

Engineering Multilevel Graph Partitioning Algorithms (Extended Abstract)

Peter Sanders, Christian Schulz

Email: sanders@kit.edu, christian.schulz@kit.edu

Karlsruhe Institute of Technology, Karlsruhe, Germany

Abstract. We present a multi-level graph partitioning algorithm using novel local improvement algorithms and global search strategies transferred from the multi-grid community. Local improvement algorithms are based 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 additional 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. 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 many aspects of the method are not well understood. We therefore have built our own graph partitioner KaPPa [17] (Karlsruhe Parallel Partitioner) and somewhat astonishingly, we found several opportunities for improvement with significant impact on partitioning quality and scalability. This motivated us to go a step backwards and look deeper into sequential graph partitioning with main focus on quality (total cut weight). That means we want to put every component of the multilevel approach on trial and engineer a graph partitioner that improves known methods especially for large graphs. This paper is a first step in this direction which focuses on local improvement methods and overall search strategies.

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 $\omega : E \rightarrow \mathbb{R}_{>0}$, node weights $c : V \rightarrow \mathbb{R}_{\geq 0}$, $n = |V|$, 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, \dots, V_k that partition V , i.e., $V_1 \cup \dots \cup V_k = V$ 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.

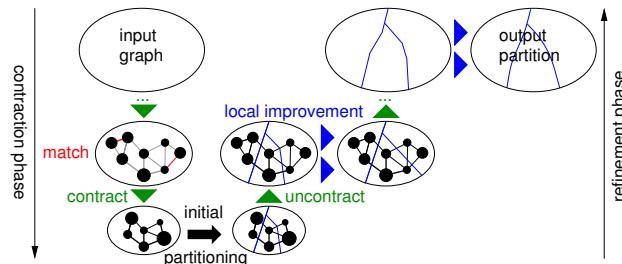


Fig. 1. Multilevel graph partitioning.

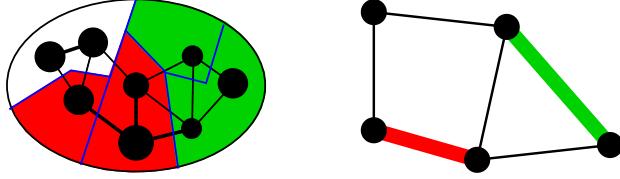


Fig. 2. A graph which is partitioned into five blocks and its corresponding quotient graph \mathcal{Q} which has five nodes and six edges. Two pairs of blocks are highlighted in red and green.

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 [17] namely the application of edge ratings, the GPA algorithm to compute high quality matchings, pairwise refinements between blocks and it also uses Scotch [22] as an initial partitioner [17].

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 expansion*²($\{u, v\}$) := $\frac{\omega(\{u, v\})^2}{c(u)c(v)}$ and innerOuter($\{u, v\}$) := $\frac{\omega(\{u, v\})}{\text{Out}(v) + \text{Out}(u) - 2\omega(u, v)}$ where $\text{Out}(v) := \sum_{x \in \Gamma(v)} \omega(\{v, x\})$, since they yielded the best results in [17]. The rating function expansion*² has the disadvantage that it evaluates to one on the first level of an unweighted graph. Therefore we use innerOuter on the first level to infer structural information of the graph and use expansion*² on the other levels of the hierarchy. 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 [19] as a synthesis of the Greedy algorithm and the Path Growing Algorithm [7]. We choose this algorithm since in [17] it gives empirically considerably better results than Sorted Heavy Edge Matching or Heavy Edge Matching [24].

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 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 [17], Scotch [22] produces better initial partitions than Metis, and therefore we also use it in KaFFPa.

Refinement Recall that the refinement phase iteratively uncontracts the matchings contracted during the contraction phase. After a matching is uncontracted, 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 refinements* [17]. The approach is a more local approach which 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 2-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 2-way local search algorithm works as in [17]. We present it here for completeness. It is basically the FM-algorithm [11]: 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 [11] alternates between both blocks. We employ the *TopGain* strategy from [17] 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 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 [11]. 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 balanced cut 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 [21]. 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. [21] derived that it is unlikely that the local search will produce a better cut if

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

for some tuning parameters α and β . 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 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 [21]. This method initializes the priority queue with only a single boundary node and its neighbors which are also boundary nodes.

The main difference of our implementation to the implementation in [21] is that we use only one priority queue. In [21] each block maintains a priority queue. 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 [12,24,30] for more material. All successful 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 [25,9] 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 [16], Jostle [30], Metis [24], Party [8], and Scotch [22].

