Engineering Multilevel Graph Partitioning Algorithms*

Peter Sanders, Christian Schulz

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
{sanders,christian.schulz}@kit.edu

Abstract. We present a multi-level graph partitioning algorithm using novel local improvement algorithms and global search strategies transferred from multigrid linear solvers. Local improvement algorithms are based on max-flow min-cut computations and more localized FM searches. By combining these techniques, we obtain an algorithm that is fast on the one hand and on the other hand is able to improve the best known partitioning results for many inputs. For example, in Walshaw's well known benchmark tables we achieve 317 improvements for the tables at 1%, 3% and 5% imbalance. Moreover, in 118 out of the 295 remaining cases we have been able to reproduce the best cut in this benchmark.

1 Introduction

Graph partitioning is a common technique in computer science, engineering, and related fields. For example, good partitionings of unstructured graphs are very valuable for parallel computing. In this area, graph partitioning is mostly used to partition the underlying graph model of computation and communication. Roughly speaking, vertices in this graph represent computation units and edges denote 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 paper we focus 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.

A successful heuristic for partitioning large graphs is the *multilevel graph partitioning* (MGP) approach depicted in Figure 1 where the graph is recursively *contracted* to achieve 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 refinement* method is used to improve the partitioning induced by the coarser level.

Although several successful multilevel partitioners have been developed in the last 13 years, we had the impression that certain aspects of the method are not well understood. We therefore have built our own graph partitioner KaPPa [4] (Karlsruhe Parallel Partitioner) with 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 MGP on trial. Our focus is on solution quality and sequential speed for large graphs. This paper reports the first results we have obtained which relate to the local improvement methods and overall

* This paper is a short version of the technical report [10].

search strategies. We obtain a system that can be configured to either achieve the best known partitions for many standard benchmark instances or to be the fastest available system for large graphs while still improving partitioning quality compared to the previous fastest system. We begin in Section 2 by introducing basic concepts. After shortly presenting Related Work in Section 3 we continue describing novel local improvement methods in Section 4. This is followed by Section 5 where we present new global search methods. Section 6 is a summary of extensive experiments done to tune the algorithm and evaluate its performance. We have implemented these techniques in the graph partitioner KaFFPa (Karlsruhe Fast Flow Partitioner) which is written in C++. Experiments reported in Section 6 indicate that KaFFPa scales well to large networks and is able to compute partitions of very high quality.

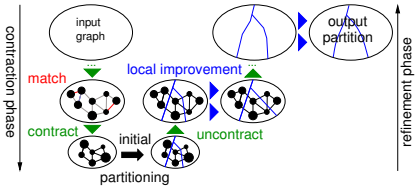

Fig. 1. Multilevel graph partitioning.

2 Preliminaries

2.1 Basic concepts

Consider an undirected graph $G = (V, E, c, \omega)$ 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 $\omega(E') := \sum_{e \in E'} \omega(e)$. $\Gamma(v) := \{u : \{v, u\} \in E\}$ denotes the neighbors of v. 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 $\forall i \in 1..k : c(V_i) \leq L_{\max} := (1 + \epsilon)c(V)/k + \max_{v \in V} c(v)$ for some parameter ϵ. The last term in this equation arises because each node is atomic and therefore a deviation of the heaviest node has to be allowed. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. A vertice $v \in V_i$ that has a neighbor $w \in V_j, i \neq j$, is a boundary vertice. An abstract view of the partitioned graph is the so called *quotient graph*, where vertices represent blocks and edges are induced by connectivity between blocks. An example can be found in Figure 2. By default, our initial inputs will have unit edge and node weights. However, even those will be translated into weighted problems in the course of the algorithm.

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. *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 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 $\{x, w\}$, we insert a single edge with $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$. *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. In the *contraction* (coarsening) phase, we 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. 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. In KaPPa [4] we have shown that the rating function $\text{expansion}^{*2}(\{u,v\}) := \omega(\{u,v\})^2/c(u)c(v)$ works best among other edge rating functions. Contraction is stopped when the graph is small enough to be directly partitioned using some expensive other algorithm. In the *refinement* (or uncoarsening) phase, the matchings are iteratively uncontracted. After uncontracting a matching, the refinement algorithm moves nodes between blocks in order to improve the cut size or balance. The succession of movements is based on priorities called *gain*, i.e., the decrease in edge cut when the node is moved to the other side. There are two main types of local search heuristics: k-*way* and *two-way* local search. k-way local search is allowed to move a node to an arbitrary block whereas two-way local search is restriced to move nodes only between a pair of blocks. The latter is usually applied to all pairs of blocks sharing a non-empty boundary. 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 refinement will quickly find a good solution. KaFFPa *makes use of techniques proposed* in KaPPa [4] and KaSPar [7]. These techniques concern coarsening (edge ratings, global paths algorithm (GPA) as matching algorithm), initial partitioning (using Scotch) and a flexible stopping criterion for local search. They are described in the TR [10].

