A Distributed Diffusive Heuristic for Clustering a Virtual P2P Supercomputer

Joachim Gehweiler and Henning Meyerhenke
Department of Computer Science
University of Paderborn
Fürstenallee 11, D-33102 Paderborn, Germany
Email: {joge,henningm}@upb.de

***Abstract*—For the management of a virtual P2P supercomputer one is interested in subgroups of processors that can communicate with each other efficiently. The task of finding these subgroups can be formulated as a graph clustering problem, where clusters are vertex subsets that are densely connected within themselves, but sparsely connected to each other. Due to resource constraints, clustering using global knowledge (i. e., knowing (nearly) the whole input graph) might not be permissible in a P2P scenario, e. g., because collecting the data is not possible or would consume a high amount of resources. That is why we present a distributed heuristic using only limited local knowledge for clustering static and dynamic graphs.**

Based on disturbed diffusion, our algorithm DIDIC implicitly optimizes cut-related quality measures such as modularity. It thus settles between distributed clustering algorithms for other quality measures (e. g., energy efficiency in the field of ad-hoc-networking) and graph clustering algorithms optimizing cut-related measures with global knowledge. Our experiments show the promising potential of our new approach: Although each node starts with a random cluster number, may communicate only with its direct neighbors within the graph, and requires only a small amount of additional memory space, the solutions computed by DIDIC converge to clusterings that are comparable in quality to those computed by the established non-distributed graph clustering library mcl, whose main algorithm uses global knowledge.

Keywords: Graph clustering, dynamic networks, distributed algorithm.

I. INTRODUCTION

In recent years the determination of clusters within graphs has received considerable attention, e. g., for the analysis of networks like the Internet (see [19] for more references). In a graph, clusters are dense vertex subsets that are only sparsely connected to each other. This notion is intentionally vague, because it is application-dependent and hard to formalize in general. (Note that we use the terms *graph* and *network* as well as *node* and *vertex* interchangeably throughout the paper.)

The main application we consider here is the management of a peer-to-peer (P2P) based virtual distributed supercomputer for parallel computations in the *bulk-synchronous parallel* (BSP) [21] style. The BSP model gives the developer an abstract view of the technical structure and the communication features of the hardware (e. g., a parallel computer, a cluster of workstations or a set of PCs interconnected by the Internet). A *BSP program* consists of a set of *BSP processes* and a sequence of *supersteps* – time intervals bounded by a barrier synchronization. Within a superstep each process performs local computations and sends messages to other processes; afterwards it indicates, by calling the sync method, that it is ready for the barrier synchronization.

The virtual supercomputer mentioned above 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 [9].

Motivation: The load balancer in [9] 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

This work is partially supported by Integrated Project IST-15964 *AEOLUS* of the European Union and by German Research Foundation (DFG) Priority Programme 1307 *Algorithm Engineering*.

slightly improved by adding more machines, while the overhead of adding more machines starts to affect the load balancer's performance. Thus, 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 [21], 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 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. Thus, single vertices with very good network links should stay in the same cluster as their neighboring vertices.

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. Thus, our algorithm will start from a random configuration if no initial configuration is provided. Note that for our scenario with stationary computers leaving and joining the network, energy efficiency is not a concern. While our algorithm exchanges a non-neglible amount of messages, these messages are small and can be coupled to the communication of the BSP processes.

Contribution: In this paper we develop a distributed heuristic called DIDIC for the clustering problem described above and formalized in Section II. The algorithm, presented in Section IV, uses the concept of disturbed diffusion to identify dense graph regions. Since diffusion on graphs can be realized as an inherently distributed process, the nodes executing our algorithm need to communicate only with their direct neighbors. Moreover, the algorithm requires only a small amount of additional memory. In a very broad sense it can be regarded as self-stabilizing since the initial random clustering is transferred into a clustering with continuously high quality, also on dynamically changing graphs.

DIDIC fills the gap between distributed clustering algorithms for other quality measures and practical algorithms that optimize cut-related measures (most of them only in static graphs) using global knowledge, both of which are described in more detail in Section III. Our experiments (see Section V) reveal that the clusterings computed by the iterative process DIDIC converge to meaningful clusters in static and dynamic graphs. For graphs generated according to the PUB-Web scenario, the clustering results are comparable to or even slightly better than those computed by MCL, an established non-distributed algorithm.

