A New Diffusion-based Multilevel Algorithm for Computing Graph Partitions of Very High Quality*

Henning Meyerhenke Burkhard Monien Thomas Sauerwald

University of Paderborn
Department of Computer Science
Fuerstenallee 11, D-33102 Paderborn
{henningm | bm | sauerwal}@uni-paderborn.de

Abstract

Graph partitioning requires the division of a graph's vertex set into k equally sized subsets such that some objective function is optimized. For many important objective functions, e. g., the number of edges incident to different partitions, the problem is $\mathcal{NP}$-hard. Graph partitioning is an important task in many applications, so that a variety of algorithms and tools for its solution have been developed. Most state-of-the-art graph partitioning libraries use a variant of the Kernighan-Lin (KL) heuristic within a multilevel framework. While these libraries are very fast, their solutions do not always meet all requirements of the users. This includes the choice of the appropriate objective function and the shape of the computed partitions. Moreover, due to its sequential nature, the KL heuristic is not easy to parallelize. Thus, its use as a load balancer in parallel numerical applications requires complicated adaptations. That is why we have developed previously an inherently parallel algorithm, called BUBBLE-FOS/C *(Meyerhenke et al., IPDPS'06), which optimizes the partition shapes by a diffusive mechanism. Yet, it is too slow to be of real practical use, despite its high solution quality.*

In this paper, besides proving that BUBBLE-FOS/C *converges towards a local optimum, we develop a much faster method for the improvement of partitionings. It is based on a different diffusive process, which is restricted to local areas of the graph and also contains a high degree of parallelism. By coupling this new technique with* BUBBLE-FOS/C *in a multilevel framework based on two different hierarchy construction methods, we obtain our new graph partitioning heuristic* DIBAP. *Compared to* BUBBLE-FOS/C*, it shows a considerable acceleration, while retaining the positive properties of the slower algorithm.*

Experiments with popular benchmark graphs show an extremely good behavior. First, DIBAP *computes consistently better results – measured by the edge-cut and the number of boundary vertices in the summation and the maximum norm – than the state-of-the-art libraries* METIS *and* JOSTLE*. Second, with our new algorithm, we have improved the best known edge-cut values for a significant number of partitionings of six widely used benchmark graphs.*

Keywords: *Graph partitioning, load balancing heuristic.*

1. Introduction

Graph partitioning is a widespread technique in computer science, engineering, and related fields. The most common formulation of the graph partitioning problem for an undirected graph $G = (V, E)$ asks for a division of V into k pairwise disjoint subsets (*partitions*) of size at most $\lceil |V|/k \rceil$ such that the *edge-cut*, i.e., the total number of edges having their incident nodes in different subsets, is minimized. Amongst others, its applications include dynamical systems [5], VLSI circuit layout [9], and image segmentation [32]. We mainly consider its use for balancing the load in numerical simulations (e. g., fluid dynamics), which have become a classical application for parallel computers. There, our task is to compute a partitioning of the (dual) mesh derived from the domain discretization [31].

*This work is partially supported by German Research Foundation (DFG) Research Training Group GK-693 of the *Paderborn Institute for Scientific Computation* (PaSCo), by Integrated Project IST-15964 *AEOLUS* of the European Union, and by DFG Priority Programme 1307 *Algorithm Engineering*.

Despite some successes on approximation algorithms (e. g., [18]) for this $\mathcal{NP}$-hard problem, heuristics are preferred in practice for its solution. Several different algorithms have been proposed, see [31] for an overview. They can be categorized as either global or local optimizers. Spectral methods [12, 34] and space-filling curves [40] are representatives of global methods. While space-filling curves work extremely fast, they do not yield satisfying partitionings for complicated domains. Spectral algorithms have been widely used, but are relatively slow and thus have been mostly superseded by faster local improvement algorithms. Integrated into a multilevel framework, these local optimizers such as Kernighan-Lin (KL) [17] can be found in several state-of-the-art graph partitioning libraries, which we describe in more detail in Section 2.

Motivation. Implementations of multilevel KL yield good solutions in very short time, but the computed partitionings do not necessarily meet the requirements of all users: As Hendrickson has pointed out [11], the number of *boundary vertices* (vertices that have a neighbor in a different partition) models the communication volume between processors in numerical simulations more accurately than the edge-cut. Moreover, the edge-cut is a *summation norm*, while often (e. g., for parallel numerical solvers) the *maximum norm* is of much higher importance. Finally, for some applications, the *shape* of the partitions, in particular small aspect ratios [7], but also connectedness and smooth boundaries, plays a significant role. Nevertheless, most partitioning-based load balancers do not take these facts fully into account.

While the total number of boundary vertices can be minimized by hypergraph partitioning [6], an optimization of partition shapes requires additional techniques (e. g., [7, 25]), which are far from being mature. Furthermore, due to their sequential nature, the heuristic KL is difficult to parallelize. Although significant progress has been made [2, 30, 39], an inherently parallel graph partitioning algorithm for load balancing can be expected to yield better solutions, possibly also in shorter time.

These issues have led us to the development of the partitioning heuristic BUBBLE-FOS/C in previous work (cf. [21] and Section 3 of this paper). It is based on a disturbed diffusion scheme that determines how well connected two nodes are in a graph. Using this, it aims at the *optimization of the partition shapes* and results in partitionings with nearly always connected partitions that have short boundaries, good edge-cut values and aspect ratios. Moreover, it contains a high degree of natural parallelism and can be used for parallel load balancing, resulting in *low migration costs* [23]. Yet, its partly global approach makes it too slow for practical relevance. It is therefore highly desirable to develop a significantly faster algorithm retaining the good properties of BUBBLE-FOS/C.

Contribution. The contribution of this paper consists of both theoretical and practical advances in shape-optimizing graph partitioning. In order to understand BUBBLE-FOS/C better theoretically, we prove its convergence in Section 4 as our main theoretical result. The proof relies on a potential function argument and a load symmetry result for the disturbed diffusion scheme FOS/C. Due to its high running time, the excellent solution quality of BUBBLE-FOS/C could previously not be exploited for large graphs. We present in this work a much faster new diffusive method for local improvement of partitionings in Section 5. The combination of BUBBLE-FOS/C and this new diffusive method within a multilevel framework with two different hierarchy construction algorithms, called DIBAP, constitutes our main algorithmic achievement. This combined algorithm is much faster than BUBBLE-FOS/C and computes multi-way graph partitionings of very high quality on large graphs in a very reasonable amount of time. In Section 6 we show experimentally that our algorithm delivers better solutions than the state-of-the-art partitioning libraries METIS [15, 16] and JOSTLE [38] in terms of the edge-cut *and* the number of boundary vertices, both in the summation *and* in the maximum norm. Certainly notable is the fact that DIBAP also improves for six benchmark graphs a large number (more than 80 out of 144) of their best known partitionings w. r. t. the edge-cut. These six graphs are among the eight largest in a popular benchmark set [33, 37].

