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 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 in the area of *high performance 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. Now this graph needs to be partitioned such there are few edges between the blocks (pieces). In particular, if we want to use k PEs (processing elements) 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 initial 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 [18] (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. We defer the question of parallelization since it introduces complications that make it difficult to try out a large number of alternatives

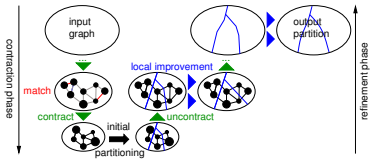


Fig. 1. Multilevel graph partitioning.

for the remaining aspects of the method. This paper reports the first results we have obtained which relate to the local improvement methods and overall 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.

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 $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. 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 undos 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. This is repeated until $|V|$ falls below some threshold. Contraction should quickly reduce the size of the input and each computed level should reflect the global structure of the input network. In particular, nodes should represent densely connected subgraphs.

Contraction is stopped when the graph is small enough to be directly partitioned in the *initial partitioning phase* using some other algorithm. We could use a trivial initial partitioning algorithm if we contract until exactly k nodes are left. However, if $|V| \gg k$ we can afford to run some expensive algorithm for initial partitioning.

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 nodes to move are often found using some kind of local search. 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.

2.2 More advanced concepts

This section gives a brief overview over the algorithms KaFFPa uses during contraction and initial partitioning. KaFFPa makes use of techniques proposed in [18] namely the application of edge ratings, the GPA algorithm to compute high quality matchings, pairwise refinements between blocks and it also uses Scotch [23] as an initial partitioner [18].

Contraction The contraction starts by rating the edges using a *rating function*. The rating function indicates how much sense it makes to contract an edge based on *local* information. Afterwards a *matching* algorithm tries to maximize the sum of the ratings of the contracted edges looking at the *global* structure of the graph. While the rating functions allows us a flexible characterization of what a "good" contracted graph is, the simple, standard definition of the matching problem allows us to reuse previously developed algorithms for weighted matching. Matchings are contracted until the graph is "small enough". We employed the ratings $\text{expansion}^{*2}(\{u, v\}) := \omega(\{u, v\})^2 / c(u)c(v)$ and $\text{innerOuter}(\{u, v\}) := \omega(\{u, v\}) / (\text{Out}(v) + \text{Out}(u) - 2\omega(u, v))$ where $\text{Out}(v) := \sum_{x \in \Gamma(v)} \omega(\{v, x\})$, since they yielded the best results in [18]. As a further measure to avoid unbalanced inputs to the initial partitioner, KaFFPa never allows a node v to participate in a contraction if the weight of v exceeds $1.5n/20k$

We used the *Global Path Algorithm (GPA)* which runs in near linear time to compute matchings. The Global Path Algorithm was proposed in [20] as a synthesis of

the Greedy algorithm and the Path Growing Algorithm [9]. It grows heavy weight paths and even length cycles to solve the matching problem on those optimally using dynamic programming. We choose this algorithm since in [18] it gives empirically considerably better results than Sorted Heavy Edge Matching, Heavy Edge Matching or Random Matching [25].

Similar to the Greedy approach, GPA scans the edges in order of decreasing weight but rather than immediately building a matching, it first constructs a collection of paths and even length cycles. Afterwards, optimal solutions are computed for each of these paths and cycles using dynamic programming.

Initial Partitioning The contraction is *stopped* when the number of remaining nodes is below $\max(60k, n/(60k))$. The graph is then small enough to be initially partitioned by some other partitioner. Our framework allows using kMetis or Scotch for initial partitioning. As observed in [18], Scotch [23] produces better initial partitions than Metis, and therefore we also use it in KaFFPa.

Refinement After a matching is uncontracted during the refinement phase, some local improvement methods are applied in order to reduce the cut while maintaining the balancing constraint.

We implemented two kinds of local improvement schemes within our framework. The first scheme is so called *quotient graph style refinement* [18]. This approach uses the underlying *quotient graph*. Each edge in the quotient graph yields a pair of blocks which share a non empty boundary. On each of these pairs we can apply a two-way local improvement method which only moves nodes between the current two blocks. Note that this approach enables us to integrate flow based improvement techniques between two blocks which are described in Section 4.1.

Our two-way local search algorithm works as in KaPPa [18]. We present it here for completeness. It is basically the FM-algorithm [13]: For each of the two blocks A, B under consideration, a priority queue of nodes eligible to move is kept. The priority is based on the *gain*, i.e., the decrease in edge cut when the node is moved to the other side. Each node is moved at most once within a single local search. The queues are initialized in random order with the nodes at the partition boundary.

There are different possibilities to select a block from which a node shall be moved. The classical FM-algorithm [13] alternates between both blocks. We employ the *TopGain* strategy from [18] which selects the block with the largest gain and breaks ties randomly if the the gain values are equal. In order to achieve a good balance, TopGain

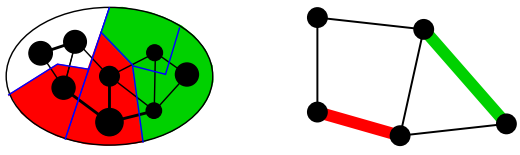


Fig. 2. A graph which is partitioned into five blocks and its corresponding quotient graph $\mathcal{Q}$ which has five nodes and six edges. Two pairs of blocks are highlighted in red and green.

adopts the exception that the block with larger weight is used when one of the blocks is overloaded. After a stopping criterion is applied we rollback to the best found cut within the balance constraint.

The second scheme is so call *k-way local search*. This method has a more global view since it is not restricted to moving nodes between two blocks only. It also basically the FM-algorithm [13]. We now outline the variant we use. Our variant uses only one priority queue P which is initialized with a subset S of the partition boundary in a random order. The priority is based on the max gain $g(v) = \max_P g_P(v)$ where $g_P(v)$ is the decrease in edge cut when moving v to block P. Again each node is moved at most once. Ties are broken randomly if there is more than one block that will give max gain when moving v to it. Local search then repeatedly looks for the highest gain node v. However a node v is not moved, if the movement would lead to an unbalanced partition. The k-way local search is stopped if the priority queue P is empty (i.e. each node was moved once) or a stopping criteria described below applies. Afterwards the local search is rolled back the lowest cut fulfilling the balance condition that occurred during this local search. This procedure is then repeated until no improvement is found or a maximum number of iterations is reached.

We adopt the stopping criteria proposed in KaSPar [22]. This stopping rule is derived using a random walk model. Gain values in each step are modelled as identically distributed, independent random variables whose expectation μ and variance σ^2 is obtained from the previously observed p steps since the last improvement. Osipov and Sanders [22] derived that it is unlikely for the local search to produce a better cut if

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

for some tuning parameters α and β. The Parameter β is a base value that avoids stopping just after a small constant number of steps that happen to have small variance. We also set it to $\ln n$.

There are different ways to initialize the queue P, e.g. the complete partition boundary or only the nodes which are incident to more than two partitions (corner nodes). Our implementation takes the complete partition boundary for initialization. In Section 4.2 we introduce multi-try k-way searches which is a more localized k-way search inspired by KaSPar [22]. This method initializes the priority queue with only a single boundary node and its neighbors that are also boundary nodes.

The main difference of our implementation to KaSPar is that we use only one priority queue. KaSPar maintains a priority queue for each block. A priority queue is called eligible if the highest gain node in this queue can be moved to its target block without violating the balance constraint. Their local search repeatedly looks for the highest gain node v in any eligible priority queue and moves this node.

3 Related Work

There has been a huge amount of research on graph partitioning so that we refer the reader to [14,25,31] 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 [26,11] 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 Chaco [17], Jostle [31], Metis [25], Party [10], and Scotch [23].

KaSPar [22] is a new graph partitioner based on the central idea to (un)contract only a single edge between two levels. It previously obtained the best results for many of the biggest graphs in [28].

KaPPa [18] is a "classical" matching based MGP algorithm designed for scalable parallel execution and its local search only considers independent pairs of blocks at a time.

DiBaP [21] is a multi-level graph partitioning package where local improvement is based on diffusion which also yields partitions of very high quality.

MQI [19] 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. Given an undirected graph with an initial partitioning, they build up a completely new directed graph which is then used to solve a max flow problem. Furthermore, they have been able to show that there is an improved quotient cut if and only if the maximum flow is less than ca, where c is the initial cut and a is the number of vertices in the smaller block of the initial partitioning. This approach is currently only feasible for $k = 2$. Improve also uses several minimum cut computations to improve the quotient cut score of a proposed partition. Improve always beats or ties MQI.

Very recently an algorithm called PUNCH [7] has been introduced. This approach is not based on the multilevel principle. However, it creates a coarse version of the graph based on the notion of natural cuts. Natural cuts are relatively sparse cuts close to denser areas. They are discovered by finding minimum cuts between carefully chosen regions of the graph. Experiments indicate that the algorithm computes very good cuts for road networks. For instances that don't have a natural structure such as road networks, natural cuts are not very helpful.

The concept of *iterated multilevel algorithms* was introduced by [27,29]. The main idea is to iterate the coarsening and uncoarsening phase and use the information gathered. That means that once the graph is partitioned, edges that are between two blocks will not be matched and therefore will also not be contracted. This ensures increased quality of the partition if the refinement algorithms guarantees not to find a worse partition than the initial one.

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 which is described in Section 4.1 is based on max-flow min-cut computations between pairs of blocks, i.e. improving a given 2-partition. Since each edge of the quotient graph yields a pair of blocks which share a non empty boundary, we integrated this method into the quotient graph style refinement scheme which is described in Section 2.2. The second method which is described in Section 4.2 is called multi-try FM which is a more localized k-way local search. Roughly speaking, a k-way

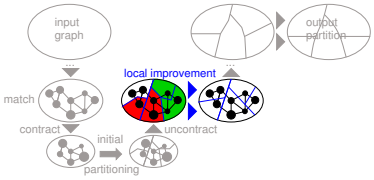


Fig. 3. After a matching is uncontracted a local improvement method is applied.

local search is repeatedly started with a priority queue which is initialized with only one random boundary node and its neighbors that are also boundary nodes. At the end of the section we shortly show how the pairwise refinements can be scheduled and how the more localized search can be incorporated with this scheduling.

4.1 Using Max-Flow Min-Cut Computations for Local Improvement

We now explain how flows can be used to improve a given partition of two blocks and therefore can be used as a refinement algorithm in a multilevel framework. For simplicity we assume $k = 2$. However it is clear that this refinement method fits perfectly into the quotient graph style refinement algorithms.

To start with the description of the constructed max-flow min-cut problem, we need a few notations. Given a two-way partition $P : V \to \{1, 2\}$ of a graph G we define the *boundary nodes* as $\delta := \{u \mid \exists (u, v) \in E : P(u) \neq P(v)\}$. We define *left boundary nodes* to be $\delta_l := \delta \cap \{u \mid P(u) = 1\}$ and *right boundary nodes* to be $\delta_r := \delta \cap \{u \mid P(u) = 2\}$. Given a set of nodes $B \subset V$ we define its *border* $\partial B := \{u \in B \mid \exists (u, v) \in E : v \notin B\}$. Unless otherwise mentioned we call B *corridor* because it will be a zone around the initial cut. The set $\partial_l B := \partial B \cap \{u \mid P(u) = 1\}$ is called *left corridor border* and the set $\partial_r B := \partial B \cap \{u \mid P(u) = 2\}$ is called *right corridor border*. We say an *B-corridor induced subgraph* G' is the node induced subgraph $G[B]$ plus two nodes s, t and additional edges starting from s or edges ending in t. An B-corridor induced subgraph has the *cut property* C if each (s,t)-min-cut in G' induces a cut within the balance constrained in G.

The main idea is to construct a B-corridor induced subgraph G' with cut property C. On this graph we solve the max-flow min-cut problem. The computed min-cut yields a feasible improved cut within the balance constrained in G. The construction is as follows (see also Figure 4).

