Think Locally, Act Globally: Highly Balanced Graph Partitioning*

Peter Sanders and Christian Schulz

Karlsruhe Institute of Technology, Karlsruhe, Germany
{sanders,christian.schulz}@kit.edu

Abstract. We present a novel local improvement scheme for graph partitions that allows to enforce strict balance constraints. Using negative cycle detection algorithms this scheme combines local searches that individually violate the balance constraint into a more global feasible improvement. We combine this technique with an algorithm to balance unbalanced solutions and integrate it into a parallel multi-level evolutionary algorithm, KaFFPaE, to tackle the problem. Overall, we obtain a system that is fast on the one hand and on the other hand is able to improve or reproduce many of the best known *perfectly* balanced partitioning results reported in the Walshaw benchmark.

1 Introduction

In computer science, engineering, and related fields *graph partitioning* is a common technique. For example, in parallel computing good partitionings of unstructured graphs are very valuable. In this area, graph partitioning is mostly used to partition the underlying graph model of computation and communication. Generally speaking, nodes in this graph represent computation units and edges denote communication. This graph needs to be partitioned such that there are few edges between the blocks (pieces). In particular, if we want to use k processors we want to partition the graph into k blocks of about equal size. Here we focus on the case when the bounds on the size are very strict, including the case of *perfect balance* when the maximal block size has to equal the average block size.

The problem is NP-hard and hard to approximate on general graphs so that mostly heuristics are used in practice. A successful heuristic for partitioning large graphs is the *multi-level* approach. Here, the graph is recursively *contracted* to achieve a smaller graph with the same basic structure. After applying an *initial partitioning* algorithm to the smallest graph in the hierarchy, the contraction is undone and, at each level, a *local refinement* method is used to improve the partitioning induced by the coarser level.

During the last years we started to put all aspects of the multi-level graph partitioning (MGP) scheme on trial since we had the impression that certain aspects of the method are not well understood. Our main focus is partition quality rather than partitioning speed. In our sequential MGP framework KaFFPa (Karlsruhe Fast Flow Partitioner) [12], we presented novel local search as well as global search algorithms.

* This paper is a short version of the TR [14].

V. Bonifaci et al. (Eds.): SEA 2013, LNCS 7933, pp. 164–175, 2013.

In the Walshaw benchmark [15], KaFFPa was beaten mostly for small graphs that combine multi-level partitioning with an evolutionary algorithm. We therefore developed an improved evolutionary algorithm, KaFFPaE (KaFFPa Evolutionary) [13], that also employs coarse grained parallelism. Both of these algorithms are able to compute partitions of very high quality in a reasonable amount of time when some imbalance $\epsilon > 0$ is allowed. However, they are not yet very good for small values of ϵ, in particular for the perfectly balanced case $\epsilon = 0$.

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

The paper is organized as follows. We begin in Section 2 by introducing basic concepts. After presenting some related work in Section 3 we describe novel improvement and balancing algorithms in Section 4. Here we start by explaining the very basic idea that allows us to find combinations of simple node movements. We then explain directed local searches and extend the basic idea to a complex model containing more node movements. This is followed by a description on how these techniques are integrated into KaFFPaE. A summary of extensive experiments done to evaluate the performance of our algorithms is presented in Section 5.

2 Preliminaries

Consider an undirected graph $G = (V, E, \omega)$ with edge weights $\omega : E \to \mathbb{R}_{>0}$, $n = |V|$, and $m = |E|$. We extend ω to sets, i.e., $\omega(E') := \sum_{e \in E'} \omega(e)$. $\Gamma(v) := \{u : \{v, u\} \in E\}$ 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 $i \neq j$. A *balancing constraint* demands that $\forall i \in \{1..k\} : |V_i| \leq L_{\max} := (1 + \epsilon)\lceil |V|/k \rceil$. In the perfectly balanced case the imbalance parameter ϵ is set to zero. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. A block V_i is called underloaded if $|V_i| < L_{\max}$ and overloaded if $|V_i| > L_{\max}$. A node $v \in V_i$ that has a neighbor $w \in V_j, i \neq j$, is a boundary node. An abstract view of the partitioned graph is the so called *quotient graph*, where nodes represent blocks and edges are induced by connectivity between blocks. Given a partition, the gain of a node v in block A with respect to a block B is defined as $g_{(A,B)} = \omega(\{(v, w) \mid w \in \Gamma(v) \cap B\}) - \omega(\{(v, w) \mid w \in \Gamma(v) \cap A)$, i.e. the reduction in the cut when v is moved from block A to block B. By default, our initial inputs will have unit node weights. However, the proposed algorithms can be easily extended to handle weighted nodes.