KaSPar [21] is a new graph partitioner based on the central idea to (un)contract only a single edge between two levels. It yields the best partitioning results for many of the biggest graphs in [27].

KaPPa [17] 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.

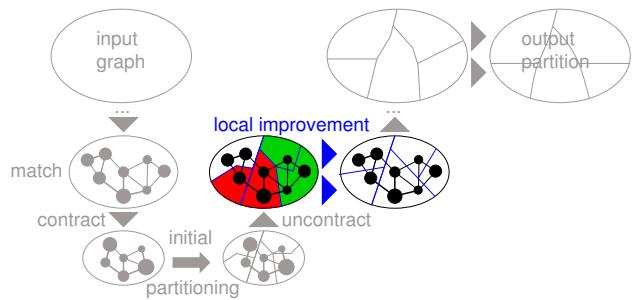
DiBaP [20] is a multi-level graph partitioning package where local improvement is based on diffusion which also yields partitions of very high quality.

MQI [18] is a flow-based method 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 $k = 2$.

The concept of *iterated multilevel algorithms* was introduced by [26,28]. 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 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 4.2 is called multi-try FM which is a more localized k -way local search. Roughly speaking, a k -way 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 pair wise 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 $P : V \rightarrow \{1, 2\}$ 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* $\partial B := \{u \in B \mid \exists(u, v) \in E : v \notin B\}$. From now on we call B *corridor* unless otherwise mentioned. 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 an 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 3).

First we need to find an 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 $(1 + \epsilon) \frac{n}{2} - w(\text{block 2})$, we stop the BFS. The second BFS is done analog for block 2.

In order to achieve the cut property C , the B -corridor induced subgraph G' gets additional s - t edges. Therefore s is connected to all left corridor border nodes $\partial_l B$ 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 $w(\text{block 2}) + (1 + \epsilon) \frac{n}{2} - w(\text{block 2}) = (1 + \epsilon) \frac{n}{2}$. 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

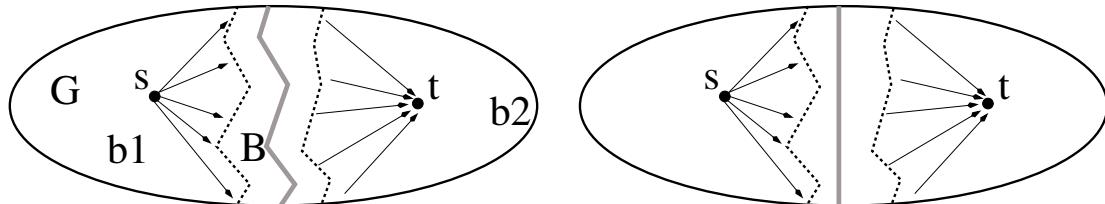


Fig. 3. The construction of a feasible flow problem which yields optimal cuts in G' and an improved cut within the balance constraint in G . On the left hand side the initial construction is shown and on the right hand side we see the improved partition.

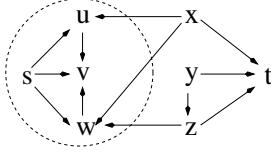


Fig. 4. A small graph where $C = \{s, u, v, w\}$ is a closed vertex set.

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 4.

Lemma 1 (Picard and Queyranne [23]). *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 $(C, V \setminus C)$. To enumerate all minimum cuts of a graph [23] 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 [10,1].

The minimum cut that is identified by the labeling procedure of Ford and Fulkerson [13] is the one with the smallest possible source set. We now define how the representation of the residual graph can be made more compact [23] 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 [3,14]. We make the representation more compact by contracting the 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 still each closed vertex set of the minimum cut representation induces a minimum cut.

As proposed in [23] 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

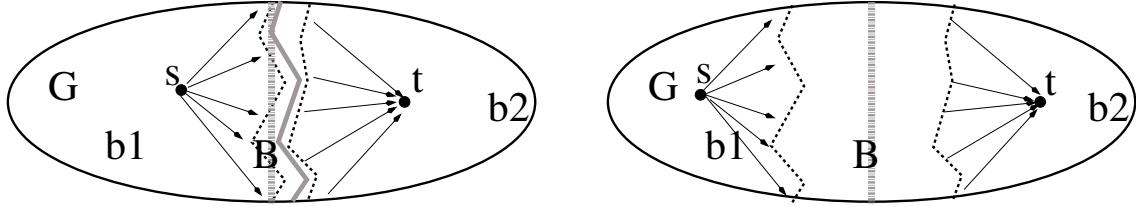


Fig. 5. On the left hand side it is not possible in the small corridor around the initial cut to find the dashed minimum cut which has optimal balance; however if we solve a larger flow problem on the right hand side and search for a cut with good balance we can find the dashed minimum cut with optimal balance but not every min cut is feasible for the underlying graph partitioning problem.

closed vertex set ($V \setminus \tilde{C}$). 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 that 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*.