First we need to find a corridor B such that the B-corridor induced subgraph will have the cut property C. This can be done by performing two Breadth First Searches (BFS). Each node touched during these searches belongs to the corridor B. The first BFS is initialized with the left boundary nodes δ_l. It is only expanded with nodes that are in block 1. As soon as the weight of the area found by this BFS would exceed $(1 + \epsilon)c(V)/2 - w(\text{block } 2)$, we stop the BFS. The second BFS is done for block 2 in an analogous fashion.

In order to achieve the cut property C, the B-corridor induced subgraph G' gets additional s-t edges. More precisely s is connected to all left corridor border nodes $\partial_l B$

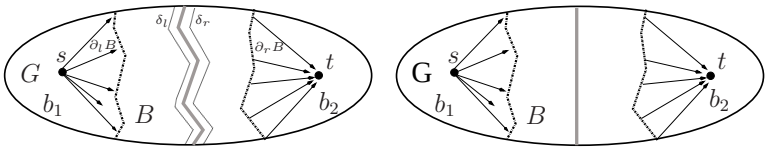


Fig. 4. The construction of a feasible flow problem which yields optimal cuts in G' and an improved cut within the balance constraint in G. On the top the initial construction is shown and on the bottom we see the improved partition.

and all right corridor border nodes $\partial_r B$ are connected to t. All of these new edges get the edge weight ∞. Note that this are directed edges.

The constructed B-corridor subgraph G' has the cut property C since the worst case new weight of block 2 is lower or equal to $w(\text{block 2}) + (1+\epsilon)c(V)/2 - w(\text{block 2}) = (1+\epsilon)c(V)/2$. Indeed the same holds for the worst case new weight of block 1.

There are multiple ways to improve this method. First, if we found an improved edge cut, we can apply this method again since the initial boundary has changed which implies that it is most likely that the corridor B will also change. Second, we can adaptively control the size of the corridor B which is found by the BFS. This enables us to search for cuts that fulfill our balance constrained even in a larger corridor (say $\epsilon' = \alpha\epsilon$ for some parameter α), i.e. 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 an decreased edge 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. Here finding all minimum cuts reduces to a straight forward enumeration. Having this in mind the idea to search for min-cuts in larger corridors becomes even more attractive. Roughly speaking, we present a heuristic that, given a max-flow, creates min-cuts that are better balanced. 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 5.

Lemma 1 (Picard and Queyranne [24]). *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.*

To be more precise for a given closed vertex set C containing s of the residual graph the corresponding min-cut is $(C, V \backslash C)$. Note that distinct maximum flows may produce different residual graphs but the set of closed vertex sets remains the same. To enumerate all minimum cuts of a graph [24] a further reduced graph is computed which is described below. However, the problem of finding the minimum cut with the best balance (most balanced minimum cut) is NP-hard [12,2].

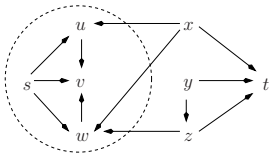


Fig. 5. A small graph where $C = \{s, u, v, w\}$ is a closed vertex set.

The minimum cut that is identified by the labeling procedure of Ford and Fulkerson [15] is the one with the smallest possible source set. We now define how the representation of the residual graph can be made more compact [24] and then explain the heuristic we use to obtain closed vertex sets on this graph to find min-cuts that have a better balance. After computing a maximum (s, t)-flow, we compute the strongly connected components of the residual graph using the algorithm proposed in [4,16]. We make the representation more compact by contracting these components and refer to it as *minimum cut representation*. This 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.

As proposed in [24] we make the minimum cut representation even more compact: We eliminate the component T containing the sink t, and all its predecessors (since they cannot belong to a closed vertex set not containing T) and the component S containing the source, and all its successors (since they must belong to a closed vertex set containing S) using a BFS.

We are now left with a further reduced graph. On this graph we search for closed vertex sets (containing S) 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 topological order yields complements of closed vertex sets quite easily. Therefore, we first compute a random topological order, e.g. using a randomized DFS. Next we sweep through this topological order and sequentially add the components to the complement of the closed vertex set. Note that each of the computed complements of closed vertex sets $\tilde{C}$ also yields a closed vertex set ($V \backslash \tilde{C}$). That means 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 topological order with respect to the original graph partitioning problem. The closed vertex set with the best balance occurred during the repetitions of this heuristic is returned. Note in large corridors this procedure may finds cuts that are not feasible, 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*.

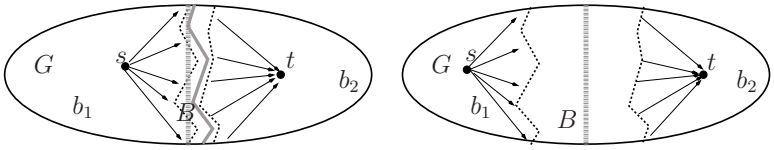


Fig. 6. In the situation on the top it is not possible in the small corridor around the initial cut to find the dashed minimum cut which has optimal balance; however if we solve a larger flow problem on the bottom and search for a cut with good balance we can find the dashed minimum cut with optimal balance but not every min cut is feasible for the underlying graph partitioning problem.

4.2 Multi-try FM

This refinement variant is organized in rounds. In each round we put *all* boundary nodes of the current block pair into a todo list. The todo list is then permuted. Subsequently, we begin a k-way local search starting with a random node of this list if it is still a boundary node and its neighboring nodes that are also boundary nodes. Note that the difference to the global k-way search described in Section 2.2 is the initialisation of the priority queue. If the selected random node was already touched by a previous k-way search in this round then no search is started. Either way, the node is removed from the todo list (simply swapping it with the last element and executing a pop_back on that list). For a k-way search it is not allowed to move nodes that have been touched in a previous run. This way we can assure that at most n nodes are touched during one round of the algorithm. This algorithm uses the adaptive stopping criteria from KaSPar which is described in Section 2.2.

4.3 Scheduling Quotient Graph Refinement

There a two possibilities to schedule the execution of two way refinement algorithms on the quotient graph. Clearly the first simple idea is to traverses the edges of Q in a random order and perform refinement on them. This is iterated until no change occurred or a maximum number of iterations is reached. The second algorithm is called *active block scheduling*. The main idea behind this algorithm is that the local search should be done in areas in which change still happens and therefore avoid unnecessary local search. The algorithm begins by setting every block of the partition *active*. Now the scheduling is organized in rounds. In each round, the algorithm refines adjacent pairs of blocks, which 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. After each pair-wise improvement a multi-try FM search (k-way) is started. It is initialized with the boundaries of the current pair of blocks. Now each block which changed during this search is also marked active. The algorithm stops if no active block is left. Pseudocode for the algorithm can be found in the appendix in Figure 11.

5 Global Search

Iterated Multilevel Algorithms where introduced by [27,29] (see Section 3). For the rest of this paper Iterated Multilevel Algorithms are called V-cycles unless otherwise mentioned. The main idea is that if a partition of the graph is available then it can be reused during the coarsening and uncoarsening phase. To be more precise, the multilevel scheme is repeated several times and once the graph is partitioned, edges between two blocks will not be matched and therefore will also not be contracted such that a given partition can be used as initial partition of the coarsest graph. This ensures increased quality of the partition if the refinement algorithms guarantees not to find a worse partition than the initial one. Indeed this is only possible if the matching includes non-deterministic factors such as random tie-breaking, so that each iteration is very likely to give different coarser graphs. Interestingly, in multigrid linear solvers Full-Multigrid methods are generally preferable to simple V-cycles [3]. 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 independent trials* using different random seeds for tie breaking during contraction, and local search. As soon as the graph is partitioned, edges that are between blocks are not matched. A F-cycle works similar to a W-cycle with the difference that the global number of independent trials on each level is bounded by 2. Examples for the different cycle types can be found in Figure 7 and Pseudocode can be found in Figure 10. Again once the graph is partitioned for the first time, then this partition is used in the sense that edges between two blocks are not contracted. In most cases the initial partitioner is not able to improve this partition from scratch or even to find this partition. Therefore no further initial partitioning is used if the graph already has a partition available. These methods can be used to find very high quality partitions but on the other hand they are more expensive than a single MGP run. However, experiments in Section 6 show that all cycle variants are more efficient than simple plain restarts of the algorithm. In order to bound the runtime we introduce a level split parameter d such that the independent trials are only performed every d'th level. We go into more detail after we have analysed the run time of the global search strategies.

Fig. 7. From left to right: A single MGP V-cycle, a W-cycle and a F-cycle.

Analysis We now roughly analyse the run time of the different global search strategies under a few assumptions. In the following the shrink factor names the factor the graph shrinks 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:*

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

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

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.

Proof. The run time of a single V-cycle is given by $T_V(n) = \sum_{i=0}^{l} T_{\text{cr}}(a^i n) = bn \sum_{i=0}^{l} a^i = bn(1-a^{l+1})/(1-a)$. The run time of a W-cycle with level split parameter d is given by the time of d coarsening and refinement steps plus the time of the two trials on the created coarse graph. For the case $2a^d < 1$ we get

$$\begin{aligned} T_{W,d}(n) &= bn \sum_{i=0}^{d-1} a^i + 2T_{W,d}(a^d n) \leq bn \frac{1-a^d}{1-a} \sum_{i=0}^{\infty} (2a^d)^i \\ &\leq \frac{1-a^d}{(1-a^{l+1})(1-2a^d)} T_V(n) \approx \frac{1-a^d}{1-2a^d} T_V(n). \end{aligned}$$

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

$$T_{F,d}(n) \leq \sum_{i=0}^{l} T_{\text{cr}}(a^{i \cdot d} n) \leq \frac{bn}{1-a} \sum_{i=0}^{\infty} (a^d)^i = \frac{1}{1-a^d} T_V(n)$$

where l is the total number of levels. This completes the proof of the theorem.

Note that if we make the optimistic assumption that $a = 1/2$ and set $d = 1$ then 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, i.e. independent trials are only performed every second level.

6 Experiments

Implementation We have implemented the algorithm described above using C++. Overall, our program consists of about 12 500 lines of code. Priority queues for the local search are based on binary heaps. Hash tables use the library (extended STL) provided with the GCC compiler. For the following comparisons we used Scotch 5.1.9., DiBaP 2.0.229 and kMetis 5.0 (pre2). The flow problems are solved using Andrew Goldbergs Network Optimization Library HIPR [5] which is integrated into our code.

System We have run our code on a cluster where each node is equipped with two Quad-core Intel Xeon processors (X5355) which run at a clock speed of 2.667 GHz, has 2x4 MB of level 2 cache each and run Suse Linux Enterprise 10 SP 1. Our program was compiled using GCC Version 4.3.2 and optimization level 3.

Instances We report experiments on two suites of instances summarized in the appendix in Table 5. These are the same instances as used for the evaluation of KaPPa [18]. We present them here for completeness. $rggX$ is a *random geometric graph* with 2^X nodes where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost connected. $DelaunayX$ is the Delaunay triangulation of 2^X random points in the unit square. Graphs $bcsstk29..fetooth$ and $ferotor..auto$ come from Chris Walshaw's benchmark archive [30]. Graphs bel, nld, deu and eur are undirected versions of the road networks of Belgium, the Netherlands, Germany, and Western Europe respectively, used in [8]. Instances af_shell9 and $af\_shell10$ come from the Florida Sparse Matrix Collection [6]. For the number of partitions k we choose the values used in [30]: 2, 4, 8, 16, 32, 64. Our default value for the allowed imbalance is 3 % since this is one of the values used in [30] and the default value in Metis.

Configuring the Algorithm We currently define three configurations of our algorithm: Strong, Eco and Fast. The configurations are described below.