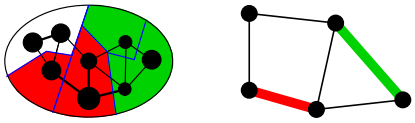

Fig. 2. A graph partitioned into five blocks and its quotient graph $\mathcal{Q}$. Two pairs of blocks are highlighted in red and green.

3 Related Work

There has been a huge amount of research on graph partitioning so that we refer the reader to [3,15] for more material. All general purpose methods that are able to obtain good partitions for large real world graphs are based on the multilevel principle outlined in Section 2. The basic idea can be traced back to multigrid solvers for solving systems of linear equations [12] but more recent practical methods are based on mostly graph theoretic aspects in particular edge contraction and local search. Well known software packages based on this approach include, Jostle [15], Metis [11], and Scotch [8]. KaSPar [7] is a graph partitioner based on the central idea to (un)contract only a single edge between two levels. KaPPa [4] is a "classical" matching based MGP algorithm designed for scalable parallel execution. DiBaP [6] is a multi-level graph partitioning package where local improvement is based on diffusion. MQI [5] and Improve [1] are flow-based methods for improving graph cuts when cut quality is measured by quotient-style metrics such as *expansion* or *conductance*. This approach is only feasible for $k = 2$. Improve uses several minimum cut computations to improve the *quotient cut* score of a proposed partition.

The concept of *iterated multilevel algorithms* was introduced by [13]. 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. This ensures increased quality of the partition if the refinement algorithms guarantees no worsening.

4 Local Improvement

Recall that once a matching is uncontracted a local improvement method tries to reduce the cut size of the projected partition. We now present two novel local improvement methods. The first method is based on max-flow min-cut computations between pairs of blocks, i.e., improving a given 2-partition. Roughly speaking, this improvement method is then applied between all pairs of blocks that share a non-empty boundary. In contrast to previous flow-based methods we improve the edge cut whereas previous systems improve conductance or expansion. The second method which is described in Section 4.2 is called multi-try FM. Roughly speaking, a k-way local search initialized with a *single* boundary node is repeatedly started. Previous methods are initialized with *all* boundary nodes. At the end of the section we show how pairwise refinements can be scheduled and how multi-try FM local search can be incorporated with this scheduling.

4.1 Max-Flow Min-Cut Computations for Local Improvement

We now explain how flows can be employed to improve a partition of *two blocks* V_1, V_2 without violating the balance constraint. That yields a local improvement algorithm.
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 subgraph* G' is the node induced subgraph $G[B]$ 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. All of these new edges get the edge weight ∞. Note that the additional edges are directed. G' has the *cut property* if each (s,t)-min-cut induces a cut within the balance constraint in G.

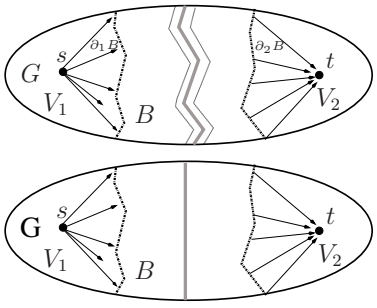

Fig. 3. The construction of a feasible flow problem G' is shown on the top and an improved cut within the balance constraint in G is shown on the bottom.

The basic idea is to construct a B induced subgraph G' 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 (BFS) we can find such a set B. Each node touched during these searches belongs to B. The first BFS is done in the subgraph of G induced by V_1. It is initialized with the boundary nodes of V_1. As soon as the weight of the area found by this BFS would exceed $(1 + \epsilon)c(V)/2 - w(V_1)$, we stop the BFS. The second BFS is done for V_2 in an analogous fashion. The constructed subgraph

G' has the cut property since the worst case new weight of V_2 is lower or equal to $w(V_2) + (1+\epsilon)c(V)/2 - w(V_2) = (1+\epsilon)c(V)/2$. Indeed the same holds for the worst case new weight of V_1.

