KaHIP v2.10 – Karlsruhe High Quality Partitioning
User Guide

Peter Sanders and Christian Schulz

Karlsruhe Institute of Technology, *Karlsruhe, Germany*
University of Vienna, *Vienna, Austria*
Email: `sanders@kit.edu, christian.schulz@univie.ac.at`

Abstract

This paper severs as a user guide to the graph partitioning framework KaHIP (Karlsruhe High Quality Partitioning). We give a rough overview of the techniques used within the framework and describe the user interface as well as the file formats used. Moreover, we provide a short description of the current library functions provided within the framework.

Contents

1 Introduction

Given a graph $G = (V, E)$ and a number $k > 1$, the graph partitioning asks for a division of the graphs vertex set into k disjoint blocks of roughly equal size such that some objective function is minimized. The most common formulation minimizes the number of edges that run between the blocks. An example is given in Figure 1. Nowadays, the graph partitioning problem has many applications in different areas such as parallel scientific computing or graph computations [36, 7, 42, 12, 4, 5, 29], route planning [20, 25, 39, 21, 19, 22, 9, 10], VLSI Design [1, 2, 17, 8] or in solving sparse linear equation systems [13][1]. There has been a vast amount of research on graph partitioning. We refer the reader to [6] for more material on graph partitioning.

KaHIP - Karlsruhe High Quality Partitioning - is a family of graph partitioning programs based on the publications [30, 31, 33, 16, 28, 32, 37]. It includes KaFFPa (Karlsruhe Fast Flow Partitioner) in its variants Strong, Eco and Fast, KaFFPaE (KaFFPaEvolutionary) which is a parallel evolutionary algorithm that uses KaFFPa to provide combine and mutation operations, as well as KaBaPE which extends the evolutionary algorithm. Moreover, we include algorithms to output a vertex separator from a given partition.

KaHIP focuses on a version of the problem that constrains the maximum block size to $(1 + \epsilon)$ times the average block size and tries to minimize the total cut size, i.e. the number of edges that run between blocks. To be more precise, consider an undirected graph $G = (V = \{0, \ldots, n-1\}, E, c, \omega)$ with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e. $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. We are looking for *blocks* of nodes $V_1, \ldots, V_k$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in \{1..k\} : |V_i| \leq (1+\epsilon) \lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter ϵ. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. Methods to optimize the maximum communication volume of a partition are also included in the package. There is also a version of the program that enables to balance nodes and edges between the blocks which is in particular important for applications in which workload is associated with the edges as well such as sparse matrix vector multiplications or in graph processing frameworks. In this case, we set the node weight to $c(v) + \deg_\omega(v)$ for a node $v \in V$ and set the balance constraint to $\forall i \in \{1..k\} : |V_i| \leq (1 + \epsilon) \lceil \frac{c(V) + \sum_v \deg_\omega(v)}{k} \rceil$.

The purpose of the manual is to give a very rough overview over the techniques used in the partitioning programs, as well as to serve as a guide and manual on how to use our algorithms. We start with a short overview of the algorithms implemented within our graph partitioning framework. This is followed by a description of the graph format that is used. It is basically the same graph format that is used by Metis [18] and Chaco [15], and that has been used during the 10th DIMACS Implementation Challenge on Graph Clustering and Graph Partitioning [3]. We then give an overview over the user interface of KaFFPa, KaFFPaE and KaBaPE and explain which program to use to derive node separators. We finish this manual with the description of the library functions provided by the current release.
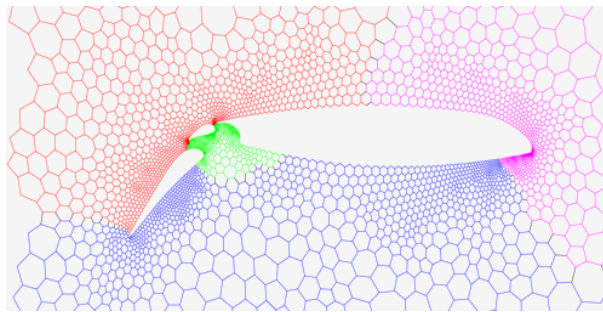

Figure 1: An example mesh that is partitioned into four blocks indicated by the colors.

---

[1]Let us know if we have missed your application here!

2   Graph Partitioning Algorithms within KaHIP

We now give a rough overview over the algorithms implemented in our graph partitioning framework. For details on the algorithms, we refer the interested reader to the corresponding papers. Figure 2 gives an overview over the techniques used with the framework.

2.1   KaFFPa

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. KaFFPa is a multilevel graph partitioning framework which contributes a number of improvements to the multilevel scheme which lead to enhanced partitioning quality. This includes flow-based methods, improved local search and repeated runs similar to the approaches used in multigrid solvers.

Local improvement algorithms are usually variants of the FM algorithm [11]. The variant KaFFPa uses is organized in rounds. In each round, a priority queue P is initialized with all vertices that are incident to more than one block, in a random order. The priority is based on the gain $g(i) = \max_P g_P(i)$ where $g_P(i)$ is the decrease in edge cut when moving i to block P. Local search then repeatedly looks for the highest gain node v and moves it to the corresponding block that maximizes the gain. However, in each round a node is moved at most once. After a node is moved, its unmoved neighbors become eligible, i.e. its unmoved neighbors are inserted into the priority queue. When a stopping criterion is reached, all movements after the best-found cut that occurred within the balance constraint are undone. This process is repeated several times until no improvement is found.

Max-Flow Min-Cut Local Improvement.   KaFFPa additionally uses more advanced local search algorithms. The first method is based on max-flow min-cut computations between pairs of blocks, in other words, a method to improve a given bipartition. Roughly speaking, this improvement method is applied between all pairs of blocks that share a non-empty boundary. The algorithm constructs a flow problem by growing an area around the given boundary vertices of a pair of blocks such that each s-t cut in this area yields a feasible bipartition of the original graph/pair of blocks *within* the balance constraint. One can then apply a max-flow min-cut algorithm to obtain a min-cut in this area and therefore a non-decreased cut between the original pair of blocks. This is improved in multiple ways, for example, by iteratively applying the method, searching in larger areas for feasible cuts, and applying most balanced minimum cut heuristics. For more details we refer the reader to [30].