KaFFPa Strong: The aim of this configuration is to obtain a graph partitioner that is able to achieve the best known partitions for many standard benchmark instances. It uses the GPA algorithm as a matching algorithm combined with the rating function $\mathrm{expansion}^{*2}$. However, the rating function $\mathrm{expansion}^{*2}$ has the disadvantage that it evaluates to one on the first level of an unweighted graph. Therefore, we employ $\mathrm{innerOuter}$ on the first level to infer structural information of the graph. We perform $100/\log k$ initial partitioning attempts using Scotch as an initial partitioner. The *refinement phase* first employs k-way refinement (since it converges very fast) which is initialized with the complete partition boundary. It uses the adaptive search strategy from KaSPar [22] with $\alpha = 10$. The number of rounds is bounded by ten. However, the k-way local search is stopped as soon as a k-way local search round did not find an improvement. We continue by performing quotient-graph style refinement. Here we use the active block scheduling algorithm which is combined with the multi-try local search (again $\alpha = 10$) as described in Section 4.3. A pair of blocks is refined as follows: We start with a pairwise FM search which is followed by the max-flow min-cut algorithm (including the most balancing cut heuristic). The FM search is stopped if more than 5%

of the number of nodes in the current block pair have been moved without yielding an improvement. The upper bound factor for the flow region size is set to $\alpha' = 8$. As *global search strategy* we use two F-cycles. Initial Partitioning is only performed if previous partitioning information is *not* available. Otherwise, we use the given input partition.

KaFFPa Eco: The aim of KaFFPa Eco is to obtain a graph partitioner that is fast on the one hand and on the other hand is able to compute partitions of high quality. This configuration matches the first $\max(2, 7 - \log k)$ levels using a random matching algorithm. The remaining levels are matched using the GPA algorithm employing the edge rating function $\mathrm{expansion}^{*2}$. It then performs $\min(10, 40/\log k)$ initial partitioning repetitions using Scotch as initial partitioner. The refinement is configured as follows: again we start with k-way refinement as in KaFFPa-Strong. However, for this configuration the number of k-way rounds is bounded by $\min(5, \log k)$. We then apply quotient-graph style refinements as in KaFFPa Strong; again with slightly different parameters. The two-way FM search is stopped if 1% of the number of nodes in the current block pair has been moved without yielding an improvement. The flow region upper bound factor is set to $\alpha' = 2$. We do not apply a more sophisticated global search strategy in order to be competitive regarding runtime.

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

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*. [1]

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

[1] 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 runtime **t** of each algorithm configuration.

corresponding probabilities such that the expected time over multiple runs is $3t$. This is repeated 5 times. The final score is computed as in the normal test using these values.

6.1 Insights about Flows

We now evaluate how much the usage of max-flow min-cut algorithms improves the final partitioning results and check its effectiveness. For this test we use a basic two-way FM configuration to compare with. This basic configuration is modified as described below to look at a specific algorithmic component regarding flows. It uses the Global Paths Algorithm as a matching algorithm and performs five initial partitioning attempts using Scotch as initial partitioner. It further employs the active block scheduling algorithm equipped with the two-way FM algorithm described in Section 2.2. The FM algorithm stopps 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. For comparisons this basic configuration is extended by specific algorithms, e.g. a configuration that uses Flow, FM and the most balanced cut heuristics (MB). This configuration is then indicated by (+Flow, +FM, +MB).

In Table 1 we see that by Flow on its own, i.e. no FM-algorithm is used at all, we obtain cuts and run times which are worse than the basic two-way FM configuration. The results improve in terms of quality and runtime if we enable the most balanced minimum cut heuristic. Now for $\alpha' = 16$ and $\alpha' = 8$, we get cuts that are 0.81% and 0.41% lower on average than the cuts produced by the basic two-way FM configuration. However, these configurations have still a factor four ($\alpha' = 16$) or a factor two ($\alpha' = 8$) larger run times. 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

Variant	(+Flow, -MB, -FM)				(+Flow, +MB, -FM)				(+Flow, -MB, +FM)				(+Flow, +MB, +FM)			
α'	Avg.	Best.	Bal.	t	Avg.	Best.	Bal.	t	Avg.	Best.	Bal.	t	Avg.	Best.	Bal.	t
16	−1.88	−1.28	1.03	4.17	0.81	0.35	1.02	3.92	6.14	5.44	1.03	4.30	7.21	6.06	1.02	5.01
8	−2.30	−1.86	1.03	2.11	0.41	−0.14	1.02	2.07	5.99	5.40	1.03	2.41	7.06	5.87	1.02	2.72
4	−4.86	−3.78	1.02	1.24	−2.20	−2.80	1.02	1.29	5.27	4.70	1.03	1.62	6.21	5.36	1.02	1.76
2	−11.86	−10.35	1.02	0.90	−9.16	−8.24	1.02	0.96	3.66	3.37	1.02	1.31	4.17	3.82	1.02	1.39
1	−19.58	−18.26	1.02	0.76	−17.09	−16.39	1.02	0.80	1.64	1.68	1.02	1.19	1.74	1.75	1.02	1.22
Ref.	(-Flow, -MB, +FM)				2 974	2 851	1.025	1.13								

Table 1. The final score of different algorithm configurations compared against the basic two-way FM configuration. The parameter α' is the flow region upper bound factor. All average and best cut values except for the basic configuration are improvements relative to the basic configuration in %.

Effectiveness	(+Flow, +MB, -FM)		(+Flow,-MB, +FM)		(+Flow,+MB,+FM)	
	Avg.	Best.	Avg.	Best.	Avg.	Best.
$\alpha' = 1$	−16.41	−16.35	1.62	1.52	1.65	1.63
2	−8.26	−8.07	3.02	2.83	3.36	3.25
4	−3.05	−3.08	4.04	3.82	4.63	4.36
8	−1.12	−1.34	4.16	4.13	4.74	4.64
16	−1.29	−1.27	3.70	3.86	4.28	4.36
(-Flow, -MB, +FM)	2 833	2 803	2 831	2 801	2 827	2 799

Table 2. Three effectiveness tests each one with six different algorithm configurations. All average and best cut values except for the basic configuration are improvements relative to the basic configuration in %.

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. Since these configurations are the basic two-way FM configuration augmented by flow algorithms they have an increased run time compared to the basic configuration. However, Table 2 shows that these combinations are also more effective than the repeated execution of the basic two-way FM configuration. The most effective configuration is the basic two-way FM configuration using flows with $\alpha' = 8$ and uses the most balanced cut heuristic. It yields 4.73% lower cuts than the basic configuration in the effectiveness test. Absolute values for the test results can be found in Table 6 and Table 7 in the Appendix.

6.2 Insights about Global Search Strategies

In Table 3 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 runtime of the algorithm. However, the effectiveness results in Table 3 indicate that repeated executions of more sophisticated global search strategies are always superior to repeated executions of one single V-cycle. The largest difference in best cut effectiveness is obtained by repeated executions of 2 W-cycles and 2 F-cycles which produce 1.5% lower best cuts than repeated executions of a normal 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. This initial partitioning is usually much better than a partition created by another initial partitioner which yields good start 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 which 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 3 are very close to the values obtained by the analysis.

Algorithm	Avg.	Best	Bal.	t	Eff. Avg.	Eff. Best
2 F-cycle	2.69	2.45	1.023	2.31	2 806	2 760
3 V-cycle	2.69	2.34	1.023	2.49	2 810	2 766
2 W-cycle	2.91	2.75	1.024	2.77	2 810	2 760
1 W-cycle	1.33	1.10	1.024	1.38	2 815	2 773
1 F-cycle	1.09	1.00	1.024	1.18	2 816	2 783
2 V-cycle	1.88	1.61	1.024	1.67	2 817	2 778
1 V-cycle	2 973	2 841	1.024	0.85	2 834	2 801

Table 3. Test results for normal and effectiveness tests for different global search strategies. The average cut and best cut values are improvements in % relative to the basic configuration (1 V-cycle). For F- and W-cycles $d = 2$. Absolute values can be found in Table 8 in the Appendix.

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.

In the following, *KWay* means the global k-way search component of KaFFPa Strong, *Multitry* stands for the more localized k-way search during the active block scheduling algorithm and *-Cyc* means that the F-Cycle component is replaced by one V-cycle. Furthermore, *MB* stands for the most balancing minimum cut heuristic, and *Flow* means the flow based improvement algorithms.

In Table 4 we see results for the component removal tests and knockout tests. More detailed results can be found in the appendix. First notice that in order to achieve high quality partitions we don't need to perform classical global k-way refinement (KWay). The changes in solution quality are negligible and both configurations (Strong without KWay and Strong) are equally effective. However, the global k-way refinement algorithm converges very quickly and therefore speeds up overall runtime of the algorithm; hence we included it into our KaFFPa Strong configuration.

In both tests the largest differences are obtained when the components Flow and/or the Multitry search heuristic are removed. When we remove all of our new algorithmic components from KaFFPa Strong, i.e global k-way search, local multitry search, F-Cycles, and Flow we obtain a graph partitioner that produces 9.3% larger cuts than KaFFPa Strong. Here the effectiveness average cut of the weakest variant in the removal test is about 6.2% larger than the effectiveness average cut of KaFFPa Strong. Also note that as soon as a component is removed from KaFFPa Strong (except for the global k-way search) the algorithm gets less effective.

Variant	Avg.	Best.	t	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

Variant	Avg.	Best.	t	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	1.27	1.11	5.52	0.83	0.99
-MB	0.26	0.08	8.34	0.11	0.11
-Flow	1.53	0.99	6.33	0.87	0.80

Table 4. Removal tests (top): each configuration is same as its predecessor minus the component shown at beginning of the row. Knockout tests (bottom): each configuration is same as KaFFPa Strong minus the component shown at beginning of the row. All average cuts and best cuts are shown as increases in cut (%) relative to the values obtained by KaFFPa Strong.

6.4 Comparison with other Partitioners

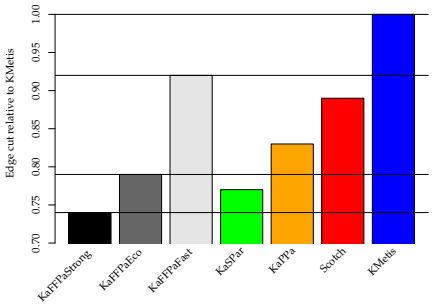
We now switch to our suite of larger graphs since that's what KaFFPa was designed for and because we thus avoid the effect of overtuning our algorithm parameters to the instances used for calibration. We compare ourselves with KaSPar Strong, KaPPa Strong, DiBaP Strong, Scotch and Metis.

Figure 8 summarizes the results. We excluded the European and German road network as well as the Random Geometric Graph for the comparison with DiBaP since DiBaP can't handle singletons. In general, we excluded the case $k = 2$ for the European road network for the comparison since it runs out of memory for this case. As recommended by Henning Meyerhenke DiBaP was run with 3 bubble repetitions, 10 FOS/L consolidations and 14 FOS/L iterations. Detailed per instance results can be found in Appendix Table 13.

kMetis produces about 33% larger cuts than the strong variant of KaFFPa. Scotch, DiBaP, KaPPa, and KaSPar produce 20%,11%, 12% and 3% larger cuts than KaFFPa respectively. The strong variant of KaFFPa now produces the average best cut results of KaSPar on average (which where obtained using five repeated executions of KaSPar). In 57 out of 66 cases KaFFPa produces a better best cut than the best cut obtained by KaSPar.

The largest absolute improvement to KaSPar Strong is obtained on *af_shell10* at $k = 16$ where the best cut produced by KaSPar-Strong is 7.2% larger than the best cut produced by KaFFPa Strong. The largest absolute improvement to kMetis is obtained on the European road network where kMetis produces cuts that are a factor 5.5 larger than the edge cuts produces by our strong configuration.

The eco configuration of KaFFPa now outperforms Scotch and DiBaP being than DiBaP while producing 4.7 % and 12% smaller cuts than DiBap and Scotch respectively. The run time difference to both algorithms gets larger with increasing number of



algorithm	large graphs		
	best	avg.	t[s]
KaFFPa Strong	12 054	12 182	121.50
KaSPar Strong	12 450	+3%	87.12
KaFFPa Eco	12 763	+6%	3.82
KaPPa Strong	13 323	+12%	28.16
Scotch	14 218	+20%	3.55
KaFFPa Fast	15 124	+24%	0.98
kMetis	15 167	+33%	0.83

Fig. 8. Averaged quality of the different partitioning algorithms.

blocks. Note that DiBaP has a factor 3 larger run times than KaFFPa Eco on average and up to factor 4 on average for $k = 64$.