There are multiple ways to improve this method. First, if we found an improved cut, we can apply this method again since the initial boundary has changed, i.e., the set B will also change. Second, we can adaptively control the size of the set B found by the BFS. This enables us to search for cuts that fulfill our balance constraint in a larger subgraph (say $\epsilon' = \alpha\epsilon$ for some parameter α). To be more precise if the found min-cut in G' for ϵ' 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*.

Most Balanced Minimum Cuts Picard and Queyranne have been able to show that *one* (s,t)-max-flow contains information about *all* minimum (s,t)-cuts in the graph. Thus the idea to search for feasable cuts in larger subgraphs becomes even more attractive. Roughly speaking, we present a heuristic that, given a max-flow, selects min-cuts with better balance in G. First we need a few notations. For a graph $G = (V, E)$ a set $C \subseteq V$ is a *closed vertex set* iff for all vertices $u, v \in V$, the conditions $u \in C$ and $(u, v) \in E$ imply $v \in C$. An example can be found in Figure 4.

Lemma 1 (Picard and Queyranne [9]). *There is a 1-1 correspondence between the minimum (s,t)-cuts of a graph and the closed vertex sets containing s in the residual graph of a maximum (s,t)-flow.*

For a given closed vertex 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 vertex sets remain the same. To enumerate all minimum cuts of a graph [9] a further reduced graph is computed which is described below. However, the problem of finding the most balanced minimum cut is NP-hard [9].

We now define how the representation of the residual graph can be made more compact [9] and then explain our heuristic to obtain closed vertex sets on this graph in order to select min-cuts with better balance. First we take a maximum (s,t)-flow and compute the strongly connected components of its residual graph. We make the representation more compact by contracting the components and refer to it as *minimum cut representation*. The reduction is possible since two vertices that lie on a cycle have to be in the same closed vertex set of the residual graph. The result is a weighted, directed and acyclic graph (DAG). Note that each closed vertex set of the minimum cut representation induces a minimum cut as well. On this graph we search for closed vertex sets (containing the component S that contains the source) since they still induce (s,t)-min-cuts in the original graph. This is done by using the following heuristic which is repeated a few times. The main idea is that a

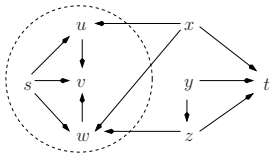

Fig. 4. The set $\{s, u, v, w\}$ is a closed vertex set.

topological order yields complements of closed vertex sets quite easily. Therefore, we first compute a random topological order using a randomized DFS[1].

We sweep through this topological order and sequentially add the components to the complement of the closed vertex set. By sweeping through the topological order we compute closed vertex sets each inducing a min-cut having a different balance. We stop when we have reached the best balanced minimum cut induced through this particular topological order. The closed vertex set with the best balance occurred for different topological orders is returned. Note that this procedure may still finds 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*.

4.2 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 are eligible for movement at the beginning. Our method is repeatedly initialized with a *single* boundary node. More details about k-way methods can be found in the TR [10].

Multi-try FM is organized in rounds. In each round we put *all* boundary nodes of the current block pair into a todo list T. Subsequently, we begin a k-way local search starting with a *single* random node v of T if it is still a boundary node. Note that the difference to the global k-way search is in the initialisation of the search. The local search is only started from v if it was not touched by a previous localized k-way search in this round. Either way, the node is removed from the todo list. A localized k-way search is not allowed to move a node that has been touched in a previous run. This assures that at most n nodes are touched during a round of the algorithm. The algorithm uses the adaptive stopping criterion from KaSPar [7].

4.3 Scheduling Quotient Graph Refinement

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 then is organized in rounds. In each round, the algorithm refines 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 a refinement of adjacent pairs of blocks can be both, two-way local search and local improvement by using flow, depending on the configuration. After each pair-wise refinement a multi-try FM search (k-way) is started. The todo list T is initialized with all boundaries 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.

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

5 Global Search

Iterated Multilevel Algorithms (V-cycles) were introduced by [13]. The main idea is to iterate coarsening and refinement several times using different seeds for random tiebreaking. Edges between blocks are not contracted as soon as the graph is partitioned. Thus a given partition can be used as initial partition of the coarsest graph. This ensures increased quality if the refinement algorithm guarantees no worsening. In multigrid linear solvers Full-Multigrid methods are preferable to simple V-cycles [2]. 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 during 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. In most cases the initial partitioner is not able to improve a given partition from scratch or even to find this partition. Therefore no further initial partitioning is used as soon as the graph is partitioned. Experiments in Section 6 show that all cycle variants are more efficient than simple restarts of the algorithm. In order to bound the execution time we introduce a level split parameter d such that further recursive calls are only performed every d'th level. We go into more detail after we have analysed the run time of the global search strategies.