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 the 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 not 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 starting 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 [21] which is described in Section 2.2.

4.3 Scheduling Quotient Graph Refinement.

There are two possibilities to schedule the execution of two way refinement algorithms on the quotient graph. Clearly the first simple idea is to traverse 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 somewhat 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 initialized 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 Figure 6.

```

1: procedure activeBlockScheduling()
2:   set all blocks active
3:   while there are active blocks
4:     A := <edge (u,v) in quotient graph : u is active or v is active>
5:     set all blocks inactive
6:     permute A randomly
7:     for each (u,v) in A do
8:       pairWiseImprovement(u,v)
9:       start multi-try fm search initialized with boundary of u and v
10:      if anything changed during local search then
11:        activate blocks that have changed during local or kway search

```

Fig. 6. Pseudocode for the active block scheduling algorithm. In our implementation the pair wise improvement step starts with a FM local search which is followed by a max-flow min-cut based improvement.

5 Global Search

Iterated Multilevel Algorithms were introduced by [26,28] (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 multi-level 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 random a factor such as random tie-breaking, so that each iteration is very likely to give different coarser graphs. Interestingly, in the multigrid community Full-Multigrid methods are generally preferable to simple *V*-cycles [2]. Therefore, we now introduce two novel global search strategies namely *W-cycles* and *F-cycles* for graph partitioning. A *W*-cycle works as follows: on *each* level we perform *two independent trials* using different random seeds for tie breaking during contraction, and local search. Again once the graph is partitioned, edges that are between blocks aren't matched. A *F*-cycle works similar to an *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 8. Again once the graph is partitioned for the first time, then this partition is used in the sense that edges between two blocks aren't 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 of the *W*- and *F*-cycle, the independent trials are only performed every *d*'th level. Our default value for *d* is 2.

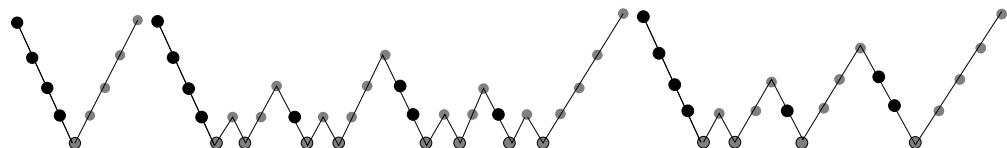


Fig. 7. From left to right: A single MGP V-cycle, a W-cycle and a F-cycle. The single V-cycle has 5 coarsening steps, the W-cycle has 15 coarsening steps and the F-cycle has 10 coarsening steps.

```

1: procedure W-Cycle(G)
2:    $G' = \text{coarsen}(G)$ 
3:   if  $G'$  small enough then
4:     initial partition  $G'$  if not partitioned
5:     apply partition of  $G'$  to  $G$ 
6:     perform refinement on  $G$ 
7:   else
8:     W-Cycle( $G'$ ) and apply partition to  $G$ 
9:     perform refinement on  $G$ 
10:     $G'' = \text{coarsen}(G)$ 
11:    W-Cycle( $G''$ ) and apply partition to  $G$ 
12:    perform refinement on  $G$ 

1: procedure F-Cycle(G)
2:    $G' = \text{coarsen}(G)$ 
3:   if  $G'$  small enough then
4:     initial partition  $G'$  if not partitioned
5:     apply partition of  $G'$  to  $G$ 
6:     perform refinement on  $G$ 
7:   else
8:     F-Cycle( $G'$ ) and apply partition to  $G$ 
9:     perform refinement on  $G$ 
10:    if no. rec. calls on cur. level < 2 then
11:       $G'' = \text{coarsen}(G)$ 
12:      F-Cycle( $G''$ ) and apply partition to  $G$ 
13:      perform refinement on  $G$ 

```

Fig. 8. Pseudocode for the different cycle variants.

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. and DiBaP 2.0.0. The flow problems are solved using Andrew Goldbergs Network Optimization Library HIPR [4] which is integrated into our code.

