

# Graph Partitioning and Disturbed Diffusion\*

Henning Meyerhenke<sup>†</sup> and Burkhard Monien<sup>‡</sup> and Stefan Schamberger<sup>§</sup>

August 14, 2008

## Abstract

The  $\mathcal{NP}$ -hard graph partitioning problem is an important subtask in load balancing and many other applications. It requires the division of a graph's vertex set into  $P$  equally sized subsets such that some objective function is optimized. State-of-the-art libraries addressing this problem show several deficiencies: they are hard to parallelize, focus on small edge-cuts instead of few boundary vertices, and often produce disconnected partitions.

This work introduces our novel graph partitioning and repartitioning heuristic BUBBLE-FOS/C. In contrast to other libraries, BUBBLE-FOS/C does not explicitly try to minimize the edge-cut, but implicitly focuses on good partition shapes. The shapes are optimized by diffusion processes that are embedded into a learning framework. This approach incorporates a high degree of parallelism.

Besides describing the evolution process that led to the new diffusion scheme FOS/C used by BUBBLE-FOS/C, we reveal some of FOS/C's properties and propose a number of enhancements for a fast and reliable implementation. Our experiments, in which we compare sequential and parallel BUBBLE-FOS/C implementations to the state-of-the-art libraries METIS and JOSTLE, reveal that our new heuristic generates high-quality solutions.

**Keywords:** Graph partitioning, load balancing heuristic, disturbed diffusion.

---

\*This work has been partially supported by the German Research Foundation (DFG) Collaborative Research Centre SFB 376 and by DFG Research Training Group GK 693 of the Paderborn Institute for Scientific Computation (PaSCo). Parts of this paper have been published in a preliminary form in the Proceedings of the 8th International Conference on Parallel Computing Technologies, p. 263–277, September 2005, the Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, p. 57 (CD), April 2006, and the Proceedings of the 12th International Euro-Par Conference, pp. 232–242, August/September 2006.

<sup>†</sup>Corresponding author, Department of Computer Science, University of Paderborn, Fuerstenallee 11, D-33102 Paderborn, +49-5251-606695 (phone), +49-5251-606697 (fax), [henningm@uni-paderborn.de](mailto:henningm@uni-paderborn.de)

<sup>‡</sup>Department of Computer Science, University of Paderborn, Fuerstenallee 11, D-33102 Paderborn, +49-5251-606695 (phone), +49-5251-606697 (fax), [bm@uni-paderborn.de](mailto:bm@uni-paderborn.de)

<sup>§</sup>Corresponding author, Google Inc. Zurich, [schamberger@google.com](mailto:schamberger@google.com). This work was performed while the third author was affiliated with the University of Paderborn.

# 1 Introduction

The efficient usage of parallel computing resources plays a key role for many large-scale scientific simulations. Such simulations are used extensively by engineers and researchers to analyze a variety of physical processes that can be expressed via partial differential equations (PDEs). The domain on which the equations have to be solved is discretized into a mesh, for example by using the finite element method (FEM). This discretization transforms the PDEs into a sparse system of linear equations. Suitable iterative solvers for these systems typically consist of a large number of small calculations which are related via data dependencies and update the values at the mesh elements in every iteration. Since the mesh size needs often to be larger than one computer's memory to guarantee a sufficiently accurate approximation, the iterative solvers are usually parallelized and follow the single-program multiple-data (SPMD) concept.

In order to minimize the overall computation time, an efficient parallelization requires the computational tasks of the numerical application to be distributed equally among all processing nodes. Because of the dependencies between the calculations, a data distribution involves communication in the processing network. Hence, additional costs in terms of latency and bandwidth occur and have to be considered when determining the data assignment onto the processors. The calculations and their dependencies can be modeled by the vertices and edges of a graph. The objective to split this graph into equally sized parts such that the number of edges between different parts is minimal, is known as the *graph partitioning problem*. It is an  $\mathcal{NP}$ -hard problem [5] occurring also in many other applications (e.g., VLSI design [3]) as a subproblem. Despite some successes on approximation algorithms (e.g., [1, 17]), simpler heuristics are preferred in practice. These heuristics provide good solutions and are quite fast (see Section 2), so that they are widely applied in the field of parallel and distributed computing. Although great progress has been made in this area, many questions remain [8].

**Motivation.** While the global *edge-cut*, i.e., the total number of edges between different partitions, is the classical metric that most graph partitioners optimize, it is not necessarily the metric that models the real costs of an application. In FEM computations for example, the true communication volume can differ significantly from the number of cut-edges. In this case, the number of vertices situated at partition boundaries reflects the amount of information to be exchanged much more accurately. Another questionable point is the commonly applied norm. In synchronized computations the slowest processor specifies the overall speed, hence the maximum norm is appropriate, while the usually applied edge-cut is a summation norm.

The discretization mesh of many applications has to be modified during the simulation to reflect where the computations need to be more (or less) accurate for the current simulation

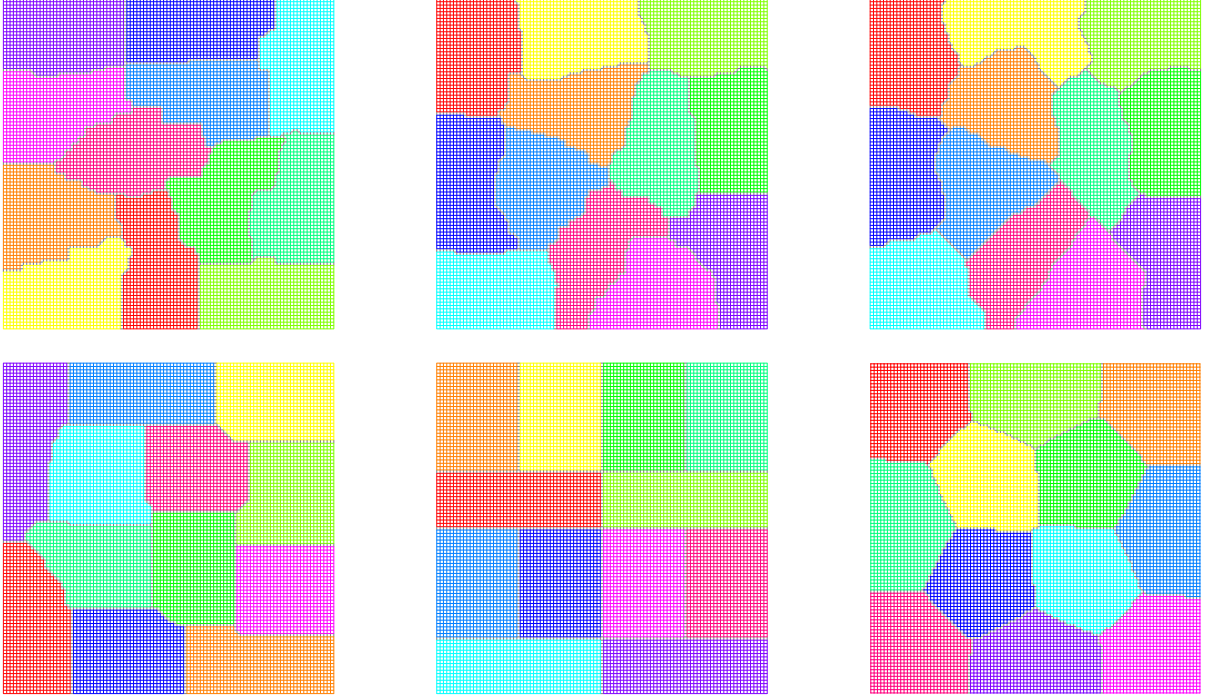


Figure 1: Dividing a grid into 12 partitions using the software libraries pMetis, kMetis, vMetis, JOSTLE, Party and the new shape optimizing approach (top left to bottom right).

state. From time to time, these dynamically changing applications require a load redistribution to ensure a balanced work load on all processors. This redistribution should guarantee both a low overhead caused by the load migration and small communication costs during the actual application. This results in a multi-objective optimization, comprising the *dynamic load balancing* as well as the graph partitioning problem. An unbalanced partition  $\Pi_t$  has to be transformed into a balanced distribution  $\Pi_{t+1}$  while obeying the two objectives.

Most of the existing implementations first determine how much load to migrate and then restrict the vertex exchange steps of the local refinement process according to this number. Better solutions can be obtained by integrating the migration costs directly into the objective function of the improvement procedure [7, 24, 32].

**Our Contribution.** In this paper we present our new graph partitioning and repartitioning heuristic algorithm BUBBLE-FOS/C. Our work extends an earlier approach also based on the shape-optimizing BUBBLE framework [2], which iteratively decreases the partitions' aspect ratios. To overcome the drawbacks of previous algorithms (see Section 3) concerning balancing and parallelization, we introduce the concept of *disturbed diffusion* in Section 4. This new concept is explored in some detail and enables the distinction of dense and sparse graph regions.

The usage of the new disturbed diffusion scheme FOS/C within the BUBBLE framework

eventually yields BUBBLE-FOS/C, which delivers solutions that differ from those of other libraries already on the first view due to the round looking domains. For example, Figure 1 displays the partitionings of different implementations for a regular grid. As one can see, the shape optimizing approach (bottom right) delivers partition shapes that are close to a circle where the graph structure allows this.

This paper is the first comprehensive treatment of BUBBLE-FOS/C, including details on additional operations such as balancing and smoothing in Section 5. Moreover, apart from the generic algorithm description, we discuss several important considerations for the sequential and parallel implementation in Section 6.

Although we cannot prove any quality bounds of our new heuristic yet, we conclude from our experiments (see Section 7) that the solutions we obtain often outperform existing state-of-the-art libraries such as METIS and JOSTLE not only regarding the shape and the number of boundary vertices, but surprisingly also to the edge-cut. Furthermore, the proposed approach is also applicable to repartition a graph and keeps the number of migrating vertices small. Last but not least, the new heuristic contains a high degree of parallelism.

## 2 Related Work

State-of-the-art graph partitioning libraries like METIS [14], JOSTLE [39] or PARTY [23] follow the multilevel scheme [9]. Vertices of the graph are contracted according to a matching and a new level consisting of a smaller graph with a similar structure is generated. This is repeated until in the lowest level only a small graph remains. The (re-)partitioning problem is then solved for this small graph; vertices in higher levels are assigned to partitions according to their representatives in the next lower level. Additionally, a local improvement heuristic is applied in every level. By exchanging vertices between partitions, it reduces the number of cut edges or the boundary size as well as balances the partition sizes. Hence, the final solution quality mainly depends on this heuristic. Implementations are usually based on the Kernighan-Lin (KL) heuristic [16] or its improvement by Fiduccia and Mattheyses [3], while the local refinement in PARTY is derived from theoretical analysis with Helpful-Sets (HS) [11].

To address the load balancing problem during parallel computations, distributed versions of the libraries METIS and JOSTLE have been developed. Both of them apply about the same multilevel techniques as their single processor version, but special attention must be paid to the local improvement heuristic due to its sequential nature. As an example, a coloring of the graph's vertices ensures in the parallel library PARMETIS [31] that during the KL refinement no two neighboring vertices change their partition simultaneously and therefore destroy the consistency



of the data structures. In contrast to METIS, where vertices stay on their partition until a new distribution has been computed, the parallel version of JOSTLE [40] maps each subdomain to a single processor, and vertices which migrate do so already during the computation of the repartitioning. The HS heuristic in PARTY exchanges sets between partitions that sometimes contain a large number of vertices. Hence, even more overhead would be necessary to ensure data consistency in a parallel implementation.

Recently, Pellegrini [25] has addressed some drawbacks of the KL heuristic. His approach aims at improved partition shapes, based on a diffusive mechanism used together with KL improvement. For the diffusion process the algorithm replaces whole partition regions not close to partition boundaries by one super-node. This reduces the number of diffusive operations and results in an acceptable overall speed. The implementation described is only capable of recursive bisection, which yields in general inferior results than direct  $P$ -way methods [33]. Meyerhenke et al. [20] extends Pellegrini's ideas to a direct  $P$ -way method that uses some diffusive partitioning concepts presented in this paper.

Another method to partition a mesh is based on geometric reasoning, more specifically on space-filling curves [43]. The vertices of the graph are sorted by a certain recursive scheme covering the whole domain. Then, the linear array of vertices constructed this way is split into equally sized parts. This method only works if vertex coordinates are present. It is extremely fast, but problems arise if the simulation area contains holes [30] since only the provided (and sometimes misleading) geometric information is used, but the structural data is ignored.