Analysis. We now roughly analyse the run 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 1. *If the time for coarsening and refinement is $T_{cr}(n) := bn$ and a constant shrink factor $a \in [1/2, 1)$ is given. Then:*

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

$$T_{F,d}(n) \leq \frac{1}{1-a^d} T_V(n) \tag{2}$$

where T_V is the time for a single V-cycle and $T_{W,d}$,$T_{F,d}$ are the time for a W-cycle and F-cycle with level split parameter d.

The proof can be found in the TR [10]. For the optimistic assumption that $a = 1/2$ and $d = 1$, a F-cycle is only twice as expensive as a single V-cycle. If we use the same parameters for a W-cycle we get a factor $\log n$ asymptotic larger execution times. However in practice the shrink factor is usually worse than $1/2$. That yields an even larger asymptotic run time for the W-cycle (since for $d = 1$ we have $2a > 1$). Therefore, in order to bound the run time of the W-cycle the choice of the level split parameter d is crucial. Our default value for d for W- and F-cycles is 2.

6 Experiments

Implementation / Instances / System. We have implemented the algorithm decribed above using C++. We report experiments on two suites of instances (medium sized graphs used in Subsection 6.1-6.3 and large sized graphs used in Subsection 6.4). The medium sized testset contains 20 graphs having between thirteen thousand and five houndred thousand vertices. The large sized testset contains 12 graphs having between seventy thousand and eighteen millon vertices. They are the same as in [7] and can be found the TR [10]. All implementation details, system information and more information about the graphs can be found in the TR [10].

Configuring the Algorithm. We currently define three configurations of our algorithm: Strong, Eco and Fast. The strong configuration is 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 available system for large graphs while still improving partitioning quality to the previous fastest system. All configurations use the FM algorithm. The strong configuration further employs Flows, Multi-Try FM and Global Search. The eco configuration also employs Flows. For a full description of the configurations we refer the reader to the TR [10].

Experiment Description. We performed two types of experiments namely normal tests and tests for effectiveness. Both are described below.

Normal Tests: Here we perform 10 repetitions for the small networks and 5 repetitions for the other. We report the arithmetic 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: Here each algorithm configuration has the same time for computing a partition. Therefore, for each graph and k each configuration is executed once and we remember the largest execution time t that occurred. Now each algorithm gets time $3t$ to compute a good partition, i.e., taking the best partition out of repeated runs. If a variant can perform a next run depends on the remaining time, i.e., we flip a coin with corresponding probabilities such that the expected time over multiple runs is $3t$. This is repeated 5 times. The final score is computed as above using these values. Note that on the middlesized testset the final eff. score of an algorithm configuration is the result of at least $1\,800$ algorithm runs.

6.1 Insights about Flows

We now evaluate max-flow min-cut based improvement algorithms. First we define a basic two-way FM configuration to compare with. It uses the GPA algorithm as a matching algorithm and performs five initial partitioning attempts using Scotch as initial

[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 run time **t**.

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 KaFFPa Strong. Note that during this test our main focus is the evaluation of flows and therefore we *don't* use k-way refinement or multi-try FM search.

To evaluate the performance of specific algorithmic components the basic configuration is extended by specific algorithms. A configuration that uses Flow, FM and the most balanced cut heuristics (MB) will be indicated by (+F, +MB, +FM). In Table 1 we see that by Flow on its own we obtain cuts and run times which are worse than those of the basic two-way FM configuration. The results improve in terms of quality and run time if we enable the most balanced minimum cut heuristic.

Variant	(+F, -MB, -FM)			(+F, +MB, -FM)			(+F, -MB, +FM)			(+F, +MB, +FM)		
α'	Avg.	Best.	$t[s]$	Avg.	Best.	$t[s]$	Avg.	Best.	$t[s]$	Avg.	Best.	$t[s]$
16	-1.88	-1.28	4.17	0.81	0.35	3.92	6.14	5.44	4.30	7.21	6.06	5.01
8	-2.30	-1.86	2.11	0.41	-0.14	2.07	5.99	5.40	2.41	7.06	5.87	2.72
4	-4.86	-3.78	1.24	-2.20	-2.80	1.29	5.27	4.70	1.62	6.21	5.36	1.76
2	-11.86	-10.35	0.90	-9.16	-8.24	0.96	3.66	3.37	1.31	4.17	3.82	1.39
1	-19.58	-18.26	0.76	-17.09	-16.39	0.80	1.64	1.68	1.19	1.74	1.75	1.22
Ref.	(-F, -MB, +FM)			2 974	2 851	1.13						

Table 1. The final score of different algorithm configurations. α' is the flow region upper bound factor. All average and best cut values are improvements relative to the basic configuration in %.

