negligible and both configurations (Strong without global k-way and Strong with global k-way) are equally effective. However, the global k-way local search algorithm speeds up the overall running time of the algorithm; hence we included it into KaFFPaStrong. In contrast, removing the multi-try FM algorithm increases average cuts by almost two percent and decreases the effectiveness of the algorithm. As soon as a component is removed from KaFFPaStrong (except for the global k-way search) the algorithm gets less effective.

Variant	Avg.	$t[s]$	Eff. Avg.
Strong	2 683	8.93	2 636
-KWay	−0.04	9.23	0.00
-Multitry	1.71	5.55	1.21
-GSearch	2.42	3.27	1.25
-MB	3.35	2.92	1.82
-Flow	9.36	1.66	6.18

Variant	Avg.	t[s]	Eff. Avg.
Strong	2 683	8.93	2 636
-KWay	−0.04	9.23	0.00
-Multitry	1.27	5.52	0.83
-MB	0.26	8.34	0.11
-Flow	1.53	6.33	0.87

Table 4.14: Left hand side, removal tests: each configuration is the same as its predecessor minus the component shown in the first column. All average cuts are shown as *increases* in cut (%) relative to the values obtained by KaFFPaStrong. Right hand side, knockout tests: each configuration is the same as KaFFPaStrong minus the component shown in the first column. All average cuts are shown as increases in cut (%) relative to the values obtained by KaFFPaStrong. Legend: KWay (classical k-way local search), Multitry (Multi-try k-way local search), GSearch (global search strategy, F-cycle), MB (most balanced minimum cut heuristic), Flow (Max-Flow Min-Cut based search).

4.5.6 Graph Families

In this section, we investigate the behaviour of our algorithms when the graph size increases. Detailed comparisons of our algorithm configurations against other graph partitioning packages can be found in Chapter 7. Here we perform tests on two graph families (*rgg*, *delaunay*). Details about the graphs can be found in Chapter 2.4. We

used machine A for the experiments and repeated them ten times using different random seeds for initialization. Scotch was used as initial partitioning algorithm. Figures 4.15 and 4.16 summarize the results. As soon as the graphs have more than 2^{19} nodes, KaFFPaFast outperforms kMetis in terms of speed and quality. In general the speed up of KaFFPaFast relative to kMetis increases with growing graph size. The largest difference is obtained on rgg24 graph where kMetis has 70% larger running times than our Fast configuration which still produces 2.5% smaller cuts.

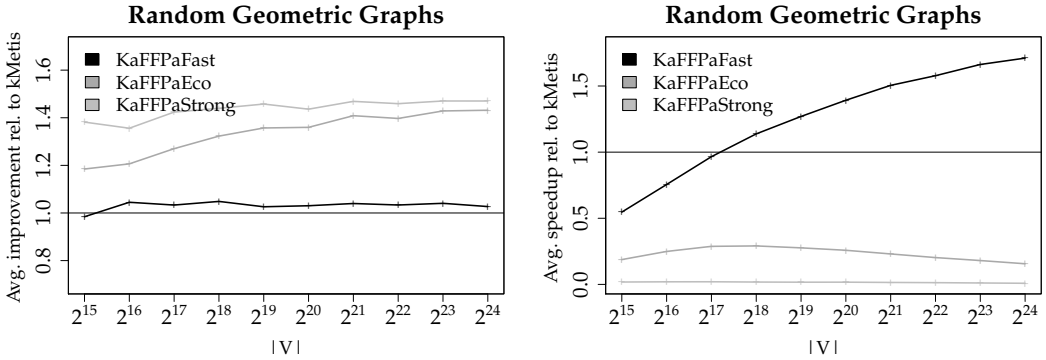


Figure 4.15: Results for Random Geometric Graphs. Source: [143].

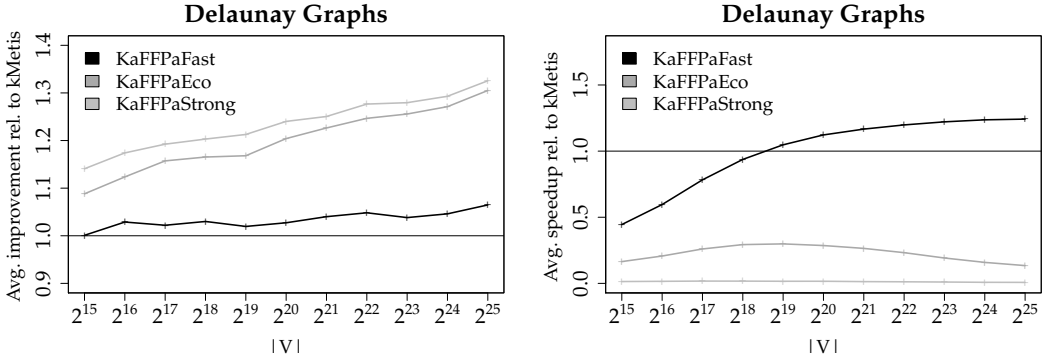


Figure 4.16: Results for Delaunay Graphs. Source: [143].

4.5.7 Walshaw Benchmark

We applied KaFFPaStrong with Scotch as initial partitioner to Walshaw's benchmark archive [157] applying the rules used there, i.e. running time is no issue, but we want to achieve minimal cut values for $k \in \{2,4,8,16,32,64\}$ and balance parameters $\varepsilon \in \{0,0.01,0.03,0.05\}$. We excluded the case $\varepsilon = 0$ since flows are not designed for this case. Note that the partitioning benchmark is very competitive. Over time many researchers submitted their solutions to the archive. Hence, throughout this work when referring to the number of records obtained by a particular system, we refer to the number of records achieved at time of submission. At submission time, KaFFPa computed

317 partitions which are better that previous best partitions reported there: 99 for 1%, 108 for 3% and 110 for 5%. Moreover, it reproduced equally sized cuts in 118 of the 295 remaining cases. After the partitions were accepted, we ran KaFFPaStrong as before and took the previous entry as input. Now overall in 560 out of 612 cases we were able to improve a given entry or have been able to reproduce the current result (in the first run). The complete list of improvements is available in the technical report [141].

4.6 Concluding Remarks

Review. In this chapter we looked carefully at different components of the multilevel graph partitioning scheme. An outcome of the chapter is the graph partitioning framework KaFFPa which is highly configurable to either achieve the best marks in the Walshaw Benchmark, to be a good trade-off between quality and running time, or to be the fastest system on some graphs while still improving partitioning quality compared to the previous fastest system. This is achieved through several improvements of the multilevel algorithm that lead to enhanced partitioning quality. In particular, we looked at two coarsening schemes – matching and AMG-inspired. On social networks the AMG-inspired coarsening scheme has a clear advantage, whereas both schemes produces similar results on graphs that are less unstructured (see Appendix A). Moreover, we looked at two novel local search techniques – max-flow min-cut based local search and a very localized local search algorithm. Our experimental evaluation emphasized that max-flow min-cut based techniques produce superior partitions compared to classical local search, if the search space is expanded and if they are combined with advanced techniques such as classical two-way local search algorithm and a heuristic that given a max-flow can find better balanced minimum cuts. Experimental results also suggest that localization of local search is helpful and that the global search techniques – V-, F- and W-cycle – are superior to repeated starts of the multilevel algorithm.

All of the contributed local and global search techniques presented within this chapter come with an increased running time overhead. Experiments in which compared algorithms get roughly the same amount of time to compute a partition indicate that the proposed local and global search techniques are more effective than previous algorithms.

Future Work. It would be very valuable to go back to parallelization. The integration of flow-based local search techniques as well as the global search techniques could be easily integrated into a distributed memory parallel partitioner such as KaPPa. The global search techniques presented in this chapter will even be more effective in a distributed parallel setting where each processor is responsible for one block of the partition. This is due to the following reasons: during the coarsening of a distributed parallel partitioner a matching algorithm is usually used to the match local edges and edges that run between the blocks, i.e. between the processors. To match edges that run between processors communication is required. Since in later cycles of a global search strategy

cut edges are not eligible for the matching algorithm there will be much less communication needed. An additional step towards a faster distributed memory parallel partitioner has been done in the diploma thesis of Marcel Birn [25]. In his thesis, Marcel Birn developed a scalable and very fast, parallel matching algorithm. On the other hand, the number of available cores in a desktop increases drastically nowadays. For this reason a shared memory parallelization of the proposed techniques is also highly desirable.

There are multiple directions to continue research in the multilevel graph partitioning scheme. Regarding the max-flow min-cut local search technique, it would be interesting to use diffusion to find the areas in which the max-flow min-cut problems are solved. Another interesting idea would be to integrate the presented techniques into the n-level scheme by Osipov and Sanders [122]. The AMG-inspired coarsening scheme is very likely to be improved if one uses a node rating function for the selection of the dominating/seed set similar to the concept of edge ratings.

Indeed, it will be interesting to transfer the ideas presented within this chapter to other combinatorial problems such has hypergraph partitioning, graph drawing and graph clustering, or to look at other objective functions. In his bachelor thesis, Florian Ziegler [168] already started to transfer some ideas presented in this work to the hypergraph bipartitioning problem.

The idea of using edge ratings during coarsening has been proposed in KaPPa [87] and has been extended in this work. However, it remains to have a well understood and unified rating function. A first step towards this direction has been in the bachelor thesis of Maximilian Schuler [151].

5

Evolutionary Graph Partitioning

In the Walshaw benchmark, KaFFPa was beaten mostly for small graphs that combine the multilevel approach with an evolutionary strategy. Hence, we develop an improved evolutionary algorithm that also employs coarse grained parallelism in this chapter.

Roughly speaking, KaFFPaE (KaFFPaEvolutionary) uses KaFFPa to create partitions and modifies the coarsening phase to provide new effective combine operations. These novel mutation and combine operators do not need random perturbations of edge weights in contrast to previous methods that use a graph partitioner by Soper et al. [157] and Delling et al. [49]. In fact, we show in Section 6.4 that using edge weight perturbations decreases the overall quality of the underlying graph partitioner. Due to the parallelization, our system is able to compute partitions that have quality comparable to or better than previous entries in Walshaw's well-known partitioning benchmark *within a few minutes* for graphs of moderate size. Previous methods such as Soper et al. [157] required running times of up to one week for graphs of that size. We therefore believe that in contrast to previous methods, our method is very valuable in the area of high performance computing. We go into more detail at the end of this chapter.

The chapter is organized as follows. We begin with the general structure of an evolutionary algorithm in Section 5.1. We then describe our combine and mutation operations in Section 5.2 and Section 5.3. The parallelization of our evolutionary system is presented in Section 5.4. Experiments in Section 5.5 then look at scalability and quality of the produced partitions of the proposed system.

References. The chapter is based on the conference papers [144, 146] that have been published together with Peter Sanders.

5.1 Evolutionary Algorithms

The general idea behind evolutionary algorithms is to use mechanisms which are highly inspired by biological evolution such as selection, mutation, recombination and survival of the fittest. An evolutionary algorithm starts with a population of individuals and

evolves the population into different populations over several rounds. In each round, the evolutionary algorithm uses a selection rule based on the fitness of the individuals of the population to select good individuals and combine them to obtain improved offspring [79]. There are many rules for the selection of individuals from the population. One possibility that has proven to be effective is the tournament selection rule by Miller et al. [118]. Here, the fittest out of two distinct random individuals from the population is selected.

When an offspring is generated, an eviction rule is used to select a member of the population and replace it with the new offspring. In general one has to take both into consideration, the fitness of an individual and the distance between individuals in the population [15]. There are multiple possibilities to generate offsprings during the course of one generation. One possibility is to only generate one offspring per generation. Such an evolutionary algorithm is called *steady-state* [47]. A typical structure of an evolutionary algorithm is depicted in Algorithm 2.

For an evolutionary algorithm it is of major importance to keep the diversity in the population high [15], i.e. the individuals should not become too similar, in order to avoid a premature convergence of the algorithm. In other words, to avoid getting stuck in local optima, a procedure is needed that randomly perturbs the individuals. In classical evolutionary algorithms, this is done using a mutation operator. It is also important to have operators that introduce unexplored search space to the population. In the next section we introduce more elaborate diversification strategies through a new kind of combine and mutation operators that allow us to explore the search space more effectively.

Interestingly, Inayoshi et al. [91] noticed that good local solutions of the graph partitioning problem tend to be close to one another. Boese et al. [26] showed that the quality of the local optima overall decreases as the distance from the global optimum increases. We will see in the following that our combine operators can exchange good parts of solutions quite effectively especially if they have a small distance.

Algorithm 2 A classical general steady-state evolutionary algorithm.

```
procedure steady-state-evolutionary-algorithm
  create initial population P
  while stopping criterion not fulfilled
    select parents p_1, p_2 from P
    combine p_1 with p_2 to create offspring o
    mutate offspring o
    evict individual in population using o
  return the fittest individual that occurred
```

5.2 Framework for Combine Operations

We now describe the general combine operator framework. This is followed by three instantiations of this framework. In contrast to previous methods that use a multilevel framework, our combine operators do not need perturbations of edge weights since we integrate the operators into our partitioner and do not use it as a complete black box.

Furthermore, all of our combine operators *assure* that the partition quality of the offspring is *at least as good as the best of both parents*. Roughly speaking, the combine operator framework combines an individual/partition $\mathscr{P} = V_1^{\mathscr{P}}, ..., V_k^{\mathscr{P}}$ (which has to fulfill a balance constraint) with a clustering $\mathscr{C} = V_1^{\mathscr{C}}, ..., V_{k'}^{\mathscr{C}}$. Note that the clustering does not necessarily fulfill a balance constraint and k' is not necessarily given in advance. All instantiations of this framework use a different kind of clustering or partition. The partition and the clustering are both used as input for our multilevel graph partitioner KaFFPa in the following sense. Let $\mathscr{E}$ be the set of edges that are cut edges, i.e. edges that run between two blocks, in either $\mathscr{P}$ *or* $\mathscr{C}$. All edges in $\mathscr{E}$ are blocked during the coarsening phase, i.e. they *are not contracted* during the coarsening phase. In other words, these edges are not eligible for the matching algorithm used during the coarsening phase and therefore are not part of any matching computed. Figure 5.1 illustrates this kind of coarsening.

The stopping criterion of the multilevel partitioner is *modified* such that it stops when no contractable edge is left. Note that the coarsest graph is now exactly the same as the quotient graph $\mathscr{Q}'$ of the overlay clustering of $\mathscr{P}$ and $\mathscr{C}$ of G (see Figure 5.2 gives an example). Hence nodes of the coarsest graph correspond to the connected components of $G_{\mathscr{E}} = (V, E \setminus \mathscr{E})$ and the weight of the edges between nodes corresponds to the sum of the edge weights running between those connected components in G.

As soon as the coarsening phase is stopped, we apply the partition $\mathscr{P}$ to the coarsest graph and use this as initial partitioning. This is possible since we did not contract any cut edge of $\mathscr{P}$. Note that due to the specialized coarsening phase and this specialized initial partitioning, we obtain a high quality initial solution on a very coarse graph which is usually not discovered by conventional partitioning algorithms. Since our local search algorithms guarantee no worsening of the input partition and use random tie breaking, we can assure nondecreasing partition quality. Note that local search algorithms can

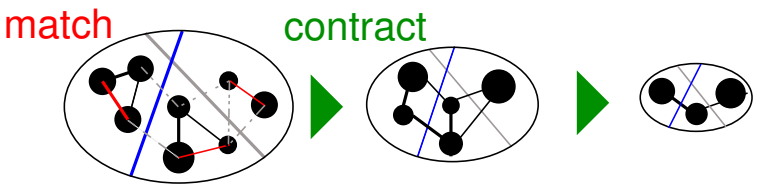


Figure 5.1: A graph G with two partitions, the dark and the light line, are shown. Cut edges are not eligible for the matching algorithm. Contraction is done until no matchable edge is left. The best of the two given partitions is used as initial partition. Source: [144].

effectively exchange good parts of the solution on the coarse levels by moving only a few nodes. Figure 5.2 gives an example.

Also note that this combine operator can be extended to be a multi-point combine operator, i.e. the operator would use p instead of two parents. However, during the course of the algorithm a sequence of two point combine steps is executed which somehow "emulates" a multi-point combine step. Therefore, we restrict ourselves to the case $p = 2$. When the offspring is generated we have to decide which solution should be evicted from the current population. We evict the solution that is *most similar* to the offspring among those individuals in the population that have a cut worse or equal to the cut of the offspring itself. The difference of two individuals is defined as the size of the symmetric difference between their sets of cut edges. This ensures some diversity in the population and hence makes the evolutionary algorithm more effective.

5.2.1 Classical Combine using Tournament Selection

The first instantiation of the combine framework corresponds to a classical evolutionary combine operator C_1. It takes two individuals P_1, P_2 of the population and performs the combine step described above. In this case, $\mathscr{P}$ corresponds to the partition having the smaller cut and $\mathscr{C}$ corresponds to the partition having the larger cut. Random tie breaking is used if both parents have the same cut. The selection process is based on the tournament selection rule by Miller et al. [118], i.e. P_1 is the fittest out of two random individuals R_1, R_2 from the population. The same is done to select P_2. Note that in contrast to previous methods the generated offspring will have a cut smaller than or equal to the cut of $\mathscr{P}$. Due to the fact that our multilevel algorithms are randomized, a combine operation performed twice using the same parents can yield a different offspring. Figure 5.2 illustrates this combine operation.

5.2.2 Cross Combine

In this instantiation of the combine framework C_2, the clustering $\mathscr{C}$ corresponds to a partition of G. But instead of choosing an individual from the population, we create a new individual in the following way. We choose k' uniformly at random in $[k/4, 4k]$ and ε' uniformly at random in $[\varepsilon, 4\varepsilon]$. We then use KaFFPa to create a k'-partition of G fulfilling the balance constraint $\max_i c(V_i) \leq (1+\varepsilon')c(V)/k'$. In general larger imbalances reduce the cut of a partition which then yields good clusterings for our combine operation. To the best of our knowledge there has been no genetic algorithm that performs operations combining individuals from different search spaces. One can extend the idea to combine a partition with an arbitrary graph clustering that fits a specific optimization domain.

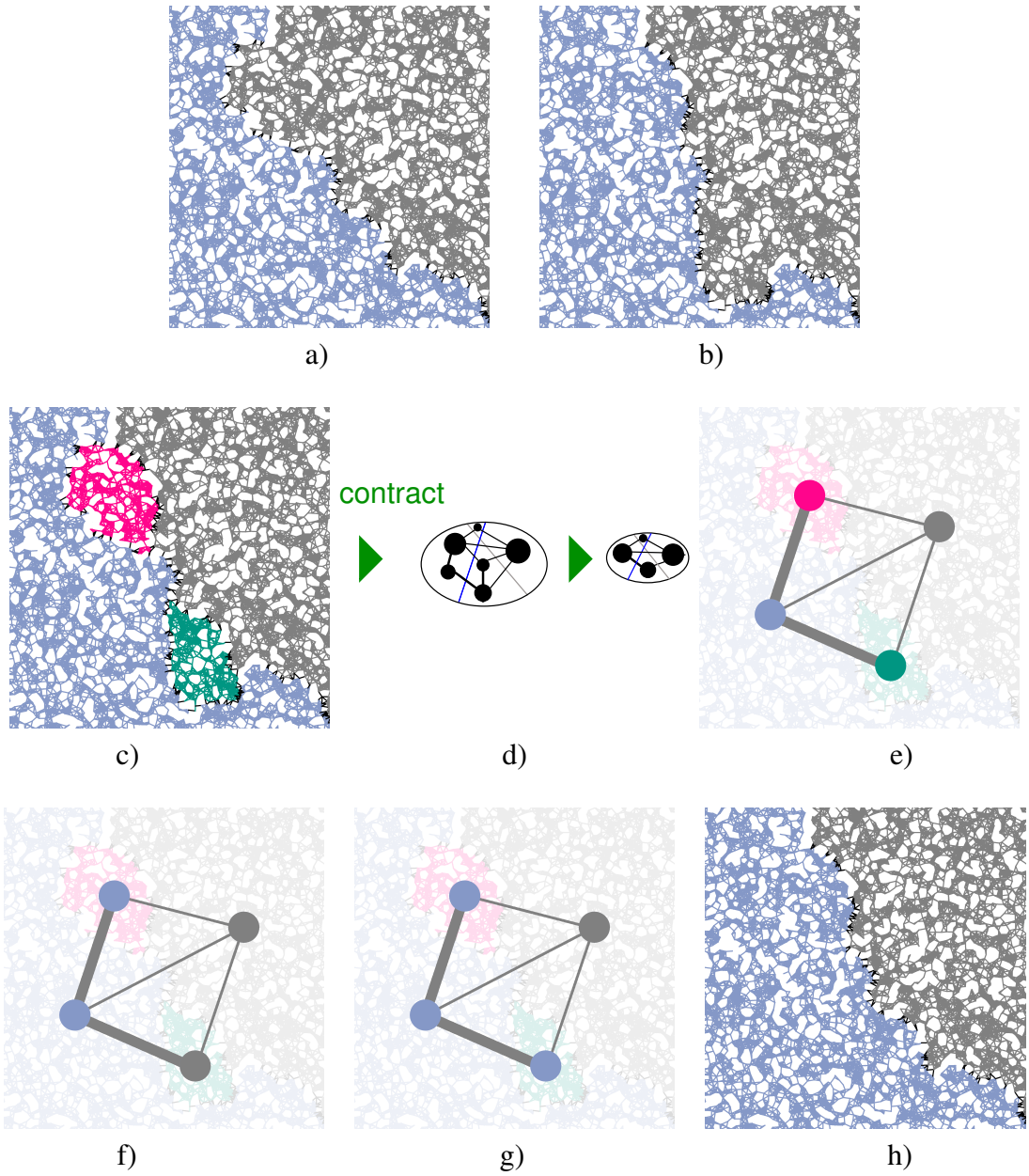


Figure 5.2: An example combine operation of two partitions a), b) of a random geometric graph. In the overlay of these partitions c), only edges that run within the same block can be contracted. The multilevel coarsening phase d) stops as soon as there is no contractable edge left. The resulting graph is the quotient graph of the overlay e). Partition b is applied to this graph f). Local search is applied on all levels of the hierarchy g). We end up with a graph that has the "good" cuts of both input partitions h). Source: [144].

5.2.3 Natural Cuts

Delling et al. [49] introduced the notion of *natural cuts* as a pre-processing technique for the partitioning of road networks. The pre-processing technique is able to find relatively sparse cuts close to denser areas. We use the computation of natural cuts to provide another combine operator, i.e. combining a k-partition with a clustering generated by the computation of natural cuts.

We closely follow their description of this technique: the computation of natural cuts works in rounds. Each round picks a center node v and grows a breadth-first search (BFS) tree. The BFS is stopped as soon as the weight of the tree, i.e. the sum of the node weights of the tree, reaches αU, for some parameters α and U. The set of the neighbors of T in $V \backslash T$ is called the *ring* of v. The *core* of v is the union of all nodes added to T before its size reached $\alpha U / f$ where $f > 1$ is another parameter.

The core is then temporarily contracted to a single node s and the ring into a single node t to compute the minimum s-t-cut between them using the given edge weights as capacities. To assure that every node eventually belongs to at least one core, and therefore is inside at least one cut, the nodes v are picked uniformly at random among all nodes that have not yet been part of any core in any round. The process is stopped when there are no such nodes left.

In the original work by Delling et al. [49] each connected component of the graph $G_C = (V, E \backslash C)$, where C is the union of all edges cut by the process above, is contracted to a single node. Since we do not use natural cuts as a pre-processing technique at this place, we don't contract these components. Instead we build a clustering $\mathscr{C}$ of G such that each connected component of G_C is a block.

This technique yields the third instantiation of the combine framework C_3 which is divided into two stages, i.e. the clustering used for this combine step is dependent on the stage we are currently in. In both stages the partition $\mathscr{P}$ used for the combine step is selected from the population using tournament selection. During the first stage we

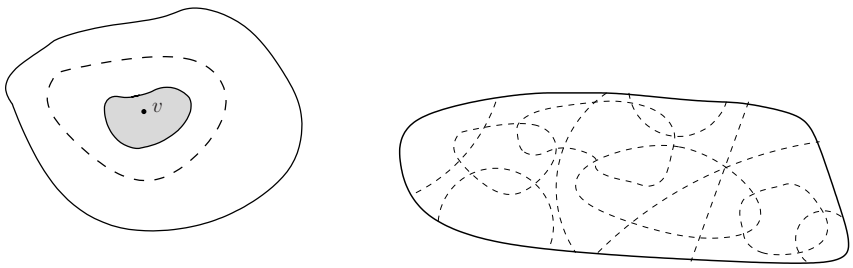


Figure 5.3: Left: the computation of a natural cut. A BFS Tree which starts from v is grown. The gray area is the core. The dashed line is the natural cut. It is the minimum cut between the contracted versions of the core and the ring (shown as the solid line). During the computation several natural cuts are detected in the input graph (right). Source: [144].

choose f uniformly at random in $[5,20]$, α uniformly at random in $[0.75,1.25]$ and we set $U = |V|/3k$. Using these parameters, we obtain a clustering $\mathscr{C}$ of the graph which is then used in the combine framework described above. This kind of clustering is used until we reach an upper bound of ten calls to this combine step. When the upper bound is reached, we switch to the second stage. In this stage we use the clusterings computed during the first stage, i.e. we extract elementary natural cuts and use them to quickly compute new clusterings. An *elementary natural cut* (ENC) consists of a set of cut edges and the set of nodes in its core. Moreover, for each node v in the graph, we store the set of ENCs $N(v)$ that contain v in their core. With these data structures it is easy to pick a new clustering $\mathscr{C}$ (see Algorithm 3) which is then used in the combine framework described above.

Algorithm 3 computeNaturalCutClustering

```
unmarked all nodes in V
for each v ∈ V in random order do
  if v is not marked then
    pick a random ENC C in N(v)
    output C
    mark all nodes in C's core
```

5.3 Mutation Operators

We define two mutation operators, an ordinary and a modified F-cycle. Both mutation operators use a random individual from the current population. The main idea is to iterate coarsening and uncoarsening several times using different seeds for random tie breaking. The first mutation operator M_1 can assure that the quality of the input partition does not decrease. It is basically an ordinary F-cycle which is an algorithm used in KaFFPa (see Chapter 4 for more details). Edges between blocks are not contracted. The given partition is then used as initial partition of the coarsest graph. In contrast to KaFFPa, we now can use the partition as input to the algorithm in the very beginning. This ensures non-decreasing quality since our local search algorithms guarantee no worsening.

The second mutation operator M_2 works quite similar with the small difference that the input partition is not used as initial partition of the coarsest graph. That means we obtain very good coarse graphs but we cannot assure that the final individual has a higher quality than the input individual. In both cases, the resulting offspring is inserted into the population using the eviction strategy described above.

5.4 Putting Things Together and Parallelization

We now explain the parallelization and describe how everything is put together. Each processing element (PE) basically performs the same operations using different random seeds (see Algorithm 4). First we estimate the population size S: each PE performs a partitioning step and measures the time $\bar{t}$ spend for partitioning. We then choose S such that the time for creating S partitions is approximately t_{total}/f where the fraction f is a tuning parameter and t_{total} is the total running time that the evolutionary algorithm is given in advance to produce a partition of the graph. Each PE then builds its own population, i.e. KaFFPa is called several times to create S individuals/partitions. Afterwards the algorithm proceeds in rounds as long as time is left. With corresponding probabilities, mutation or combine operations are performed and the new offspring is inserted into the population.

We choose a parallelization/communication protocol that is quite similar to *randomized rumor spreading* by Doerr et al. [55]. Let p denote the number of PEs used. A communication step is organized in rounds. In each round, a PE chooses a communication partner and sends her the currently best partition P of the local population. The selection of the communication partner is done uniformly at random among the eligible PEs. For a PE, a communication partner p' is called *eligible* if P has not been sent to p' in a previous round. Afterwards, a PE checks if there are incoming individuals and inserts them into the local population using the eviction strategy described above. If P is improved, all PEs are again eligible. This is repeated $\log p$ times. The principle is visualized in Figure 5.4.

Note that the algorithm can be implemented *completely asynchronously*, i.e. there is no need for a global synchronisation. The process of creating individuals is parallelized as follows: Each PE makes $s' = |S|/p$ calls to KaFFPa using different seeds to create s' individuals. Afterwards we do the following $S - s'$ times: the root PE computes a random cyclic permutation of all PEs and broadcasts it to all PEs. Each PE then sends a random individual to its successor in the cyclic permutation and receives an individual from its predecessor in the cyclic permutation. We call this particular part of the algorithm *quick start*.

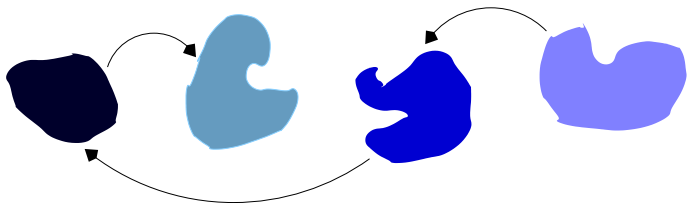


Figure 5.4: Islands of the evolutionary algorithm. Each PE has its own population and performs combine and mutation operations using different random seeds. From time to time the locally best partition is send to other PEs. Source: [144].