One approach to optimize more realistic metrics instead of the edge-cut is undertaken in Diekmann et al. [2]. Since the convergence rate of the CGBI solver in the PadFEM environment depends on the geometric shape of a partition, the integrated load balancer iteratively decreases the aspect ratios by applying the algorithm BUBBLE, whose basic idea appeared already in [42]. Yet, its previous implementations contain strictly sequential parts and suffer from some other difficulties. Details about this algorithm, the drawbacks of previous implementations and how to overcome them are discussed in the following.

### 3 The Bubble Framework and Its Previous Implementations

The BUBBLE Framework [2] has evolved from simple greedy algorithms computing bisections of graphs. It is related to Lloyd's  $k$ -means clustering algorithm [18], but in contrast to graphs the latter operates on points in  $n$ -dimensional space and their implicitly given Euclidean distances. Starting with an initial, often randomly chosen vertex (seed) for each of the  $P$  partitions, all subdomains are grown simultaneously by assigning each vertex to the partition of its nearest

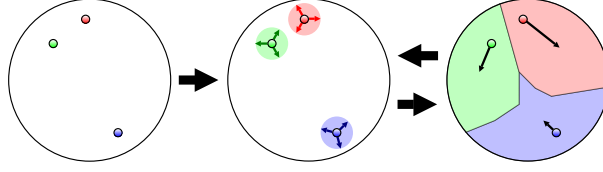


Figure 2: The three operations of the learning BUBBLE framework: Init: Determination of initial seeds for each partition (left). Grow: Growing around the seeds (middle). Move: Movement of the seeds to the partition centers (right).

seed. This assignment process can be seen as soap bubbles which grow around the seeds and form common borders where they collide. After the whole graph has been covered and all vertices of the graph have been assigned this way, each partition computes its new center, which acts as the seed in the next iteration. To improve the solution, this process is repeated several times or even until a stable state with hardly moving seeds is reached. Summarized, the BUBBLE algorithm mainly consists of the following three phases that are also illustrated in Figure 2:

**Init** One initial seed vertex for each partition is determined.

**Grow** Starting from their seeds, all vertices are assigned to the partition of the nearest seed, e. g., by a breadth-first mechanism that proceeds until all vertices are assigned.

**Move** All partitions determine a new center vertex. These vertices become the seeds in the next iteration.

These three operations can be implemented in several ways. We first summarize existing implementations before introducing the new diffusion-based method.

To our knowledge and apart from simple greedy heuristics there are two follow-up implementations of the BUBBLE idea [42] that apply the BUBBLE framework to solve FEM graph partitioning problems. The first one is part of a former version of the PARTY graph partitioning library. There, the implementation of the three phases can be described roughly as follows. After selecting initial seeds randomly, a breadth-first search (BFS) algorithm is started from every seed. During this BFS process the partitions alternately acquire one of their free neighbor vertices until all vertices are assigned. Then, in each partition the vertex with the minimal maximal distance to all other vertices of the same partition becomes the new seed.

This approach shows several problems. The initial placement of the partitions may be very bad, requiring many iterations, until it is fixed. Even then, the partition sizes usually vary extremely and the partition quality is not considered at all. Another important disadvantage is that the growing phase cannot be parallelized because vertices are assigned in a serial manner and earlier assignments have a large impact on later decisions.

A second approach is implemented in a former version of the FEM simulation tool Pad-FEM [2]. There, the first initial seed is chosen randomly among the vertices with smallest degree. Then, to determine the seed for the next partition, a breadth-first search is performed with all already chosen seeds as starting points. The last vertex found becomes the seed for the next partition. This is repeated until all seeds have been determined. In the growing process the smallest partition with at least one adjacent unassigned vertex grabs the vertex with the smallest Euclidean distance to its seed. Within the `move` operation the new seed of a partition becomes the vertex for which the sum of Euclidean distances to all other vertices of the same partition is minimal. To find this vertex quickly, some approximation is used.

This algorithm solves some of the problems we have seen in the first approach. The initial seeds are distributed more evenly over the graph. Since the smallest possible partition gathers the next vertex, more attention is paid to the balance, and also the determination of the center has been improved to work faster. By including coordinates in the choice of the next vertex, the partitions are usually also geometrically well shaped (and connected), which is the main goal of this approach. Other quality metrics are not considered. By relying on vertex coordinates, this approach is only applicable if these are provided. Furthermore, the Euclidean distance might not coincide at all with the path length between vertices. This can often be observed if an FEM mesh contains holes or fissures, in which case a partition may be placed around it – just like with space-filling curve partitioning. Experiments also reveal that the selection mechanism, although improved by preferring under-weighted partitions, still does not lead to sufficiently well balanced domains [2]. Concerning a possible parallelization, the situation stays the same as described before because the selection process of the vertices is still strictly serial.

## 4 Diffusion-based Mechanisms for Bubble

This section describes how we integrate diffusion into the BUBBLE framework. The main idea is based on the observation that load diffuses faster into densely connected regions of the graph rather than into sparsely connected ones. Following this observation, we expect to identify sets of vertices that possess a high number of internal and a small number of external edges.

To assign vertices to partitions, we execute the diffusion algorithm exactly  $P$  times, with a unique kind of load for each partition. These loads are distinguished by coloring them with colors from 1 to  $P$ . The implementation of the operations `Grow` and `Move` can be described as:

**Grow** Independently for each partition, load is placed on the center vertices and distributed by a diffusion scheme. Each vertex is assigned to the partition from which it has received the highest load amount.

**Move** The vertex with the highest load of color  $p$  becomes the new seed vertex of partition  $p$ .

Most reasonable non-balanced load distributions based on diffusion would cause the former seed of partition  $p$  to receive the highest load of color  $p$ , too. Hence, the centers and therefore the partitions do not move and no learning of a better partition placement occurs. To overcome this problem, we introduce an alternative initial load placement for the diffusion process. Instead of only placing load on the single center vertex, we distribute it evenly among all vertices of a partition. By doing so, we obtain three different combinations of initial load placements and resulting vertex assignments:

**Grow/Assignment** Place load on the center vertex, perform the diffusion process, and assign the vertices according to the highest load amount to obtain a partitioning.

**Grow/Consolidation** Place load evenly distributed on all vertices of the partition, perform the diffusion process, and assign the vertices according to the highest load amount to obtain a partitioning.

**Move/Centers** Place load evenly distributed on all vertices of the partition, perform the diffusion process, and for each partition select the vertex with the highest load as the new center.

The three combinations are also illustrated in Figure 3 for a path graph and  $P = 3$ . On the left hand side, load is placed on the center vertices and the diffusion process leads to a partitioning, while placing load on the partitions can either be used to improve a partitioning or to determine new center vertices as shown on the right. Note that we denote the vertices which we place load on as set of source vertices. Depending on the operation, this set contains either the single center vertex or all vertices of a partition.

The operations **Grow/Assignment** and **Move/Centers** are required for the BUBBLE Framework, **Grow/Consolidation** is optional and can be applied in between, even several times. Our experiments show that multiple invocations often improve the solution quality. Apparently, this follows from an implicit balancing performed by **Grow/Consolidation** due to the higher initial load on vertices of smaller partitions.

The total load amount  $W$  that we place into the system is set to the number of vertices  $|V|$  of the graph. Note that the choice of  $W$  only scales the final load distribution and therefore has no real impact on the solution, as long as it is the same value for all partitions. When performing a **Grow/Consolidation** or **Move/Centers** operation, we evenly distribute that load on all source vertices. Hence, vertices of smaller partitions will contain more load to be spread

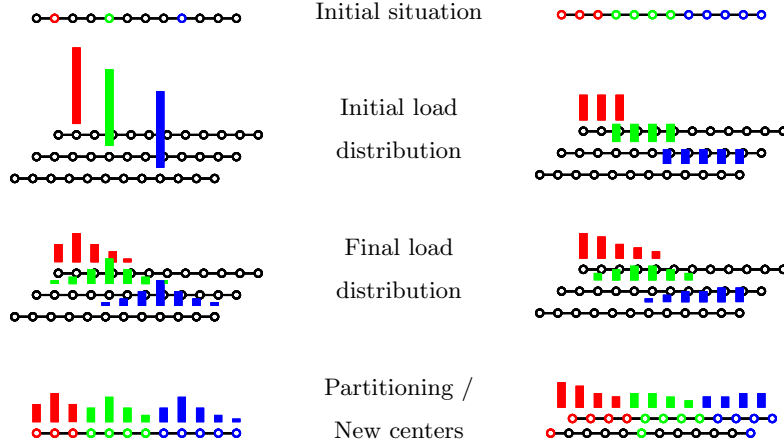


Figure 3: Schematic view: Placing load on single vertices (left) or a partition (right), the diffusion process and the assignment of the vertices to the partitions (left and right) or determination of the centers (right) according to the load.

into the graph, which supports balancing. Adjusting the load amount per partition is a way to balance the partition sizes and is discussed later.

For the **Init** phase, several alternatives for computing initial centers are possible. In case a partitioning is part of the input, we simply perform a **Move/Centers** operation.

As mentioned, on the one hand a random distribution may place vertices suboptimally, but on the other hand it is more likely that vertices in dense regions are chosen. In case of a repartitioning, it is possible to perform a **Move/Centers** operation to obtain centers. This can also be done after vertices have been assigned to a partition randomly. Currently, we have implemented the following choices:

**Init/Random Center** Randomly determine  $P$  single disjoint vertices as partition centers.

**Init/Repartition** Perform **Move/Centers** on a given partitioning to obtain the center vertices.

**Init/Random Partition** Compute a balanced random partitioning and proceed with **Init/Repartition**.

Inserting the proposed diffusive operations into the **BUBBLE Framework**, results in the generic algorithm sketched in Figure 4. The input consists of the graph  $G$  and the parameters  $i$  and  $l$ , which specify the number of the different iterations to be performed.

The outer iteration (line 01) starts with determining single center vertices for each of the partitions. If no partitioning  $\Pi$  is present, it either selects random vertices (line 03) or computes a random vertex assignment to partitions (line 05). Now, in case of an existing partitioning, the center vertices are then obtained by performing **Move/Centers**, consisting of the initial load distribution (line 09), the diffusion process (line 10) and the center determination (line 11).

```

00  Algorithm DiffusiveBubble( $G, i, l$ )
01      in each iteration  $i$ 
02          if not  $\Pi$  exists
03              case random-center
04                   $c = \text{determine-random-centers}(G)$ 
05              case random-partition
06                   $\Pi = \text{random-partitioning}(G)$ 
07          if  $\Pi$  exists
08              parallel for each partition  $p$ 
09                   $w_p = \text{distribute-load-on-parts}(G, p, \Pi)$ 
10                   $w_p = \text{diffusion}(G, p, w_p)$ 
11                   $c = \text{determine-centers}(G, w)$ 
12          in each loop  $l$ 
13              parallel for each partition  $p$ 
14                   $w_p = \text{distribute-load-on-sources}(G, p, S)$ 
15                   $w_p = \text{diffusion}(G, p, w_p)$ 
16                   $\Pi = \text{determine-partitioning}(G, w)$ 
17      return  $\Pi$ 

```

Figure 4: Sketch of the diffusive BUBBLE algorithm.

The following inner loop (line 12), which has to be executed at least once, contains the **Grow/Assignment** and **Grow/Consolidation** operation. Note that for each partition the set of source vertices  $S$  contains either the single center vertex or all the partition's vertices, depending on the previous operation. The load  $w_p$  is distributed evenly among the source vertices (line 14), distributed via the diffusion process (line 15), and finally the vertices are assigned to obtain a partitioning  $\Pi$  (line 16).

The sketched algorithm mainly contains a collection of loops. Of those, the loops over the partitions (lines 08 and 13) and the diffusion operations can be fully parallelized. Also the vertex assignment is a distributed operation, only the maximum computations during **Move/Centers** require a global view on the whole partition. Another interesting point is that the presented algorithm does not contain any explicit objectives. These are hidden within the diffusion processes of the growth and movement operations.