On the largest graphs available to us (*delaunay, rgg, eur*) KaFFPa Fast outperforms KMetis in terms of quality and runtime. For example on the *european road network* kMetis has about 44% larger run times and produces up to a factor 3 (for $k = 16$) larger cuts.

We now turn into graph sequence tests. Here we take two graph families (*rgg*, *delaunay*) and study the behaviour of our algorithms when the graph size increases. In Figure 9, we see for increasing size of random geometric graphs the run time advantage of KaFFPa Fast relative to kMetis increases. The largest difference is obtained on the largest graph where kMetis has 70% larger run times than our fast configuration which still produces 2.5% smaller cuts. We observe the same behaviour for the delaunay based graphs (see appendix for more details). Here we get a run time advantage of up to 24% with 6.5% smaller cuts for the largest graph. Also note that for these graphs the improvement of KaFFPa Strong and Eco in terms of quality relative to kMetis increases with increasing graph size (up to 32% for delaunay and up to 47% for rgg for our strong configuration).

6.5 The Walshaw Benchmark

We now apply KaFFPa to Walshaw's benchmark archive [30] 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 tried all combinations except the case $\epsilon = 0$ because flows are not made for this case.

We ran KaFFPa Strong with a time limit of two hours per graph and k and report the best result obtained in the appendix. 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. The complete list of improvements is available at Walshaw's archive [30]. We obtain only a few improvements for $k = 2$. However, in this case we are able to reproduce the currently best result in 91 out of 102 cases. For the large graphs (using 78000 nodes as

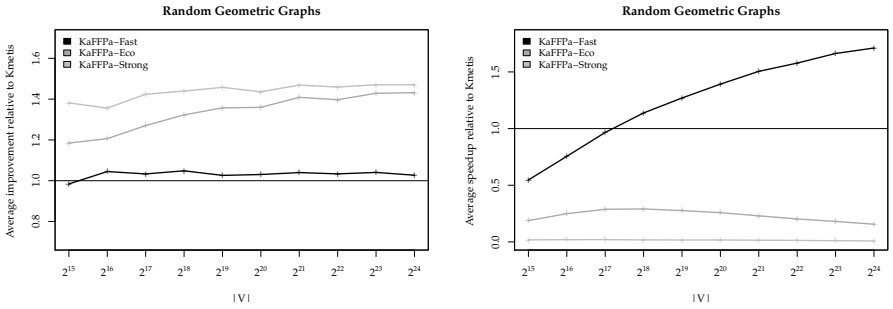


Fig. 9. Graph sequence test for Random Geometric Graphs.

a cut off) we obtain cuts that are lower or equal to the current entry in 92% of the cases. The biggest absolute improvement is observed for instance *add32* (for each imbalance) and $k = 4$ where the old partitions cut 10 % more edges. The biggest absolute difference is obtained for *m14b* at 3 % imbalance and $k = 64$ where the new partition cuts 3183 less edges.

After the partitions were accepted, we ran KaFFPa Strong as before and took the previous entry as input. Now in 560 out of 612 cases we where able to improve a given entry or have been able to reproduce the current result.

7 Conclusions and Future Work

KaFFPa is an approach to graph partitioning which currently computes the best known partitions for many graphs, at least when a certain imbalance is allowed. This success is due to new local improvement methods, which are based on max-flow min-cut computations and more localized local searches, and global search strategies which were transferred from multigrid linear solvers.

A lot of opportunities remain to further improve KaFFPa. For example we did not try to handle the case $\epsilon = 0$ since this may require different local search strategies. Furthermore, we want to try other initial partitioning algorithms and ways to integrate KaFFPa into other metaheuristics like evolutionary search.

Moreover, we would like to go back to parallel graph partitioning. Note that our max-flow min-cut local improvement methods fit very well into the parallelization scheme of KaPPa [18]. We also want to combine KaFFPa with the n-level idea from KaSPar [22]. Other refinement algorithms, e.g., based on diffusion or MQI could be tried within our framework of pairwise refinement.

The current implementation of KaFFPa is a research prototype rather than a widely usable tool. However, we are planing an open source release available for download.

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. P. Bonsma. Most balanced minimum cuts. *Discrete Applied Mathematics*, 158(4):261–276, 2010.
3. W.L. Briggs and S.F. McCormick. *A multigrid tutorial*. Society for Industrial Mathematics, 2000.
4. J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.
5. B.V. Cherkassky and A.V. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997.
6. T. Davis. The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices`, 2008.
7. D. Delling, A.V. Goldberg, I. Razenshteyn, and R.F. Werneck. Graph Partitioning with Natural Cuts. Technical report, Microsoft Research, MSR-TR-2010-164, 2010.
8. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS State-of-the-Art Survey*, pages 117–139. Springer, 2009.
9. D. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85:211–213, 2003.
10. R. Preis et al. PARTY partitioning library. `http://wwwcs.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html`.
11. R. P. Fedorenko. A relaxation method for solving elliptic difference equations. *USSR Comput. Math. and Math. Phys*, 5(1):1092–1096, 1961.
12. U. Feige and M. Mahdian. Finding small balanced separators. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 375–384. ACM, 2006.
13. C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Conference on Design Automation*, pages 175–181, 1982.
14. P.O. Fjallstrom. Algorithms for graph partitioning: A survey. *Linkoping Electronic Articles in Computer and Information Science*, 3(10), 1998.
15. L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
16. H.N. Gabow. Path-Based Depth-First Search for Strong and Biconnected Components. *Information Processing Letters*, 74(3-4):107–114, 2000.
17. B. Hendrickson. Chaco: Software for partitioning graphs. `http://www.sandia.gov/~bahendr/chaco.html`.
18. M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. *24th IEEE International Parallal and Distributed Processing Symposium*, 2010.
19. 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.
20. J. Maue and P. Sanders. Engineering algorithms for approximate weighted matching. In *6th Workshop on Exp. Algorithms (WEA)*, volume 4525 of *LNCS*, pages 242–255. Springer, 2007.

21. 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.
22. V. Osipov and P. Sanders. n-Level Graph Partitioning. *18th European Symposium on Algorithms (see also arxiv preprint arXiv:1004.4024)*, 2010.
23. F. Pellegrini. Scotch home page. `http://www.labri.fr/pelegrin/scotch`.
24. 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.
25. 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.
26. 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.
27. M. Toulouse, K. Thulasiraman, and F. Glover. Multi-level cooperative search: A new paradigm for combinatorial optimization and an application to graph partitioning. *Euro-Par 99 Parallel Processing*, pages 533–542, 1999.
28. C. Walshaw. The Graph Partitioning Archive, `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`, 2008.
29. C. Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1):325–372, 2004.
30. C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.
31. 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).

```
procedure W-Cycle(G)
  G' =coarsen(G)
  if G' small enough then
    initial partition G' if not partitioned
    apply partition of G' to G
    perform refinement on G
  else
    W-Cycle(G') and apply partition to G
    perform refinement on G
    G'' =coarsen(G)
    W-Cycle(G'') and apply partition to G
    perform refinement on G
procedure F-Cycle(G)
  G' =coarsen(G)
  if G' small enough then
    initial partition G' if not partitioned
    apply partition of G' to G
    perform refinement on G
  else
    F-Cycle(G') and apply partition to G
    perform refinement on G
    if no. trails. calls on cur. level < 2 then
      G'' =coarsen(G)
      F-Cycle(G'') and apply partition to G
      perform refinement on G
```

Fig. 10. Pseudocode for the different global search strategies.

```
procedure activeBlockScheduling()
  set all blocks active
  while there are active blocks
    A := <edge (u,v) in quotient graph : u active or v active>
    set all blocks inactive
    permute A randomly
    for each (u,v) in A do
      pairWiseImprovement(u,v)
      multitry FM search starting with boundary of u and v
      if anything changed during local search then
        activate blocks that have changed during pairwise
        or multitry FM search
```

Fig. 11. Pseudocode for the active block scheduling algorithm. In our implementation the pairwise improvement step starts with a FM local search which is followed by a max-flow min-cut based improvement.

Medium sized instances		
graph	n	m
rgg17	2^{17}	1 457 506
rgg18	2^{18}	3 094 566
Delaunay17	2^{17}	786 352
Delaunay18	2^{18}	1 572 792
bcsstk29	13 992	605 496
4elt	15 606	91 756
fesphere	16 386	98 304
cti	16 840	96 464
memplus	17 758	108 384
cs4	33 499	87 716
pwt	36 519	289 588
bcsstk32	44 609	1 970 092
body	45 087	327 468
t60k	60 005	178 880
wing	62 032	243 088
finan512	74 752	522 240
rotor	99 617	1 324 862
bel	463 514	1 183 764
nld	893 041	2 279 080
af_shell9	504 855	17 084 020
Large instances		
rgg20	2^{20}	13 783 240
Delaunay20	2^{20}	12 582 744
fetooth	78 136	905 182
598a	110 971	1 483 868
ocean	143 437	819 186
144	144 649	2 148 786
wave	156 317	2 118 662
m14b	214 765	3 358 036
auto	448 695	6 629 222
deu	4 378 446	10 967 174
eur	18 029 721	44 435 372
af_shell10	1 508 065	51 164 260

Table 5. Basic properties of the graphs from our benchmark set. The large instances are split into four groups: geometric graphs, FEM graphs, street networks, sparse matrices. Within their groups, the graphs are sorted by size.

Variant	(+Flow, -MB, -FM)				(+Flow, +MB, -FM)				(+Flow, -MB, +FM)				(+Flow, +MB, +FM)			
α'	Avg.	Best.	Bal.	t	Avg.	Best.	Bal.	t	Avg.	Best.	Bal.	t	Avg.	Best.	Bal.	t
16	3 031	2 888	1.025	4.17	2 950	2 841	1.023	3.92	2 802	2 704	1.025	4.30	2 774	2 688	1.023	5.01
8	3 044	2 905	1.025	2.11	2 962	2 855	1.023	2.07	2 806	2 705	1.025	2.41	2 778	2 693	1.023	2.72
4	3 126	2 963	1.024	1.24	3 041	2 933	1.021	1.29	2 825	2 723	1.025	1.62	2 800	2 706	1.022	1.76
2	3 374	3 180	1.022	0.90	3 274	3 107	1.018	0.96	2 869	2 758	1.024	1.31	2 855	2 746	1.021	1.39
1	3 698	3 488	1.018	0.76	3 587	3 410	1.016	0.80	2 926	2 804	1.024	1.19	2 923	2 802	1.023	1.22
(-Flow, -MB, +FM)	2 974	2 851	1.025	1.13												

Table 6. The final score of different algorithm configurations compared against the basic two-way FM configuration. Here α' is the flow region upper bound factor. The values are average values as described in Section 6.

Effectiveness (+Flow, +MB, -FM)	Avg.	Best.	Bal.
$\alpha' = 1$	3 389	3 351	1.016
2	3 088	3 049	1.017
4	2 922	2 892	1.022
8	2 865	2 841	1.023
16	2 870	2 839	1.023
(-Flow, -MB, +FM)	2 833	2 803	1.025

Effectiveness (+Flow,-MB, +FM)	Avg.	Best.	Bal.
$\alpha' = 1$	2 786	2 759	1.024
2	2 748	2 724	1.024
4	2 721	2 698	1.025
8	2 718	2 690	1.025
16	2 730	2 697	1.025
(-Flow, -MB, +FM)	2 831	2 801	1.025

Effectiveness (+Flow,+MB,+FM)	Avg.	Best.	Bal.
$\alpha' = 1$	2 781	2 754	1.023
2	2 735	2 711	1.021
4	2 702	2 682	1.022
8	2 699	2 675	1.023
16	2 711	2 682	1.022
(-Flow, -MB, +FM)	2 827	2 799	1.025

Table 7. Each table is the result of an effectiveness test for six different algorithm configurations. All values are average values as described in Section 6.