3 Related Work

There has been a huge amount of research on graph partitioning so that we refer the reader to [3]. Well known software packages based on this multi-level approach include, Jostle [17], Metis [9], and Scotch [11]. However, for various reasons they are not able guarantee strict balance constraints. KaFFPaE [13] is a distributed parallel evolutionary algorithm that uses our multi-level graph partitioning framework KaFFPa [12] to create individuals and modifies the coarsening phase to provide new effective combine operations. It currently holds the best results for many graphs in Walshaw's Benchmark Archive [15] when some imbalance is allowed. Benlic et al. [2] provided multi-level memetic algorithms for perfectly balanced graph partitioning. Their approach is able to compute many entries in Walshaw's Benchmark Archive [15] for the case $\epsilon = 0$. However, they are not able to guarantee that the computed partition is perfectly balanced especially for larger values of k.

4 Globalized Local Search by Negative Cycle Detection

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

We begin by describing the very basic algorithm and go on by presenting an advanced model which enables us to combine complex local searches. This is followed by a description on how local search and balancing algorithms are put together. At the end of this section we show how we integrate these algorithms into our evolutionary framework KaFFPaE.

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

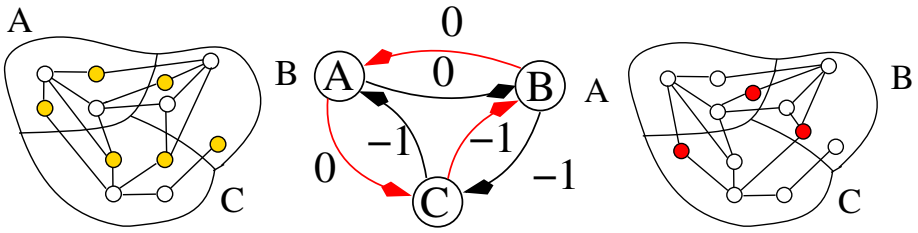

Fig. 1. Left: example graph partitioned into three parts (A, B and C). Possible candidates are highlighted. Middle: corresponding model and one negative cycle is highlighted. Right: updated partition after associated node movements of cycle are performed. Moved nodes are highlighted.

directed and weighted version of the quotient graph and that the selected nodes form an independent set.

Note that each cycle in this model defines a set of node movements and furthermore when the associated nodes of a cycle are moved, then each block contains the same number of nodes as before. Also the weight of a cycle in the model is equal to the reduction in the cut when the associated node movements are performed. However, the most important aspect is that a *negative cycle* in the model corresponds to a set of node movements that will decrease the overall cut and maintain the balance of the partition. To detect a negative cycle in this model we introduce a node s and connect it to all nodes in $\mathcal{Q}$. The weight of the inserted edges is set to zero. We can apply a standard shortest path algorithm [4] that can handle negative edge weights to detect a negative cycle. If the model contains a negative cycle we can perform a set of node movements that will not alter the balance of the blocks since each block obtains and emits a node.

We can find additional useful augmentations by connecting blocks which can take at least one node without becoming overloaded to s by a zero weight edge. Now, negative cycles containing s change some block weights but will not violate any additional balance constraints. Indeed, when the node following s is overloaded initially, this overload will be reduced.

If there is no negative cycle in the model, we apply a diversification strategy based on cycles of weight zero. This strategy is explained in the TR [14]. Moreover, we apply a balancing algorithm which is explained in the following sections. An interesting observation is that the algorithm can be seen as an extension of the classical FM algorithm [6] which swaps nodes between two adjacent blocks (two at a time) which is basically a negative cycle of length two in our model if the gain of the two node movements is positive.

