High Quality Graph Partitioning

Peter Sanders and Christian Schulz

Abstract. We present an overview over our graph partitioners KaFFPa (Karlsruhe Fast Flow Partitioner) and KaFFPaE (KaFFPa Evolutionary). KaFFPa is a multilevel graph partitioning algorithm which on the one hand uses novel local improvement algorithms based on max-flow and min-cut computations and more localized FM searches and on the other hand uses more sophisticated global search strategies transferred from multi-grid linear solvers. KaFFPaE is a distributed evolutionary algorithm to solve the Graph Partitioning Problem. KaFFPaE uses KaFFPa and provides new effective crossover and mutation operators. By combining these with a scalable communication protocol we obtain a system that is able to improve the best known partitioning results for many inputs.

1. Introduction

Problems of *graph partitioning* arise in various areas of computer science, engineering, and related fields. For example in route planning, community detection in social networks and high performance computing. In many of these applications large graphs need to be partitioned such that there are few edges between blocks (the elements of the partition). For example, when you process a graph in parallel on k processors you often want to partition the graph into k blocks of about equal size so that there is as little interaction as possible between the blocks. 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. It is well known that this problem is NP-complete [**5**] and that there is no approximation algorithm with a constant ratio factor for general graphs [**5**]. Therefore mostly heuristic algorithms are used in practice. 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 structure as the input graph. After applying an *initial partitioning* algorithm to the smallest graph, the contraction is undone and, at each level, a *local refinement* method is used to improve the partitioning induced by the coarser level.

Although several successful multilevel partitioners have been developed in the last 13 years, we had the impression that certain aspects of the method are not

Partially supported by DFG SA 933/10-1.

well understood. We therefore have built our own graph partitioner KaPPa [**13**] (Karlsruhe Parallel Partitioner) with focus on scalable parallelization. Somewhat astonishingly, we also obtained improved partitioning quality through rather simple methods. This motivated us to make a fresh start putting all aspects of MGP on trial. This paper gives an overview over our most recent work, KaFFPa [**22**] and KaFFPaE [**21**]. KaFFPa is a classical matching based graph partitioning algorithm with focus on local improvement methods and overall search strategies. It is a system that can be configured to either achieve the best known partitions for many standard benchmark instances or to be the fastest available system for large graphs while still improving partitioning quality compared to the previous fastest system.

KaFFPaE is a technique which integrates an evolutionary search algorithm with our multilevel graph partitioner KaFFPa and its scalable parallelization. It uses novel mutation and combine operators which in contrast to previous evolutionary methods that use a graph partitioner [**23, 8**] do not need random perturbations of edge weights. The combine operators enable us to combine individuals of different kinds (see Section 5 for more details). Due to the parallelization our system is able to compute partitions that have quality comparable or better than previous entries in Walshaw's well known partitioning benchmark *within a few minutes* for graphs of moderate size. Previous methods of Soper et. al [**23**] required runtimes of up to one week for graphs of that size. We therefore believe that in contrast to previous methods, our method is very valuable in the area of high performance computing.

The paper is organized as follows. We begin in Section 2 by introducing basic concepts which is followed by related work in Section 3. In Section 4 we present the techniques used in the multilevel graph partitioner KaFFPa. We continue describing the main components of our evolutionary algorithm KaFFPaE in Section 5. A summary of extensive experiments to evaluate the performance of the algorithm is presented in Section 6. We have implemented these techniques in the graph partitioner KaFFPaE (Karlsruhe Fast Flow Partitioner Evolutionary) which is written in C++. Experiments reported in Section 6 indicate that KaFFPaE is able to compute partitions of very high quality and scales well to large networks and machines.

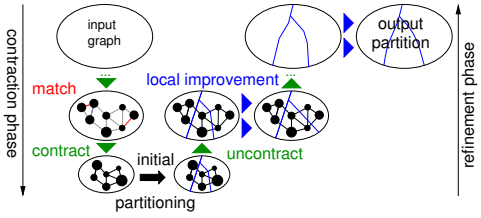

FIGURE 1. Multilevel graph partitioning.

2. Preliminaries

2.1. Basic concepts. Consider an undirected graph $G = (V, E, c, \omega)$ with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. $\Gamma(v) := \{u : \{v, u\} \in E\}$ denotes the neighbors of v. We are looking for *blocks* of nodes $V_1, \ldots, V_k$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in \{1..k\} : c(V_i) \leq L_{\max} := (1 + \epsilon)c(V)/k + \max_{v \in V} c(v)$ for some parameter ϵ. The last term in this equation arises because each node is atomic and therefore a deviation of the heaviest node has to be allowed. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where

$E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. A clustering is also a partition of the nodes, however k is usually not given in advance and the balance constraint is removed. A vertex $v \in V_i$ that has a neighbor $w \in V_j, i \neq j$, is a boundary vertex. 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. Given two clusterings $\mathcal{C}_1$ and $\mathcal{C}_2$ the *overlay clustering* is the clustering where each block corresponds to a connected component of the graph $G_{\mathcal{E}} = (V, E \backslash \mathcal{E})$ where $\mathcal{E}$ is the union of the cut edges of $\mathcal{C}_1$ and $\mathcal{C}_2$, i.e. all edges that run between blocks in $\mathcal{C}_1$ or $\mathcal{C}_2$. We will need the of overlay clustering to define a combine operation on partitions in Section 5. By default, our initial inputs will have unit edge and node weights. However, even those will be translated into weighted problems in the course of the algorithm.