Multi-try FM.   The second method for improving a given partition is called multi-try FM. This local improvement method moves nodes between blocks in order to decrease the cut. Previous k-way methods were initialized with *all* boundary nodes, i.e., all boundary nodes were eligible for movement at the beginning. Roughly speaking, the multi-try FM algorithm is a k-way improvement method that is initialized with a *single* boundary node, thus achieving a more localized search. This is repeated several rounds. The algorithm has a higher chance to escape local optima. For more details about the multi-try FM algorithm we refer the reader to [30, 37].

Iterated Multilevel Algorithms.   KaFFPa extends the concept of *iterated multilevel algorithms* which was introduced for graph partitioning by Walshaw et al. [40]. The main idea is to iterate the multilevel-scheme using different random seeds for coarsening and uncoarsening. Once the graph is partitioned, edges that are between two blocks are not contracted so that the given partition can be used as initial partition on the coarsest level. This ensures non-decreased partition quality since the refinement algorithms of KaFFPa guarantee no worsening. KaFFPa

uses F-cycles as a potentially stronger iterated multilevel algorithm. The detailed description of F-cycles for the multilevel graph partitioning framework can be found in [30].

2.2 KaFFPaE

KaFFPaE (KaFFPaEvolutionary) is a distributed evolutionary algorithm to tackle the graph partitioning problem. KaFFPaE uses KaFFPa to provide new effective combine and mutation operators. Roughly speaking, to combine two partitions of a population of the algorithm, the coarsening phase of KaFFPa is modified such that no cut edge of either of the input partitions is contracted. This ensures that both input partitions can be used as initial partition on the coarsest level and moreover that exchanging good parts of solutions can be exchanged effectively. Intuitively, the combine operator assembles good parts of solutions into a single partition. The combine operation framework is very flexible so that a partition can be combined with an arbitrary domain specific graph clustering. Moreover, the algorithm is parallelized such that each process has its own population of partitions and independently performs combine and mutation operations. This is combined with a scalable communication protocol similar to randomized rumor spreading to distribute high quality partitions among the processes. Overall, the system is able to improve the best known partitioning results for many inputs and also in a short amount of time for graphs of moderate size.
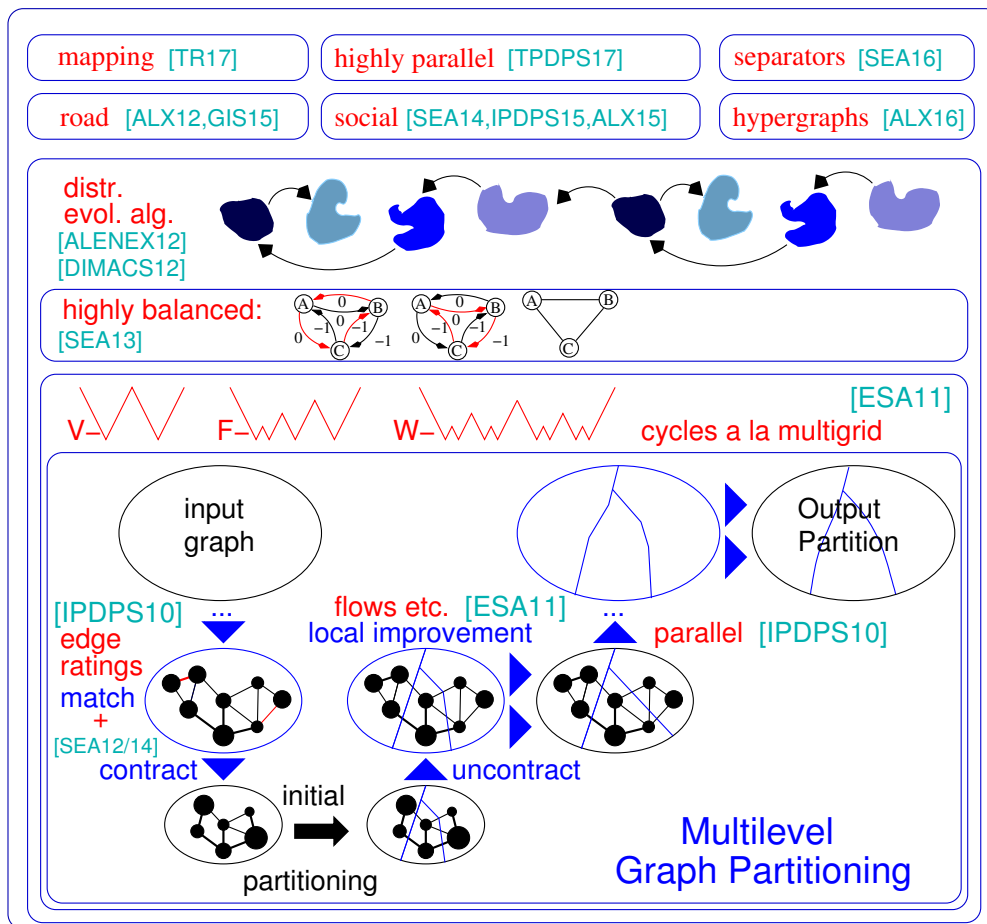

Figure 2: Overview over the techniques used in the KaHIP graph partitioning framework.

2.3 KaBaPE

KaFFPa and KaFFPaE compute partitions of very high quality when some imbalance $\epsilon > 0$ is allowed. However, they are not very good for small values of ϵ, in particular for the perfectly balanced case $\epsilon = 0$. Hence, 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. KaBaPE also provides balancing variants of these techniques that are able to make infeasible partitions feasible. In contrast to previous algorithms such as Scotch [26], Jostle [41] and Metis [18], KaBaPE is able to *guarantee* that the output partition is feasible.