The most important and most costly part of the algorithm is the calculation of the diffusion. To be applicable, the solution of the applied scheme must possess two properties. As already



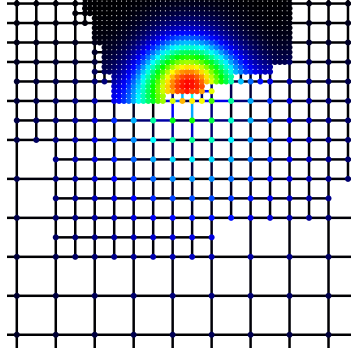


Figure 5: Load distribution originated at a single seed after 50 FOS iterations. Vertices with high load are colored red, empty vertices are black.

mentioned, it is necessary for the learning framework that load spreads faster into densely connected regions of the graph. Traditional diffusion schemes fulfill this constraint because they are known to compute a  $\|\cdot\|_2$ -minimal flow [12]. However, the load must be distributed in a hill-like manner with its peak close to the originating center for two reasons. First, the old assignment of vertices to partitions should not be altered too much to keep migration costs low. The second reason is to obtain connected subdomains, which is very desirable. Figure 5 gives an example of such a distribution. Although the vertices in the lower part are closer to the seed vertex, much more load reaches the vertices inside the denser upper part than those in the lower one since there are more paths into this region.

Traditional diffusion schemes used for load balancing converge towards a completely balanced load distribution. Since this contradicts the requested hill-like load distribution, they are not directly applicable and have to be modified. Three possible ways are described and discussed in the following. In an evolutionary process, the first two have been superseded by FOS/C. We therefore mainly focus on this latter scheme.

#### 4.1 The Limited First Order Scheme

One idea to get a hill-like load distribution is to stop FOS before it converges. This means that we perform some FOS iterations until we abort the iteration and use the current load distribution as the final result. We call this approach *Limited First Order Scheme* (FOS/L).

Although FOS/L is able to yield good experimental results in principle already, one major problem remains: Determining the number of FOS iterations until the diffusion process is interrupted. This number is difficult to choose since it depends on the graph, the number of partitions, as well as on the current partition placement. If too few steps are performed, the load has not spread widely enough and vertices may remain completely empty, while too many iterations equalize the load such that it is no longer possible to determine a good partitioning.

Since we are unable to state a general rule to obtain a good number, we have selected some appropriate values by hand for our experiments, often determined in a large number of runs. Of course, this is hardly applicable in practice. Thus, although FOS/L serves as a working proof of concept, a more reliable diffusion process is required.

## 4.2 The First Order Scheme with Absolute Drain

In contrast to the original First Order Scheme, the FOS/A diffusion scheme (A for absolute drain) does not converge towards the completely balanced load situation. Instead, it is disturbed such that the converged solution has similar properties as the load distribution after some FOS iterations described in the last subsection. This eliminates the problem of finding a suitable number of FOS/L iterations since one can always iterate until convergence.

The disturbance is realized by decreasing the load on each vertex by (at most) an absolute value  $\delta_A$  after each diffusive load exchange. If a vertex contains less load than  $\delta_A$ , only the existing load is subtracted, and therefore all load values remain non-negative. To keep the total load amount in the system constant, all subtracted load is added equally distributed onto the set of source vertices  $\emptyset \neq S \subseteq V$ . As before, for each partition this set contains either the single center vertex or all vertices from that partition.

The behavior of FOS/A on unstructured FEM graphs can be sketched as follows. During the first iteration, all load is placed on the source vertices. Then, it diffuses into the graph, similar to the original First Order Scheme, because the subtracted load is relatively small compared to the amount on a vertex. While the load continues to spread, more and more vertices are reached so that the total amount of subtracted load  $\Delta$  increases. At some point, however, the furthest vertices only obtain less than  $\delta_A$  load, which is completely subtracted and sent back to the sources. Hence, nothing remains on these vertices, the spreading slows down and eventually stops. Yet, the convergence is not smooth. In the beginning, the load on the seed vertices is much higher than  $\delta_A$ , hence it spreads from them into the graph in all directions. At some point, it might reach an area with a different structure, so that the distribution speed changes. Hence, there occur wave-like effects in the distribution until every reachable part of the graph is 'known' by the diffusion process and the load has been placed accordingly.

Although all our experiments show that the FOS/A scheme converges, this could not be proven yet. Finding the iteration from which on the sets of vertices  $v$  with load  $w_v < \delta_A$  and with load  $w_v \geq \delta_A$  become stable, turns out to be an important unsolved question. Since also the running time of FOS/A is extremely high due to an even slower convergence than undisturbed diffusion schemes, we have developed another disturbed diffusion scheme, which shows the positive properties of FOS/A, but not its negative ones.

### 4.3 The First Order Scheme with Constant Drain

Now we introduce the FOS/C diffusion scheme, where C stands for constant drain. Like FOS/A, this scheme is also based on FOS and disturbed in every iteration. However, in contrast to FOS/A, the disturbance is not restricted to the vertices containing load, but performed on all vertices. Hence, negative loads become possible. In the following we assume all graphs under consideration to be connected and undirected.

The FOS/C scheme executes two operations in each iteration. While the first one is the original diffusive load exchange, the second step introduces the disturbance by shifting a small load amount  $\delta > 0$  from all vertices of the graph to the set of selected source vertices  $\emptyset \neq S \subset V$ . This disturbance can be described by the *drain vector*  $d \in \mathbb{R}^n$ , which is defined as

$$d_v = \begin{cases} \delta \cdot |V|/|S| - \delta & : v \in S \\ -\delta & : \text{otherwise} \end{cases}$$

This vector is added to the load vector resulting from the original diffusion step. Note that, since  $\langle d, \mathbf{1} \rangle = 0$ , this does not change the total amount of system load. The new scheme is formally described in the following definition.

**Definition 1** (FOS/C). *Given a connected graph  $G = (V, E)$  and a suitable constant  $\alpha$ . Let  $\delta > 0$  be the drain constant,  $\emptyset \neq S \subset V$  the set of source vertices, and  $d$  the corresponding drain vector. In iteration  $k$ ,  $w_v^{(k)}$  denotes the load on vertex  $v$  and  $x_e^{(k)}$  the exchange over edge  $e$ . Let  $w^{(0)}$  represent the initial load situation. In each iteration  $k$ , the FOS/C scheme performs the computations:*

$$\begin{aligned} x_{e=(u,v)}^{(k)} &= \alpha \cdot (w_u^{(k)} - w_v^{(k)}) \\ w_v^{(k+1)} &= w_v^{(k)} - \left( \sum_{e=(v,*)} x_e^{(k)} \right) + d_v. \end{aligned}$$

In matrix notation FOS/C can be written as

$$w^{(k+1)} = \mathbf{M}w^{(k)} + d,$$

where  $\mathbf{M} = \mathbf{I} - \alpha \mathbf{L}$  is the doubly-stochastic diffusion matrix also used for FOS ( $\mathbf{L}$  is the Laplacian matrix matrix of  $G$ ).  $\mathbf{L}$  can be expressed as  $\mathbf{L} = \mathbf{A}\mathbf{A}^T$ , where  $\mathbf{A}$  is the vertex-edge incidence matrix of  $G$ . Note that  $\mathbf{L}$  does not have full rank.

In general, the existence of a solution of a linear system involving a Laplacian matrix  $\mathbf{L}$  depends on the right hand side of the linear equation. The equation  $\mathbf{L}x = b$  has a solution (and then infinitely many), iff  $b \perp \mathbf{1}$ . The next lemma states that the  $\|\cdot\|_2$ -minimal balancing flow can be computed by solving a system of linear equations.

**Lemma 1** ([12]). *Consider the quadratic flow minimization problem*

$$\min! \|f\|_2 \text{ with respect to } \mathbf{A}f = b$$

*Provided that  $b \perp \mathbf{1}$ , the solution to this problem is given by*

$$f = \mathbf{A}^T x, \text{ where } \mathbf{L}x = b$$

To prove the convergence of FOS/C, we require the upcoming lemma and the notion of the Moore-Penrose pseudoinverse of a matrix [6, p. 257f.]. Like  $\mathbf{L}$ , its pseudoinverse  $\mathbf{L}^\dagger$  is symmetric positive semidefinite, and  $w = \mathbf{L}^\dagger d$  is the unique solution to the least square problem  $\min_{w \in \mathbb{R}^n} \|d - \mathbf{L}w\|_2$ . If  $(\lambda_i \neq 0, z_i)$  is the  $i$ -th pair of eigenvalues/-vectors of  $\mathbf{L}$ ,  $(\lambda_i^{-1}, z_i)$  is the analogous  $i$ -th pair of  $\mathbf{L}^\dagger$  (as in the case of nonsingular matrices). All pairs  $(\lambda_i = 0, z_i)$  are eigenvalues/-vectors of both  $\mathbf{L}$  and  $\mathbf{L}^\dagger$  (comp. [4]).

**Lemma 2.** *Let  $\mathbf{M} = \mathbf{I} - \alpha\mathbf{L}$  be a diffusion matrix,  $\mathbf{L}^\dagger$  the Moore-Penrose pseudoinverse of  $\mathbf{L}$ , and  $b$  be a vector perpendicular to  $\mathbf{1}$ . Then,*

$$\lim_{k \rightarrow \infty} (\mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \cdots + \mathbf{M}^k)b = (\mathbf{I} - \mathbf{M})^{-1}b = \alpha^{-1}\mathbf{L}^\dagger b$$

*Proof.* The vector  $\mathbf{1} = (1, \dots, 1)^T$  is an eigenvector to the simple eigenvalue 1 of  $\mathbf{M}$ . Since  $b \perp \mathbf{1}$ , i. e.,  $\sum_{j=1}^n b_j = 0$ , it follows that  $\lim_{k \rightarrow \infty} \mathbf{M}^{k+1}b = 0$ . Hence,

$$\begin{aligned} & \lim_{k \rightarrow \infty} (\mathbf{I} - \mathbf{M})(\mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \cdots + \mathbf{M}^k)b \\ &= \lim_{k \rightarrow \infty} (\mathbf{I} - \mathbf{M}^{k+1})b = \lim_{k \rightarrow \infty} b - \mathbf{M}^{k+1}b \\ &= b \end{aligned}$$

Therefore, when multiplied with any vector  $b$  perpendicular to  $\mathbf{1}$ ,  $(\mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \cdots + \mathbf{M}^k)$  is the inverse to  $(\mathbf{I} - \mathbf{M}) = \alpha\mathbf{L}$  for  $k \rightarrow \infty$ . Next we prove the last equality of the claim.

The vector  $\mathbf{1}$  is the eigenvector corresponding to the smallest eigenvalue  $\lambda_1 = 0$  of  $\mathbf{L}$ . Since  $\mathbf{L}$  is a real symmetric matrix, its eigenvectors form a basis of  $\mathbb{R}^n$  [37, Ch. 24]. Hence, we can represent  $b$  as a linear combination of the eigenvectors  $z_j$  of  $\mathbf{L}$ :  $b = \sum_{j=1}^n a_j z_j$  with  $a_j \in \mathbb{R}$ . The property  $b \perp \mathbf{1} = z_1$  can be written as

$$0 = \langle b, z_1 \rangle = \sum_{i=1}^n b_i \cdot [z_1]_i = \sum_{i=1}^n \sum_{j=1}^n a_j [z_j]_i \cdot [z_1]_i = \sum_{j=1}^n a_j \langle z_j, z_1 \rangle.$$

Since all the eigenvectors are orthogonal to each other, we have:  $\sum_{j=1}^n a_j \langle z_j, z_1 \rangle = a_1 \langle z_1, z_1 \rangle$ . As  $\langle z_1, z_1 \rangle > 0$ , the coefficient  $a_1$  must be zero and  $(\alpha\mathbf{L})^{-1}b = \alpha^{-1} \sum_{j=2}^n a_j \lambda_j^{-1} z_j = \alpha^{-1} \mathbf{L}^\dagger b$ .  $\square$

Now, we are able to state the following theorem.

**Theorem 1** (Convergence of FOS/C). *The FOS/C scheme converges for any arbitrary initial load vector  $w^{(0)}$ .*

*Proof.* Repeatedly applying the diffusion matrix to  $w^{(0)}$ , we obtain

$$\begin{aligned} w^{(1)} &= \mathbf{M}w^{(0)} + d \\ w^{(2)} &= \mathbf{M}w^{(1)} + d = \mathbf{M}(\mathbf{M}w^{(0)} + d) + d \\ &= \mathbf{M}^2w^{(0)} + (\mathbf{M} + \mathbf{I})d \\ &\vdots \\ w^{(k)} &= \mathbf{M}^k w^{(0)} + (\mathbf{M}^{k-1} + \dots + \mathbf{M} + \mathbf{I})d. \end{aligned}$$

Due to Lemma 2, this yields

$$\begin{aligned} w^{(\infty)} &= \lim_{k \rightarrow \infty} \mathbf{M}^k w^{(0)} + (\mathbf{I} - \mathbf{M})^{-1}d \\ &= \lim_{k \rightarrow \infty} \mathbf{M}^k w^{(0)} + \alpha^{-1} \mathbf{L}^\dagger d \end{aligned}$$

□

The first part of the convergence load  $w^{(\infty)}$ ,  $\lim_{k \rightarrow \infty} \mathbf{M}^k w^{(0)}$ , is the evenly balanced load  $\bar{w}$  computed by FOS. Hence, the load differences only depend on  $\mathbf{L}^\dagger$  and  $d$ . Unfortunately, computing  $\mathbf{L}^\dagger$  is in general very time- and space-consuming (both quadratic in the number of unknowns). Yet, the solution of the disturbed diffusion scheme FOS/C can also be determined by solving a system of linear equations, for which faster methods can be used.

