# On Dynamic Graph Partitioning and Graph Clustering using Diffusion

Henning Meyerhenke and Joachim Gehweiler

University of Paderborn, Department of Computer Science
Fuerstenallee 11, D-33102 Paderborn, Germany
E-mail: `{henningm,joge}@upb.de`

**Abstract.** Load balancing is an important requirement for the efficient execution of parallel numerical simulations. In particular when the simulation domain changes over time, the mapping of computational tasks to processors needs to be modified accordingly. State-of-the-art libraries for this problem are based on graph repartitioning. They have a number of drawbacks, including the optimized metric and the difficulty of parallelizing the popular repartitioning heuristic Kernighan-Lin (KL).
Here we further explore the very promising diffusion-based graph partitioning algorithm DIBAP (Meyerhenke et al., JPDC 69(9):750–761, 2009) by adapting DIBAP to the related problem of load balancing. Experiments with graph sequences that imitate adaptive numerical simulations demonstrate the applicability and high quality of DIBAP for load balancing by repartitioning. Compared to the faster state-of-the-art repartitioners PARMETIS and parallel JOSTLE, DIBAP's solutions have partitions with significantly fewer external edges and boundary nodes and the resulting average migration volume in the important maximum norm is also the best in most cases.
We also prove that one of DIBAP's key components optimizes a relaxed version of the minimum edge cut problem. Moreover, we hint at a distributed algorithm based on ideas used in DIBAP for clustering a virtual P2P supercomputer.

**Keywords**: Dynamic graph partitioning/clustering, disturbed diffusion, load balancing, relaxed cut optimization.

## 1 Introduction

Partitioning the vertices of a graph such that certain optimization criteria are met, occurs in many applications in computer science, engineering, and related fields. The most common formulation of the graph partitioning problem for an undirected (possibly edge-weighted) graph $G = (V,E)$ (or $G = (V,E,\omega)$) asks for a division $\Pi$ of $V$ into $k$ pairwise disjoint subsets (*parts*) $\{\pi_1,\dots,\pi_k\}$ of size at most $\lceil |V|/k \rceil$ each, such that the *edge cut* is minimized. The edge cut is defined as the total number (or total weight) of edges having their incident nodes in different subsets. Among many others, the applications of this $\mathcal{NP}$-hard problem include load balancing in numerical simulations [37] and image segmentation [18,39].

The related task of clustering refers to the combination of objects to groups (clusters) such that objects of the same group are more similar to each other than to objects from other groups. Graph clustering searches for dense node subsets that are only sparsely connected to each other. These notions are intentionally vague [48], as they are application-dependent and in general hard to formalize. In contrast to graph partitioning, the number of clusters is not always part of the input, and no explicit constraints on the cluster sizes are given. In recent years the determination of clusters within graphs has received considerable attention, e. g., for image segmentation [39], detection of protein families [12], or the analysis of social networks [31].

Despite recent approximation algorithms for graph partitioning or related clustering tasks, simpler heuristics are preferred in practice, many of which can be found in surveys on graph partitioning [37] and graph clustering [34]. Spectral algorithms have been widely used [21]; they are global optimizers based on graph eigenvectors. For computational efficiency or quality reasons, they have been mostly superseded by local improvement algorithms in the area of graph partitioning. Integrated into a multilevel framework, local optimizers such as Kernighan-Lin (KL) [23] can be found in several popular partitioning libraries [7,22]. Unfortunately, theoretical quality guarantees are not known for KL. Another class of improvement strategies comprises diffusion-based methods [29,32]. While they are slower than KL, diffusive methods often yield a better quality, also when repartitioning dynamic graphs [29,30].

**Outline of the Paper.** In Sections 2.1 and 2.2 we describe the general approach of partitioning and clustering graphs with diffusion (and in particular with disturbed diffusion schemes). After that, we refer to related work in Section 2.3.

The technical part of the paper is divided into two parts. In the first one, we are concerned with the repartitioning of dynamic graphs. Major applications for this problem are parallel adaptive numerical simulations. More motivation is provided in Section 3.1. After that, we describe our multilevel algorithm DIBAP and its components BUBBLE-FOS/C and TRUNCCONS in Section 3.2. Later on, we investigate BUBBLE-FOS/C's optimization criterion (Section 3.3) and also describe DIBAP's good experimental results when repartitioning dynamic graph sequences (Section 3.4).

The second major part deals with the clustering of a virtual peer-to-peer supercomputer. Such a clustering is used for load balancing in parallel computations submitted by users to the system and for system management tasks. For this application area, local clustering algorithms are of importance. That is why we employ the local process diffusion in a suitable manner. After a more detailed problem motivation (Section 4.1) and formulation (Section 4.2), we describe how to obtain a fully distributed diffusive algorithm and report some experimental results in Sections 4.3 and 4.4, respectively.

## 2 Disturbed Diffusion for Graph Partitioning and Clustering

### 2.1 The Disturbed Diffusion Scheme FOS/C

Diffusion models many important transport phenomena such as heat flow. Generally speaking, a diffusion problem in a graph consists of distributing splittable load entities from some given seed vertex (or vertices) into the whole graph by iterative load exchanges between neighbor vertices. Typical diffusion schemes have the property to result in a balanced load distribution, in which every node has the same amount of load.

This is one reason why diffusion has been studied extensively for load balancing [46]. In our previous work [30], we have developed algorithms based on diffusion for the optimization of partition shapes. Before that, repartitioning methods employed diffusion mostly for computing *how much* load needs to be migrated between subdomains [36], not *which* elements should be migrated.

We call a diffusion scheme *disturbed* if it is modified such that its steady state does not yield a balanced distribution. The similarity measure FOS/C (first order scheme with constant drain) introduces a drain-based disturbance into the first order diffusion scheme. With the disturbance, FOS/C reaches a steady state whose load vector $w$ represents node similarities. These similarities reflect whether nodes are connected by many paths of short length. In the field of graph-based image segmentation, similar arguments based on the isoperimetric constant are used to find well-shaped segments [18].

**Definition 1.** *(FOS/C) [30] Given a connected and undirected graph $G = (V, E, \omega)$ free of self-loops, a set of source nodes $\emptyset \neq S \subset V$, initial load vector $w^{(0)}$, and constants $0 < \alpha \leq (\deg(G) + 1)^{-1}$ and $\delta > 0$.*[1] *Let the drain vector $d$ (which is responsible for the disturbance) be defined as $[d]_v = (\delta n/|S|) - \delta$ if $v \in S$ and $[d]_v = -\delta$ otherwise. 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 and $\mathbf{L}$ the Laplace matrix of $G$.*

**Lemma 1.** *[30] For any $d \perp (1, \dots, 1)^T$ (i. e., $\langle d, (1, \dots, 1)^T \rangle = 0$), the FOS/C iteration reaches a steady state, which can be computed by solving and normalizing the linear system $\mathbf{L}w = d$ .*

## 2.2 Partitioning/Clustering Graphs with Disturbed Diffusion

Since previous similarity measures related to diffusion to compute well-shaped partitions [18,30] are rather slow, Pellegrini has developed a faster bipartitioning mechanism [32] with a simplified diffusion process. In previous work we extended Pellegrini's approach and obtained a slightly different method, which is capable of direct $k$-way partitioning [29]. Since the diffusive concepts used in [29] are important to understand the remainder of this paper, we discuss them subsequently. In this section we use the terms clustering/partitioning and cluster/part interchangeably.