System. We have run our code on a cluster with 200 nodes each equipped with two Quad-core Intel Xeon processors (X5355) which run at a clock speed of 2.667 GHz, have 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 Table 6. These are the same instances as used for the evaluation of KaPPa [17]. 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 [29]. Graphs *bel*, *nld*, *deu* and *eur* are undirected versions of the road networks of Belgium, the Netherlands, Germany, and Western Europe respectively, used in [6]. Instances *af_shell9* and *af_shell10* come from the Florida Sparse Matrix Collection [5]. *coAuthorsDBLP*, *coPapersDBLP*, *citationCiteseer*, *coAuthorsCiteseer* and *cnr2000* are examples of social networks taken from [15]. For the number of partitions k we choose the values used in [29]: 2, 4, 8, 16, 32, 64. Our default value for the allowed imbalance is 3 % since this is one of the values used in [29] and the default value in Metis.

Configuring the Algorithm. We currently only define a *Strong* version (the final paper will also contain a fast version). The *coarsening phase* uses GPA as matching algorithm. InnerOuter is employed as rating on the first level and expansion^{*2} on the other levels. In the *initial partitioning phase* we perform 100 / $\log k$ initial partitioning attempts using Scotch as an initial partitioner. The *refinement phase* first employs k -way

Medium sized instances		
graph	<i>n</i>	<i>m</i>
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
brack2	62 631	733 118
finan512	74 752	522 240
bel	463 514	1 183 764
nld	893 041	2 279 080
af_shell9	504 855	17 084 020

Large instances		
graph	<i>n</i>	<i>m</i>
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
Social networks		
coAuthorsDBLP	299 067	1 955 352
citationCiteseer	434 102	32 073 440
coAuthorsCiteseer	227 320	1 628 268
cnr2000	325 557	3 216 152
coPapersDBLP	540 486	30 491 458

Table 1. Basic properties of the graphs from our benchmark set. The large instances are split into five groups: geometric graphs, FEM graphs, street networks, sparse matrices, and social networks. Within their groups, the graphs are sorted by size.

refinement (since it converges very fast) which uses the adaptive search strategy from [21] with $\alpha = 10$. We continue by performing quotient-graph style refinements. Here we use the active block scheduling algorithm which is combined with the multi-try local search (again $\alpha = 10$). A refinement of a pair of blocks works as follows: It is started by a pair wise FM search which is followed by the max-flow min-cut algorithms (including the most balancing heuristics). We set the upper bound factor for the flow regions sizes to $\alpha' = 8$. As *global search* we use at most two F-cycles. Initial Partitioning is only performed if previous partitioning information is *not* available. Otherwise, we use the given input partition.

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 figure.¹

Effectiveness Tests: Here each algorithm configuration has the same time for computing a partition. Therefore, for each graph and k we compute a timing fingerprint of all algorithm variants and remember the largest execution time t . Now each algorithm gets time $3t$ to compute a good partition, i.e. taking the best partition out of repeated runs. If a variant can perform a next run depends on the remaining time, i.e. we flip a coin with corresponding probabilities such that the expected time over multiple runs is $3t$.

¹ Because we have multiple repetitions for each instance, 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.

6.1 Algorithmic Insights about Flows

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 (-Flow, -MB, +FM). This configuration uses the Global Paths Algorithm as matching algorithm, Scotch as initial partitioner and the active block scheduling algorithm equipped with the two way FM algorithm described in Section 2.2. Edge rating functions are used as described in Section 2.2. Note that during this test our main focus is the evaluation of flows and therefore we don't use k -way refinement and multi-try fm search. For comparisons this configuration is then extended by a specific algorithm, e.g. (+Flow, -MB, +FM) with a flow upper bound factor $\alpha' = 8$ means that the underlying algorithm configuration uses Flow, FM but not the most balancing heuristics.

Variant	(+Flow, -MB, -FM)				(+Flow, +MB, -FM)				(+Flow, -MB, +FM)				(+Flow, +MB, +FM)			
	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 2. The final score of different algorithm configurations compared against the basic FM configuration.

In Table 2 we see that using Flow on its own, i.e. no FM-algorithm is used at all, yields cuts and run times which are worse than the basic FM configuration. However, the results get better if we enable the most balanced minimum cut heuristic (MB), e.g. for $\alpha' = 16$ we get cuts that are lower than the cuts produced by the basic FM heuristic. In some cases, flows and flows with MB aren't able to produce results that are comparable to the basic 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 2 we can see that the combination of flows with local search produces lower cuts and the results get even better if we enable MB, albeit increased run time. Table 3 shows that these combinations are also more effective than the repeated execution of the basic two-way FM configuration.