Algorithm 4 LocallyEvolve

```
estimate population size S
while time left
  if elapsed time < t_total/f then
    create individual
    insert into local population
  else
    flip coin c with corresponding probabilities
    if c shows head then
      perform a mutation operation
    else
      perform a combine operation
    insert offspring into population if possible
  communicate according to communication protocol
```

The ratio $\frac{c}{10} : \frac{10-c}{10}$ of mutation to combine operations yields a tuning parameter c. As we will see in Section 5.5.1 the ratio $1 : 9$ is a good choice. After some experiments, we fixed the ratio of the mutation operators $M_1 : M_2$ to $4 : 1$ and the ratio of the combine operators $C_1 : C_2 : C_3$ to $3 : 1 : 1$. Note that the communication step in the last line of the algorithm could also be performed only every x-iterations (where x is a tuning parameter) to save communication time. Since the communication network of our test system is very fast (see experimental section), we perform the communication step in each iteration.

5.5 Experimental Evaluation

Methodology. We mostly present two kinds of data: average values and plots that show the evolution of solution quality (*convergence plots*). In both cases, we perform multiple repetitions. The number of repetitions is dependent on the test that we perform. Average values over multiple instances are obtained as follows: for each instance (graph, k), we compute the geometric mean of the average edge cut values for each instance. We now explain how we compute the convergence plots. We start explaining how we compute them for a single instance I: whenever a PE creates a partition it reports a pair (t, cut), where the timestamp t is the currently elapsed time on the particular PE and cut refers to the cut of the partition that has been created. When performing multiple repetitions, we report average values ($\bar{t}$, avgcut) instead. After the completion of KaFFPaE we are left with P sequences of pairs (t, cut) which we now merge into one sequence. The merged sequence is sorted by the timestamp t. The resulting sequence is called T^I. Since we are interested in the evolution of the solution quality, we compute another sequence $T^I_{\min}$. For each entry (in sorted order) in T^I, we insert the entry $(t, \min_{t' \leq t} \text{cut}(t'))$ into $T^I_{\min}$. Here, $\min_{t' \leq t} \text{cut}(t')$ is the minimum cut that occurred until time t. $N^I_{\min}$ refers to the normalized sequence, i.e. each entry (t, cut) in $T^I_{\min}$ is replaced by (t_n, cut) where $t_n = t/t_I$ and t_I is the average time that KaFFPa needs to compute a partition for the instance I. To obtain average values over *multiple instances* we do the following: for each instance we label all entries in $N^I_{\min}$, i.e. (t_n, cut) is replaced by (t_n, cut, I). We then merge all sequences $N^I_{\min}$ and sort by t_n. The resulting sequence is called S. The final sequence S_g presents *event based* geometric averages values. We start by computing the geometric mean cut value $\mathscr{G}$ using the first value of all $N^I_{\min}$ (over I). To obtain S_g, we basically sweep through S: for each entry (in sorted order) (t_n, c, I) in S, we update $\mathscr{G}$, i.e. the cut value of I that took part in the computation of $\mathscr{G}$ is replaced by the new value c, and insert $(t_n, \mathscr{G})$ into S_g. Note, c can be only smaller than or equal to the old cut value of I.

Instances. In this section, we tune the parameters and look at the scalability of our algorithms on the following graphs: rgg15, rgg16, delaunay15, delaunay16, uk, luxemburg, 3elt, 4elt, fe_sphere, cti and fe_body. Different combine operators are compared using larger instances: rgg17, rgg18, delaunay17, delaunay18, bel, nld, t60k, wing, fe_tooth, fe_rotor and memplus. Basic properties of these graphs can be found in Chapter 2.4.

Implementation. We have implemented the algorithms described in this chapter using C++. We use KaFFPaStrong as base case partitioner (with Scotch as initial partitioning algorithm). Experiments in Section 5.5.1 and Section 5.5.2 have been performed on machine A and experiments in Section 5.5.3 have been conducted on machine C.

5.5.1 Parameter Tuning

We now tune the fraction parameter f and the ratio between mutation and combine operations using KaFFPaStrong as a partitioner. We do this on the small testset because running times for a single graph partitioner call are not too large. To save running time, we focus on $k = 64$ for tuning the parameters. For each instance we gave KaFFPaE ten minutes and 16 PEs to compute a partition. The quick start option is disabled.

We start by tuning the fraction parameter f. For this test the flip coin parameter c is set to one. In Figure 5.5 we can see that the algorithm is not too sensitive about the exact choice of this parameter. However, larger values of f speed up the convergence rate and improve the result achieved in the end. Since $f = 10$ and $f = 50$ are the best parameter in the end, we choose $f = 10$ as our default value. For tuning the ratio $\frac{c}{10} : \frac{10-c}{10}$ of mutation and combine operations, we set f to ten. We can see that for smaller values of c the algorithm is not too sensitive about the exact choice of the parameter. However, if c exceeds 8 the convergence speed slows down which yields worse average results in the end. We choose $c = 1$ because it has a slight advantage in the end.

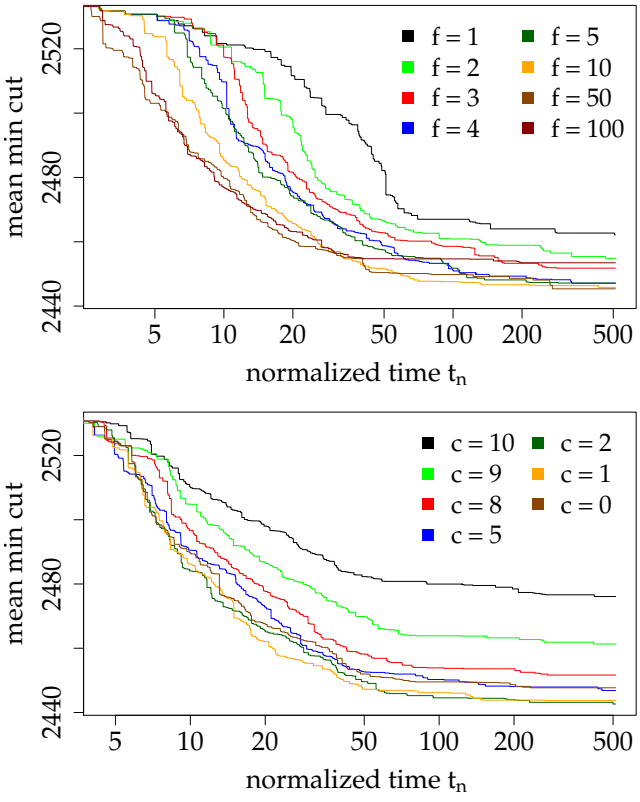


Figure 5.5: Convergence plots for the parameter tuning of the *fraction* parameter f using $c = 1$ (top) and the *flip coin* parameter c using $f = 10$ (bottom). Source: [144].

5.5.2 Scalability

In this section we study the scalability of our algorithm. We do the following to obtain a fair comparison: basically each configuration has the same amount of computing time, i.e. when doubling the number of PEs used, we divide the time that KaFFPaE has to compute a partition per instance by two. To be more precise, when we use one PE, KaFFPaE has $t_1 = 15360s$ to compute a partition of an instance. When KaFFPaE uses p PEs, then it gets time $t_p = t_1/p$ to compute a partition of an instance. For all the following tests the quick start option is enabled. To save running time, we fix k to 64. We perform five repetitions per instance. We can see in Figure 5.6 that using more processors speeds up the convergence speed and up to $p = 128$ also *improves* the quality in the end (in these cases the speedups are optimal in the end). This might be due to island effects [6]. For $p = 256$ results are worse compared to $p = 1$. This is because the algorithm is barely able to perform combine and mutation steps, due to the very small amount of time given to KaFFPaE (60 seconds). On the largest graph of the testset (delaunay16) we need about 20 seconds to create a partition into $k = 64$ blocks.

We now define the notion of pseudo speedup $S_p(t_n)$ which is a measure for speedup at a particular normalized time t_n compared to the configuration using one PE. Let $c_p(t_n)$ be the mean minimum cut that KaFFPaE has computed using p PEs until normalized time t_n. The pseudo speedup is then defined as $S_p(t_n) = c'_1(t_n)/c'_p(t_n)$, where $c'_i(t_n) = \min_{c_i(t') \leq c_1(t_n)} t'$. If $c'_p(t) > c'_1(t_n)$ for all t, we set $S_p(t_n) = 0$ (in this case the parallel algorithm is not able to compute the result computed by the sequential algorithm at normalized time t_n; this is only the case for $p = 256$). We can see in Figure 5.6 that after a short amount of time we reach super linear pseudo speedups in most cases.

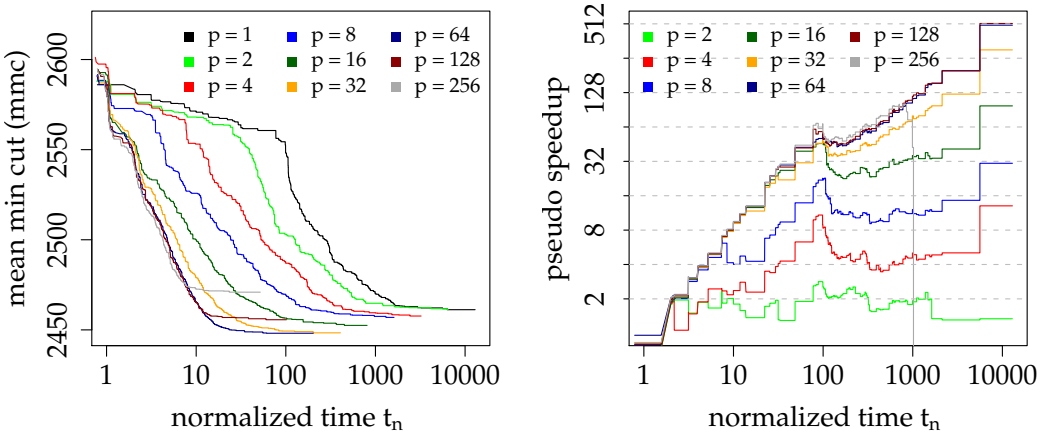


Figure 5.6: Scalability of our algorithm: (left) a normal convergence plot, (right) pseudo speedup $S_p(t_n)$. Source: [144].

5.5.3 Quality of Combine Operators

We now look into the effectiveness of our combine operator C_1. We conduct the following experiment: we compare the best result of three repeated executions of KaFFPa (*K3R*) against a combine step (*KC*), i.e. after creating two partitions we report the result of the combine step C_1 combining both individuals. The same is done using the combine operator of Soper et al. [157] (*SC*), i.e. we create two individuals using perturbed edge weights as in [157] and report the cut produced by the combine step proposed there (the best out of the three individuals). For more information on this particular combine operation see Chapter 3. We also present best results out of three repetitions when using perturbed edge weights as in Soper et al. (*S3R*). Since our partitioner does not support double type edge weights, we computed the perturbations and scaled them by a factor of 10 000 (for S3R and SC). We performed ten repetitions on the middle sized testset. Results are reported in Table 5.7. A table presenting absolute average values and comparing the running time of these algorithms can be found in the Appendix. We can see that for large k our new combine operator yields improved partition quality in comparable or less time (KC vs. K3R). Most importantly, we can see that edge biases decrease the solution quality (K3R vs. S3R). This is due to the fact that edge biases may make edge cuts optimal that are not close to optimal in the unbiased problem. For example on 2D grid graphs, we have straight edge cuts that are optimal. Random edge biases make bended edge cuts optimal. However, these cuts are not close to optimal cuts of the original graph partitioning problem. Moreover, local search algorithms (Flow-based, FM-based) work better if there are a lot of equally sized cuts.

Algo.	S3R	K3R	KC	SC
k	Avg.	improvement %		
2	591	2.4	1.6	0.2
4	1 304	3.4	4.0	0.2
8	2 336	3.7	3.6	0.2
16	3 723	2.9	2.0	0.2
32	5 720	2.7	3.3	0.0
64	8 463	2.8	3.0	−0.6
128	12 435	3.6	4.5	0.0
256	17 915	3.4	4.2	−0.1

Table 5.7: Comparison of quality of different algorithms relative to S3R.

5.5.4 Walshaw Benchmark

As we did in Chapter 4, we also applied KaFFPaE [144] to Walshaw's benchmark archive [157]. We focus on $\varepsilon \in \{1\%, 3\%, 5\%\}$ since KaFFPaE (more precisely KaFFPa) is not made for the case $\varepsilon = 0$. At submission time, KaFFPaE computed 300 partitions which are better than previous best partitions reported there: 91 for 1%, 103 for 3% and 106 for 5%. Moreover, it reproduced *equally sized* cuts in 170 of the 312 remaining cases. The complete list of improvements is reported in the technical report [142]. Overall our systems (including KaPPa, KaSPar, KaFFPa, KaFFPaE) improved or reproduced the entries in 550 out of 612 cases (for $\varepsilon \in \{0.01, 0.03, 0.05\}$) at submission time of KaFFPaE's solutions.

5.6 Concluding Remarks

Review. In this chapter we integrated our multilevel graph partitioner KaFFPa into an evolutionary strategy. More precisely, the coarsening phase was modified such that KaFFPa could provide effective combine and mutation operations. Intuitively, the combine operations assemble good parts of solutions into a single partition. The presented combine operation framework is very flexible so that a partition can be combined with an arbitrary domain specific graph clustering. We believe that this framework could be of more general interest to the genetic algorithm community.

The resulting partitioner, KaFFPaE, uses scalable coarse grained parallelism to speed up computations. Due to the scalable parallelization, KaFFPaE is able to compute the best known partitions for many standard benchmark instances in only a *few minutes*, whereas previous evolutionary algorithms that combined the multilevel idea with evolutionary ideas by perturbation of the edge weights needed up to one week per instances. Hence, KaFFPaE could be still helpful in the area of high performance computing.

More precisely, the running time of KaFFPaFast, e.g. on a continental size road network such as europe, is within a factor three of the running time of Dijkstra's algorithm. While the running time of KaFFPaEco and KaFFPaStrong is larger (all of them can be used as base case partitioner within the evolutionary framework), the time actually needed for partitioning the graph by the evolutionary algorithm is small compared to the execution time of a numeric simulation or pre-processing technique.

Future Work. It would be interesting to integrate other partitioners if they implement the possibility to block edges during coarsening and to use the given partitioning as initial solution. The current implementation of the parallelization is very coarse grained. Yet again it would be interesting to add further parallelization, e.g. use a parallel graph partitioner to provide the combine operations. On the other hand, it would be interesting to try other domain specific combine operators, e.g. on social networks it could be interesting to use a modularity clusterer to compute a clustering for the combine operation.

6

Highly Balanced Graph Partitioning

In the previous chapters, we have developed an improved multilevel algorithm KaFFPa and based on that an improved evolutionary algorithm, KaFFPaE. Both of these algorithms compute partitions of very high quality in a reasonable amount of time when some imbalance $\varepsilon > 0$ is allowed. However, they are not yet very good for small values of ε, in particular for the perfectly balanced case $\varepsilon = 0$. Hence, we develop new techniques that work well for small values of ε in this chapter.

State-of-the-art local search algorithms exchange nodes between blocks of the partition trying to decrease the cut size while also maintaining balance. This highly restricts the set of possible improvements. In this chapter, we introduce new techniques that relax the balance constraint for node movements but globally maintain balance by combining multiple local searches. We reduce the combination problem to finding negative cycles in a directed graph, exploiting the existence of efficient algorithms for this problem. We also provide balancing variants of these techniques that are able to make infeasible partitions feasible. This makes our partitioner the only current system which is able to guarantee any balance constraint. From a meta-heuristic point of view, our techniques are an example for a local improvement technique that vastly increases the size of the neighborhood by efficiently combining many highly localized infeasible improvements into a feasible one.

We begin this chapter in Section 6.1 by describing the very basic idea that allows us to find combinations of simple node movements. We then explain directed local searches and extend the basic idea to a complex model containing more node movements. This is followed by a description on how these techniques are integrated into KaFFPaE in Section 6.2. We shortly present further algorithms in Section 6.3. A summary of extensive experiments done to evaluate the performance of the proposed algorithms is presented in Section 6.4.

References. This chapter is based on the conference paper [147] which has been published together with Peter Sanders.

6.1 Globalized Local Search

In this section we describe our local search and balancing algorithms for strictly balanced graph partitioning. Roughly speaking, all of our algorithms consist of two components. The *first component* are local searches on pairs of blocks that share a non-empty boundary, i.e. all edges in the quotient graph. These local searches are not restricted to the balance constraint of the graph partitioning problem and are undone after they have been performed. The *second component* uses the information gathered in the first component. That means we build a model using the node movements performed in the first step enabling us to find combinations of those node movements that *maintain balance*.

We begin by describing the very basic algorithm and go on by presenting an advanced model which enables us to combine complex local searches. This is followed by a description on how local search and balancing algorithms are put together.

6.1.1 Basic Idea – Using A Negative Cycle Detection Algorithm

We start with a very simple case where the first component only moves single nodes. A node in the graph G can have two states *marked* and *unmarked*. By default a node is unmarked. It is called *eligible* if it is not adjacent to a previously marked node. We now build the model of the underlying partition of the graph G, $\mathcal{Q} = (\{1,\ldots,k\},\mathcal{E})$, where $(A,B) \in \mathcal{E}$ if there is an edge in G that runs between the blocks A and B. We define edge weights $\omega_{\mathcal{Q}} : \mathcal{E} \to \mathbb{R}$ in the following way: for each *directed* edge $e = (A,B) \in \mathcal{E}$ in a random order, find an *eligible* boundary node v in block A having maximum gain $g_{\max}(A,B)$, i.e. a node v that maximizes the reduction in cut size when moving it from block A to block B. If there is more than one such node, we break ties randomly. Node v is then marked. The weight of e is then $\omega_{\mathcal{Q}}(e) := -g_{\max}(A,B)$, i.e. the negative gain value associated with moving v from A to B. Note that, in general, $\omega_{\mathcal{Q}}((A,B)) \neq \omega_{\mathcal{Q}}((B,A))$. An example for this basic model is shown in Figure 6.1. Observe that the

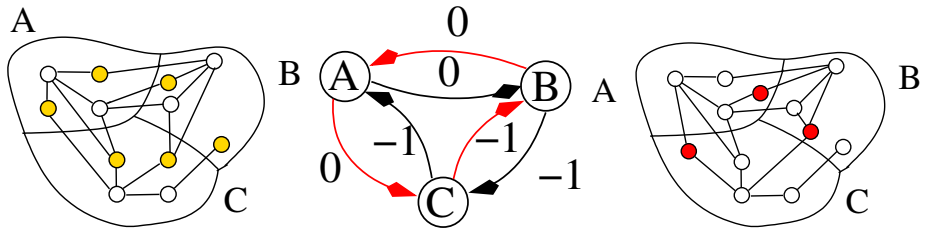


Figure 6.1: Left: an example graph that is partitioned into three parts (A, B and C) of four nodes each. Possible candidates for movement are highlighted. Middle: the corresponding model is shown and one negative cycle is highlighted. Right: the updated partition after the node movements associated with the cycle are performed is shown. Moved nodes are highlighted. The reduction in the number of edges cut is equal to the weight of the cycle. Source: [147].

basic model is a directed and weighted version of the quotient graph and that the selected nodes form an independent set of the input graph G.

Note that each cycle in this model defines a set of node movements and furthermore when the associated nodes of a cycle are moved, then each block contains the same number of nodes as before. Also the weight of a cycle in the model is equal to the reduction in the cut when the associated node movements are performed. However, the most important aspect is that a *negative cycle* in the model corresponds to a set of node movements that will decrease the overall cut and maintain the balance of the partition.

To detect a negative cycle in this model, we introduce a node s and connect it to all nodes in $\mathcal{Q}$. The weight of the inserted edges is set to zero. We can apply a standard shortest path algorithm [39] that can handle negative edge weights to detect a negative cycle. If the model contains a negative cycle, we can perform a set of node movements that will not alter the balance of the blocks since each block obtains and emits a node.

We can find additional useful augmentations by connecting underloaded blocks to s by a zero weight edge. Now, negative cycles containing s change some block weights but will not violate any additional balance constraints. Indeed, when the node following s is overloaded initially, this overload will be reduced.

An interesting observation is that the algorithm can be seen as an extension of the classical FM algorithm that swaps nodes between two adjacent blocks (two at a time) which is basically a negative cycle of length two in our model if the gain of the two node movements is positive. If there is no negative cycle in the model, we have to think about diversification and balancing strategies which is done in the following sections.

Diversification by Zero Gain Cycle Moves

A zero weight cycle in the basic model is associated with a set of node movements that keep the cut unchanged and the block weights constant. After such a movement is performed, it might be possible to find further negative cycles in the basic model since candidates for movements and gain values may have changed. Hence these cycles can be useful to introduce some diversification.

Nonetheless, on general graphs it is NP-complete to decide whether a weighted graph contains a simple cycle that has weight zero [102], i.e. the sum of the edge weights of this cycle is zero. However, we will see that if a graph does not contain a negative cycle, we can decide whether it contains a cycle of weight zero in polynomial time and output one if one exists. This can be done by using the following technique. As soon as the model described above does not contain negative cycles, we compute a shortest path tree starting at s. By doing this we can define node potentials $\Pi : \{1,\ldots,k\} \to \mathbb{R}$ as the shortest path distances from s to all other nodes. We then define modified edge weights $\ell_{\mathcal{Q}}(e = (A,B)) := \omega_{\mathcal{Q}}(e) + \Pi(A) - \Pi(B)$. It is quite easy to see that the weight of a cycle in $\mathcal{Q}$ does not change when we use $\ell_{\mathcal{Q}}$ instead of $\omega_{\mathcal{Q}}$. In particular cycles that have weight zero w.r.t $\omega_{\mathcal{Q}}$ will have weight zero w.r.t. $\ell_{\mathcal{Q}}$. Another important observation is that $\ell_{\mathcal{Q}}$ is a non-negative function. Hence, in order to detect a zero weight cycle we can

evict all edges e with $\ell_{\mathcal{Q}}(e) > 0$ since they cannot be a part of a cycle having weight zero. After this is done, we compute all strongly connected components of this graph. If there is a strongly connected component that contains more than one node, then the graph contains a cycle that has weight zero. To output one zero weight cycle, we pick a random node N out of the components having more than one node. Starting at this node we perform a random walk in its component until we find a node that we have already seen M (which is not necessarily N). It is then fairly simple to output the respective cycle starting at M. Note that if the component contains j nodes then the random walk will stop after at most j iterations. As soon as we have found a cycle of weight zero, we can perform the node movements that are associated with the edges of the cycle.

6.1.2 Advanced Model

We now integrate advanced local search algorithms. Each edge in the advanced model stands for a *set* of node movements found by a local search. Hence, a negative cycle corresponds to a combination of local searches with positive overall gain that maintain balance or that can improve balance. Before we build the advanced model, we perform *directed local search* on each pair of blocks that share a non-empty boundary, i.e. each pair of blocks that is adjacent in the quotient graph. A local search on a directed pair of blocks (A,B) is only allowed to move nodes from block A to block B. The order in which the directed local search between a directed pair of blocks is performed is random. That means we pick a random directed adjacent pair of blocks on which local search has not been performed yet and perform local search as described below. This is done until local search was done between all directed adjacent pairs of blocks once.

Directed Local Search

We now explain how we perform a directed local search between a pair of blocks (A,B). A directed local search between two blocks A and B is very localized akin to the multi-try method used in Chapter 4. However, a directed local search between A and B is restricted to moving nodes from block A to block B. It is similar to the FM-algorithm: we start with a *single* random eligible boundary node of block A having maximum gain $g_{\max}(A,B)$ and put this node into a priority queue. The priority queue contains nodes of the block A that are valid to move. The priority is based on the *gain*, i.e. the decrease in edge cut when the node is moved from block A to block B. We always move the node that has the highest priority to block B. After a node is moved, its eligible neighbors that are in block A, are inserted into the priority queue. We perform at most τ steps per directed local search, where τ is a parameter. Note that during a directed local search we only move nodes that are not incident to a node moved during a previous directed local search. This restriction is necessary to keep the model described below accurate. Thus we *mark all nodes* touched during a directed local search *after* it was performed which also implies that each node is moved at most once. In addition, all moved nodes are *moved back* to

their origin, since these movements would make the partition imbalanced. We stress that all nodes incident to nodes that have been moved during a directed local search are not *eligible* for any later local search during the construction since this would make the gain values computed imprecise.

The Model Graph

The advanced model allows us to find combinations of directed local searches such that the balance of the given partition is at least maintained. The challenge here is that, in contrast to movements of single nodes, we cannot combine arbitrary local searches since they do not all move the same number of nodes. Hence, we specify a more sophisticated graph with the property that a negative cycle maintains feasibility.

The local search process described above yields, for each pair of blocks $e = (A,B)$ in the quotient graph, a sequence of node movements S_e and a sequence of gain values g_e. The dth value in g_e corresponds to the reduction in the cut between the pair of blocks (A,B) when the first d nodes in S_e are moved from their source block A to their target block B. By construction, a node $v \in V$ can occur in at most one of the sequences created and in its sequence only once.

Generally speaking, the *advanced model* consists of τ layers. Essentially each layer is a copy of the quotient graph. An edge starting and ending in layer d of this model corresponds to the movement of exactly d nodes. The weight of an edge $e = (A,B)$ in layer d of the model is set to the negative value of the dth entry in g_e. In other words, it encodes the negative value of the gain, when the first d nodes in S_e are moved from block A to block B. Hence, a negative cycle whose nodes are all in layer d will move exactly d nodes between each of the respective block pairs contained in the cycle and results in an overall decrease in the edge cut. We add additional edges to the model such that it contains *more possibilities* in presence of underloaded blocks. To be more precise, in these cases we want to get rid of the restriction that each block sends and obtains the same number of nodes. To do so, we insert *forward* edges between all consecutive layers, i.e. block k in layer d is connected by an edge of weight zero to block k in layer $d+1$. These edges are not associated with node movements. Furthermore, we add *backward* edges as follows: for an edge (A,B) in layer d, we add an edge with the same weight between block A in layer d and block B in layer $d-\ell$ if block B can take ℓ nodes without becoming overloaded. The newly inserted edge is associated with the same node movements as the initial edge (A,B) within layer d. This way we encode movements in the model where a block can emit more nodes then it gets and vice versa without violating the balance constraint. Additionally, we connect each node in layer d back to s, if the associated block can take at least d nodes without becoming overloaded. Again this means that the model might contain cycles through s which stand for paths in the quotient graph being associated with movements that decrease the overall cut. An example advanced model is shown in Figure 6.2. Note that zero weight cycle diversification can be used as in the basic model.

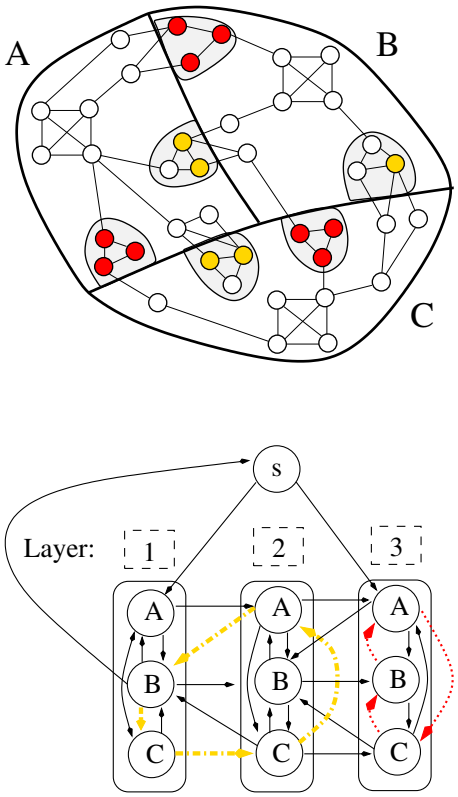


Figure 6.2: On top a graph that is partitioned into three parts ($|A| = 14, |B| = 12, |C| = 14$). Directed local searches on each directed pair of blocks are highlighted ($\tau = 3$). The corresponding advanced model is shown on the bottom. Each layer is a copy of the quotient graph of the partition. Edges within layer d represent node movements consisting of d nodes that have been found previously using directed local search. Node s is connected to all nodes (most of the edges are not shown), edges back to s are inserted if the corresponding block can take some nodes without becoming overloaded (in this example block B), backward edges between layers are inserted if the block can take nodes without becoming overloaded, forward edges between the layers are inserted in any case. Within layer 3, a negative cycle is highlighted (red/dark dashed) which corresponds to the movement of the nine red/dark nodes on top. Another negative cycle is highlighted in yellow/light grey dashed. It corresponds to the movement of the five yellow/light gray nodes on top. The weight of both cycles is -2. After these movements are performed the block weights are $|A| = 14, |B| = 13, |C| = 13$. Source: [147].

There is an interesting connection between the advanced algorithm and the helpful set algorithm by Diekmann et al. [53, 121]. Given a bipartition of a graph, a set is called ℓ-helpful if moving the nodes to the opposite block reduces the cut by ℓ. Recall that the helpful set algorithm tries to find an ℓ-helpful set in one block of the partition. Afterwards it attempts to find a balancing set that has the same cardinality as the found

ℓ-helpful set and is at least $-\ell+1$ helpful. If this attempt is successful, it performs the node movements and starts the next round. We call such node movements positive gain ℓ-exchange. Our advanced algorithm can be seen as an extension of this algorithm. More precisely, in the bipartition case our algorithms starts by finding a set of node movements in block A and afterwards in the opposite block B (very localized in both cases). If there *exists* a positive gain ℓ'-exchange, using subsets of the node movements performed, then our algorithm will find one and performs the corresponding node movements. Moreover, the advanced algorithm extends this idea to k-way partitioning and to the exchange of node sets that not necessarily have the same cardinality.