II. The Distributed Clustering Problem

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 ω_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 $\Pi_i : V_i \to \{1,\ldots,k\}$. Such a clustering divides the vertex set V_i into k disjoint subsets $V_i = \pi_{i,1} \dot\cup \pi_{i,2} \dot\cup \ldots \dot\cup \pi_{i,k}$. Edges connecting vertices of two different clusters belong to the so-called *cut* of Π_i.

For an undirected, edge-weighted graph $G = (V, E, \omega)$ with n vertices and its k-way clustering Π (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 intra-weight$(\pi_c) := \sum_{e=\{u,v\}\in E; u,v\in\pi_c} \omega(e)$ and 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 π_c, respectively. Finally, denote the total edge weight of G by $\mathrm{ew}(G) := \sum_{e\in E} \omega(e)$. Then, the popular clustering quality measure *modularity* [16] is defined as

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

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 [16]. Unfortunately, it is $\mathcal{NP}$-hard to optimize modularity [4] for general graphs – but this hardness is true for nearly all interesting clustering metrics [20].

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

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

As argued in the introduction, 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.

III. Related Work

A. Graph Clustering

The area of graph clustering has grown very much in the last decade. That is why we restrict ourselves to highly relevant work and refer the interested reader to a recent survey [19] for a broader overview. Note that from a heuristic point of view, the optimization of a global measure is more difficult when only restricted local (and no global) knowledge is available. That is why most techniques described in this section make use of global knowledge.

Clustering without global knowledge is an important technique in mobile ad-hoc and sensor networks [5] for the improvement of certain management or communication tasks, see Yu and Chong's survey [24]. However, the objectives for clustering such networks usually differ from our goals. In case of distributed agents powered by small batteries, energy efficiency is a major target, while cut-related metrics are less important.

Diffusive processes can be used to model important transport phenomena such as heat flow, particle motion in solvents, and the spread of diseases. Computer scientists have studied diffusion in graphs as one of the major tools for balancing the load in parallel computations [23]. In such a discrete setting, diffusion is a local iterative process which exchanges splittable load entities between neighboring vertices, usually until all vertices have the same amount of load.

The use of diffusion for graph clustering is motivated by its close relation to random walks [13]. The intuitive idea both concepts have in common is that a random walk (or the related diffusion process) is likely to stay a very long time in a dense graph region before leaving it via one of the few outgoing edges. There exist many graph clustering techniques exploiting this notion (see [18] or [19]). Related to our diffusive method is the algorithm by Harel and Koren [12], 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 [8], [22] bypasses the general 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.

Görke et al. [10] develop an algorithm for clustering dynamic graphs. They use minimum-cut trees to compute a clustering and present a method to update this data structure efficiently when the graph changes. However, computing the initial tree 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. [7]. The algorithm works without an explicit optimization criterion. Its key component recursively identifies small dense regions and contracts them into single vertices. While the identification of small dense regions is a local process, the order of contractions is governed by a global priority queue.

B. Diffusive Graph Partitioning

In the area of graph partitioning, similarity measures related to diffusion have been used to compute well-shaped partitions [11], [15]. Pellegrini has developed a

faster bipartitioning mechanism [17] with a simplified diffusion process. In previous work we have extended Pellegrini's approach and have obtained a slightly different method, which is capable of direct k-way partitioning [14]. Since the diffusive concepts used in [14] are important to understand the results of this paper, we discuss them subsequently.

To assign nodes to clusters by diffusive clustering, we associate k diffusion load values with each node, one load value per cluster. These loads are distinguished by coloring them with colors from 1 to k. In each iteration the diffusive algorithm is executed for each cluster individually, with a unique kind of load for each diffusion system belonging to a cluster. Afterwards 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 π_c: First, the nodes of π_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 disturbance, leading to the concept of *disturbed diffusion*. The last row in Figure 1 shows the cluster assignment according to maximum load values. This whole procedure can be repeated to improve the solution quality. 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.

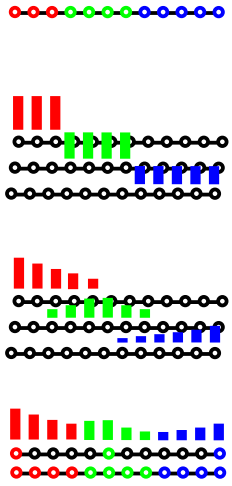


Figure 1. Schematic view of diffusive clustering.