A matching $M \subseteq E$ is a set of edges that do not share any common nodes, i.e., the graph (V, M) has maximum degree one. *Contracting* an edge $\{u, v\}$ means to replace the nodes u and v by a new node x connected to the former neighbors of u and v. We set $c(x) = c(u) + c(v)$ so the weight of a node at each level is the number of nodes it is representing in the original graph. If replacing edges of the form $\{u, w\}$,$\{v, w\}$ would generate two parallel edges $\{x, w\}$, we insert a single edge with $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$. *Uncontracting* an edge e undoes its contraction. In order to avoid tedious notation, G will denote the current state of the graph before and after a (un)contraction unless we explicitly want to refer to different states of the graph. The *multilevel approach* to graph partitioning consists of three main phases. In the *contraction* (coarsening) phase, we iteratively identify matchings $M \subseteq E$ and contract the edges in M. Contraction should quickly reduce the size of the input and each computed level should reflect the structure of the input network. Contraction is stopped when the graph is small enough to be directly partitioned using some expensive other algorithm. In the *refinement* (or uncoarsening) phase, the matchings are iteratively uncontracted. After uncontracting a matching, a refinement algorithm moves nodes between blocks in order to improve the cut size or balance.

3. Related Work

There has been a huge amount of research on graph partitioning so that we refer the reader to [**26**] for more material on multilevel graph partitioning and to [**15**] for more material on genetic approaches for graph partitioning. All general purpose methods that are able to obtain good partitions for large real world graphs are based on the multilevel principle outlined in Section 2. Well known software packages based on this approach include, Jostle [**26**], Metis [**14**], and Scotch [**20**]. KaSPar [**19**] is a graph partitioner based on the central idea to (un)contract only a single edge between two levels. KaPPa [**13**] is a "classical" matching based MGP algorithm designed for scalable parallel execution. MQI [**16**] and Improve [**2**] are flow-based methods for improving graph cuts when cut quality is measured by quotient-style metrics such as *expansion* or *conductance*. This approach is only feasible for $k = 2$. Improve uses several minimum cut computations to improve the *quotient cut* score of a proposed partition. Soper et al. [**23**] provided the first algorithm that combined an evolutionary search algorithm with a multilevel graph partitioner. Here crossover and mutation operators have been used to compute edge biases, which yield hints for the underlying multilevel graph partitioner. Benlic et al. [**4**] provided a multilevel memetic algorithm for balanced graph partitioning.

This approach is able to compute many entries in Walshaw's Benchmark Archive [**23**] for the case $\epsilon = 0$. Very recently an algorithm called PUNCH [**8**] has been introduced. This approach is not based on the multilevel principle. However, it creates a coarse version of the graph based on the notion of natural cuts. Natural cuts are relatively sparse cuts close to denser areas. They are discovered by finding minimum cuts between carefully chosen regions of the graph. They introduced an evolutionary algorithm which is similar to Soper et al. [**23**], i.e. using a combine operator that computes edge biases yielding hints for the underlying graph partitioner. Experiments indicate that the algorithm computes very good partitions for road networks. For instances without a natural structure natural cuts are not very helpful.

4. Karlsruhe Fast Flow Partitioner

The aim of this section is to provide an overview over the techniques used in KaFFPa which is used by KaFFPaE as a base case partitioner. KaFFPa [**22**] is a classical matching based multilevel graph partitioner. Recall that a multilevel graph partitioner basically has three phases: coarsening, initial partitioning and uncoarsening.

Coarsening. KaFFPa makes contraction more systematic by separating two issues: A *rating function* indicates how much sense it makes to contract an edge based on *local* information. A *matching* algorithm tries to maximize the sum of the ratings of the contracted edges looking at the *global* structure of the graph. While the rating function allows a flexible characterization of what a "good" contracted graph is, the simple, standard definition of the matching problem allows to reuse previously developed algorithms for weighted matching. Matchings are contracted until the graph is "small enough". In [**13**] we have observed that the rating function $\text{expansion}^{*2}(\{u,v\}) := \frac{\omega(\{u,v\})^2}{c(u)c(v)}$ works best among other edge rating functions, so that this rating function is also used in KaFFPa.

KaFFPa employs the *Global Path Algorithm (GPA)* as a matching algorithm. It was proposed in [**17**] as a synthesis of the Greedy algorithm and the Path Growing Algorithm [**10**]. This algorithm achieves a half-approximation in the worst case, but empirically, GPA gives considerably better results than Sorted Heavy Edge Matching and Greedy (for more details see [**13**]). 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 the threshold $\max(60k, n/(60k))$. The graph is then small enough to be partitioned by some initial partitioning algorithm. KaFFPa employs Scotch as an initial partitioner since it empirically performs better than Metis.

Uncoarsening. Recall that the refinement phase iteratively uncontracts the matchings contracted during the contraction phase. After a matching is uncontracted, local search based refinement algorithms move nodes between block boundaries in order to reduce the cut while maintaining the balancing constraint. Local improvement algorithms are usually variants of the FM-algorithm [**12**]. Our variant of the algorithm is organized in rounds. In each round, a priority queue P is used which is initialized with all vertices that are incident to more than one block, in a random order. The priority is based on the gain $g(v) = \max_P g_P(v)$ where $g_P(v)$ is the

decrease in edge cut when moving v to block P. Ties are broken randomly if there is more than one block that yields the maximum gain when moving v to it. Local search then repeatedly looks for the highest gain node v. Each node is moved at most once within a round. After a node is moved its unmoved neighbors become eligible, i.e. its unmoved neighbors are inserted into the priority queue. When a stopping criterion is reached all movements to the best found cut that occurred within the balance constraint are undone. This process is repeated several times until no improvement is found.