Multiple Directed Local Searches

The algorithm can be further improved by performing multiple directed local searches (MDLS) between each pair of blocks that share a non-empty boundary. More precisely, after we have computed node movements on *each* pair of blocks $e = (A,B)$, we start again using the nodes that are still eligible. This is done μ times. The model is then slightly modified in the following way: for the creation of edges in the model that correspond to the movement of d nodes from block A to block B, we use the directed local search on $e = (A,B)$ from the process above with the best gain when moving d nodes from block A to block B (and use this gain value for the computation of the weight of corresponding edges).

Conflicts

The advanced model can contain *conflicting* cycles that cannot be used. We now explain how we handle such cycles. There are two types of conflicting cycles that are due to the edges that run between the layers.

First, the model can contain cycles that do not correspond to a *simple* cycle in the quotient graph. Such a cycle is problematic because it contains the *same* edge $e = (A,B)$ in the quotient graph multiple times. An example is given in Figure 6.3. Let us assume that one associated edge runs in layer d and one in layer ℓ with $\ell < d$. The associated node movements cannot be performed fully since the edges correspond to subsets of the *same* directed local search. This is due to the fact that the edge in layer ℓ corresponds to the movement of the first ℓ nodes in S_e. These movements are a subset of the node movements associated with the edge in layer d, which corresponds to movement of the first d nodes in S_e. In other words, when we want to move the nodes associated with the edge in layer d then they are already in block B, if the node movements of the edge in layer ℓ have been performed before and vice versa depending on the order of execution. This means that for at least one of those two edges its weight does not correspond to the reduction in the cut of the underlying node movements. Hence, the weight of the cycle does not reflect the reduction in the number of edges cut.

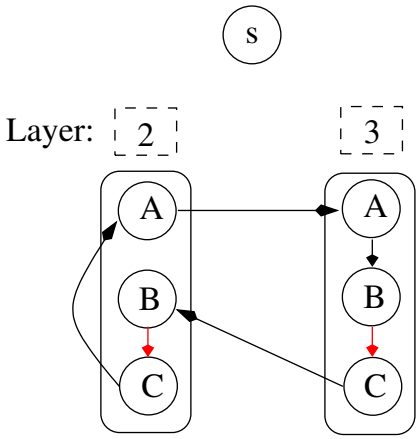


Figure 6.3: The first type of conflict that can occur in the advanced model. In this example two layers of the advanced model corresponding to graph in Figure 6.2 are shown. Only the edges of a conflicting cycle are drawn. The problem of the cycle are the highlighted edges running in layer two and three from the nodes representing block B to the nodes representing block C. They are associated with node movements where subsets are equal. The drawn cycle does not correspond to a simple cycle in the quotient graph. Source: [147].

Secondly, since we have both, edges between the layers *and* edges back to s, a cycle in the model can lead to node movements that *overload* a block. An example is given in Figure 6.4. A conflict can only occur if we have edges running between the layers. Our experiments indicate that conflicts do not occur very often. Furthermore, a conflict is easily detected. We can simply check if the cycle in the model is a simple cycle in the quotient graph or if one block would get overloaded when performing the node movements of that cycle. If our algorithm returns a cycle that contains a conflict, we *remove a random edge* of the cycle in the model and start the negative cycle detection strategy again. Note that if we remove all edges in the model that run between the layers, then the model is conflict-free but encodes less possible combinations of node movements.

6.1.3 Balancing

As we will see in Section 6.2, to create highly balanced partitions we start our algorithm with an ε-balanced partition, i.e. a partition where larger imbalance is allowed. Hence, to achieve perfect balance we have to think about balancing strategies. A balancing step will only be applied if the model does not contain a negative cycle (see next section for more details). We can modify the advanced model such that we can find a set of node movements that will decrease the total number of overloaded nodes by at least one and minimize the increase in the number of edges cut. Specifically, we introduce a

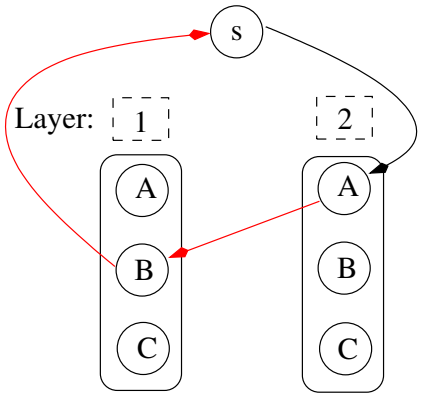


Figure 6.4: The second type of conflict that can occur in the advanced model. In this example two layers of the advanced model from Figure 6.2 are drawn. Only the edges of a conflicted cycle are shown. The edge in layer one from the node representing block B back to s was created because block B can take one node without becoming overloaded ($|B| = 12, |V|/k = 13$). For the same reason there is the edge between the layers from the node representing block A in layer two to the node representing block B in layer one. In the model there is no edge from block B in layer two back to s since block two can only take one node without becoming overloaded. However, when performing the associated node movements block B receives two nodes from block A and is overloaded afterwards. Source: [147].

second node t. Now instead of connecting s to all nodes, we connect it only to nodes representing overloaded blocks, i.e. $|V_i| > \lceil |V|/k \rceil$. Additionally, we connect a node in layer ℓ to t if the associated block can take at least ℓ nodes without becoming overloaded. Since the underlying model does not contain negative cycles, we can apply a *shortest path* algorithm to find a shortest path from s to t. We use a variant of the algorithm of Bellman and Ford since edge weights might still be negative (for more details see Section 6.4). It is now easy to see that a shortest path in this model yields a set of node movements with the smallest increase in number of cut edges and that the total number of overloaded nodes decreases by at least one. If τ is set to one, we call this algorithm basic balancing otherwise advanced balancing.

However, we have to make sure that there is at least one s-t path in the model. Let us assume for now that the graph is connected. If the graph is connected, then the directed version of the quotient graph is strongly connected. Hence an s-t path exists in the model if we are able to perform local search between *all* pairs of blocks that share a non-empty boundary. Because a directed local search can only start from an eligible node, we might not be able to perform directed local search between all adjacent pairs of blocks, e.g. if there is no eligible node between a pair of blocks left. We try to *ensure* that there is at least one s-t path in the model by doing the following. Roughly speaking, we try to integrate an s-t path into the model by changing the order in which directed local searches are performed. First, we perform a breadth-first search (BFS) in the quo-

tient graph which is initialized with all nodes that correspond to overloaded blocks in a random order. We then pick a random node in the quotient graph that corresponds to a block A that can take nodes without becoming overloaded. Using the BFS-forest we find a path $\mathscr{P} = B \to \ldots \to A$ from an overloaded block B to A. We now first perform directed local search on all consecutive pairs of blocks in $\mathscr{P}$. Here, we use $\tau = 1$ for the number of node movements to minimize the number of non-eligible nodes. If this was successful, i.e. we have been able to move one node between all directed pairs of blocks in that path, we perform directed local searches as before on *all* pairs of blocks that share a non-empty boundary. Otherwise, we undo the searches done (every node is eligible again) and start with the next random block that can take a node without becoming overloaded.

In some rare cases the algorithm fails to find such a path, i.e. each time we look at a path we have one directed pair of blocks where no eligible node is left. An example is shown in Figure 6.5. In this case, we apply a fallback balance routine that guarantees to reduce the total number of overloaded nodes by one if the input graph is connected. Given the BFS-forest of the quotient graph from above, we look at all paths in it from an overloaded block to a block that can take a node without becoming overloaded. At this point, there are at most order of k such paths in our BFS-forest. Concretely, for a path $\mathscr{P} = Z \to Y \to X \to \cdots \to A$ we select a node having maximum gain $g_{Z,Y}$ in Z and move it to Y. We then look at Y and do the same with respect to X and so on until we move a node to block A. Note that this time we can ensure to find nodes because after a node has been moved it is not blocked for later movements. After the operations have been performed they are undone and we continue with the next path. In the end we use the movements of the path that resulted in the smallest number of edges cut.

If the graph contains more than one connected component, then the algorithms de-

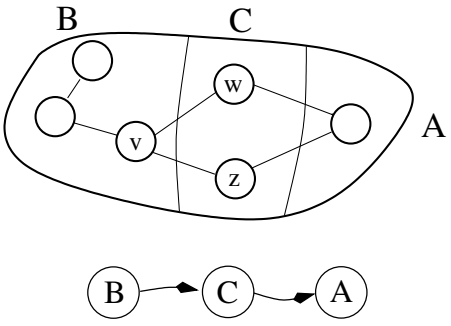


Figure 6.5: On top an example graph that is partitioned into three parts and on bottom a BFS-tree in the quotient graph starting in overloaded block B. It is not possible to integrate this path into the model since after directed local search is done on the pair (B,C), v will be marked and hence there is no eligible node left for the local search on the pair (C,A). A similar argument holds if local search is done on the pair (C,A) first. Source: [147].

scribed above may not work. For example if there is a non-perfectly balanced block in the input partition that is the union of some of the graphs connected components. More precisely, when we want to integrate a path into the model we detect at some point that there is no path in the quotient graph that contains this block and that can yield a balance improvement, e.g. if the block corresponds to a singleton in the quotient graph. To reduce the total number of overloaded nodes by one we do the following: if the block is overloaded, we move a random node from this block to an underloaded block; otherwise we move a random node from an overloaded block to this block. Note that the advanced balancing model can contain conflicts, too. This is again because of the edges that run between the layers. We handle potential conflicts in paths analogously to the conflicts in the advanced model case.

6.1.4 Putting Things Together

In practice we start our algorithms with an unbalanced input partition (see Section 6.2 for more details). We define two algorithms, basic and advanced, depending on the models used. Both, the basic and the advanced algorithm operate in rounds. In each round we iterate the negative cycle based local search algorithm until there are no negative cycles in the corresponding model (basic or advanced). After each negative cycle local search step we try to find zero weight cycles in the model to introduce some diversification. In Section 6.4.1 we also use a variant of the basic algorithm that does not use zero weight cycle diversification. Since we have random tie breaking at multiple places we iterate this part of the algorithm. If we do not succeed to find an improved cut using these two operations for λ iterations, we perform a single balancing step if the partition is still unbalanced, otherwise we stop. The parameter λ basically controls how fast the unbalanced input partition is transformed into a partition that satisfies the balance constraint. After the balancing operation, the total number of overloaded nodes is reduced by at least one depending on the balancing model. In the basic algorithm we use the basic balancing model ($\tau = 1$) and in the advanced algorithm we use the advanced balancing model. Since the balance operation can introduce new negative cycles in the model we start the next round.

6.2 Integration into KaFFPaE

We now describe how we integrate our new algorithms into our distributed evolutionary algorithm. Recall that KaFFPaE starts with a population of individuals (in our case partitions of the graph) and evolves the population into different populations over several rounds. In each round, the evolutionary algorithm uses a selection rule based on the *fitness* of the individuals (in our case the edge cut) of the population to select good individuals and *combine* them to obtain improved offspring.

It is well-known that allowing temporarily larger imbalance is useful to create good partitions [163, 156]. Hence, we adopt this idea. To obtain perfectly balanced partitions, we modify the create and combine operations as follows: each time we perform such an operation, we randomly choose an imbalance parameter $\varepsilon' \in [0.005, \hat{\varepsilon}]$ where $\hat{\varepsilon}$ is an upper bound for the allowed imbalance (a tuning parameter). This imbalance is then used to perform the operation, i.e. after the operation is performed, the offspring/partition has blocks with size at most $(1+\varepsilon')\lceil |V|/k \rceil$. Giving a larger imbalance to the operation yields smaller cuts and makes local search more effective since the combine and create operations use the multilevel graph partitioner KaFFPa. After the respective operation is performed, we apply our advanced balancing and advanced negative cycle local search (including zero weight cycle diversification and the packing approach) to obtain a partition of the graph that is *perfectly* balanced. This individual is the final offspring created by the performed create or combine operation and inserted into the population using the techniques of KaFFPaE. Note that *at all times* each individual in the population of the evolutionary algorithm is *perfectly balanced*. Also note that allowing larger imbalance enables us to use previously developed techniques that otherwise would not be applicable, e.g. max-flow min-cut based local search methods. We call the overall algorithm Karlsruhe Balanced Partitioner Evolutionary (KaBaPE). As experiments will show in Section 6.4 the new kind of local search is also helpful if some imbalance is allowed. When we use KaBaPE to create ε-balanced partitions we choose $\varepsilon' \in [\varepsilon + 0.005, \varepsilon + \hat{\varepsilon}]$ for the combine and create operations. The created individual is then transformed into a partition where each block has size at most $(1+\varepsilon)\lceil |V|/k \rceil$ using our balancing and negative cycle local search strategies.

6.3 Miscellanea

We also tried to integrate the negative cycle detection strategies into the multilevel scheme of KaFFPa. However, experiments did not indicate large improvements and furthermore the running time increased drastically. This is due to the fact that the size of the model of the negative cycle detection strategies depends heavily on the sum of the *weights* of the nodes moved (the number of layers in the model is the maximum of the sum of the weights moved between a pair of blocks during construction of the directed local searches). Also recall that a multilevel graph partitioning algorithm creates a sequence of smaller graphs, e.g. by computing matchings and contracting matched edges. This kind of compression is not helpful for our model in the strictly balanced local search scheme.

Additionally, we looked at an algorithm which iteratively uses random cycles in the quotient graph. In each round, a random cycle is found by performing a breadth-first search (BFS) in the quotient graph starting at a random node. We use a cycle that is created by adding a random non-tree edge to the BFS-tree. Let $C = a \to b \to \ldots \to a$ be that cycle. We then proceed in transactions. A transaction has the property that after

it is performed the weights of the blocks are the same as before. In each transaction *each* block in the cycle sends a node having maximum gain towards its successing block in the cycle. Note that if the input partition is perfectly balanced then the partition is still perfectly balanced after a transaction is performed. This is repeated until no improvement was found for several transactions and roll back complete transactions to the best partition found. We then proceed with the next random cycle or stop if no improvement was found for several rounds.

This greedy algorithm did not perform as well as the basic negative cycle detection strategies described above. Also it is not clear how this algorithm could be extended to graphs having node weights, or to more complex node movements in case of an imbalanced input partition. We therefore do not include further elaborations on this algorithm.

6.4 Experimental Evaluation

Implementation. We have implemented the algorithms described within this chapter using C++. We implemented negative cycle detection with subtree disassembly and distance updates as described in [39]. The implementation of the balanced local search algorithms has about 3 400 lines of code.

Parameters. After an extensive evaluation of the parameters, we fixed the number of multiple directed local searches to $\mu = 20$ (larger values of μ, e.g. iterating until no boundary node is eligible did not yield further improvements). The maximum number of node movements per directed local search is set to $\tau = 15$ for $k \leq 8$ and to $\tau = 7$ for $k > 8$ since this turned out to be a good tradeoff between quality and running time. The number of unsuccessful iterations until we perform a balancing step λ is set to three. When using KaBaPE to create perfectly balanced or ε-balanced partitions, we choose random values around the parameters above for each create or combine operation. To be more precise, each time we perform a create or combine operation, we pick a random number of node movements per directed local search $\tau \in [1,30]$, a random number of multiple directed local searches $\mu \in [1,20]$ and $\lambda \in [1,10]$ and use these parameters for the balancing and negative cycle detection strategies.

6.4.1 Walshaw Benchmark

As in the previous chapters, we apply our techniques to all graphs in Chris Walshaw's benchmark archive [157]. We start to look at the performance of the different algorithms if the previous record is used as an input and we compute new records from scratch.

Improving Existing Partitions

When we started to look at highly balanced partitioning, we counted the number of perfectly balanced partitions in the benchmark archive that contain nodes having positive gain, i.e. nodes that could reduce the cut when being moved to a different block. Astonishingly, we found that 55% of the perfectly balanced partitions in the archive contain nodes with positive gain (some of them have up to 1400 of such nodes). These nodes usually cannot be moved by simple local search due to the balance constraint. Therefore, we now use the existing perfectly balanced partitions from the benchmark archive and use them as input to our local search algorithms. This experiment has been performed on machine C and for all configurations of the algorithm we used $\lambda = 20$ for the number of unsuccessful tries. Table 6.6 shows the relative number of partitions that have been improved by different algorithm configurations and k (in total there are 34 graphs per number of blocks k).

It is somewhat surprising that already the most basic variant of the algorithm, i.e. negative cycle detection without the zero weight cycle diversification mechanism, can improve 47% of the existing entries. All of the algorithms have a tendency to improve more partitions when the number of blocks k increases. Less surprisingly, more advanced local searches and models increase this percentage further. When applying the advanced algorithm with multiple directed local searches enabled (the most expensive configuration of the algorithm), we are able to improve 128 partitions, i.e. 63% of the entries. Note that it took *overall* roughly two hours to compute these entries using one core of machine C. This is *very affordable* considering the fact that some of the previous approaches, such as Soper et al. [157], have taken many days to compute *one* entry to the benchmark tables. Of course in practice we want to find high quality partitions without using input partitions generated by other algorithms. We therefore compute partitions from scratch in the next section.

k	Basic	+ZeroGain	Advanced	+MDLS
2	0%	0%	0%	**0%**
4	18%	24%	41%	**44%**
8	38%	50%	64%	**74%**
16	64%	68%	71%	**79%**
32	76%	76%	88%	**91%**
64	82%	82%	79%	**88%**
sum	47%	50%	57%	**63%**

Table 6.6: Relative number of improved instances in the Walshaw Benchmark. Configurations: Basic (Most Basic Negative Cycle Improvement), +ZeroGain (As Before Plus Zero Weight Cycle Diversification), Advanced (Advanced Model, Directed Local Searches and Zero Weight Cycle Diversification), +MDLS (As Before Plus MDLS Enabled)

Computing Partitions from Scratch

We now compute perfectly balanced partitions from scratch, i.e. we do not use existing partitions as input to our algorithm. We use machine A and run KaBaPE with a time limit $t_k = 225 \cdot k$ seconds using 32 cores (four nodes of the cluster) per graph and $k > 2$. On the eight largest graphs of the archive, we gave KaBaPE a time limit of $\hat{t}_k = 4 \cdot t_k$ seconds per graph and $k > 2$. For $k = 2$, we gave KaBaPE one hour of time and 32 cores. $\hat{\varepsilon}$ was set to 4% for the small graphs and to 3% for the eight largest graphs in the archive. We summarize the results in Table 6.7 and report the complete list of results obtained in the technical report [145]. At submission time of [147], we have been able to improve or reproduce 86% of the entries reported in this benchmark. In the bipartition case we mostly reproduced the entries reported in the benchmark (instead of improving). This is not surprising since the models presented in this chapter can contain only trivial cycles of length two in this case and our previous algorithms have shown the same behaviour for larger imbalance values ([87, 122, 143, 144]). Also recently it has been shown by Delling et al. [52] that some of the perfectly balanced bipartitions reported there are optimal.

We also applied our algorithm for larger imbalances, i.e. 1%, 3% and 5%, in the Walshaw Benchmark. For the case $\varepsilon = 1\%$, we run our algorithm KaBaPE on all instances using the same parameters $\hat{\varepsilon}$ and t_k as in the perfectly balanced case. Doing so we have been able to improve or reproduce the cut in 160 out of 204 cases. Tables reporting detailed results can be found in [147]. Afterwards we performed additional partitioning trials on all instances where our systems (including [87], [122], [143], [144]) currently *not* have been able to reproduce or improve the entry reported there using different parameters and different machines. Doing so our systems improved or reproduced 98%, 99%, 99%, 99% of the entries reported there for the cases $\varepsilon = 0, 1\%, 3\%, 5\%$ respectively at submission time. These numbers include the entries where we used the current record as an input to our algorithms and improved this input partition. They contribute roughly 4%, 7%, 11%, 9% for the cases $\varepsilon = 0, 1\%, 3\%, 5\%$ respectively.

k	**2**	**4**	**8**	**16**	**32**	**64**	$\sum$
$<$	4	19	24	25	30	29	64%
$\leq$	29	31	27	27	31	30	86%

Table 6.7: Number of improvements (from scratch) for $\varepsilon = 0$.

6.4.2 Costs for Perfect Balance

It is hard to perform a meaningful comparison to other partitioners since publicly available tools such as Scotch [127], Jostle [165] and Metis [95] are either not able to take the desired balance as an input parameter or are not able to guarantee perfect balance. This is a major problem for the comparison with these tools since allowing larger imbalances, i.e. $\varepsilon = 3\%$, decreases the number of edges cut significantly [156]. However, we will show in Chapter 7 that KaFFPa produces better partitions compared to other partitioning packages, such as Scotch and Metis. Hence, we have a look at the number of edges cut by our algorithm when perfect balance is enforced, i.e. the increase in the number of edges cut when we seek a perfectly balanced partition. To do so, we use machine A and KaFFPaStrong to create partitions having an imbalance of $\varepsilon = 1\%$ and then create perfectly balanced partitions using our advanced negative cycle model and advanced balancing. For each instance (graph, k), we repeat the experiment ten times using different random seeds. We then compare the final cuts of the perfectly balanced partitions to the number of edges cut before the balancing and negative cycle search started, i.e. when $\varepsilon = 1\%$ imbalance is allowed. We use the same instances as in Chapter 4 for this experiment.

Table 6.8 summarizes the results of the experiment. On average the number of edges cut increased by roughly 5% when enforcing perfect balance and the running time of the negative cycle local search and balancing strategies is comparable with the average running time of KaFFPaStrong. Note that the running time of the algorithm increases with growing k. This is due to the fact that directed local searches are repeatedly performed between all adjacent pairs of blocks in each round of the algorithm. Moreover, the running time of the algorithm depends on the number of nodes that have to be moved from an overloaded block to an underloaded block. This number can grow if the number of blocks is increased.

k	**2**	**4**	**8**	**16**	**32**	**64**
Rel. Cut	9.0%	7.1%	4.7%	5.3%	3.6%	2.6%
Rel. Time t	12.0%	56.4%	98.8%	107.6%	133.6%	163.4%

Table 6.8: Costs for perfect balance, relative to KaFFPaStrong when $\varepsilon = 1\%$ imbalance is allowed. Rel. Cut reports the average increase in the cut after the 1% partitions have been balanced and Rel. Time reports the average time used by our local search algorithms relative to the running time of KaFFPaStrong.

6.5 Concluding Remarks

Review. In this chapter we presented novel algorithms to tackle the balanced graph partitioning problem, including the case of *perfect balance* when the maximal block size is bounded by the average block size. The techniques relax the balance constraint for node movements, but globally maintain balance by combining multiple local searches. This is done by reducing the combination problem to finding negative cycles in a graph, exploiting the existence of efficient algorithms for this problem. Experiments indicate that previous algorithms have not been able to find such rather complex movements. We also provide balancing variants of these techniques that are able to make infeasible partitions feasible. In contrast to previous algorithms such as Scotch [127], Jostle [165] and Metis [95], our algorithms are able to *guarantee* that the output partition is feasible. An integration into our parallel multilevel evolutionary algorithm has been able to improve or reproduce *most* of the entries reported at submission time in the Walshaw Benchmark using a reasonable amount of time.

Future Work. An open question is whether it is possible to define a *conflict-free* model that encodes the same kind of node movements as our advanced model. In future work, it could be interesting to see if one can integrate other types of local searches from KaFFPa, such as multi-try FM or max-flow min-cut based local search, into our models. The multiple directed local search algorithm can be improved such that it finds the best combination of the computed local searches. Currently, only boundary nodes can serve as candidates for movements. The neighborhood relation of local search could be increased if all nodes of a block would be eligible nodes for this kind of local search.

It will be interesting to see whether our techniques are useful for other problems where local search is restricted by constraints, e.g. hypergraph partitioning or multi-constraint graph partitioning. For example the proposed techniques might be useful in a setting where a graph needs to be repartitioned, e.g. the structure of the graph has been changed slightly so that the balance constraint is violated. Such problems arise in the area of adaptive partial differential equations where the mesh is adaptively refined in areas with large errors.

Shortly *after* we submitted our results to the benchmark archive, we lost entries to an implementation of [72] by Frank Schneider (the original work does not provide perfectly balanced partitions). However, we are still able to improve more than half of these entries when using those as input to KaBaR. Furthermore, we integrated the techniques of Galinier et al. [72] into our system, i.e. the proposed combine operator and tabu search, and again have been able to improve many entries. We conclude that the algorithms presented in this chapter are still very useful.

7

Comparison to Other Systems

In the previous chapters, we already compared ourselves to other graph partitioners by reporting the results achieved by our partitioners in the Walshaw benchmark (currently there are over forty reported submissions to this benchmark). In this chapter, we carefully compare our algorithms against other publicly available tools. This includes the packages DibaP [114], hMetis [97], KaSPar [122], kMetis [95] and Scotch [127]. We start by comparing the edge cut values produced by all of these tools on a large set of instances in Section 7.1. Since the running times of the various tools can be quite different, Section 7.2 compares the quality of solutions in a setting where algorithms get roughly the same amount of time to compute a partition. The results that KaFFPa and KaFFPaE achieved in the 10th DIMACS Implementation Challenge on Graph Clustering and Graph Partitioning are presented in Section 7.3.

References. The chapter is based on the conference papers [143, 144, 146, 147] that have been published together with Peter Sanders. However, in contrast to the results in the publications, we use our own partitioning algorithm for the experimental evaluation (except for the results that have been achieved during the 10th DIMACS Implementation Challenge). The description of the rules of the 10th DIMACS Implementation Challenge in Section 7.3 has been taken from [1]. We included them here for completeness.

7.1 Partitioning Packages

We now compare ourselves to other state-of-the-art graph partitioning libraries. To do so, we use the following twenty-two large graphs from our instances section: 144, 598a, PGPgiantcompo, af_shell10, as-22july06, asia, auto, delaunay20, deu, email-EuAll, europe, fe_ocean, fe_tooth, g3circuit, great-britain, htric00, loc-brightkite, nlr, p2p-gnu04, rgg20, slashdot0902 and wave. These graphs are random geometric graphs, delaunay graphs, graphs from numeric simulations, sparse matrices, road networks and social networks. Basic properties of these graphs can be found in Chapter 2.4. Experiments have been performed on machine B. For the comparisons we used DibaPFull 2.0.229,

Algorithm	large graphs		
	best	avg.	t[s]
KaFFPaStrong	7691	7894	102.74
KaSPar	+6%	+5%	92.27
hMetis	+9%	+8%	103.61
KaFFPaEco	+12%	+13%	5.72
Scotch	+20%	+23%	1.77
KaFFPaFast	+29%	+33%	0.94
kMetis	+35%	+48%	0.48

Table 7.1: Averaged quality and running times of the different partitioning algorithms (relative to KaFFPaStrong) on our large benchmark set which includes graphs from numeric simulations, sparse matrices, road networks and social networks. Comparisons with DibaP on graphs from numeric simulations can be found in Table 7.2.

DibaPLite 0.3.230, hMetis 2.0 (p1), KaSPar, kMetis 5.0 (p2), and Scotch 5.1.9. We run each of the partitioners ten times using different random seeds for initialization. We used the k-way variant of hMetis when computing partitions with hMetis. A fair comparison with hMetis is difficult, due to the fact that the partitions created by hMetis are very often imbalanced. We gave hMetis 1% of allowed imbalance to obtain partitions with up to 3% imbalance more often. Still many of the created partitions are not within a balance constraint of 3% imbalance and in some case the partitions produced hMetis have up to 12% imbalance. As recommended by Henning Meyerhenke, DibaPFull was run with 3 bubble repetitions, 10 FOS/L consolidations and 14 FOS/L iterations. The same parameters where used for DibaPLite. When computing partitions with KaSPar, we used a slightly stronger parameter $\alpha = 20$ since it improves the average cut and results in balanced partitions more often. In general, most of the partitioners (except KaFFPa) have sometimes trouble to compute feasible partitions, especially on social networks, which yields an advantage for our competitors.

For comparisons with other partitioners we have the following rules. In general, we excluded the case $k = 64$ for the european road network since hMetis runs out of memory for this case (it required more than 64 GB of memory). Since DibaP is built for graphs that have a mesh like structure and cannot handle graphs with more than one connected component, we focused on graphs from numeric simulations and sparse matrices for the comparison with DibaP. Hence, we restricted ourselves to use the following graphs in this case: 144, 598a, g3circuit, af_shell10, auto, delaunay_n20, fe_ocean, fe_tooth, htric00, nlr, wave. Table 7.1 summarizes the main results on all instances from our benchmark set. For the comparison with DibaP, we present the results on graphs from numeric simulations and sparse matrices in Table 7.2. Detailed per instance results can be found in the Appendix. On our large benchmark set, kMetis produces about 48% larger cuts than KaFFPaStrong. KaSPar, hMetis and Scotch produce 5%, 8% and 23% larger cuts than KaFFPaStrong respectively. The running time of hMetis is comparable

Algorithm	meshes		
	best	avg.	t[s]
KaFFPaStrong	15477	15726	91.02
Kaspar	+2%	+2%	76.92
KaFFPaEco	+5%	+6%	4.43
hMetis	+11%	+12%	136.89
Scotch	+12%	+13%	2.63
DibaPFull	+10%	+14%	4.46
DibaPLite	+14%	+15%	16.04
kMetis	+19%	+20%	0.46
KaFFPaFast	+17%	+21%	0.58

Table 7.2: Averaged quality and running times of the different partitioning algorithms (relative to KaFFPaStrong) on graphs from numeric simulations and sparse matrices.

to the running time of KaFFPaStrong. On graphs from numeric simulations hMetis is 50% slower than KaFFPaStrong and yields 12% worse cuts. On the other hand, KaFFPaEco is a factor thirty faster than hMetis and produces 6% smaller cuts on average on this benchmark set. Moreover, hMetis has trouble to return partitions that are feasible on many graphs which yields an advantage for hMetis. Even with this advantage, we conclude that hMetis is outperformed by KaFFPa on average. The running time of DibaPFull is comparable to the running time of KaFFPaEco. However, DibaPFull employs shared memory parallelism, i.e. when partitioning a graph into k blocks it uses k threads for partitioning (the machine used for the experiments had 16 cores). Moreover, the edge cut produced by DibaPFull is about 9% larger compared to KaFFPaEco. DibaPLite performs slightly worse than DibaPFull and, since it is not parallelized, needs about a factor 3.5 more time on average. Hence, on average DibaP is dominated by KaFFPaEco in terms of quality and running time.