Advanced Model. We now integrate advanced local search algorithms. Each edge in the advanced model stands for a *set* of node movements found by a local search. Hence, a negative cycle corresponds to a combination of local searches with positive overall gain that maintain balance or that can improve balance. Before we build the advanced model we perform *directed local search* on each pair of blocks that share a non-empty boundary, i.e. each pair of blocks that is adjacent in the quotient graph. A local search on a directed pair of blocks (A, B) is only allowed to move nodes from block A to block B. The order in which the directed local search between a directed pair of blocks

is performed is random. That means we pick a random directed adjacent pair of blocks on which local search has not been performed yet and perform local search as described below. This is done until local search was done between all directed adjacent pairs of blocks once.

Directed Local Search. We now explain how we perform a directed local search between a pair (A, B) of blocks. A directed local search between two blocks A and B is very localized akin to the multi-try method used in KaFFPa [12]. However, a directed local search between A and B is restricted to move nodes from block A to block B. It is similar to the FM-algorithm: We start with a *single* random eligible boundary node of block A having maximum gain $g_{\max}(A, B)$ and put this node into a priority queue. The priority queue contains nodes of the block A that are valid to move. The priority is based on the *gain*, i.e. the decrease in edge cut when the node is moved from block A to block B. We always move the node that has the highest priority to block B. After a node is moved its eligible neighbors that are in block A are inserted into the priority queue. We perform at most τ steps per directed local search, where τ is a parameter. Note that during a directed local search we only move nodes that are not incident to a node moved during a previous directed local search. This restriction is necessary to keep the model described below accurate. Thus we *mark all nodes* touched during a directed local search *after* it was performed which also implies that each node is moved at most once. In addition all moved nodes are *moved back* to their origin, since these movements would make the partition imbalanced. We stress that all nodes incident to nodes that have been moved during a directed local search are not *eligible* for any later local search during the construction since this would make the gain values computed imprecise.

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

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

Generally speaking, the *advanced model* consists of τ layers. Essentially each layer is a copy of the quotient graph. An edge starting and ending in layer d of this model corresponds to the movement of exactly d nodes. The weight of an edge $e = (A, B)$ in layer d of the model is set to the negative value of the d'th entry in g_e. In other words it encodes the negative value of the gain, when the first d nodes in S_e are moved from block A to block B. Hence, a negative cycle whose nodes are all in layer d will move exactly d nodes between each of the respective block pairs contained in the cycle and

results in a overall decrease in the edge cut. We add additional edges to the model such that it contains *more possibilities* in presence of underloaded blocks. To be more precise, in these cases we want to get rid of the restriction that each block sends and emits the same number of nodes. To do so we insert *forward* edges between all consecutive layers, i.e. block k in layer d is connected by an edge of weight zero to block k in layer $d + 1$. These edges are not associated with node movements. Furthermore, we add *backward* edges as follows: for an edge (A, B) in layer d we add an edge with the same weight between block A in layer d and block B in layer $d - \ell$ if block B can take ℓ nodes without becoming overloaded. The newly inserted edge is associated with the same node movements as the initial edge (A, B) within layer d. This way we encode movements in the model where a block can emit more nodes then it gets and vice versa without violating the balance constraint. Additionally we connect each node in layer d back to s if the associated block can take at least d nodes without becoming overloaded. Again this means that the model might contain cycles through s which stand for paths in the quotient graph being associated with node movements that decrease the overall cut. Moreover, these moves never increase the imbalance of the input partition. An example for the advanced model can be found in the TR [14]. We can apply the same zero weight cycle diversification as in the basic model. The advanced model can contain *conflicting* cycles that cannot be used. Due to space constraints, we explain when conflicts occur and how we handle them in the TR [14].

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

Balancing. As we will see, to create ϵ-balanced partitions we start our algorithm with partitions where larger imbalance is allowed. Hence, to satisfy the balance constraint, we have to think about balancing strategies. A balancing step will only be applied if the model does not contain a negative cycle (see next section for more details). Hence, we can modify the advanced model such that we can find a set of node movements that will decrease the total number of overloaded nodes by at least one and minimizes the increase in the number of edges cut. Specifically, we introduce a second node t. Now instead of connecting s to all vertices, we connect it only to nodes representing overloaded blocks, i.e. $|V_i| > \lceil |V|/k \rceil$. Additionally, we connect a node in layer ℓ to t if the associated block can take at least ℓ nodes without becoming overloaded. Since the underlying model does not contain negative cycles, we can apply a shortest path algorithm to find a shortest path from s to t. We use a variant of the algorithm of Bellman and Ford since edge weights might still be negative (for more details see Section 5).