Effectiveness (+Flow, +MB, -FM)	Avg.	Best.	Bal.	Effectiveness (+Flow, -MB, +FM)	Avg.	Best.	Bal.	Effectiveness (+Flow, +MB, +FM)	Avg.	Best.	Bal.
$\alpha' = 1$	3 389	3 351	1,016	$\alpha' = 1$	2 786	2 759	1,024	$\alpha' = 1$	2 781	2 754	1,023
2	3 088	3 049	1,017	2	2 748	2 724	1,024	2	2 735	2 711	1,021
4	2 922	2 892	1,022	4	2 721	2 698	1,025	4	2 702	2 682	1,022
8	2 865	2 841	1,023	8	2 718	2 690	1,025	8	2 699	2 675	1,023
16	2 870	2 839	1,023	16	2 730	2 697	1,025	16	2 711	2 682	1,022
(-Flow, -MB, +FM)	2 833	2 803	1,025	(-Flow, -MB, +FM)	2 831	2 801	1,025	(-Flow, -MB, +FM)	2 827	2 799	1,025

Table 3. Each table is the result of an effectiveness test for six different algorithm configurations.

6.2 Algorithmic Insights about Global Search Strategies.

In Table 4 we compared the different global search strategies against a normal V-cycle. The only parameter varied during this test is the global search strategy. Clearly, for each global search strategy cuts decreases albeit increased run time. However, the effectiveness results also indicate that each configuration is superior to repeated executions of one single V-cycle. On the one hand this is due to the time saved using the active block strategy which converges very quickly in later cycles. On the other we save time for initial partitioning which is only performed the first time the algorithm arrives in the initial partitioning phase.

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

6.3 Knockout / Removal Tests

We now turn into two kinds of experiments: Component removal tests and Knockout tests. 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.

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.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t	Avg.	Best.	t		
2	550	547	550	548	550	548	549	548	549	552	549	581	573	573	573	581	573	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	1 299	1 299	1 316	1 299	1 299	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	2 387	2 408	2 387	2 408	2 387	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	3 996	4 029	3 996	4 029	3 996	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	6 369	6 403	6 369	6 403	6 369	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	10 085	10 139	10 085	10 139	10 085	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	2 799	2 775	2 799	2 775	2 799	2 775	2 799	2 775

Table 5. 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.

In Table 5 and in Table 6 details results for normal tests and for effectivity tests are shown for all k . Notice that in order to achieve high quality partitions we don't need to perform classical k -way refinement. Table 6 shows that changes in solution quality are really small. However, the usage of k -way refinement speeds up overall runtime of the algorithm; hence we included it into our Strong configuration.

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.	Avg.	Best.	Avg.	Avg.	Best.	Avg.	Avg.	Best.	
2	550	547	550	548	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	1 251	1 241	1 266	1 252			
8	2 263	2 242	2 270	2 249	2 276	2 262	2 270	2 246	2 270	2 246	2 281	2 259			
16	3 771	3 742	3 767	3 741	3 810	3 781	3 773	3 747	3 773	3 747	3 797	3 767			
32	6 000	5 943	6 002	5 950	6 090	6 055	6 006	5 955	6 006	5 955	6 028	5 977			
64	9 523	9 463	9 502	9 437	9 681	9 636	9 525	9 470	9 525	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 6. 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.

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 over tuning our algorithm parameters to the instances used for calibration. We compare ourselves with KaSPar Strong, KaPPa Strong, DiBaP Strong, Scotch and Metis. Table 7 summarizes the results. We excluded the European and German road network as well as the Random Geometric Graph since

k	KaFFPa Strong			KaSPar Strong			KaPPa Strong			DiBaP			Scotch			Metis		
	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t	Best.	Avg.	t
2	3 988	4 001	22,68	4 013	4 047	24,94	4 089	4 180	11,63	4 285	5 155	2,25	4 331	-	0,74	4 581	4 809	0,32
4	10 467	10 559	50,18	10 548	10 610	32,09	10 940	11 168	19,76	11 133	11 341	2,79	11 648	-	1,62	11 815	12 399	0,33
8	19 288	19 553	76,39	19 332	19 507	44,11	20 255	20 609	25,46	20 980	21 451	4,31	21 806	-	2,60	22 398	23 006	0,34
16	31 474	31 953	111,49	31 676	32 000	65,43	32 821	33 219	26,66	33 859	34 389	7,19	36 052	-	3,72	36 164	37 087	0,36
32	48 195	48 506	145,04	48 770	49 254	94,42	50 085	50 573	21,84	51 088	51 773	12,14	54 340	-	4,99	54 896	55 746	0,40
64	69 936	70 363	199,84	71 506	72 024	126,59	72 837	73 316	16,44	74 144	74 676	21,17	78 207	-	6,37	78 498	79 318	0,44
	20 986	21 172	80,93	21 185	21 364	54,97	21 839	22 163	19,56	22 460	23 461	6,07	23 503	-	2,68	23 954	24 661	0,36

Table 7. Results for our large benchmark suite.