Figure 7.3 compares solution quality of KaFFPaStrong for different graph classes using scatterplots. More precisely, we plot four graph classes: graphs from numeric simulations (this includes delaunay and random geometric graphs), social networks, matrices and road networks. It is worth noting that KaFFPaStrong outperforms hMetis, kMetis and Scotch for all values of k on *all* graphs from numeric simulations and matrices. KaSPar performs much better on these networks and on road networks computes sometimes better partitions than KaFFPaStrong, in particular for small values of k. On the other hand, kMetis and Scotch perform very poorly on road networks. The partitions produced by kMetis are sometimes up to a factor 5 worse than those computed by KaFFPaStrong and those created by Scotch are sometimes up to a factor 2 worse. Note that we introduce specialized techniques for road networks in Chapter 8 which improve the cut on such networks even more. The results of KaFFPaStrong on social networks, compared to kMetis and Scotch, are more or less similar to the results on graphs from numeric simulations and matrices. Compared to KaSPar, KaFFPaStrong has a clear ad-

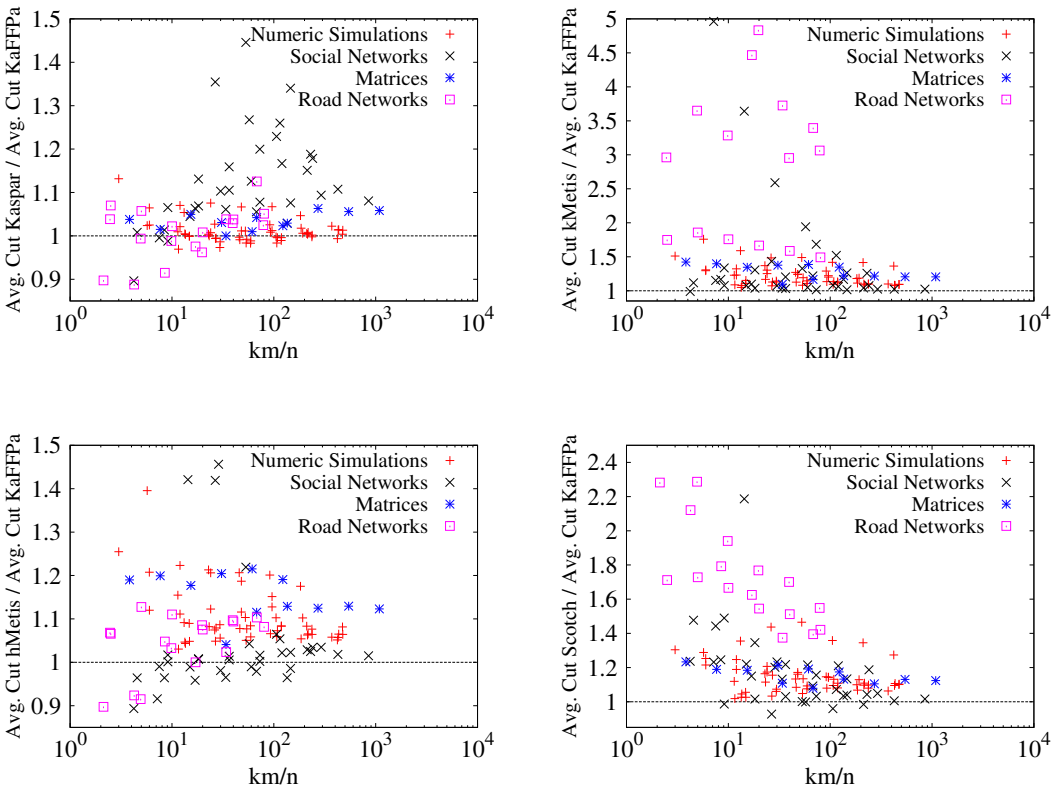


Figure 7.3: Scatterplots: average cuts of different algorithms (KaSPar, kMetis, hMetis, Scotch) for different graph classes relative to KaFFPaStrong.

vantage on these networks. The partitions of social networks computed by hMetis are sometimes better than those created by KaFFPaStrong.

We still have to argue about the quality vs. time tradeoff. One can see in Table 7.1 and Table 7.2 that the average and minimum cuts out of ten repetitions of the different algorithms are usually not too far apart. That means that it is usually not possible to use a simple method and repeat it several times to obtain cuts that are as good as those computed by KaFFPaStrong. For example, on our large benchmark set the average cuts of KaFFPaStrong are 3% smaller than the best cuts computed by KaSPar. Note that this yields a clear running time advantage (roughly a factor nine) for KaSPar. The same argument holds for KaFFPaEco against Scotch. Here, KaFFPaEco produces 3% smaller average cuts than the best cuts of Scotch. In the next section, we compare KaFFPa, Scotch and Metis against our evolutionary algorithms using an experimental setup in which all methods get the same amount of time to compute a partition.

7.2 Convergence Partitioning

In this section we compare KaFFPa, KaFFPaE, KaBaPE, Metis and Scotch in a setting where all algorithms get the same (fairly large) amount of time to compute a partition. We do this on the following subset of the graphs from our benchmark section: rgg17, rgg18, delaunay17, delaunay18, bel, nld, t60k, wing, fe_tooth, fe_rotor and memplus. Basic properties of these graphs can be found in Chapter 2.4. We perform two tests: first we compare KaFFPa, KaFFPaE, Scotch and Metis using $\varepsilon = 3\%$ imbalance as input to the partitioners. We then compare KaFFPa, KaFFPaE and KaBaPE using $\varepsilon = 0\%$ imbalance as input to our partitioners since KaBaPE was designed for small values of ε. Note that we do not include a comparison with Scotch and Metis for perfectly balanced case since the imbalance parameter is not configurable in those packages (3% imbalance is the default value in Scotch and Metis). Also note that, in contrast to KaBaPE, KaFFPa and KaFFPaE cannot ensure that the output partition is perfectly balanced which yields a slight advantage for those algorithms (larger imbalances yield smaller cuts). We use 16 cores of machine B (basically one node of the cluster) and two hours of time per instance when we use KaFFPaE and KaBaPE to create partitions. We parallelized repeated executions of KaFFPa, Metis and Scotch (embarrassingly parallel, different seeds) and also gave them 16 PEs and two hours of time to compute a partition. We look at $k \in \{2,4,8,16,32,64,128,256\}$ and performed three repetitions per instance.

To see how the solution quality of the different algorithms evolves over time, we use convergence plots which were introduced in Chapter 5.5. Figure 7.6 and Figure 7.7 show all convergence plots for the case $\varepsilon = 3\%$. Table 7.4 presents more data for this case. As expected the improvements of KaFFPaE relative to repeated executions of KaFFPa increase with growing k. The largest improvement is obtained for $k = 128$. Here, KaFFPaE produces partitions that have a 4.2% smaller cut value than plain restarts of KaFFPa. Moreover, the best partitions of Scotch and Metis are 13.9% and 19.0% larger on average compared to best partition of our evolutionary algorithm KaFFPaE.

k/Algo.	KaFFPaE. Avg.	KaFFPa Avg.	Scotch Avg.	Metis Avg.
2	568	+0.5%	+13.6%	+16.2%
4	1 214	+1.0%	+18.7%	+22.3%
8	2 165	+2.1%	+20.0%	+23.3%
16	3 483	+2.6%	+18.9%	+23.5%
32	5 308	+3.7%	+19.4%	+24.8%
64	7 892	+3.6%	+17.0%	+24.3%
128	11 472	+4.2%	+15.7%	+23.7%
256	16 653	+3.9%	+13.0%	+20.4%
overall	3 882	+2.7%	+13.9%	+19.0%

Table 7.4: Different algorithms after two hours of time on 16 PEs ($\varepsilon = 3\%$).

We have seen in the previous Section that the running times of single executions of Scotch and Metis are much faster than the running times of a single executions of KaFFPaStrong. On the other hand, KaFFPaStrong computes much better partitions. The results of our experiments performed in this section *emphasize* that one cannot simply take the best result out of multiple repetitions of a faster algorithm to achieve the same quality as KaFFPaStrong (or even KaFFPaE). We can strengthen the argument by comparing the average result of KaFFPa after one partitioner call to the best results of repeated executions of Metis and Scotch using 32 hours of sequential running time. Note that this yields a large running time advantage for Metis and Scotch. For example, the average result of KaFFPa after one partitioner call for the case $k = 64$ is 17.9% and 10.8% smaller than the best cuts after 32 hours of Metis and Scotch, respectively.

Figure 7.8 shows the convergence plots and Table 7.5 presents more data for the second experiment in the section, the perfectly balanced case. In this case, KaBaPE clearly outperforms KaFFPa and KaFFPaE for all values of k. For larger values of k, the quality gap between KaBaPE and KaFFPa or KaFFPaE increases with more time invested. On average the partitions produced by KaBaPE have 32.4% and 51.1% smaller cuts compared to KaFFPaE and KaFFPa. Note that KaFFPa and KaFFPaE have a slight advantage since they do not necessarily produce perfectly balanced partitions.

k/Algo.	KaBaPE. Avg.	KaFFPaE Avg.	KaFFPa Avg.
2	587	+46.9%	+56.4%
4	1 266	+45.5%	+70.6%
8	2 282	+46.5%	+71.1%
16	3 713	+33.6%	+55.7%
32	5 733	+32.0%	+50.3%
64	8 541	+21.7%	+40.0%
128	12 518	+18.5%	+36.4%
256	18 036	+14.7%	+28.2%
overall	4030	+32.4%	+51.1%

Table 7.5: Different algorithms after two hours of time on 16 PEs (perfectly balanced case, $\varepsilon = 0\%$).

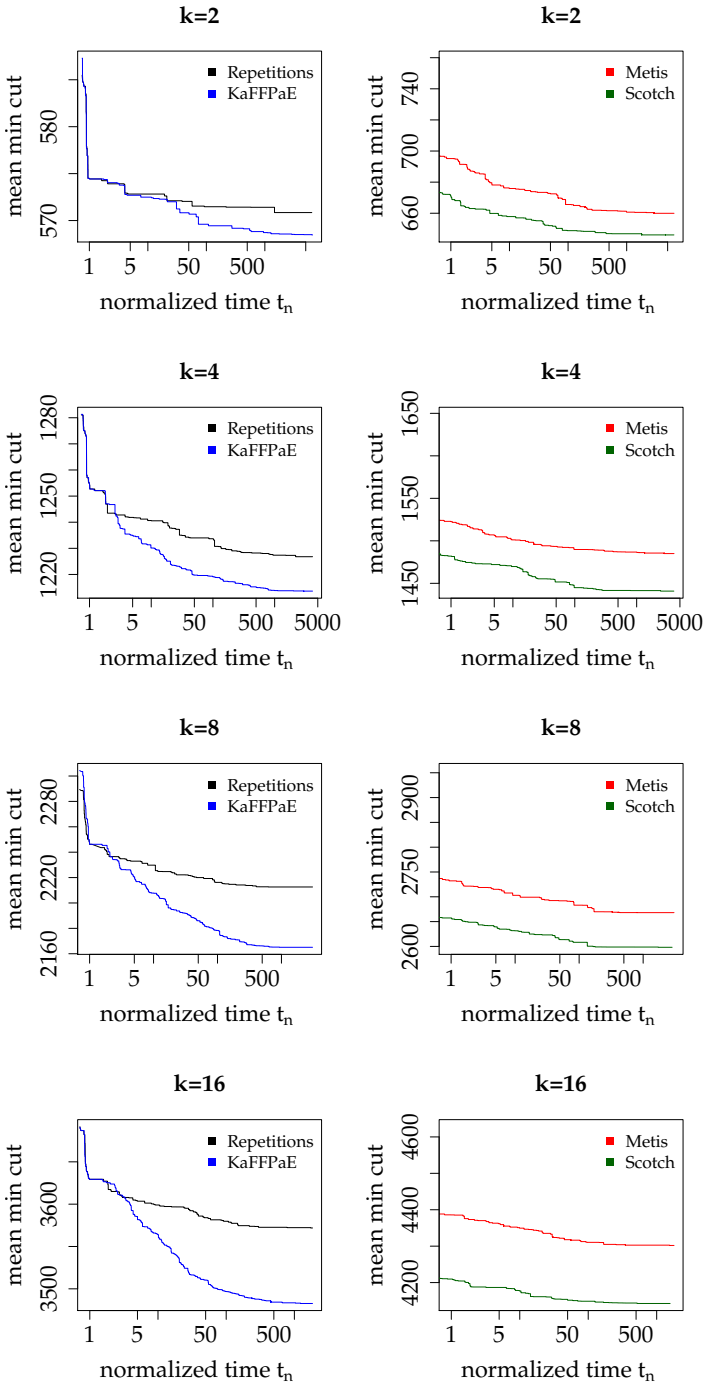


Figure 7.6: Convergence plots for the comparison of solution quality of KaFFPaE with repeated executions of KaFFPa, Scotch and Metis (over time) ($k \in \{2,\ldots,16\}$, $\varepsilon = 3\%$).

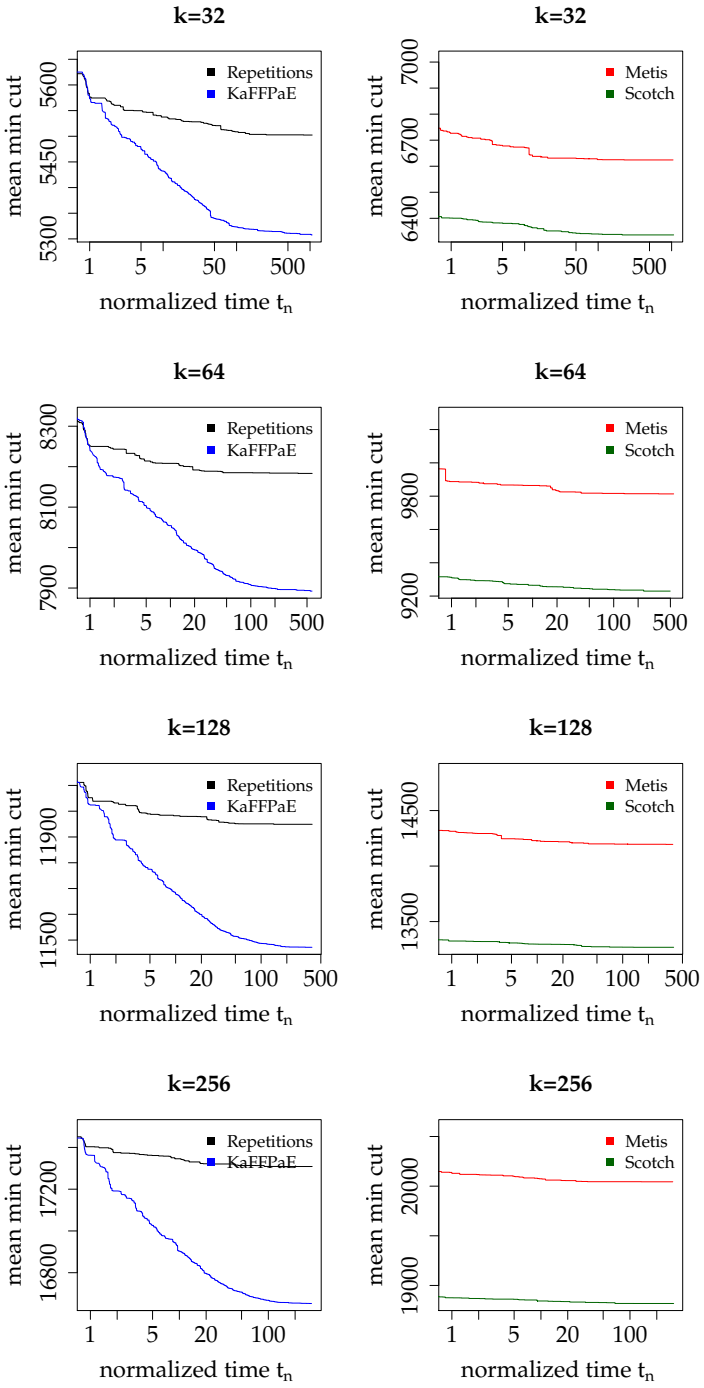


Figure 7.7: Convergence plots for the comparison of solution quality of KaFFPaE with repeated executions of KaFFPa, Scotch and Metis (over time) ($k \in \{32,\ldots,256\}$, $\varepsilon = 3\%$).

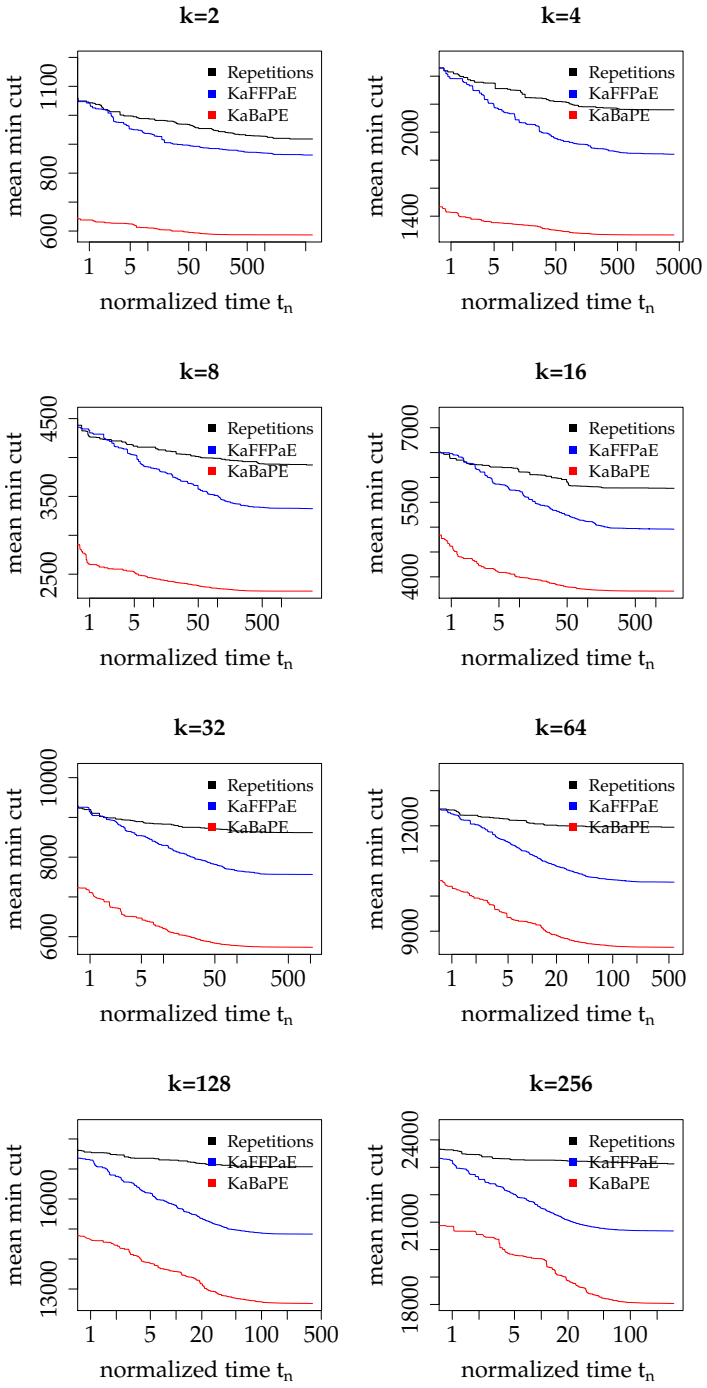


Figure 7.8: Convergence plots for the comparison of solution quality of KaBaPE with KaFFPaE and with repeated executions of KaFFPa (over time) ($k \in \{2,\ldots,256\}$, *perfectly balanced*: $\varepsilon = 0\%$).

7.3 10th DIMACS Implementation Challenge

There have been numerous DIMACS Implementation Challenges on interesting combinatorial problems such as the shortest path problem, the travelling salesman problem or nearest neighbor search. A DIMACS Challenge is usually focused on a particular problem with the goal to compare state-of-the-art algorithms and to estimate algorithm performance when a worst case analysis is not available or very pessimistic. The 10th DIMACS Implementation Challenge was on Graph Partitioning and Graph Clustering [16] – it has been co-organized by David Bader, Henning Meyerhenke, Peter Sanders and Dorothea Wagner. We also participated in the challenge using our graph partitioners KaFFPa and KaFFPaE. In this section we shortly outline the challenge rules and describe how we obtained the challenge results. Moreover, we evaluate the performance of kMetis 5.0 (p2) and Scotch 5.1.9 on these graphs in the Pareto subchallenge.

Challenge Rules. We now describe the challenge rules used in the 10th DIMACS Challenge on Graph Clustering and Graph Partitioning. We closely follow the description of [1]: there have been two challenges on graph partitioning, one Quality Challenge and one Pareto Challenge. The rationale of the Pareto Challenge is to take the work into account an algorithm requires to compute a solution. Hence, the two dimensions have been considered here: quality and work. Since not all submissions have been run on the same (or at least similar) hardware, a measure to account for different execution speeds has been used. In the challenge a graph-based benchmark was provided (based on breadth-first searches) to be executed to measure the system performance. Work is then normalized with respect to the machine performance, measured by this benchmark.

Within each challenge two objective functions have been taken into account, the edge cut (EC) and the maximum communication volume (CV). Both objective functions are described in Chapter 2. We now describe the scoring rules that have been used in the challenge. For each challenge instance result (EC and CV results are counted as one instance each), points are given to the best ranks based on the Formula 1 scoring rules used between 1991 and 2002. This means that the first six ranks receive a descending number of points (10, 6, 4, 3, 2, 1). The solver with the highest total number of points wins the Quality Challenge. The Formula 1 scoring scheme is also used in the Pareto Challenge. It is slightly different from the scoring scheme used in the Quality Challenge. Recall that the two dimensions considered here are quality and work. For each challenge instance result, each algorithm gets a Pareto dominance count, which expresses by how many other algorithms it has been Pareto-dominated; then algorithms are ranked by this number (lower count = better) and receive points according to the Formula 1 scoring scheme. Overall there have been 90 instances (graph, k) that had to be partitioned such that the edge cut metric is minimized and such that the communication volume metric is minimized. Hence, a complete submission contained 180 partitions. The instances are presented in Table 7.6.

Pareto Challenge. For this particular challenge we run all configurations of KaFFPa – KaFFPaStrong, KaFFPaEco, KaFFPaFast – and KaFFPaE, kMetis 5.0 (p2) and Scotch 5.1.9 on machine A. To compute a partition for an instance (graph, k), we repeatedly run the corresponding partitioner (except KaFFPaE) using different random seeds until the resulting partition is feasible. We stopped the process after one day of computation or after one hundred repetitions yielding unbalanced partitions. In this subchallenge the resulting partition was used for both parts of the challenge, i.e. optimizing for edge cut and optimizing for maximum communication volume. The running time of each iteration was added if more than one iteration was needed to obtain a feasible partition. KaFFPaE was given four nodes of machine A and a time limit of eight hours for each instance. When optimizing the maximum communication volume, we simply *altered* the fitness function to this objective. This ensures that individuals having a smaller maximum communication volume are more often selected for a combine operation and the individual with the smallest maximum communication volume in the population is the final re-

Graph	*Values of k*				
hugebubbles-00010	4	32	64	256	512
hugetric-00000	2	4	32	64	256
er-fact1.5-scale23	16	32	64	128	256
krong500-simple-logn17	2	4	8	16	32
krong500-simple-logn21	64	128	256	512	1024
delaunayn15	8	16	32	64	128
coAuthorsCiteseer	4	8	16	32	64
uk-2007-05	8	16	32	64	128
asia.osm	64	128	256	512	1024
great-britain.osm	32	64	128	256	1024
M6	2	8	32	128	256
NLR	8	32	128	256	512
AS365	64	128	256	512	1024
auto	64	128	256	512	1024
rggn218s0	8	16	32	64	128
G3circuit	2	4	32	64	256
kktpower	16	32	64	256	512
nlpkkt160	4	8	16	32	64

Table 7.6: The ninety instances that have been used in the Graph Partitioning subchallenges of the DIMACS Implementation Challenge on Graph Clustering and Graph Partitioning. The graphs are available at [16]. Each instance had to be partitioned to a) optimize the edge cut and b) minimize the maximum communication volume. Hence, a complete submission had 180 partitions. In both cases, the balance parameter ε was required to be 3%. Basic properties of these graphs can be found in Chapter 2.4.

sult of the evolutionary algorithm. Using this methodology KaFFPaStrong, KaFFPaEco, KaFFPaFast, KaFFPaE, Metis and Scotch were able to solve 136, 150, 170, 130, 146 and 110 instances respectively. The resulting points achieved in the Pareto challenge can be found in Table 7.7. Note that KaFFPaFast gained more points than KaFFPaEco, KaFFPaStrong and KaFFPaE. Since it is much faster than the other KaFFPa configurations it is almost never dominated by them and therefore scores a lot of points in this particular challenge. For some instances the partitions produced by Metis always exceeded the balance constraint by exactly one node. We assume that a small modification of Metis would increase the number of instances solved and most probably also the score achieved.

Solver	Points
KaFFPaFast	1372
Metis	1265
KaFFPaEco	1174
KaFFPaE	1134
KaFFPaStrong	1085
UMPa [36]	624
Scotch	361
Mondrian [61]	225

Solver	Points
KaFFPaFast	1680
KaFFPaEco	1305
KaFFPaE	1145
KaFFPaStrong	1106
UMPa [36]	782
Mondrian [61]	462

Table 7.7: Pareto challenge results including Metis and Scotch (left hand side) and original Pareto challenge results (right hand side).

Quality Challenge. Our quality submission KaPa (Karlsruhe Partitioners) assembles the best solutions of the partitions obtained of our partitioners in the Pareto challenge. On road networks we also run Buffoon, which is described in Chapter 8, to create partitions. The resulting points achieved in the quality challenge are reported in Table 7.8.

Solver	Points
KaPa	1574
UMPa [36]	1066
Mondrian [61]	616

Table 7.8: Original quality challenge results.

7.4 Concluding Remarks

Review. In this chapter, we compared our algorithms against other publicly available graph partitioning packages. We performed two kinds of experiments. We started by comparing the edge cut values produced by all of these tools on a large set of instances in Section 7.1. Here, we used a large subset of our benchmark graphs from Chapter 2.4, including graphs from numeric simulations, road networks, sparse matrices, random geometric graphs, delaunay graphs and social networks. We have shown that the partitions of KaFFPaStrong are on average superior in terms of quality compared to the competitors. Moreover, we argued that the partitions created by KaFFPa yield a good quality vs. time tradeoff. Our second experiment in Section 7.2, gave KaFFPa, KaFFPaE and KaBaPE as well as Metis and Scotch about 32 hours time (work) to compute a partition. We were able to show that the partitions produced by KaFFPaE are superior to repeated executions of KaFFPa. Additionally, KaFFPaE and KaFFPa compute much better partitions than Metis and Scotch. The best cuts of KaFFPaE are on average 13.9% smaller than the best partition of computed by Scotch and 19% smaller than the best partition computed by Metis. This emphasized that one cannot simply take the best result out of multiple repetitions of a faster algorithm to achieve the same quality as KaFFPaStrong or KaFFPaE. In the perfectly balanced case, KaBaPE clearly outperformed KaFFPaE and KaFFPa, although KaFFPaE and KaFFPa had a slight imbalance advantage since they cannot guarantee that the output partition is perfectly balanced.

The third section in this chapter presented the results of the graph partitioning subchallenges of the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering. Our partitioners, KaFFPa and KaFFPaE, *won* the graph partitioning subchallenges, i.e. they achieved the *best marks* among all participants in both graph partitioning subchallenges: the quality subchallenge in which partition quality was the main objective and the Pareto subchallenge in which the running time versus quality tradeoff is paramount. A surprising result was obtained for a part of the challenge where the objective function was not cut size but a measure of communication volume which can be expressed as a hypergraph partitioning problem. Interestingly, KaFFPaE outperformed dedicated hypergraph partitioners by just changing the fitness function to prefer solutions with low communication volume – the multilevel algorithm still optimized cuts.

8

Algorithmic Extensions

In this chapter, we evaluate two pre-processing techniques and one post-processing technique. The first pre-processing technique is tailored to road networks and presented in Section 8.1 and the second technique, explained in Section 8.2, is useful when it comes to the partitioning of large social networks or web graphs. Moreover, we evaluate a post-processing technique to obtain node separators from a given partition in Section 8.3.

References. The chapter is based on the conference papers [144, 146] that have been published together with Peter Sanders. Results on social networks are joint work with Peter Sanders and are unpublished. The results on node separators are also unpublished.