Algorithm	Avg.	Best.	Bal.	t	Eff. Avg.	Eff. Best.
2 F-cycle	2 895	2 773	1.023	2.31	2 806	2 760
3 V-cycle	2 895	2 776	1.023	2.49	2 810	2 766
2 W-cycle	2 889	2 765	1.024	2.77	2 810	2 760
1 W-cycle	2 934	2 810	1.024	1.38	2 815	2 773
1 F-cycle	2 941	2 813	1.024	1.18	2 816	2 783
2 V-cycle	2 918	2 796	1.024	1.67	2 817	2 778
1 V-cycle	2 973	2 841	1.024	0.85	2 834	2 801

Table 8. Test results for normal and effectiveness tests for different global search strategies and different parameters.

k	Strong			-Kway			-Multitry			-Cyc			-MB			-Flow		
	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t
2	561	548	2.85	561	548	2.87	564	549	2.68	568	549	1.42	575	551	1.33	627	582	0.85
4	1 286	1 242	5.13	1 287	1 236	5.28	1 299	1 244	4.26	1 305	1 248	2.40	1 317	1 254	2.18	1 413	1 342	1.02
8	2 314	2 244	7.52	2 314	2 241	7.82	2 345	2 273	5.34	2 356	2 279	3.11	2 375	2 295	2.70	2 533	2 441	1.32
16	3 833	3 746	11.26	3 829	3 735	11.73	3 907	3 813	6.40	3 937	3 829	3.79	3 970	3 867	3.32	4 180	4 051	1.80
32	6 070	5 936	16.36	6 064	5 949	17.12	6 220	6 087	7.72	6 269	6 138	4.77	6 323	6 177	4.20	6 573	6 427	2.60
64	9 606	9 466	25.09	9 597	9 449	26.09	9 898	9 742	9.69	9 982	9 823	6.35	10 066	9 910	5.71	10 359	10 199	3.94
Avg.	2 683	2 617	8.93	2 682	2 614	9.23	2 729	2 656	5.55	2 748	2 668	3.27	2 773	2 686	2.92	2 934	2 823	1.66

Effectiveness	Strong		-Kway		-Multitry		-Cyc		-MB		-Flow	
k	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.
2	550	547	550	548	550	548	549	548	552	549	581	573
4	1 251	1 240	1 251	1 243	1 257	1 246	1 255	1 245	1 263	1 252	1 316	1 299
8	2 263	2 242	2 270	2 249	2 280	2 267	2 277	2 263	2 289	2 273	2 408	2 387
16	3 773	3 745	3 769	3 742	3 830	3 795	3 828	3 799	3 846	3 813	4 029	3 996
32	6 000	5 943	6 001	5 947	6 116	6 078	6 139	6 099	6 170	6 128	6 403	6 369
64	9 523	9 463	9 502	9 437	9 745	9 702	9 811	9 754	9 881	9 829	10 139	10 085
Avg.	2 636	2 616	2 636	2 618	2 668	2 650	2 669	2 653	2 684	2 666	2 799	2 775

Table 9. Removal tests: each configuration is same as left neighbor minus the component shown at the top of the column. The first table shows detailed results for all k in a normal test. The second table shows the results for an effectivity test.

k	Strong			-Kway			-Multitry			-Cyc			-MB			-Flow		
	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t
2	561	548	2.85	0.00	0.00	2.87	0.53	0.18	2.68	1.25	0.18	1.42	2.50	0.55	1.33	11.76	6.20	0.85
4	1 286	1 242	5.13	0.08	−0.48	5.28	1.01	0.16	4.26	1.48	0.48	2.40	2.41	0.97	2.18	9.88	8.05	1.02
8	2 314	2 244	7.52	0.00	−0.13	7.82	1.34	1.29	5.34	1.82	1.56	3.11	2.64	2.27	2.70	9.46	8.78	1.32
16	3 833	3 746	11.26	−0.10	−0.29	11.73	1.93	1.79	6.40	2.71	2.22	3.79	3.57	3.23	3.32	9.05	8.14	1.80
32	6 070	5 936	16.36	−0.10	0.22	17.12	2.47	2.54	7.72	3.28	3.40	4.77	4.17	4.06	4.20	8.29	8.27	2.60
64	9 606	9 466	25.09	−0.09	−0.18	26.09	3.04	2.92	9.69	3.91	3.77	6.35	4.79	4.69	5.71	7.84	7.74	3.94
Avg.	2 683	2 617	8.93	−0.04	−0.11	9.23	1.71	1.49	5.55	2.42	1.95	3.27	3.35	2.64	2.92	9.36	7.87	1.66

Effectiveness	Strong		-Kway		-Multitry		-Cyc		-MB		-Flow	
k	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.
2	550	547	0.00	0.18	0.00	0.18	−0.18	0.18	0.36	0.37	5.64	4.75
4	1 251	1 240	0.00	0.24	0.48	0.48	0.32	0.40	0.96	0.97	5.20	4.76
8	2 263	2 242	0.31	0.31	0.75	1.12	0.62	0.94	1.15	1.38	6.41	6.47
16	3 773	3 745	−0.11	−0.08	1.51	1.34	1.46	1.44	1.93	1.82	6.79	6.70
32	6 000	5 943	0.02	0.07	1.93	2.27	2.32	2.62	2.83	3.11	6.72	7.17
64	9 523	9 463	−0.22	−0.27	2.33	2.53	3.02	3.08	3.76	3.87	6.47	6.57
Avg.	2 636	2 616	0.00	0.08	1.21	1.30	1.25	1.41	1.82	1.91	6.18	6.08

Table 10. Removal tests: each configuration is same as its left neighbor minus the component shown at the top of the column. The first table shows detailed results for all k in a normal test. The second table shows the results for an effectivity test. All values are increases in cut are relative to the values obtained by KaFFPa Strong.

k	Strong			-Kway			-Multitry			-MB			-Flows		
	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t
2	561	548	2.85	561	548	2.86	561	548	2.72	564	548	2.70	582	559	1.94
4	1 286	1 242	5.14	1 287	1 236	5.29	1 293	1 240	4.23	1 290	1 239	4.68	1 312	1 252	2.95
8	2 314	2 244	7.52	2 314	2 241	7.81	2 337	2 271	5.24	2 322	2 249	6.88	2 347	2 270	4.88
16	3 833	3 746	11.19	3 829	3 735	11.69	3 894	3 799	6.27	3 838	3 747	10.41	3 870	3 779	8.22
32	6 070	5 936	16.38	6 064	5 949	17.15	6 189	6 055	7.67	6 082	5 948	15.42	6 110	5 977	13.17
64	9 606	9 466	25.08	9 597	9 449	26.02	9 834	9 680	9.78	9 617	9 478	24.02	9 646	9 509	21.19
Avg.	2 683	2 617	8.93	2 682	2 614	9.23	2 717	2 646	5.52	2 690	2 619	8.34	2 724	2 643	6.33

Effectiveness	Strong		-Kway		-Multitry		-MB		-Flows	
k	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.
2	550	547	550	548	550	548	550	548	560	556
4	1 251	1 240	1 251	1 243	1 254	1 243	1 251	1 241	1 266	1 252
8	2 263	2 242	2 270	2 249	2 276	2 262	2 270	2 246	2 281	2 259
16	3 771	3 742	3 767	3 741	3 810	3 781	3 773	3 747	3 797	3 767
32	6 000	5 943	6 002	5 950	6 090	6 055	6 006	5 955	6 028	5 977
64	9 523	9 463	9 502	9 437	9 681	9 636	9 525	9 470	9 548	9 494
Avg.	2 636	2 616	2 636	2 618	2 658	2 642	2 639	2 619	2 659	2 637

Table 11. Knockout tests: each configuration is the same as KaFFPa Strong minus the component shown at the top of the column. The first table shows detailed results for all k in a normal test. The second table shows the results for an effectivity test.

k	Strong			-Kway			-Multitry			-MB			-Flows		
	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t
2	561	548	2.85	0.00	0.00	2.86	0.00	0.00	2.72	0.53	0.00	2.70	3.74	2.01	1.94
4	1 286	1 242	5.14	0.08	−0.48	5.29	0.54	−0.16	4.23	0.31	−0.24	4.68	2.02	0.81	2.95
8	2 314	2 244	7.52	0.00	−0.13	7.81	0.99	1.20	5.24	0.35	0.22	6.88	1.43	1.16	4.88
16	3 833	3 746	11.19	−0.10	−0.29	11.69	1.59	1.41	6.27	0.13	0.03	10.41	0.97	0.88	8.22
32	6 070	5 936	16.38	−0.10	0.22	17.15	1.96	2.00	7.67	0.20	0.20	15.42	0.66	0.69	13.17
64	9 606	9 466	25.08	−0.09	−0.18	26.02	2.37	2.26	9.78	0.11	0.13	24.02	0.42	0.45	21.19
Avg.	2 683	2 617	8.93	−0.04	−0.11	9.23	1.27	1.11	5.52	0.26	0.08	8.34	1.53	0.99	6.33

Effectiveness	Strong		-Kway		-Multitry		-MB		-Flows	
k	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.	Avg.	Best.
2	550	547	0.00	0.18	0.00	0.18	0.00	0.18	1.82	1.65
4	1 251	1 240	0.00	0.24	0.24	0.24	0.00	0.08	1.20	0.97
8	2 263	2 242	0.31	0.31	0.57	0.89	0.31	0.18	0.80	0.76
16	3 771	3 742	−0.11	−0.03	1.03	1.04	0.05	0.13	0.69	0.67
32	6 000	5 943	0.03	0.12	1.50	1.88	0.10	0.20	0.47	0.57
64	9 523	9 463	−0.22	−0.27	1.66	1.83	0.02	0.07	0.26	0.33
Avg.	2 636	2 616	0.00	0.08	0.83	0.99	0.11	0.11	0.87	0.80

Table 12. Knockout tests: each configuration is the same as KaFFPa Strong minus the component shown at the top of the column. The first table shows detailed results for all k in a normal test. The second table shows the results for an effectivity test. All values are increases in cut relative to the values obtained by KaFFPa Strong.