DiBaP can't handle singletons and KaPPa runs out of memory for $k = 2$ for the European road network. As

recommended by Henning Meyerhenke DiBaP was with 3 bubble repetitions, 14 FOS/L consolidations and 10 FOS/L iterations. Detailed per instance results can be found in the Appendix Table 11.

6.5 The Walshaw Benchmark

We now apply KaFFPa to Walshaw's benchmark archive [29] 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 aren't made for this case.

We ran KaFFPa Strong with a time limit of two hours and report the best result obtained in the appendix. KaFFPa computed 317 partitions which are better than previous best partitions reported there: 99 for 1%, 108 for 3% and 110 for 5%. Moreover, it reproduced equally sized cuts in 118 additional cases.

7 Conclusions and Future Work

KaFFPa is an approach to graph partitioning which currently computes the best known partitions for many large 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 the multigrid community.

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 [17]. We also want to combine KaFFPa with the n -level idea from KaSPar [21]. 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 planning 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 and Robert Görke for concrete advises regarding balanced min cuts.

References

1. P. Bonsma. Most balanced minimum cuts. *Discrete Applied Mathematics*, 158(4):261–276, 2010.
2. W.L. Briggs and S.F. McCormick. *A multigrid tutorial*. Society for Industrial Mathematics, 2000.
3. J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.
4. B.V. Cherkassky and A.V. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997.
5. T. Davis. The University of Florida Sparse Matrix Collection, <http://www.cise.ufl.edu/research/sparse/matrices>, 2008.
6. 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.
7. D. Drake and S. Hougaard. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85:211–213, 2003.

8. R. Preis et al. PARTY partitioning library. <http://wwwcs.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/part.html>.
9. R. P. Fedorenko. A relaxation method for solving elliptic difference equations. *USSR Comput. Math. and Math. Phys.*, 5(1):1092–1096, 1961.
10. 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.
11. 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.
12. P.O. Fjallstrom. Algorithms for graph partitioning: A survey. *Linkoping Electronic Articles in Computer and Information Science*, 3(10), 1998.
13. L. Ford and D. Fulkerson. Flows in networks. 1962.
14. H.N. Gabow. Path-Based Depth-First Search for Strong and Biconnected Components. *Information Processing Letters*, 2000.
15. R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *10th Workshop on Algorithm Engineering and Experimentation*, pages 90–108, San Francisco, 2008. SIAM.
16. B. Hendrickson. Chaco: Software for partitioning graphs. <http://www.sandia.gov/~bahendr/chaco.html>.
17. M. Holtgrieve, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. *24th IEEE International Parallel and Distributed Processing Symposium*, 2010.
18. 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.
19. 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.
20. 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.
21. V. Osipov and P. Sanders. n-Level Graph Partitioning. *18th European Symposium on Algorithms (see also arxiv preprint arXiv:1004.4024)*, 2010.
22. F. Pellegrini. Scotch home page. <http://www.labri.fr/pelegrin/scotch>.
23. J.C. Picard and M. Queyranne. On the structure of all minimum cuts in a network and applications. *Combinatorial Optimization II*, pages 8–16, 1980.
24. 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.
25. 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.
26. 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.
27. C. Walshaw. The Graph Partitioning Archive, <http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/>, 2008.
28. C. Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1):325–372, 2004.
29. C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.
30. 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).

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 8. 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 9. 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.

Graph/k	2		4		8		16		32		64	
3elt	87	87	197	197	330	330	558	560	952	950	1528	1539
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 10. Computing partitions from scratch $\epsilon = 5\%$. In each k -column the results computed by KaFFPa are on the left and the current Walsh cuts are presented on the right side.