8.1 Partitioning Road Networks

In Chapter 5, we explained the notion of natural cuts which has been introduced by Delling et al. [49]. We used natural cuts to obtain another instantiation of our combine operator framework. In this section, we are focused on the partitioning of road networks. Hence, we adopt the technique of Delling et al. [49]. That is we use natural cuts as a pre-processing technique to obtain a clustering of the graph and build the contracted version of the graph as in the original work.

Recall, that the computation of natural cuts works in rounds. In each round we pick a center node v and grow a breadth-first search tree. The breadth-first search is stopped as soon as the weight of the tree, i.e. the sum of the node weights of the tree, reaches αU, for some parameters α and U. The set of the neighbors of T in $V \setminus T$ is called the *ring* of v. The *core* of v is the union of all nodes added to T before its size reached $\alpha U / f$ where $f > 1$ is another parameter.

The core is then temporarily contracted to a single node s and the ring into a single node t to compute the minimum s-t-cut between them using the given edge weights as capacities. To assure that every node eventually belongs to at least one core, and therefore is inside at least one cut, the nodes v are picked uniformly at random among all nodes that have not yet been part of any core in any round. The process is stopped when there

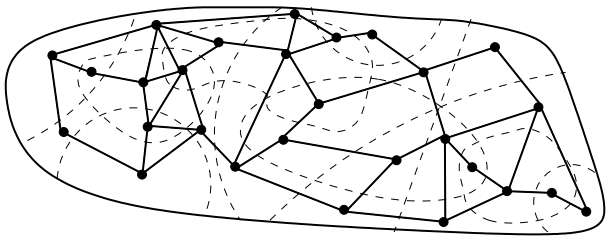


Figure 8.1: The coarse graph that is obtained from a natural cut clustering. A partition of the coarse graph corresponds to a partition of the original graph.

are no such nodes left. Note that the shared memory parallelization of this algorithm is quite simple – each flow problem can be constructed and solved independently.

Now, each connected component of the graph $G_C = (V, E \setminus C)$, where C is the union of all edges cut by the process above, induces a block of a clustering $\mathscr{C}$ of the graph. As in the original work by Delling et al. [49], we contract each block of the clustering into a single node. More precisely, the contracted graph is constructed as follows. First, each block of the clustering is contracted into a single node. The weight of the node is set to the number of nodes in the original block. There is an edge between two nodes in the contracted graph if there is connectivity between the corresponding blocks in the clustering. The weight of a particular edge (A,B) is set to the number of edges that run between block A and block B in the clustering.

It is easy to see that a partition of the coarse graph corresponds to a partition of the input graph. Hence, we use our (shared memory) parallelized version of natural cut pre-processing to obtain a coarse version of the graph and use KaFFPaE to partition the coarse graph. The contracted graph is often two orders of magnitude smaller than the input network and hence drastically speeds up the performance of the partitioning algorithm. Delling et al. [49] also only partition the contracted version of the graph. In contrast to the technique used by Delling et al. [49], KaFFPaE is not tailored to the partitioning of road networks. The overall algorithm is called *Buffoon*. Our pre-processing uses slightly different parameters than PUNCH (using the notation of Delling et al. [49], we use $\mathscr{C} = 2$, $U = (1+\varepsilon)\frac{n}{2k}$, $f = 10$, $\alpha = 1$).

8.1.1 Experiments

We apply Buffoon on two road networks (germany and europe) which are frequently used in the route planning community and compare ourselves to PUNCH [49]. We gave KaFFPaE $t_{\text{eur},k} = k \times 3.75$ min on europe and $t_{\text{ger},k} = k \times 0.9375$ min on germany, to compute a partition after pre-processing was done. In both cases, we used all 16 cores (hyperthreading active) of machine C for pre-processing and for KaFFPaE. When KaFFPaE used KaFFPa to create the initial population/partition, Scotch was employed as initial partitioning algorithm. The experiments were repeated ten times. A summary of the results is shown in Table 8.2. The results obtained by PUNCH are taken from [49].

grp, k	algorithm/running time t				
ger.	P_{best}	t_{total}	B_{avg}	t_{avg}	B_{best}
2	164	83	161	6	**161**
4	400	96	394	6	**393**
8	711	102	694	9	**693**
16	1 144	83	1 148	16	**1 137**
32	1 960	71	1 928	31	**1 898**
64	3 165	83	3 164	62	**3 143**
eur.	P_{best}	t_{total}	B_{avg}	t_{avg}	B_{best}
2	129	423	149	39	**129**
4	**309**	358	313	39	310
8	**634**	293	693	47	659
16	1 293	252	1 261	73	**1 238**
32	2 289	217	2 259	130	**2 240**
64	3 828	241	3 856	248	**3 825**

Table 8.2: Results on road networks: best results of PUNCH (P) out of 100 repetitions and total time [m] needed to compute these results; average and best cut results of Buffoon (B) as well as average running time [m] (including pre-processing).

Interestingly, on germany already our average values are smaller than or equal to the best result out of 100 repetitions obtained by PUNCH. Overall in 9 out of 12 cases we compute a best cut that is better or equal to the best cut obtained by PUNCH. Note that for obtaining the best cut values, we invest significantly more time than PUNCH. However, their machine is about a factor two faster (12 cores running at 3.33GHz compared to 8 cores running at 2.67GHz) and KaFFPaE is not tuned for road networks. We also run KaFFPa, KaSPar, Scotch and Metis on these networks. The number of cut edges are 9%, 12%, 93% and 288% larger on average respectively.

Figure 8.3 shows example partitionings into 64 blocks of the european road network created by Metis and Buffoon. In this case, the cut produced by Metis is roughly a factor 2.5 worse compared to the partition produced by Buffoon. Apparently, Metis is not able to find natural structures such as mountains or national borders. On the other hand, Buffoon (and PUNCH) can successfully find such natural structures, e.g. the border between germany and denmark or the Pyrenees. Moreover, in contrast to the result produced by Buffoon, the visual impression of the blocks created by Metis is already bad.

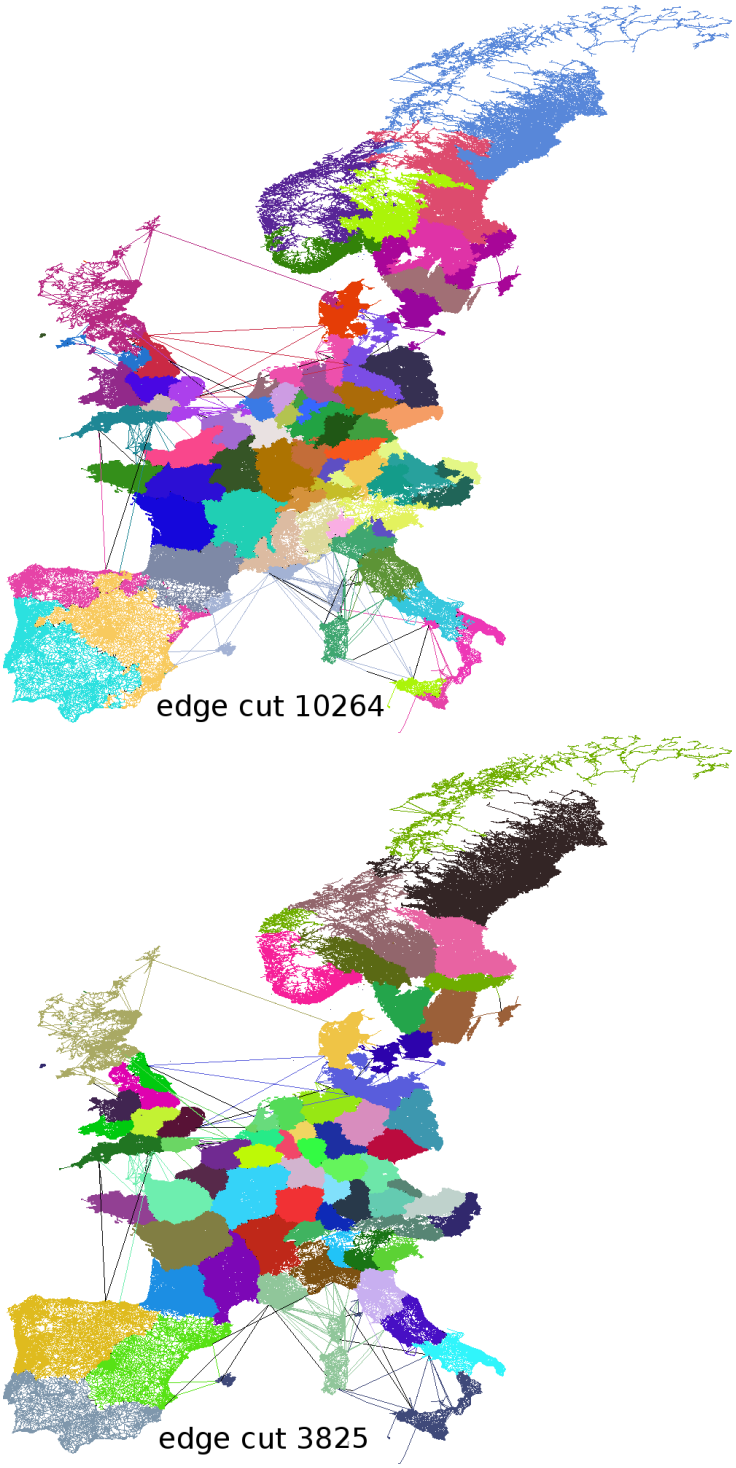


Figure 8.3: A partition of the road network of europe into 64 blocks created by Metis (top). A partition of the same graph into 64 blocks created by Buffoon (bottom).

8.2 Partitioning Large Social Networks

The second pre-processing technique that we look at, is tailored to the partitioning of large social networks or large web-graphs. The algorithm that we propose roughly works as follows: first we compute a clustering of the graph using a fast clustering algorithm, then we contract the obtained clustering such that a partitioning of the contracted graph corresponds to a partitioning of the input network. As we will see, the size of the contracted graph is much smaller than the size of the input graph. We then use KaFFPa to compute a partition of the contracted graph. This is followed by transfering the partition to the input graph and performing a refinement step which uses ideas from the clustering algorithm that was applied in the first step of the algorithm.

The intuition behind this technique is that a clustering of the graph, one hopes, contains many edges running inside the clusters and only a few edges running between the clusters. Thus the contracted graph will have good properties to be partitioned. Experiments in Section 8.2.2 indicate that the number edges per node of the contracted graph is smaller than the number of edges per node of the input network. On the other hand, the clustering algorithm that we use is very fast and the contracted graph is usually much smaller than the input network. This enables us to partition the largest publicly available web-graph with about 3 billion edges into 16 blocks using one core of a machine with only 64GB RAM in roughly *twenty-five* minutes.

8.2.1 Label Propagation with Size Constraints

The *label propagation clustering* algorithm was proposed by Raghavan et al. [135]. It is a very fast, near linear time, cut based graph clustering algorithm. We shortly outline the algorithm. Initially, each node is in its own cluster/block. The algorithm then works in rounds. In each round, the graph is traversed in a random order. When a node v is visited, it is *moved* to the cluster that most of its neighbors are in, i.e. it is moved to the cluster V_i that maximizes $|N(v) \cap V_i|$. This is similar to the concept of the gain of a node that is used within this work. Ties are broken randomly. The process is repeated until each node is in a block that most of its neighbors are in. Here, we perform ℓ iterations of the algorithm where ℓ is a tuning parameter. Note that one round of the algorithm can be implemented in $O(n+m)$ time.

In contrast to the original clustering algorithm [135], we have to ensure that each cluster fulfills a size-constraint. To see this, consider a clustering of the graph in which one cluster would have more than $(1+\varepsilon)\frac{|V|}{k}$ nodes. In this case, it would be impossible to find a partition of the contracted graph that fulfills the balance constraint. We ensure the cluster size constraint by introducing an upper bound $U \geq 1$ to size of the clusters. Clearly, when the algorithm starts this constraint is fulfilled since each of the clusters contains exactly one node. A cluster V_i is called underloaded if $|V_i| < U$. Now when we visit a node, we move it to an *underloaded* cluster that most of its neighbors are in. Hence, after moving a node, the size of each cluster is still smaller than or equal

to U. One round of the modified version of the algorithm can still be implemented in linear time by using an array of size $|V|$ to store the sizes of the clusters. Note that the algorithm without size constraint on the blocks can be easily parallelized (shared memory, distributed memory). Furthermore, there is a MapReduce implementation of this algorithm by Ovelgönne [124].

After we computed a clustering, we contract it and use our multilevel graph partitioner KaFFPa to partition the contracted graph. Contracting the clustering works as follows: each block of the clustering is contracted into a single node. The weight of the node is set to the number of nodes in the original block. There is an edge between two nodes in the contracted graph if there is connectivity between the corresponding blocks in the clustering. The weight of a particular edge (A,B) is set to the number of edges that run between block A and block B in the clustering. We are now ready to apply our partitioning algorithms to the contracted graph.

After the contracted graph is partitioned, we project its partition onto the input graph, i.e. each node of the input graph is assigned to the block of its coarse representative. To improve the partition after the projection is done, we apply r iterations of the label propagation algorithm with size constraints as local search algorithm (setting $U = (1+\varepsilon)\frac{|V|}{k}$).

8.2.2 Experiments

Our experiments in this section focus on the three web-graphs (which have up to 3 billion edges) and on the three largest social networks (with up to 16 million edges) from our benchmark set in Chapter 2.4. To save running time, our experiments focus on the

graph	t_P[s]	t_{LPSC}[s]	t_{CP}[s]	t_{PLS}[s]	t_{total}[s]	n/n'	m/n	m'/n'
coP.Cites.	41.2	1.5	4.2	1.0	**6.7**	*44.3*	36.9	4.6
webgoogle	26.7	0.8	2.8	0.3	**3.9**	*19.9*	5.9	4.5
coP.DBLP	92.1	2.1	12.0	1.2	**15.3**	*41.3*	28.2	7.8
as-skitter	960.5	1.5	13.7	0.6	**15.8**	*30.2*	10.4	4.7
uk-2002	2 453.7	24.4	117.4	13.4	**155.2**	*52.7*	14.1	4.4
uk-2007-05	*	217.2	1 123.9	119.6	**1 460.7**	*80.9*	31.2	2.7

Table 8.4: Running times on large social networks of our algorithms and average degree of the input graphs and its contracted versions. t_P is the running time needed by KaFFPa when applied on the input graph. t_{LPSC} refers to the running time of the label propagation algorithm with size constraints and contraction, t_{CP} refers to the time needed by KaFFPa to partition the contracted graph and t_{PLS} refers to the running time needed for projection and local search on the finest level, $t_{total} = t_{LPSC} + t_{CP} + t_{PLS}$. n is the number of nodes of the input graph and n' the number of nodes of the contracted graph (m and m' are defined similarly for the number of edges).

graph	KaFFPa avg.	LPSC+KaFFPa avg. initial cut	LPSC+KaFFPa avg. final cut
coP.Cites	1 131 172	1 083 546	**1 072 971**
webgoogle	35 056	32 181	**30 723**
coP.DBLP	2 113 903	1 884 510	**1 865 401**
as-skitter	1 301 532	1 353 862	**1 268 961**
uk-2002	4 402 424	2 383 411	**2 307 339**
uk-2007-05	*	5 858 770	**4 574 302**

Table 8.5: Average edge cut results on large social networks. KaFFPa avg. reports the average cut that KaFFPa produced when applied on the input graph, LPSC+KaFFPa avg. initial cut reports the average cut produced by KaFFPa on the contracted graph, LPSC+KaFFPa avg. final cut reports the final cut produced by the label propagation algorithm with KaFFPa after projection and local search.

Eco and Fast configurations of KaFFPa (using our own initial partitioning algorithm) and $k = 16$. More precisely, we used KaFFpaFast for all partitionings involving the two largest web-graphs and KaFFPaEco in all other cases. All experiments have been done using one core of one node of machine B (which has 64GB RAM). Experiments have been repeated ten times using different random seeds for initialization. After some experiments, we fixed the upper bound for this case to $U = |V|/4k$ and use four label propagation iterations in pre-processing as well as in post-processing.

Table 8.4 and Table 8.5 summarize the results. Interestingly, already the initial cut obtained by KaFFPa on the contracted graph is almost always better than the cut produced by KaFFPa when it is applied on the input graph. When comparing the final cut value obtained, then the new method always produces partitions that are better than those computed by pure KaFFPa. The improvements achieved by the new algorithm range from 2.5% on as-skitter to 90.8% on uk-2005. On the other hand, the new algorithm is much faster than KaFFPa itself. We achieve speed ups between 6 and 90 compared to KaFFPa being applied on the input network. There are more observations in Table 8.4 that are worth mentioning. For example, the contracted graph is on average a factor 45 smaller than the input network. This number can be increased if one increases the cluster upper bound U. However, increasing U can potentially result in imbalanced partitions. Moreover, the average degree of the contracted graph is always much smaller than the average degree of the input network. Figure 8.6 shows the degree distribution of uk-2002 before and after contraction.

KaFFPaFast has not been able to partition uk-2007 since it required too much memory (even on a different machine with 512GB RAM). Note that this is due to the fact that the matching base contraction cannot reduce the size of the graph significantly – in contrast to the label propagation pre-processing technique presented in this chapter. We also tried to partition uk-2007 using Metis on the machine with 512GB memory. However, Metis also crashes when trying to partition this graph.

8.3 Node Separators

The node separator problem asks to partition the node set of a graph into three sets A, B and S such that the removal of S disconnects A and B. A common way to obtain a node separator is the following. First, we compute a partition of the graph into two sets V_1 and V_2. Clearly, the boundary nodes in V_1 would yield a feasible separator and so would the boundary nodes in the opposite block V_2. Since we are interested in a small separator, we could simply use the smaller set of boundary nodes.

We can do better by using the method of Pothen et al. [132] which employs the set of cut edges of the partition. As in the original work, we use this method as a post-processing step to compute a node separator from a set of cut edges. The method computes the smallest node separator that can be found by using a subset of the boundary nodes. The main idea is to compute a subset S of the boundary nodes such that each cut edge is incident to at least one of the nodes in S. Such a set called a *vertex cover*. It is easy to see that S is a node separator since the removal of S eliminates all cut edges.

We are interested in a minimum vertex cover of the *bipartite* graph induced by the boundary nodes and the set of cut edges. A minimum vertex cover of a bipartite graph is obtained by solving a max-flow min-cut problem [5] that is similar to the max-flow min-cut problem used to find a maximum matching in a bipartite graph. For completeness, we shortly sketch the construction. First, a source s and a sink t are inserted. The source is connected to all boundary nodes in block V_1 and all boundary nodes in block V_2 are connected to the sink t. The capacity of these edges is set to one. The edges between $B[V_1]$ and $B[V_2]$ are directed towards $B[V_2]$ and their capacity is set to ∞. Now, a

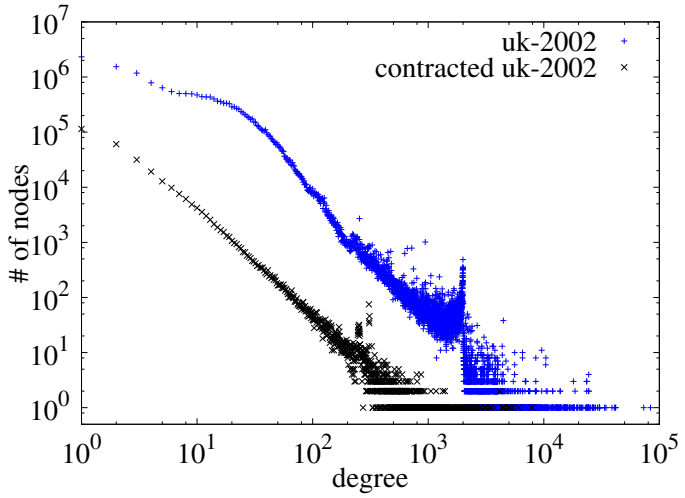


Figure 8.6: Unweighted degree distribution of uk-2002 before and after contraction.

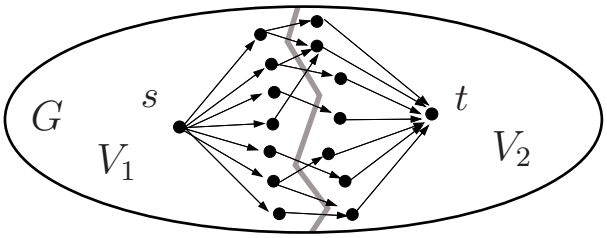


Figure 8.7: The flow problem to find a minimum vertex cover in the bipartite graph induced by the boundary nodes and the set of cut edges. This minimum vertex cover is the smallest separator that can be obtained from the set of cut edges. Cut edges have capacity ∞, all other capacities are set to one.

minimum *s*-*t* cut $(C, \bar{C})$ defines a minimum vertex cover $S = (B[V_1] \backslash C) \cup (B[V_2] \cap C)$ of the bipartite graph [5]. Figure 8.7 illustrates the construction and relates it to the original edge separator in the graph. It is worth mentioning that the method can also be used to obtain a k-way node separator, i.e. k blocks $V_1, \ldots, V_k$ and a set S such that after the removal of the nodes in S there no edge running between the blocks $V_1, \ldots, V_k$. This can be done computing a k-partition and then by applying the described flow problem between all pairs of blocks that share a non-empty boundary. All pair-wise separators together can then be used as a k-way separator. Indeed, also the simple algorithm, which takes the smaller boundary node set as a node separator, can be used in this framework to compute a k-way separator. However, the advanced method *always* computes a separator which has *less or an equal amount* of nodes compared to the separator produced by the simple method.

8.3.1 Experiments

We now compare the performance of the advanced flow-based algorithm to construct a node separator against the performance of the simple algorithm. The algorithms described above have been implemented using C++ within the KaFFPa framework. In this section, we don't compare ourselves against other software packages. This is due to the fact that none of the software packages used in Chapter 7 directly provides a method to output a node separator. Instead, the node separator methods are usually used internally to compute block orderings of sparse matrices which are represented as graphs. We use the same set of instances that we utilized in Chapter 7 to compare the performance of KaFFPa against other state-of-the-art graph partitioning software packages. The instances are 144, 598a, PGPgiantcompo, af_shell10, as-22july06, asia, auto, delaunay20, deu, email-EuAll, europe fe_ocean, fe_tooth, g3circuit, great-britain, htric00, loc-brightkite, nlr, p2p-gnu04, rgg20, slashdot0902 and wave. Basic properties of these graph can be found in Chapter 2.4. Experiments were performed using one core of one node of machine B.

In our experiments, we use KaFFPa to create partitions with at most 3% imbalance and then use both methods on this partition to create a k-way node separator. The experiments were repeated ten times. Table 8.8 summarizes the main results. The running time of the advanced algorithm to compute a separator is very small. It is roughly a factor twenty smaller than the running time of KaFFPaFast. On average the produced node separators by the advanced algorithm are 15% smaller than the separators computed by the simple method. The largest improvements are obtained on the graph as-22july06 which is an internet topology graph. Here, the advanced algorithm computes node separators that are more than a factor two smaller compared to the size of the separators of the simple method.

k	2	4	8	16	32	64	overall
KaFFPaFast	13.1%	15.6%	14.7%	14.1%	15.5%	16.1%	14.8%
KaFFPaEco	9.8%	13.3%	12.6%	14.4%	15.5%	16.8%	13.7%
KaFFPaStrong	15.4%	16.0%	16.6%	18.2%	18.3%	17.7%	17.0%

Table 8.8: Average improvements over the simple method to create a node separator for different values of k. Different configurations of KaFFPa where used to create partitions as starting point for both algorithms.

8.4 Concluding Remarks

Review. In this chapter, we looked at three algorithmic extension of our framework. The first extension is Buffoon, a specialized graph partitioner for road networks that relies on the notion of natural cuts by Delling et al. [49]. Using our shared memory parallel version of natural cuts, we obtain a clustering of the graph which is contracted. The contracted version is then partitioned by KaFFPaE. Experiments in this chapter show that the quality achieved by Buffoon on road networks is comparable to or better than PUNCH [49] and much better than the partitions created by Metis and Scotch.

The second algorithmic extension is *tailored* to the partitioning of large *social networks* or large *web-graphs*. These graphs usually have a very *irregularly structure* which makes it hard for a matching algorithm to compute "good" matchings that are used for contraction in a multilevel algorithm. By applying a very fast, cut based clustering algorithm with constraints on the cluster sizes, one can obtain a clustering which is internally dense and, one hopes, has not many edges between the clusters. As in the previous method, the clustering is contracted and the contracted graph is then partitioned using our graph partitioning framework. This pre-processing technique drastically reduces the size of the graph that needs to be partitioned and also reduces the number of edges per node. Hence, the running time of the partitioning algorithm is also reduced. This extension enabled us to partition the web graph of the uk in 2007 with roughly 3 billion edges using a single core of a machine with only 64GB RAM in roughly twenty-five minutes.

The last algorithmic extension presented in this chapter was a post-processing technique that can be used to obtain a node separator from a set of cut edges. Moreover, the method can be used to obtain k-way separators. We have seen that on average the size of the node separators is about 15% percent smaller compared to the simplest method that uses the smaller set of boundary nodes as a separator.

Future Work. The partitions produced by Buffoon are not necessarily connected. However, in some applications involving road networks this is a desirable property. Modifying the objective function of the evolutionary algorithm to $f(\mathscr{P}) = \text{cut}(\mathscr{P}) + |E|\mathbb{1}_{\text{blocks are not connected}}$ already helps to increase the cases where each block of output partition is connected, but cannot ensure this property. On the other hand, one can modify the partitioning problem such that k is not directly a part of the input in the sense that it is not required to output exactly k blocks, but one wants to find a clustering such that each cluster fulfills $|V_i| \leq (1+\varepsilon)\frac{|V|}{k}$. In this case, the output partition of Buffoon can be "reinterpreted" such that each connected component induced by a block becomes a block in the final partition.

It will be interesting to implement a parallel version of the social network preprocessing technique and to test the method on larger real-world graphs. Nowadays, a single computer can have 1TB of RAM and many cores so that the method can be used to obtain partitions of *huge* web-graphs in a very short amount of time. In general, the technique could be extended to become a general multilevel algorithm – by applying the clustering and contraction algorithm on each level of the hierarchy and in reverse, using a refinement algorithm on each level. Moreover, it could be valuable to employ the notion of algebraic distance as a measure of connectivity strength for the movements of nodes during the course of the label propagation algorithm and to employ the AMG-inspired coarsening, introduced in Chapter 4, for the partitioning of the contracted graph.

Nowadays, block orderings of sparse matrices which are represented by graphs are obtained by recursively computing node separators. In general, direct k-way methods yield superior graph partitioning quality compared to recursive bisection techniques. In this chapter, we proposed a method to obtain a k-way separator from a given k-way partition. It would be interesting to see whether a method using k-way separators could yield better node orderings. Moreover, the most balancing minimum cut heuristic presented in Chapter 4 could be used to derive a node separator with better balanced blocks.

One of the key features of the multilevel graph partitioning scheme is that in case of edge cut minimization the value of the objective function on a coarse graph is the same as on the finest graph. We are not aware of a coarsening scheme that has this property if the objective is to minimize the size of node separators.

9

Discussion

9.1 Conclusion

In this work we looked at the balanced graph partitioning problem. We focused mostly on the minimization of the edge cut metric with the constraint that the sum of the node weights in each block is smaller than or equal to $(1+\varepsilon)$ times the average block weight. Figure 9.1 gives an overview over the techniques that have been partly developed together with Manuel Holtgrewe, Ilya Safro and Peter Sanders. The parallel graph partitioner KaPPa [87] was developed together with Manuel Holtgrewe and Peter Sanders, partly during my diploma thesis [152], and is not a part of this work.

Throughout this work we looked at various local and global search techniques, different coarsening strategies as well as serveral meta-heuristics to tackle the graph partitoning problem. In many cases, the algorithms developed in advanced chapters of this work build upon the methods from earlier chapters. For example, we developed novel multilevel algorithms in Chapter 4 which provide new combine operations for our evolutionary algorithms in Chapter 5, and the techniques developed in Chapter 6, to tackle the highly balanced case, are integrated into this evolutionary framework.

Concretely, we started by carefully looking at the different components of the multilevel graph partitioning scheme in Chapter 4. KaFFPa, which is a framework developed in that chapter, is highly configurable to either achieve the best marks in the Walshaw Benchmark, to be a good trade-off between quality and running time, or to be the fastest system on some graphs while still improving partitioning quality compared to the previous fastest system. This has been achieved through several improvements of the multilevel algorithm that lead to enhanced partitioning quality. In particular, we looked at two coarsening schemes – matching and AMG-inspired – as well as the notion of algebraic distance for graph partitioning. On social networks the AMG-inspired coarsening scheme has a clear advantage, whereas both schemes produce similar results on graphs that are less unstructured (see Appendix A). Moreover, we looked at two novel local search techniques in that chapter – max-flow min-cut based local search and a very localized local search algorithm. Our experimental evaluation emphasized that max-flow min-cut based techniques produce superior partitions if the search space is expanded and

if they are combined with advanced techniques such as classical two-way local search algorithm and a heuristic that can find better balanced minimum cuts. Experimental results also suggest that localization of local search is highly helpful and that the global search techniques – V-, F- and W-cycle – are superior to repeated starts of the multilevel algorithm.