2.4 Specialized Techniques for Social Networks

We also include specialized techniques for social networks. On social networks matching-based multilevel algorithms have the problem that they can not shrink the graph effectively due to the irregular structure of the graphs, To overcome this, we defined an algorithm that contracts size-constraint clusterings [23]. Here, a fast and cut-based label propagation algorithm was used to compute the clusterings. The same algorithm can be used during uncoarsening as a fast and very simple local search algorithms. To enable the methods use a preconfiguration of the algorithm that has the word social in its name. For more details on this particular method we refer the reader to [23]. Additionally, these algorithms can now also be used within the evolutionary algorithms KaFFPaE/KaBaPE. This work was joint work with Henning Meyerhenke. Moreover, in Glantz et al. [14] we looked at algorithms and edge ratings improving the maximum communication volume of a partition (which are not yet integrated into the system).

2.5 ParHIP – Parallel High Quality Partitioning

A large part of the project are distributed memory parallel algorithms designed for networks having a hierarchical cluster structure such as web graphs or social networks. Unfortunately, previous parallel graph partitioners originally developed for more regular mesh-like networks do not work well for complex networks. Here we address this problem by parallelizing and adapting the *label propagation* technique originally developed for graph clustering. By introducing size constraints, label propagation becomes applicable for both the coarsening and the refinement phase of multilevel graph partitioning. This way we exploit the hierarchical cluster structure present in many complex networks. We obtain very high quality by applying a highly parallel evolutionary algorithm to the coarsest graph. The resulting system is both more scalable and achieves higher quality than state-of-the-art systems like ParMetis or PT-Scotch. For large complex networks the performance differences are very big. As an example, our algorithm partitions a web graph with 3.3G edges in 16 *seconds* using 512 cores of a high-performance cluster while producing a high quality partition – none of the competing systems can handle this graph on our system. For more details, we refer the reader to [24].

2.6 Better Process Mapping

Communication and topology aware process mapping is a powerful approach to reduce communication time in parallel applications with known communication patterns on large, distributed memory systems. We address the problem as a quadratic assignment problem (QAP), and include algorithms to construct initial mappings of processes to processors as well as fast local search algorithms to further improve the mappings. By exploiting assumptions

that typically hold for applications and modern supercomputer systems such as sparse communication patterns and hierarchically organized communication systems, we arrive at significantly more powerful algorithms for these special QAPs. Our multilevel construction algorithms employ perfectly balanced graph partitioning techniques and excessively exploit the given communication system hierarchy. For more details, we refer the reader to [38].

2.7 Edge Partitioning

Edge-centric distributed computations have appeared as a recent technique to improve the shortcomings of think-like-a-vertex algorithms on large scale-free networks. In order to increase parallelism on this model, *edge partitioning*—partitioning edges into roughly equally sized blocks—has emerged as an alternative to traditional (node-based) graph partitioning. We include a fast parallel and sequential split-and-connect graph construction algorithm that yield high-quality edge partitions in a scalable way. Our technique scales to networks with billions of edges, and runs efficiently on thousands of PEs. Experiments have shown that our algorithm computes solutions of high quality on large real-world networks [35].

2.8 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. We use flow-based and localized local search algorithms withing a multilevel framework to compute node separators [34]. 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.

It is worth mentioning that we also provide an extended method which can 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 $V_1, \ldots, V_k$. The method of Pothen et al. [27] creates a 2-way node separator from the set of cut edges of a previously computed partition. As in the original work, we can use this method in our framework 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 can then compute a k-way node separator by computing a k-partition using KaFFPa and applying the described flow problem between all pairs of blocks that share a non-empty boundary afterwards. All pair-wise separators together are then be used as a k-way separator.

3 Graph Format

3.1 Input File Format

3.1.1 Metis File Format

The graph format used by our partitioning programs is the same as used by Metis [18], Chaco [15] and the graph format that has been used during the 10th DIMACS Implementation Challenge on Graph Clustering and Partitioning. The input graph has to be undirected, without self-loops and without parallel edges.

To give a description of the graph format, we follow the description of the Metis 4.0 user guide very closely. A graph $G = (V, E)$ with n vertices and m edges is stored in a plain text file that contains $n + 1$ lines (excluding comment lines). The first line contains information about the size and the type of the graph, while the remaining n lines contain information for each vertex of G. Any line that starts with % is a comment line and is skipped.

The first line in the file contains either two integers, $n\ m$, or three integers, $n\ m\ f$. The first two integers are the number of vertices n and the number of undirected edges of the graph, respectively. Note that in determining the number of edges m, an edge between any pair of vertices v and u is counted *only once* and not twice, i.e. we do not count the edge (v, u) from (u, v) separately. The third integer f is used to specify whether or not the graph has weights associated with its vertices, its edges or both. If the graph is unweighted then this parameter can be omitted. It should be set to 1 if the graph has edge weights, 10 if the graph has node weights and 11 if the graph has edge and node weights.

The remaining n lines of the file store information about the actual structure of the graph. In particular, the ith line (again excluding comment lines) contains information about the ith vertex. Depending on the value of f, the information stored in each line is somewhat different. In the most general form (when $f = 11$, i.e. we have node and edge weights) each line has the following structure:

$$c\ v_1\ w_1\ v_2\ w_2 \ldots v_k\ w_k$$

where c is the vertex weight associated with this vertex, $v_1, \ldots, v_k$ are the vertices adjacent to this vertex, and $w_1, \ldots, w_k$ are the weights of the edges. Note that the vertices are numbered starting from 1 (not from 0). Furthermore, the vertex-weights must be integers greater or equal to 0, whereas the edge-weights must be strictly greater than 0.

3.1.2 Binary File Format for ParHIP

In order to read files efficiently in parallel, we also use a binary file format. We give a brief description here, for an in-depth description have a look at parallel_graph_io.cpp:483-535). However note that we provide tools to convert the input file format from the previous section into the binary file format. This includes an external memory converter if the graph that needs to be converted does not fit into the memory that is used for conversion. Additionally, ParHIP can also read the simple Metis format (less efficiently).