2. Related Work

In this section we give a short introduction to the state-of-the-art of practical general-purpose graph partitioning algorithms and libraries. General purpose means here that these algorithms and libraries only require the adjacency information about the graph and no additional problem-related information. Our focus lies on implementations included in the experimental evaluation in Section 6 and on methods with related techniques for improving partitions. For a broader overview the reader is referred to [31], for the authors' previous work on shape-optimizing graph partitioning using diffusion to Section 3.

It should also be noted that a number of metaheuristics have been used for graph partitioning recently, e. g., [1, 3, 33]. These algorithms also focus on low

edge-cuts instead of good partition shapes and some of them require a very high running time to obtain high quality results.

2.1. Graph Partitioning by Multilevel Local Improvement

Refining a given partitioning by local considerations usually yields better running times on large graphs than global approaches. The problem of how to obtain a good starting solution is overcome by the multilevel approach [13], which consists of three phases. Instead of computing a partitioning immediately for large input graphs, one computes a hierarchy of graphs $G_0, \ldots, G_l$ by recursive coarsening in the first phase. G_l is supposed to be very small in size, but similar in structure to the input graph G_0. In the second phase a very good initial solution for G_l (which is easy due to its size) is computed. Finally, in the third phase, the solution is interpolated to the next-finer graph, where it is improved using a local improvement algorithm. This process of interpolation and local improvement is repeated recursively up to G_0.

A very common local improvement algorithm is based on the method by Fiduccia and Mattheyses (FM) [9], a running time optimized version of the Kernighan-Lin heuristic (KL) [17]. The main idea of both is to exchange nodes between partitions in the order of the cost reductions possible, while maintaining balanced partition sizes. After every node has been moved once, the solution with the best gain is chosen. This is repeated several times until no further improvements are found.

State-of-the-art graph partitioning libraries such as METIS [15, 16] and JOSTLE [38] use KL/FM for local improvement and edge-contractions based on matchings for coarsening. With this combination these libraries compute solutions of a good quality very fast. However, as argued in the introduction, for some applications their solutions are not totally satisfactory.

To address the load balancing problem in parallel applications, distributed versions of these libraries [30, 39] and parallel (hyper)graph partitioners such as Zoltan [2] have been developed. This is very complex due to inherently sequential parts in KL/FM improvement, e. g., no two neighboring vertices should change their partitions simultaneously.

2.2. Diffusive Partitioning Approaches

In the area of graph clustering there exist techniques for dividing nodes into groups based on random walks. Their common idea is that a random walk stays a very long time in a dense graph region before leaving it via one of the few outgoing edges. Somewhat related to our new diffusive method is the algorithm by Harel and Koren [10], which computes separator edges iteratively based on the similarity of their incident nodes. This similarity is derived from the sum of transition probabilities of random walks with very few steps. The procedure focuses on clusters of different sizes and we do not know of any attempt to use it for the graph partitioning problem.

Very recently, Pellegrini [25] has addressed some drawbacks of the KL/FM heuristic. His approach aims at improved partition shapes, based on a diffusive mechanism used together with FM improvement. For the diffusion process the algorithm replaces whole partition regions not close to partition boundaries by one super-node. This reduces the number of diffusive operations and results in an acceptable overall speed. The implementation described is only capable of recursive bisection. As Pellegrini points out, a "full k-way diffusion algorithm is therefore required" [25, p. 202] to improve the quality for large k. In this paper we fill this gap by providing a related full k-way partitioning algorithm.

3. BUBBLE Framework and Disturbed Diffusion

In this section we describe our own previous work on graph partitioning with diffusive mechanisms. This is necessary to understand the results of this paper. In particular, we explain the partitioning algorithm BUBBLE-FOS/C, which is proved to converge in Section 4.

3.1. BUBBLE Framework

The BUBBLE framework is related to Lloyd's geometric k-means clustering algorithm [19] and transfers its ideas to graphs. In the first step of BUBBLE, initial partition representatives (centers) are chosen, one for each partition. All remaining vertices are assigned to their closest center vertex w. r. t. some distance measure. After that each partition computes its new center for the next iteration. The two operations *assigning vertices to partitions* and *computing new centers* can be repeated alternately a fixed number of times or until a stable state is reached. The actual implementation of the framework operations can differ significantly, see [21] for how it has evolved. Its current version overcomes previous problems and optimizes the partitions' shapes

implicitly, as indicated in Section 3.3 after the definition of the employed diffusive distance measure in the next section.

3.2. Disturbed Diffusion FOS/C

Diffusive processes can be used to model important transport phenomena or as a tool for iterative local load balancing in parallel computations. The FOS/C algorithm is a modification of the first order diffusion scheme (FOS) [4] from the latter domain. This modification results in a non-balanced load distribution in the steady state to represent distances between nodes reflecting their connectedness.

Definition 1. *[21] Let $[x]_v$ denote the component of the vector x corresponding to node v. Given a connected and undirected graph $G = (V, E)$ free of self-loops with n nodes and m edges, a set of source nodes $S \subset V$, and constants $0 < \alpha \leq (\mathrm{maxdeg}(G) + 1)^{-1}$ and $\delta > 0$.*[1] *Let the initial load vector $w^{(0)}$ and the drain vector d (which is responsible for the disturbance) be defined as:*

$$[w^{(0)}]_v = \begin{cases} \frac{n}{|S|} & v \in S \\ 0 & \textit{otherwise} \end{cases} \quad \textit{and}$$

$$[d]_v = \begin{cases} \frac{\delta n}{|S|} - \delta & v \in S \\ -\delta & \textit{otherwise} \end{cases}$$

Then, the FOS/C iteration in time-step $t \geq 1$ is defined as $w^{(t)} = \mathbf{M}w^{(t-1)} + d$, where $\mathbf{M} = \mathbf{I} - \alpha\mathbf{L}$ is the doubly-stochastic diffusion matrix [4] and $\mathbf{L}$ the Laplacian matrix of G.

Note that the extension of FOS/C to edge-weighted graphs is straightforward.