IV. The Distributed Diffusive Clustering Algorithm DiDiC

In this section we describe our new 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 framework described in Section III-B, 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 the (re)partitioning algorithms of [14], [15], which use the above framework. 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, see Section IV-B. 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.

A. 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.

B. Eliminating Global Knowledge with Suitable Diffusive Processes

The scheme FOS/C, which acts as a diffusive similarity measure for partitioning in [15], 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 the scheme FOS/T. FOS/T is a modification of the general diffusion scheme FOS [6]. It is disturbed by stopping it after very few iterations – instead of iterating until all vertices have the same amount of load. The difference to FOS/T in [14] is the addition of the secondary load values in vector l.

Definition 1: The truncated first order diffusion scheme (FOS/T) has five parameters: a graph $G=(V,E)$, its k-way clustering Π, the cluster number c, and the load vectors $w^{(0)} \in \mathbb{R}^n$ and $l \in \mathbb{R}^n$. Let the constants $\alpha(e)$ for each edge $e \in E$ (flow scale) be suitably chosen. FOS/T performs the following operations in each iteration $0 < s \leq \psi$:

$$x^{(s-1)}_{e=\{u,v\}}(c) = \omega(e)\cdot\alpha(e)(w^{(s-1)}_u(c) - w^{(s-1)}_v(c)),$$
$$w^{(s)}_u(c) = w^{(s-1)}_u(c) + l^{(s)}_u(c) - \sum_{e=\{u,v\}\in E} x^{(s-1)}_e(c).$$

Note that in the definition of the flow $x_{\{u,v\}}$ between vertices u and v, the edge $\{u,v\}$ is directed implicitly (hence, the flow changes its sign when viewed from the other direction). Further note that references to l with an exponent s apply to the corresponding FOS/B time step $s\cdot\rho$ (see below).

In *global* matrix-vector notation one can write the final FOS/T load vector as

$$w^{(\psi)}(c) = \mathbf{M}^{\psi} w^{(0)}(c) + \sum_{s=0}^{\psi-1} \mathbf{M}^s l^{(s+1)}(c),$$

where $\mathbf{M}$ is the diffusion matrix [6] of G. Since $\mathbf{M}$ is stochastic, it can be regarded as the transition matrix of a random walk. It is well-known that the entry (i,j) of $\mathbf{M}^{\psi}$ denotes the probability of a random walk starting on node i to reach node j after ψ steps. Hence, the first part of the final load w on node u after ψ steps is the sum of products whose factors are the ψ-step transition probability and the initial load. If iterated until infinity, the first part converges to the balanced load distribution [6]. The second part, mainly determined by the secondary load l, is explored next.

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 ith cluster.

Definition 2: The first order diffusion scheme disturbed by vertex benefits (FOS/B) has four parameters: a graph $G=(V,E)$, its k-way clustering Π, the cluster number c, and the initial load vector $l^{(0)} \in \mathbb{R}^n$. Let the constants $\alpha(e)$ for edge $e \in E$ (flow scale) and B (benefit) be suitably chosen larger than 0. FOS/B performs the following operations in each iteration $0 < r \leq \rho$:

$$y^{(r-1)}_{e=\{u,v\}}(c) = \omega(e)\cdot\alpha(e)\left(\frac{l^{(r-1)}_u(c)}{b_u(c)} - \frac{l^{(r-1)}_v(c)}{b_v(c)}\right),$$
$$l^{(r)}_u(c) = l^{(r-1)}_u(c) - \sum_{e=\{u,v\}\in E} y^{(r-1)}_e(c),$$

where $y^{(r)}_{e=\{u,v\}}$ denotes the load exchange via edge e in iteration r and $b_u(c)$ denotes the benefit of vertex u in cluster π_c, which is defined as

$$b_u(c) := \begin{cases} 1 & u \notin \pi_c \\ B >> 1 & \text{otherwise} \end{cases}.$$

The benefit values $b_u(c)$ and $b_v(c)$ (one suitable value for B in experiments is 10) 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.

C. Determine Clustering

After each time step the new cluster affiliation of each vertex v can be chosen generically as $\text{argmax}_{c=1,\ldots,k} w_v(c)$ (as in line 19 of Algorithm 1). In our implementation of the algorithm, however, we use additional techniques to accelerate the clustering process. More precisely after 10 time steps we enforce that a node is assigned to the cluster with the number

$$\text{argmax}_{\{c=1,\ldots,k \,|\, \text{intdeg}(v,\pi_c)>0\}} w_v(c),$$

where $\text{intdeg}(v,\pi_c) := |\{\{u,v\} \in E \mid u \in \pi_c\}|$. Introducing the condition with the internal degree intdeg ensures that isolated vertices (those without neighbors in the same cluster) vanish promptly and convergence to a good clustering is reached faster. Note that the order of cluster number updates is not fixed and queries to neighbors can yield data from different time steps. Yet, the final results are hardly affected by this behavior.