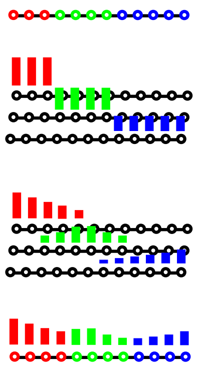
Fig. 1. Schematic view of diffusive clustering on a path graph and $k = 3$.

For assigning nodes to clusters by diffusive clustering, we associate $k$ diffusion load values with each node. These loads are distinguished by coloring them with colors from 1 to $k$, one color for each cluster. The basic idea is then to assign each node $v$ to the cluster with the highest diffusion load on $v$.

Of course we need to compute meaningful load values that yield a good clustering first. To achieve

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

this, a diffusive algorithm – such as FOS/C – is executed $k$ times (once for each of the $k$ clusters and their unique loads) to change the initial load distribution. The algorithm distributes the load into the graph. By doing so, it identifies densely connected regions by the intuition that random walks (and in a similar way also diffusion) are likely to stay in dense regions for a long time. After the load has been distributed, each node $v$ is assigned to the cluster for which $v$ has the highest load value.

Figure 1 illustrates how this principle works for a path graph and $k = 3$. One performs the following independently for each cluster $\pi_c$: First, the nodes of $\pi_c$ (see the input cluster assignment in the topmost row of Figure 1) receive an equal amount of initial load $n/|\pi_c|$, while the other nodes' initial load is set to 0 (second row). Then, a diffusive method is used to distribute this load (third row). To obtain meaningful clusters, the diffusive method must result in an unbalanced load distribution, preferably with peaks not far from the old cluster centers. Such a distribution can be obtained by introducing a suitable disturbance. The last row in Figure 1 shows the cluster assignment according to the new maximum load values. This whole procedure can be repeated to improve the solution quality. If necessary, additional operations such as balancing can be integrated into the procedure.

Note that the most important ingredient of this generic iterative procedure is the particular (disturbed) diffusion process to distribute the load. It decides the running time and the quality of the whole method.

## 2.3 Other Related Work

The areas graph partitioning and graph clustering have grown very much in the last 15 years. Hence, in the description of related work, we concentrate on techniques that are closely related and at least partially applicable in our scenarios.

To address the load balancing problem in parallel applications, distributed versions of the partitioners METIS, JOSTLE, and SCOTCH [38,45,7] (and the parallel hypergraph partitioners ZOLTAN [6] and PARKWAY [41]) have been developed. An efficient parallelization of the KL/FM heuristic that these tools use is very complex due to inherently sequential parts in this heuristic. For example, one needs to ensure that during the KL/FM improvement no two neighboring vertices change their partition simultaneously and therefore destroy the consistency of the data structures.

There exist many graph clustering/partitioning techniques (see [34]) using random walk [25] techniques. These techniques exploit that random walks are likely to stay a long time in a dense graph region before leaving it via one of the few outgoing edges. Many diffusive processes are described by stochastic matrices and are therefore related to random walks [25].

Meila and Shi [26] connect random walks to spectral partitioning. Spectral methods such as [39] solve relaxations of IPs that minimize the edge cut or the related ratio cut. They build on Fiedler's seminal work on spectral partitioning [13] and use eigenvectors of Laplace or adjacency matrices for partitioning. A spectral relaxation to the geometric $k$-means clustering problem is given by Zha et al. [47]. It is used by Dhillon et al. [10] to develop a kernel-based graph clustering algorithm without using eigenvectors.

Further related to our localized diffusive methods for graph clustering is the algorithm by Harel and Koren [19], which computes separator edges iteratively based on the similarity of their incident vertices. This similarity is derived from the sum of transition probabilities of random walks with few steps. The actual partitioning into clusters requires a global statistical test to obtain suitable threshold values. The algorithm MCL [12,43] bypasses the problem of choosing a suitable random walk length by a nonlinear matrix operator, which strengthens the differences between all rows of the matrix. The operator is combined with the traditional multiplication of the random walk transition matrix. This combination leads to meaningful clusters. In the actual implementation (called mcl) of the algorithm, the problem of densely populated intermediate matrices is avoided by pruning, i. e., setting small matrix entries to zero. Yet, the fill-in can be significant.

Recent clustering algorithms for dynamic graphs include the one by Görke et al. [17]. It uses minimum-cut trees to compute a clustering and a method to update this data structure efficiently when the graph changes. Computing the initial tree, however, is a global and expensive operation. A recent algorithm with a running time that depends only weakly on the graph size is given by Andersen and Peres [2]. It extends and improves previous theoretical research on local clustering algorithms and uses Markov chain arguments to find local cuts with small conductance, a cut measure. We are not aware of any implementations of this algorithm for a practical scenario. Another mostly local algorithm has been presented by Delling et al. [9]. The algorithm works without an explicit optimization criterion. Its key component recursively identifies small dense regions and contracts them into single vertices. The identification of small dense regions is a local process, but the contraction order is governed by a global priority queue.

## 3 Graph (Re-)Partitioning with the Algorithms BUBBLE-FOS/C and DIBAP

### 3.1 Motivation

Numerical simulations are very important tools in science and engineering for the analysis of physical processes modeled by partial differential equations (PDEs) [14]. To make the PDEs solvable, they are discretized within the simulation domain, e. g., by the finite element method (FEM). Such a discretization yields a mesh, which can be regarded as a graph with geometric (and possibly other) information.

The solutions of discretized PDEs are usually computed by iterative numerical solvers, which have become classical applications for massively parallel computers. To utilize all available processors in an efficient manner, the computational tasks, represented by the mesh elements, must be distributed onto the processors evenly. Moreover, the computational tasks of an iterative numerical solver depend on each other. Neighboring elements of the mesh need to exchange their values in every iteration to update their own value. Since inter-processor communication is much more expensive than local computation, neighboring mesh elements should reside on the same processor. A good initial assignment of subdomains to processors can be found by solving the graph partitioning problem (GPP) [37].