In the Walshaw benchmark, KaFFPa was beaten mostly for small graphs that combine the multilevel approach with an evolutionary strategy. Hence, we integrated our multilevel graph partitioner KaFFPa into an evolutionary strategy in Chapter 5. The coarsening phase was modified such that KaFFPa could provide effective combine and mutation operations. Intuitively, the combine operations assemble good parts of solutions into a single partition. The presented combine operation framework is very flexible so that a partition can be combined with an arbitrary domain specific graph clustering. Due to a scalable coarse grained parallelization, KaFFPaE is able to compute the best known partitions for many standard benchmark instances in only a *few minutes*. The time needed for partitioning the graph by the evolutionary algorithm is small compared to the execution time of a numeric simulation yielding the graph. Hence, we believe that KaFFPaE will still be helpful in the area of high performance computing and also that this framework could be of more general interest to the genetic algorithm community.

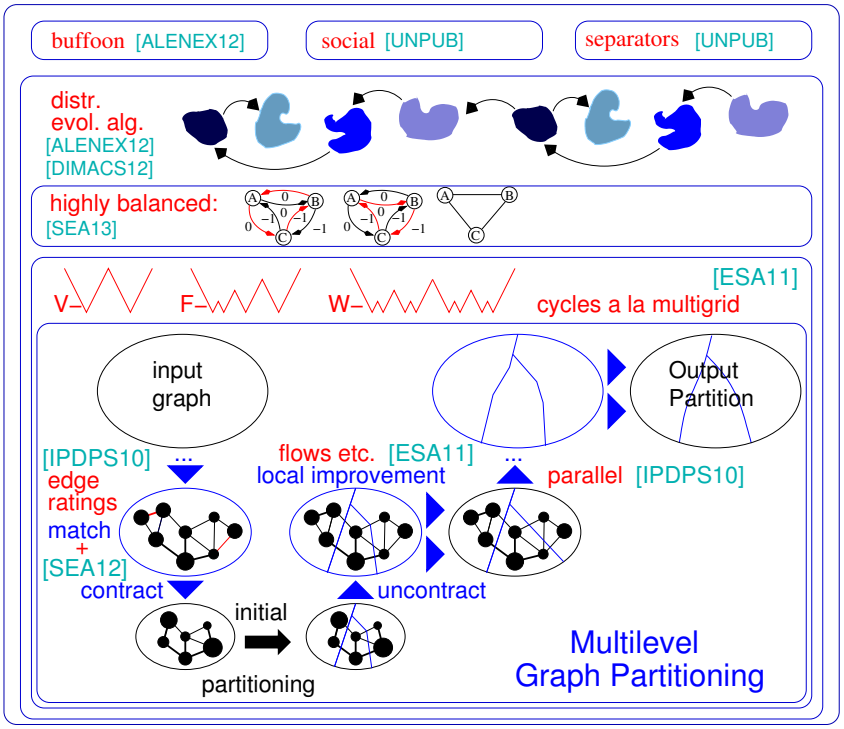


Figure 9.1: Overview of graph partitioning techniques. Source: [123].

KaFFPa and KaFFPaE compute partitions of very high quality when some imbalance $\varepsilon > 0$ is allowed. However, they are not very good for small values of ε, in particular for the perfectly balanced case. Hence, in Chapter 6 we developed new techniques for the graph partitioning problem with strict balance constraints, that work well for small values of ε including the perfectly balanced case. The techniques relax the balance constraint for node movements, but globally maintain balance by combining multiple local searches. This is done by reducing the combination problem to finding negative cycles in a directed graph, exploiting the existence of efficient algorithms for this problem. From a meta-heuristic point of view the proposed algorithms increase the neighborhood of a strictly balanced solution in which local search is able to find better solutions. Moreover, we provide efficient ways to explore this neighborhood. Experiments indicate that previous algorithms have not been able to find such rather complex movements. We also provided balancing variants of these techniques that are able to make infeasible partitions feasible. In contrast to previous algorithms such as Scotch [127], Jostle [165] and Metis [95], our algorithms are able to *guarantee* that the output partition is feasible.

Throughout this work, we extensively compared our algorithms to many other partitioners by submitting our partitions to Chris Walshaw's Benchmark archive and also performed direct comparisons with publicly available partitioning packages such as DibaP, hMetis, KaSPar, kMetis and Scotch. We have shown that the partitions of KaFFPaStrong are on average superior in terms of quality compared to the competitors. For example, the edge cuts created by KaFFPaStrong are on average 48% smaller than those computed by Metis. In an experiment in which partitioning packages had the same (fairly large) amount of time to create a partition, we could show that the partitions produced by our algorithms were much better than the best partitions computed by Metis and Scotch. This *emphasized* that one cannot simply take the best result out of multiple repetitions of a faster algorithm to achieve the same quality as KaFFPaStrong or KaFFPaE and hence the importance of the algorithms developed in this work.

Moreover, our partitioners KaFFPa and KaFFPaE, *won* the graph partitioning subchallenge of the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering. That means that our partitioners achieved the *best marks* among all participants in both graph partitioning subchallenges: the quality subchallenge in which partition quality was the main objective and the Pareto subchallenge in which the running time versus quality tradeoff is paramount. A surprising result was obtained for a part of the challenge where the objective function was not cut size, but a measure of communication volume which can be expressed as a hypergraph partitioning problem. Interestingly, KaFFPaE outperformed dedicated hypergraph partitioners by just changing the fitness function to prefer solutions with low communication volume – the multilevel algorithm still optimized cuts.

Another outcome of this work are algorithmic extensions to partition continental-sized road networks as well as large social networks and webgraphs, and an algorithm to derive a k-way separator from a given k-partition. Buffoon, is a specialized graph partitioner for road networks that relies on the notion of natural cuts which have been

introduced by Delling et al. [49]. Natural cuts are used as a pre-processing technique before KaFFPaE is applied. Experiments have shown that the quality achieved by Buffoon on road networks is comparable to or better than PUNCH [49]. The second algorithmic extension is tailored to the partitioning of large social networks or large web-graphs. It drastically reduces the size of the graph that needs to be partitioned by contracting a size constrained clustering of the input graph. The method improves partition quality for these graphs and drastically speeds up the computations. For example, the technique enabled us to partition the web graph of the uk in 2007 which has roughly 3 billion edges using a single core of a machine with only 64GB RAM in roughly twenty-five minutes.

The perspective taken in this work is that we developed our graph partitioners in a benchmark driven way achieving a system that has been able to improve or reproduce *most* of the entries reported in the Walshaw benchmark. Another equally valid perspective is that we applied the methodology of algorithm engineering to all aspects of the multilevel graph partitioning approach, achieving improvements in coarsening, local search, parallelization, global search guidance, and embedding into meta-heuristics as well as the strictly balanced case.

Although the problem is NP-hard and hard to approximate on general graphs, an astonishingly large set of "easier" graph algorithms are used to tackle the problem which makes the problem even more interesting. This includes problems such as weighted matching, breadth-first search, dominating sets, maximum flows, strongly connected components in Chapter 4, distributed evolutionary algorithms, maximum flows and randomized rumor spreading in Chapter 5 and problems such as shortest paths, negative and zero weight cycle detection in Chapter 6.

9.2 Outlook and Future Work

We summarize the future work discussed at the end of Chapters 4–8 and add some general ideas. In nearly every chapter of this thesis it would be very valuable to go back to parallelization. For example the integration of flow-based local search techniques as well as the global search techniques could be easily integrated into a distributed memory parallel partitioner such as KaPPa. The global search techniques presented in this chapter will even be more effective in a distributed parallel setting where each processor is responsible for one block of the partition. A step towards a faster distributed memory parallel partitioner has been done in the diploma thesis of Marcel Birn [25]. In his thesis, Marcel Birn developed a scalable and very fast, parallel matching algorithm. Another point of parallelization showed up in Chapter 5 where it would be interesting to use a parallel graph partitioner providing the combine operations (instead of the sequential KaFFPa). More precisely, each population of the evolutionary algorithm would be improved by using combine operations that would also be parallel. Since the number of available cores in a desktop increases drastically nowadays a shared memory parallelization of the proposed techniques is also highly desirable.

Partitioning a model of computation and communication is just one part of the story of high performance computing using modern systems. High performance clusters can have a very complex communication network with varying communication speeds between the different processing elements. Intuitively, one wants to embed the partition of the graph into this communication network such that the time spent for communication is minimized. In particular, one has to find a "good" mapping of the blocks of the partition to the processors of the system. Algorithms that tackle the problem often need perfectly balanced partitionings of the underlying communication network. The techniques presented in Chapter 6 could be very useful to achieve this. An integration of such techniques into our algorithms and perhaps interleaving methods that approximate the problem with the multilevel scheme of KaFFPa for partitioning would be valuable.

A part of the 10th DIMACS Challenge revealed that changing the fitness function of the evolutionary algorithm to communication volume, while the multilevel algorithm still optimizes cut, is already helpful to minimize the maximum communication volume. While this is a first step towards other objective functions, it remains to have specialized techniques. An open problem arises when the objective is to minimize the size of a node separator. One of the key features of the multilevel graph partitioning scheme is that, in case of edge cut minimization, the value of the objective function on a coarse graph is the same as on the finest graph. We are not aware of a coarsening scheme providing this property if the objective is to minimize the size of node separators. Connectedness of blocks is an important issue in some applications and currently not directly enforced by our algorithms.

The idea of using edge ratings during coarsening has been proposed in KaPPa [87] and has been extended in this work. However, it remains to have a well understood and unified rating function. A first step towards this direction has been done at our institute in the bachelor thesis of Maximilian Schuler [151].

Transferring ideas of this work to problems such as graph clustering, graph drawing, or to look at other objective functions will be interesting. In particular, localization of local search and global search techniques from Chapter 4 or the strictly balanced local search techniques from Chapter 6 if local search is limited by constraints, might be helpful. Global search techniques may help independently of the objective function if local searches assure nondecreasing quality. One example can be multilevel algorithms for graph clustering with the objective to optimize modularity. The strictly balanced local search techniques from Chapter 6 might be useful in a setting where a graph needs to be repartitioned, e.g. the structure of the graph has been changed slightly so that the balance constraint is violated. Such problems arise in the area of adaptive partial differential equations where the mesh is adaptively refined in areas with large errors. In his bachelor thesis, Florian Ziegler [168] already started to transfer some ideas presented in this work to the hypergraph bipartitioning problem.

The partitioning algorithms presented in this work or partitions created by them have already been used in multiple projects. They have been used to partition coarse models of communication and computation [69, 68], to speed up parallel multigrid methods

for Maxwell's equations [45, 113], in a parallel Lattice Boltzmann solver [3], for the partitioning of matrices arising in the Dantzig-Wolfe decomposition context [4] and to partition graphs from VLSI design [2]. Additionally, the algorithms have been used for alternative route planning [111] and in multiple student projects aiming to speed up Dijkstra's algorithm. Moreover, with this work the algorithms presented herein are released as an open source release.

References

Chapter 4 is based on the publications [143, 146] which were published together with Peter Sanders and the conference paper [139] which was published together with Ilya Safro and Peter Sanders. Results in Chapter 5 are based on the conference papers [144, 146] that have been published together with Peter Sanders. Chapter 6 stems from the conference paper [147] which has been published together with Peter Sanders. Chapter 7 is based on the conference papers [143, 144, 146, 147] that have been published together with Peter Sanders. However, most of the experiments presented in that chapter have been repeated using our own initial partitioning algorithm. These experimental results are unpublished. The description of the rules of the 10th DIMACS Implementation Challenge in Section 7.3 has been taken from [1]. We included them for completeness. The results on social networks, which are joint work with Peter Sanders, and the results on node separators in Chapter 8 have not been published before. Results on road networks are based on the conference papers [144, 146].

Bibliography

[1] Competition Rules and Objective Functions for the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering, `http://www.cc.gatech.edu/dimacs10/data/dimacs10-rules.pdf`.

[2] Personal Communication with D. Madina. 2011.

[3] Personal Communication with M. Wittmann. 2011.

[4] Personal Communication with M. Bergner and M. Lübbecke. 2012.

[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[6] E. Alba and M. Tomassini. Parallelism and Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.

[7] C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995.

[8] C. J. Alpert, A. B. Kahng, and S. Z. Yao. Spectral Partitioning with Multiple Eigenvectors. *Discrete Applied Mathematics*, 90(1):3–26, 1999.

[9] R. Andersen and K. J. Lang. An Algorithm for Improving Graph Partitions. In *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 651–660. Society for Industrial and Applied Mathematics, 2008.

[10] K. Andreev and H. Räcke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[11] M. Armbruster. *Branch-and-Cut for a Semidefinite Relaxation of Large-Scale Minimum Bisection Problems*. PhD thesis, 2007.

[12] M. Armbruster, M. Fügenschuh, C. Helmberg, and A. Martin. A Comparative Study of Linear and Semidefinite Branch-and-Cut Methods for Solving the Minimum Graph Bisection Problem. In *Proceedings of the 13th International Conference on Integer Programming and Combinatorial Optimization*, volume 5035 of *LNCS*, pages 112–124. Springer, 2008.

[13] S. Arora, E. Hazan, and S. Kale. $O(\sqrt{\log n})$ Approximation to Sparsest Cut in $\tilde{O}(n^2)$ Time. *SIAM Journal on Computing*, 39(5):1748–1771, 2010.

[14] S. Arora, S. Rao, and U. Vazirani. Expander Flows, Geometric Embeddings and Graph Partitioning. In *Proceedings of the 36th Annual ACM Symposium on the Theory of Computing*, pages 222–231. ACM Press, 2004.

[15] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. PhD thesis, 1996.

[16] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, `http://www.cc.gatech.edu/dimacs10/`.

[17] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. A Benchmark Set for Graph Clustering and Graph Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, to appear.

[18] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[19] U. Benlic and J. K. Hao. An Effective Multilevel Memetic Algorithm for Balanced Graph Partitioning. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence*, pages 121–128, 2010.

[20] U. Benlic and J. K. Hao. A Multilevel Memetic Approach for Improving Graph k-Partitions. *IEEE Transactions on Evolutionary Computation*, 15(5):624–642, 2011.

[21] U. Benlic and J. K. Hao. An Effective Multilevel Tabu Search Approach for Balanced Graph Partitioning. *Computers & Operations Research*, 38(7):1066–1075, 2011.

[22] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, 100(5):570–580, 1987.

[23] C. Bichot and P. Siarry, editors. *Graph Partitioning*. Wiley, 2011.

[24] C. E. Bichot. A New Method, the Fusion Fission, for the Relaxed k-Way Graph Partitioning Problem, and Comparisons with some Multilevel Algorithms. *Journal of Mathematical Modelling and Algorithms*, 6(3):319–344, 2007.

[25] M. Birn. Engineering Fast Parallel Matching Algorithms. Masters's Thesis, Karlsruhe Institute of Technology, 2012.

[26] K. D. Boese, A. B. Kahng, and S. Muddu. A New Adaptive Multi-Start Technique for Combinatorial Global Optimizations. *Operations Research Letters*, 16(2):101–113, 1994.

[27] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco. *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; Developer's Guide*. Sandia National Laboratories, 2007. Technical Report SAND2007-4749W, `http://www.cs.sandia.gov/Zoltan/dev_html/dev.html`.

[28] P. Bonsma. Most Balanced Minimum Cuts. *Discrete Applied Mathematics*, 158(4):261–276, 2010.

[29] R. B. Boppana. Eigenvalues and Graph Bisection: An Average-Case Analysis (Extended Abstract). In *Proceedings of the 28th Symposium on Foundations of Computer Science*, pages 280–285, 1987.

[30] A. Brandt. Multiscale Scientific Computation: Review 2001. In *Multiscale and Multiresolution Methods*, volume 20 of *LNCSE*, pages 3–95. Springer, 2002.

[31] W. L. Briggs and S. F. McCormick. *A Multigrid Tutorial*, volume 72. Society for Industrial Mathematics, 2000.

[32] L. Brunetta, M. Conforti, and G. Rinaldi. A Branch-and-Cut Algorithm for the Equicut Problem. *Mathematical Programming*, 78(2):243–263, 1997.

[33] T. Bui, S. Chaudhuri, F. Leighton, and M. Sipser. Graph Bisection Algorithms with Good Average Case Behavior. *Combinatorica*, 7:171–191, 1987.

[34] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992.

[35] Ü. V. Çatalyürek and C. Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In *Proceedings of the 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 1117, pages 75–86. Springer, 1996.

[36] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. UMPa: A Multi-objective, Multi-level Partitioner for Communication Minimization. In *10th DIMACS Impl. Challenge Workshop: Graph Partitioning and Graph Clustering*. Georgia Institute of Technology, Atlanta, GA, February 13-14 2012.

[37] P. Chardaire, M. Barake, and G. P. McKeown. A PROBE-Based Heuristic for Graph Partitioning. *IEEE Transactions on Computers*, 56(12):1707–1720, 2007.

[38] J. Chen and I. Safro. Algebraic Distance on Graphs. *SIAM Journal on Scientific Computing*, 33(6):3468–3490, 2011.

[39] B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. In *Proceedings of the 4th European Symposium on Algorithms*, volume 1136 of *LNCS*, pages 349–363, 1996.

[40] B. V. Cherkassky and A. V. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997.

[41] C. Chevalier and F. Pellegrini. Improvement of the Efficiency of Genetic Algorithms for Scalable Parallel Graph Partitioning in a Multi-level Framework. In *Proceedings of the 12th International Conference on Parallel Processing*, volume 4128 of *LNCS*, pages 243–252. Springer, 2006.

[42] C. Chevalier and F. Pellegrini. PT-Scotch: A Tool for Efficient Parallel Graph Ordering. *Parallel Computing*, 34(6):318–331, 2008.

[43] C. Chevalier and I. Safro. Comparison of Coarsening Schemes for Multilevel Graph Partitioning. In *Proceedings of the 3rd International Conference on Learning and Intelligent Optimization*, volume 5851 of *LNCS*, pages 191–205, 2009.

[44] F. Comellas and E. Sapena. A Multiagent Algorithm for Graph Partitioning. In *Proceedings of EvoWorkshops*, volume 3907 of *LNCS*, pages 279–285. Springer, 2006.

[45] Mauer D. and C. Wieners. Parallel Multigrid Methods for Maxwell's Equations. In *ASIL Status Report - Mai 13, 2011*.

[46] T. Davis. The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices`, 2008.

[47] K. A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.

[48] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms*, volume 6630 of *LCNS*, pages 376–387. Springer, 2011.

[49] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *Proceedings of the 25th International Parallel and Distributed Processing Symposium*, pages 1135–1146, 2011.

[50] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Exact Combinatorial Branch-and-Bound for Graph Bisection. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 30–44, 2012.

[51] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS State-of-the-Art Survey*, pages 117–139. Springer, 2009.

[52] D. Delling and R. F. Werneck. Better Bounds for Graph Bisection. In *Proceedings of the 20th European Symposium on Algorithms*, volume 7501 of *LNCS*, pages 407–418, 2012.

[53] R. Diekmann, B. Monien, and R. Preis. Using Helpful Sets to Improve Graph Bisections. *Interconnection Networks and Mapping and Scheduling Parallel Computations*, 21:57–73, 1995.

[54] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized Mesh Partitioning and Load Balancing for Parallel Adaptive FEM. *Parallel Computing*, 26(12):1555–1581, 2000.

[55] B. Doerr and M. Fouz. Asymptotically Optimal Randomized Rumor Spreading. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming, Proceedings, Part II*, volume 6756 of *LNCS*, pages 502–513. Springer, 2011.

[56] W. E. Donath and A. J. Hoffman. Algorithms for Partitioning of Graphs and Computer Logic Based on Eigenvectors of Connection Matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.

[57] W. E. Donath and A. J. Hoffman. Lower Bounds for the Partitioning of Graphs. *IBM Journal of Research and Development*, 17(5):420–425, 1973.

[58] D. Drake and S. Hougardy. A Simple Approximation Algorithm for the Weighted Matching Problem. *Information Processing Letters*, 85:211–213, 2003.

[59] S. Dutt. New Faster Kernighan-Lin-type Graph-Partitioning Algorithms. In *Proceedings of the 4th IEEE/ACM International Conference on Computer-Aided Design*, pages 370–377, 1993.

[60] G. Even, J. (S.) Naor, S. Rao, and B. Schieber. Fast Approximate Graph Partitioning Algorithms. *SIAM Journal on Computing*, 28(6):2187–2214, 1999.

[61] B. O. Fagginger Auer and R. H. Bisseling. Abusing a Hypergraph Partitioner for Unweighted Graph Partitioning. In *10th DIMACS Implementation Challenge Workshop: Graph Partitioning and Graph Clustering*. Georgia Institute of Technology, Atlanta, GA, February 13–14 2012.

[62] U. Feige and R. Krauthgamer. A Polylogarithmic Approximation of the Minimum Bisection. *SIAM Journal on Computing*, 31(4):1090–1118, 2002.

[63] A. Feldmann and P. Widmayer. An $O(n^4)$ Time Algorithm to Compute the Bisection Width of Solid Grid Graphs. In *Proceedings of the 19th European Conference on Algorithms*, volume 6942 of *LNCS*, pages 143–154. Springer, 2011.

[64] A. Felner. Finding Optimal Solutions to the Graph Partitioning Problem with Heuristic Search. *Annals of Mathematics and Artificial Intelligence*, 45:293–322, 2005.

[65] C. E. Ferreira, A. Martin, C. C. De Souza, R. Weismantel, and L. A. Wolsey. The Node Capacitated Graph Partitioning Problem: A Computational Study. *Mathematical Programming*, 81(2):229–256, 1998.

[66] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Conference on Design Automation*, pages 175–181, 1982.

[67] M. Fiedler. A Property of Eigenvectors of Nonnegative Symmetric Matrices and its Application to Graph Theory. *Czechoslovak Mathematical Journal*, 25(4):619–633, 1975.

[68] J. Fietz. Performance Optimization of Parallel LB Fluid Flow Simulations on Complex Geometries. Masters's Thesis, Karlsruhe Institute of Technology, 2011.

[69] J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In *Proceedings of Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 818–829. Springer, 2012.

[70] P. O. Fjällström. Algorithms for Graph Partitioning: A Survey. *Linköping Electronic Articles in Computer and Information Science*, 3(10), 1998.

[71] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[72] P. Galinier, Z. Boujbel, and M. C. Fernandes. An Efficient Memetic Algorithm for the Graph Partitioning Problem. *Annals of Operations Research*, 191(1):1–22, 2011.

[73] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Problems. In *Proceedings of the 6th ACM Symposium on Theory of Computing*, STOC '74, pages 47–63. ACM, 1974.

[74] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[75] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[76] J. R. Gilbert, G. L. Miller, and S. H. Teng. Geometric Mesh Partitioning: Implementation and Experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[77] F. Glover. Tabu Search — Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[78] F. Glover. Tabu Search — Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[79] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[80] O. Goldschmidt and D. S. Hochbaum. A Polynomial Algorithm for the k-Cut Problem for Fixed k. *Mathematics of Operations Research*, 19(1):24–37, 1994.

[81] W. W Hager, D. T. Phan, and H. Zhang. An Exact Algorithm for Graph Partitioning. *Mathematical Programming*, 137:531–556, 2013.

[82] M. T. Heath and P. Raghavan. A Cartesian Parallel Nested Dissection Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 16(1):235–253, 1995.

[83] B. Hendrickson. Chaco: Software for Partitioning Graphs. `http://www.cs.sandia.gov/~bahendr/chaco.html`.

[84] B. Hendrickson. Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes? In *Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *LNCS*, pages 218–225. Springer, 1998.

[85] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing'95*. ACM, 1995.

[86] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

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

[88] M. Holzrichter and S. Oliveira. A Graph Based Method for Generating the Fiedler Vector of Irregular Problems. In *Proceedings of the 11th Parallel and Distributed Processing Workshop*, volume 1586 of *LNCS*, pages 978–985. Springer, 1999.

[89] J. Hromkovič and B. Monien. The Bisection Problem for Graphs of Degree 4 (Configuring Transputer Systems). In *Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science*, pages 211–220. Springer, 1991.

[90] L. Hyafil and R. Rivest. Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems. Technical Report 33, IRIA – Laboratoire de Recherche en Informatique et Automatique, 1973.

[91] H. Inayoshi and B. Manderick. The Weighted Graph Bi-Partitioning Problem: A Look at GA Performance. *Parallel Problem Solving from Nature — PPSN III*, pages 617–625, 1994.

[92] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation. Part I, Graph Partitioning. *Operations Research*, 37(6):865–892, 1989.

[93] S. E. Karisch, F. Rendl, and J. Clausen. Solving Graph Bisection Problems with Semidefinite Programming. *INFORMS Journal on Computing*, 12(3):177–191, 2000.

[94] G. Karypis and V. Kumar. Parallel Multilevel *k*-way Partitioning Scheme for Irregular Graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing'96*, 1996.

[95] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[96] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal on Parallel and Distributed Compututing*, 48(1):96–129, 1998.

[97] G. Karypis and V. Kumar. Multilevel *k*-Way Hypergraph Partitioning. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 343–348. ACM, 1999.

[98] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(1):291–307, 1970.

[99] B.W. Kernighan. *Some Graph Partitioning Problems Related to Program Segmentation.* PhD thesis, Princeton, 1969.

[100] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. Distributed Time-Dependent Contraction Hierarchies. In *Proceedings of the 9th International Symposium on Experimental Algorithms*, volume 6049 of *LNCS*, pages 83–93. Springer, 2010.

[101] J. Kim, I. Hwang, Y. H. Kim, and B. R. Moon. Genetic Approaches for Graph Partitioning: A Survey. In *Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO'11)*, pages 473–480. ACM, 2011.

[102] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson Education, 2005.

[103] P. Korošec, J. Šilc, and B. Robič. Solving the Mesh-Partitioning Problem with an Ant-Colony Algorithm. *Parallel Computing*, 30(5):785–801, 2004.

[104] University of Milano Laboratory of Web Algorithms. Datasets, `http://law.dsi.unimi.it/datasets.php`.

[105] C. Lanczos. An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. *Journal of Research of the National Bureau of Standards*, 45(4):255–282, 1950.

[106] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.

[107] K. Lang and S. Rao. A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts. In *Proceedings of 10th International Integer Programming and Combinatorial Optimization Conference*, volume 3064 of *LNCS*, pages 383–400. Springer, 2004.

[108] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Proceedings of the Münster GI-Days*, 2004.

[109] J. Lescovec. Stanford Network Analysis Package (SNAP). `http://snap.stanford.edu/index.html`.

[110] S. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[111] D. Luxen and D. Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 260–270. Springer, 2012.

[112] J. Maue and P. Sanders. Engineering Algorithms for Approximate Weighted Matching. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *LNCS*, pages 242–255. Springer, 2007.

[113] D. Maurer. *Ein Hochskalierbarer Paralleler Direkter Löser für Finite Elemente Diskretisierungen*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2013.

[114] H. Meyerhenke. *Disturbed Diffusive Processes for Solving Partitioning Problems on Graphs*. PhD thesis, Universität Paderborn, 2008.

[115] H. Meyerhenke, B. Monien, and T. Sauerwald. A New Diffusion-Based Multilevel Algorithm for Computing Graph Partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009.

[116] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating Shape Optimizing Load Balancing for Parallel FEM Simulations by Algebraic Multigrid. In *Proceedings of 20th International Parallel and Distributed Processing Symposium*, 2006.

[117] H. Meyerhenke and S. Schamberger. Balancing Parallel Adaptive FEM Computations by Solving Systems of Linear Equations. In *Proceedings of Euro-Par 2005 Parallel Processing*, volume 3648 of *LNCS*, pages 209–219. Springer, 2005.

[118] B. L Miller and D. E Goldberg. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Evolutionary Computation*, 4(2):113–131, 1996.

[119] G. L. Miller, S. Teng, W. Thurston, and S. A. Vavasis. Automatic Mesh Partitioning. Technical report, 1992.

[120] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *Journal of Experimental Algorithmics (JEA)*, 11(2006), 2007.

[121] B. Monien and S. Schamberger. Graph Partitioning with the Party Library: Helpful-Sets in Practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 198–205, 2004.

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

[123] V. Osipov, P. Sanders, and C. Schulz. Engineering Graph Partitioning Algorithms. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276, pages 18–26. Springer, 2012.

[124] M. Ovelgönne. Distributed Community Detection in Web-Scale Networks. Technical report, 2012.

[125] B. N. Parlett. The Rayleigh Quotient Iteration and Some Generalizations for Nonnormal Matrices. *Mathematics of Computation*, 28(127):679–693, 1974.

[126] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, 1980.

[127] F. Pellegrini. Scotch Home Page. `http://www.labri.fr/pelegrin/scotch`.

[128] F. Pellegrini. A Parallelisable Multi-level Banded Diffusion Scheme for Computing Balanced Partitions with Smooth Boundaries. In *Proceedings of Euro-Par 2007 Parallel Processing*, volume 4641 of *LNCS*, pages 195–204. Springer, 2007.

[129] J. C. Picard and M. Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Mathematical Programming Studies*, 13:8–16, 1980.

[130] J. R. Pilkington and S. B. Baden. Partitioning with Spacefilling Curves. Technical report, 1994.

[131] J. R. Pilkington and S. B. Baden. Dynamic Partitioning of Non-Uniform Structured Workloads with Spacefilling Curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300, 1996.

[132] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[133] R. Preis. Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In *Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *LNCS*, pages 259–269. Springer, 1999.

[134] P. Raghavan. Line and Plane Separators. Technical report, 1993.

[135] 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.

[136] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasitiy, and S. Haridi. Ja-be-Ja: A Distributed Algorithm for Balanced Graph Partitioning. Technical report, 2012.