Max-Flow Min-Cut Local Improvement. During the uncoarsening phase KaFFPa additionally uses more advanced refinement algorithms. The first method is based on max-flow min-cut computations between pairs of blocks, i.e., a method to improve a given bipartition. Roughly speaking, this improvement method is applied between all pairs of blocks that share a non-empty boundary. The algorithm basically constructs a flow problem by growing an area around the given boundary vertices of a pair of blocks such that each min cut in this area yields a feasible bipartition of the original graph *within* the balance constraint. We explain how flows can be employed to improve a partition of *two blocks* V_1, V_2 without violating the balance constraint. That yields a local improvement algorithm. First we introduce a few notations. Given a set of nodes $B \subset V$ we define its *border* $\partial B := \{u \in B \mid \exists (u,v) \in E : v \notin B\}$. The set $\partial_1 B := \partial B \cap V_1$ is called *left border* of B and the set $\partial_2 B := \partial B \cap V_2$ is called *right border* of B. A B *induced subgraph* G' is the node induced subgraph $G[B]$ plus two nodes s, t that are connected to the border of B. More precisely s is connected to all left border nodes $\partial_1 B$ and all right border nodes $\partial_2 B$ are connected to t. All of these new edges get the edge weight ∞. Note that the additional edges are directed. G' has the *cut property* if each (s,t)-min-cut induces a cut within the balance constraint in G.

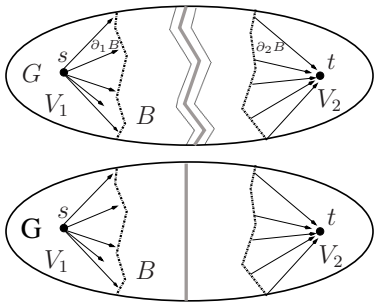

FIGURE 2. The construction of a feasible flow problem G' is shown on the top and an improved cut within the balance constraint in G is shown on the bottom.

The basic idea is to construct a B induced subgraph G' having the cut property. Each min-cut will then yield a feasible improved cut within the balance constraint in G. By performing two Breadth First Searches (BFS) we can find such a set B. Each node touched during these searches belongs to B. The first BFS is done in the subgraph of G induced by V_1. It is initialized with the boundary nodes of V_1. As soon as the weight of the area found by this BFS would exceed $(1+\epsilon)c(V)/2 - c(V_1)$, we stop the BFS. The second BFS is done for V_2 in an analogous fashion. The constructed subgraph G' has the cut property since the worst case new weight of V_2 is lower or equal to $c(V_2) + (1+\epsilon)c(V)/2 - c(V_2) = (1+\epsilon)c(V)/2$. Indeed the same holds for the worst case new weight of V_1. There are multiple ways to improve this method, i.e. iteratively applying the method, searching in larger areas for feasible cuts and

applying most balanced minimum cut heuristics. For more details we refer the reader to [**22**].

Multi-try FM. The second novel method for improving a given partition is called multi-try FM. This local improvement method moves nodes between blocks in order to decrease the cut. Previous k-way methods were initialized with *all* boundary nodes, i.e., all boundary nodes are eligible for movement at the beginning. Our method is repeatedly initialized with a *single* boundary node, thus achieving a more localized search. More details about k-way methods can be found in [**22**]. Multi-try FM is organized in rounds. In each round we put *all* boundary nodes of the current block pair into a todo list T. Subsequently, we begin a k-way local search starting with a *single* random node v of T if it is still a boundary node. Note that the difference to the global k-way search is in the initialisation of the search. The local search is only started from v if it was not touched by a previous localized k-way search in this round. Either way, the node is removed from the todo list. A localized k-way search is not allowed to move a node that has been touched in a previous run. This assures that at most n nodes are touched during a round of the algorithm. The algorithm uses the adaptive stopping criterion from KaSPar [**19**].

Global Search. KaFFPa extended the concept of *iterated multilevel algorithms* which was introduced by [**24**]. The main idea is to iterate the coarsening and uncoarsening phase. Once the graph is partitioned, edges that are between two blocks are not contracted. An F-cycle works as follows: on *each* level we perform at most *two recursive calls* using different random seeds during contraction and local search. A second recursive call is only made the second time that the algorithm reaches a particular level. Figure 3 illustrates a F-cycle. As soon as the graph is partitioned, edges that are between blocks are not contracted. This ensures nondecreasing quality of the partition since our refinement algorithms guarantee no worsening and break ties randomly. These so called *global search strategies* are more effective than plain restarts of the algorithm. *Extending this idea* will yield the combine and mutation operators described in Section 5.

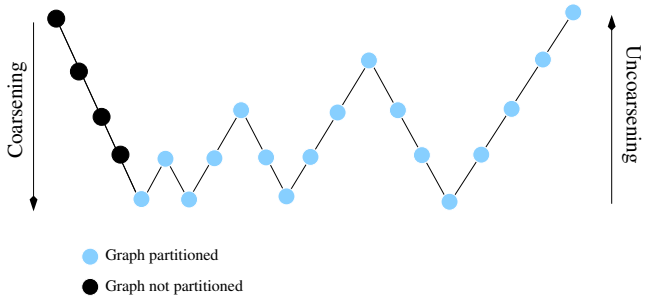

FIGURE 3. An F-cycle for the graph partitioning problem.

5. KaFFPa Evolutionary

We now describe the techniques used in KaFFPaE. The general idea behind evolutionary algorithms (EA) is to use mechanisms which are highly inspired by biological evolution such as selection, mutation, recombination and survival of the fittest. An EA 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 EA 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. Note that we can use the cut as a fitness function since our partitioner almost always generates partitions that are within the given balance constraint. Our algorithm generates only one offspring per generation. Such an evolutionary algorithm is called *steady-state* [**7**]. A typical structure of an evolutionary algorithm is depicted in Algorithm 1.