In many numerical simulations some areas of the mesh are of higher interest than others. For instance, during the simulation of the interaction of a gas bubble with a surrounding liquid, one is interested in the conditions close to the boundary of the fluids. To obtain an accurate solution, a high resolution of the mesh is required in the areas of interest. A uniformly high resolution is often not feasible due to limited main memory. That is why one has to work with different resolutions in different areas. Moreover, the areas of interest may change during the simulation, which requires *adaptations* in the mesh and may result in undesirable load imbalances. Hence, after the mesh has been adapted, its elements need to be redistributed such that every processor has a similar computational effort again. While this can be done by solving the GPP for the new mesh, the *repartitioning* process not only needs to find new partitions of high quality. Also as few nodes as possible should be moved to other processors since this *migration* causes high communication costs and changes in the local mesh data structure.

The most popular graph partitioning and repartitioning libraries use local node-exchanging heuristics like Kernighan-Lin (KL) [23] within a multilevel improvement process to compute good solutions very quickly. Yet, their deployment can have certain drawbacks. First of all, minimizing the edge-cut with these tools does not necessarily mean to minimize the total running time of parallel numerical simulations [44]. The number of *boundary vertices* (vertices that have a neighbor in a different partition), for instance, models the communication volume between processors often more accurately than the edge-cut [20]. While the total number of boundary vertices can be minimized by hypergraph partitioning [5], synchronous parallel applications need to wait for the processor computing longest. Hence, the *maximum norm* (i. e., the worst partition) of the load balancing costs and the simulation's communication costs is of higher importance. Moreover, for some applications, the *shape* of the subdomains plays a significant role. It can be assessed by various measures such as aspect ratio [11], maximum diameter [32], connectedness, or smooth boundaries. Finally, due to their sequential nature, the most popular repartitioning heuristics are difficult to parallelize – although significant progress has been made.

**Outline.** Our previously developed partitioning algorithm DIBAP aims at computing well-shaped partitions and uses disturbed diffusive schemes to decide not only *how many* nodes move to other partitions, but also *which* ones. It is described in some detail in Section 3.2. That DIBAP's key ingredient BUBBLE-FOS/C is in fact a relaxed edge cut optimizer, is stated in Section 3.3. Moreover, DIBAP is inherently parallel and overcomes many of the above mentioned difficulties, as could be shown experimentally for static graph partitioning [29]. While it is much slower than state-of-the-art partitioners, it often obtains better results. We report on experimental results from dynamic repartitioning scenarios in Section 3.4. These results confirm DIBAP's high solution quality, but also its high running time compared to established parallel KL-based repartitioners such as PARMETIS.

### 3.2 BUBBLE-FOS/C + TRUNCCONS + Multilevel = DIBAP

DIBAP is a hybrid multilevel combination of the two (re-)partitioning methods BUBBLE-FOS/C and TRUNCCONS, which are both based on disturbed diffusion [29]. As noted before, disturbed diffusion schemes can be helpful to determine if two graph nodes or regions are densely connected to each other. This property is due to the similarity of diffusion to random walks and the notion that a random walk stays in a dense region for a long time before leaving it via one of the few external edges. Before we explain the multilevel scheme DIBAP, we describe its two main (re-)partitioning components in more detail.

**BUBBLE-FOS/C** In contrast to Lloyd's related $k$-means algorithm [24], BUBBLE-FOS/C partitions or clusters graphs instead of geometric inputs. Given a graph $G = (V, E)$ and $k \geq 2$, initial partition representatives (centers) are chosen in the first step of the algorithm, one center for each of the $k$ partitions. All remaining vertices are assigned to their closest center vertex. While for $k$-means one usually uses Euclidean distance, BUBBLE-FOS/C employs the disturbed diffusion scheme FOS/C as distance measure (or, more precisely, as similarity measure). The similarity of a node $v$ to a non-empty node subset $S$ is computed by solving the linear system $\mathbf{L}w = d$ for $w$, where $\mathbf{L}$ is the Laplacian matrix of the graph and $d$ a suitably chosen vector that disturbs the underlying diffusion system. After the assignment step, each partition computes its new center for the next iteration – again using FOS/C, but with a different right-hand side vector $d$. The two operations *assigning vertices to partitions* and *computing new centers* are repeated alternately a fixed number of times or until a stable state is reached. Each operation requires the solution of $k$ linear systems, one for each partition.

It turns out that this iteration of two alternating operations yields very good partitions. Apart from the distinction of dense and sparse regions, FOS/C tends to produce similarity isolines with a circular shape. Thus, the final partitions are very compact and have short boundaries. However, the repeated solution of linear systems makes BUBBLE-FOS/C slow.

**TRUNCCONS** The algorithm TRUNCCONS (for *truncated consolidations*) is also an iterative method for the diffusion-based local improvement of partitions, but it is much faster than BUBBLE-FOS/C. Within each TRUNCCONS iteration the following is performed independently for each partition $\pi_c$: First, the initial load vector $w^{(0)}$ is set. Nodes of $\pi_c$ receive an equal amount of initial load $|V|/|\pi_c|$, while the other nodes' initial load is set to 0. Then, this load is distributed within the graph by performing a small number $\psi$ of FOS (first order diffusion scheme) [8] iterations. The final load vector $w$ is computed as $w = \mathbf{M}^{\psi}w^{(0)}$, where $\mathbf{M} = \mathbf{I} - \alpha\mathbf{L}$ denotes the diffusion matrix [8] of $G$. A common choice for $\alpha$ is $\alpha := (1 + \deg(G))^{-1}$. After the load vectors have been computed this way independently for all $k$ partitions, each node $v$ is assigned to the partition it has obtained the highest load from. This completes one TRUNCCONS iteration, which can be repeated several times (the total number is denoted by $\Lambda$ subsequently) to facilitate sufficiently large movements of the partitions.

There are two reasons why TRUNCCONS is usually much faster than BUBBLE-FOS/C. First, solving linear systems in BUBBLE-FOS/C is much costlier than a few matrix-vector products in TRUNCCONS. Also, within TRUNCCONS a node with the same amount of load as all its neighbors does not change its load in the next FOS iteration. Due to the choice of initial loads, such an *inactive* node is a certain distance away from the partition boundary. By avoiding load computations for inactive nodes, we restrict the computational effort to areas close to the partition boundaries.

**DIBAP** The main components of DIBAP are depicted in Figure 2. To build a multilevel hierarchy, the fine levels are coarsened (1) by approximate maximum weight matchings [33]. Once the graphs are sufficiently small, we switch the construction mechanism (2) to the more expensive coarsening based on algebraic multigrid (AMG) [30]. This is advantageous regarding running time because, after computing an initial partitioning (3), BUBBLE-FOS/C is used as local improvement algorithm on the coarse levels (4). Since BUBBLE-FOS/C uses AMG as a linear solver, such a hierarchy needs to be built anyway. Eventually, the partitionings on the fine levels are improved by the local improvement scheme TRUNCCONS. DIBAP includes additional components, e. g., for balancing partition sizes and smoothing partition boundaries. Their description is outside the scope of this paper and can be found in [35].