Algorithm 1 Distributed Diffusive Clustering Algorithm

```
DiDiC(v, N(v), π, k, T, ψ, ρ) → π
if (π is undefined)
  π := RandomValue(1, k);
w := SetInitialLoad(π);
for t := 1 to T do begin
  for each cluster system c do begin
    for s := 1 to ψ do begin (* FOS/T *)
      for r := 1 to ρ do begin (* FOS/B *)
        for each neighbor u in N(v)
          l_v(c) := l_v(c) − α(e)·ω(e)·( l_v(c)/b_v(c) − l_u(c)/b_u(c) );
        end
      end
      for each neighbor u in N(v)
        w_v(c) := w_v(c) − α(e)·ω(e)·(w_v(c) − w_u(c));
      end
      w_v(c) := w_v(c) + l_v(c);
    end
  end
  π := argmax_{c=1,...,k} w_v(c);
  adaptToGraphChanges(v, N(v)) → N(v);
end
```

A further modification to eliminate isolated nodes is the following. The change of a node from one cluster to another is not only based on the load but also on the time step t. As an example, $v \in \pi_c$ changes only to $\pi_{c'}$ $(c \neq c')$ if v receives the highest amount of load w from system c' and this amount is $1+0.0001*t$ times higher than the load of system c. This way areas are only flooded by a new cluster if the diffusion process shows a really strong desire for this.

D. Discussion of the complete 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 $v \in V$. Its components have been described above. A few remarks are still necessary, though. Note that it is possible to assign an initial cluster number to v by specifying it in the parameter π. However, while the possibility exists, we assume that in our scenario it is usually not used and π 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 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. The flow scale constant $\alpha(e)$ for an edge $e = \{u,v\} \in E$ is set in our implementation as $\alpha(e) := 1/\max\{\deg(u), \deg(v)\}$. This choice avoids large amounts of load being swapped back and forth. Whenever data from a neighbor vertex is required, a request message must be sent and an answer received. Since no vertices other than neighbors are contacted by DiDiC, we hide the message calls within the data accesses. The issue of synchronicity can be enforced by message tags.

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 can be found in the upcoming experimental section. Note that we assume the input graph to change constantly. Thus, the clustering needs to be adapted constantly 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.

V. Experimental Results

In this section we present 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 and 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 distort the clustering results since the algorithm does not use the coordinate data. To create the 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.

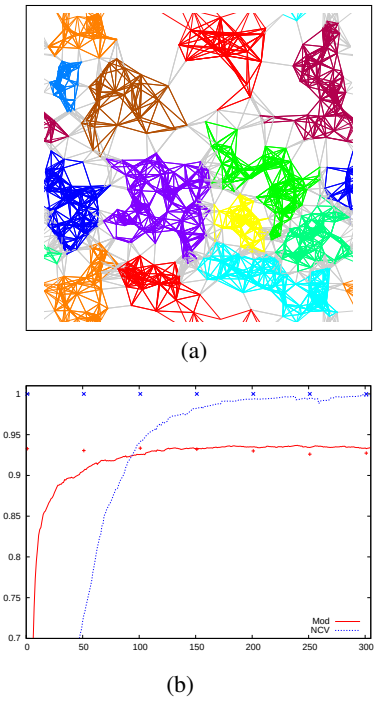
Analyzing the PUB-Web network structure for constructing a closely resembling graph class, we identify two types of computers in the PUB-Web network (which is a subset of the Internet): home users with single computers and companies participating with lots of computers. In the latter case, the computers within a company are typically well connected, whereas the Internet connection is slower; additionally, there might be medium speed connections to partner companies. Our goal is to identify subsets of the PUB-Web network that allow efficient communication and synchronization.