Theorem 1. *[21] The FOS/C iteration reaches a steady state for any $d \perp (1, \ldots, 1)^T$. This steady state can be computed by solving the linear system $\mathbf{L}w = d$ and normalizing w s. t. $\sum_{v \in V}[w]_v = n$.*

Definition 2. *If $|S| = 1$ ($|S| > 1$), we call the FOS/C iteration to the steady state a* single-source (multiple-source) FOS/C procedure. *Also, let $[w^{(t)}]_v^u$ ($[w]_v^u$) denote the load on node v in time-step t (in the steady state) of a single-source FOS/C procedure with node u as source.*

Remark 1. *Note: $[w]_v^u = \lim_{t\to\infty}([\mathbf{M}^t w^{(0)}]_v^u + n\delta(\sum_{l=0}^{t-1}\mathbf{M}_{vu}^l) - t\delta)$ [22] and $\mathbf{M}_{vu}^l$ is the probability of a random walk starting at v to be on u after l steps. Since $[\mathbf{M}^t w^{(0)}]_v^u$ converges towards the balanced load distribution, the important part of an FOS/C load in the steady state can be seen as the sum of transition probabilities of random walks with increasing lengths.*

[1] Here, the maximum degree of G is defined as $\mathrm{maxdeg}(G) := \max_{u \in V} \deg(u)$.

```
Algorithm Bubble-FOS/C(G, k) → Π
01  Z = InitialCenters(G, k)      /* Arbitrary initial centers */
02  for τ = 1, 2, ... until convergence
      /* AssignPartition */
03    parallel for each partition π_c
04      Initialize d_c (S = {z_c}),
          solve and normalize L w_c = d_c
05      for each node v ∈ π_c
06        Π(v) = p : [w_p]_v ≥ [w_q]_v ∀q ∈ {1, ..., k}
      /* ComputeCenters */
07    parallel for each partition π_c
08      Initialize d_c (S = π_c) and solve L w_c = d_c
09      z_c = argmax_{v∈π_c} [w_c]_v
10  return Π
```

Figure 1. Sketch of the main BUBBLE-FOS/C **algorithm.**

3.3. BUBBLE-FOS/C with Algebraic Multigrid

BUBBLE-FOS/C implements the operations of the BUBBLE framework with FOS/C procedures, single-source ones for `AssignPartition` and multiple-source ones for `ComputeCenters`. Its outline is shown in Figure 1, where $\Pi = \{\pi_1, \ldots, \pi_k\}$ denotes the set of partitions and $Z = \{z_1, \ldots, z_k\}$ the set of the corresponding center nodes. First, the algorithm determines pairwise disjoint initial centers (line 1), which can be done in an arbitrary manner. After that, with the new centers at hand, the main loop is executed. It determines in alternating calls a new partitioning (`AssignPartition`, lines 3-6) and new centers (`ComputeCenters`, lines 7-9). The loop can be iterated until convergence is reached or, if running time is important, a constant number of times.

This approach yields very good partitions, because FOS/C can distinguish sparsely connected from densely connected regions by its random walk notion pointed out above. Hence, one usually obtains partition centers in dense regions and boundaries tend to be in sparse ones (as desired). Moreover, since the isolines of the FOS/C load in the steady state tend to have a circular shape, the final partitions are very compact and have short boundaries. Note that the rare case of ties in the load values can be handled easily and that additional

operations not originating from the BUBBLE framework can be integrated into BUBBLE-FOS/C. The latter includes balancing operations [23], which are omitted here for ease of presentation.

Most work performed by BUBBLE-FOS/C consists in solving linear systems. It is therefore necessary to employ a very efficient solver. Multigrid methods [36] are among the fastest algorithms for preconditioning and solving linear systems of equations arising from partial differential equations. Algebraic multigrid (AMG) [35] is an extension to cases where no problem-related information such as geometry is available. It constructs a multilevel hierarchy based on weighted interpolation with a carefully chosen set of nodes for the coarser level. The actual solution process is performed by iterative algorithms traversing this hierarchy, e. g., V-cycles or FMV-cycles [36, p. 46ff.]. We use AMG as a linear solver, since the same system matrix $\mathbf{L}$ is used repeatedly, so that the hierarchy construction is amortized. Furthermore, as an AMG hierarchy is also a sequence of coarser graphs retaining the structure of the original one, we use it in our BUBBLE-FOS/C implementation for providing a multilevel hierarchy (instead of the standard matching approach).

4. Convergence Results on BUBBLE-FOS/C

In this section we settle the question if the algorithm BUBBLE-FOS/C depicted in Figure 1 converges, in the affirmative. The proof relies on a load symmetry property and a potential function. These results are supposed to provide insights for future work on faster disturbed diffusion schemes and on how BUBBLE-FOS/C can be used for related problems such as graph clustering, where the subdomain sizes do not need to be balanced.

Definition 3. *Let the function $F(\Pi, Z, \tau)$ for time-step τ be defined as follows:*

$$F(\Pi, Z, \tau) := \sum_{c=1}^{k} \sum_{v \in \pi_c(\tau)} [w]_v^{z_c(\tau)},$$

where $\pi_c(\tau)$ and $z_c(\tau)$ denote the c-th partition and center node in iteration τ, respectively.

Our objective is to maximize F. It is obvious that F has a finite upper bound on any finite graph, so that it is sufficient to show that the operations `AssignPartition` and `ComputeCenters` each maximize the value of F w. r. t. their input. This is clearly the case for `AssignPartition`, since nodes are assigned to partitions sending the highest amount of load. Yet, it is not obvious for `ComputeCenters`, so that we require first the following result on the load symmetry between two single-source FOS/C procedures.

Lemma 1. *For any graph $G = (V, E)$ and two arbitrary nodes $u, v \in V$ holds $[w]_v^u = [w]_u^v$.*

Proof. Consider an FOS/C procedure with source node u. Recall that its drain vector d is defined as $d = (-\delta, \ldots, -\delta, \delta(n-1), -\delta, \ldots, -\delta)^T$, where $\delta(n-1)$ appears in row u. The FOS/C iteration scheme in timestep $t+1$ for node v and source u can be written as [22]:

$$\begin{aligned} [w^{(t+1)}]_v^u &= [\mathbf{M}^{t+1} w^{(0)}]_v^u \\ &\quad + [(\mathbf{I} + \mathbf{M}^1 + \ldots + \mathbf{M}^t) d]_v^u \\ &= [\mathbf{M}^{t+1} w^{(0)}]_v^u + n\delta \sum_{l=0}^{t} \mathbf{M}_{v,u}^l \\ &\quad - (t+1)\delta. \end{aligned}$$

Observe that $\mathbf{M}^{t+1} w^{(0)}$ converges towards $\overline{w} = (1, \ldots, 1)^T$, the balanced load distribution [4], even in the edge-weighted case [8]. Hence, we obtain:

$$[w]_u^v - [w]_v^u = n\delta \left(\sum_{l=0}^{t} \mathbf{M}_{u,v}^l - \mathbf{M}_{v,u}^l \right).$$

Since $\mathbf{M}$ and therefore also its powers are symmetric, all summands vanish. □

The generality of this load symmetry is somewhat surprising, because one would not expect such a property in graphs without any symmetry. In that regard, it is of independent interest for the disturbed diffusion scheme FOS/C. Here, it helps us to prove the next crucial lemma.