It is now easy to see that a shortest path in this model yields a set of node movements with the smallest increase in the number of cut edges and that the total number of overloaded nodes decreases by at least one. If τ is set to one we call this algorithm basic balancing otherwise advanced balancing.

However, we have to make sure that there is at least one s-t path in the model. Let us assume for now that the graph is connected. If the graph is connected then the directed version of the quotient graph is strongly connected. Hence an s-t path exists in the model if we are able to perform local search between *all* pairs of blocks that share a non-empty boundary. Because a directed local search can only start from an eligible node we might not be able to perform directed local search between all adjacent pairs of blocks, e.g. if there is no eligible node between a pair of blocks left. We try to *ensure* that there is at least one s-t path in the model by doing the following. Roughly speaking we try to integrate a s-t path in the model by changing the order in which directed local searches are performed. First we perform a breadth first search (BFS) in the quotient graph which is initialized with all nodes that correspond to overloaded blocks in a random order. We then pick a random node in the quotient graph that corresponds to a block A that can take nodes without becoming overloaded. Using the BFS-forest we find a path $\mathcal{P} = B \to \cdots \to A$ from an overloaded block B to A. We now first perform directed local search on all consecutive pairs of blocks in $\mathcal{P}$. Here we use $\tau = 1$ for the number of node movements to minimize the number of non-eligible nodes. If this was successful, i.e. we have been able to move one node between all directed pairs of blocks in that path, we perform directed local searches as before on *all* pairs of blocks that share a non-empty boundary. Otherwise we undo the searches done (every node is eligible again) and start with the next random block that can take a node without becoming overloaded. In some rare cases the algorithm fails to find such a path, i.e. each time we look at a path we have one directed pair of blocks where no eligible node is left. An example is shown in Figure 2. In this case we apply a fallback balance routine that guarantees to reduce the total number of overloaded nodes by one if the input graph is connected. Given the BFS-forest of the quotient graph from above, we look at all paths in it from an overloaded block to a block that can take a node without becoming overloaded. At this point there are at most $\mathcal{O}(k)$ such paths in our BFS-forest. Specifically for a path $\mathcal{P} = Z \to Y \to X \to \cdots \to A$ we select a node having maximum gain $g_{Z,Y}$ in Z and move it to Y. We then look at Y and do the same with respect to X and so on until we move a node to block A. Note that this time we can ensure to find nodes because after a node has been moved it is not blocked for later movements. After the operations have been performed they are undone and we continue with the next path. In the end we use the movements of the path that resulted in the smallest number of edges cut. If

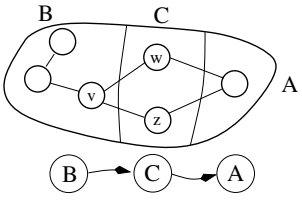

Fig. 2. Top: a graph partitioned into three parts. Bottom: BFS-tree in the quotient graph starting in overloaded block B. This path cannot be integrated into the model. After a directed local search on pair (B, C), v is marked and there is no eligible node left for the local search on pair (C, A). A similar argument holds if local search is done on the pair (C, A) first.

the graph contains more than one connected component then the algorithms described above may not work. If this is the case we use a fall back algorithm which is described in the TR [14].

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

Integration into KaFFPaE. We now describe how we integrate our new algorithms into our distributed evolutionary algorithm KaFFPaE [13]. An evolutionary algorithm starts with a population of individuals (in our case partitions of the graph) and evolves the population into different populations over several rounds. In each round, the evolutionary algorithm uses a selection rule based on the fitness of the individuals (in our case the edge cut) of the population to select good individuals and combine them to obtain improved offspring. Roughly speaking, KaFFPaE uses KaFFPa to create individuals and modifies the coarsening phase to provide new effective combine operations.