For an evolutionary algorithm it is of major importance to keep the diversity in the population high, i.e. the individuals should not become too similar, in order to avoid a premature convergence of the algorithm. In classical evolutionary algorithms, this is done using a mutation operator. It is also important to have operators that introduce unexplored search space to the population. Through a new kind of crossover and mutation operators, introduced in Section 5.1, we introduce more elaborate diversification strategies which allow us to search the search space more effectively.

Algorithm 1 A classic general steady-state evolutionary algorithm.

```
procedure steady-state-EA
  create initial population P
  while stopping criterion not fulfilled
    select parents p1, p2 from P
    combine p1 with p2 to create offspring o
    mutate offspring o
    evict individual in population using o
  return the fittest individual that occurred
```

5.1. Combine Operators. We now describe the general combine operator framework. This is followed by three instantiations of this framework. In contrast to previous methods that use a multilevel framework our combine operators do not need perturbations of edge weights since we integrate the operators into our partitioner and do not use it as a complete black box. Furthermore all of our combine operators assure that the offspring has a partition quality *at least as good as the best of both parents*. Roughly speaking, the combine operator framework combines an individual/partition $\mathcal{P} = V_1^{\mathcal{P}}, ..., V_k^{\mathcal{P}}$ (which has to fulfill a balance constraint) with a clustering $\mathcal{C} = V_1^{\mathcal{C}}, ..., V_{k'}^{\mathcal{C}}$. Note that the clustering does not necessarily has to fulfill a balance constraint and k' is not necessarily given in advance. All instantiations of this framework use a different kind of clustering or partition. The partition and the clustering are both used as input for our multi-level graph partitioner KaFFPa in the following sense. Let $\mathcal{E}$ be the set of edges that are cut edges, i.e. edges that run between two blocks, in $\mathcal{P}$ *or* $\mathcal{C}$. All edges in $\mathcal{E}$ are blocked during the coarsening phase, i.e. they *are not contracted* during the coarsening phase. In other words these edges are not eligible for the matching algorithm used during the coarsening phase and therefore are not part of any matching computed. An illustration of this can be found in Figure 4.

The stopping criterion for the multi-level partitioner is modified such that it stops when no contractable edge is left. Note that the coarsest graph is now exactly the same as the quotient graph $\mathcal{Q}'$ of the overlay clustering of $\mathcal{P}$ and $\mathcal{C}$ of G (see Figure 5). Hence vertices of the coarsest graph correspond to the

connected components of $G_{\mathcal{E}} = (V, E \backslash \mathcal{E})$ and the weight of the edges between vertices corresponds to the sum of the edge weights running between those connected components in G. As soon as the coarsening phase is stopped, we apply the partition $\mathcal{P}$ to the coarsest graph and use this as initial partitioning. This is possible since we did not contract any cut edge of $\mathcal{P}$. Note that due to the specialized coarsening phase and this specialized initial partitioning we obtain a high quality initial solution on a very coarse graph which is usually not discovered by conventional partitioning algorithms. Since our refinement algorithms guarantee no worsening of the input partition and use random tie breaking we can assure nondecreasing partition quality. Note that the refinement algorithms can effectively exchange good parts of the solution on the coarse levels by moving only a few vertices. Figure 5 gives an example.

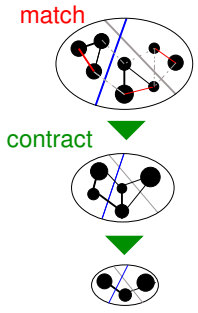

FIGURE 4. On the top a graph G with two partitions, the dark and the light line, are shown. Cut edges are not eligible for the matching algorithm. Contraction is done until no matchable edge is left. The best of the two given partitions is used as initial partition.

When the offspring is generated we have to decide which solution should be evicted from the current population. We evict the solution that is *most similar* to the offspring among those individuals in the population that have a cut worse or equal than the offspring itself. The difference of two individuals is defined as the size of the symmetric difference between their sets of cut edges. This ensures some diversity in the population and hence makes the evolutionary algorithm more effective.

5.1.1. *Classical Combine using Tournament Selection.* This instantiation of the combine framework corresponds to a classical evolutionary combine operator C_1. That means it takes two individuals P_1, P_2 of the population and performs the combine step described above. In this case $\mathcal{P}$ corresponds to the partition having the smaller cut and $\mathcal{C}$ corresponds to the partition having the larger cut. Random tie breaking is used if both parents have the same cut. The selection process is based on the tournament selection rule [**18**], i.e. P_1 is the fittest out of two random individuals R_1, R_2 from the population. The same is done to select P_2. Note that in contrast to previous methods the generated offspring will have a cut smaller or equal to the cut of $\mathcal{P}$. Due to the fact that our multi-level algorithms are randomized, a combine operation performed twice using the same parents can yield different offspring.