[137] E. Rolland, H. Pirkul, and F. Glover. Tabu Search for Graph Partitioning. *Annals of Operations Research*, 63(2):209–232, 1996.

[138] D. Ron, I. Safro, and A. Brandt. Relaxation-Based Coarsening and Multiscale Graph Organization. *Multiscale Modeling & Simulation*, 9(1):407–423, 2011.

[139] I. Safro, P. Sanders, and C. Schulz. Advanced Coarsening Schemes for Graph Partitioning. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 369–380. Springer, 2012.

[140] L. A. Sanchis. Multiple-Way Network Partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[141] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms (see ArXiv preprint arXiv:1012.0006v3). Technical report, Karlsruhe Institute of Technology, 2010.

[142] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning (see ArXiv preprint arXiv:1110.0477v1). Technical report, 2011.

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

[144] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 16–29, 2012.

[145] P. Sanders and C. Schulz. Think Locally, Act Globally: Perfectly Balanced Graph Partitioning (see ArXiv preprint arXiv:1210.0477). Technical report, 2012.

[146] P. Sanders and C. Schulz. High Quality Graph Partitioning. In *Proceedings of the 10th DIMACS Implementation Challenge - Graph Clustering and Graph Partitioning*, pages 1–17. AMS, 2013.

[147] P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'12)*, LNCS. Springer, 2013.

[148] S. Schamberger. On Partitioning FEM Graphs Using Diffusion. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[149] S. Schamberger and J.-M. Wierum. Partitioning Finite Element Meshes Using Space-Filling Curves. *Future Generation Computer Systems*, 21(5):759–766, 2005.

[150] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.

[151] M. Schuler. Engineering Edge Ratings and Matching Algorithms for Multilevel Graph Partitioning. Bachelor's Thesis, Karlsruhe Institute of Technology, 2011.

[152] C. Schulz. Scalable Parallel Refinement of Graph Partitions. Diploma Thesis, Universität Karlsruhe, 2009.

[153] M. Sellmann, N. Sensen, and L. Timajev. Multicommodity Flow Approximation used for Exact Graph Partitioning. In *Proceedings of the 11th European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 752–764. Springer, 2003.

[154] N. Sensen. Lower Bounds and Exact Algorithms for the Graph Partitioning Problem Using Multicommodity Flows. In *Proceedings of the 9th European Symposium on Algorithms*, volume 2161 of *LNCS*, pages 391–403. Springer, 2001.

[155] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2(2):135–148, 1991.

[156] H. D. Simon and S. H. Teng. How Good is Recursive Bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997.

[157] 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):225–241, 2004.

[158] R. V. Southwell. Stress-Calculation in Frameworks by the Method of "Systematic Relaxation of Constraints". *Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences*, 151(872):56–95, 1935.

[159] U. Trottenberg and A. Schuller. *Multigrid*. Academic Press, Inc., 2001.

[160] D. Wagner and F. Wagner. Between Min Cut and Graph Bisection. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, pages 744–750. Springer, 1993.

[161] C. Walshaw. Website of the Walshaw Benchmark. `http://staffweb.cms.gre.ac.uk/~wc06/partition/`.

[162] C. Walshaw. Multilevel Refinement for Combinatorial Optimisation Problems. *Annals of Operations Research*, 131(1):325–372, 2004.

[163] C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

[164] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.

[165] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007.

[166] C. Walshaw, M. Cross, and M. G. Everett. A Localized Algorithm for Optimizing Unstructured Mesh Partitions. *International Journal of High Performance Computing Applications*, 9(4):280–295, 1995.

[167] C. Walshaw, M. Cross, and M. G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997.

[168] F. Ziegler. *n*-Level Hypergraph Partitioning. Bachelor's Thesis, Karlsruhe Institute of Technology, 2012.

A

AMG-inspired Coarsening

A.1 Experimental Evaluation

The AMG coarsening was implemented separately based on the coarsening for linear ordering solvers from Safro et al. [138] and was used to create the multilevel hierarchies for KaFFPa [143]. The computational experiments have been performed with five configurations of KaFFPa (see Table A.1). We concentrate on two groups of algorithms configurations, fast versions (ECO, ECO-ALG, AMG-ECO) and strong versions (STRONG, AMG). Since the main goal of this section is to evaluate the performance of the AMG-inspired coarsening scheme, most of the comparisons will be AMG vs. respective non-AMG ratios (between corresponding averages over 10 trials for each configuration). All experiments are performed with fixed imbalance $\varepsilon = 3\%$.

ECO	KaFFPaEco configuration, Scotch as initial partitioner
STRONG	KaFFPaStrong configuration, Scotch as initial partitioner, no F-cycles
ECO-ALG	coarsening using GPA algorithm on each level, edge rating function $\text{ex\_alg}(e) := \text{expansion}^{*2}(e)/\rho_e$, local search as in KaFFPaEco
AMG-ECO	AMG-inspired coarsening, local search as in KaFFPaEco
AMG	AMG-inspired coarsening, local search as in STRONG

Table A.1: Description of the five configurations used for the experiments.

Walshaw's Partitioning Archive. In this section, we compare our algorithms using all 34 graphs of Walshaw's Benchmark archive. In contrast to the next section, the comparison of our methods has not demonstrated surprisingly new results. Overall, we observed that uncoarsening performance of fast versions (ECO, ECO-ALG, AMG-ECO) are more or less similar to each other and algebraic distance can improve the quality.

k	$\frac{\text{ECO}}{\text{ECO-ALG}}$	$\frac{\text{ECO-ALG}}{\text{AMG-ECO}}$	$\frac{\text{STRONG}}{\text{AMG}}$
2	1.03	1.03	1.01
4	1.05	1.02	1.00
8	1.02	1.02	1.00
16	1.02	1.01	1.00
32	1.01	1.02	1.00
64	1.00	1.01	1.00

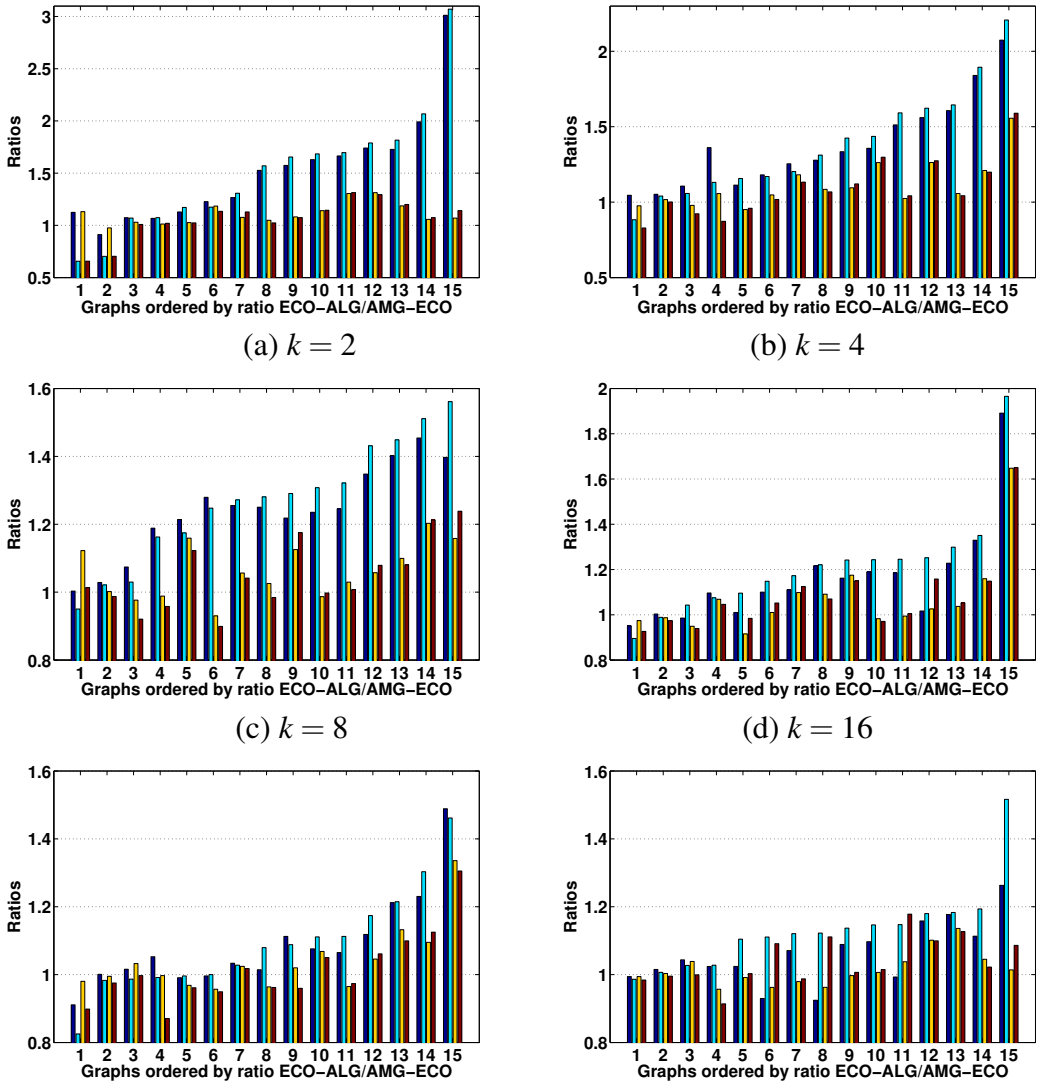
Table A.2: Computational comparison on graphs from the Walshaw Benchmark. Each number corresponds to the ratio of averages of final cuts for the pair of methods shown in the column title and the number of blocks k given in the row.

Scale-free Graphs. The node degree distribution in scale-free graphs follows asymptotically the power-law distribution. These types of networks often contain irregular parts and long-range links that can confuse both contraction and AMG-inspired coarsening. Since Walshaw's benchmark does not contain such graphs, we use the following instances for the comparison of the different algorithms: as-22july06, as-skitter, citCiteseer, coAutCiteseer, coAutDBLP, coPapDBLP, email-EuAll, loc-brightkite, loc-gowalla, p2p-gnu04, PGPgcomp, slashdot0902, web-google, wiki-talk, coPapCiteseer. The main properties of the graphs are summarized in Chapter 2.4. Because of the large running time of the strong configurations on these graphs, we compare only the fast versions of AMG-inspired and matching-based coarsenings.

The results of the comparison on scale-free graphs are presented in Figure A.3 and Table A.4. In Figure A.3 each plot corresponds to a different number of blocks k. The horizontal axes represent graphs from our test set. The vertical axes shows ratios representing average values for a pair of methods. Each graph corresponds to one quadruple of bars. The first, second, third and fourth bar represent averages of ratios ECO/AMG-ECO, ECO-ALG/AMG-ECO after local search on the finest level, ECO/AMG-ECO, ECO-ALG/AMG-ECO before local search on the finest level, respectively.

k	$\frac{\text{ECO}}{\text{ECO-ALG}}$ quality	$\frac{\text{ECO}}{\text{ECO-ALG}}$ full time	$\frac{\text{ECO}}{\text{ECO-ALG}}$ uncoarsening time	$\frac{\text{ECO-ALG}}{\text{AMG-ECO}}$ quality	$\frac{\text{ECO-ALG}}{\text{AMG-ECO}}$ uncoarsening time
2	1.38	0.77	1.62	1.16	3.62
4	1.24	1.32	1.85	1.11	2.14
8	1.15	1.29	1.45	1.07	1.94
16	1.09	1.27	1.33	1.06	1.69
32	1.06	1.18	1.23	1.00	1.60
64	1.06	1.13	1.13	1.01	2.99

Table A.4: Computational comparison for scale-free graphs.



(a) $k = 2$ (b) $k = 4$

(c) $k = 8$ (d) $k = 16$

(e) $k = 32$ (f) $k = 64$

Figure A.3: Comparison of coarsening schemes on scale-free graphs. Figures (a)-(f) contain results of comparison for $k = 2$, 4, 8, 16, 32, and 64, respectively. Each quadruple of bars correspond to one graph. First, second, third and fourth bars represent averages of ratios ECO/AMG-ECO, ECO-ALG/AMG-ECO after local search, ECO/AMG-ECO, and ECO-ALG/AMG-ECO before local search, respectively.

The role of algebraic distance. In this section, we emphasized the importance of the algebraic distance as a measure of connectivity strength. The price for improvements is the additional running time for the Jacobi over-relaxation to compute the distances. This can be implemented by using the most suitable (parallel) matrix-vector multiplication method. In cases of strong configurations and/or large irregular instances, the difference in the running time becomes less influential due to the amount of work that is spend in the uncoarsening phase.

Does AMG-inspired coarsening help? The positive answer to this question is given mostly by the results on scale-free graphs that contain relatively complex and irregular instances. This result is in contrast to graphs from the Walshaw Benchmark, in which we did not observe any particular class of graphs that corresponded to a difference in favor of one of the methods. We did not present exact comparison of coarsening running times because their underlying algorithm implementations are very different.

A.2 Concluding Remarks

Both matching and AMG-inspired coarsening schemes have been compared using fast and strong configurations of local search. As the main conclusion of this chapter, we emphasize the success of the proposed AMG-inspired coarsening and the algebraic distance connectivity measure between nodes demonstrated on *highly irregular* instances. One still has to take the trade-off between increased running time when using algebraic distance and improved quality of the partitions into account. The increased running time becomes less tangible with growth of graph size compared with the complexity of the uncoarsening phase.

Detailed per Instance Results

		KaFFPaStrong			KaFFPaEco			KaFFPaFast			hMetis		
graph	k	Best	Avg	t	Best	Avg	t	Best	Avg	t	Best	Avg	t
144	2	6 456	6 488	7.93	6 543	6 562	0.65	6 844	7 965	0.17	6 744	6 801	43.14
144	4	15 586	15 784	17.36	16 806	16 928	1.56	16 494	17 480	0.20	16 450	16 665	45.81
144	8	25 466	25 855	30.01	26 266	26 641	2.32	27 460	29 463	0.26	26 995	27 367	46.62
144	16	38 083	38 510	52.44	39 950	40 979	2.70	41 767	43 120	0.32	41 321	41 723	48.30
144	32	56 934	57 290	79.28	59 235	60 037	3.76	62 079	62 584	0.44	60 301	60 974	54.14
144	64	79 109	79 899	117.46	82 756	83 663	4.25	86 371	87 620	0.65	84 798	85 040	66.26
598a	2	2 367	2 370	4.75	2 383	2 387	0.35	2 475	2 593	0.13	2 459	2 481	25.58
598a	4	7 854	7 973	9.25	8 043	8 067	0.96	8 471	9 354	0.16	8 264	8 360	30.18
598a	8	15 909	16 254	15.46	16 389	16 964	1.30	17 390	19 075	0.21	16 882	17 077	32.19
598a	16	26 183	26 641	28.95	26 678	27 063	1.89	28 226	28 831	0.28	27 822	28 220	34.28
598a	32	39 287	40 017	46.76	40 230	41 490	2.55	43 422	43 742	0.39	42 286	42 533	39.34
598a	64	57 655	58 007	84.30	59 872	60 449	3.36	63 697	64 191	0.62	60 756	61 331	49.51
g3circuit	2	1 071	1 071	47.25	1 412	1 412	1.69	1 334	1 430	0.54	1 223	1 274	260.70
g3circuit	4	2 483	2 651	87.22	2 717	2 776	3.48	2 894	3 035	0.55	3 109	3 179	279.42
g3circuit	8	4 685	4 787	119.52	4 737	4 887	7.84	5 545	5 740	0.58	5 443	5 634	284.56
g3circuit	16	7 690	7 868	180.48	7 945	8 188	5.75	9 337	9 845	0.63	9 246	9 475	288.13
g3circuit	32	11 635	11 779	214.81	12 687	13 194	7.67	14 085	14 786	0.71	14 028	14 314	292.46
g3circuit	64	18 408	18 698	248.59	19 998	20 373	9.28	22 445	23 125	0.85	21 637	22 262	307.32
PGPgcomp	2	362	382	0.90	565	612	0.12	497	536	0.04	357	368	0.43
PGPgcomp	4	646	670	1.29	831	889	0.19	881	1 001	0.05	661	681	0.59
PGPgcomp	8	995	1 024	2.35	1 182	1 305	0.24	1 377	1 591	0.07	1 018	1 032	1.02
PGPgcomp	16	1 507	1 560	5.04	1 610	1 699	0.40	1 912	2 191	0.08	1 541	1 568	1.84
PGPgcomp	32	2 091	2 143	8.03	2 313	2 391	0.58	2 626	2 753	0.12	2 122	2 148	4.01
PGPgcomp	64	2 823	2 863	10.74	3 090	3 207	0.66	3 363	3 495	0.16	2 811	2 822	6.08
af_shell10	2	26 225	26 225	206.07	28 700	28 700	4.04	30 500	30 860	1.28	26 425	27 295	459.99
af_shell10	4	53 075	53 207	560.90	54 175	54 512	20.60	57 300	57 600	1.32	58 725	59 342	588.56
af_shell10	8	94 150	97 640	922.59	100 875	111 697	26.74	114 500	119 004	1.32	108 550	110 247	612.76
af_shell10	16	153 650	156 167	1 100.65	163 725	165 822	21.20	184 300	187 535	1.38	173 400	175 655	648.02
af_shell10	32	235 075	241 002	1 137.77	255 500	258 745	29.43	284 050	289 532	1.46	269 650	272 200	652.43
af_shell10	64	349 275	357 653	1 044.50	381 575	385 817	25.47	426 675	432 763	1.60	397 000	401 635	672.15
as-22july06	2	3 722	3 938	72.98	5 177	5 378	5.90	5 365	5 653	1.43	3 517	3 518	3.60
as-22july06	4	7 557	7 730	65.83	9 491	9 833	11.69	9 629	10 579	2.71	7 413	7 454	5.94
as-22july06	8	11 031	11 285	66.84	13 131	14 068	12.22	13 832	14 463	2.73	10 759	10 816	9.63
as-22july06	16	13 961	14 165	124.61	16 437	17 090	13.63	17 555	18 382	3.01	13 584	13 673	14.82
as-22july06	32	16 505	16 857	254.91	19 644	20 200	14.61	19 983	20 936	2.97	16 433	16 506	21.36
as-22july06	64	20 061	20 420	262.23	22 693	23 458	11.63	23 534	24 480	3.08	19 606	19 694	27.77
asia	2	7	7	111.01	14	15	8.17	19	21	4.26	7	7	791.77
asia	4	19	22	255.25	22	33	12.82	31	40	4.28	19	20	830.16
asia	8	50	56	287.79	62	67	11.74	67	87	4.25	56	59	902.43
asia	16	124	134	306.83	149	162	13.30	191	206	4.27	131	134	981.14
asia	32	284	292	350.22	326	341	15.86	396	419	4.24	288	299	1 000.93
asia	64	521	533	324.81	581	631	18.60	692	725	4.41	572	588	1 085.00
auto	2	9 682	9 768	37.71	9 985	10 012	2.35	10 556	17 155	0.59	10 429	10 643	201.27
auto	4	25 937	26 814	79.86	26 349	26 570	7.08	28 817	34 607	0.65	28 654	29 137	215.55
auto	8	44 806	46 064	119.62	47 179	48 873	9.72	51 354	55 386	0.73	49 243	49 934	220.48
auto	16	76 348	77 421	193.85	78 530	79 207	13.49	83 611	87 513	0.89	83 145	83 896	222.09
auto	32	120 291	121 999	283.34	125 989	126 944	12.87	133 231	135 454	1.15	130 084	131 321	228.39
auto	64	172 247	173 788	375.68	181 231	182 690	13.16	189 865	191 579	1.50	186 656	187 952	242.99
delaunay20	2	1 680	1 686	31.50	1 692	1 727	1.68	2 023	2 038	0.57	1 832	1 888	185.62
delaunay20	4	3 346	3 376	66.78	3 387	3 413	3.79	3 918	4 044	0.59	3 710	3 751	186.60
delaunay20	8	6 183	6 279	92.04	6 296	6 545	5.79	7 497	7 864	0.60	6 834	7 049	187.22
delaunay20	16	9 968	10 108	123.74	10 355	10 447	6.44	11 821	12 069	0.64	11 060	11 279	191.79
delaunay20	32	15 697	15 862	157.06	16 111	16 271	7.24	18 955	19 156	0.70	17 637	17 886	195.21
delaunay20	64	23 568	23 826	183.10	24 373	24 689	7.92	27 981	28 383	0.80	25 919	26 260	203.48
deu	2	161	162	112.38	166	169	5.58	236	261	2.53	170	173	473.82
deu	4	395	404	176.32	408	450	10.88	660	694	2.53	441	456	502.41
deu	8	712	749	196.99	820	848	13.21	1 094	1 187	2.57	815	832	524.13
deu	16	1 255	1 313	274.12	1 360	1 423	14.19	1 751	1 926	2.56	1 355	1 412	531.58
deu	32	2 051	2 135	404.52	2 311	2 353	14.94	2 892	3 053	2.60	2 226	2 336	536.14
deu	64	3 388	3 469	479.46	3 753	3 856	16.12	4 785	4 891	2.73	3 642	3 752	549.63
email-EuAll	2	617	739	31.51	827	940	1.93	831	1 602	0.46	658	676	3.55
email-EuAll	4	2 569	2 624	25.34	2 722	2 880	3.24	2 971	3 835	0.81	3 497	3 729	6.68
email-EuAll	8	9 208	9 757	31.91	12 484	12 960	3.93	12 566	14 024	0.90	13 640	14 209	11.57
email-EuAll	16	19 860	20 377	34.92	23 161	23 664	4.34	23 311	24 056	1.01	21 128	21 247	17.50
email-EuAll	32	26 231	26 410	71.61	29 660	30 433	5.17	31 015	31 659	1.10	27 689	27 839	24.27
email-EuAll	64	32 076	32 676	109.02	35 091	35 635	4.82	36 822	37 384	1.29	33 305	33 489	33.74

Table A.5: Detailed per instance results of KaFFPa and hMetis. The experiments have been repeated ten times. Best shows the edge cut of the best partition that occured. Avg. shows the average edge cut of the partitions created, t denotes the average running time.

		KaFFPaStrong			KaFFPaEco			KaFFPaFast			hMetis		
graph	k	Best	Avg	t	Best	Avg	t	Best	Avg	t	Best	Avg	t
eur	2	130	143	517.73	256	292	35.13	458	559	12.61	142	153	2 323.86
eur	4	346	413	977.92	454	603	66.58	656	1 092	12.37	335	378	2 460.65
eur	8	745	791	1 204.99	875	1 039	88.32	1 510	1 773	12.37	754	817	2 652.47
eur	16	1 375	1 523	1 436.04	1 743	1 873	91.02	2 478	2 665	12.25	1 569	1 653	2 729.56
eur	32	2 481	2 575	1 704.70	2 699	2 816	89.14	3 901	4 108	12.30	2 693	2 825	2 746.53
eur	64	4 364	4 427	1 818.93	4 768	4 917	115.51	6 408	6 702	13.33			
fe_ocean	2	311	311	2.88	311	311	0.13	373	373	0.06	416	434	13.08
fe_ocean	4	1 697	1 778	5.68	1 800	1 811	0.38	2 046	2 185	0.07	2 004	2 054	17.94
fe_ocean	8	3 946	4 056	8.82	4 615	4 862	0.51	5 671	6 045	0.09	4 747	4 920	18.81
fe_ocean	16	7 903	8 120	15.54	8 599	9 022	0.71	10 303	10 603	0.14	9 276	9 796	21.19
fe_ocean	32	12 805	13 036	24.87	14 824	15 098	1.15	16 302	16 827	0.22	14 487	15 656	25.33
fe_ocean	64	20 447	20 701	56.95	22 515	22 793	1.90	24 363	24 975	0.37	23 174	24 329	36.36
fe_tooth	2	3 789	4 017	3.28	4 174	4 222	0.14	4 319	4 703	0.08	4 093	4 139	13.37
fe_tooth	4	6 819	7 012	8.82	6 954	7 328	0.88	7 670	8 214	0.09	7 333	7 589	13.82
fe_tooth	8	11 418	11 681	12.59	12 094	12 293	1.02	12 897	13 477	0.13	12 335	12 588	14.46
fe_tooth	16	17 541	17 850	19.28	18 698	19 034	1.22	20 183	20 534	0.16	18 793	19 026	16.25
fe_tooth	32	25 373	25 847	31.23	26 535	27 162	1.59	28 507	29 028	0.24	26 920	27 246	20.96
fe_tooth	64	35 490	35 804	49.75	37 247	37 776	2.01	38 950	39 527	0.39	37 714	37 876	30.56
great-britain	2	82	84	159.16	82	87	9.93	118	118	3.52	84	85	604.80
great-britain	4	217	221	229.86	226	241	16.21	294	324	3.53	227	234	681.65
great-britain	8	389	403	290.73	404	451	16.58	561	587	3.54	432	441	716.38
great-britain	16	643	668	334.59	707	735	16.84	867	919	3.53	685	699	793.61
great-britain	32	1 169	1 196	431.02	1 253	1 336	16.55	1 539	1 593	3.58	1 226	1 273	781.94
great-britain	64	1 925	1 954	433.05	2 111	2 169	17.79	2 432	2 564	3.63	2 074	2 107	803.92
htric00	2	978	978	255.86	1 057	1 066	8.08	1 330	1 406	2.43	1 167	1 227	1 092.37
htric00	4	2 611	2 713	464.44	2 672	2 876	29.37	3 943	4 150	2.47	3 074	3 277	1 117.07
htric00	8	4 903	5 058	727.15	5 116	5 398	38.97	6 456	6 809	2.53	6 020	6 187	1 119.45
htric00	16	7 693	7 821	1 083.87	8 011	8 177	40.46	9 394	9 988	2.63	9 253	9 434	1 124.69
htric00	32	11 838	11 979	1 231.94	12 148	12 391	33.42	14 466	15 058	2.70	14 004	14 216	1 129.34
htric00	64	17 656	17 883	1 530.36	17 932	18 258	30.50	21 577	21 963	2.88	20 128	20 590	1 161.32
loc-brightkite	2	17 887	18 172	41.82	29 168	29 497	7.02	30 118	30 342	1.69	17 860	17 992	11.30
loc-brightkite	4	33 319	34 569	63.58	56 684	59 672	12.81	58 824	61 528	2.97	33 640	34 193	14.48
loc-brightkite	8	45 309	46 250	138.03	76 100	77 906	14.05	80 369	82 123	3.40	45 082	45 374	19.02
loc-brightkite	16	53 572	54 069	321.37	82 379	85 100	15.45	91 550	93 754	3.58	53 248	53 506	26.74
loc-brightkite	32	61 438	62 116	730.97	85 499	87 043	16.43	102 871	104 979	3.91	63 329	63 473	37.51
loc-brightkite	64	70 473	71 051	1 856.11	90 596	91 310	19.91	109 071	111 791	3.66	73 354	73 470	58.29
nlr	2	3 543	3 543	244.46	3 551	3 603	13.12	4 250	4 487	4.02	3 965	4 050	1 250.23
nlr	4	7 775	7 838	610.41	7 868	7 930	28.56	9 012	9 321	4.07	8 763	8 950	1 250.84
nlr	8	13 744	13 918	754.45	14 254	14 580	38.83	16 633	17 090	4.08	15 773	16 308	1 266.14
nlr	16	22 067	22 243	943.16	22 740	23 041	45.45	25 792	26 804	4.19	25 452	26 004	1 282.67
nlr	32	34 215	34 402	975.52	34 836	35 216	44.43	41 003	41 409	4.27	39 704	40 133	1 298.60
nlr	64	50 789	51 222	982.09	52 091	52 663	43.56	59 975	60 502	4.48	58 255	59 355	1 305.62
p2p-gnu04	2	7 172	7 260	6.75	8 130	8 396	0.38	10 160	10 186	0.11	7 163	7 268	2.38
p2p-gnu04	4	11 755	11 833	14.80	12 997	13 363	1.83	14 794	15 189	0.51	11 880	11 914	3.83
p2p-gnu04	8	14 609	14 724	16.22	15 897	16 103	2.10	16 994	17 214	0.55	14 868	14 927	5.33
p2p-gnu04	16	16 787	16 908	23.60	17 422	17 718	2.84	18 141	18 490	0.52	17 125	17 184	8.41
p2p-gnu04	32	18 100	18 200	48.97	18 534	18 659	4.07	19 486	19 565	0.56	18 592	18 618	11.69
p2p-gnu04	64	18 992	19 048	86.86	19 507	19 574	3.96	20 374	20 444	0.52	19 673	19 712	13.04
rgg20	2	2 088	2 101	52.07	2 139	2 178	2.50	2 932	2 968	0.63	2 163	2 294	276.48
rgg20	4	4 157	4 248	90.73	4 300	4 377	6.19	5 770	5 967	0.65	4 346	4 565	284.51
rgg20	8	7 676	7 824	111.87	8 255	8 481	7.89	10 969	11 551	0.66	8 307	8 632	292.18
rgg20	16	12 454	12 836	130.28	13 437	13 672	8.42	18 006	18 379	0.70	13 797	14 153	293.56
rgg20	32	19 945	20 428	145.08	21 545	22 131	8.87	27 513	28 491	0.76	21 693	22 130	296.72
rgg20	64	31 058	31 336	160.63	33 207	33 683	9.38	41 735	42 663	0.88	32 400	32 951	304.92
slashdot0902	2	67 694	68 601	28.26	104 978	105 272	3.83	118 515	118 732	0.58	97 287	97 357	77.71
slashdot0902	4	137 647	141 842	147.34	212 334	212 532	3.81	212 174	212 820	1.67	172 444	172 941	93.26
slashdot0902	8	205 785	209 595	311.76	268 923	269 392	8.00	269 409	269 883	3.24	222 297	223 124	136.28
slashdot0902	16	248 441	250 348	566.81	291 493	292 641	17.41	301 186	301 655	2.77	255 779	257 580	175.62
slashdot0902	32	274 898	276 147	1 271.27	301 431	303 125	39.98	318 435	318 879	3.48	280 489	281 176	205.69
slashdot0902	64	293 791	294 320	3 415.84	311 628	312 230	98.99	325 601	325 946	4.62	298 067	298 715	247.89
wave	2	8 623	8 663	8.49	8 848	9 090	0.51	9 219	10 972	0.16	8 986	9 039	39.91
wave	4	16 781	17 043	18.09	18 717	20 010	1.96	20 152	23 203	0.19	17 707	18 391	41.66
wave	8	28 750	29 105	37.88	31 028	31 505	2.99	32 910	34 761	0.25	30 654	31 346	44.85
wave	16	42 629	43 232	65.02	44 720	45 692	3.60	48 233	49 019	0.31	45 399	46 376	47.54
wave	32	62 237	62 618	103.68	64 495	65 473	4.08	68 752	69 444	0.42	66 225	66 710	52.65
wave	64	85 401	86 225	154.80	88 767	89 960	4.83	94 218	95 040	0.64	91 443	91 820	65.95

Table A.6: Detailed per instance results of KaFFPa and hMetis. The experiments have been repeated ten times. Best shows the edge cut of the best partition that occured. Avg. shows the average edge cut of the partitions created, t denotes the average running time.