To resemble the real PUB-Web network as closely as possible, we have built the following test scenario: The well connected subnets in companies etc. are represented by circular dense areas of nodes. The fact 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 with computers connected to another switch. 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. This is a rather realistic assumption as well: Not only world-wide but also within several major countries there are different Internet providers; customers of the same provider share a much higher bandwidth than customers of different providers. To represent the single computers, $\frac{2}{x+2} \cdot n$ vertices are inserted uniformly at random into the unit-square. The remaining vertices are spread over all dense areas such that each area receives a fraction of $r_i^{1.5}/(\sum_{j\in 0,\ldots,x} r_j^{1.5})$, where r_i is the radius of dense area i, i. e., smaller areas have a slightly higher density.

We did extensive tests with varying parameters, e. g., with graph sizes of 1600, 2400, and 3200. In order to simulate the dynamics of a P2P network, we randomly deleted and inserted 1%, 2% or 5% 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 [22], which implements the algorithm MCL (see Section III-A). 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.



(a)

(b)

Figure 2. (a) Clustering computed with DIDIC 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. (b) Aggregated results (x-axis: time step, y-axis: modularity, NCV) for the same graph class as in (a), but for a graph with 2400 vertices.

Figure 2(a) shows a clustering obtained by DIDIC after 55 time steps for a graph with 800 vertices, 2% dynamics, and the diffusion iteration counts ψ and ρ set to 11. Figures 4 – 6 in the appendix illustrate a whole

clustering process. Plot 2(b) is based on aggregated data of two experiment series with six graphs for each plot and shows how modularity and NCV improve and converge over time for graphs of the same class with 2400 nodes. Further results are available in Figure 3 in the appendix. For validation purposes, we ran mcl every 50-th time step on the dynamic graphs and added the modularity and NCV values obtained to the plots as single points in the same colors as the corresponding lines. As one can see, once the clustering begins to stabilize after approximately 150 time steps, the clustering obtained by DIDIC has a comparable or even sightly better modularity than mcl's. Considering that each node must send roughly 100-150 very small messages in its local neighborhood per time step, a time step would only 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.

VI. CONCLUSIONS

In this paper we have presented a new 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 could be presented here) we have evaluated suitable parameter values and 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.

This promising approach should receive further exploration in future theoretical and practical work. For example, one could investigate bounds on the speed of convergence for certain graph classes and add a heuristic for splitting clusters when they grow too large. Additionally, we plan to explore if the number of message transmissions can be reduced, and if the approach is feasible for directed weighted graphs because this allows us to better model asymmetric network connections (e. g., firewalls or asymmetric up- and downlink).

REFERENCES

[1] K. Alzoubi, X.-Y. Li, Y. Wang, P.-J. Wan, and O. Frieder. Geometric spanners for wireless ad hoc networks. *IEEE Trans. Parallel Distributed Systems*, 14(4):408–421, 2003.

[2] R. Andersen and Y. Peres. Finding sparse cuts locally using evolving sets. In *41st ACM Symposium on Theory of Computing (STOC'09)*, pages 235–244, 2009.

[3] O. Bonorden, J. Gehweiler, and F. Meyer auf der Heide. A web computing environment for parallel algorithms in java. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006.

[4] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. On modularity clustering. *IEEE Trans. Knowledge and Data Engineering*, 20(2):172–188, 2008.

[5] N. Chatterjee, A. Potluri, and A. Negi. A scalable and adaptive clustering scheme for manets. In *4th Int. Conference on Distributed Computing and Internet Technology (ICDCIT'07)*, pages 73–78, 2007.

[6] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Parallel and Distrib. Comp.*, 7:279–301, 1989.

[7] D. Delling, R. Görke, C. Schulz, and D. Wagner. Orca reduction and contraction graph clustering. In *5th Int. Conf. on Algorithmic Aspects in Information and Management (AAIM)*, pages 152–165, 2009.

[8] 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.

[9] J. Gehweiler and G. 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.

[10] R. Görke, T. Hartmann, and D. Wagner. Dynamic graph clustering using minimum-cut trees. In *11th Int. Symp. on Algorithms and Data Structures (WADS)*, pages 339–350, 2009.

[11] L. Grady and E. L. Schwartz. Isoperimetric graph partitioning for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(3):469–475, 2006.

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

[13] L. Lovász. Random walks on graphs: A survey. *Combinatorics, Paul Erdös is Eighty*, 2:1–46, 1993.

[14] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. In *22nd Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–13, 2008. Best Algorithms Paper Award.

[15] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid. In *20th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, page 57 (CD), 2006.

[16] 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.

[17] F. Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *13th Int. Euro-Par Conference*, volume 4641 of *LNCS*, pages 195–204, 2007.