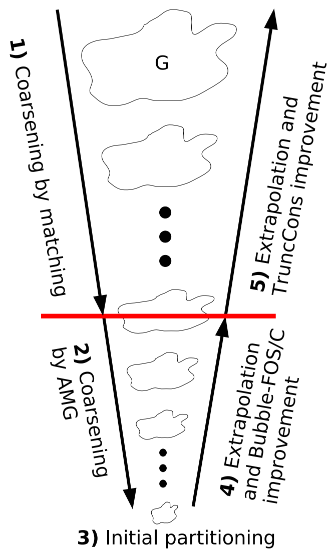
1) Coarsening by matching
G
5) Extrapolation and TruncCons improvement
2) Coarsening by AMG
4) Extrapolation and Bubble-FOS/C improvement
3) Initial partitioning

**Fig. 2.** Sketch of the combined multilevel hierarchy and the corresponding repartitioning algorithms used within DIBAP.

The rationale behind DIBAP can be explained as follows. While BUBBLE-FOS/C computes high-quality graph partitions with good shapes, its similarity measure FOS/C is quite expensive to compute compared to established partitioning heuristics. Consequently, BUBBLE-FOS/C's running time is too high for real practical value. To overcome this problem, we use the simpler process TRUNCCONS, a truly local algorithm to improve partitions generated in a multilevel process. It exploits the observation that, once a reasonably good solution has been found, alterations during a local improvement step take place mostly at the partition boundaries. The disturbing truncation within TRUNCCONS allows for a concentration of the computations around the partition boundaries, where the changes in subdomain affiliation occur. Moreover, since TRUNCCONS is also based on disturbed diffusion, the good properties of the partitions generated by BUBBLE-FOS/C are mostly preserved.

## 3.3 BUBBLE-FOS/C is a Relaxed Cut Minimizer

It has been shown before [29, Thm. 10], that the iterative optimization performed by the graph partitioning heuristic BUBBLE-FOS/C can be described by a potential function. This function $F$ sums up the diffusion load of each vertex $v \in V$ in a single-source

FOS/C procedure in time step $\tau$ with $v$'s closest center vertex $z_p$ as source. In fact, the results computed by the operations `AssignPartition` and `ComputeCenters` each maximize $F$ for their fixed input (centers or parts, respectively). However, it has been unclear until recently how these facts relate to the good experimental results of BUBBLE-FOS/C with respect to metrics more specific to graph partitioning. This question is – at least partially – addressed by the following theorem:

**Theorem 1.** *[28] Let $k \geq 2$. Given a graph $G = (V, E, \omega)$ with $n$ nodes ($n/k \in \mathbb{N}$) and a set $Z$ with one center vertex for each of the $k$ parts. Then, the two consecutive operations `AssignPartition` and `ScaleBalance` with suitable $\beta_p$ ($1 \leq p \leq k$) together compute the global minimum of an optimization problem which is a relaxed version of a binary quadratic program BQP. If $Z = \{z_{1,\dots,}z_k\}$ is given such that $z_p \in \pi_p$ and $\Pi = \{\pi_1, \dots, \pi_p\}$ is an (unknown) optimal (with respect to the edge cut) partition, then BQP computes an optimal partition.*

The result above shows that – under mild conditions – BUBBLE-FOS/C solves a relaxed edge cut minimization problem. This is slightly surprising: In previous experiments with numerical simulation graphs [30], BUBBLE-FOS/C was compared to the popular partitioning libraries KMETIS and JOSTLE. The best improvements by BUBBLE-FOS/C could be seen regarding the number of boundary nodes and the shape of the parts. Yet, concerning the edge cut, the improvement over the other libraries was not as clear, probably because KMETIS and JOSTLE focus chiefly on the edge cut.

## 3.4 Experimental Repartitioning Results

We now turn our attention to experiments that require the repartitioning of graph sequences. Unlike the parallel versions of METIS and JOSTLE, the implementation of our load balancer DIBAP is not prepared yet for a distributed-memory parallelization.[2] That is why we concentrate in the following on the quality of the experiments and neglect their running time. Comparing the latter is part of future work with an MPI parallel version of DIBAP. Preliminary results suggest that the average slowdown factor for using DIBAP instead of PARMETIS (which is faster than parallel JOSTLE) depends very much on the graph size and $k$ and lies approximately between 30 and 60.

The experimental setup is described in more detail in [27], here we provide only a summary. Our benchmark set comprises of two types of graph sequences. Twelve sequences are made of 101 frames of small graphs (around 10,000 to 15,000 nodes each), which are repartitioned into $k = 12$ subdomains. The second set consists of three sequences of larger graphs (between 110,000 and 1,100,000 nodes each), which are repartitioned into $k = 16$ subdomains. While the sequence *bigtric* has 101 frames, the sequences *bigbubbles* and *bigtrace* have only 46 frames.

All graphs of these 12+3 sequences have a two-dimensional geometry and are generated to resemble adaptive numerical simulations such as fluid dynamics. The graph of frame $i+1$ in a given sequence is obtained from the graph of frame $i$ by changes

[2] We would like to point out that the *algorithm* DIBAP is very well suited for such a parallelization approach. Only its corresponding implementation has not been finished yet. The implementation used for this paper is based on POSIX threads instead.

restricted to local areas. As an example, some areas are coarsened, whereas others are refined. These changes often result in unbalanced subdomain sizes.

Our results can be summarized as follows: The averaged graph partitioning metrics show that DIBAP is able to compute the best partitions on average. DIBAP's advance is highest for the boundary nodes in the maximum norm, which can be considered a more accurate measure for the communication costs of typical numerical solvers than the edge-cut. With about 12-15% on parallel JOSTLE and about 23-30% on PARMETIS these improvements are clearly higher than the approximately 7% for static partitioning obtained in [29], which is due to the fact that parallel KL (re-)partitioners often compute worse solutions than their serial counterparts for static partitioning.

DIBAP's implicit optimization with the iterative algorithms BUBBLE-FOS/C and TRUNCCONS focusses more on good partitions than on small migration costs. In some special cases the latter objective should receive more attention. Concerning the migration volume, the results are not as clear. The $\ell_1$-norm values are slightly in favor of JOSTLE (eight times best) compared to DIBAP (six times best). Yet, in the $\ell_\infty$-norm of the migration volume, DIBAP is the clear winner again. The strategy of PARMETIS to migrate either very few or very many nodes does not seem to pay off on average since PARMETIS computes in most cases the worst solutions.

We would like to stress that a high repartitioning quality is often very important. Usually, the most time consuming parts of numerical simulations are the numerical solvers. Hence, a reduced communication volume provided by an excellent partitioning can pay off unless the repartitioning time is extremely high.