		KaSPar			Scotch			kMetis			DibaP-Lite			DibaPFull		
graph	k	Best	Avg	t	Best	Avg	t	Best	Avg	t	Best	Avg	t	Best	Avg	t
144	2	6 458	6 483	9.11	6 562	6 657	0.47	6 865	7 480	0.15	7 182	7 182	1.75	6 678	7 124	1.59
144	4	15 253	15 567	22.88	16 768	16 946	1.04	17 507	17 828	0.16	15 627	16 252	3.80	16 225	16 548	2.01
144	8	25 368	25 559	39.67	27 814	28 247	1.60	28 143	28 671	0.16	26 400	27 948	6.85	26 095	26 340	2.23
144	16	37 808	38 357	65.60	42 099	42 683	2.23	42 175	43 048	0.17	38 799	40 504	12.19	38 888	39 227	2.60
144	32	56 771	57 387	109.67	62 217	62 890	3.00	62 318	63 006	0.18	58 925	59 751	23.74	57 081	58 106	3.41
144	64	80 084	80 948	163.77	87 653	88 144	3.85	86 740	87 409	0.21	82 548	83 226	45.82	81 026	81 808	6.69
598a	2	2 376	2 382	4.41	2 417	2 433	0.28	2 464	2 489	0.10	2 430	2 430	1.26	2 452	2 800	1.87
598a	4	7 900	7 925	9.05	8 175	8 236	0.68	8 340	8 556	0.11	8 189	8 195	2.67	8 320	8 416	2.09
598a	8	15 873	15 987	21.30	16 850	17 036	1.15	17 143	17 573	0.12	16 497	16 803	5.20	16 116	16 418	2.31
598a	16	25 899	26 205	38.94	28 081	29 173	1.68	28 530	29 180	0.12	26 983	28 312	9.58	26 459	26 944	2.55
598a	32	39 575	40 162	66.64	43 421	43 896	2.29	43 793	43 942	0.13	41 830	42 537	18.98	39 865	40 430	3.92
598a	64	58 220	58 799	101.06	62 968	63 518	2.94	62 549	62 764	0.15	59 672	60 364	36.54	58 617	59 218	6.29
g3circuit	2	1 098	1 111	124.92	1 245	1 320	0.85	1 468	1 523	0.87	1 388	1 388	4.36	1 181	1 181	2.92
g3circuit	4	2 622	2 690	95.49	3 069	3 152	1.79	3 372	3 703	0.89	3 655	4 089	8.51	3 735	4 075	3.77
g3circuit	8	4 872	5 025	130.33	5 473	5 669	2.76	6 041	6 446	0.89	6 149	6 976	16.55	5 575	5 965	4.06
g3circuit	16	7 978	8 110	130.06	9 068	9 543	3.88	10 394	10 835	0.92	10 601	11 150	33.40	8 156	9 701	5.29
g3circuit	32	11 792	11 893	152.52	13 606	14 036	5.08	16 342	16 342	0.93	15 337	16 095	64.26	13 782	14 523	8.49
g3circuit	64	18 942	19 134	213.99	21 375	21 960	6.79	24 715	25 208	0.96	22 374	22 974	138.57	20 743	21 349	15.08
PGPgcomp	2	367	385	0.61	471	564	0.05	405	426	0.01						
PGPgcomp	4	670	713	1.02	936	998	0.10	851	895	0.01						
PGPgcomp	8	1 100	1 158	1.49	1 204	1 378	0.13	1 261	1 337	0.01						
PGPgcomp	16	1 725	1 808	2.29	1 780	1 899	0.17	1 799	1 877	0.01						
PGPgcomp	32	2 511	2 572	2.83	2 401	2 478	0.20	2 467	3 610	0.02						
PGPgcomp	64	3 438	3 837	3.56	3 127	3 246	0.23	3 540	3 616	0.03						
af_shell10	2	26 225	26 225	135.83	26 225	29 072	2.44	28 125	28 755	1.83	26 225	26 225	6.08	26 225	26 225	2.86
af_shell10	4	54 725	55 450	102.50	56 425	57 397	4.80	60 750	62 077	1.83	69 975	70 205	10.33	57 900	57 945	3.00
af_shell10	8	97 309	100 505	97.41	108 800	110 562	7.66	114 075	119 062	1.84	114 475	116 270	15.39	107 000	108 010	3.33
af_shell10	16	163 100	166 031	138.12	171 225	172 557	10.64	186 125	190 017	1.85	184 875	191 665	30.18	167 250	170 367	4.21
af_shell10	32	249 888	254 475	144.87	270 150	272 470	13.85	290 400	290 565	1.85	289 200	292 211	54.24	255 950	258 510	7.10
af_shell10	64	374 677	378 555	180.91	399 950	401 950	17.63	426 933	430 767	1.88	421 307	427 531	129.33	383 650	387 313	12.36
as-22july06	2	3 515	3 530	10.38	4 573	4 871	0.03	3 672	3 898	0.08						
as-22july06	4	7 802	7 835	17.34	9 393	9 625	0.06	8 091	9 029	0.11						
as-22july06	8	11 798	12 000	23.72	12 699	12 978	0.08	11 689	12 394	0.15						
as-22july06	16	14 854	15 028	25.13	15 611	16 040	0.10	14 326	14 722	0.15						
as-22july06	32	17 578	17 789	31.17	18 167	18 412	0.11	16 997	20 549	0.51						
as-22july06	64	20 769	21 010	40.58	21 046	21 122	0.13	23 447	23 819	0.25						
asia	2	7	7	3 788.12	7	17	4.49	78	1 520	7.97						
asia	4	19	19	1 650.24	33	47	8.70	45	381	7.69						
asia	8	50	51	1 856.76	78	101	12.56	295	295	7.58						
asia	16	127	131	1 031.91	200	218	16.15	434	601	7.48						
asia	32	294	304	753.96	352	402	19.67	807	1 091	7.72						
asia	64	587	600	407.27	700	744	22.59	1 435	1 812	7.83						
auto	2	9 723	9 759	30.61	10 225	10 307	1.08	10 806	10 830	0.47	12 324	12 324	4.41	10 027	10 845	4.14
auto	4	25 924	26 094	52.36	27 940	28 443	2.41	27 950	30 664	0.50	27 340	28 019	9.44	26 601	26 634	5.62
auto	8	45 161	45 285	84.70	48 512	49 544	3.71	51 747	52 769	0.50	48 922	49 365	13.83	48 140	48 201	5.31
auto	16	76 113	76 596	154.86	82 454	83 682	5.37	84 981	87 570	0.53	80 899	82 968	24.60	78 658	79 852	6.77
auto	32	121 047	121 727	272.09	130 269	131 706	7.38	132 339	134 179	0.56	126 748	128 344	42.21	124 359	124 786	9.31
auto	64	173 818	174 453	416.10	189 304	191 308	9.63	189 180	189 950	0.59	178 406	179 868	75.82	173 527	175 207	14.30
delaunay20	2	1 700	1 728	80.22	1 956	2 048	0.77	2 204	2 204	0.63	2 138	2 362	2.92	1 912	2 779	2.55
delaunay20	4	3 412	3 447	56.17	3 891	4 014	1.46	4 060	4 156	0.64	4 061	4 061	4.44	3 960	4 015	2.60
delaunay20	8	6 280	6 325	72.54	7 170	7 368	2.18	7 865	7 865	0.64	7 341	7 485	9.55	6 908	7 025	2.66
delaunay20	16	10 186	10 225	54.70	11 583	11 737	2.90	12 123	12 545	0.67	11 833	11 985	27.06	11 220	11 311	3.33
delaunay20	32	15 973	16 055	77.56	17 801	18 146	3.72	18 587	19 052	0.66	18 305	18 498	59.49	17 130	17 386	6.12
delaunay20	64	23 891	23 969	91.07	26 634	26 900	4.65	28 081	28 264	0.67	27 353	27 674	155.82	25 871	26 043	10.96
deu	2	172	174	142.15	242	278	2.03	262	284	2.84						
deu	4	415	427	132.60	636	699	4.07	748	751	2.82						
deu	8	749	766	132.38	1 144	1 248	6.00	1 212	1 317	2.81						
deu	16	1 296	1 323	133.82	1 945	2 028	7.98	2 086	2 187	2.82						
deu	32	2 182	2 216	154.25	3 119	3 228	9.89	3 183	3 387	2.83						
deu	64	3 594	3 646	200.22	4 776	4 930	11.94	5 128	5 173	2.85						
email-EuAll	2	1 246	1 665	3.26	909	910	0.04	1 796	3 667	0.03						
email-EuAll	4	3 642	4 279	4.07	3 452	5 737	0.08	6 317	9 557	0.04						
email-EuAll	8	14 128	16 202	11.59	10 058	11 724	0.11	23 137	25 271	0.08						
email-EuAll	16	24 603	25 826	17.90	19 721	20 347	0.13	38 587	39 553	0.12						
email-EuAll	32	32 003	33 273	28.12	27 896	28 300	0.16	39 277	40 146	0.14						
email-EuAll	64	37 393	38 822	37.28	33 600	34 077	0.17	39 591	41 080	0.17						

Table A.7: Detailed per instance results of KaSPar, Scotch, kMetis and DibaP. The experiments have been repeated ten times. Best shows the edge cut of the best partition that occured. Avg. shows the average edge cut of the partitions created, t denotes the average running time.

		KaSPar			Scotch			kMetis			DibaP-Lite			DibaPFull		
graph	k	Best	Avg	t	Best	Avg	t	Best	Avg	t	Best	Avg	t	Best	Avg	t
eur	2	137	148	617.00	444	502	8.28	269	424	15.43						
eur	4	396	410	595.16	765	945	16.20	919	1 508	15.38						
eur	8	764	782	579.65	1 385	1 535	24.13	1 858	2 599	15.49						
eur	16	1 425	1 465	565.10	2 608	2 693	31.99	3 262	7 362	15.98						
eur	32	2 618	2 651	583.25	4 184	4 380	39.85	5 611	7 609	15.70						
eur	64	4 484	4 537	683.97	6 688	6 856	47.36	13 075	13 566	16.23						
fe_ocean	2	317	318	3.02	389	400	0.12	519	546	0.08	396	396	1.09	332	1 256	0.81
fe_ocean	4	1 723	1 795	7.13	1 943	1 990	0.29	2 142	2 186	0.08	1 969	1 969	1.77	1 963	2 071	0.85
fe_ocean	8	4 026	4 070	14.19	4 414	4 721	0.55	5 306	5 556	0.09	4 975	5 327	3.77	4 965	5 232	0.90
fe_ocean	16	7 808	8 096	25.34	8 886	9 200	0.87	10 013	10 319	0.09	9 329	9 460	6.91	9 419	9 516	1.09
fe_ocean	32	13 181	13 370	42.52	13 786	14 941	1.26	16 353	16 820	0.10	15 675	16 029	15.18	16 346	16 580	1.64
fe_ocean	64	21 231	21 664	72.64	21 800	22 764	1.72	24 256	24 427	0.11	24 340	24 554	30.94	24 379	24 799	2.96
fe_tooth	2	3 796	3 894	4.77	3 943	4 092	0.25	4 160	4 369	0.06	4 460	4 545	1.07	4 672	5 067	1.28
fe_tooth	4	6 934	7 012	8.08	7 338	7 817	0.49	7 871	8 189	0.06	7 417	7 417	1.78	7 738	8 228	1.29
fe_tooth	8	11 465	11 577	13.92	12 418	12 669	0.76	13 157	13 160	0.07	12 219	12 594	3.62	12 064	12 285	1.39
fe_tooth	16	17 581	18 000	22.82	18 958	19 272	1.11	19 543	20 252	0.07	18 458	19 178	7.13	18 305	18 605	1.56
fe_tooth	32	25 849	26 303	36.70	27 200	27 605	1.49	28 598	28 776	0.08	26 950	27 819	13.86	26 608	26 882	2.53
fe_tooth	64	36 144	36 604	53.54	37 537	38 060	1.90	39 116	39 360	0.09	37 476	38 078	33.04	36 813	37 103	3.90
great-britain	2	82	84	1 852.55	119	129	3.59	141	155	5.12						
great-britain	4	217	220	1 054.88	308	337	6.91	360	1 359	5.15						
great-britain	8	385	394	732.33	565	625	9.93	689	1 088	5.11						
great-britain	16	652	662	743.91	951	1 007	12.71	1 053	1 323	5.13						
great-britain	32	1 194	1 213	504.69	1 598	1 653	15.31	1 863	2 037	5.01						
great-britain	64	2 021	2 046	412.96	2 533	2 604	17.79	2 953	3 084	4.96						
htric00	2	1 088	1 106	768.16	1 188	1 275	3.03	1 353	1 477	4.29	1 209	1 209	17.40	1 254	1 915	11.36
htric00	4	2 835	2 888	844.19	3 277	3 400	5.99	3 381	3 514	4.30	3 146	3 146	29.42	3 265	3 457	12.89
htric00	8	5 330	5 414	586.27	5 936	6 308	8.79	6 676	6 691	4.36	6 126	6 329	60.44	6 057	6 271	15.24
htric00	16	8 379	8 413	433.17	9 132	9 438	11.19	10 037	10 196	4.34	9 131	9 380	113.03	8 957	9 158	19.31
htric00	32	12 663	12 778	386.54	13 590	14 036	13.49	14 586	14 891	4.39	13 897	14 040	208.77	13 194	13 511	31.71
htric00	64	19 021	19 083	488.04	20 186	20 433	15.72	21 591	21 844	4.38	19 921	20 308	530.64	19 673	19 887	55.37
loc-brightkite	2	17 755	18 088	93.48	22 602	26 219	0.52	19 801	20 918	0.13						
loc-brightkite	4	35 312	36 111	234.95	40 679	42 173	0.87	37 103	37 224	0.16						
loc-brightkite	8	49 854	51 017	479.86	55 079	57 028	1.16	48 835	49 698	0.18						
loc-brightkite	16	59 072	60 881	782.96	64 053	65 710	1.39	56 304	56 626	0.21						
loc-brightkite	32	71 135	72 466	1 162.00	73 323	75 172	1.59	65 345	65 951	0.31						
loc-brightkite	64	82 979	83 738	1 460.56	82 802	84 325	1.77	76 069	76 377	0.49						
nlr	2	3 631	3 643	705.74	3 937	4 110	5.79	4 136	4 275	4.01	4 029	4 050	19.09	4 224	4 290	13.26
nlr	4	7 952	7 994	358.67	8 767	8 935	10.50	9 211	9 664	4.03	9 046	9 132	27.73	8 666	9 170	14.60
nlr	8	13 996	14 070	435.25	16 118	16 403	14.40	17 301	17 682	4.03	15 880	16 099	49.37	15 113	15 285	15.81
nlr	16	22 361	22 469	349.37	25 269	25 823	17.97	27 106	27 529	4.05	25 378	25 950	88.56	24 312	24 604	19.44
nlr	32	34 674	34 786	295.19	38 843	39 402	21.18	40 799	41 677	4.08	38 808	39 523	180.84	37 178	37 288	30.08
nlr	64	51 654	51 893	323.96	56 952	57 758	24.44	59 964	60 097	4.14	57 999	58 508	380.38	55 782	56 120	47.78
p2p-gnu04	2	7 100	7 172	9.86	7 097	7 163	0.03	7 597	7 817	0.01						
p2p-gnu04	4	12 479	12 648	18.70	11 940	12 016	0.05	12 099	12 276	0.02						
p2p-gnu04	8	15 982	16 273	28.90	15 113	15 172	0.07	15 033	15 204	0.03						
p2p-gnu04	16	18 023	18 221	38.28	17 394	17 450	0.08	17 056	17 142	0.04						
p2p-gnu04	32	19 470	19 582	43.44	18 878	18 917	0.09	18 427	18 480	0.06						
p2p-gnu04	64	20 497	20 833	34.37	19 926	19 978	0.11	19 496	19 509	0.07						
rgg20	2	2 179	2 214	81.69	2 719	2 848	0.96	3 119	3 341	0.86						
rgg20	4	4 298	4 363	58.54	5 839	6 104	1.89	5 983	6 303	0.85						
rgg20	8	7 793	7 913	54.32	11 125	11 463	2.87	11 178	11 651	0.87						
rgg20	16	12 633	12 808	53.45	17 190	17 437	3.83	17 255	18 207	0.88						
rgg20	32	20 273	20 539	74.38	26 775	27 479	5.01	28 419	28 918	0.88						
rgg20	64	30 965	31 135	91.37	38 640	39 906	6.24	41 229	42 735	0.90						
slashdot0902	2	89 698	92 926	291.47	58 990	63 657	2.29	97 931	98 258	0.16						
slashdot0902	4	199 345	205 069	593.89	138 702	141 969	2.79	182 859	188 298	0.17						
slashdot0902	8	256 043	257 595	1 117.12	199 656	201 154	3.04	224 801	226 926	0.22						
slashdot0902	16	286 802	288 152	1 826.87	244 799	246 440	3.22	259 601	260 434	0.31						
slashdot0902	32	305 169	305 818	2 818.33	276 634	277 676	3.37	281 543	282 657	0.61						
slashdot0902	64	316 627	318 042	4 451.21	297 892	299 037	3.42	301 519	301 519	1.06						
wave	2	8 657	8 690	11.62	8 909	9 059	0.60	9 030	9 337	0.14	9 120	9 338	1.66	8 894	8 986	1.64
wave	4	16 847	16 955	26.29	18 672	19 693	1.24	18 885	19 246	0.15	17 594	19 167	3.20	17 383	18 672	1.80
wave	8	28 742	28 878	46.37	32 007	33 015	1.88	33 993	34 704	0.16	31 093	32 058	5.83	30 000	30 938	1.72
wave	16	42 810	43 053	76.31	47 917	48 919	2.55	47 462	49 020	0.16	44 287	45 755	10.51	43 550	44 446	2.10
wave	32	62 538	63 129	118.64	68 617	69 442	3.36	67 402	67 943	0.17	64 176	65 194	20.95	63 702	64 489	2.75
wave	64	86 683	87 444	168.74	94 703	95 447	4.27	92 808	93 039	0.19	89 070	90 384	42.39	86 899	88 211	5.80

Table A.8: Detailed per instance results of KaSPar, Scotch, kMetis and DibaP. The experiments have been repeated ten times. Best shows the edge cut of the best partition that occured. Avg. shows the average edge cut of the partitions created, t denotes the average running time.

Zusammenfassung

Viele Anwendungen der Informatik beinhalten die Verarbeitung und Partitionierung von großen Graphen z.B. die Finite Element Methode, Digitaler Schaltkreisentwurf, Routenplanung, Analyse des Webgraphen oder die Analyse von Sozialen Netzwerken. Ein bekanntes Beispiel, in der gute Partitionierungen von unstrukturierten Graphen benötigt werden, ist die Parallelverarbeitung. In diesem Gebiet wird Graphpartitionierung häufig verwendet, um ein zugrundeliegendes Modell von Kommunikation und Berechnung zu partitionieren. Grob gesagt reflektieren Knoten in diesem Modell Berechnungseinheiten und Kanten Kommunikationseinheiten. Dieser Graph muss nun so partitioniert werden, dass möglichst wenig Kanten zwischen den Blöcken verlaufen, da Kommunikation in der Parallelverarbeitung teuer ist. Wenn man dazu k Prozessoren verwenden möchte, muss der Graph in k ungefähr gleich große Blöcke aufgeteilt werden. In dieser Arbeit wird die Variante des Graphpartitionierungsproblems untersucht, in der die Blockgrößen auf $(1+\varepsilon)$ mal der durchschnittlichen Blockgröße beschränkt werden und Anzahl der Schnittkanten minimiert werden soll.

Da das allgemeine Problem NP-schwer ist, werden in der Praxis häufig Heuristiken verwendet um Partitionen von Graphen zu erzeugen. Eine sehr erfolgreiche Heuristik ist das Mehrschichtverfahren, welches ausgehend vom Eingabegraphen zunächst durch Kontraktion von z.B. Matchings eine Hierarchie von immer kleineren Graphen erzeugt. Diese kleineren Graphen haben in der Regel ähnliche Eigenschaften wie der Eingabegraph. Der kleinste Graph in dieser Folge wird dann mithilfe einer Heuristik initial partitioniert, d.h. die Knoten des Graphen werden Blöcken zugewiesen. Anschließend wird die Kontraktion schrittweise rückgängig gemacht, in dem eine Partitionierung auf die nächste, feinere Ebene in der Hierarchie übertragen wird und jeweils anschließend ein lokaler Verfeinerungsalgorithmus benutzt wird, um den Kantenschnitt auf der aktuellen Stufe zu verbessern.

Obwohl in den letzten zwei Jahrzehnten schon mehrere Mehrschicht Graphpartitionierer entwickelt wurden, hatten wir den Eindruck, dass viele Aspekte des Mehrschichtverfahrens nicht gut verstanden wurden. Daher hatten wir angefangen unseren eigenen Graphpartitionierer KaPPa (Karlsruhe Parallel Partitioner) [87], mit Fokus auf Skalierbarkeit, zu entwickeln. Überraschenderweise gelang es uns, durch verhältnismäßig einfache Methoden, ebenfalls verbesserte Lösungsqualität zu erhalten. Dies hat uns motiviert, jede einzelne Komponente des Mehrschichtverfahrens auf den Prüfstand zu stellen. Unsere Ziele in dieser Arbeit sind zum einen hohe Partitionierungsqualität und zum anderen schnelle sequentielle Ausführungszeit für große Graphen. Wir präsentieren eine Reihe von Verfahren, die zu verbesserter Lösungsqualität führen. Dies beinhaltet verschiedene Kontraktionsheuristiken, flussbasierte Methoden, verbesserte lokale Suchen, wiederholte Versuche ähnlich zu Verfahren, die in Mehrgitterlösern verwendet werden, einen verteilt evolutionären Algorithmus und einen neuen Algorithmus für den Fall, dass stark balancierte Partitionen benötigt werden.

Die Arbeit beschäftigt sich zunächst mit dem Mehrschichtverfahren. Wir vergleichen verschiedene Matching und Mehrgitter-inspirierte Vergröberungsverfahren und experimentieren mit algebraischer Distanz zwischen Knoten. Weiterhin schauen wir uns zwei neue lokale Verbesserungsheuristiken an, also Algorithmen die Knoten zwischen den Blöcken bewegen, um den Kantenschnitt zu reduzieren. Das erste Verfahren ist im Vergleich mit existierenden Ansätzen stark lokalisiert und das zweite Verfahren basiert auf der mehrmaligen Verwendung von Max-Flow Min-Cut Berechnungen in Bereichen um den Schnitt eines schon partitionierten Graphens.

Darauf aufbauend liefern verschiedene Meta-Heuristiken einen weiteren Beitrag dieser Arbeit. Zunächst schauen wir uns fortgeschrittene globale Suchstrategien an, sogenannte iterierte Mehrschichtverfahren. Der iterierte V-Zyklus wurde von Soper et al. [157] eingeführt und kann auf Mehrgitterlöser für das Lösen von linearen Gleichungssystemen zurückgeführt werden. Die Hauptidee ist es, die Vergröberungs- und Verfeinerungsphase des Mehrschichtverfahrens mehrfach zu durchlaufen. Sobald der Graph partitioniert ist, werden Schnittkanten nicht mehr kontrahiert. Wir schauen uns zwei weitere Strategien an, den F- und den W-Zyklus, und evaluieren experimentell, dass diese Algorithmen einen signifikanten Vorteil gegenüber mehrfachen Neustarts des Multilevelverfahrens haben, falls ein verhältnismäßig schwacher Verfeinerungsalgorithmus verwendet wird. Insgesamt erhalten wir ein System, das zum einen für viele bekannte Benchmarkinstanzen zu besserer Lösungsqualität führt und zum anderen zu einem guten Tradeoff zwischen Laufzeit und Lösungsqualität.

Im Walshaw Benchmark wurde KaFFPa auf kleinen Graphen geschlagen. Die besseren Ergebnisse wurden von einem Partitionierer berechnet, der ein Mehrschichtverfahren mit einer evolutionären Strategie verbinden. Daher ist ein weiterer Beitrag dieser Arbeit ein neuer verteilter evolutionärer Partitionierungsalgorithmus, KaFFPaE (KaFFPaEvolutionary). KaFFPaE verwendet KaFFPa, um neue effektive Kombinations- und Mutationsoperationen bereitzustellen. Dies wird mit einem skalierbaren Kommunikationsprotokoll kombiniert, das ähnlich zu einer Technik namens Randomized Rumor Spreading funktioniert. Insgesamt erhalten wir einen Algorithmus, der für viele Graphen in sehr kurzer Zeit Partitionierungen mit sehr hoher Qualität berechnen kann.

Die bisher präsentierten Algorithmen berechnen sehr gute Partitionen in einem angemessenen Zeitrahmen, wenn eine gewisse Unbalanciertheit $\varepsilon > 0$ erlaubt ist. Allerdings sind diese Algorithmen für den Fall von stark balancierten Partitionen noch nicht sehr gut, insbesondere im perfekt balancierten Fall $\varepsilon = 0$. Unter dieser Einschränkung sind Verfeinungsalgorithmen bisher gezwungen Knoten zwischen genau zwei Blöcken auszutauschen, um einerseits den Schnitt zu verbessern und andererseits die Balancebedingung dabei nicht zu verletzen. Wir entwickeln daher spezialisierte Methoden, die die Nachbarschaftsrelation von lokalen Suchen, für den Fall von sehr strengen Balancebedingungen, stark vergrößern. Diese Techniken versuchen mehrere lokale Suchen zwischen verschiedenen Blöcken zu kombinieren, so dass die Balance der Partition nicht verschlechtert wird. Das Finden solcher Kombinationen kann auf Suchen von negativen Kreisen in einem gerichteten Graphen zurückgeführt werden. Wir erweitern den

Algorithmus durch Verfahren, die auch unbalancierte Lösungen balancieren können und integrieren die Techniken in unseren evolutionären Algorithmus.

Weiterhin präsentieren wir in dieser Arbeit verschiedene algorithmische Erweiterungen unserer Verfahren, z.B. zum effizienten Partitionieren von kontinental großen Straßennetzwerken, großen sozialen Netzwerken oder Webgraphen, sowie ein Verfahren um k-Wege Knotenseparatoren aus einer gegebenen k-Partition zu berechnen.

Die Partitionierer in dieser Arbeit wurden Benchmark-getrieben entwickelt. Insgesamt haben wir so ein System erstellt, das die *meisten* Einträge im Walshaw Benchmark verbessern oder reproduzieren konnte. Eine andere Perspektive ist die Verwendung der Algorithm Engineering Methodik auf alle Aspekte der Mehrschicht Graphpartitionierung. Hierdurch wurden Verbesserungen in den Bereichen der Vergröberungs- und Verfeinerungsmethoden, global gesteuerter Suchen und im Bereich der Meta-Heuristiken erzielt.

Unsere Partitionierer funktioneren auch auf Instanzen des 10ten DIMACS Implementierungs Wetterbewerbs zum Clustern und Partitionieren von Graphen sehr gut. Im Teilwettbewerb Graphpartitionierung erreichten wir die *besten Ergebnisse* unter allen Teilnehmern in der Kategorie Lösungsqualität und in der Kategorie Laufzeit vs. Lösungsqualität. Ein überraschendes Ergebnis erhielten wir in dem Teil des Wettbewerbs, in dem die Zielfunktion nicht die Schnittgröße, sondern das tatsächliche Kommunikationsvolumen war. Dieses Problem kann als Hypergraph Partitionierungsproblem formuliert werden. Interessanterweise konnte KaFFPaE spezialisierte Hypergraphpartitionier schlagen, in dem nur die Fitnessfunktion des evolutionären Algorithmus auf diese Zielfunktion, Kommunikationsvolumen, geändert wurde – der eigentliche Mehrschichtalgorithmus optimierte immer noch die Anzahl der Schnittkanten.