5.1.2. *Cross Combine / (Transduction).* In this instantiation of the combine framework C_2, the clustering $\mathcal{C}$ corresponds to a partition of G. But instead of choosing an individual from the population we create a new individual in the following way. We choose k' uniformly at random in $[k/4, 4k]$ and ϵ' uniformly at random in $[\epsilon, 4\epsilon]$. We then use KaFFPa to create a k'-partition of G fulfilling the

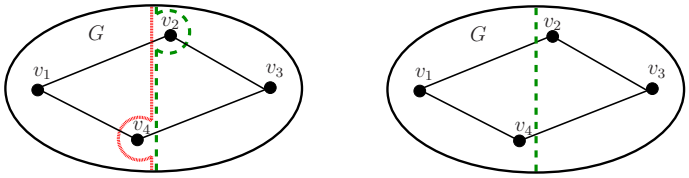

FIGURE 5. A graph G and two bipartitions; the dotted and the dashed line (left). Curved lines represent a large cut. The four vertices correspond to the coarsest graph in the multilevel procedure. Local search algorithms can effectively exchange v_2 or v_4 to obtain the better partition depicted on the right hand side (dashed line).

balance constraint $\max c(V_i) \leq (1+\epsilon')c(V)/k'$. In general larger imbalances reduce the cut of a partition which then yields good clusterings for our crossover. To the best of our knowledge there has been no genetic algorithm that performs combine operations combining individuals from different search spaces.

5.1.3. *Natural Cuts.* Delling et al. [**8**] introduced the notion of *natural cuts* as a preprocessing technique for the partitioning of road networks. The preprocessing technique is able to find relatively sparse cuts close to denser areas. We use the computation of natural cuts to provide another combine operator, i.e. combining a k-partition with a clustering generated by the computation of natural cuts. We closely follow their description: The computation of natural cuts works in rounds. Each round picks a center vertex v and grows a breadth-first search (BFS) tree. The BFS is stopped as soon as the weight of the tree, i.e. the sum of the vertex weights of the tree, reaches αU, for some parameters α and U. The set of the neighbors of T in $V \backslash T$ is called the *ring* of v. The *core* of v is the union of all vertices added to T before its size reached $\alpha U/f$ where $f > 1$ is another parameter. The core is then temporarily contracted to a single vertex s and the ring into a single vertex t to compute the minimum s-t-cut between them using the given edge weights as capacities. To assure that every vertex eventually belongs to at least one core, and therefore is inside at least one cut, the vertices v are picked uniformly at random among all vertices that have not yet been part of any core in any round. The process is stopped when there are no such vertices left. In the original work [**8**] each connected component of the graph $G_C = (V, E \backslash C)$,

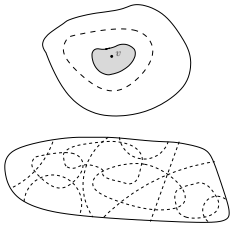

FIGURE 6. On the top we see the computation of a natural cut. A BFS Tree which starts from v is grown. The gray area is the core. The dashed line is the natural cut. It is the minimum cut between the contracted versions of the core and the ring (shown as the solid line). During the computation several natural cuts are detected in the input graph (bottom).

where C is the union of all edges cut by the process above, is contracted to a single vertex. Since we do not use natural cuts as a preprocessing technique at this place we don't contract these components. Instead we build a clustering $\mathcal{C}$ of G such that each connected component of G_C is a block.

This technique yields the third instantiation of the combine framework C_3 which is divided into two stages, i.e. the clustering used for this combine step is dependent on the stage we are currently in. In both stages the partition $\mathcal{P}$ used for the combine step is selected from the population using tournament selection. During the first stage we choose f uniformly at random in $[5, 20]$, α uniformly at random in $[0.75, 1.25]$ and we set $U = |V|/3k$. Using these parameters we obtain a clustering $\mathcal{C}$ of the graph which is then used in the combine framework described above. This kind of clustering is used until we reach an upper bound of ten calls to this combine step. When the upper bound is reached we switch to the second stage. In this stage we use the clusterings computed during the first stage, i.e. we extract elementary natural cuts and use them to quickly compute new clusterings. An *elementary natural cut* (ENC) consists of a set of cut edges and the set of nodes in its core. Moreover, for each node v in the graph, we store the set of ENCs $N(v)$ that contain v in their core. With these data structures it is easy to pick a new clustering $\mathcal{C}$ (see Algorithm 2) which is then used in the combine framework described above.

Algorithm 2 computeNaturalCutClustering (second stage)

```
1: unmark all nodes in V
2: for each v ∈ V in random order do
3:   if v is not marked then
4:     pick a random ENC C in N(v)
5:     output C
6:     mark all nodes in C's core
```

5.2. Mutation Operators. We define two mutation operators, an ordinary and a modified F-cycle. Both mutation operators use a random individual from the current population. The main idea is to iterate coarsening and refinement several times using different seeds for random tie breaking. The first mutation operator M_1 can assure that the quality of the input partition does not decrease. It is basically an ordinary F-cycle which is an algorithm used in KaFFPa. Edges between blocks are not contracted. The given partition is then used as initial partition of the coarsest graph. In contrast to KaFFPa, we now can use the partition as input to the partition in the very beginning. This ensures nondecreasing quality since our refinement algorithms guarantee no worsening. The second mutation operator M_2 works quite similar with the small difference that the input partition is not used as initial partition of the coarsest graph. That means we obtain very good coarse graphs but we cannot assure that the final individual has a higher quality than the input individual. In both cases the resulting offspring is inserted into the population using the eviction strategy described in Section 5.1.

5.3. Putting Things Together and Parallelization. We now explain the parallelization and describe how everything is put together. Each processing element (PE) basically performs the same operations using different random seeds (see Algorithm 3). First we estimate the population size S: each PE performs a

partitioning step and measures the time $\bar{t}$ spent for partitioning. We then choose S such that the time for creating S partitions is approximately t_{total}/f where the fraction f is a tuning parameter and t_{total} is the total running time that the algorithm is given to produce a partition of the graph. Each PE then builds its own population, i.e. KaFFPa is called several times to create S individuals/partitions. Afterwards the algorithm proceeds in rounds as long as time is left. With corresponding probabilities, mutation or combine operations are performed and the new offspring is inserted into the population.