Lemma 2. *The output of `ComputeCenters` maximizes the value of F for a given Π.*

Proof. Let Π be the current partitioning. `ComputeCenters` solves for each partition $\pi_c, c \in \{1, \ldots, k\}$, a multiple-source FOS/C procedure, where the whole respective partition acts as source. Consider one of these partitions π_c and its multiple-source FOS/C procedure, which computes w in $\mathbf{L}w = d$ with its respective drain vector d. Our aim is to split this procedure into subprocedures that solve $\mathbf{L}w_i = d_i, i \in \pi_c$, and that satisfy $\sum_{i \in \pi_c} d_i = d$. Such a splitting $\mathbf{L}(w_1 + w_2 + \cdots + w_{|\pi_c|}) = d_1 + d_2 + \cdots + d_{|\pi_c|}$

indeed exists. Note that each subprocedure $\mathbf{L}w_i = d_i$ corresponds to a single-source procedure ($|S| = 1$), where the drain vector is scaled by $\frac{1}{|\pi_c|}$ (cf. Definition 1):

$$[d_i]_v = \begin{cases} \frac{\delta n}{|\pi_c|} - \frac{\delta}{|\pi_c|} & v \in \pi_c,\ v \text{ source of subproc. } i \\ -\frac{\delta}{|\pi_c|} & \text{otherwise} \end{cases}$$

It is easy to verify that $\sum_{i \in \pi_c} d_i = d$ and $d_i \perp (1, \ldots, 1)^T$ hold, so that each subprocedure has a solution. Due to this and the linearity of $\mathbf{L}$, we also have $\sum_{i \in \pi_c} w_i = w$. Recall that the new center of partition π_c is the node with the highest load of the considered multiple-source FOS/C procedure. From the above it follows that this is the node u for which $[w]_u = \sum_{i \in \pi_c} [w]_u^i$ is maximal. Due to Lemma 1 we have $\sum_{i \in \pi_c} [w]_u^i = \sum_{i \in \pi_c} [w]_i^u$, so that the new center z_c is the node u for which the most load remains within partition π_c in a single-source FOS/C procedure. Consequently, the contribution $\sum_{v \in \pi_c} [w]_v^{z_c}$ of each π_c to F is maximized. □

Proposition 1. *Consider the load vector w in the steady state of FOS/C. The maximum load value in w belongs to the set of source nodes S. Consequently, after selecting k pairwise disjoint initial center nodes, there are always exactly k different center nodes and exactly k partitions during the execution of* BUBBLE-FOS/C.

Proof. The first statement follows from the fact that the steady state of FOS/C is equivalent to a $\|\cdot\|_2$-minimal-flow problem where the source nodes send load to the remaining nodes (see [22, p. 431]) and must therefore have a higher load.

For the second one "$\leq$" is obvious, so that it remains to show that there are at least k different center nodes and partitions in each iteration. The initial placement of centers can easily ensure that the former is true. In any case the centers determined by `ComputeCenters` belong to their own partition and must therefore be different. Also, `AssignPartition` keeps each center in its current partition: Consider two arbitrary, but distinct centers z_i and z_j. Due to Lemma 1 we know that $[w]_{z_i}^{z_j} = [w]_{z_j}^{z_i}$. As $[w]_{z_i}^{z_i} > [w]_{z_j}^{z_i}$, we obtain $[w]_{z_i}^{z_i} > [w]_{z_i}^{z_j}$. Therefore, all center nodes remain in their partition and the claim follows. □

The main theorem follows now directly from the results above.

Theorem 2. BUBBLE-FOS/C *converges and returns a locally optimal k-partitioning.*

5. Accelerating Shape-optimizing Partitioning

Our previous work on shape-optimizing graph partitioning [21, 23] has already indicated that this approach is able to compute high-quality partitionings meeting the requirements mentioned in the introduction. The main reason for its very high running time is the repeated solution of linear systems on the whole graph (or at least on an approximation of the whole graph, as in [24]). Yet, once a reasonably good solution has been found, alterations during an improvement step take place mostly at the partition boundaries. That is why we introduce in the following a local approach considering only these boundary regions. Our idea is to use the high-quality, but slow algorithm BUBBLE-FOS/C on the coarse levels of a multilevel hierarchy and a faster local scheme on its finer levels.

5.1. A New Local Improvement Method: TRUNCCONS

As a mixture of `AssignPartition` and `ComputeCenters`, the `Consolidation` operation is used to determine a new partitioning from a given one. As illustrated in Figure 2 with an example of a path graph and $k = 3$, one performs the following independently for each partition π_c: First, the source set S is initialized with π_c. The nodes of π_c receive an equal amount of initial load $n/|S|$, while the other nodes' initial load is set to 0. Then, a diffusive method (e. g., FOS/C, but this should be avoided for large graphs, because of its high running time) is used to distribute this load within the graph. To restrict the computational effort to areas close to the partition boundaries, we use a small number ψ of FOS [4] iterations for this.

The final load of a node v for π_c is then just $[w_c^{(\psi)}]_v = [\mathbf{M}^\psi \cdot w_c^{(0)}]_v$, where $\mathbf{M}$ and $w^{(0)}$ are like in Definition 1. This can be computed by iterative load exchanges for $1 \leq t \leq \psi$:

$$[w_c^{(t)}]_v = [w_c^{(t-1)}]_v - \alpha \sum_{\{u,v\} \in E} ([w_c^{(t-1)}]_v - [w_c^{(t-1)}]_u)$$

After the load is distributed this way for all k partitions, we assign each node v to the partition it has obtained the highest load from. This completes one `Consolidation` operation, which can be repeated several times to facilitate sufficiently large movements of the partitions. We denote the number of repetitions by Λ and call the whole method with this particular dif-

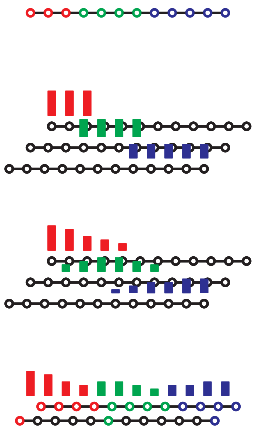


Figure 2. Schematic view of one `Consolidation`.

Algorithm TRUNCCONS($\mathbf{M}$, k, Π, Λ, ψ) $\rightarrow \Pi$

```
for τ = 1 to Λ
  parallel for each partition π_c
    S = π_c; w_c = (0, ..., 0)^T /* initial load */
    for each v ∈ S /* initial load */
      [w_c]_v = n/|S|
    for t = 1 to ψ /* FOS iterations */
      w_c = M · w_c
    /* after synchronization: update Π */
    for each v ∈ π_c
      Π(v) = argmax_{1≤c'≤k} [w_c']_v
return Π
```

Figure 3. Algorithmic sketch of TRUNCCONS.

fusive process TRUNCCONS (*truncated diffusion consolidations*), see Figure 3. This new approach makes Schamberger's ideas [28] robust, practicable, and fast. Moreover, although showing some differences, it can be viewed as a k-way extension of Pellegrini's work [25] mentioned in Section 2.2.