We adopt the idea of allowing larger imbalance since this is useful to create good partitions [16]. To do so, we modify the create and combine operations as follows: each time we perform such an operation, we randomly choose an imbalance parameter $\epsilon' \in [0.005, \hat{\epsilon}]$ where $\hat{\epsilon}$ is an upper bound for the allowed imbalance (a tuning parameter). This imbalance is then used to perform the operation, i.e. after the operation is performed, the offspring/partition has blocks with size at most $(1 + \epsilon')\lceil |V|/k \rceil$. After the respective operation is performed, we apply our advanced algorithms to obtain a partition of the graph that fulfils the required balance constraint. This individual is the final offspring of the operation. We insert it into the population using the techniques of KaFFPaE [13]. Note that *at all times* each individual in the population of the evolutionary algorithm fulfils the balance constraint. Also note that allowing larger imbalance enables us to use previously developed techniques that otherwise would not be applicable, e.g. max-flow min-cut based local search methods from [12]. We call the overall algorithm Karlsruhe Balanced Partitioner Evolutionary (KaBaPE). When we use KaBaPE to create ϵ-balanced partitions we choose $\epsilon' \in [\epsilon + 0.005, \epsilon + \hat{\epsilon}]$ for the combine and create operations and transform the offspring into an ϵ-balanced partition afterwards.

5 Experiments

Implementation. We have implemented the algorithm described above using C++. We implemented negative cycle detection with subtree disassembly and distance updates as described in [4]. Overall, our program (including KaFFPa(E)) consists of about 23 000 lines of code. The implementation of the presented local search algorithms has about 3 400 lines of code.

System. Experiments have been done on two machines. Machine A has four Quad-core Opteron 8350 (2.0GHz), 64GB RAM, running Ubuntu 10.04. Machine B is a cluster where each node has two Quad-core Intel Xeon processors (X5355, 2.667 GHz) and 16 GB RAM, 2x4 MB of L2 cache and runs Suse Linux Enterprise 11 SP 1. All programs were compiled using GCC Version 4.7 and optimization level 3 using OpenMPI 1.5.5.

Parameters. After an extensive evaluation of the parameters we fixed the number of multiple directed local searches to $\mu = 20$ (larger values of μ, e.g. iterating until no boundary node is eligible did not yield further improvements). The maximum number of node movements per directed local search is set to $\tau = 15$ for $k \leq 8$ and to $\tau = 7$ for $k > 8$. The number of unsuccessful iterations until we perform a balancing step λ is set to three. Each time we perform a create or combine operation we pick a random number of node movements per directed local search $\tau \in [1, 30]$, a random number of multiple directed local searches $\mu \in [1, 20]$ and $\lambda \in [1, 10]$ and use these parameters for the balancing and negative cycle detection strategies.

5.1 Walshaw Benchmark

In this section we apply our techniques to all graphs in Chris Walshaw's benchmark archive [15]. This archive is a collection of real-world instances for the graph partitioning problem. The rules used there imply that the running time is not an issue, but one wants to achieve minimal cut values for $k \in \{2, 4, 8, 16, 32, 64\}$ and balance parameters $\epsilon \in \{0, 0.01, 0.03, 0.05\}$. It is the most used graph partitioning benchmark in the literature. Most of the graphs of the benchmark come from finite-element applications, VLSI design. A road network is also included.

Improving Existing Partitions. When we started to look at perfectly balanced partitioning we counted the number of perfectly balanced partitions in the benchmark archive that contain nodes having positive gain, i.e. nodes that could reduce the cut when being moved to a different block. Astonishingly, we found that 55% of the perfectly balanced partitions in the archive contain nodes with positive gain (some of them have up to 1400 of such nodes). These nodes usually cannot be moved by simple local search due to the balance constraint. Therefore, we now use the existing perfectly balanced partitions in the benchmark archive and use them as input to our local search algorithms KaBaR. This experiment has been performed on machine A and for all configurations of the algorithm we used $\lambda = 20$ for the number of unsuccessful tries. Table 1 shows the relative number of partitions that have been improved by different algorithm configurations and k (in total there are 34 graphs per number of blocks k).