The file uses 64bit unsigned longs. It starts with three 64bit unsigned longs, the version number (currently set to 3), the number of nodes n and the third one the number of edges of the graph. The next $n + 1$ unsigned longs store offsets. More precisely, the ith long is the *position* in which the outgoing edges of vertex i start. Starting at this position the edge targets are stored (one unsigned long for each). The $n + 1$ long in the set of offsets stores can be used to determine the end of the file. Node IDs start at 0.

3.2 Output File Formats

3.2.1 Partition

The output format of a partition is also similar to the output format of a partition provided by Metis. It is basically a text file named *tmppartitionk* where k is the number of blocks given as input to the partitioning program. This
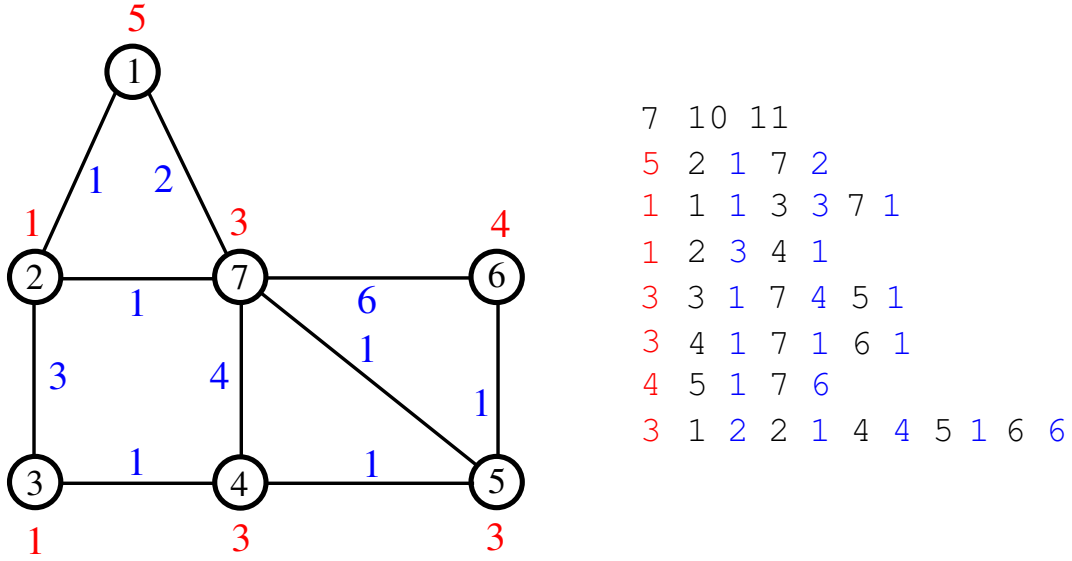
Figure 3: An example graph and its representation in the graph format. The IDs of the vertices are drawn within the cycle, the vertex weight is shown next to the circle (red) and the edge weight is plotted next to the edge (blue).

file contains n lines. In each line the block ID of the corresponding vertex is given, i.e. line i contains the block ID of the vertex i (here the vertices are numbered from 0 to $n-1$). The block IDs are numbered consecutively from 0 to $k-1$. ParHip also has the option to store partitions in a binary format. These can also be converted using the toolbox program. In case of an edge partition, the file contains m lines in which line i constains the block ID of edge i (here the edges are numbered from 0 to $m-1$).

3.2.2 Node Separator

If the output is a node separator then the same format as used for a partition is used. However, in this case the nodes of the separator get the block ID k where as the other nodes maintain their original block id.

3.3 Troubleshooting

KaHIP should not crash! If KaHIP crashes it is mostly due to the following reasons: the provided graph contains self-loops or parallel edges, there exists a forward edge but the backward edge is missing or the forward and backward edges have different weights, or the number of vertices or edges specified does not match the number of vertices or edges provided in the file. Please use the *graphcheck* tool provided in our graph partitioning package to verify whether your graph has the right input format. If our graphcheck tool tells you that the graph that you provided has the correct format and KaHIP crashes anyway, please write us an email.

4 User Interface

KaHIP contains the following programs: kaffpa, kaffpaE, partition_to_vertex_separator, node_separator, graphchecker, evaluator, parhip, graph2binary, graph2binary_external, toolbox. To compile these programs you need to have Argtable, g++, OpenMPI and scons installed (we use argtable-2.10, g++-4.8.0, OpenMPI-1.4.1 and scons-1.2). Once you have that you can execute *compile.sh* in the main folder of the release. When the process is finished the binaries can be found in the folder *deploy*. We now explain the parameters of each of the programs briefly.

General Guide:

There are many programs contained already in the framework. The following table may helps you to find the right program for your use case:

Use case	Programs
Default Partitioning Problem	kaffpa, kaffpaE
Checking Graph for Correctness	graph_checker
Evaluate Partitioning Metrics	evaluator
Fast Sequential Partitioning, Mesh	kaffpa with preconfiguration set to fast
Good Sequential Partitioning, Mesh	kaffpa with preconfiguration set to eco
Very Good Sequential Partitioning, Mesh	kaffpa with preconfiguration set to strong
Fast Sequential Partitioning, Social	kaffpa with preconfiguration set to fastsocial
Good Sequential Partitioning, Social	kaffpa with preconfiguration set to ecosocial
Very Good Sequential Partitioning, Social	kaffpa with preconfiguration set to strongsocial
Mapping to Processor Networks	use the enable_mapping option
Highest Quality, Mesh	kaffpaE, use mpirun, large time limit
Highest Quality, Social	kaffpaE, use mpirun, large time limit, preconfig strongsocial
Parallel Partitioning	parhip, graph2binary, graph2binary_external, toolbox
Distributed Memory Parallel, Mesh	parhip, preconfigs ecomesh, fastmesh, ultrafastmesh
Distributed Memory Parallel, Social	parhip, preconfigs ecosocial, fastsocial, ultrafastsocial
Convert Metis to Binary	graph2binary, graph2binary_external
Evaluate and Convert Partitions	toolbox
Node Separators	partition_to_vertex_separator, node_separator
Edge Partitioning	edge_partitioning, distributed_edge_partitioning
Two Node Separators	node_separator
k-way Separators	use kaffpa to create k-way partition, and then partition_to_vertex_separator to create separator

4.1 KaFFPa

Description: This is the multilevel graph partitioning program.