To understand why TRUNCCONS works well, consider the following analogy. Recall that the stochastic diffusion matrix $\mathbf{M}$ can be seen as the transition matrix of a random walk. For each node $v \in \pi_c$ we have one random walker starting on v. Then, the final load on node u is proportional to the sum of the probabilities for each random walker to reach u after ψ steps. Since random walks need relatively long to leave dense regions, each node should be assigned to the partition sending the highest load. With this partition the node is most densely connected, i. e., connected best via a large number of short paths.

Since an actual load exchange happens only at the partition boundary, not all nodes have to take part in this process. Instead, one keeps track of *active nodes*. During the course of the iteration, these nodes are the ones either directly at the partition boundary or whose load value has already differed from the initial one. By neglecting inactive nodes, the diffusive improvement process is restricted to local areas close to the partition boundaries, which greatly reduces the computational costs. The choice of the initial load takes partition sizes into account and therefore improves the balance. For cases where this is not sufficient, additional balancing heuristics have been integrated, see below.

5.2. The New Algorithm DIBAP: Combining BUBBLE-FOS/C and TRUNCCONS

Now that we have a slow, but high-quality partitioner and a fast local improvement algorithm, we combine them to obtain an efficient multilevel graph partitioning algorithm that we call DIBAP (*Diffusion-based Partitioning*). The fine levels of its multilevel hierarchy are constructed by approximate maximum weight matchings [27]. Once the graphs are sufficiently small, we switch the construction mechanism to the more expensive AMG coarsening. This is advantageous, because we use BUBBLE-FOS/C as the improvement strategy on the coarse levels and employ AMG to solve the occurring linear systems. That is why such a hierarchy needs to be built anyway. On the finer parts of the hierarchy, the faster TRUNCCONS is used for local improvement. As this does not involve linear systems, AMG is not required, so that it is much cheaper to use a matching hierarchy instead.

Note that it is questionable if TRUNCCONS can be adapted to partition graphs from scratch (instead of local improvement) with an equally high quality as BUBBLE-FOS/C. Preliminary experiments indicate that the partition shapes and other important properties of the solutions suffer in quality if TRUNCCONS is used too early or even exclusively. Apparently, BUBBLE-FOS/C is required to obtain a good starting solution on a sufficiently large hierarchy level.

Initial Centers. Instead of selecting all initial center vertices randomly or to coarsen the graph until the number of nodes is k, we employ the following procedure to distribute the centers. It takes the graph structure into account by choosing only one center randomly. After that new centers are selected one after another, where the newest one is chosen farthest away (i. e., with mini-

mum FOS/C loads: $\text{argmin}_v\{\sum_{z\in Z}[w]_v^z\}$) from all already chosen centers in the set Z. On a very coarse graph of a multilevel hierarchy, this is inexpensive and can even be repeated to choose the best set of centers from a sample. By this repetition, outliers with a rather poor solution quality hardly occur, as indicated by preliminary experiments.

Repartitioning. For cases where a partitioning is part of the input and needs to be repartitioned (e. g., to restore its balance), we propose the following procedure. The initial partitioning is sent down the multilevel hierarchy, where the maximum hierarchy depth depends on the input quality. If the input is not too bad, BUBBLE-FOS/C does not need to be used and we can solely use the faster TRUNCCONS with a matching hierarchy. Preliminary tests have shown that repartitioning this way is about three times faster than partitioning from scratch.

Implementation Details. The `Consolidation` operation can also be used with FOS/C in lines 6 and 7 of TRUNCCONS instead of a few FOS iterations. This again global operation can be optionally integrated into BUBBLE-FOS/C after an `AssignPartition` operation. To ensure that the balance constraints are definitely met, explicit balancing procedures are integrated into the improvement process. They are mostly based on our previous implementation [23], but are, where necessary, slightly adapted to TRUNCCONS. This also holds for the operation that improves partitions by moving vertices (which lie directly on their boundary) once if this results in fewer cut-edges.

There are several important parameters controlling the quality and run-time of DIBAP; their values have been determined experimentally. Multilevel hierarchy levels with graphs of more than 8000 nodes are coarsened by matchings and improved with TRUNCCONS. Once they are smaller than this threshold, we switch to BUBBLE-FOS/C with AMG coarsening. The latter is performed by our implementation without using any scientific libraries. Details about the implementation of BUBBLE-FOS/C and the AMG hierarchy construction can be found in our previous work [21]. It should be noted, however, that we have made some changes to our AMG implementation, for example a different interpolation scheme. A detailed description of these changes is outside the scope of this paper. Keeping track of active nodes is currently done with an array that stores for each node its status. This could be improved by a faster data structure considering only the active nodes.

In the experiments presented in this paper, BUBBLE-FOS/C has performed one iteration of `ComputeCenters` and `AssignPartition`, followed by three `Consolidations` with FOS/C as distance measure. The AMG coarsening is stopped when the graph has at most $24k$ nodes to compute an initial set of centers. The most important parameters for the finer parts of the multilevel hierarchy are Λ (the number of `Consolidations`) and ψ (the number of FOS iterations). As most other parameters, they can be specified by the user, whose choice should consider the time-quality trade-off. Two possible choices ($\Lambda = 6$ / $\psi = 9$ and 12/18) are used in the subsequent experiments.

Table 1. Benchmark graphs.

Graph	|V|	|E|
fe_tooth	78,136	452,591
rotor	99,617	662,431
598a	110,971	741,934
ocean	143,437	409,593
144	144,649	1,074,393
wave	156,317	1,059,331
m14b	214,765	1,679,018
auto	448,695	3,314,611

6. Experimental Results

In this section we present some of our experiments with the new DIBAP C/C++ implementation. After comparing it to METIS and JOSTLE, two state-of-the-art partitioning tools, we show that DIBAP performs extremely well on six popular benchmark graphs. For these graphs, DIBAP has computed a large number of partitionings with the best known edge-cut values, improving records derived from numerous algorithms.

6.1. Comparison of Partitioning Quality

Settings. The experiments have been conducted on a desktop computer equipped with an Intel Core 2 Duo 6600 CPU and 2 GB RAM. To have a fair comparison to the sequential libraries METIS and JOSTLE, the use of threads is completely deactivated in our program, so that the second core of the CPU is not used. The operating system is Linux (openSUSE 10.2, Kernel 2.6.18) and the main code has been compiled with GCC 4.1, using level 2 optimization. We distinguish DIBAP-short ($\Lambda = 6$, $\psi = 9$) and DIBAP-long ($\Lambda = 12$, $\psi = 18$) to determine how the quality is affected by different settings in the new method.