[18] P. Pons and M. Latapy. Computing communities in large networks using random walks. *Journal on Graph Algorithms and Applications*, 10(2):191–218, 2006.

[19] S. E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, August 2007.

[20] J. Šíma and S. E. Schaeffer. On the NP-completeness of some graph cluster measures. In *32nd Int. Conf. on Current Trends in Theory and Practice of Informatics (SOFSEM)*, volume 3831 of *LNCS*, pages 530–537, 2006.

[21] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[22] S. van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, Univ. of Utrecht, 2000.

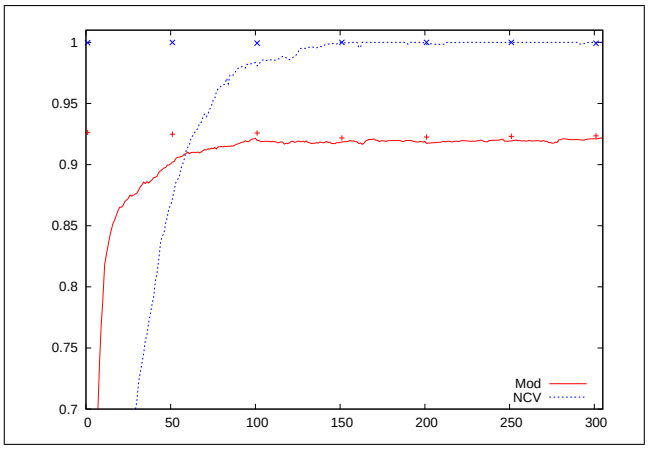
[23] C. Xu and F. C. M. Lau. *Load Balancing in Parallel Computers*. Kluwer, 1997.

[24] J. Y. Yu and P. H. J. Chong. A survey of clustering schemes for mobile ad hoc networks. *IEEE Communications Surveys & Tutorials*, 7(1), 2005.

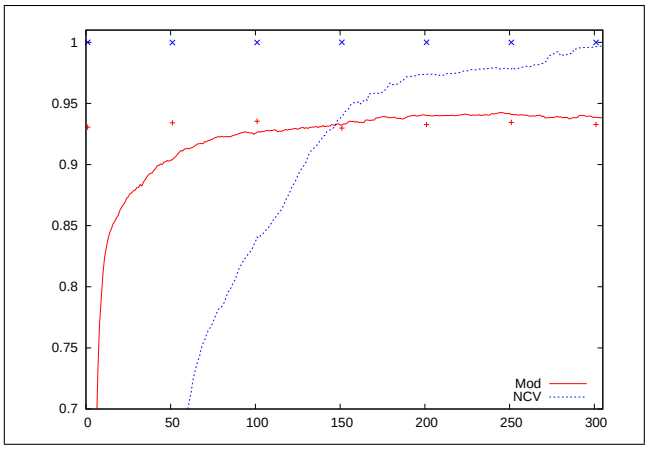
APPENDIX

A. Further Experimental Results

1) Quality Measures: Figure 3 shows the aggregated results for graphs with (a) 1600 and (b) 3200 nodes omitted in Figure 2 due to space limitations. From timestep 100 on, the modularity values are again comparable to the validation results computed with mcl. They are slightly worse in case (a) and become slightly better in case (b) later on than mcl's results. Both DIDIC clusterings certainly have a high quality and case (a) is even perfectly cluster-connected after stabilization around time step 150.



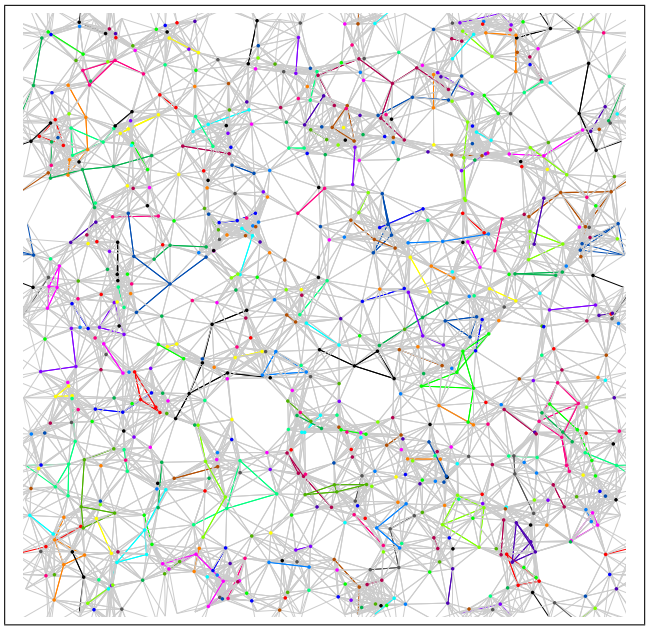
(a)