Usage:

kaffpa   file --k=<int> [--help] [--seed=<int>] [--preconfiguration=variant] [--imbalance=<double>]
          [--time_limit=<double>] [--enforce_balance] [--input_partition=<string>]
          [--balance_edges] [--output_filename=<string>]
          [–enable_mapping --hierarchy_parameter_string=<string>
          --distance_parameter_string=<string> [--online_distances]]

Options:

file	Path to graph file that you want to partition.
--k=<int>	Number of blocks to partition the graph into.
--help	Print help.
--seed=<int>	Seed to use for the random number generator.
--preconfiguration=variant	Use a preconfiguration. (Default: eco) [strong| eco | fast | fastsocial| ecosocial| strongsocial]. Strong should be used if quality is paramount, eco if you need a good tradeoff between partition quality and running time, and fast if partitioning speed is in your focus. Configurations with a social in their name should be used for social networks and web graphs.
--imbalance=<double>	Desired balance. Default: 3 (%).
--time_limit=<double>	Time limit in seconds s. The default value is set to 0s, i.e. one partitioner call will be made. If you set a time limit t, kaffpa will repeatedly call the multilevel method until the time limit is reached and return the best solution found.
--enforce_balance	Use this option only on graphs without vertex weights. If this option is enabled, kaffpa guarantees that the output partition is feasible, i.e. fulfills the specified balance constraint.
--balance_edges	Use this option to balance the edges among the blocks as well as the nodes. In this case node weights are set to $c(v) + \deg_\omega(v)$ for a node $v \in V$ and the balance constraint is adapted accordingly.
--input_partition=<string>	You can specify an input partition. If you do so, KaFFPa will try to improve it.
--enable_mapping	Enable mapping algorithms to map quotient graph onto processor graph defined by hierarchy and distance options. (Default: disabled)
--hierarchy_parameter_string=<string>	Specify as 4:8:8 for 4 cores per PE, 8 PEs per rack, ... and so forth.
--distance_parameter_string=<string>	Specify as 1:10:100 if cores on the same chip have distance 1, PEs in the same rack have distance 10, ... and so forth.
--online_distances	Do not store processor distances in a matrix, but do recomputation. (Default: disabled)
--output_filename=<string>	Specify the output filename (default tmppartition$k).

4.2 KaFFPaE / KaBaPE

Description: This is the distributed evolutionary algorithm to tackle the graph partitioning problem. It includes also the perfectly balance case $\epsilon = 0$.

Usage:

mpirun -n P kaffpaE   file --k=<int> [--help] [--seed=<int>] [--preconfiguration=variant] [--imbalance=<double>]
                [--time_limit=<double>] [--mh_enable_quickstart] [--mh_optimize_communication_volume]
                [--mh_enable_kabapE] [--mh_enable_tabu_search] [--kabaE_internal_bal=<double>]
                [--input_partition=<string>] [--balance_edges] [--output_filename=<string>]

Options:

P	Number of processes to use.
file	Path to graph file that you want to partition.
--k=<int>	Number of blocks to partition the graph into.
--help	Print help.
--seed=<int>	Seed to use for the random number generator.
--preconfiguration=variant	Use a preconfiguration. (Default: strong) [strong	eco	fast	fastsocial	ecosocial	strongsocial]. Strong should be used if quality is paramount, eco if you need a good tradeoff between partition quality and running time, and fast if partitioning speed is in your focus. Configurations with a social in their name should be used for social networks and web graphs.
--imbalance=<double>	Desired balance. Default: 3 (%).
--time_limit=<double>	Time limit in seconds s. The default value is set to 0s, i.e. one partitioner call will be made by each PE. In order to use combine operations you *have to* set a time limit $t > 0$. kaffpaE will return the best solution after the time limit is reached. A time limit $t = 0$ means that the algorithm will only create the initial population.
--mh_enable_quickstart	Enables the quickstart option. In this case each PE creates a few partitions and these partitions are distributed among the PEs.
--mh_optimize_communication_volume	Modifies the fitness function of the evolutionary algorithm so that communication volume is optimized.
--mh_enable_kabapE	Enables the combine operator of *KaBaPE*.
--mh_enable_tabu_search	Enables our version of combine operation by block matching.
--kabaE_internal_bal=<double>	The internal balance parameter for KaBaPE (Default: 0.01) (1 %)
--balance_edges	Use this option to balance the edges among the blocks as well as the nodes. In this case node weights are set to $c(v) + \deg_{\omega}(v)$ for a node $v \in V$ and the balance constraint is adapted accordingly.
--input_partition=<string>	You can specify an input partition. If you do so, KaFFPaE will try to improve it.
--output_filename=<string>	Specify the output filename (default tmppartition$k).

4.3 ParHIP – Parallel High Quality Partitioning

4.3.1 ParHIP

Description: This is the distributed memory parallel program to compute a partition.

Usage:
  mpirun -n P parhip   file --k=<int> [--help] [--seed=<int>] [--preconfiguration=variant] [--imbalance=<int>]
                    [--vertex_degree_weights] [--save_partition] [--save_partition_binary]

Options:

P	Number of processes to use.
file	Path to graph file to partition. (Either Metis format or binary format)
--seed=<int>	Seed to use for the PRNG.
--k=<int>	Number of blocks to partition the graph.
--help	Print help.
--imbalance=<int>	Desired balance. Default: 3 (%).
--preconfiguration=variant	Use a preconfiguration. (Default: fast) [ecosocial	fastsocial	ultrafastsocial	ecomesh	fastmesh	ultrafastmesh].
--vertex_degree_weights	Use 1+deg(v) as vertex weights.
--save_partition	Enable this tag if you want to store the partition to disk.
--save_partition_binary	Enable this tag if you want to store the partition to disk in a binary format.