**Corollary 1.** *When converged, the load  $w^{(\infty)}$  of FOS/C can be characterized as:*

$$\begin{aligned} w^{(\infty)} &= \mathbf{M}w^{(\infty)} + d \\ \Leftrightarrow (\mathbf{I} - \mathbf{M})w^{(\infty)} &= d \\ \Leftrightarrow \alpha \mathbf{L}w^{(\infty)} &= d \\ \Leftrightarrow \mathbf{L}w^{(\infty)} &= d/\alpha \end{aligned}$$

Hence, the load distribution of FOS/C in its converged state can be determined by solving the system of linear equations  $\mathbf{L}w^{(\infty)} = d/\alpha$ .

For our purpose, the scaling factor  $\alpha$  can be omitted. Due to the rank deficiency of the Laplacian matrix, the solution  $\mathbf{L}w = d$  is not unique. It contains some multiple of the vector  $\mathbf{1}$ , reflecting the total load in the system. Hence, by solving the linear system, this constant cannot be determined, meaning that only the load differences between the vertices are unique.

In fact, in the converged state of FOS/C, all load that is moved onto the source vertices via the disturbance described by  $d$  has to be sent back in one iteration step. Hence, according to Lemma 1, the following corollary can be made.

**Corollary 2.** *The load differences  $f = \mathbf{A}^T w^{(\infty)}$  in the converged state of the disturbed diffusion scheme FOS/C equal the  $\|\cdot\|_2$ -minimal flow  $f$  that balances the load vector  $d/\alpha$ , sending from the vertices in  $S$  the load  $\delta$  to every vertex in the graph:  $\mathbf{A}f = \mathbf{A}\mathbf{A}^T w^{(\infty)} = \mathbf{L}w^{(\infty)} = d/\alpha$ .*

The reformulations of the original problem make it possible to use well-known methods to calculate the sought-after  $\|\cdot\|_2$ -minimal flow. Among these methods are the genuine diffusion schemes FOS and SOS or solvers for systems of linear equations like multigrid algorithms or Conjugate Gradient (CG). Once the balancing flow has been determined, the vertex loads  $w^{(\infty)}$  of FOS/C can be assigned accordingly with a suitable chosen constant, e. g., by a normalization s. t.  $\sum_{v \in V} w_v^{(\infty)} = 0$ . This also ensures that the load distributions computed for each partition have a common reference point and are therefore comparable. An alternative way, which generalizes this method and additionally improves the numerical stability, is discussed next.

#### 4.4 Numerical Stability by Influence Range Reduction

Recall that, for the linear system  $\mathbf{L}w^{(\infty)} = d$  to have a solution, it is necessary that  $d \perp \mathbf{1}$ . Due to the limited numerical precision of computers, this condition (or similar ones) may not be fulfilled exactly during the course of a numerical solver. Resulting errors may then cause a solver to converge very slowly or even to diverge. In the following, we present a modification of the linear system such that the involved matrix becomes positive definite instead of semidefinite, which eases numerical problems.

To understand the meaning of the altered linear system, we first generalize the flow problem to solve. We construct a new graph  $G_\phi$  that is composed of the graph  $G$  and an extra vertex  $x$  connected with every other vertex of  $G$ . All edges  $e \in E$  of  $G$  keep their weights (if  $G$  is unweighted, they get a weight of 1), while the capacity of the edges incident to  $x$  are set to some constant  $\phi > 0$ . Again, we place load equally onto the set of source vertices  $S$ . Yet, now we solve the flow problem from Corollary 2 with the extra vertex  $x$  as sink. Since we minimize  $f$  according to the  $\|\cdot\|_2$ -norm, the load will not be sent directly to  $x$ , but also makes some 'detours' via other vertices in  $G$ .

The weight constant  $\phi$  determines the spread of the flow. If  $\phi$  is large, it is 'cheaper' to send most load directly to  $x$ , while if  $\phi$  is small, the costs of the 'detour' into the graph are compensated by less utilized edges incident to  $x$ . There are two extreme cases: if  $\phi \rightarrow \infty$ , all load is sent directly to  $x$ , while if  $\phi \rightarrow 0$ , the  $\|\cdot\|_2$ -minimal flow will converge towards the





Figure 6: Flow towards  $x$  in case of a small (left) and large (right) parameter  $\phi$ .

solution obtained previously for the unmodified graph.

The modification weakens the influence of vertices that are far away from the set of source vertices and therefore not of interest to a partition. Of course, all edges still carry some load since we compute the  $\|\cdot\|_2$ -minimal flow. However, due to the limited computational accuracy, the vertices distant to the sources nodes remain empty in practice as sketched in Figure 6, displaying the flow over the extra edges depending on the choice of  $\phi$ .

Formally, let  $G = (V, E)$  be an undirected, connected graph. If we extend  $G$  by an additional vertex  $x$  and connect it to every other vertex with an edge of weight  $\phi$ , we obtain the graph  $G_\phi = (V \cup \{x\}, E \cup \{\{v, x\} : v \in V\})$  with edge weights  $c_e = 1 \ \forall e \in E$  and  $c_{\{v, x\}} = \phi \ \forall v \in V$ . The weighted Laplacian matrix  $\mathbf{L}_\phi \in \mathbb{R}^{|V|+1 \times |V|+1}$  of  $G_\phi$  can be written as:

$$\mathbf{L}_\phi = \begin{pmatrix} \begin{pmatrix} \mathbf{L} + \phi \mathbf{I} \\ -\phi \quad \dots \quad -\phi \end{pmatrix} & \begin{pmatrix} -\phi \\ \vdots \\ -\phi \\ |V| \cdot \phi \end{pmatrix} \end{pmatrix}$$

The drain vector is extended accordingly and adapted to the modified problem. To subtract  $|V| \cdot \delta$  load from the extra vertex  $x$  and add it equally to the set  $S$  of source vertices,  $d_\phi$  looks like:

$$d_{\phi_v} = \begin{cases} \delta \cdot |V|/|S| & : v \in S \\ -\delta \cdot |V| & : v \text{ is the extra vertex } x \\ 0 & : \text{otherwise} \end{cases}$$

Now, we compute the load distribution by solving the according linear system  $\mathbf{L}_\phi w_\phi^{(\infty)} = d_\phi$ .

The extra vertex seems to introduce global information since  $x$  is connected to every other vertex. Accordingly, the last row and column of  $\mathbf{L}_\phi$  are filled completely. Hence, some solvers like algebraic multigrid cannot be applied without modifications. Furthermore, the rank of the modified matrix is not full since  $\mathbf{L}_\phi$  is still a Laplacian matrix. However, solving an underdetermined system of linear equations can be simplified [13]. If  $r := \dim\{x \mid \mathbf{L}x = 0\}$  is the dimension

of the null space of  $\mathbf{L}$ , fixing  $r$  entries of the solution vector and deleting the corresponding rows and columns from the matrix and right hand side, one can obtain a solution by solving a fully determined system.

Hence, the numerical problems can be overcome by fixing the solution value of the extra vertex to be zero and deleting the row and column appended to  $\mathbf{L}$  before. What remains is the addition of  $\phi$  to the diagonal values of  $\mathbf{L}$ . This results in a symmetric positive definite matrix  $\mathbf{L}_\phi' = \mathbf{L} + \phi\mathbf{I}$ , whose condition is controlled by the parameter  $\phi$ . To obtain the load differences on the modified graph, we can now solve the fully determined linear system  $\mathbf{L}_\phi' w_\phi' = d_\phi'$ . Although  $w_\phi' \neq w_\phi$  in the general case, this simple approach delivers meaningful results for the BUBBLE operations. Moreover, it improves speed and robustness of linear solvers significantly and has a meaningful interpretation due to the notion of the extra vertex.

After eliminating the entries for the extra vertex from the matrix, the non-zero structure of  $\mathbf{L}_\phi'$  is the same as of  $\mathbf{L}$ , so that the introduced global information is eliminated again. Furthermore, the entry for the extra vertex in the solution vector  $w_{\phi_x}'$  is defined to be zero. Hence, it automatically acts as a common reference point, all load values are non-negative, and the load distributions for different partitions are now comparable without post-processing. That is why we use the notion of the extra vertex in the following unless stated otherwise.

## 5 Balancing and Smoothing

Our experiments on a torus show that the proposed shape optimizing approach leads to almost equal partition sizes. The partitionings displayed in Figure 7 illustrate the learning process. However, when being applied to unstructured graphs, additional balancing becomes necessary. In the following, we describe two methods to improve the balance.

### 5.1 Scale Balancing

A first idea to establish equal partition sizes is to adjust the amount of load that is placed onto the partitions. If a partition is too small or too large, its total load amount can be increased or decreased for the next learning step of the BUBBLE framework. However, this requires the execution of additional diffusion processes and is therefore quite costly.

Instead of recomputing the load distribution, it is possible to adjust already existing solutions. Remember that the BUBBLE framework assigns a vertex  $v$  to the partition with the highest load value  $\Pi_v = p : w_{\phi_v}^{p'} \geq w_{\phi_v}^{q'} \forall q$ . As described in the previous section, we obtain the load vectors  $w_\phi^{p'}$  by solving the linear system of equations  $\mathbf{L}_\phi' w_\phi^{p'} = d_\phi^{p'}$  for each partition  $p$ , where the extra vertex  $x$  implicitly serves as common point of reference with a predefined load

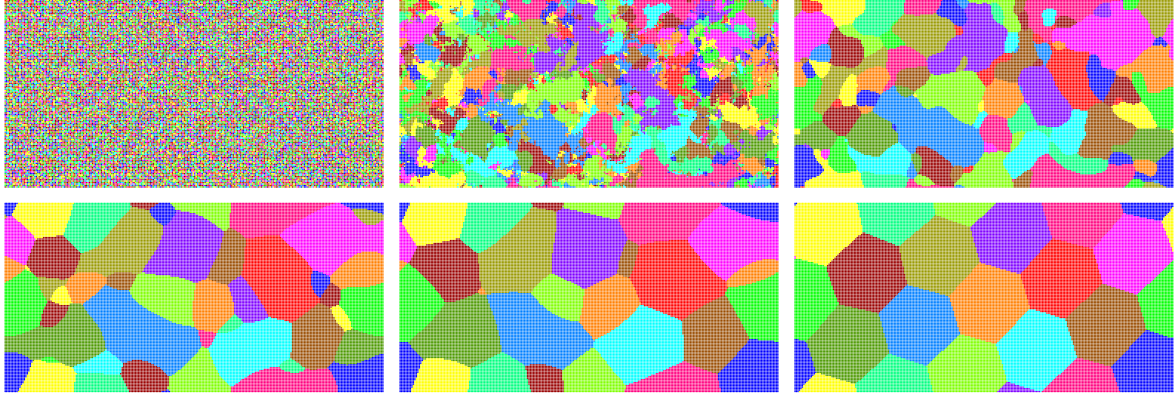


Figure 7: An initial random assignment on a  $256 \times 128$  torus (top left) and the partitionings after 1, 2, 5, and 10 **Grow/Consolidation** operations, respectively. The final solution (bottom right), where no more changes occur, is obtained after 120 operations. In this example a **Move/Centers** operation is performed after every 10 invocations of **Grow/Consolidation**. Note that the final partitioning is composed of almost perfect hexagons, which is the regular shape that fully covers the 2-dimensional plane with the smallest boundary compared to its area.

value of 0. Hence, since the computed load values correspond to a  $\|\cdot\|_2$ -minimal flow, changing the total load amount in the system is equivalent to scaling the final load vector  $w_\phi^{p'}$ .