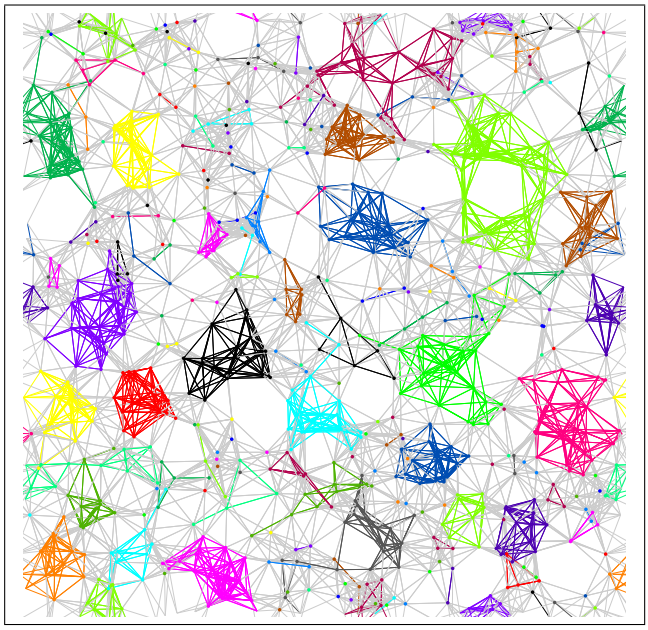
(b)

Figure 3. Aggregated results (x-axis: time steps, y-axis: modularity, NCV) for graphs with (a) 1600 nodes, and (b) 3200 nodes and parameters $k = 20$, $maxNeigh = 16$, $rad = 0.33$, $\psi = 11$, $\rho = 11$, $B = 10$, $x = 10$ (same as in Figure 2, except for the graph size).

2) Visual Impression: Figures 4(a) – 6(b) illustrate the clustering process for a generated PUB-Web P2P graph with 800 vertices, 2% dynamics, and the diffusion iteration counts ψ and ρ set to 11. Figure 4(a) shows the initial random situation. The snapshots 4(b) – 6(b) were taken at timesteps 4, 8, 10, 20, and 40. One can see that small clusters emerge quickly after only a few timesteps, and that this clustering soon converges to a state where all clusters are connected (note that the unit-square has wrap-around boundaries and cut edges are shown in grey).

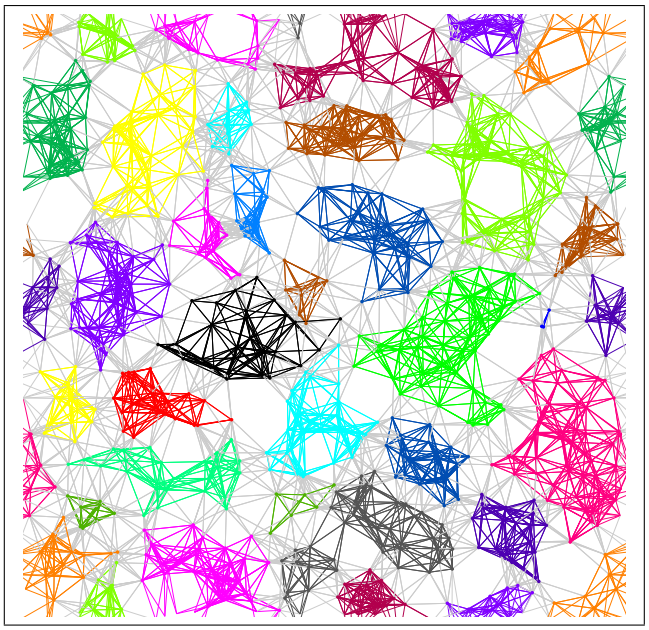


(a)