In some cases, flows and flows with the MB heuristic are not able to produce results that are comparable to the basic two-way FM configuration. Perhaps, this is due to the lack of the method to accept suboptimal cuts which yields small flow problems and therefore bad cuts. Consequently, we also combined both methods to fix this problem. In Table 1 we can see that the combination of flows with local search produces up to 6.14% lower cuts on average than the basic configuration. If we enable the most balancing cut heuristic we get on average 7.21% lower cuts than the basic configuration. Experiments in the TR [10] show that these combinations are more effective than the repeated executions of the basic two-way FM configuration. The most effective configuration is the basic two-way FM configuration using flows with $\alpha' = 8$ combined with the most balanced cut heuristic. It yields 4.73% lower cuts than the basic configuration in the effectiveness test.

6.2 Insights about Global Search Strategies

In Table 2 we compared different global search strategies against a single V-cycle. This time we choose a relatively fast configuration of the algorithm as basic configuration since the global search strategies are at focus. The coarsening phase is the same as in KaFFPa Strong. We perform one initial partitioning attempt using Scotch. The refinement employs k-way local search followed by quotient graph style refinements.

Flow algorithms are not enabled for this test. The only parameter varied during this test is the global search strategy. Clearly, more sophisticated global search strategies decrease the cut but also increase the run time of the algorithm. However, the effectiveness results in Table 2 indicate that repeated executions of more sophisticated global search strategies are always superior to repeated executions of one single V-cycle. The increased effectiveness of more sophisticated global search strategies is due to different reasons. First of all by using a given partition in later cycles we obtain a very good initial partitioning for the coarsest graph which yields good starting points for local improvement on each level of refinement. Furthermore, the increased effectiveness is due to time saved using the active block strategy which converges very quickly in later cycles. On the other hand we save time for initial partitioning since it is only performed the first time the algorithm arrives in the initial partitioning phase. It is interesting to see that although the analysis in Section 5 makes some simplified assumptions the measured run times in Table 2 are very close to the values obtained by the analysis.

Algorithm	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 2. Test results for normal and effectiveness tests for different global search strategies. Shown are improvements in % relative to the basic configuration .

6.3 Removal / Knockout Tests

We now turn into two kinds of experiments to evaluate interactions and relative importance of our algorithmic improvements. In the component *removal tests* we take KaFFPa Strong and remove components step by step yielding weaker and weaker variants of the algorithm. For the *knockout tests* only one component is removed at a time, i.e., each variant is exactly the same as KaFFPa Strong minus the specified component. Table 3 summarizes the knockout test results. More detailed results of the tests can be found in the TR [10]. We shortly summarize the main results. First, in order to achieve high quality partitions we don't need to perform classical global k-way refinement. 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 refinement algorithm speeds up overall run time of the algorithm; hence we included it into

Variant	Avg.	Best.	$t[s]$	Eff. Avg.	Eff. Best.
Strong	2 683	2 617	8.93	2 636	2 616
-KWay	−0.04	−0.11	9.23	0.00	0.08
-Multitry	1.71	1.49	5.55	1.21	1.30
-Cyc	2.42	1.95	3.27	1.25	1.41
-MB	3.35	2.64	2.92	1.82	1.91
-Flow	9.36	7.87	1.66	6.18	6.08

Table 3. Removal tests: each configuration is same as its predecessor minus the component shown in the first column. All average cuts and best cuts are shown as increases in cut (%) relative to the values obtained by KaFFPa Strong.

KaFFPa Strong. In contrast, removing the multi-try FM algorithm increases average cuts by almost two percent and decreases the effectiveness of the algorithm. It is also interesting to see that as soon as a component is removed from KaFFPa Strong (except for the global k-way search) the algorithm gets less effective.