		KaFFPa Strong			KaFFPa Eco			KaFFPa Fast			KaSPar Strong			KaPPa Strong			DiBaP			Scotch			Metis		
graph	k	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t
fe_tooth	2	3 789	3 829	5.43	4 159	4 594	0.13	4 308	4 491	0.12	3 844	3 987	5.86	3 951	4 336	3.75	4 390	4 785	0.99	3 945	4 312	0.36	4 319	4 695	0.09
fe_tooth	4	6 812	6 946	12.62	7 378	7 438	0.38	8 047	8 773	0.13	6 937	6 999	8.54	7 012	7 189	5.22	7 492	8 081	1.11	7 464	7 770	0.66	7 853	8 155	0.10
fe_tooth	8	11 595	11 667	18.22	11 995	12 670	0.58	12 909	13 367	0.17	11 482	11 564	13.43	12 272	12 721	6.83	12 186	12 532	1.79	12 638	12 953	1.04	12 976	13 728	0.10
fe_tooth	16	17 907	18 056	27.53	18 812	19 182	0.81	19 753	20 387	0.21	17 744	17 966	21.24	18 302	18 570	7.18	19 389	19 615	2.86	19 179	19 761	1.52	20 145	20 196	0.11
fe_tooth	32	25 585	25 738	41.42	26 945	27 320	1.27	28 471	29 108	0.28	25 888	26 248	35.12	26 397	26 617	5.28	26 518	27 073	5.06	27 852	28 680	2.03	28 699	28 909	0.12
fe_tooth	64	35 497	35 597	57.23	37 353	37 864	1.80	39 547	39 843	0.41	36 259	36 469	49.65	36 862	37 002	4.71	37 271	37 458	8.78	39 013	39 208	2.60	39 164	39 403	0.13
598a	2	2 367	2 372	7.73	2 388	2 388	0.37	2 546	2 547	0.22	2 371	2 384	6.50	2 387	2 393	5.64	2 414	2 435	1.90	2 409	2 414	0.38	2 485	2 530	0.17
598a	4	7 896	7 993	13.29	8 141	8 190	0.59	8 415	8 700	0.25	7 897	7 921	11.15	8 235	8 291	10.24	8 200	8 200	2.40	8 214	8 256	0.92	8 351	8 737	0.18
598a	8	15 830	16 182	25.60	16 565	16 764	0.89	18 361	20 561	0.30	15 929	15 984	22.31	16 502	16 641	12.21	16 585	16 663	3.59	16 949	17 203	1.54	17 501	18 019	0.19
598a	16	26 211	26 729	41.81	27 639	27 941	1.48	28 955	29 571	0.41	26 046	26 270	38.39	26 467	26 825	17.74	26 693	27 131	6.14	28 932	29 415	2.28	29 377	30 149	0.20
598a	32	39 863	39 976	68.82	41 553	42 012	2.20	43 746	44 365	0.55	39 625	40 019	60.60	40 946	41 190	18.16	40 908	41 456	10.97	43 960	44 232	3.08	42 986	43 910	0.22
598a	64	57 325	57 860	107.20	60 519	60 838	3.14	62 993	63 677	0.75	58 362	58 945	87.52	59 148	59 387	14.15	58 978	59 371	18.50	64 071	64 380	4.00	62 293	62 687	0.24
fe_ocean	2	311	311	5.27	311	311	0.20	372	376	0.10	317	317	5.55	314	317	3.21	348	1 067	0.62	398	400	0.18	523	524	0.13
fe_ocean	4	1 789	1 789	9.36	1 801	1 809	0.34	1 938	2 085	0.11	1 801	1 810	9.40	1 756	1 822	6.30	1 994	1 994	0.70	1 964	2 026	0.41	2 126	2 183	0.13
fe_ocean	8	4 012	4 087	13.58	4 675	4 826	0.43	5 976	6 299	0.13	4 044	4 097	14.33	4 104	4 252	6.33	5 208	5 305	1.24	4 448	4 596	0.77	5 369	5 502	0.14
fe_ocean	16	7 966	8 087	21.14	8 794	8 991	0.71	10 047	10 299	0.20	7 992	8 145	22.41	8 188	8 350	5.62	9 356	9 501	1.97	9 025	9 180	1.25	9 886	10 015	0.15
fe_ocean	32	12 660	12 863	31.73	14 487	14 898	1.25	16 266	16 590	0.28	13 320	13 518	36.53	13 593	13 815	4.34	15 893	16 230	3.09	14 971	15 239	1.78	15 456	15 908	0.17
fe_ocean	64	20 606	20 739	66.39	22 241	22 590	2.01	24 421	24 728	0.42	21 326	21 739	62.46	21 636	21 859	3.68	24 692	24 894	6.02	22 270	22 887	2.40	24 448	24 737	0.19
144	2	6 451	6 482	16.12	6 616	6 625	0.52	6 803	6 911	0.28	6 455	6 507	12.81	6 559	6 623	7.45	7 146	7 146	2.38	6 702	7 046	0.63	6 753	6 837	0.25
144	4	15 485	15 832	34.62	16 238	16 334	0.92	16 557	17 363	0.32	15 312	15 471	24.73	16 870	16 963	13.33	16 169	16 550	3.17	16 843	17 315	1.41	17 119	17 636	0.26
144	8	25 282	25 626	53.65	26 606	26 934	1.40	29 298	30 489	0.38	25 130	25 409	38.13	26 300	26 457	20.11	26 121	26 871	4.54	28 674	29 257	2.16	27 892	28 475	0.27
144	16	38 483	38 669	85.52	40 312	40 992	2.10	42 762	43 415	0.52	37 872	38 404	69.35	39 010	39 319	26.04	39 618	40 066	7.77	42 591	43 291	3.01	42 643	43 399	0.28
144	32	56 672	56 827	121.75	59 423	59 866	2.90	62 353	63 039	0.66	57 082	57 492	106.40	58 331	58 631	24.60	57 683	58 592	13.03	62 627	63 215	3.99	62 345	62 792	0.30
144	64	78 828	79 477	147.98	83 510	84 464	3.85	87 268	88 082	0.87	80 313	80 770	144.77	82 286	82 452	19.11	81 997	82 216	23.23	87 475	88 341	5.16	85 861	86 426	0.34
wave	2	8 665	8 681	14.23	9 017	9 100	0.39	9 778	10 847	0.26	8 661	8 720	16.19	8 832	9 132	8.24	8 994	10 744	2.03	9 037	9 144	0.79	9 136	9 499	0.23
wave	4	16 804	16 908	38.36	18 464	18 834	0.84	17 927	22 697	0.30	16 806	16 920	29.56	17 008	17 250	14.51	17 382	17 608	2.53	19 454	19 945	1.69	20 652	22 060	0.25
wave	8	28 882	29 339	62.99	30 753	31 248	1.51	33 268	36 900	0.37	28 681	28 817	46.61	30 690	31 419	20.63	29 893	32 246	3.74	32 592	33 285	2.54	33 174	34 384	0.27
wave	16	42 292	43 538	97.53	45 605	46 647	2.10	47 632	48 176	0.49	42 918	43 208	75.97	44 831	45 048	20.54	45 227	45 596	6.33	48 233	49 139	3.50	47 686	48 594	0.27
wave	32	62 566	62 647	124.43	65 301	65 871	3.06	67 029	68 692	0.63	63 025	63 159	112.19	63 981	64 390	14.94	63 594	64 464	10.51	68 645	69 469	0.29			
wave	64	84 970	85 649	195.61	89 886	90 743	4.03	93 700	94 326	0.84	87 243	87 554	150.37	88 376	88 964	12.51	87 741	88 487	18.61	95 627	95 983	5.87	93 232	93 592	0.33
m14b	2	3 823	3 823	19.82	3 826	3 826	0.90	4 136	4 151	0.46	3 828	3 846	20.03	3 862	3 954	11.16	3 898	3 941	3.53	3 861	3 910	0.67	3 981	4 220	0.39
m14b	4	12 953	13 031	38.87	13 368	13 401	1.34	14 096	14 196	0.51	13 015	13 079	26.51	13 543	13 810	18.77	13 494	13 519	4.73	13 408	13 528	1.59	13 881	14 070	0.40
m14b	8	26 006	26 179	65.15	26 958	27 230	2.07	28 388	29 438	0.59	25 573	25 756	45.33	27 330	27 393	24.97	26 743	26 916	7.10	27 664	27 786	2.67	28 009	29 373	0.42
m14b	16	43 176	43 759	91.08	45 143	46 377	3.04	48 678	49 529	0.78	42 212	42 458	83.25	45 352	45 762	28.11	44 666	45 515	12.76	49 015	49 968	4.03	47 828	49 342	0.43
m14b	32	67 417	67 512	142.37	70 875	71 369	4.29	72 729	74 109	1.00	66 314	66 991	133.88	68 107	69 075	29.94	67 888	68 957	22.30	73 291	74 200	5.48	73 500	74 476	0.46
m14b	64	98 222	98 536	189.96	103 705	104 460	5.48	108 504	109 706	1.30	99 207	100 014	198.23	101 053	101 455	25.26	99 994	100 653	37.38	109 021	109 844	7.21	105 591	107 296	0.50
auto	2	9 725	9 775	74.25	9 739	9 837	2.30	10 282	10 517	1.03	9 740	9 768	68.39	9 910	10 045	30.09	10 094	11 494	6.95	10 243	11 525	1.53	10 611	10 744	1.01
auto	4	25 841	25 891	151.14	26 594	26 858	3.25	38 710	42 402	1.10	25 988	26 062	75.60	28 218	29 481	64.01	26 523	27 958	9.93	28 269	28 695	3.28	29 131	30 828	1.02
auto	8	44 847	45 299	257.71	46 263	48 104	5.47	51 725	55 373	1.20	45 099	45 232	97.60	46 272	46 652	85.89	48 326	48 346	14.24	49 596	50 080	5.08	50 188	52 740	1.05
auto	16	75 792	77 429	317.81	79 129	80 116	7.31	83 190	86 195	1.63	76 287	76 715	153.46	78 713	79 769	87.41	80 198	81 742	24.60	83 506	84 254	7.35	83 717	87 104	1.08
auto	32	121 016	121 687	366.47	126 261	127 037	9.86	131 608	133 300	2.05	121 269	121 862	246.50	124 606	125 500	71.77	124 443	125 043	40.77	131 481	132 960	10.11	134 554	135 459	1.14
auto	64	173 155	173 624	490.74	181 173	182 964	11.87	187 766	189 928	2.61	174 612	174 914	352.09	177 038	177 595	62.64	175 091	175 758	66.23	190 464	192 242	13.27	188 572	189 695	1.23
delaunay_n20	2	1 680	1 687	57.94	1 725	1 744	2.55	2 021	2 051	1.09	1 711	1 731	196.33	1 858	1 882	35.43	1 994	2 265	2.91	1 859	1 873	1.11	2 042	2 105	1.31
delaunay_n20	4	3 368	3 380	124.29	3 393	3 414	4.19	3 931	3 996	1.11	3 418	3 439	130.67	3 674	3 780	64.08	3 804	3 804	3.05	3 688	3 753	2.17	3 970	4 121	1.32
delaunay_n20	8	6 247	6 283	154.95	6 328	6 404	5.11	7 681	7 877	1.13	6 278	6 317	104.37	6 670	6 854	70.07	6 923	7 102	5.02	7 174	7 319	3.29	7 804	7 929	1.33
delaunay_n20	16	10 012	10 056	210.39	10 291	10 375	5.37	11 756	12 011	1.18	10 183	10 218	84.33	10 816	11 008	67.92	11 174	11 382	8.01	11 107	11 187	4.30	12 320	12 471	1.33
delaunay_n20	32	15 744	15 804	220.40	16 306	16 502	6.85	18 802	19 251	1.27	15 905	16 026	101.69	16 813	17 086	42.67	17 343	17 408	13.60	17 818	17 949	5.49	18 860	19 304	1.38
delaunay_n20	64	23 472	23 551	237.76	24 383	24 547	7.86	27 615	27 828	1.40	23 935	23 962	97.09	24 799	25 179	22.04	25 884	26 148	23.94	25 982	26 113	6.86	27 849	28 419	1.38
rgg_n_2_20_s0	2	2 088	2 119	94.68	2 177	2 177	3.96	2 824	2 944	1.15	2 162	2 201	198.61	2 377	2 498	33.24				2 596	2 728	1.29	2 941	3 112	1.81
rgg_n_2_20_s0	4	4 184	4 241	167.88	4 308	4 313	7.34	5 713	5 847	1.17	4 323	4 389	130.00	4 867	5 058	38.50				5 580	5 712	2.63	5 870	5 980	1.82
rgg_n_2_20_s0	8	7 684	7 729	192.45	8 123	8 324	7.63	10 524	11 139	1.20	7 745	7 915	103.66	8 995	9 391	46.06				10 812	11 164	4.10	10 411	12 002	1.80
rgg_n_2_20_s0	16	12 504	12 673	205.29	13 281	13 675	8.16	17 378	17 997	1.30	12 596	12 792	86.19	14 953	15 199	35.86				16 311	16 687	5.54	17 773	18 221	1.80
rgg_n_2_20_s0	32	20 078	20 400	207.80	21 311	21 897	8.83	27 936	28 428	1.42	20 403	20 478	100.03	23 430	23 917	26.04				26 262	26 666	7.17	27 392	28 328	1.81
rgg_n_2_20_s0	64	30 518	30 893	230.28	33 166	33 603	9.85	41 537	42 137	1.58	30 860	31 066	97.83	34 778	35 354	11.62				38 401	38 958	8.98	42 274	42 666	1.86
af_shell10	2	26 225	26 225	367.08	28 700	28 700	12.53	29 900	30 260	2.51	26 225	26 225	317.11	26 225	26 225	78.65	26 225	26 225	3.74	26 225	28 980	3.43	27 575	30 230	3.72
af_shell10	4	53 450	53 825	1 326.09	54 500	55 165	22.35	57 150	58 290	2.54	55 075	55 345	210.61	54 950	55 265	91.96	56 075	56 075	4.93	56 075	57 305	7.05	60 750	61 975	3.76
af_shell10	8	94 350	96 667	1 590.61	111 975	112 650	24.81	116 875	117 894	2.59	97 709	100 233	179.51	101 425	102 335	136.99	107 125	108 400	7.53	107 025	109 685	11.01	115 475	118 725	3.73
af_shell10	16	152 050	155 092	2 154.59	162 250	164 383	22.85	180 100	182 705	2.71	163 125	165 770	212.12	165 025	166 427	106.63	168 450	171 940	11.98	168 850	170 160	15.23	185 325	188 795	3.75
af_shell10	32	238 575	242 992	1 803.05	259 450	260 911	24.48	288 900	291 758	2.83	248 268	252 939	191.53	253 525	255 535	80.85	255 850	258 795	19.74	268 000	270 945	20.13	286 600	288 250	3.78
af_shell10	64	356 975	360 867	1 945.30	382 321	385 210	25.08	406 925	410 505	2.99	372 823	376 512	207.76	379 125	382 923	43.01	382 675	387 624	34.20	395 900	397 565	25.24	423 432	428 881	3.83
deu	2	163	166	197.17	181	185	10.22	237	257	4.87	167	172	231.47	214	221	68.20				265	279	2.96	271	296	6.18
deu	4	395	403	314.83	407	438	14.84	597	651	4.92	419	426	244.12	533	542	76.87				608	648	6.03	592	710	6.07
deu	8	726	729	350.84	781	809	17.18	1 087	1 143	4.93	762	773	250.50	922	962	99.76				1 109	1 211	9.07	1 209	1 600	6.02
deu	16	1 263	1 278	423.09	1 376	1 418	17.34	1 808	1 857	4.96	1 308	1 333	278.31	1 550	1 616	105.96				1 957	2 061	12.05	2 052	2 191	5.93
deu	32	2 115	2 146	460.84	2 230	2 338	20.57	2 951	3 076	5.02	2 182	2 217	283.79	2 548	2 615	73.17				3 158	3 262	15.12	3 225	3 607	5.92
deu	64	3 432	3 440	512.77	3 724	3 800	24.91	4 659	4 770	5.15	3 610	3 631	293.53	4 021	4 093	49.55				4 799	4 937	18.24	4 985	5 320	5.96
eur	2	130	130	1 013.00	214	246	61.35	423	434	22.33	133	138	1 946.34							369	448	11.86	412	454	33.00
eur	4	412	430	1 823.90	468	496	102.19	632	815	22.44	355	375	2 168.10	543	619	441.11				727	851	23.86	902	1 698	32.46
eur	8	749	772	2 067.02	831	875	108.79	1 280	1 334	22.48	774	786	2 232.31	986	1 034	418.29				1 338	1 461	35.99	2 473	3 819	33.01
eur	16	1 454	1 493	2 340.64	1 595	1 646	112.81	2 145	2 408	22.55	1 401	1 440	2 553.40	1 760	1 900	497.93				2 478	2 563	48.30	3 314	8 554	33.85
eur	32	2 428	2 504	2 445.72	2 747	2 777	120.06	3 865	3 918	22.65	2 595	2 643	2 598.84	3 186	3 291	417.52				4 057	4 249	60.29	5 811	7 380	32.84
eur	64	4 240	4 264	2 533.56	4 733	4 830	143.04	6 431	6 534	22.80	4 502	4 526	2 533.56	5 290	5 393	308.17				6 518	6 739	73.94	10 264	13 947	32.86