We choose a parallelization/communication protocol that is quite similar to *randomized rumor spreading* [**9**]. Let p denote the number of PEs used. A communication step is organized in rounds. In each round, a PE chooses a communication partner and sends her the currently best partition P of the local population. The selection of the communication partner is done uniformly at random among those PEs to which P not already has been sent to. Afterwards, a PE checks if there are incoming individuals and if so inserts them into the local population using the eviction strategy described above. If P is improved, all PEs are again eligible. This is repeated $\log p$ times. Note that the algorithm is implemented *completely asynchronously*, i.e. there is no need for a global synchronisation. The process of creating individuals is parallelized as follows: Each PE makes $s' = |S|/p$ calls to KaFFPa using different seeds to create s' individuals. Afterwards we do the following $S - s'$ times: The root PE computes a random cyclic permutation of all PEs and broadcasts it to all PEs. Each PE then sends a random individual to its successor in the cyclic permutation and receives a individual from its predecessor in the cyclic permutation which is then inserted into the local population. When this particular part of the algorithm (*quick start*) is finished, each PE has $|S|$ partitions.

After some experiments we fixed the ratio of mutation to crossover operations to $1 : 9$, the ratio of the mutation operators $M_1 : M_2$ to $4 : 1$ and the ratio of the combine operators $C_1 : C_2 : C_3$ to $3 : 1 : 1$. Note that the communication step in the last line of the algorithm could also be performed only every x iterations (where x is a tuning parameter) to save communication time. Since the communication network of our test system is very fast (see Section 6), we perform the communication step in each iteration.

Algorithm 3 All PEs perform the same operations using different random seeds.

```
procedure locallyEvolve
  estimate population size S
  while time left
    if elapsed time < t_total/f then create individual and insert into local population
    else
      flip coin c with corresponding probabilities
      if c shows head then
        perform a mutation operation
      else
        perform a combine operation
      insert offspring into population if possible
    communicate according to communication protocol
```

6. Experiments

Implementation. We have implemented the algorithm described above using C++. Overall, our program (including KaFFPa and KaFFPaE) consists of about 22 500 lines of code. We use three configurations of KaFFPa: KaFFPaStrong, KaFFPaEco and KaFFPaFast. KaFFPaFast is the fastest configuration, KaFFPaEco is a good tradeoff between quality and speed, and KaFFPaStrong is focused on quality (see [**22**] for more details).

Systems. Experiments have been done on three machines. Machine A is a cluster with 200 nodes where each node is equipped with two Quad-core Intel Xeon processors (X5355) which run at a clock speed of 2.667 GHz. Each node has 2x4 MB of level 2 cache each and runs Suse Linux Enterprise 10 SP 1. All nodes are attached to an InfiniBand 4X DDR interconnect which is characterized by its very low latency of below 2 microseconds and a point to point bandwidth between two nodes of more than 1300 MB/s. Machine B has four Quad-core Opteron 8350 (2.0GHz), 64GB RAM, running Ubuntu 10.04. Machine C has two Intel Xeon X5550, 48GB RAM, running Ubuntu 10.04. Each CPU has 4 cores (8 cores when hyperthreading is active) running at 2.67 GHz. Experiments in Section 6.1 were conducted on machine A. Shortly after these experiments were conducted the machine had a file system crash and was not available for two weeks (and after that the machine was very full). Therefore we switched to the much smaller machines B and C, focused on a small subset of the challenge and restricted further experiments to $k = 8$. Experiments in Section 6.2 have been conducted on machine B, and experiments in Section 6.3 have been conducted on machine C. All programs were compiled using GCC Version 4.4.3 and optimization level 3 using OpenMPI 1.5.3. Henceforth, a PE is one core of a machine.

graph	n	m
Random Geometric Graphs		
rgg16	2^{16}	≈342 K
rgg17	2^{17}	≈729 K
Delaunay		
delaunay16	2^{16}	≈197 K
delaunay17	2^{17}	≈393 K
Kronecker G500		
kron_simple_16	2^{16}	≈2 M
kron_simple_17	2^{17}	≈5 M
Numerical		
adaptive	≈6 M	≈14 M
channel	≈5 M	≈42 M
venturi	≈4 M	≈8 M
packing	≈2 M	≈17 M
2D Frames		
hugetrace-00000	≈5 M	≈7 M
hugetric-00000	≈6 M	≈9 M
Sparse Matrices		
af_shell9	≈500 K	≈9 M
thermal2	≈1 M	≈4 M
Coauthor Networks		
coAutCiteseer	≈227 K	≈814 K
coAutDBLP	≈299 K	≈978 K
Social Networks		
cnr	≈326 K	≈3 M
caidaRouterLvl	≈192 K	≈609 K
Road Networks		
luxembourg	≈144 K	≈120 K
belgium	≈1 M	≈2 M
netherlands	≈2 M	≈2 M
italy	≈7 M	≈7 M
great-britain	≈8 M	≈8 M
germany	≈12 M	≈12 M
asia	≈12 M	≈13 M
europe	≈51 M	≈54 M

TABLE 1. Basic properties of choosen subset (except Walshaw Instances).

Instances. We report experiments on a subset of the graphs of the 10th DIMACS Implementation Challenge [**3**]. Experiments in Section 6.1 were done on all

graphs of the Walshaw Benchmark. Here we used $k \in \{2, 4, 8, 16, 32, 64\}$ since they are the default values in [**25**]. Experiments in Section 6.2 focus on the graph subset depicted in Table 1 (except the road networks). In Section 6.3 we have a closer look on all road networks of the Challenge. We finish the experimental evaluation with Section 6.4 describing how we obtained the results on the challenge testbed and comparing the performance of Metis and Scotch. Our default value for the allowed imbalance is 3% since this is one of the values used in [**25**] and the default value in Metis. Our default number of PEs is 16.