The scaling factor  $\xi_p$  for a partition  $p$  is determined in an iterative process. Initially,  $\xi_p^0$  is set to 1. If we denote the actual partition weights by  $\omega_p$  and the total weight by  $\Omega$ , we first calculate the desired balanced weight  $\bar{\omega} = \Omega/P$ . Next, we determine the scale coefficients  $\zeta_p = (\bar{\omega}/\omega_p)^2$ , which are limited to the range  $[0.5, \dots, 2]$ . Now, the new scale factors  $\xi_p$  are computed as  $\xi_p^{i+1} = \xi_p^i \cdot \theta + (1 - \theta) \cdot \zeta_p$ . The parameter  $\theta$  acts as damping factor and prevents that too small partitions become too large in the next step and vice versa. It is usually set to 0.9, and we perform twice the partition graph's diameter many iterations. The vertex assignment is then altered according to

$$\Pi_v = p : \xi_p \cdot w_{\phi_v}^{p'} \geq \xi_q \cdot w_{\phi_v}^{q'} \forall q.$$

The proposed simple scale balancing often works very well and the good partition shapes are preserved. However, in some cases, mostly depending on the extent of the imbalance, the placement of the center vertices, and the diameter of the partition graph, this scaling approach is not able to properly balance the subdomain sizes. In extreme cases, the too large and too small subdomains are situated in opposite regions of the graph, so that the scaling requires a huge number of iterations to propagate through the partition graph.

## 5.2 Greedy Flow Balancing

If the scale balancing is not able to reduce the maximal partition overweight to less than 3%, we apply a greedy flow balancing. This approach is based on a  $\|\cdot\|_2$ -minimal balancing flow on the partition graph computed by using traditional diffusion schemes for load balancing. Vertices to be migrated are chosen based on this flow, i.e., a vertex  $v$  in partition  $p$  is considered if the flow from partition  $p$  to partition  $q$  is positive and  $v$  is adjacent to a vertex  $u$  in  $q$ . Among all considered vertices we move that one which causes the smallest error  $w_{\phi_v}^{q'} - w_{\phi_v}^{p'}$  w.r.t. the load situation on the vertices. Once the vertex is moved, we upgrade the vertex values and update the balancing flow. Note that these changes are local and only neighboring vertices are affected. The process is repeated until the flow on all edges of the partition graph is less than 0.5.

## 5.3 Smoothing

To further enhance the partition quality, we perform a final post-processing step called *smoothing*. Once the learning BUBBLE process has been completed, the smoothing further straightens the partition boundaries. This is achieved by migrating vertices that possess more neighbors in a different partition than neighbors in their own. Vertices with this property are considered as candidates and a very similar migration process as described for the greedy flow balancing is performed. This procedure also eliminates possible artifacts in form of single isolated vertices that sometimes occur due to the limited numerical precision in the load calculations.

In its current implementation, the smoothing does not consider the balancing. Hence, it might destroy a completely balanced distribution. However, from our experiments we conclude that the imbalance usually does not increase by more than 1% since the fraction of affected vertices at the partition boundaries is relatively small.

# 6 The Bubble-FOS/C Graph (Re)Partitioning Heuristic

## 6.1 The Algorithm

By integrating the diffusive approach FOS/C of Section 4 into the BUBBLE framework, we obtain the BUBBLE-FOS/C algorithm sketched in Figure 8. Initially, it proceeds as described in Figure 4 (lines 01 – 06), either choosing some center vertices randomly, or performing a consolidation step on either a random or the currently existing partitioning. The computation of the **Move/Centers** (lines 09 – 11) and the **Grow/Assignment** and **Grow/Consolidation** operations (lines 14-16) is adapted. Instead of iterating the FOS/C diffusion, we compute the solution of FOS/C by solving linear systems. For each partition  $p$ , we initialize the drain vectors  $d_{\phi}^{p'}$  and

```

00  Algorithm FLUX( $G, i, l, \phi$ )
01      in each iteration  $i$ 
02          if not  $\Pi$  exists
03              case random-center
04                   $c = \text{determine-random-centers}(G)$ 
05              case random-partition
06                   $\Pi = \text{random-partitioning}(G)$ 
07          if  $\Pi$  exists /* centers */
08              parallel for each partition  $p$ 
09                  set  $d_{\phi_v}^{p'} = \begin{cases} \delta \cdot |V|/|\{v : \Pi_v = p\}| & : \Pi_v = p \\ 0 & : \text{otherwise} \end{cases}$ 
10                  solve  $\mathbf{L}_{\phi}' w_{\phi}^{p'} = d_{\phi}^{p'}$ 
11                   $\Pi_v = \begin{cases} p : w_{\phi_v}^{p'} \geq w_{\phi_u}^{p'} \forall u \in V \\ -1 : \text{otherwise} \end{cases}$ 
12          in each loop  $l$  /* assignment & optional consolidations */
13              parallel for each partition  $p$ 
14                  set  $d_{\phi_v}^{p'} = \begin{cases} \delta \cdot |V|/|\{v : \Pi_v = p\}| & : \Pi_v = p \\ 0 & : \text{otherwise} \end{cases}$ 
15                  solve  $\mathbf{L}_{\phi}' w_{\phi}^{p'} = d_{\phi}^{p'}$ 
16                   $\Pi_v = p : w_{\phi_v}^{p'} \geq w_{\phi_v}^{q'} \forall q \in \{1, \dots, P\}$ 
17                   $\Pi = \text{balance}(\Pi, w_{\phi})$  /* balancing */
18          return  $\text{smooth}(\Pi)$  /* smoothing */

```

Figure 8: Sketch of the BUBBLE-FOS/C heuristic.

source sets according to the current partitioning  $\Pi$ . As described, the source sets contain either a single center vertex (line 09) or the whole partition (line 14). The linear systems are solved, and the new center vertices (line 11) or partitionings (line 14) are determined.

The balancing is inserted after the **Grow/Assignment** and **Grow/Consolidation** operations (line 17). It consists of the scale balancing and additionally performs a greedy flow balancing if the partition sizes vary by more than a given factor. Furthermore, the final solution is smoothed (line 18), before it is returned.

An interesting point is the lack of an explicit objective function. Except for the balancing process, the BUBBLE-FOS/C heuristic does not contain any explicit directives concerning the metric to optimize. Hence, the diffusion process and the resulting load distribution are fully responsible for the solution quality. The running time of BUBBLE-FOS/C greatly depends on the linear equation solver, and, of course, the parameters  $i$  and  $l$ . All other computations require

only linear running time or are negligible.

## 6.2 Numerical Solvers for Bubble-FOS/C

The by far most time-consuming task of BUBBLE-FOS/C is the repeated solution of linear systems. That is why a very efficient solver is required to keep the running time tolerable. In the following we discuss two possibilities we have implemented. In our sequential and thread-parallel version we use Algebraic Multigrid (AMG), which is very fast but needs to be implemented with care. For our parallel implementation we use the simpler Conjugate Gradient (CG) since the combination of AMG with other parallel acceleration techniques would be extremely challenging.

### 6.2.1 Sequential Case: Algebraic Multigrid

AMG is an efficient and state-of-the-art method for solving certain linear systems. In our case, it might even appear to be predestined since its costly preprocessing only depends on the matrix while solving the system afterwards is quite fast. As we solve many systems with the same matrix  $\mathbf{L}_\phi'$  that differ only in the right hand side  $d_\phi^{p'}$ , the preprocessing step has to be performed only once.

Multigrid methods (e.g., [38]) are in general among the fastest iterative solvers and preconditioners for large linear systems derived from a wide class of partial differential equations. They are based on the observation that relaxation methods such as Jacobi or Gauss-Seidel eliminate high-frequency (unsmooth) error components in the solution vector very effectively. However, the reduction of low-frequency error components takes them a very large number of iterations. That is why these relaxation methods are also called *smoothers*. In order to eliminate also the low-frequency error quickly, a multigrid algorithm uses a hierarchy of matrices (also called *grids*, leading to the name *multigrid*), whose size decreases from one hierarchy level to the next one. If one passes a linear system with smooth error to the next coarser matrix of the hierarchy, the low-frequency components become oscillatory (unsmooth) again and can be smoothed efficiently by relaxation methods. Similar to the multilevel paradigm used for graph partitioning, this process is continued recursively, until the linear system on the coarsest level is small enough to be solved directly with adequate resource consumption.

AMG is an extension of classical multigrid to cases where no geometric information is available in connection with the matrix. One of the major differences between the two methods is the construction of the hierarchy. Classical geometric multigrid methods operate on fixed hierarchies, which sometimes have very simple construction rules. These hierarchies can also be derived from successive mesh refinements performed by the meshing algorithm of the underlying numerical application. In contrast to this, AMG constructs its own hierarchy by a top-down



coarsening approach. For this, only the matrix corresponding to the finest mesh is necessary. Note that AMG is also applicable to FOS/C if the notion of the extra vertex is not used and we have to deal with singular Laplacian matrices [21]. Yet, to be highly effective, AMG requires for non-standard problems a wise choice of components as well as parameters.

**Implementation** In previous experiments [22] we have performed the learning process on the original graph only. If the initial partitioning is undefined or of low quality, many iterations are required to find a good solution. Therefore, we adopt the multilevel scheme presented in Section 2. Rather than computing an additional hierarchy based on matchings, we use the existing AMG hierarchy. This is possible since each matrix in this hierarchy corresponds to an edge weighted graph, and two graphs of consecutive levels have a similar structure. We perform BUBBLE-FOS/C as a refinement heuristic on each level. This reduces the number of required learning iterations and therefore the running time considerably.

For the AMG hierarchy construction we use PMIS coarsening [34] to reduce the number of nodes in the next level substantially, so that the number of created levels remains modest. In cases where PMIS coarsens too much, we neglect its result and apply CLJP coarsening [10] instead. While standard CLJP coarsening reduces the weight of vertices whose influence has been taken into account by 1, we vary this value adaptively to control the coarse grid size.

Our experiments with different interpolation schemes confirm general experience [36] that this choice is crucial in order to obtain a satisfying convergence of the solver. A simple M-matrix interpolation [35, p. 448] leads to a very slow convergence in connection with the coarsening schemes used, when our FOS/C procedures have more than approximately 50,000 nodes.

Our second choice, called *classical interpolation*, works well for our class of problems. It has also been used in related work by Safro et al. [27] within a multilevel approach for optimizing linear orderings of matrices. For variables (resp. nodes)  $i, j$  of the current matrix (resp. graph) let  $I(j)$  denote the index of node  $j$  in the coarse matrix and define  $N_i$  as the neighbors  $i$  that are in the coarse set  $C$ .

$$[\mathbf{P}]_{i,I(j)} = \begin{cases} \omega_{i,j} / \sum_{k \in N_i} \omega_{i,k} & \text{for } i \in F, j \in N_i, \\ 1 & \text{for } i \in C, j = i, \\ 0 & \text{otherwise.} \end{cases}$$

Weights below a given threshold  $\eta$  (e.g.,  $\eta = 1/16$ ) are not included. This truncation reduces the number of nonzero entries in the coarse matrix. Consequently, it saves memory space and solution time. If the threshold is not too large (one even finds  $\eta = 1/5$  in the literature [27]), the convergence speed is hardly affected. After a truncation the remaining values of a row are scaled such that the row sum in  $\mathbf{P}$  is always 1.

The algorithm then follows the multilevel paradigm by starting the computation on the lowest hierarchy level. On each level, the BUBBLE-FOS/C algorithm is applied and its partitioning result is interpolated to the next level according to the respective prolongation matrix  $\mathbf{P}$ . All linear systems are solved by Full Multigrid V-cycles, which join the concept of nested iteration with V-cycles [38], until the desired error tolerance is reached. A standard CG implementation serves as the direct solver on the lowest level inside the V-Cycle.

### 6.2.2 Parallel Case: Conjugate Gradient

To solve the linear equations of BUBBLE-FOS/C in parallel, we have decided to apply a Conjugate Gradient (CG) solver (cf. e. g., [26]). It is known to be a fast and reliable method to solve sparse positive definite linear systems. Moreover, it is relatively easy to implement in parallel, even in connection with some acceleration techniques described in the following.

Note that no work is necessary to set up the sparse matrix  $\mathbf{L}_\phi'$ , because it contains the vertex degree plus  $\phi$  on the main diagonal and off-diagonal entries for every edge. Hence, it matches our graph representation and the same sparse indexed data structures can be used.

Our implementation of the BUBBLE-FOS/C heuristic for shared memory machines is straightforward. We follow exactly the formulation given in Figure 8. Optionally, the linear systems can be solved by multiple threads. Usually, this version of our library is applied to compute an initial mesh distribution for a parallel simulation.