## 4 Distributed Graph Clustering with DIDIC

Diffusion is in general a local process. Load entities are exchanged between neighbors without any kind of central control. That is why disturbed diffusion is also very suitable in distributed environments. In the following we describe a heuristic algorithm for clustering a virtual peer-to-peer (P2P) supercomputer.

### 4.1 Motivation

The main application we consider in this section is the management of a P2P based virtual distributed supercomputer for parallel computations in the *bulk-synchronous parallel* (BSP) [42] style. This virtual supercomputer is the *Paderborn University BSP-based Web Computing (PUB-Web)* library [3], formerly known as PUBWCL. It was originally designed as a hybrid P2P system: People can join the PUB-Web system by installing the peer component software, which enables them to donate unused computing power (the CPU's idle time) and to run their own parallel programs. The execution of a parallel program is carried out in pure P2P fashion among the peers assigned to the job. Administrative tasks such as scheduling and load balancing, however, are performed by a server. To obtain a better scalability by making PUB-Web a pure peer-to-peer system, a distributed load balancing algorithm has been developed [16].

The load balancer in [16] assigns the BSP processes to machines such that each BSP process receives approximately the same amount of computing power, and it rebalances the process assignment accordingly upon changes in the machines' availabilities. Once a certain number of machines are handled by the load balancer, the results are quite good and can only be slightly improved by adding more machines, while the overhead of adding more machines starts to affect the load balancer's performance. For this reason and to consider the network speed in addition to the load of the machines, our goal is to split a large network into clusters.

As the network speed appears to be a rather static parameter of a machine in contrast to its processor's idle time, we use the clustering approach presented in this paper to determine subsets of machines among which the network speed is high. All processes belonging to a BSP program will then be assigned to one cluster, and inside a cluster the existing load balancer will schedule them according to the machines' availabilities. In our application the network speed is characterized by bandwidth and latency. Since the importance of latency rapidly decreases with growing message sizes in the BSP model [42], we focus on bandwidth. We model the machines in the PUB-Web network as vertices in a graph and choose the bandwidth as our similarity measure (or the inverse of the bandwidth as our distance measure), i. e., high edge weights correspond to high bandwidths and vice versa.

The clusters do not have the same size for this application. Yet, they should preferably be connected because we cannot influence to which vertices in a cluster the load balancer will assign the processes of a BSP program; in case they are spread over two or more non-connected vertex subsets, the communication performance of the BSP program will suffer. Moreover, the clustering should identify those vertices as clusters which have very good network connections among each other. On the other hand, the clusters should have a certain minimum size because the performance of the load balancer is quite poor on very small instances. An important constraint is that the clustering must be computed by a distributed algorithm executed on the peers. Moreover, the algorithm must be able to start from an arbitrary initial configuration.

## 4.2 Problem Formulation

Let $\mathscr{G} := \bigcup_{i=0}^{T} G_i = (V_i, E_i, \omega_i)$ be a dynamic undirected and edge-weighted graph, i. e., a collection of static graphs $G_i$ with vertex set $V_i$, edge set $E_i$, and corresponding edge weight set $\omega_i$. The graph $G_{i+1}$ is constructed from $G_i$ by inserting and deleting certain vertices and/or edges. A $k$-way clustering of a graph is a function $\Pi_i : V_i \to \{1, \dots, k\}$. Such a clustering divides the vertex set $V_i$ into $k$ disjoint subsets $V_i = \pi_{i,1} \dot\cup \pi_{i,2} \dot\cup \dots \dot\cup \pi_{i,k}$. Edges connecting vertices of two different clusters belong to the so-called *cut* of $\Pi_i$.

For an undirected, edge-weighted graph $G = (V, E, \omega)$ with $n$ vertices and its $k$-way clustering $\Pi$ (as from now we omit the index $i$ for ease of presentation), let $\deg(v) := \sum_{e=\{\cdot,v\}\in E} \omega(e)$ be the weighted degree of vertex $v$ and $N(v) := \{u \mid \{u,v\} \in E\}$ its neighborhood. Let $\text{intra-weight}(\pi_c) := \sum_{e=\{u,v\}\in E; u,v\in\pi_c} \omega(e)$ and $\text{cut-weight}(\pi_c) := \sum_{e=\{u,v\}\in E; u\in\pi_c, v\notin\pi_c} \omega(e)$ be the weight of intra-cluster and cut edges of $\pi_c$, respectively. Finally, denote the total edge weight of $G$ by $\text{ew}(G) := \sum_{e\in E} \omega(e)$. Then, the popular clustering quality measure *modularity* [31] is defined as

$$Mod(\Pi) := \sum_{c=1}^{k} \left( \frac{\text{intra-weight}(\pi_c)}{\text{ew}(G)} - \left( \frac{\sum_{v\in\pi_c} \deg(v)}{2\,\text{ew}(G)} \right)^2 \right) \quad (1)$$

The above version of the modularity definition is derived from Brandes et al. [4], who consider the unweighted case. They point out that maximizing modularity involves a trade-off between producing many intra-cluster edges (first part of the main sum) and producing a large number of clusters with small degree (second part), yielding more cut-cluster edges. A probabilistic interpretation of modularity states that it measures the fraction of intra-cluster edges minus the expected value of the same quantity in a graph with the same clustering but random connections between the nodes [31]. It is $\mathscr{NP}$-hard to optimize modularity [4] and nearly all interesting clustering metrics [40] for general graphs.

Each connected component of the subgraph induced by the vertices of cluster $\pi_c$ is called a *cluster-connected component* of $\pi_c$. The set of all cluster-connected components of $\pi_c$ is denoted by $CCC(\pi_c)$. The *nearly connected value* (NCV) of $\pi_c$ is then defined as

$$NCV(\pi_c) := \frac{\max_{S \in CCC(\pi_c)} |S|}{|\pi_c|} . \tag{2}$$

As argued above, a good clustering for the PUB-Web application is preferably connected and groups nodes with a high mutual bandwidth. Hence, our clusterings should have a good cut-related measure (such as high modularity) and clusters with high NCV. In case of dynamic graphs, a high-quality clustering should be obtained and maintained from a certain time step on. Moreover, local changes in the graph structure should entail also only local changes in the clustering.

### 4.3 Distributed Diffusive Clustering

In this section we describe our distributed diffusive clustering algorithm DIDIC. While its general idea of clustering by distributing $k$ different kinds of load with a diffusive method is based on the BUBBLE framework described before, additional techniques have to be introduced to make the algorithm work in a distributed setting, where we assume that data used by node $v$ is either stored at $v$ or at $v$'s neighbors.