For the further presentation we utilize the eight widely used benchmark graphs shown in Table 1. We

have chosen them, because they are publicly available from Chris Walshaw's well-known graph partitioning archive [33, 37] and are the eight largest therein w. r. t. the number of nodes. More importantly, they represent the general trends in our experiments and constitute a good sample, since they model large enough problems from 3-dimensional numerical simulations (e. g., according to [14], *598a* and *m14b* are meshes of submarines and *auto* of a GM Saturn).

We evaluate our algorithm DIBAP against METIS (more precisely KMETIS [2] [16], which implements direct k-way KL/FM improvement) and JOSTLE [38], because these two are probably the most popular sequential graph partitioners due to their speed and adequate quality. (Future work includes comparisons to their parallel counterparts, the library SCOTCH [26], and the load balancer Zoltan [2].) Both are used with default settings so that their optimization objective is the edge-cut. We allow all programs to generate partitionings with at most 3% imbalance, i. e., whose largest partition is at most 3% larger than the average partition size. To specify this is important, because a higher imbalance can result in better partitionings.

The order in which the vertices of the graph are stored has a great impact on the partitioning quality of KL/FM partitioners, as it affects the order of the node exchanges. Hence, METIS and JOSTLE are run three times on the same graph, but with a randomly permuted vertex set. For DIBAP the order of the vertices is insignificant. This is because the diffusive partitioning operations are only affected by it in the rare case of ties in the load values. That is why we perform three runs on the same graph with different random seeds, resulting in different choices for the first center vertex.

Metrics and Norms. How to measure the quality of a partitioning, depends mostly on the application. Besides the edge-cut, we also include the number of boundary nodes, since this measures communication costs in parallel numerical simulations more accurately [11]. The measures for a partition p are defined as:

$$\begin{aligned} ext(p) &:= |\{e = \{u, v\} \in E : \Pi(u) = p \wedge \\ &\quad \Pi(v) \neq p\}| \text{ (external or cut-edges)}, \\ bnd(p) &:= |\{v \in V : \Pi(v) = p \wedge \exists \{u, v\} \in E : \\ &\quad \Pi(u) \neq p\}| \text{ (boundary nodes)}. \end{aligned}$$

Note that the edge-cut is the summation norm of the external edges divided by 2 to account for counting each edge twice. For some applications not only the summation norm ℓ_1 of ext and bnd over all k partitions has to be considered, but also the maximum norm ℓ_∞. This is particularly the case for parallel simulations, where all processors have to wait for the one computing longest. That is why we record ext and bnd in both norms.

Results. Tables 2 (ℓ_1-norm) and 3 (ℓ_∞-norm) show the results in a condensed form. They provide the average values obtained by the different programs in the three runs on the benchmark set for a common variety of partition numbers k. For an easier judgment we present for KMETIS the averaged actual values obtained, while the values of the other implementations are given *relative* to the respective ones of KMETIS. Best values are printed in bold font.

Table 2 shows that, in the summation norm, DIBAP-short improves on KMETIS in all cases and on JOSTLE in all cases but one (EC for $k = 32$). Better than this, DIBAP-long achieves the best values for all k and both measures. An even larger improvement can be observed for the maximum norm in Table 3. In this norm DIBAP-short is always superior to KMETIS and JOSTLE. Surprisingly, it is occasionally slightly better than DIBAP-long, which achieves the remaining and total best values.

The average improvement to KMETIS w. r. t. the number of boundary nodes in the maximum norm – which can be considered a more accurate measure for communication in parallel numerical solvers than the edge-cut – is 7.3% for DIBAP-short and 8.7% for DIBAP-long. The gain on JOSTLE is even approximately 12-13%. Furthermore, the number of disconnected partitions is much smaller for DIBAP-short (7) and DIBAP-long (5) than for KMETIS (11) and JOSTLE (25). It is part of future work to explore when DIBAP produces disconnected partitions and how to avoid this.

To provide the reader with a visual impression on how DIBAP's results differ from those of METIS and JOSTLE, we include a 12-partitioning of the 2D graph t60k (also available from Walshaw's archive; our benchmark set contains only 3D graphs), see Figure 4. The partitioning computed by DIBAP-long has not only fewer cut-edges and boundary nodes in both norms than the other libraries. Its partition boundaries also appear to be smoother and the subdomains have a smaller maximum *diameter* (165, compared to 253 (KMETIS) and 179 (JOSTLE)), a metric that can be used to measure their compactness and shape.

The partitionings shown in Figure 5 confirm the previous observation. While the maximum diameter of DIBAP-long and of JOSTLE are close together,

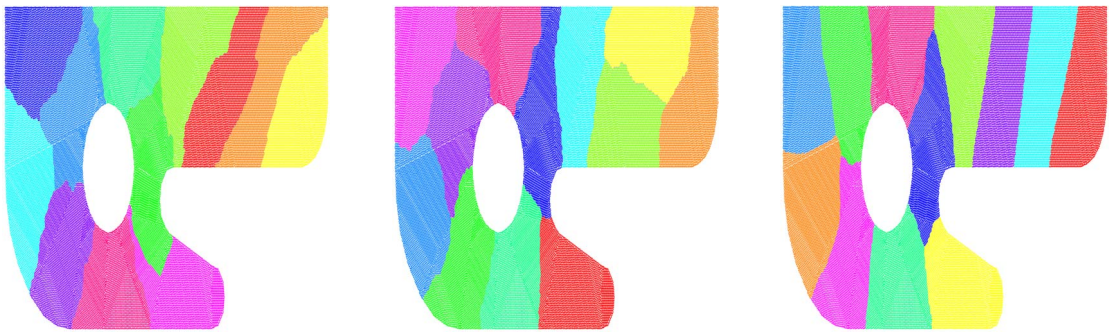
[2] The variant of METIS which yields shorter boundaries than KMETIS is not chosen, because its results are still worse than those of DIBAP regarding boundary length and they show much higher edge-cut values than KMETIS.

Table 2. Average edge-cut (EC) and number of boundary nodes (BN) for three randomized runs on the eight benchmark graphs in the *summation norm* ℓ_1 (EC, the edge-cut, denotes half the l_1-norm value); the values for JOSTLE, DIBAP-short, and DIBAP-long are relative to the respective value obtained by KMETIS.