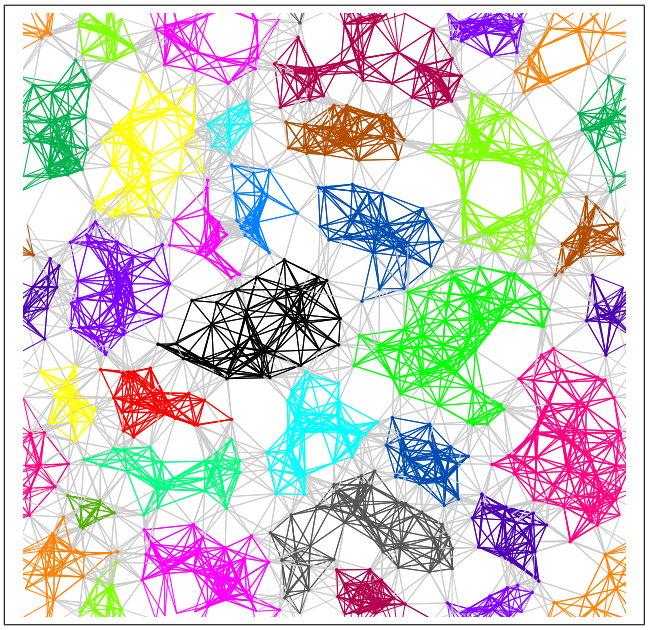


(b)

Figure 4. (a) Generated PUB-Web P2P graph with 800 nodes. (b) Clustering computed with DIDIC after timestep 4.

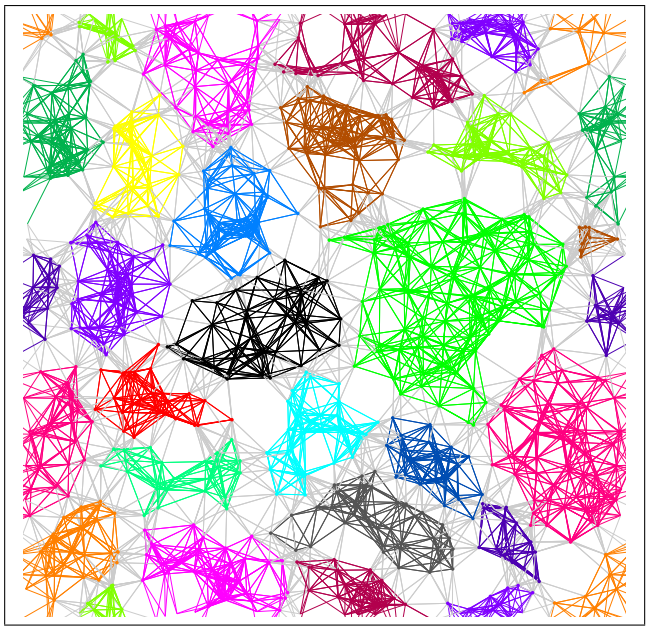


(a)

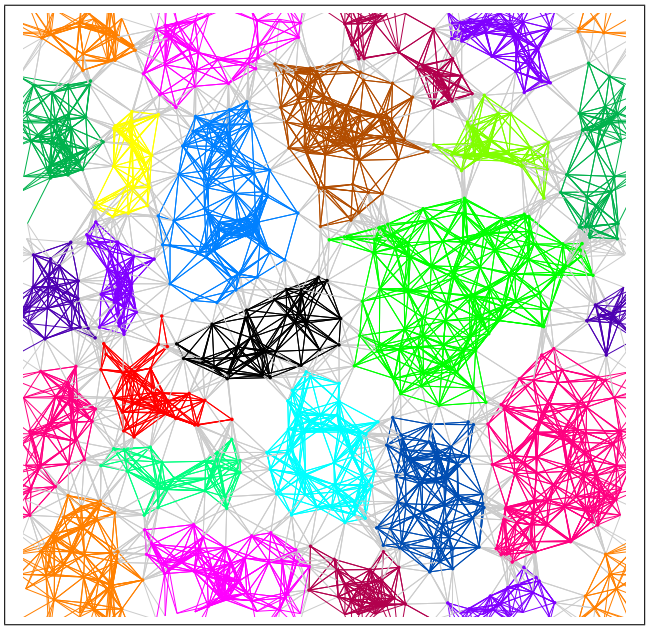


(b)

Figure 5. Clustering computed with DiDiC of a generated PUB-Web P2P graph with 800 nodes after (a) 8 and (b) 10 timesteps.



(a)



(b)

Figure 6. Clustering computed with DiDiC of a generated PUB-Web P2P graph with 800 nodes after (a) 20 and (b) 40 timesteps.