4.3.2 Graph Conversion

Description: This is the program to convert a Metis graph format into the binary graph format. The program ending with _external does the task in external memory.

Usage:
  graph2binary[_external]   metisfile outputfilename

4.3.3 Evaluation

Description: This is the toolbox program to convert partitions and to evaluate them.

Usage:
  mpirun -n P toolbox   file --k=<int> [--help] =<int>] --input_partition=<string>
                    [--save_partition] [--save_partition_binary] [--evaluate]

Options:

P	Number of processes to use.
file	Path to graph file to partition. (Either Metis format or binary format)
--k=<int>	Number of blocks to graph is partitioned in.
--help	Print help.
--input_partition=file	Path to partition file to convert.
--save_partition	Enable this tag if you want to store the partition to disk.
--save_partition_binary	Enable this tag if you want to store the partition to disk in a binary format.
--evaluate	Enable this tag the partition to be evaluated.

4.4 Node Separators

4.4.1 Partition Converter

Description: This is the program that computes a k-way vertex separator given a k-way partition of the graph. Use this approach if $k > 2$.

Usage:
partition_to_vertex_separator   file --k=<int> --input_partition=<string> [--help] [--seed=<int>] [--output_filename]

Options:

file	Path to the graph file.
--k=<int>	Number of blocks the graph is partitioned in by using the input partition.
--input_partition=<string>	Input partition to compute the vertex separator from.
--help	Print help.
--seed=<int>	Seed to use for the random number generator.
--output_filename=<string>	Specify the output filename (default tmpseparator).

4.4.2 Biseparators

Description: This is the program that computes a 2-way vertex separator. Use this approach if $k = 2$.

Usage:
node_separator   file [--seed=<int>] [--preconfiguration=variant] [--help] [--output_filename]

Options:

file	Path to the graph file.
--help	Print help.
--imbalance=<double>	Desired balance. Default: 20 (%).
--seed=<int>	Seed to use for the random number generator.
--preconfiguration=variant	Use a preconfiguration. (Default: strong) [strong | eco | fast | fastsocial | ecosocial | strongsocial]. Strong should be used if quality is paramount, eco if you need a good tradeoff between partition quality and running time, and fast if partitioning speed is in your focus. Configurations with a social in their name should be used for social networks and web graphs (they use a different kind of coarsening scheme).
--output_filename=<string>	Specify the output filename (default tmpseparator).

4.5 Edge Partitioning

Description: This is the edge partitioning program.

Usage:

edge_partitioning   file --k=<int> [--help] [--seed=<int>] --preconfiguration=variant [--imbalance=<double>]
                     [--infinity=<int>] [--output_filename=<string>]

Options:

file	Path to graph file that you want to partition.
--k=<int>	Number of blocks to partition the graph into.
--help	Print help.
--seed=<int>	Seed to use for the random number generator.
--preconfiguration=variant	Use a preconfiguration. (Default: eco) [strong| eco | fast | fastsocial| ecosocial| strongsocial]. Strong should be used if quality is paramount, eco if you need a good tradeoff between partition quality and running time, and fast if partitioning speed is in your focus. Configurations with a social in their name should be used for social networks and web graphs.
--imbalance=<double>	Desired balance. Default: 3 (%).
--infinity=<int>	Infinity edge weight used in the SPAC model. Default: 1000.
--output_filename=<string>	Specify the output filename (default tmpedgepartition$k).

4.6 Distributed Edge Partitioning

Description: This is the distributed memory parallel edge partitioning program.

Usage:

mpirun -n N distributed_edge_partitioning   file --k=<int> [--help] [--seed=<int>] --preconfiguration=variant
                                             [--infinity=<int>] [--output_filename=<string>] [--imbalance=<double>]

Options:

file	Path to graph file that you want to partition.
--k=<int>	Number of blocks to partition the graph into.
--help	Print help.
--seed=<int>	Seed to use for the random number generator.
--preconfiguration=variant	Use a preconfiguration. (Default: eco) [strong| eco | fast | fastsocial| ecosocial| strongsocial]. Strong should be used if quality is paramount, eco if you need a good tradeoff between partition quality and running time, and fast if partitioning speed is in your focus. Configurations with a social in their name should be used for social networks and web graphs.
--imbalance=<double>	Desired balance. Default: 3 (%).
--infinity=<int>	Infinity edge weight used in the SPAC model. Default: 1000000.
--output_filename=<string>	Specify the output filename (default tmpedgepartition$k).

4.7   Label Propagation

Description: This is the program that a size-constrained label propagation clustering.

Usage:
label_propagation   file [--cluster_upperbound=<int>] [--label_propagation_iterations] [--help] [--seed=<int>]
                     [--output_filename]

Options:

file	Path to the graph file.
--cluster_upperbound=<int>	Set a size-constraint on the size of a cluster. For example, specifying a value of 10 means that each cluster can have at most 10 vertices (or a weight of 10 if the graph is weighted). By default there is no size-constraint, so that each node can potentially be in on cluster.
--label_propagation_iterations=<int>	Set the number of label propagation iterations. The default value is 10.
--help	Print help.
--seed=<int>	Seed to use for the random number generator.
--output_filename=<string>	Specify the output filename (default tmpclustering).

4.8   Graph Format Checker

Description: This program checks if the graph specified in a given file is valid.

Usage:
graphchecker   file

Options:

file	Path to the graph file.

5   Library

Some of the programs above can be directly accessed in C/C++ by using the library that we provide, i.e. the graph partitioners KaFFPaFast, KaFFPaEco and KaFFPaStrong (more will follow) can be called using this library. In this section we describe how the library can be used, the methods can be accessed and describe the parameters of these functions. The functions described in this section can be found in the file *interface/kaHIP_interface.h*. **An example program that calls and links the library can be found in *misc/example_library_call/*.** To try, go into this directory, run *./compile.sh* and then the program *./interfactest*.