	KMETIS		JOSTLE		DIBAP-short		DIBAP-long	
k	EC	BN	EC (rel.)	BN (rel.)	EC (rel.)	BN (rel.)	EC (rel.)	BN (rel.)
8	24651.5	13344.6	0.998	1.002	0.976	0.955	**0.948**	**0.928**
12	32406.6	17418.3	1.011	1.023	0.967	0.958	**0.945**	**0.931**
16	39122.3	20975.3	0.968	0.980	0.955	0.942	**0.930**	**0.914**
20	44854.0	23870.3	0.981	0.997	0.964	0.953	**0.935**	**0.921**
32	58481.5	30982.8	0.963	0.974	0.967	0.944	**0.937**	**0.914**
avg. (rel.)	1.0	1.0	0.984	0.995	0.966	0.951	**0.939**	**0.922**

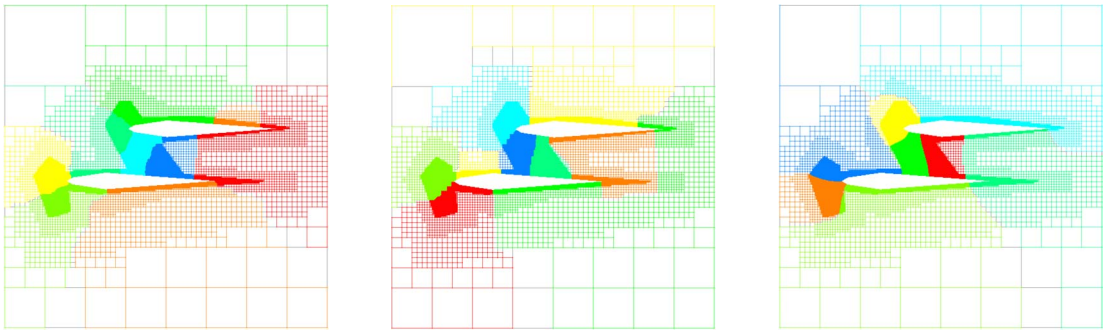
Table 3. Average number of external edges (EE) and of boundary nodes (BN) for three randomized runs on the eight benchmark graphs in the *maximum norm* ℓ_∞; the values for JOSTLE, DIBAP-short, and DIBAP-long are relative to the respective value obtained by KMETIS.

	KMETIS		JOSTLE		DIBAP-short		DIBAP-long	
k	EE	BN	EE (rel.)	BN (rel.)	EE (rel.)	BN (rel.)	EE (rel.)	BN (rel.)
8	8523.0	2308.8	1.068	1.058	0.950	0.932	**0.929**	**0.910**
12	7551.3	2039.0	1.071	1.084	0.945	0.947	**0.935**	**0.932**
16	7079.4	1909.2	1.015	1.008	0.933	0.913	**0.905**	**0.883**
20	6484.8	1721.5	1.027	1.043	**0.935**	0.920	0.938	**0.918**
32	5346.4	1406.9	1.085	1.067	**0.954**	**0.921**	0.958	0.924
avg. (rel.)	1.0	1.0	1.053	1.052	0.944	0.927	**0.933**	**0.913**



(a) KMETIS (b) JOSTLE (c) DIBAP-long

Figure 4. Partitionings of the graph t60k ($|V| = 60005$, $|E| = 89440$) into $k = 12$ subdomains with the three partitioners.



(a) KMETIS (b) JOSTLE (c) DIBAP-long

Figure 5. Partitionings of biplane9 ($|V| = 21701$, $|E| = 42038$) into $k = 8$ subdomains with the three partitioners.

the one of KMETIS is clearly worse. Also note the again smoother boundaries produced with DIBAP-long. Moreover, both other libraries generate a partition with two large disconnected node sets.

6.2. Running Times

The average *sequential* running times required by the programs to partition the "average graph" of the benchmark set are given in Table 4. Clearly, KMETIS is the fastest and JOSTLE a factor of roughly 2.5 slower. Compared to this, the running times of DIBAP-short and DIBAP-long are significantly higher. This is in particular true for larger k, which is mainly due to the fact that – in contrast to KMETIS and JOSTLE – DIBAP scales nearly linearly with k, i. e., doubling k results in a nearly doubled running time. A remedy of this problem is part of future work.

Table 4. Average sequential running times (seconds).

k	KMETIS	JOSTLE	DIBAP-sh.	DIBAP-lo.
8	0.32	0.68	8.19	38.06
12	0.32	0.75	11.51	53.96
16	0.33	0.81	14.84	69.67
20	0.34	0.87	17.93	84.78
32	0.36	0.99	26.28	126.27

Nonetheless, DIBAP constitutes a vast improvement over previous implementations that use only BUBBLE-FOS/C for partitioning ([21], [29, p. 112]). The acceleration factor lies between one and two orders of magnitude (increasing with the graph size).

Parallel DIBAP. On a closer look the absolute sequential running times of DIBAP are already quite satisfactory (a few seconds to a few minutes for the benchmark graphs), even if higher than those of the established libraries. A relatively simple way to further improve our program's performance is multithreading to utilize the second core of our test machine. The use of POSIX threads for the three most time-consuming tasks (AMG hierarchy construction, solving linear systems for BUBBLE-FOS/C, and the FOS calculations within TRUNCCONS) yields a speedup of about 1.6 for most benchmark graphs on our dual-core test machine. This corresponds to an efficiency of 0.8, which is acceptable considering the thread overhead and our program's sequential parts. For the two largest graphs, the speedup becomes slightly slower (about 1.5) due to more conflicts in the shared cache during TRUNCCONS.

Among other improvements, we plan for an even higher exploitation of the algorithm's natural parallelism. This includes a distributed-memory parallelization and the use of general purpose graphics hardware for the simple diffusive operations within TRUNCCONS. If one assumes a parallel load balancing scenario with k processors for k partitions, one may divide the sequential running times of DIBAP by $k \cdot e$ (where $0 < e \leq 1$ denotes the efficiency of the parallel program). Hence, its parallel running time on k processors can be expected to be at most a few seconds, which is certainly acceptable.

6.3. Best Known Edge-Cut Results

Walshaw's benchmark archive also collects the best known partitionings for each of the 34 graphs con-

tained therein, i. e., partitionings with the lowest edge-cut. Currently, results of more than 20 algorithms are considered. Many of them are significantly more time-consuming than METIS and JOSTLE as used in our experiments above.

With each graph 24 partitionings are recorded, one for six different numbers of partitions ($k \in \{2, 4, 8, 16, 32, 64\}$) in four different imbalance settings ($0\%, 1\%, 3\%, 5\%$). Using DIBAP (in many cases DIBAP-long, but also slightly different parameter settings) we have been able to improve more than 80 of these currently best known edge-cut values for six of the eight largest graphs in the archive. The complete list of improvements with the actual edge-cut values and the corresponding partitioning files are available at the respective website of our project [20] and of Walshaw's archive [37].