Table 1. Rel. no. of improved instances in the Walshaw Benchmark. Configurations: Basic (Basic Neg. Cycle Impr.), +ZG (Basic + Cycle Diversification), Adv. (Adv. Model + Cycle Div.), +MDLS. (Adv. + MDLS)

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

It is somewhat surprising that already the most basic variant of the algorithm, i.e. negative cycle detection without the zero weight cycle diversification mechanism, can improve 47% of the existing entries. All of the algorithms have a tendency to improve more partitions when the number of blocks k increases. Less surprisingly, more advanced local searches and models increase this percentage further. When applying the advanced algorithm with multiple directed local searches (the most expensive configuration of the algorithm), we are able to improve 128 partitions, i.e. 63% of the entries. Note that it took *overall* roughly two hours to compute these entries using one core of machine A. This is *very affordable* considering the fact that some of the previous approaches, such as Soper et. al. [15], have taken many days to compute *one* entry to the benchmark tables. Of course in practice we want to find high quality partitions without using input partitions generated by other algorithms. We therefore compute partitions from scratch in the next section.

Computing Partitions from Scratch. We now compute perfectly balanced partitions from scratch. We use machine B and run KaBaPE with a time limit $t_k = 225 \cdot k$ seconds using 32 cores (four nodes of the cluster) per graph and $k > 2$. On the eight largest graphs of the archive we gave KaBaPE a time limit of $\hat{t}_k = 4 \cdot t_k$ per graph and $k > 2$. For $k = 2$ we gave KaBaPE one hour of time. $\hat{\epsilon}$ was set to 4% for the small graphs and to 3% for the eight largest graph in the archive. We summarize the results in Table 2 and report the complete list of results obtained in the TR [14]. Currently we are able to improve or reproduce 86% of the entries reported in this benchmark[1]. In the bipartition case we mostly reproduce the entries reported in the benchmark (instead of improving). This is not surprising since the models presented in this paper can contain only trivial cycles of length two in this case. Also recently it has been shown by Delling et. al [5] that some of the balanced bipartitions reported there are optimal. We also applied our algorithm for larger imbalances, i.e. 1%, 3% and 5%, in the Walshaw Benchmark. For the case $\epsilon = 1\%$ we run our algorithm KaBaPE on all instances using the same parameters $\hat{\epsilon}$ and t_k as above. Here we are able to improve or reproduce the cut in 160 out of 204 cases. A table reporting detailed results can be found in the TR [14]. Afterwards we performed additional partitioning trials on all instances where our systems (including [8]. [10]. [12], [13]) currently *not* have been able to reproduce or improve the entry reported there using different parameters and different machines.

Table 2. Number of improvements computed from scratch for the perfectly balanced case

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

[1] 1. Oct. 2012.

We now improved or reproduced 97%, 99%, 99%, 99% of the entries reported there for the cases $\epsilon = 0, 1\%, 3\%, 5\%$ respectively. These numbers include the entries where we used the current record as an input to our algorithms and actually improved the input partition. They contribute roughly 4%, 7%, 11%, 9% for the cases $\epsilon = 0, 1\%, 3\%, 5\%$ respectively.

Costs for Perfect Balance. It is hard to perform a meaningful comparison to other partitioners since publicly available tools such as Scotch [11], Jostle [17] and Metis [9] are either not able to take the desired balance as an input parameter or are not able to guarantee perfect balance. This is a major problem for the comparison with these tools since allowing larger imbalances, i.e. $\epsilon = 3\%$, decreases the number of edges cut significantly. However, we have shown in [12] that KaFFPa produces better partitions compared to Scotch and Metis. Hence, we have a look at the number of edges cut by our algorithm when perfect balance is enforced, i.e. the increase in the number of edges cut when we seek a perfectly balanced partition. We use machine B and KaFFPaStrong to create partitions having an imbalance of $\epsilon = 1\%$ and then create perfectly balanced partitions using our advanced negative cycle model and advanced balancing. KaFFPaStrong is designed to achieve very good partition quality. For each instance (graph, k) we repeat the experiment ten times using different random seeds. We compare the final cuts of the perfectly balanced partitions to the number of edges cut before the balancing and negative cycle search started. We further measure the runtime consumed by the algorithm and report it relative to the runtime of KaFFPa. The instances used for this experiment are the same as in KaFFPa [12] and are available for download at [1]. The main properties of these graphs can be found in the TR [14]. Table 3 summarizes the results, detailed results are reported in the TR [14].