5.1   Graph Data Structure

To make it easy for you to try KaHIP, we use a more or less similar interface to that provided by Metis. Most importantly, we use the same graph data structure. The graph data structure is an adjacency structure of the graph which includes weights for the vertices and edges if there are any. The adjacency structure is a compressed sparse row (CSR) format. We closely follow the description of the Metis 4.0 user guide:

The CSR format is a widely used scheme for storing sparse graphs. In this format the adjacency structure of a graph with n vertices and m edges is represented using two arrays *xadj* and *adjncy*. The *xadj* array is of size $n + 1$ whereas the *adjncy* array is of size $2m$ (this is because for each edge between vertices v and u we store both a forward edge (v, u) and a backward edge (u, v)).

The adjacency structure of the graph is stored as follows. First of all, the vertex numbering starts from 0. The adjacency list of vertex i is stored in the array *adjncy* starting at index *xadj[i]* and ending at (but not including) index *xadj[i + 1]* (i.e. *adjncy[xadj[i]]* through and including *adjncy[xadj[i + 1]−1]*). That is, for each vertex i, its adjacency list is stored in consecutive locations in the array *adjncy*, and the array *xadj* is used to point to where it begins and where it ends.

The weights of the vertices (if any) are stored in an additional array called *vwgt*. If the graph has n vertices then the array contains n elements, and *vwgt[i]* will store the weight of the ith vertex. The vertex-weights must be integers greater or equal to zero. If all the vertices of the graph have the same weight (e.g. the graph is unweighted), then the array *vwgt* can be set to NULL.

The weights of the edges (if any) are stored in additional array called adjwgt. This array contains $2m$ elements, and the weight of edge *adjncy*[j] is stored at location *adjwgt*[j]. The edge-weights must be integers greater than zero. The weight of a forward edge (u, v) has to be equal to the weight of the backward edge (v, u) and parallel edges as well as self-loops are not allowed. If all the edges of the graph have the same weight (e.g. the graph is unweighted), then the array *adjwgt* can be set to NULL.
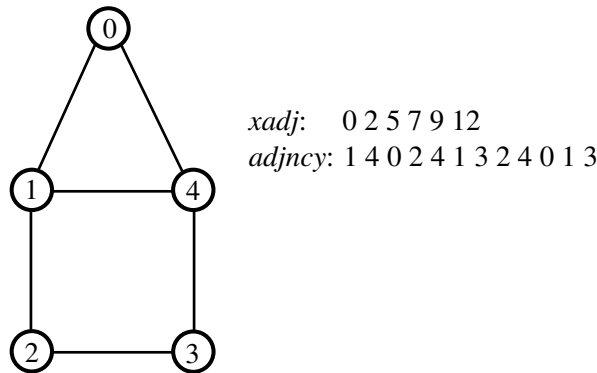

xadj: 0 2 5 7 9 12
adjncy: 1 4 0 2 4 1 3 2 4 0 1 3

Figure 4: An example unweighted graph with its graph data structure.

5.2 Functions

Currently we provide two functions, e.g. kaffpa and node_separator. The function kaffpa corresponds to the mulilevel partitioner KaFFPa and the function node_separator used KaFFPa to compute a partition and from that a node separator is derived.

Main Partitioner Call

void kaffpa(int* n, int* *vwgt*, int* *xadj*, int* *adjcwgt*, int* *adjncy*,
int* *nparts*, double* *imbalance*, bool *suppress_output*, int *seed*, int *mode*,
int* *edgecut*, int* *part*);

Parameters

n	Number of vertices of the graph.
vwgt	This is the vertex weight array described in Section 5.1. The array should have size n. If your graph does not have vertex weight you can use a null pointer as an argument.
xadj	This is the xadj array described in Section 5.1 which holds the pointers to the adjacency lists of the vertices. The array should have size $n + 1$.
adjcwgt	This is the adjacency weight array described in Section 5.1 which holds the weights of the edges if they exists. The array should have size $2m$. If your graph does not have edge weights you can use a null pointer as an argument.
adjncy	This is the adjacency array described in Section 5.1 which holds the adjacency lists of the vertices. The array should have size $2m$.
nparts	This parameter specifies the number of blocks you want the graph to be partitioned in.
imbalance	This parameter controls the amount of imbalance that is allowed. For example, setting it to 0.03 specifies an imbalance of 3% which means on unweighted graphs that each block has to fulfill the constraint $|V_i| \leq (1 + 0.03)|V|/k$.
suppress_output	If this option is enabled then no output of the partitioning library is printed to stdout.
seed	Seed to use for the random number generator.
mode	One out of FAST, ECO, STRONG, FASTSOCIAL, ECOSOCIAL, STRONGSOCIAL. Configuration names correpond to the default configuration names of the multilevel partitioner KaFFPa.
edgecut	This is an *output* parameter. It represents the edge cut of the computed partition.
part	This is an *output* parameter. It has to be an already allocated array of size n. After the function call this array contains the information of the blocks of the vertices, i.e. the block of the ith vertex is given in *part*[i].

Node+Edge Balanced Partitioner Call

This function will try to balance nodes *and* edges among the blocks. Use this function if you want to use KaFFPa for applications in which workload is also associated with edges, e.g. sparse matrix vector multiplications or in graph processing frameworks.

void kaffpa_balance_NE(int* n, int* *vwgt*, int* *xadj*, int* *adjcwgt*, int* *adjncy*,
                     int* *nparts*, double* *imbalance*, bool *suppress_output*, int *seed*, int *mode*,
                     int* *edgecut*, int* *part*);