Table 13. Detailed per instance basis results for the large testset.

	KaFFPa Strong			KaFFPa Eco			KaFFPa Fast			KaSPar Strong		
k	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t
2	3 988	4 001	22.68	4 117	4 178	0.79	4 573	4 459	0.40	4 013	4 047	24.94
4	10 467	10 559	50.18	10 878	10 969	1.42	11 897	12 732	0.43	10 548	10 610	32.09
8	19 288	19 553	76.39	20 612	21 061	2.06	23 026	24 295	0.50	19 332	19 507	44.11
16	31 474	31 953	111.49	33 284	33 858	2.82	35 952	36 730	0.64	31 676	32 000	65.43
32	48 195	48 506	145.04	51 117	51 686	3.94	54 725	55 685	0.80	48 770	49 254	94.42
64	69 936	70 363	199.84	73 946	74 661	5.09	78 553	79 305	1.03	71 506	72 024	126.59
Avg.	20 986	21 172	80.93	22 088	22 393	2.25	23 952	24 742	0.60	21 185	21 364	54.97

	KaPPa Strong			DiBaP			Scotch			Metis		
	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t
2	4 089	4 180	11.63	4 285	5 155	2.25	4 238	4 430	0.71	4 543	4 722	0.39
4	10 940	11 168	19.76	11 133	11 341	2.79	11 336	11 581	1.53	11 906	12 355	0.40
8	20 255	20 609	25.46	20 980	21 451	4.31	21 391	21 805	2.46	22 416	23 195	0.42
16	32 821	33 219	26.66	33 859	34 389	7.19	35 007	35 562	3.54	36 275	37 006	0.43
32	50 085	50 573	21.84	51 088	51 773	12.14	53 628	54 323	4.75	54 669	55 437	0.46
64	72 837	73 316	16.44	74 144	74 676	21.17	77 379	78 042	6.14	78 415	79 200	0.50
Avg.	21 839	22 163	19.56	22 460	23 461	6.07	23 033	23 505	2.56	23 945	24 568	0.44

	KaFFPa Strong			KaFFPa Eco			KaFFPa Fast			KaSPar Strong		
k	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t
2	2 812	2 828	31.44	2 925	2 966	1.16	3 276	3 382	0.55	2 842	2 873	36.89
4	5 636	5 709	87.25	5 891	5 996	2.83	6 829	7 408	0.80	5 642	5 707	60.66
8	10 369	10 511	123.31	11 111	11 398	3.82	13 149	13 856	0.89	10 464	10 580	75.92
16	17 254	17 525	168.96	18 354	18 731	4.84	20 854	21 508	1.08	17 345	17 567	102.52
32	26 917	27 185	208.25	28 690	29 136	6.41	32 527	33 155	1.29	27 416	27 707	137.08
64	40 193	40 444	270.30	42 880	43 385	8.10	47 785	48 344	1.58	41 286	41 570	170.54
Avg.	12 054	12 182	121.50	12 763	12 988	3.82	14 562	15 124	0.98	12 450	12 584	87.12

	KaPPa Strong			DiBaP			Scotch			Metis		
	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t
2	2 977	3 054	15.03	-	-	-	3 151	3 298	0.85	3 379	3 535	0.58
4	6 190	6 384	30.31	-	-	-	6 661	6 909	2.26	7 049	7 770	0.83
8	11 375	11 652	37.86	-	-	-	12 535	12 939	3.58	13 719	15 118	0.85
16	18 678	19 061	39.13	-	-	-	20 716	21 153	5.06	22 041	24 396	0.88
32	29 156	29 562	31.35	-	-	-	32 183	32 751	6.69	33 820	35 289	0.92
64	43 237	43 237	22.36	-	-	-	47 109	47 714	8.55	49 972	51 970	0.98
Avg.	13 323	13 600	28.16	-	-	-	14 218	14 615	3.55	15 167	16 275	0.83

Table 14. Results for our large benchmark suite. The table on top contains average values for the comparison with DiBaP on our large testsuite without road networks and rgg. The table on the bottom contains average value for the comparisons with other general purpose partitioners on our large testsuite without the road network Europe for the case $k = 2$. The average values are computed as described in Section 6.

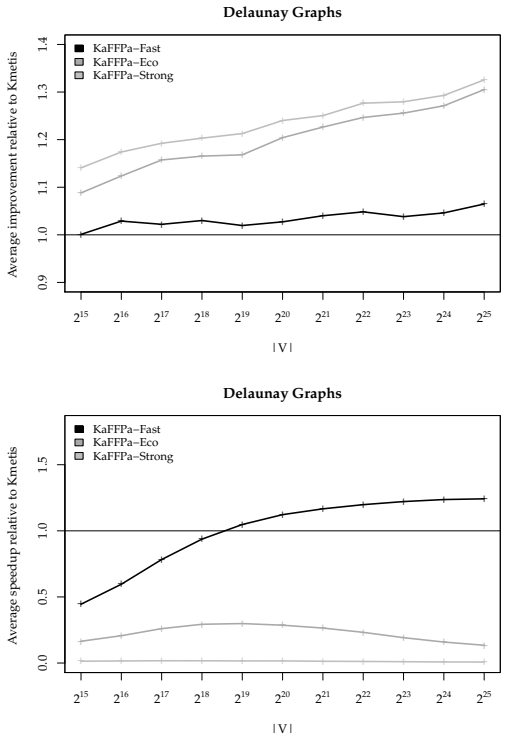


Fig. 12. Graph sequence test for Delaunay Graphs.

Graph/k	2		4		8		16		32		64	
3elt	**89**	89	**199**	199	**342**	342	571	**569**	987	**969**	1 595	**1 564**
add20	678	**594**	1 197	**1 177**	1 740	**1 704**	2 156	**2 121**	**2 565**	2 687	**3 071**	3 236
data	**188**	188	**378**	383	**659**	660	1 170	**1 162**	2 002	**1 865**	2 954	**2 885**
uk	**19**	19	**40**	41	**82**	84	**150**	152	260	**258**	**431**	438
add32	**10**	10	**30**	33	**66**	66	**117**	117	**212**	212	498	**493**
bcsstk33	**10 097**	10 097	21 556	**21 508**	34 183	**34 178**	55 447	**54 860**	79 324	**78 132**	110 656	**108 505**
whitaker3	**126**	126	**380**	380	**655**	656	1 105	**1 093**	**1 700**	1 717	2 588	**2 567**
crack	**183**	183	**362**	362	**677**	678	1 109	**1 092**	1 720	**1 707**	2 620	**2 566**
wing_nodal	**1 695**	1 696	3 576	**3 572**	5 445	**5 443**	**8 417**	8 422	12 129	**11 980**	16 332	**16 134**
fe_4elt2	**130**	130	**349**	349	**605**	605	**1 006**	1 014	**1 647**	1 657	2 575	**2 537**
vibrobox	11 538	**10 310**	**19 155**	19 199	24 702	**24 553**	34 384	**32 167**	42 711	**41 399**	49 924	**49 521**
bcsstk29	**2 818**	2 818	8 070	**8 035**	14 291	**13 965**	23 280	**21 768**	36 125	**34 886**	58 613	**57 054**
4elt	**138**	138	**320**	321	**534**	534	**938**	939	1 576	**1 559**	2 623	**2 596**
fe_sphere	**386**	386	**766**	768	**1 152**	1 152	**1 710**	1 730	**2 520**	2 565	3 670	**3 663**
cti	**318**	318	**944**	944	**1 752**	1 802	**2 865**	2 906	**4 180**	4 223	6 016	**5 875**
memplus	5 596	**5 489**	9 805	**9 559**	12 126	**11 785**	13 564	**13 241**	15 232	**14 395**	17 595	**16 857**
cs4	**366**	367	**938**	940	**1 455**	1 467	**2 124**	2 195	**2 990**	3 048	**4 141**	4 154
bcsstk31	**2 699**	2 701	**7 296**	7 444	**13 274**	13 371	24 546	**24 277**	38 860	**38 086**	60 612	**60 528**
fe_pwt	**340**	340	**704**	704	**1 437**	1 441	**2 799**	2 806	**5 552**	5 612	**8 314**	8 454
bcsstk32	**4 667**	4 667	**9 208**	9 247	21 253	**20 855**	**36 968**	37 372	62 994	**61 144**	97 299	**95 199**
fe_body	**262**	262	**598**	599	**1 040**	1 079	**1 806**	1 858	**2 968**	3 202	**5 057**	5 282
t60k	**75**	75	**208**	211	**454**	465	**818**	849	**1 361**	1 391	**2 143**	2 211
wing	**784**	787	**1 616**	1 666	**2 509**	2 589	**3 889**	4 131	**5 747**	5 902	**7 842**	8 132
brack2	**708**	708	**3 013**	3 027	**7 110**	7 144	**11 745**	11 969	**17 751**	17 798	26 766	**26 557**
finan512	**162**	162	**324**	324	**648**	648	**1 296**	1 296	**2 592**	2 592	10 752	**10 560**
fe_tooth	**3 815**	3 819	**6 870**	6 938	**11 492**	11 650	**17 592**	18 115	**25 695**	25 977	**35 722**	35 980
fe_rotor	**2 031**	2 045	7 538	**7 405**	13 032	**12 959**	20 888	**20 773**	**32 678**	32 783	47 980	**47 461**
598a	**2 388**	2 388	**7 956**	7 992	**16 050**	16 179	**25 892**	26 196	**40 003**	40 513	**57 795**	59 098
fe_ocean	**387**	387	**1 831**	1 856	**4 140**	4 251	**8 035**	8 276	**13 224**	13 660	**20 828**	21 548
144	**6 478**	6 479	15 635	**15 196**	**25 281**	25 455	**38 221**	38 940	**56 897**	58 126	**80 451**	81 145
wave	**8 665**	8 682	**16 881**	16 891	**29 124**	29 207	**43 027**	43 697	**62 567**	64 198	**86 127**	88 863
m14b	**3 826**	3 826	**12 981**	13 034	**25 854**	25 921	**42 358**	42 513	**67 454**	67 770	**99 661**	101 551
auto	**9 958**	10 004	**26 669**	26 941	45 892	**45 731**	**77 163**	77 618	**121 645**	123 296	**174 527**	175 975