To repartition the data during an adaptive parallel simulation, we have implemented a distributed version of the BUBBLE-FOS/C library that is based on the message passing interface (MPI). In this case we assume that the subdomains are already placed on the processing nodes. In the following we describe important enhancements that further speed up the computations in our parallel implementation using the CG solver.

### 6.2.3 Partial Graph Coarsening

As described before, a vertex is assigned to a partition according to the maximum load value. Hence, we notice that due to the hill-like manner of the solutions only a part of them is relevant to the vertex assignment. Although the load distribution corresponds to a  $\|\cdot\|_2$ -minimal flow and every vertex receives some load from every partition, in some areas the load from some partitions is negligible for the vertex assignments. The introduction of the parameter  $\phi$  bars the load from spreading too far into the graph even more. Hence, it is not necessary to compute the exact solution for all vertices of the graph, but only in important areas surrounding the respective partition, and refer to an approximation elsewhere. Of course, the important parts depend on the partition placement and are different for each of the  $P$  linear systems.

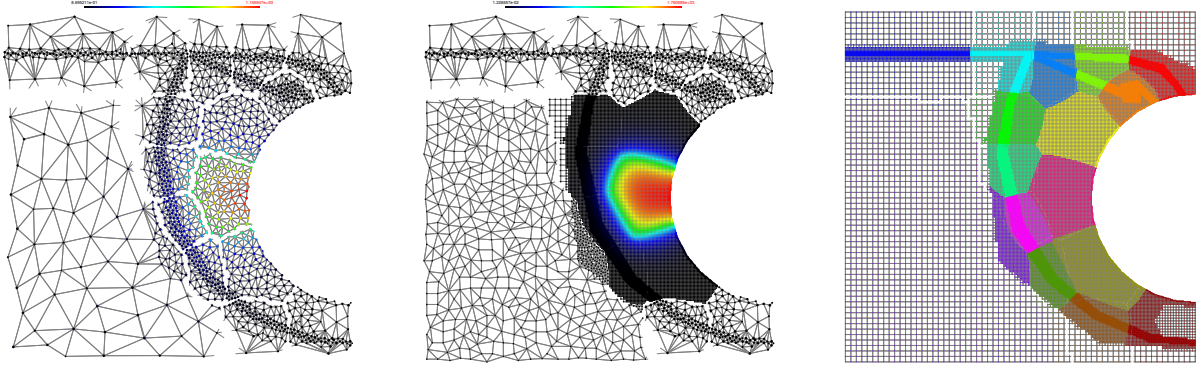


Figure 9: Vertex heights on the lowest levels (left) and the final solution with local accuracy on the respective levels (middle). The shown solution has been computed for the pink domain leading to the displayed partitioning (right). Edges between vertices of different domains (the initial partitioning) are cut.

The observation can be exploited to both speed up the computations and reduce the memory requirements in a parallel implementation. Prior to the first computation, each domain creates a local multilevel hierarchy. This hierarchy is based on a 2-approximation of a maximum weighted matching that is restricted to edges connecting local vertices. The nodes of matching edges are combined to form the vertices of the graph in the next level. Now, the idea is to solve the linear systems with different solution accuracies on the domains by utilizing a composition of different hierarchy levels. To solve a linear system, we first project the drain vector onto the respective vertices of the lowest hierarchy level and compute the load values there. Figure 9 (left) illustrates a solution for one partition on the lowest levels. One can see that the highest solution values can be found in and close to the originating domain.

Since the matching process preserves the graph structure, the solution on the lowest level is similar to the expected load distribution in the original graph. Hence, we are able to use it to determine the most relevant parts of the solution. Important domains within 10% of the highest occurring load value will be switched to a higher hierarchy level while the unimportant ones remain on the lowest one. Figure 9 (middle) gives an example of a load distribution that has been calculated with varying accuracy. In the important regions of the graph, the linear system is solved on the highest hierarchy level, that is the original graph, while in areas further away from the respective domain lower levels are used.

Note that for each of the  $P$  linear systems a different part of the graph is important. Thus, the hierarchy levels that contribute to the respective solution differ for every system. To perform the communication, our distributed graph data structure allows to send and map the numerical data between the boundary vertices of different levels. Furthermore, our parallel library solves

all systems simultaneously with a CG solver and we concatenate the data sent by all  $P$  instances. Hence, in every iteration of the solver only one message is sent in every direction between two neighboring partitions.

Using different levels in one linear system raises the question of how to obtain the halo values of vertices that do not exist in the current level of the neighboring partition. Our implementation proceeds as follows. To transfer the solution values between different levels at the domain boundaries, the domain using the higher level is in charge of either combining the numbers to be sent or distributing them on receiving, according to the calculated matching. For the reverse direction, the values for the higher hierarchy level are simply copied from their representatives in the lower level. Note that this described transformation is applicable not only between two consecutive levels, but between arbitrary ones in the hierarchy.

Although we now have to solve two linear systems per partition, a small one on the lowest hierarchy levels and the second one on the mixed levels, less running time is required in total compared to solving one system on the original graph if the number of partitions is large enough. The lowest levels of the hierarchy are very small and can be processed quickly. The additional time spent in this computation is compensated by the reduction of the system sizes in the second computation.

#### 6.2.4 Domain Decomposition and Domain Sharing

The domain decomposition (DD) approach in which each processor stores its own (local) part of the whole graph is the usual way to distribute a graph representing a system of linear equations on a parallel computer. Following this practice, the implemented preconditioned CG solver requires three communications per iteration, one matrix communication that updates the halo values and two scalar products. Hence, the number of messages is proportional to the number of iterations, which typically grows with the system size.

Since we solve  $P$  linear systems concurrently, an alternative distribution scheme for the computations exists. Instead of having every node process the chosen hierarchy level of its own domain for each of the  $P$  systems, it is possible to assemble one complete linear system on every processor. The systems are then solved locally without any communication, and finally the solution is sent back to the domains. We call this approach *domain sharing* (DS).

Domain sharing requires copies of all domains on every other node, which is usually impossible due to the memory requirements. However, we have seen that an accurate solution is not required in all areas of the graph, especially if the number of partitions is large. Hence, mainly lower levels of the hierarchy are requested, reducing the memory requirements significantly.

## 7 Experiments

This section contains some of the experimental results we obtain with BUBBLE-FOS/C. We first introduce the metrics and norms that we consider relevant for the quality of a partitioning. Next, we present experimental results for partitioning known benchmark graphs from scratch. Afterwards, results from load balancing experiments on created test instances are described. These test instances are sequences of graphs and reflect changes of a simulation mesh during an adaptive simulation. As a standard of reference we use the two state-of-the-art libraries METIS and JOSTLE in their sequential and parallel versions.

### 7.1 Metrics and Norms

To measure the quality of a partitioning, a number of metrics are possible. The traditional one is the edge-cut, that is the number of edges between different partitions. It is known, however, that the edge-cut usually does not model the real costs [8]. Depending on the application, some of the metrics might be more important than others. Hence, we consider a number of them which can be described as follows and are measured per partition  $p$ .

**External edges** Number of edges incident to exactly one vertex of partition  $p$ , meaning that they are in-between two partitions:  $\text{ext}(p) = |\{e = (u, v) \in E : \Pi(u) = p \wedge \Pi(v) \neq p\}|$

**Boundary vertices** Number of vertices of partition  $p$  that are incident to at least one external edge:  $\text{bnd}(p) = |\{v \in V : \Pi(v) = p \wedge \exists u \in V : \Pi(u) \neq p \wedge (v, u) \in E\}|$

**Diameter** The longest shortest path between two vertices of the partition  $p$ . Infinity, if the partition is not connected.

**Outgoing migration** Number of vertices migrated from partition  $p$  to another partition.

**Incoming migration** Number of vertices migrated from another partition to partition  $p$ .

The number of vertices at the partition boundary reflects the amount of information to be sent to a neighboring partition more precisely than the edge-cut. Usually, the boundary vertices are mirrored in the neighboring partition and updated before every iteration, such that the locally operating algorithms can access their data. Hence, data has to be communicated only once for each vertex, even if it has more than one neighbor in that partition.<sup>1</sup> Additionally, the

---

<sup>1</sup>Note that the number of boundary vertices does not exactly match the amount of information to be sent due to vertices that are adjacent to more than one neighboring partition. However, only a few such vertices exist and in our experiments these numbers are negligible.

Table 1: Graphs used in the experiments of Section 7.2.

Graph	Size		Degree			Origin
	$ V $	$ E $	min	max	avg	
airfoil1	4,253	12,289	3	9	5.779	FEM 2D
crack	10,240	30,380	3	9	5.934	FEM 2D
whitacker (dual)	19,190	28,581	2	3	2.979	FEM 2D dual
biplane9	21,701	42,038	2	4	3.874	FEM 2D
stufel0	24,010	46,414	2	4	3.866	FEM 2D
altr4	26,089	163,038	5	24	12.499	FEM 3D
shock9	36,476	71,290	2	4	3.909	FEM 2D
wing	62,032	121,544	2	4	3.919	FEM 3D dual

quality of a partitioning depends on its balance. A less balanced solution does not necessarily cause problems during the computation, but allows other metrics to improve further.

The metrics recorded for every partition are summarized w.r.t. two different norms. Given the values  $x_1, \dots, x_P$ , the summation ( $\ell_1$ ) and the maximum ( $\ell_\infty$ ) norm are defined as follows:

$$\|X\|_1 := |x_1| + \dots + |x_P| \quad \text{and} \quad \|X\|_\infty := \max_{i=1..P} |x_i|$$

The  $\|\cdot\|_1$ -norm (summation norm) is a global norm. The global edge cut belongs into this category (it equals half the external edges in this norm). In contrast to the  $\|\cdot\|_1$ -norm, the  $\|\cdot\|_\infty$ -norm (maximum norm) is a local norm only considering the worst value. This norm is favorable if synchronized processes are involved.

## 7.2

For the experiments presented in this section we have chosen eight graphs of small to medium size, see Table 1, that are or have been frequently used as benchmark instances. This sample is on the one hand large enough to draw valid conclusions from its results, on the other hand it is small enough to keep the presentation of the results concise. Note that we have used more graphs than these eight ones in further experiments with BUBBLE-FOS/C and its competitors. These additional results confirm the general trend and are therefore omitted here.

### 7.2.1 Influence of Linear Solver

To assess the improvement of the more complicated AMG linear solver, we compare the speed and quality of BUBBLE-FOS/C with both solvers, CG and AMG. Note that the CG variant



of BUBBLE-FOS/C constructs a multilevel hierarchy from matchings, while the AMG variant uses AMG coarsening mechanisms.

When we compare the solution quality achieved by both variants (with AMG and with CG, both use three iterations of one **Grow/Assignment** and one **Move/Centers** operation, followed by two **Grow/Consolidation** operations), the following conclusions can be made. The solution quality of both BUBBLE-FOS/C variants are very similar, although they show a slight advantage to the CG solver combined with a multilevel matching hierarchy. A possible reason for this small discrepancy could be the different coarsening approaches of AMG and the matching algorithm. While the matching algorithm is adjusted such that star-like subgraphs (few nodes with high weights and many nodes with low weights) are avoided, AMG coarsening often produces these stars. Sometimes such subgraphs can be disadvantageous for multilevel partitioning [23]. Yet, the loss in quality is well below 1% and therefore rather small. In comparison, the speed improvement of the AMG version is significant since the running times are reduced by a factor between 4 and 5.

These data show that the introduction of AMG within BUBBLE-FOS/C constitutes a considerable acceleration for the benchmark graphs. Hence, as expected, the much more involved implementation of a multigrid solver – compared to the relatively simple CG – pays off. Since the convergence rate of the CG solver, unlike that of AMG, depends on the system size, one can expect that the speedup achieved by AMG increases for larger graphs. One has to consider, however, that solving large linear systems with AMG is not inexpensive, either, because of the time-consuming hierarchy construction.

The speed improvement obtained by the influence range reduction (here:  $\phi = 1/512$ ) discussed in Section 4.4 is about 23% for both solvers (details can be found in [19, Ch. 4.7]) compared to standard FOS/C without extra vertex. This is due to the fact that the convergence rate of the solvers are affected positively by the better condition of the positive definite system matrix obtained by the notion of the extra vertex.