There are two main occurrences of global knowledge in our (re-)partitioning algorithms TRUNCCONS and BUBBLE-FOS/C [29,30]. The first one is the initial assignment of vertices to clusters, which is important for these algorithms to compute high-quality solutions quickly. Here we replace it by default random initialization. The second occurrence is the drain concept responsible for the disturbance that results in meaningful unbalanced load distributions. Here we realize the disturbing drain concept by a second diffusion system. That is why each node stores two load vectors $w$ and $l$ of length $k$, the number of clusters.

Subsequently we denote by $w_v$ the primary load vector of node $v \in V$ and by $w_v^{(t)}(c)$ the load of $v$ in the diffusion system $c$ at time step $t$. If the index $v$ in $w_v$ is omitted, we refer to the vector of all load values $w_i(c)$, $1 \leq i \leq n$, within a particular diffusion system $c$. A missing $^{(t)}$ means a vector belonging to no fixed time step. The notation for the secondary load vector $l$ is analogous.

**Setting the Initial Situation** If the initial cluster affiliation of a node is undefined (which we assume to be the default case), the only information a node has in our distributed scenario (besides its neighborhood structure and the loop durations) is the maximum number of clusters $k$. Thus, a vertex's initial affiliation is chosen as a random number between 1 and $k$. Afterwards each node $v$ sets its entries of the initial load vectors $w_v^{(0)}(c)$, $c = 1, \ldots, k$, to 0 with one exception. The load value corresponding to the own cluster is set to a high constant value, e. g., 100. The load vectors $l$ of the secondary diffusion system, whose purpose is explained in the following sections, are initialized in the same way.

**Eliminating Global Knowledge with Suitable Diffusive Processes** Recall that FOS/C uses a drain concept to disturb the process and obtain unbalanced load distributions that are meaningful for clustering. In each iteration a small amount of load (the drain) is subtracted from all nodes. It is reinserted into the system by adding the total drain equally divided onto a specified set of source vertices $S \subset V$. In our case $S$ would be the cluster corresponding to the diffusion system. This equal division requires knowledge on the size of $V$ and $S$. In particular $|S|$ is not accessible in our distributed scenario. Also, we have to cope with the fact that in our case the initial clustering is random. Hence, the fast forming of cluster-connected vertex sets should be facilitated. We address these issues by using not only one diffusion system per cluster, but two, each representing the same load color. The purpose of the secondary system, explained in more detail later, is to send load of system $i$ very quickly to nodes belonging to cluster $i$. It thus takes over the role of the drain sent to the nodes of $S$ and also accelerates the forming of large cluster-connected components.

The task of the primary diffusion system is to exploit the property of diffusion and random walks to identify dense graph regions. The primary load values are computed by using a small number $\psi$ of FOS iterations – instead of iterating until all vertices have the same amount of load. The difference to a TRUNCCONS phase is the addition of the secondary load values in vector $l$.

When starting with a random initial clustering, it is advisable to form large cluster-connected regions quickly. This is the task of the secondary diffusion system. By using node weights (which we call benefits to express their purpose: a node of the corresponding cluster benefits from the secondary system), the secondary system directs load of system $i$ quickly to nodes of the $i$th cluster. We call this method *first order diffusion scheme disturbed by vertex benefits* (FOS/B). Its calculations are shown in Algorithm 1. The benefit values $b_u(c)$ and $b_v(c)$ in the denominators of the load exchange formula ensure that nodes not in cluster $i$ send most of their load in $l$ to their neighbors in cluster $i$ (if they have any). Once load of color $i$ is accumulated in a larger cluster-connected region, it will be used within the primary diffusion system to flood adjacent areas.

How the diffusive schemes are actually intertwined, can be seen in Algorithm 1. More details can also be found in [15].

**Discussion of the Algorithm** Our distributed diffusive clustering algorithm, abbreviated DIDIC, is shown as Algorithm 1. It is executed in a distributed way, parametrized by the respective node $v \in V$. Note that it is possible to assign an initial cluster number

**Algorithm 1** Distributed Diffusive Clustering Algorithm

```
DiDiC($v$, $N(v)$, $\pi$, $k$, $T$, $\psi$, $\rho$) $\to \pi$
if ($\pi$ is undefined)
  $\pi$ := RandomValue(1, $k$);
$w_v$ := SetInitialLoad($\pi$); $l_v := w_v$;
for time step t := 1 to T do
  for each cluster system c do
    for s := 1 to $\psi$ do (* FOS/T *)
      $\overline{w}_v(c) := w_v(c)$;
      for r := 1 to $\rho$ do (* FOS/B *)
        $\overline{l}_v(c) := l_v(c)$;
        for each $u \in N(v)$ (* $e = \{u,v\} \in E$ *)
          $\overline{l}_v(c) := \overline{l}_v(c) - \alpha(e) \cdot \omega(e) \cdot \left(\frac{l_v(c)}{b_v(c)} - \frac{l_u(c)}{b_u(c)}\right)$;
      for each $u \in N(v)$ (* $e = \{u,v\} \in E$ *)
        $\overline{w}_v(c) := \overline{w}_v(c) - \alpha(e) \cdot \omega(e) \cdot (w_v(c) - w_u(c))$;
      $w_v(c) := \overline{w}_v(c) + \overline{l}_v(c)$; $l_v(c) := \overline{l}_v(c)$;
  $\pi := \mathrm{argmax}_{c=1,\ldots,k}\, w_v(c)$;
  $N(v)$ := adaptToGraphChanges($v$, $N(v)$);
```

to $v$ by specifying it in the parameter $\pi$. However, while the possibility exists, we assume that in our scenario it is usually not used and $\pi$ is undefined until it is initialized randomly. Recall from the introduction that the algorithm is expected to work with arbitrary initial clusterings. Even with the simple strategy of random initialization DiDiC performs well in our experiments (if the number of time steps $T$ is reasonably large).

After the clustering has been initialized, the diffusion load vectors are set accordingly. Then the actual diffusive clustering process is started by the outermost loop, which runs for `T` time steps. At the end of each time step, the new cluster affiliation of each vertex $v$ can be chosen generically as $\mathrm{argmax}_{c=1,\ldots,k}\, w_v(c)$. Additional adaptations to accelerate the clustering process are possible [15]. After that the graph may be modified by local changes. Such changes include the addition or deletion of nodes and/or edges. In case of the deletion of a node $v$, the execution of the algorithm on $v$ is stopped and its current load is distributed evenly among its neighbors. This simple strategy is why our diffusive method works well on dynamic graphs. Local changes in the graph affect the distribution of the diffusion loads only slightly. Thus our algorithm recovers quickly from small alterations of $G$ and adapts the former clustering accordingly.