Parameters

n	Number of vertices of the graph.
vwgt	This is the vertex weight array described in Section 5.1. The array should have size n. If your graph does not have vertex weight you can use a null pointer as an argument.
xadj	This is the xadj array described in Section 5.1 which holds the pointers to the adjacency lists of the vertices. The array should have size $n + 1$.
adjcwgt	This is the adjacency weight array described in Section 5.1 which holds the weights of the edges if they exists. The array should have size $2m$. If your graph does not have edge weights you can use a null pointer as an argument.
adjncy	This is the adjacency array described in Section 5.1 which holds the adjacency lists of the vertices. The array should have size $2m$.
nparts	This parameter specifies the number of blocks you want the graph to be partitioned in.
imbalance	This parameter controls the amount of imbalance that is allowed. For example, setting it to 0.03 specifies an imbalance of 3% which means on unweighted graphs that each block has to fulfill the constraint $|V_i| \leq (1 + 0.03)|V|/k$.
suppress_output	If this option is enabled then no output of the partitioning library is printed to stdout.
seed	Seed to use for the random number generator.
mode	One out of FAST, ECO, STRONG, FASTSOCIAL, ECOSOCIAL, STRONGSOCIAL. Configuration names correpond to the default configuration names of the multilevel partitioner KaFFPa.
edgecut	This is an *output* parameter. It represents the edge cut of the computed partition.
part	This is an *output* parameter. It has to be an already allocated array of size n. After the function call this array contains the information of the blocks of the vertices, i.e. the block of the ith vertex is given in *part*[i].

Node Separator

void node_separator(int* n, int* *vwgt*, int* *xadj*, int* *adjcwgt*, int* *adjncy*,
                    int* *nparts*, double* *imbalance*, bool *suppress_output*, int *seed*, int *mode*,
                    int* *num_separator_vertices*, int** *separator*);

Parameters

n	Number of vertices of the graph.
vwgt	This is the vertex weight array described in Section 5.1. The array should have size n. If your graph does not have vertex weight you can use a null pointer as an argument.
xadj	This is the xadj array described in Section 5.1 which holds the pointers to the adjacency lists of the vertices. The array should have size $n + 1$.
adjcwgt	This is the adjacency weight array described in Section 5.1 which holds the weights of the edges if they exists. The array should have size $2m$. If your graph does not have edge weights you can use a null pointer as an argument.
adjncy	This is the adjacency array described in Section 5.1 which holds the adjacency lists of the vertices. The array should have size $2m$.
nparts	This parameter specifies the number of blocks you want the graph to be partitioned in as a base to compute a separator from. If the size of the separator is your objective, we recommend to set this parameter to 2.
imbalance	This parameter controls the amount of imbalance that is allowed. For example, setting it to 0.03 specifies an imbalance of 3% which means on unweighted graphs that each block has to fulfill the constraint $|V_i| \leq (1 + 0.03)|V|/k$.
suppress_output	If this option is enabled then no output of the partitioning library is printed to stdout.
seed	Seed to use for the random number generator.
mode	One out of FAST, ECO, STRONG, FASTSOCIAL, ECOSOCIAL, STRONGSOCIAL. Configuration names correpond to the default configuration names of the multilevel partitioner KaFFPa.
num_separator_vertices	This is an *output* parameter. It represents the number of separator vertices, i.e. the size of the array separator.
separator	This is an *output* parameter. After the function call this array contains the ids of the separator vertices.

Using KaHIP within Java Projects

KaHIP can be used within Java project via JNI. A small example on how to do this is provided in the folder *misc/java_jni_wrapper/* of the project. To run the example perform the following commands in the main folder:

./compile.sh
cd misc/java_jni_wrapper
./makeWrapper.sh
java KaHIPWrapper

References

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

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

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

[4] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable Matrix Computations on Large Scale-Free Graphs Using 2D Graph Partitioning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'13. ACM, 2013.

[5] A. Buluç and K. Madduri. Graph partitioning for scalable distributed graph computations. In *Graph Partitioning and Graph Clustering*, pages 83–102, 2012.

[6] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering – Selected Topics,* to appear, 2014.

[7] 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 of *LNCS*, pages 75–86. Springer, 1996.

[8] J. Cong and J. Shinnerl. *Multilevel Optimization in VLSICAD*. Springer, 2003.

[9] 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.

[10] D. Delling and R. F. Werneck. Faster Customization of Road Networks. In *12th Symposium on Experimental Algorithms*, volume 7933 of *LNCS*, pages 30–42. Springer, 2013.

[11] 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.

[12] 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.

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

[14] R. Glantz, H. Meyerhenke, and C. Schulz. Tree-based Coarsening and Partitioning of Complex Networks. Technical Report ArXiv:1402.2782, 2014.

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

[16] 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.

[17] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu. *VLSI Physical Design - From Graph Partitioning to Timing Closure.* Springer, 2011.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] J. Maue, P. Sanders, and D. Matijevic. Goal Directed Shortest Path Queries Using Precomputed Cluster Distances. *ACM Journal of Experimental Algorithmics*, 2007.

[23] H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-constrained Clustering. Technical Report ArXiv:1402.3281, 2014.

[24] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 1055–1064, 2015.

[25] 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.

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

[27] 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.

[28] 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.

[29] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, page 22, 2013.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Peter Sanders and Christian Schulz. Advanced Multilevel Node Separator Algorithms. In *Proceedings of the 15th International Symposium on Experimental Algorithms*, 2016.

[35] Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Darren Strash. Scalable Edge Partitioning. Technical report, 2019, to appear.

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

[37] C. Schulz. *Hiqh Quality Graph Partititioning*. PhD thesis, Karlsruhe Institute of Technology, 2013.

[38] C. Schulz and J. L. Träff. Better Process Mapping and Sparse Quadratic Assignment. Technical report, Karlsruhe Institute of Technology, University of Vienna and Technical University Vienna, 2017.

[39] F. Schulz, D. Wagner, and C. D. Zaroliagis. Using Multi-Level Graphs for Timetable Information. In *4th Workshop on Algorithm Engineering and Experiments*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

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

[41] 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.

[42] M. Zhou, O. Sahni, K. D Devine, M. S. Shephard, and K. E Jansen. Controlling Unstructured Mesh Partitions for Massively Parallel Simulations. *SIAM Journal on Scientific Computing*, 32(6):3201–3227, 2010.