### 7.2.2 Comparison to Metis and Jostle

Next, we evaluate our algorithm BUBBLE-FOS/C against METIS (more precisely kMETIS 4.0<sup>2</sup> [15], which implements direct k-way KL/FM improvement) and JOSTLE 3.0<sup>3</sup> [41] because these two are state-of-the-art KL/FM partitioners. They are probably also the most popular general purpose sequential graph partitioners due to their speed and adequate quality. Both

---

<sup>2</sup>The variant of METIS which yields shorter boundaries than kMETIS is not chosen because its results show much higher edge-cut values than kMETIS.

<sup>3</sup>Our experiments indicate that release 3.0 yields the same or comparable results as the latest release 3.1.

Table 2: Comparison of BUBBLE-FOS/C using with kMETIS and JOSTLE for  $\ell_1$ -norm and  $k = 16$ , detailed for each graph.

	kMETIS		JOSTLE		BUBBLE-FOS/C	
Graph	EC	bnd	EC	bnd	EC	bnd
airfoil1	551.2	550.6	<b>541.2</b>	<b>537.2</b>	555.8	556.9
crack	1251.8	1231.1	<b>1182.4</b>	<b>1160.4</b>	1220.0	1197.6
whitacker_dual	640.2	1271.6	624.9	1239.2	<b>591.2</b>	<b>1149.2</b>
biplane9	822.8	1403.2	<b>779.5</b>	1408.8	813.3	<b>1241.0</b>
stufel10	<b>712.8</b>	1167.7	769.0	1289.0	748.0	<b>939.8</b>
altr4	7759.7	4551.5	7608.8	4457.2	<b>7214.1</b>	<b>4206.6</b>
shock9	1247.6	2099.4	<b>1129.4</b>	1980.0	1169.9	<b>1766.3</b>
wing	<b>4611.6</b>	8277.7	4639.4	8315.5	4765.8	<b>7521.4</b>

are used with default settings, so that their optimization objective is the edge-cut. We allow all programs to generate partitions with at most 3% imbalance, i. e., whose largest partition is at most 3% larger than the average partition size.

The first comparison between BUBBLE-FOS/C – here using three outer iterations with three inner loops and the extra vertex notion ( $\phi = 1/512$ ) – and its KL/FM counterparts shows the detailed average values for each graph obtained in ten runs on the benchmark set for  $k = 16$ . Table 2 displays the results in the summation norm, while Table 3 shows them in the maximum norm. The summation norm results reveal that BUBBLE-FOS/C is able to compute partitions with the best total number of boundary nodes in most cases. There is no clear winner w. r. t. the edge-cut, but JOSTLE obtains most best values (four out of eight). In the maximum norm BUBBLE-FOS/C is clearly the best. Except for the smallest graph airfoil1, it attains the best results regarding both the number of external edges and boundary nodes.

In order to estimate how the quality of the three programs relates to each other over a variety of values for  $P$ , we adopt the following evaluation scheme. For all values obtained for a graph (time, external edges, and boundary nodes) we use the results of kMETIS as standard of reference. This means that each value of the other two partitioners is divided by the respective value of kMETIS. Then, for each  $P$  and each metric, an average value of these ratios over all graphs is computed. These average values are displayed in Figures 10 (summation norm) and 11 (maximum norm) in the rows for the respective  $P$ . The column termed *avg* contains the respective average value of the averaged ratios. In this way each graph enters the averaging process equally to ensure a fair comparison.

Table 3: Comparison of BUBBLE-FOS/C using with kMETIS and JOSTLE for  $\ell_\infty$ -norm and  $P = 16$ , detailed for each graph.

Graph	kMETIS		JOSTLE		BUBBLE-FOS/C	
	ext	bnd	ext	bnd	ext	bnd
airfoil1	<b>97.4</b>	<b>48.9</b>	104.4	50.9	105.3	52.6
crack	221.6	108.7	211.7	103.5	<b>209.6</b>	<b>103.0</b>
whitacker_dual	110.1	108.9	108.5	107.5	<b>101.8</b>	<b>97.2</b>
biplane9	150.2	125.8	147.4	131.1	<b>144.6</b>	<b>103.1</b>
stufel10	130.6	106.9	172.2	141.7	<b>111.8</b>	<b>69.2</b>
altr4	1291.3	373.8	1260.8	362.9	<b>1110.2</b>	<b>318.7</b>
shock9	223.3	186.5	204.9	179.1	<b>200.1</b>	<b>143.1</b>
wing	790.9	697.4	902.7	781.5	<b>713.3</b>	<b>559.0</b>

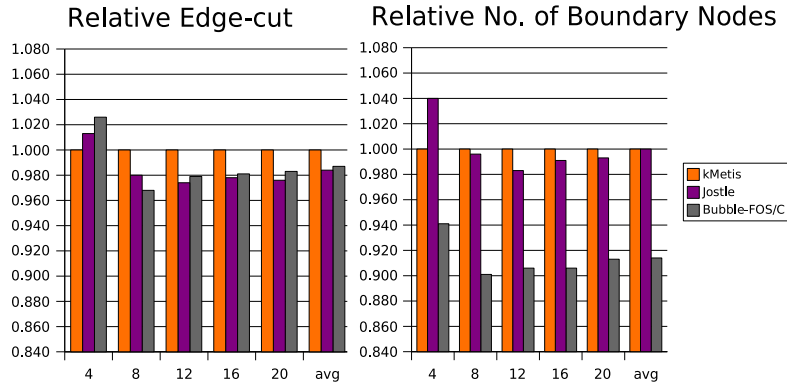


Figure 10: Partitioning quality of JOSTLE and BUBBLE-FOS/C relative to kMETIS in the  $\ell_1$ -norm for different  $k$ .

Clearly, our algorithm BUBBLE-FOS/C is able to compute the partitions with the shortest boundaries, both in the summation and the maximum norm. It is also able to compute partitions with the fewest maximum external edges except for  $P = 4$ . The traditional edge-cut metric is best optimized in most cases by JOSTLE, but BUBBLE-FOS/C is not far behind. It is therefore possible to conclude from these data that the partitions computed by BUBBLE-FOS/C show the best overall properties compared to its competitors, at least for  $P \in \{8, 12, 16, 20\}$ . The largest improvement can be seen for the maximum number of boundary nodes, the metric which probably measures best the communication costs of parallel numerical solvers. In this metric our algorithm is 13.2% better than kMETIS and 16.3% better than JOSTLE.

Regarding the shape of the subdomains, Figure 12 makes some of the major differences between the three programs visible. In particular kMETIS computes solutions with jagged

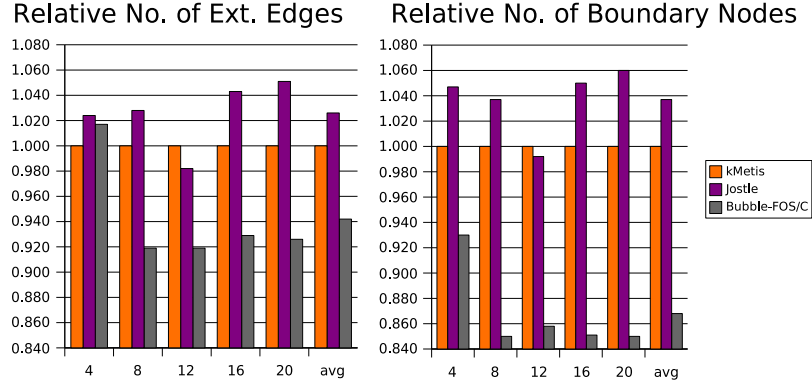


Figure 11: Partitioning quality of JOSTLE and BUBBLE-FOS/C relative to kMETIS in the  $\ell_\infty$ -norm for different  $k$ .

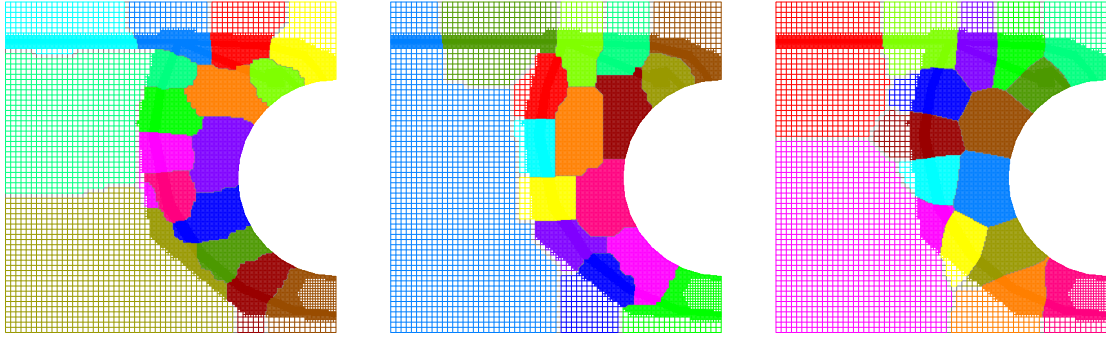


Figure 12: Partitioning the *shock9* graph into 16 subdomains. The solution of METIS (left) shows jagged boundaries and elongated subdomain parts. JOSTLE's partition (middle) has somewhat smoother boundaries. BUBBLE-FOS/C (right) computes a solution where the shape-optimizing approach becomes apparent in the nearly convex subdomains.

boundaries. JOSTLE on the other hand seems to aim at rectangular shapes, which is a good idea for edge-cut minimization. For short boundaries, however, convex subdomains are preferable. Although not all subdomains in the solution of BUBBLE-FOS/C are convex, the shape optimizing approach based on FOS/C is certainly recognizable. We are also interested in the diameter and connectedness of the subdomains. BUBBLE-FOS/C computes partitions with disconnected subdomains only in 3.5% of our test runs. In contrast to this, kMETIS and JOSTLE perform considerably worse, as they produce disconnected subdomains in 11.3% and 14.5% of the cases, respectively. To compare the diameter of the partitions, we evaluate the experiments for a medium number of subdomains,  $P = 12$ . Both in the summation norm and the maximum norm, BUBBLE-FOS/C computes solutions whose diameter is the smallest on average. Compared to JOSTLE, which is on average better than kMETIS in this category, our algorithm performs 4.6% ( $\ell_1$ ) and 10.5% ( $\ell_\infty$ ) better, respectively.

### 7.2.3 Running Times

The running times of the three programs are clearly in favor of `KMETIS` and `JOSTLE`. `KMETIS` requires only approximately two hundredth of a second to partition the graphs of the benchmark set. The other KL/FM partitioner `JOSTLE` is about 2.5 times slower than `KMETIS`, which is still very fast. Compared to these state-of-the-art libraries, `BUBBLE-FOS/C` requires much more running time. The average values range from about seven seconds for  $P = 4$  to approximately thirty seconds for  $P = 20$  for the variant with AMG and the extra vertex. Despite these two introduced acceleration methods, our algorithm is still about three orders of magnitude slower than its established competitors.

## 7.3 Graph Sequences

In order to evaluate `BUBBLE-FOS/C` for load balancing of adaptive simulations by repartitioning, we construct a test set of graph sequences. A graph sequence reflects the changes of the discretization caused by the mesh refinement and coarsening procedure. Each graph, also called frame, is the dual to the static mesh at that point when the load balancing algorithm is started. Performing the mesh adaptation in parallel, new elements usually belong to the domain they have been created in. To be able to provide this information independently of a given partitioning, each of the inserted elements must be assigned to one element of the preceding graph. This allows to place the new objects onto the same processor as their predecessors and therefore to model the parallel behavior, even if the actual partitioning is not known.

The included experiments are based on ten graph sequences. The 'refine' benchmark refines the mesh around a circle positioned in the center of the simulation area. In the 'change' sequence, also some elements are combined and the mesh is coarsened around the second circle. In the 'heat' benchmark, the geometry covers large parts of the simulation area, leading to some long and thin simulation spaces. The refined area moves from the top left corner to the bottom right. The 'ring' and 'ring2' sequences rotate the refined area in a ring like geometry, and 'ring2' additionally contains an object therein. In the 'circles' benchmark, three circles of different size rotate in a row around the center. The smaller a circle, the finer is the mesh discretization. 'Slowtric' 'fasttric' rotate the same circles, too, but they are placed equidistant. In the 'fasttric' benchmark, the movement speed is twice as high as in the 'slowtric' case. In the benchmarks 'bubbles' and 'trace' some circles cross the simulation area. While in the first case the area is refined according to the distance to the geometry, the 'trace' benchmark delays the coarsening such that many elements remain on the path of the geometry.

All instances shown can be downloaded from our website on load balancing benchmarks [28].