Within each time step the clustering is performed by calculating diffusion systems for each cluster, more precisely by an outer FOS/T loop, into which an inner FOS/B loop is embedded. A straightforward upper bound for the resulting time complexity per time step for each node $v$ is $\mathcal{O}(k \cdot \psi \cdot \rho \cdot \deg(v))$. Taking all nodes into account, a non-distributed version of the algorithm would require $\mathcal{O}(k \cdot \psi \cdot \rho \cdot maxNeigh \cdot n)$ operations per time step. Reasonable values for the parameters appearing in this expression and other details can be found in [15]. Note that we assume the input graph to change continuously. Thus, the clustering needs to be adapted continuously as well and we can forgo a solution-aware termination mechanism. Moreover, it is important to point out that DiDiC is not designed to deliver good clusterings from early time steps on. Instead, the random initial clustering needs to be refined from time step to time step. How long it takes to obtain reasonably good results, will be discussed during the presentation of our experiments.

## 4.4 Experimental Results

In this section we discuss some of our experimental results with emphasis on a scenario that resembles a P2P environment occurring in PUB-Web. All experiments are carried out within a simulator written in C/C++. Except for the running time, the simulator computes the same results as if a real distributed system were executing DiDiC (under the assumption of precise floating point calculations). For generating the test graphs, the vertices are embedded into the two-dimensional unit square with wrap-around boundaries (hence, it is actually a torus, but we use the word square to avoid confusion with torus graphs). Such an embedding with the assignment to coordinates is not strictly necessary, but the generation of graphs with certain properties is simplified. Moreover, vertex coordinates do not affect the clustering results since the algorithm does not use the coordinates. To create edges, we employ a slight variation of the disc graph model (e. g., [1]). It uses a uniform communication radius *rad* for all vertices. A vertex is connected to up to *maxNeigh* nearest neighbors within its communication radius, where *maxNeigh* is a user-defined parameter.

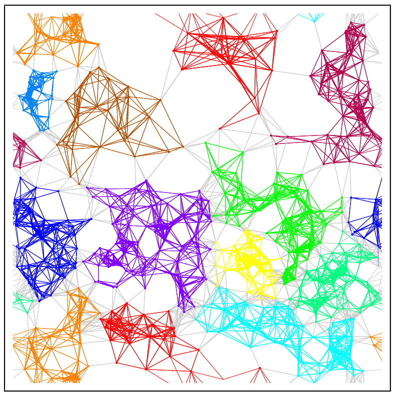
**Fig. 3.** The DiDiC clustering of a generated PUB-Web P2P graph after time step 55 with 800 nodes. Note that the unit-square has wrap-around boundaries. Cut edges are shown in grey. Parameters: $n = 800$, $k = 20$, $maxNeigh = 16$, $rad = 0.33$, $\psi = 11$, $\rho = 11$, $B = 10$, $x = 10$, dynamics: 2%.

To resemble the real PUB-Web network closely, we have built the following test scenario: Well connected subnets in companies, universities, etc. are represented by circular dense areas of nodes. That the nodes equally spread over the circular region do not have a pairwise equal distance to each other, is a realistic assumption because big networks are usually organized hierarchically. Thus, computers connected to the same switch can simultaneously communicate with each other at full speed, whereas the bandwidth decreases when they have to share the same up- und downlinks while communicating over several hops. The coordinates of the $x$ dense areas are chosen uniformly at random within $[0,1]^2$ and their radii range from 0.01 to 0.36 with an anti-quadratic probability distribution (small radii have higher probability). In addition to these dense areas, single vertices representing private home computers are randomly spread over the unit-square. More details can be found in [15].

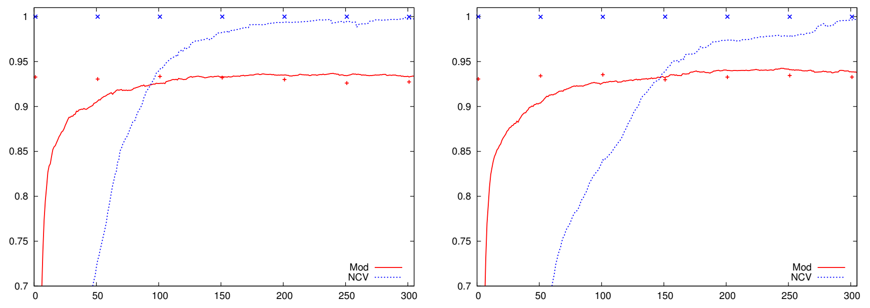

**Fig. 4.** Aggregated results (x-axis: time step, y-axis: modularity, NCV) for the same graph class as in Figure 3, but for a graph with 2400 (left) and 3200 (right) vertices.

We did extensive tests with varying parameters and dozens of graphs with e. g., 1600, 2400, and 3200 vertices. In order to simulate the dynamics of a P2P network, we randomly deleted and inserted 1%, 2% or 5% of the vertices each, every second, or every fifth time step, respectively. We are aware that the real PUB-Web network may be magnitudes larger than just a few thousands of nodes, but it is not appearing out of a sudden with a random configuration. Rather, it will be dynamically growing or shrinking over time. Thus, we are convinced that it is sufficient to perform tests for initial instances with a few thousands of vertices.

Evaluation criteria are modularity and the NCV value (the latter is averaged over all clusters). Note that empty clusters are seen as non-existing when computing these measures. The clusterings computed by DIDIC are compared with those computed by the graph clustering library mcl [43], which implements the algorithm MCL (see Section 2.3). As MCL is not a distributed algorithm, it should not be seen as a competitor. We use mcl only to validate DIDIC and it is not apparent that other algorithms are more suitable for this purpose. The inflation parameter of mcl strongly affects the granularity of clusters, which we set to 1.2. This choice avoids an extremely large number of small clusters and allows for a better comparison. Figure 3 shows a clustering obtained by DIDIC after 55 time steps for a graph with 800 vertices, 2% dynamics, and the diffusion iteration counts $\psi$ and $\rho$ set to 11. The modularity value obtained is 0.867 (mcl: 0.871). Smaller values of $\psi$ and $\rho$ make DIDIC faster, but often the quality worsens. The plots in Figure 4 are based on aggregated data of two experiment series with six graphs for each plot ($\psi = \rho = 11$) and show how modularity and NCV evolve over time for graphs of the same class with 2400 and 3200 nodes, respectively. For validation purposes, we ran mcl every 50-th time step on the dynamic graphs and added its modularity and NCV values to the plots as single points in the same colors as the corresponding lines. As one can see, once DIDIC's clusterings stabilize after about 150 time steps, they have a comparable or even sightly better modularity than mcl's. Considering that each node sends roughly 100-150 very small messages in its local neighborhood per time step, a time step would take a couple of seconds to complete in a real world scenario (the simulator requires roughly one second). Thus, the clustering would stabilize after a few minutes.