6.1. Walshaw Benchmark. [1] We now apply KaFFPaE to Walshaw's benchmark archive [**23**] using the rules used there, i.e., running time is not an issue but we want to obtain 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 focus on $\epsilon \in \{1\%, 3\%, 5\%\}$ since KaFFPaE (more precisely KaFFPa) is not made for the case $\epsilon = 0$. We run KaFFPaE with a time limit of two hours using 16 PEs (two nodes of the cluster) per graph, k and ϵ. On the eight largest graphs of the archive we gave KaFFPaE eight hours per graph, k and ϵ. KaFFPaE computed 300 partitions which are better than previous best partitions reported there: 91 for 1%, 103 for 3% and 106 for 5%. Moreover, it reproduced *equally sized* cuts in 170 of the 312 remaining cases. When only considering the 15 largest graphs and $\epsilon \in \{1.03, 1.05\}$ we are able to reproduce or improve the current result in 224 out of 240 cases. Overall our systems (including KaPPa, KaSPar, KaFFPa, KaFFPaE) now improved or reproduced the entries in 550 out of 612 cases (for $\epsilon \in \{0.01, 0.03, 0.05\}$).

6.2. Various DIMACS Graphs. In this Section we apply KaFFPaE (and on some graphs KaFFPa) to a meaningful subset of the graphs of the DIMACS Challenge. Here we use all cores of machine B and give KaFFPaE eight hours of time per graph to compute a partition into eight blocks. When using KaFFPa to create a partition we use one core of this machine. The experiments were repeated three times. A summary of the results can be found in Table 2.

graph	best	avg.	graph	best	avg.
rgg16	1 067	1 067	coAutDBLP	94 866	95 866
rgg17	1 777	1 778	channel$_*$	333 396	333 396
delaunay16	1 547	1 547	packing$_*$	108 771	111 255
delaunay17	2 200	2 203	adaptive	8 482	8 482
kron_simple_16$_*$	1 257 512	1 305 207	venturi	5 780	5 788
kron_simple_17$_*$	2 247 116	2 444 925	hugetrace-00000	3 656	3 658
cnr	4 687	4 837	hugetric-00000	4 769	4 785
caidaRouterLevel	42 679	43 659	af_shell9	40 775	40 791
coAutCiteseer	42 875	43 295	thermal2	6 426	6 426

TABLE 2. Results achieved for $k = 8$ various graphs of the DIMACS Challenge. Results which were computed by KaFFPa are indicated by *.

[1] see KaFFPaE [**21**] for more details on this experiment.

6.3. Road Networks. In this Section we focus on finding partitions of the street networks of the DIMACS Challenge. We implemented a specialized algorithm, *Buffoon*, which is similar to PUNCH [**8**] in the sense that it also uses natural cuts as a preprocessing technique to obtain a coarser graph on which the graph partitioning problem is solved. For more information on natural cuts, we refer the reader to [**8**]. Using our (shared memory) parallelized version of natural cut preprocessing we obtain a coarse version of the graph. Note that our preprocessing uses slightly different parameters than PUNCH (using the notation of [**8**], we use $\mathcal{C} = 2, U = (1+\epsilon)\frac{n}{2k}, f = 10, \alpha = 1$). Since partitions of the coarse graph correspond to partitions of the original graph, we use KaFFPaE to partition the coarse version of the graph. After preprocessing, we gave KaFFPaE one hour of time to compute a partition. In both cases we used all 16 cores (hyperthreading active) of machine C for preprocessing and for KaFFPaE. We also used the strong configuration of KaFFPa to partition the road networks. In both cases the experiments were repeated ten times. Table 3 summarizes the results.

grp.	algorithm/runtime t					
	B_{best}	$\mathrm{B}_{\mathrm{avg}}$	$t_{\mathrm{avg}}[m]$	K_{best}	$\mathrm{K}_{\mathrm{avg}}$	$t_{\mathrm{avg}}[m]$
lux.	**79**	79	60.1	81	83	0.1
bel.	**307**	307	60.5	320	326	0.9
net.	**191**	193	60.6	207	217	1.2
ita.	**200**	200	64.3	205	210	3.9
gb.	**363**	365	63.0	381	395	6.5
ger.	**473**	475	65.3	482	499	11.3
asia.	**47**	47	67.6	52	55	6.4
eur.	**526**	527	131.5	550	590	76.1

TABLE 3. Results on road networks for $k = 8$: average and best cut results of Buffoon (B) and KaFFPa (K) as well as average runtime [m] (including preprocessing) .

6.4. The Challenge Testbed. We now describe how we obtained the results on the challenge testbed and evaluate the performance of kMetis and Scotch on these graphs in the Pareto challenge.