6.4 Comparison with other Partitioners

We now switch to our suite of larger graphs since that's what KaFFPa was designed for. We compare ourselves with KaSPar Strong, KaPPa Strong, DiBaP Strong[3], Scotch and Metis. Table 4 summarizes the results. Detailed per instance results can be found in the TR [10]. kMetis produces about 33% larger cuts than KaFFPa Strong. Scotch, DiBaP, KaPPa, and KaSPar produce 20%,11%, 12% and 3% larger cuts than KaFFPa Strong respectively. In 57 out of 66 cases KaFFPa produces a better best cut than the best cut obtained by KaSPar. KaFFPa Eco now outperforms Scotch and DiBaP producing 4.7 % and 12% smaller cuts than DiBaP and Scotch respectively. Note that DiBaP has a factor 3 larger run times than KaFFPa Eco on average. In the TR [10] we take two graph families and study the behaviour of our algorithms when the graph size increases. As soon as the graphs have more than 2^{19} nodes, KaFFPa Fast outperforms kMetis in terms of speed and quality. In general the speed up of KaFFPa Fast relative to kMetis increases with increasing graph size. The largest difference is obtained on the largest graphs where kMetis has up to 70% larger run times than our fast configuration which still produces 2.5% smaller cuts.

Algorithm	large graphs		
	Best	Avg.	$t[s]$
KaFFPa Strong	12 054	12 182	121.50
KaSPar Strong	+3%	+3%	87.12
KaFFPa Eco	+6%	+6%	3.82
KaPPa Strong	+10%	+12%	28.16
Scotch	+18%	+20%	3.55
KaFFPa Fast	+25%	+24%	0.98
kMetis	+26%	+33%	0.83

Table 4. Averaged quality of the different partitioning algorithms.

6.5 The Walshaw Benchmark

We now apply KaFFPa Strong to Walshaw's benchmark archive [14] using 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 $\epsilon \in \{0, 0.01, 0.03, 0.05\}$.

We ran KaFFPa Strong with a time limit of two hours per graph and k (we excluded $\epsilon = 0$ since flows are not made for this case). 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 KaFFPa Strong as before and took the previous entry as input. Now overall in 560 out of 612 cases we where 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 at [10].

[3] We exluded the European and German road network as well as the Random Geometric Graphs for the comparison with DiBaP since DiBaP can't handle singletons. In general, we excluded the case $k = 2$ for the European road network since KaPPa runs out of memory for this case.

7 Conclusions and Future Work

KaFFPa is an approach to graph partitioning that can be configured to either achieve the best known partitions for many standard benchmark instances or to be the fastest available system for large graphs while still improving partitioning quality compared to the previous fastest system. This success is due to new local improvement methods and global search strategies which were transferred from multigrid linear solvers. Regarding future work, we want to try other initial partitioning algorithms and ways to integrate KaFFPa into metaheuristics like evolutionary search.

Acknowledgements

We would like to thank Vitaly Osipov for supplying data for KaSPar and Henning Meyerhenke for providing a DiBaP-full executable. We also thank Tanja Hartmann, Robert Görke and Bastian Katz for valuable advice regarding balanced min cuts.

References

1. R. Andersen and K.J. Lang. An algorithm for improving graph partitions. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 651–660. Society for Industrial and Applied Mathematics, 2008.
2. W.L. Briggs and S.F. McCormick. *A multigrid tutorial*. Soc. for Ind. Mathe., 2000.
3. P.O. Fjallstrom. Algorithms for graph partitioning: A survey. *Linkoping Electronic Articles in Computer and Information Science*, 3(10), 1998.
4. M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. *24th IEEE International Parallal and Distributed Processing Symposium*, 2010.
5. K. Lang and S. Rao. A flow-based method for improving the expansion or conductance of graph cuts. *Integer Programming and Combinatorial Optimization*, pages 383–400, 2004.
6. H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.*, pages 1–13, 2008.
7. V. Osipov and P. Sanders. n-Level Graph Partitioning. *18th European Symposium on Algorithms (see also arxiv preprint arXiv:1004.4024)*, 2010.
8. F. Pellegrini. Scotch home page. `http://www.labri.fr/pelegrin/scotch`.
9. J.C. Picard and M. Queyranne. On the structure of all minimum cuts in a network and applications. *Mathematical Programming Studies, Volume 13*, pages 8–16, 1980.
10. P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms (see ArXiv preprint arXiv:1012.0006v3). Technical report, Karlsruhe Institute of Technology, 2010.
11. K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra et al., editor, *CRPC Par. Comp. Handbook*. Morgan Kaufmann, 2000.
12. R. V. Southwell. Stress-calculation in frameworks by the method of "Systematic relaxation of constraints". *Proc. Roy. Soc. Edinburgh Sect. A*, pages 57–91, 1935.
13. C. Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1):325–372, 2004.
14. C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.
15. C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).