Table 3. Cost for Perfect Balance, Rel. to KaFFPa with $\epsilon = 1\%$ imbalance. Rel. EC average increase in cut after 1% partitions are balanced and Rel. t is average time used by KaBaR rel. time of KaFFPa.

k	**2**	**4**	**8**	**16**	**32**	**64**
Rel. EC [%]	9	7	5	6	4	3
Rel. t [%]	12	56	99	107	134	163

6 Conclusion and Future Work

In this paper we have presented novel algorithms to tackle the balanced graph partitioning problem, including the case of *perfect balance* when the maximal block size has to equal the average block size. These algorithms combine local searches by a model in which a cycle corresponds to a set of node movements in the original partitioned graph that roughly speaking do not alter the balance of the partition. Experiments indicate that previous algorithms have not been able to find such rather complex movements. In contrast to previous algorithms such as Scotch [11], Jostle [17] and Metis [9], our algorithms are able to *guarantee* that the output partition is feasible.

An open question is whether it is possible to define a *conflict-free* model that encodes the same kind of node movements as our advanced model. In future work, it could be interesting to see if one can integrate other types of local searches from KaFFPa [12] into our models. The MDLS algorithm can be improved such that it finds the best

combination of the computed local searches. It will be interesting to see whether our techniques are useful for other problems where local search is restricted by constraints, e.g. multi-constraint or hypergraph partitioning.

Shortly *after* we submitted our results to the benchmark archive we lost entries to an implementation of [7] by Frank Schneider (the original work does not provide perfectly balanced partitions). However, we are still able to improve more than half of these entries when using those as input to KaBaR. Furthermore, we integrated the techniques of [7] and again have been able to improve many entries. We conclude that the algorithms presented in this paper are still very useful.

Acknowledgements. Financial support by the Deutsche Forschungsgemeinschaft (DFG) is gratefully acknowledged (DFG grant SA 933/10-1).

References

1. Bader, D., Meyerhenke, H., Sanders, P., Wagner, D.: 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering
2. Benlic, U., Hao, J.-K.: An effective multilevel tabu search approach for balanced graph partitioning. Computers & OR 38(7), 1066–1075 (2011)
3. Bichot, C., Siarry, P. (eds.): Graph Partitioning. Wiley (2011)
4. Cherkassky, B.V., Goldberg, A.V.: Negative-cycle detection algorithms. In: Díaz, J. (ed.) ESA 1996. LNCS, vol. 1136, pp. 349–363. Springer, Heidelberg (1996)
5. Delling, D., Werneck, R.F.: Better bounds for graph bisection. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 407–418. Springer, Heidelberg (2012)
6. Fiduccia, C.M., Mattheyses, R.M.: A Linear-Time Heuristic for Improving Network Partitions. In: 19th Conference on Design Automation, pp. 175–181 (1982)
7. Galinier, P., Boujbel, Z., Coutinho Fernandes, M.: An efficient memetic algorithm for the graph partitioning problem. Annals of Operations Research, 1–22 (2011)
8. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a Scalable High Quality Graph Partitioner. In: 24th IEEE IPDPS, pp. 1–12 (2010)
9. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. SIAM Review 41(2), 278–300 (1999)
10. Osipov, V., Sanders, P.: *n*-level graph partitioning. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part I. LNCS, vol. 6346, pp. 278–289. Springer, Heidelberg (2010)
11. Pellegrini, F.: `http://www.labri.fr/perso/pelegrin/scotch/`
12. Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 469–480. Springer, Heidelberg (2011)
13. Sanders, P., Schulz, C.: Distributed evolutionary graph partitioning. In: ALENEX, pp. 16–29. SIAM/Omnipress (2012)
14. Sanders, P., Schulz, C.: Think Locally, Act Globally: Perfectly Balanced Graph Partitioning. Technical Report. arXiv:1210.0477 (2012)
15. Soper, A.J., Walshaw, C., Cross, M.: A combined evolutionary search and multilevel optimisation approach to graph-partitioning. J. of Global Optimization 29(2), 225–241 (2004)
16. Walshaw, C., Cross, M.: Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. SIAM Journal on Scientific Computing 22(1), 63–80 (2000)
17. Walshaw, C., Cross, M.: JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In: Mesh Partitioning Techniques and Domain Decomposition Techniques, pp. 27–58. Civil-Comp Ltd. (2007)