## 5 Conclusions

This paper shows that partitioning and clustering graphs by means of diffusive techniques is a very promising approach. The disturbed diffusive (re-)partitioning algorithm DIBAP is a clear alternative to traditional KL-based methods for balancing the load in parallel adaptive numerical simulations. While DIBAP is still significantly slower than the state-of-the-art, it usually computes considerably better solutions. In situations where the quality of the load balancing phase is more important than its running time – e. g., when the computation time between the load balancing phases is relatively high – the use of DIBAP is expected to pay off.

In the second major part of this paper, we presented a distributed heuristic for clustering graphs. The use of diffusion makes it suitable for the implicit optimization of cut-related quality measures such as modularity. Although communication takes only place between neighboring graph nodes, random initial configurations are transferred into meaningful clusterings with a low memory footprint. With extensive experiment series (only few of which have been presented here), we have shown that for our main application the clustering results are comparable to (and partially even better than) those of an established non-distributed algorithm.

Future work is mostly concerned with the acceleration of our algorithms and their tuning for difficult instances. Deeper theoretical insights, in particular about guarantees on the solution quality, are also of interest. For this, our result on the relaxed cut optimization of BUBBLE-FOS/C might help as a starting point.

## References

1. Khaled Alzoubi, Xiang-Yang Li, Yu Wang, Peng-Jun Wan, and Ophir Frieder. Geometric spanners for wireless ad hoc networks. *IEEE Trans. Parallel Distributed Systems*, 14(4):408–421, 2003.
2. Reid Andersen and Yuval Peres. Finding sparse cuts locally using evolving sets. In *41st ACM Symposium on Theory of Computing (STOC'09)*, pages 235–244, 2009.
3. Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. A web computing environment for parallel algorithms in java. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006.
4. Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008.
5. U. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed System*, 10(7):673–693, 1999.
6. U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE Computer Society, 2007. Best Algorithms Paper Award.
7. C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, 2008.
8. G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Parallel and Distributed Computing*, 7:279–301, 1989.

9. Daniel Delling, Robert Görke, Christian Schulz, and Dorothea Wagner. Orca reduction and contraction graph clustering. In *5th Int. Conf. on Algorithmic Aspects in Information and Management (AAIM)*, pages 152–165, 2009.
10. Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Weighted graph cuts without eigenvectors: A multilevel approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(11):1944–1957, 2007.
11. R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26:1555–1581, 2000.
12. A. J. Enright, S. van Dongen, and C. A. Ouzounis. An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584, 2002.
13. M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25:619–633, 1975.
14. G. Fox, R. Williams, and P. Messina. *Parallel Computing Works!* Morgan Kaufmann, 1994.
15. Joachim Gehweiler and Henning Meyerhenke. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *Proc. 7th High-Performance Grid Computing Workshop (HGCW'10) in conjunction with 24th Intl. Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE Computer Society, 2010.
16. Joachim Gehweiler and Gunnar Schomaker. Distributed load balancing in heterogeneous peer-to-peer networks for web computing libraries. In *10th IEEE/ACM Int. Symp. on Distributed Simulation and Real Time Applications (DS-RT)*, pages 51–58, 2006.
17. Robert Görke, Tanja Hartmann, and Dorothea Wagner. Dynamic graph clustering using minimum-cut trees. In *11th Int. Symp. on Algorithms and Data Structures (WADS)*, pages 339–350, 2009.
18. Leo Grady and Eric L. Schwartz. Isoperimetric graph partitioning for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(3):469–475, 2006.
19. D. Harel and Y. Koren. On clustering using random walks. In *Proceedings of 21st Foundations of Software Technology and Theoretical Computer Science (FSTTCS'01)*, volume 2245 of *Lecture Notes in Computer Science*, pages 18–41. Springer-Verlag, 2001.
20. Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, 2000.
21. Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.
22. George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.
23. B. W. Kernighan and S. Lin. An efficient heuristic for partitioning graphs. *Bell Systems Technical Journal*, 49:291–308, 1970.
24. Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–136, 1982.
25. L. Lovász. Random walks on graphs: A survey. *Combinatorics, Paul Erdös is Eighty*, 2:1–46, 1993.
26. M. Meila and J. Shi. A random walks view of spectral segmentation. In *Eighth International Workshop on Artificial Intelligence and Statistics (AISTATS)*, 2001.
27. Henning Meyerhenke. Dynamic load balancing for parallel numerical simulations based on repartitioning with disturbed diffusion. In *Proc. Internatl. Conference on Parallel and Distributed Systems (ICPADS'09)*, pages 150–157. IEEE Computer Society, 2009.
28. Henning Meyerhenke. Beyond good shapes: Diffusion-based graph partitioning is relaxed cut optimization. In *Proc. 21st International Symposium on Algorithms and Computation (ISAAC'10)*. Springer-Verlag, 2010. To appear.

29. Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009. Best Paper Awards and Panel Summary: IPDPS 2008.
30. Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. Graph partitioning and disturbed diffusion. *Parallel Computing*, 35(10–11):544–569, 2009.
31. M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 69(2), 2004.
32. François Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *Proc. 13th International Euro-Par Conference*, volume 4641 of *Lecture Notes in Computer Science*, pages 195–204. Springer-Verlag, 2007.
33. Robert Preis. Linear time $1/2$-approximation algorithm for maximum weighted matching in general graphs. In *Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *Lecture Notes in Computer Science*, pages 259–269. Springer-Verlag, 1999.
34. Satu E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, August 2007.
35. Stefan Schamberger. *Shape Optimized Graph Partitioning*. PhD thesis, Universität Paderborn, 2006.
36. K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001.
37. 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.
38. Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.
39. J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.
40. J. Šíma and S. E. Schaeffer. On the NP-completeness of some graph cluster measures. In *Proceedings of the 32nd International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'06)*, volume 3831 of *Lecture Notes in Computer Science*, pages 530–537. Springer-Verlag, 2006.
41. Aleksandar Trifunović and William J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.*, 68(5):563–581, 2008.
42. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
43. S. van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, Univ. of Utrecht, 2000.
44. Denis Vanderstraeten, R. Keunings, and Charbel Farhat. Beyond conventional mesh partitioning algorithms and the minimum edge cut criterion: Impact on realistic applications. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing (PPSC'95)*, pages 611–614. SIAM, 1995.
45. C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.
46. C. Xu and F. C. M. Lau. *Load Balancing in Parallel Computers*. Kluwer, 1997.
47. Hongyuan Zha, Xiaofeng He, Chris H. Q. Ding, Ming Gu, and Horst D. Simon. Spectral relaxation for k-means clustering. In *Proceedings of Advances in Neural Information Processing Systems 14 (NIPS'01)*, pages 1057–1064. MIT Press, 2001.
48. Y. Zhao and G. Karypis. Clustering in the life sciences. In M. Brownstein, A. Khodursky, and D. Conniffe, editors, *Functional Genomics: Methods and Protocols*, pages 183–218. Humana Press, 2003.