graph	k	KaFFPa Strong			KaSPar Strong			KaPPa Strong			DiBaP			Scotch			Metis		
		Best	Avg.	t	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t	Best	Avg.	t
fe_tooth	2	3789	3829	5.43	3844	3987	5.86	3951	4336	3.75	4390	4785.2	0.98	4259	4259	0.38	4372	4529	0.08
fe_tooth	4	6812	6946	12.62	6937	6999	8.54	7012	7189	5.22	7492	8081.8	1.10	8304	8304	0.72	7805	8280	0.08
fe_tooth	8	11595	11667	18.22	11482	11564	13.43	12272	12721	6.83	12186	12532.4	1.78	12999	12999	1.09	13334	13768	0.08
fe_tooth	16	17907	18056	27.53	17744	17966	21.24	18302	18570	7.18	19389	19615.8	2.86	20816	20816	1.59	20035	20386	0.09
fe_tooth	32	25585	25738	41.42	25888	26248	35.12	26397	26617	5.28	26518	27073.4	5.05	28430	28430	2.13	28547	29052	0.10
fe_tooth	64	35497	35597	57.23	36259	36469	49.65	36862	37002	4.71	37271	37458	8.78	38401	38401	2.69	39233	39381	0.12
598a	2	2367	2372	7.73	2371	2384	6.50	2387	2393	5.64	2414	2435.6	1.90	2417	2417	0.39	2444	2513	0.14
598a	4	7896	7993	13.29	7897	7921	11.15	8235	8291	10.24	8200	8200	2.39	8246	8246	0.95	8466	8729	0.15
598a	8	15830	16182	25.60	15929	15984	22.31	16502	16641	12.21	16585	16663	3.59	17490	17490	1.63	17170	17533	0.16
598a	16	26211	26729	41.81	26046	26270	38.39	26467	26825	17.74	26693	27131	6.14	29804	29804	2.37	27857	28854	0.17
598a	32	39863	39976	68.82	39625	40019	60.60	40946	41190	18.16	40908	41456.6	10.96	44756	44756	3.21	43256	44213	0.19
598a	64	57325	57860	107.20	58362	58945	87.52	59148	59387	14.15	58978	59371.6	18.50	64561	64561	4.11	61888	62703	0.22
fe_ocean	2	311	311	5.27	317	317	5.55	314	317	3.21	348	10676	0.62	402	402	0.18	540	579	0.11
fe_ocean	4	1789	1789	9.36	1801	1810	9.40	1756	1822	6.30	1994	1994	0.69	2000	2000	0.44	2102	2140	0.11
fe_ocean	8	4012	4087	13.58	4044	4097	14.33	4104	4252	6.33	5208	5305.2	1.24	4956	4956	0.81	5256	5472	0.12
fe_ocean	16	7966	8087	21.14	7992	8145	22.41	8188	8350	5.62	9356	9501.8	1.97	9351	9351	1.27	10115	10377	0.13
fe_ocean	32	12660	12863	31.73	13320	13518	36.53	13593	13815	4.34	15893	16230.2	3.09	15089	15089	1.83	16565	16877	0.15
fe_ocean	64	20606	20739	66.39	21326	21739	62.46	21636	21859	3.68	24692	24894.8	6.02	23246	23246	2.49	24198	24531	0.17
144	2	6451	6482	16.12	6455	6507	12.81	6559	6623	7.45	7146	7146	2.38	6695	6695	0.66	6804	6972	0.20
144	4	15485	15832	34.62	15312	15471	24.73	16870	16963	13.33	16169	16550.4	3.17	16899	16899	1.44	17144	17487	0.21
144	8	25282	25262	53.65	25130	25409	38.13	26300	26457	20.11	26121	26871.2	4.54	28172	28172	2.24	28006	28194	0.22
144	16	38483	38669	85.52	37872	38404	69.35	39010	39319	26.04	39618	40066.6	7.77	43712	43712	3.12	42861	43041	0.24
144	32	56672	56827	121.75	57082	57492	106.40	58331	58631	24.60	57683	58592.2	13.03	63224	63224	4.14	61716	62481	0.26
144	64	78828	79477	147.98	80313	80770	144.77	82286	82452	19.11	81997	82216.6	23.22	88246	88246	5.25	86534	87208	0.30
wave	2	8665	8681	14.23	8661	8720	16.19	8832	9132	8.24	8994	10744	2.02	9337	9337	0.83	9169	9345	0.19
wave	4	16804	16908	38.36	16806	16920	29.56	17008	17250	14.51	17382	17608.8	2.52	19995	19995	1.72	19299	21906	0.20
wave	8	28882	29339	62.99	28681	28817	46.61	30690	31419	20.63	29893	32246.4	3.74	33357	33357	2.61	33223	33639	0.21