Table 15. Computing partitions from scratch $\epsilon = 1\%$. In each k-column the results computed by KaFFPa are on the left and the current Walshaw cuts are presented on the right side.

Graph/k	2		4		8		16		32		64	
3elt	**87**	87	**198**	198	**335**	336	**563**	565	962	**958**	1 558	**1 542**
add20	702	**576**	1 186	**1 158**	1 724	**1 690**	2 104	**2 095**	**2 490**	2 493	**3 035**	3 152
data	**185**	185	**369**	378	**640**	650	**1 127**	1 133	1 846	**1 802**	2 922	**2 809**
uk	**18**	18	**39**	40	**78**	81	**141**	148	**245**	251	418	**414**
add32	**10**	10	**30**	33	**66**	66	**117**	117	**212**	212	496	**493**
bcsstk33	**10 064**	10 064	**20 865**	21 035	**34 078**	34 078	54 847	**54 510**	78 129	**77 672**	108 668	**107 012**
whitaker3	**126**	126	**378**	378	**652**	655	**1 090**	1 092	**1 680**	1 686	2 539	**2 535**
crack	**182**	182	**360**	360	**673**	676	1 086	**1 082**	1 692	**1 679**	2 561	**2 553**
wing_nodal	**1 678**	1 680	**3 545**	3 561	**5 374**	5 401	**8 315**	8 316	11 963	**11 938**	16 097	**15 971**
fe_4elt2	**130**	130	**342**	343	**597**	598	**996**	1 007	**1 621**	1 633	**2 513**	2 527
vibrobox	11 538	**10 310**	18 975	**18 778**	24 268	**24 171**	33 721	**31 516**	42 159	**39 592**	49 270	**49 123**
bcsstk29	**2 818**	2 818	7 993	**7 983**	13 867	**13 817**	22 494	**21 410**	34 892	**34 407**	56 682	**55 366**
4elt	**137**	137	**319**	319	**523**	523	918	**914**	1 539	**1 537**	**2 570**	2 581
fe_sphere	**384**	384	**764**	764	**1 152**	1 152	**1 705**	1 706	2 483	**2 477**	3 568	**3 547**
cti	**318**	318	**916**	917	**1 714**	1 716	**2 773**	2 778	**4 029**	4 132	**5 683**	5 763
memplus	5 466	**5 355**	9 593	**9 418**	12 085	**11 628**	13 384	**13 130**	15 124	**14 264**	17 183	**16 724**
cs4	**360**	361	**928**	936	**1 439**	1 467	**2 090**	2 126	**2 935**	3 014	**4 080**	4 107
bcsstk31	**2 676**	2 676	**7 150**	7 181	**13 020**	13 246	23 536	**23 504**	38 048	**37 459**	58 738	**58 667**
fe_pwt	**340**	340	**700**	704	**1 411**	1 416	**2 776**	2 784	**5 496**	5 606	**8 228**	8 346
bcsstk32	**4 667**	4 667	**8 742**	8 778	20 223	**20 035**	**35 572**	35 788	60 766	**59 824**	**92 094**	92 690
fe_body	**262**	262	**598**	598	**1 016**	1 033	**1 734**	1 767	**2 810**	2 906	**4 799**	4 982
t60k	**71**	71	**203**	207	**449**	454	**805**	822	**1 343**	1 391	**2 115**	2 198
wing	**773**	774	**1 605**	1 636	**2 471**	2 551	**3 862**	4 015	**5 645**	5 832	**7 727**	8 043
brack2	**684**	684	**2 834**	2 839	**6 871**	6 980	**11 462**	11 622	**17 211**	17 491	**26 026**	26 366
finan512	**162**	162	**324**	324	**648**	648	**1 296**	1 296	**2 592**	2 592	10 629	**10 560**
fe_tooth	**3 788**	3 792	**6 796**	6 862	**11 313**	11 422	**17 318**	17 655	**25 208**	25 624	**35 044**	35 830
fe_rotor	**1 959**	1 960	**7 128**	7 182	**12 479**	12 546	20 397	**20 356**	**31 345**	31 763	**46 783**	47 049
598a	**2 367**	2 367	**7 842**	7 873	**15 740**	15 820	**25 704**	25 927	**38 803**	39 525	**57 070**	58 101
fe_ocean	**311**	311	**1 696**	1 698	**3 921**	3 974	**7 648**	7 838	**12 550**	12 746	**20 049**	21 033
144	**6 438**	6 438	15 128	**15 122**	**25 119**	25 301	**37 782**	37 899	**56 399**	56 463	**78 626**	80 621
wave	**8 594**	8 616	**16 668**	16 822	**28 513**	28 664	**42 308**	42 620	**61 756**	62 281	**85 254**	86 663
m14b	**3 823**	3 823	**12 948**	12 977	**25 522**	25 550	**42 015**	42 061	66 401	**65 879**	**96 881**	100 064
auto	**9 683**	9 716	**25 836**	25 979	**44 841**	45 109	**75 792**	76 016	**120 174**	120 534	**171 584**	172 357

Table 16. Computing partitions from scratch $\epsilon = 3\%$. In each k-column the results computed by KaFFPa are on the left and the current Walshaw cuts are presented on the right side.

Graphi/k	2		4		8		16		32		64	
3elt	**87**	87	**197**	197	**330**	330	**558**	560	952	**950**	**1 528**	1 539
add20	691	**550**	1 171	**1 157**	1 703	**1 675**	2 112	**2 081**	**2 440**	2 463	**2 996**	3 152
data	182	**181**	**363**	368	629	**628**	1 092	**1 086**	1 813	**1 777**	2 852	**2 798**
uk	**18**	18	**39**	39	**76**	78	**139**	139	**242**	246	**404**	410
add32	**10**	10	**30**	33	**63**	63	**117**	117	**212**	212	**486**	491
bcsstk33	**9 914**	9 914	20 216	**20 198**	**33 922**	33 938	54 692	**54 323**	77 564	**77 163**	107 832	**106 886**
whitaker3	**126**	126	**378**	378	**647**	650	1 087	**1 084**	**1 673**	1 686	**2 512**	2 535
crack	**182**	182	**360**	360	**667**	667	**1 077**	1 080	1 682	**1 679**	**2 526**	2 548
wing_nodal	1 669	**1 668**	**3 524**	3 536	**5 346**	5 350	**8 266**	8 316	**11 855**	11 879	16 111	**15 873**
fe_4elt2	**130**	130	**335**	335	**581**	583	**986**	991	**1 600**	1 633	**2 493**	2 516
vibrobox	11 486	**10 310**	18 856	**18 778**	23 948	**23 930**	33 113	**31 235**	41 812	**39 592**	48 841	**48 200**
bcsstk29	**2 818**	2 818	7 942	**7 936**	**13 575**	13 614	21 971	**20 924**	34 452	**33 818**	55 873	**54 935**
4elt	**137**	137	**315**	315	**516**	516	**901**	902	**1 520**	1 532	**2 554**	2 565
fe_sphere	**384**	384	**762**	764	**1 152**	1 152	**1 688**	1 692	**2 433**	2 477	**3 535**	3 547
cti	**318**	318	**889**	890	**1 684**	1 708	2 735	**2 725**	**3 957**	4 037	**5 609**	5 684
memplus	5 362	**5 267**	9 690	**9 299**	12 078	**11 555**	13 349	**13 078**	14 992	**14 170**	16 758	**16 454**
cs4	**353**	356	**922**	936	**1 435**	1 467	**2 083**	2 126	**2 923**	2 958	4 055	**4 052**
bcsstk31	**2 670**	2 676	**7 088**	7 099	**12 865**	12 941	**23 202**	23 254	**37 282**	37 459	57 748	**57 534**
fe_pwt	**340**	340	**700**	700	**1 405**	1 405	**2 748**	2 772	**5 431**	5 545	**8 136**	8 310
bcsstk32	**4 622**	4 622	**8 441**	8 454	**19 601**	19 678	**35 014**	35 208	**59 456**	59 824	91 110	**91 006**
fe_body	**262**	262	**589**	596	**1 014**	1 017	**1 701**	1 723	**2 787**	2 807	**4 642**	4 834
t60k	**65**	65	**195**	196	**445**	454	**801**	818	**1 337**	1 376	**2 106**	2 168
wing	**770**	770	**1 597**	1 636	**2 456**	2 528	**3 842**	3 998	**5 586**	5 806	**7 651**	7 991
brack2	**660**	660	**2 731**	2 739	**6 634**	6 671	**11 240**	11 358	**17 137**	17 256	**25 827**	26 281
finan512	**162**	162	**324**	324	**648**	648	**1 296**	1 296	**2 592**	2 592	10 604	**10 560**
fe_tooth	**3 773**	3 773	**6 718**	6 825	**11 185**	11 337	**17 230**	17 404	**24 977**	25 216	**34 704**	35 466
fe_rotor	**1 940**	1 950	**6 999**	7 045	**12 353**	12 380	**19 935**	20 132	**31 016**	31 450	**46 006**	46 608
598a	**2 336**	2 336	**7 738**	7 763	**15 502**	15 544	**25 560**	25 585	**38 884**	39 144	**56 586**	57 412
fe_ocean	**311**	311	**1 686**	1 697	**3 902**	3 941	**7 457**	7 618	**12 373**	12 720	**19 764**	20 667
144	**6 361**	6 362	15 321	**15 122**	25 078	**25 025**	37 505	**37 433**	**56 041**	56 463	**78 645**	79 296
wave	**8 535**	8 563	**16 543**	16 662	**28 493**	28 615	**42 179**	42 482	**61 386**	61 788	**84 247**	85 658
m14b	**3 802**	3 802	**12 945**	12 976	**25 151**	25 292	**41 538**	41 750	**65 087**	65 231	**96 580**	98 005
auto	**9 450**	9 450	**25 310**	25 399	**44 360**	44 520	75 195	**75 066**	**119 125**	120 001	**171 355**	171 459

Table 17. Computing partitions from scratch $\epsilon = 5\%$. In each k-column the results computed by KaFFPa are on the left and the current Walshaw cuts are presented on the right side.