Pareto Challenge. For this particular challenge we run all configurations of KaFFPa (KaFFPaStrong, KaFFPaEco, KaFFPaFast, see [**22**] for details), KaFFPaE, Metis 5.0 and Scotch 5.1 on machine A. To compute a partition for an instance (graph, k) we repeatedly run the corressponding partitioner (except KaFFPaE) using different random seeds until the resulting partition is feasable. We stopped the process after one day of computation or after one hundred repetitions yielding unbalanced partitions. The resulting partition was used for both parts of the challenge, i.e. optimizing for edge cut and optimizing for maximum communication volume. The runtime of each iteration was added if more then one iteration was needed to obtain a feasable partition. KaFFPaE was given four nodes of machine A and a time limit of eight hours for each instance. When computing partitions for the objective function maximum communication volume we altered the fitness function to this objective. This ensures that individuals having a better maximum communication volume are more often selected for a combine operation. Using this methodology KaFFPaStrong, KaFFPaEco, KaFFPaFast, KaFFPaE, Metis and Scotch were able

Solver	Points
KaFFPaFast	1372
Metis	1265
KaFFPaEco	1174
KaFFPaE	1134
KaFFPaStrong	1085
UMPa [**6**]	624
Scotch	361
Mondrian [**11**]	225

Solver	Points
KaFFPaFast	1680
KaFFPaEco	1305
KaFFPaE	1145
KaFFPaStrong	1106
UMPa [**6**]	782
Mondrian [**11**]	462

TABLE 4. Pareto challenge results including Metis and Scotch (left hand side) and original Pareto challenge results (right hand side).

to solve 136, 150, 170, 130, 146 and 110 instances respectively. The resulting points achieved in the Pareto challenge can be found in Table 4 (see [**1**] for a description on how points are computed for the challenges). Note that KaFFPaFast gained more points than KaFFPaEco, KaFFPaStrong and KaFFPaE. Since it is much faster than the other KaFFPa configurations it is almost never dominated by them and therefore scores a lot of points in this particular challenge. For some instances the partitions produced by Metis always exceeded the balance constraint by exactly one vertex. We assume that a small modification of Metis would increase the number of instances solved and most probably also the score achieved.

Quality Challenge. Our quality submission KaPa (Karlsruhe Partitioners) assembles the best solutions of the partitions obtained of our partitioners in the Pareto challenge. Furthermore, on road networks we also run Buffoon to create partitions. The resulting points achieved in the quality challenge can be found in Table 5.

Solver	Points
KaPa	1574
UMPa [**6**]	1066
Mondrian [**11**]	616

TABLE 5. Original quality challenge results.

7. Conclusion and Future Work

We presented two approaches to the graph partitioning problem, KaFFPa and KaFFPaE. KaFFPa uses novel local improvement methods and more sophisticated global search strategies to tackle the problem. KaFFPaE is an distributed evolutionary algorithm which uses KaFFPa as a base case partitioner. Due to new crossover and mutation operators as well as its scalable parallelization it is able to compute the best known partitions for many standard benchmark instances in only a *few minutes* for graphs of moderate size. We therefore believe that KaFFPaE is still helpful in the area of high performance computing. Regarding future work, we want look at more DIMACS Instances, more values of k and more values of ϵ. In particular we want to investigate at the case $\epsilon = 0$.

References

[1] Competition rules and objective functions for the 10th dimacs implementation challenge on graph partitioning and graph clustering, `http://www.cc.gatech.edu/dimacs10/data/dimacs10-rules.pdf`.

[2] R. Andersen and K.J. Lang. An algorithm for improving graph partitions. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 651–660. Society for Industrial and Applied Mathematics, 2008.
[3] David Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, `http://www.cc.gatech.edu/dimacs10/`.
[4] Una Benlic and Jin-Kao Hao. A multilevel memtetic approach for improving graph k-partitions. In *22nd Intl. Conf. Tools with Artificial Intelligence*, pages 121–128, 2010.
[5] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is NP-hard. *Inf. Process. Lett.*, 42(3):153–159, 1992.
[6] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. Umpa: A multi-objective, multi-level partitioner for communication minimization. In *10th DIMACS Impl. Challenge Workshop: Graph Partitioning and Graph Clustering*. Georgia Institute of Technology, Atlanta, GA, February 13-14, 2012.
[7] Kenneth Alan De Jong. *Evolutionary computation : a unified approach*. MIT Press, 2006.
[8] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *25th IPDPS*. IEEE Computer Society, 2011.
[9] Benjamin Doerr and Mahmoud Fouz. Asymptotically optimal randomized rumor spreading. In *ICALP (2)*, volume 6756 of *LNCS*, pages 502–513. Springer, 2011.
[10] D. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85:211–213, 2003.
[11] B. O. Fagginger Auer and R. H. Bisseling. Abusing a hypergraph partitioner for unweighted graph partitioning. In *10th DIMACS Impl. Challenge Workshop: Graph Partitioning and Graph Clustering*. Georgia Institute of Technology, Atlanta, GA, February 13-14, 2012.
[12] 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.
[13] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. *24th IEEE International Parallal and Distributed Processing Symposium*, 2010.
[14] G. Karypis, V. Kumar, Army High Performance Computing Research Center, and University of Minnesota. Parallel multilevel k-way partitioning scheme for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.
[15] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung Ro Moon. Genetic approaches for graph partitioning: a survey. In *GECCO*, pages 473–480. ACM, 2011.
[16] 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.
[17] J. Maue and P. Sanders. Engineering algorithms for approximate weighted matching. In *6th Workshop on Exp. Alg. (WEA)*, volume 4525 of *LNCS*, pages 242–255. Springer, 2007.
[18] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.
[19] V. Osipov and P. Sanders. n-Level Graph Partitioning. *18th European Symposium on Algorithms (see also arxiv preprint arXiv:1004.4024)*, 2010.
[20] F. Pellegrini. Scotch home page. `http://www.labri.fr/pelegrin/scotch`.
[21] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. *12th Workshop on Algorithm Engineering and Experimentation*, 2011.
[22] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. *19th European Symposium on Algorithms (see also arxiv preprint arXiv:1012.0006v3)*, 2011.
[23] A.J. Soper, C. Walshaw, and M. Cross. A combined evol. search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.
[24] C. Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1):325–372, 2004.
[25] C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.
[26] 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).

Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
E-mail address: {`sanders, christian.schulz`}`@kit.edu`