7. Conclusions

In this paper we have developed the new heuristic algorithm DIBAP for multilevel graph partitioning. Based on an accelerated diffusion-based local improvement procedure, it attains a very high quality on widely used benchmark graphs: For six of the eight largest graphs of a well-known benchmark set, DIBAP improves the best known edge-cut values in more than 80 (out of 144) settings. Additionally, the very high quality of our new algorithm has been verified in extensive experiments, which demonstrate that DIBAP delivers better partitionings than METIS and JOSTLE – two state-of-the-art sequential partitioning libraries using the KL heuristic. These results show that diffusive shape optimization is a successful approach for providing partitionings of superior quality and very promising to overcome the drawbacks of traditional KL-based algorithms. It should therefore be explored further, both in theory and in practice.

Future Work. To improve the speed of DIBAP, a more sophisticated mechanism for keeping track of active nodes and a remedy for the nearly linear dependence of the run-time on k are of importance. A future MPI parallelization and an implementation of TRUNCCONS on very fast general purpose graphics hardware can be expected to exploit our algorithm's inherent parallelism better and thereby accelerate it significantly in practice. Moreover, it would be interesting to examine how DIBAP or just TRUNCCONS act as a load balancer compared to related libraries. Theoretically, starting from our convergence results, it would be interesting to obtain more knowledge on the relation of the BUBBLE framework and disturbed diffusion schemes. Of particular concern is the behavior of TRUNCCONS and how to guarantee connected partitions.

Acknowledgments. We would like to thank the anonymous referees for their comments and suggestions. Moreover, we would like to thank Stefan Schamberger for the many fruitful discussions we had on this topic.

References

[1] C.-E. Bichot. A new method, the fusion fission, for the relaxed k-way graph partitioning problem, and comparisons with some multilevel algorithms. *Journal of Mathematical Modelling and Algorithms*, 2007. Accepted for publication: 1st December 2006, published online: 16th March 2007.

[2] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. 21st Intl. Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007.

[3] C. Chevalier and F. Pellegrini. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In *Proc. Intl. Euro-Par Conf. (Euro-Par'06)*, volume 4128 of *LNCS*, pages 243–252. Springer, 2006.

[4] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Parallel and Distributed Computing*, 7:279–301, 1989.

[5] M. Dellnitz, M. Hessel-von Molo, P. Metzner, R. Preis, and C. Schütte. Graph algorithms for dynamical systems. In *Modeling and Simulation of Multiscale Problems*, pages 619–646. Springer, 2006.

[6] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. 20th Intl. Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[7] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *J. Parallel Computing*, 26:1555–1581, 2000.

[8] R. Elsässer, B. Monien, and R. Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systems*, 35:305–320, 2002.

[9] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th conference on Design automation (DAC'82)*, pages 175–181. IEEE Press, 1982.

[10] D. Harel and Y. Koren. On clustering using random walks. In *Proc. 21st Foundations of Software Technology and Theoretical Computer Science (FSTTCS'01)*, volume 2245 of *LNCS*, pages 18–41. Springer, 2001.

[11] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Proc. Irregular'98*, volume 1457 of *LNCS*, pages 218–225. Springer, 1998.

[12] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16(2):452–469, 1995.

[13] B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. In *Proc. ACM/IEEE Conf. Supercomputing (CDROM)*, page 28, 1995.

[14] S. Huang, E. Aubanel, and V. C. Bhavsar. PaGrid: A mesh partitioner for computational grids. *J. Grid Comput.*, 4(1):71–88, 2006.

[15] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstrctured Graphs, Partitioning Meshes, [...], Version 4.0*, 1998.

[16] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[17] B. W. Kernighan and S. Lin. An efficient heuristic for partitioning graphs. *Bell Systems Technical Journal*, 49:291–308, 1970.

[18] R. Khandekar, S. Rao, and U. Vazirani. Graph partitioning using single commodity flows. In *Proc. 38th Symp. on Theory of Computing (STOC'06)*, pages 385–390. ACM, 2006.

[19] S. P. Lloyd. Least squares quantization in PCM. *IEEE Trans. on Information Theory*, 28(2):129–136, 1982.

[20] H. Meyerhenke. Graph partitioning for balancing parallel adaptive numerical simulations. `http://wwwcs.upb.de/cs/henningm/graph.html`, 2007.

[21] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid. In *Proc. 20th Intl. Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[22] H. Meyerhenke and T. Sauerwald. Analyzing disturbed diffusion on networks. In *Proc. 17th International Symposium on Algorithms and Computation (ISAAC'06)*, volume 4288 of *LNCS*, pages 429–438. Springer, 2006.

[23] H. Meyerhenke and S. Schamberger. Balancing parallel adaptive FEM computations by solving systems of linear equations. In *Proc. 11th Intl. Euro-Par Conf. (Euro-Par'05)*, volume 3648 of *LNCS*, pages 209–219, 2005.

[24] H. Meyerhenke and S. Schamberger. A parallel shape optimizing load balancer. In *Proc. 12th Intl. Euro-Par Conf. (Euro-Par'06)*, volume 4128 of *LNCS*, pages 232–242, 2006.

[25] F. Pellegrini. A parallelisbale multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *13th Proc. Intl. Euro-Par Conf. (Euro-Par'07)*, volume 4641 of *LNCS*, pages 195–204. Springer, 2007.

[26] F. Pellegrini. Scotch and libscotch 5.0 user's guide. Technical report, LaBRI, Université Bordeaux I, December 2007.

[27] R. Preis. Linear time $1/2$-approximation algorithm for maximum weighted matching in general graphs. In *Proc. 16th Symp. on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *LNCS*, pages 259–269, 1999.

[28] S. Schamberger. On partitioning FEM graphs using diffusion. In *Proc. HPGC Workshop of 18th Intern. Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[29] S. Schamberger. *Shape Optimized Graph Partitioning*. PhD thesis, Universität Paderborn, 2006.

[30] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[31] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In *The Sourcebook of Parallel Computing*, pages 491–541. Morgan Kaufmann, 2003.

[32] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[33] A. J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph partitioning. *J. Global Optimization*, 29(2):225–241, 2004.

[34] D. A. Spielman and S.-H. Teng. Spectral partitioning works: Planar graphs and finite element meshes. In *Proc. 37th Symp. on Foundations of Computer Science (FOCS'96)*, pages 96–105. IEEE, 1996.

[35] K. Stüben. An introduction to algebraic multigrid. In U. Trottenberg, C. W. Oosterlee, and A. Schüller, editors, *Multigrid*, pages 413–532. Academic Press, London, 2000. Appendix A.

[36] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2000.

[37] C. Walshaw. The graph partitioning archive. `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`, 2007.

[38] C. Walshaw and M. Cross. Mesh partitioning: a multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000.

[39] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *J. Parallel Computing*, 26(12):1635–1660, 2000.

[40] G. Zumbusch. *Parallel Multilevel Methods: Adaptive Mesh Refinement and Loadbalancing*. Teubner, 2003.