wave	16	42292	43538	97.53	42918	43208	69.35	43010	43219	26.04	45227	45596.8	6.32	48903	48903	3.53	48404	49000	0.22
wave	32	62566	62647	124.43	63025	63159	112.19	63981	64390	14.94	63594	64464	10.50	70581	70581	4.68	68062	68604	0.25
wave	64	84970	85649	195.61	87243	87554	150.37	88376	88964	12.51	87741	88487.2	18.61	96759	96759	5.90	92148	94083	0.29
m14b	2	3823	3823	19.82	3828	3846	20.03	3862	3954	11.16	3898	3941.6	3.53	3872	3872	0.70	4036	4155	0.31
m14b	4	12953	13031	38.87	13015	13079	26.51	13543	13810	18.77	13494	13519.2	4.72	13484	13484	1.71	13932	14560	0.33
m14b	8	26006	26179	65.15	25573	25756	45.33	27330	27393	24.97	26743	26916.2	7.10	27839	27839	2.86	28138	28507	0.34
m14b	16	43176	43759	91.08	42212	42458	83.25	45352	45762	28.11	44666	45515.6	12.75	50778	50778	4.25	48314	49269	0.36
m14b	32	67417	67512	142.37	66314	66991	133.88	68107	69075	29.94	67888	68957	22.30	75453	75453	5.72	72746	74135	0.40
m14b	64	98222	98536	189.96	99207	100014	198.23	101053	101455	25.26	99994	100653	37.37	109404	109404	7.38	107384	108141	0.44
auto	2	9725	9775	74.25	9740	9768	68.39	9910	10045	30.09	10094	11494.2	6.94	10666	10666	1.61	10781	12147	0.83
auto	4	25841	25891	151.14	25988	26062	75.60	28218	29481	64.01	26523	27958	9.92	29046	29046	3.52	27469	30318	0.86
auto	8	44847	45299	257.71	45099	45232	97.60	46272	46652	85.89	48326	48346.4	14.24	49999	49999	5.42	49691	52422	0.87
auto	16	75792	77429	317.81	76287	76715	153.46	78713	79769	87.41	80198	81742.6	24.60	84462	84462	7.84	85562	89139	0.91
auto	32	121016	121687	366.47	121269	121862	246.50	12460	125500	71.77	12443	125043	40.76	133403	133403	10.58	133026	134086	0.99
auto	64	173155	173624	490.74	174612	174914	352.09	177038	177595	62.64	175091	175758	66.23	193170	193170	13.68	188555	189699	1.08
delaunay_n20	2	1680	1687	197.11	1711	1731	196.33	1858	1882	35.43	1994	22654	2.91	1874	1874	1.18	2054	2194	1.11
delaunay_n20	4	3368	3380	124.29	3418	3439	130.67	3674	3780	64.08	3804	3804	3.05	3723	3723	2.35	4046	4094	1.15
delaunay_n20	8	6247	6283	154.95	6278	6317	104.37	6670	6854	70.07	6923	71028	5.01	7180	7180	3.58	7705	8029	1.13
delaunay_n20	16	10012	10056	210.39	10183	10218	84.33	10816	11008	67.92	11174	11382	8.01	11266	11266	4.77	11854	12440	1.14
delaunay_n20	32	15744	15804	220.40	15905	16026	101.69	16813	17086	42.67	17343	17408.8	13.5	17784	17784	6.04	18816	19304	1.18
delaunay_n20	64	23472	23551	237.76	23935	23962	97.09	24799	25179	22.04	25884	26148.6	23.9	26163	26163	7.34	28318	28543	1.21
rgg_n_2_20_s0	2	2088	2119	94.68	2162	2201	198.61	2377	2498	33.24	26225	26225	3.74	26825	26825	3.64	27625	28955	1.57
rgg_n_2_20_s0	4	4184	4241	167.88	4323	4389	130.00	4867	5058	38.50	56075	56075	4.92	5737	5737	2.82	5786	6174	1.56
rgg_n_2_20_s0	8	7684	7729	192.45	7745	7915	103.66	8995	9391	46.06	11251	11251	4.48	11365	11771	1.54	17157	17157	6.13
rgg_n_2_20_s0	16	12504	12673	205.29	12596	12792	86.19	14953	15199	35.86	168450	171940	11.98	17125	17125	16.45	184350	188765	3.06
rgg_n_2_20_s0	32	20708	20400	207.80	20403	20478	100.03	23430	23917	26.04	28078	28078	7.96	27765	28495	1.58	38815	38815	9.83
rgg_n_2_20_s0	64	30518	30893	230.28	30860	31066	97.83	34778	35354	11.62	41066	42465	1.58						
af_shell10	2	26225	26225	367.08	26225	26225	317.11	26225	26225	78.65									