The graph sequences are designed to contain about 20,000 elements per frame. Depending on the setting, this number varies over time due to the mesh adaptation.

## 7.4 Graph Repartitioning Results with Parallel Bubble-FOS/C

In this subsection we present an extraction of the results we obtained when applying the distributed BUBBLE-FOS/C heuristic on the graph sequences of our test set. Not surprisingly, when comparing the results of the BUBBLE-FOS/C library with different parameter settings, we come to the conclusion that a smaller choice of  $\phi$  and more (outer) iterations and (inner) loops usually improve the solution quality, but also enlarge the running time. Even more detailed numerical results are available in Schamberger's thesis [29].

### 7.4.1 Comparison to ParMetis and Jostle

As before, we evaluate our heuristic by comparisons to the state-of-the-art repartitioners METIS and JOSTLE, more precisely their parallel variants. The charts listed in Figure 13 display the solutions of the BUBBLE-FOS/C heuristic together with those of the parallel versions of METIS and JOSTLE for the 'slowtric' and 'bigslowtric' sequence, respectively. Since the experiments have been performed on different machines, the running time is not listed, but as before BUBBLE-FOS/C performs by far slowest.

According to the top chart in Figure 13, the libraries stay inside their balance bounds of 3%, with some exceptions in case of METIS. Furthermore, JOSTLE ignores the imbalance allowance and almost delivers totally balanced solutions. In the following rows the metrics are shown in summation (left column) and maximum norm (right column).

The second row contains the number of external edges. Most of the time, BUBBLE-FOS/C is able to find the best solutions while JOSTLE produces results with an about 5% and PARMETIS with an about 20% higher edge-cut. In the maximum norm, this gap rises to about 12% and 35%, respectively. Similar observations can be made for the boundary displayed in the next row. The number of external edges and boundary vertices seems to be linked and differences are hard to spot for this sequence. It can be explained with the limited vertex degrees (at most 3) in combination with the small graph size.

The diameter shows the same tendencies as the number of external edges or boundary vertices. The advantage of BUBBLE-FOS/C is slightly larger in case of the maximum norm than in the summation norm. Furthermore, there are some frames missing in case of PARMETIS which indicates disconnected partitions.

The last row displays the migration. The results for PARMETIS differs to those of BUBBLE-FOS/C or JOSTLE because the former library migrates either very few or a large number

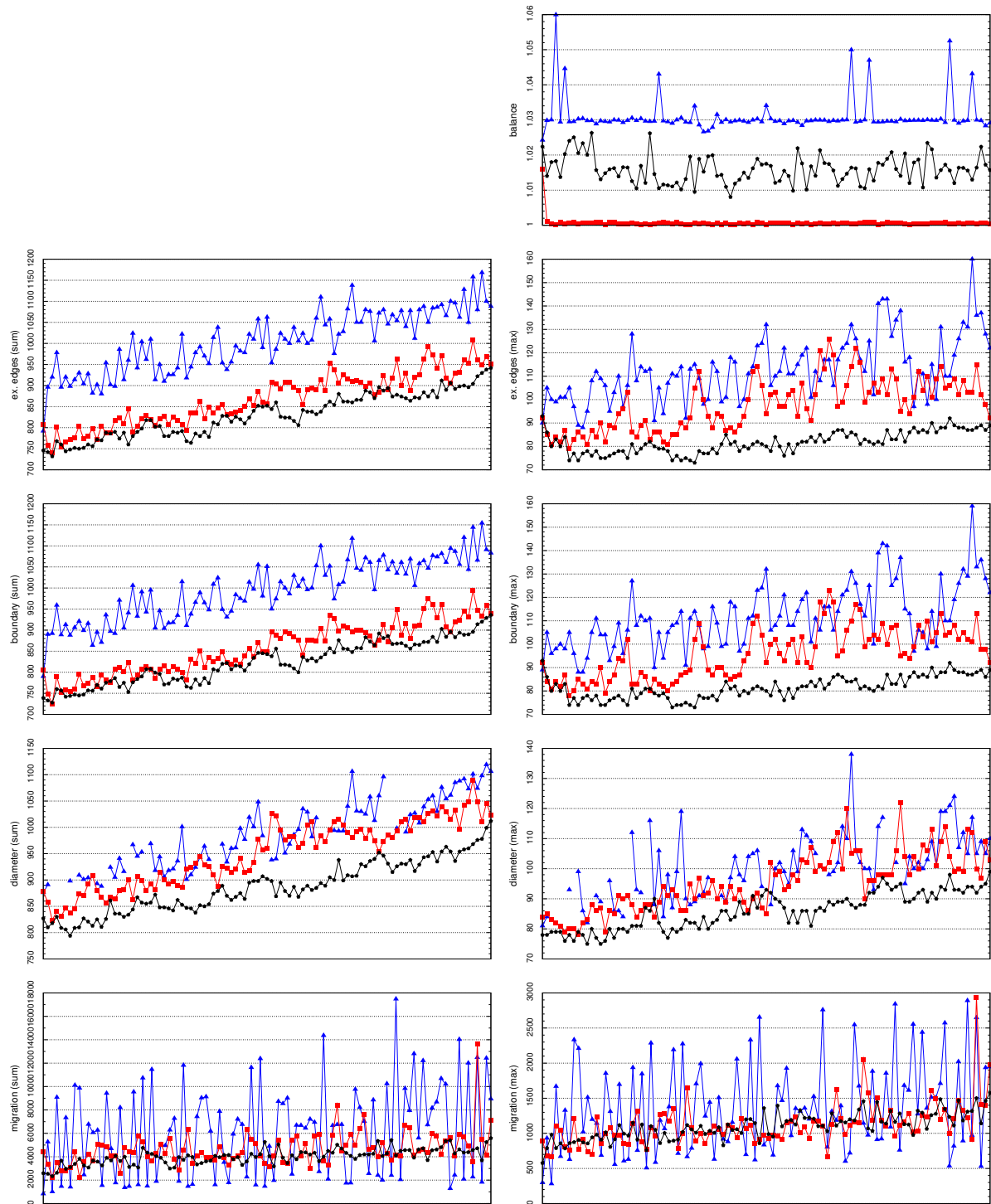


Figure 13: Repartitioning results of all 101 frames with 12 partitions of the 'slowtric' benchmark with PARMETIS (blue triangles), JOSTLE (red squares), and BUBBLE-FOS/C (black circles) (4 iterations, 4 loops,  $\phi = 0.001$ ). Each triangle, square, or circle represents the value of one frame.



of vertices. JOSTLE and BUBBLE-FOS/C proceed much more steadily, and BUBBLE-FOS/C moves a slightly smaller amount of elements than JOSTLE.

Since the 'slowtric' benchmark consists of rather small graphs, we use the same geometry description to generate the 'bigslowtric' sequence. It is identical except for the number of vertices, which is increased from around 20,000 to 100,000. The results of our experiments with this benchmark are very similar to what we observed in case of the 'slowtric' benchmark before. BUBBLE-FOS/C computes the partitionings with the fewest cut edges. In the summation norm the advantage to JOSTLE increases slightly to around 7.5%, and to around 25% when comparing with PARMETIS. In the maximum norm, the values of 25% and 55% are even larger. Again, the number of boundary vertices is closely coupled to the number of cut edges, hence a similar improvement can be observed. Looking at the diameter, we see that the partitions computed with PARMETIS are disconnected even more often. As for the 'slowtric' benchmark, BUBBLE-FOS/C determines the domains with the smallest diameter. Concerning the vertex migration, PARMETIS follows its alternative strategy, while BUBBLE-FOS/C moves a very steady number of vertices during the transitions between all frames of the benchmark.

The 'slowtric' and 'bigslowtric' benchmarks indicate that the observations made on the small instances are in general transferable to the larger ones, which we have confirmed on several other sequences. The benefit of well shaped partitions with straight boundaries becomes larger with an increasing problem size, resulting in a better solution quality in case of BUBBLE-FOS/C compared to the other libraries.

The results for the remaining graph sequences of our test set are similar to the shown 'slowtric' example and endorse the previous observations. To be comprehensive, we only list the values of an arbitrary frame in Table 4. More precisely, we compare the results of PARMETIS, JOSTLE and BUBBLE-FOS/C after processing frame 66.

As shown, the solution quality is highest when applying the BUBBLE-FOS/C heuristic. In both included norms the subdomains usually have fewer external edges, the least number of boundary vertices, the smallest diameter. Additionally, a comparable very steady amount of migration occurs. There are only a few exceptions as in case of the 'trace' sequence, where our library finds slightly worse results than JOSTLE. In some cases, JOSTLE and BUBBLE-FOS/C migrate many more vertices than METIS, but this is only due to the alternative strategy and METIS average value is worse than that of JOSTLE or BUBBLE-FOS/C.

To provide a better impression of the results, Figures 14 and 15 display the domains computed by PARMETIS, JOSTLE and BUBBLE-FOS/C in frame 66 for the 'ring', and 'bubbles' benchmarks, respectively. In both examples it is clearly visible that BUBBLE-FOS/C determines a good partition placement that allows well shaped domains with straight boundaries.



Table 4: Solution quality of the 12 partitionings after processing frame 66 with METIS, JOSTLE and BUBBLE-FOS/C. The best results are printed bold.

seq.	lib.	wgt.	ex. edges		boundary		diameter		migration	
		max	sum	max	sum	max	sum	max	sum	max
refine	METIS	1.05	57.0	492.0	57.0	492.0	524.0	151.8	<b>0.0</b>	<b>0.0</b>
	JOSTLE	1.02	49.0	452.0	<b>48.0</b>	445.0	492.0	142.4	748.0	244.5
	BUBBLE-FOS/C	1.02	<b>51.0</b>	<b>446.0</b>	<b>48.0</b>	<b>437.0</b>	<b>461.0</b>	<b>133.2</b>	196.0	72.0
change	METIS	1.04	79.0	668.0	78.0	652.0	711.0	207.1	<b>14.0</b>	<b>14.0</b>
	JOSTLE	1.00	62.0	558.0	62.0	552.0	674.0	195.3	508.0	171.9
	BUBBLE-FOS/C	1.03	<b>60.0</b>	<b>514.0</b>	<b>60.0</b>	<b>509.0</b>	<b>620.0</b>	<b>179.2</b>	638.0	208.7
heat	METIS	1.04	50.0	300.0	50.0	291.0			<b>14.0</b>	<b>12.2</b>
	JOSTLE	1.00	43.0	268.0	43.0	265.0	1008.0	298.5	1254.0	482.9
	BUBBLE-FOS/C	1.02	<b>40.0</b>	<b>262.0</b>	<b>39.0</b>	<b>261.0</b>	<b>989.0</b>	<b>294.4</b>	436.0	155.9
ring	METIS	1.03	125.0	954.0	125.0	947.0	1299.0	385.9	14026.0	4963.4
	JOSTLE	1.00	119.0	806.0	119.0	798.0	1216.0	365.9	<b>10714.0</b>	4031.3
	BUBBLE-FOS/C	1.01	<b>92.0</b>	<b>744.0</b>	<b>92.0</b>	<b>740.0</b>	<b>1146.0</b>	<b>346.7</b>	11694.0	<b>3879.6</b>
ring2	METIS	1.03	74.0	554.0	73.0	550.0			8700.0	2820.5
	JOSTLE	1.00	67.0	482.0	67.0	479.0	888.0	265.1	5740.0	1981.8
	BUBBLE-FOS/C	1.01	<b>55.0</b>	<b>454.0</b>	<b>55.0</b>	<b>452.0</b>	<b>858.0</b>	<b>255.4</b>	<b>4770.0</b>	<b>1649.1</b>
circles	METIS	1.03	84.0	760.0	83.0	754.0	794.0	230.5	8096.0	2637.8
	JOSTLE	1.00	83.0	682.0	83.0	664.0	755.0	219.5	<b>3662.0</b>	<b>1449.3</b>
	BUBBLE-FOS/C	1.01	<b>77.0</b>	<b>642.0</b>	<b>77.0</b>	<b>639.0</b>	<b>685.0</b>	<b>198.2</b>	4714.0	1796.2
slowtric	METIS	1.03	106.0	976.0	106.0	974.0	994.0	288.1	6746.0	2367.2
	JOSTLE	1.00	119.0	938.0	118.0	927.0	1011.0	295.5	8418.0	2982.9
	BUBBLE-FOS/C	1.02	<b>86.0</b>	<b>854.0</b>	<b>86.0</b>	<b>847.0</b>	<b>900.0</b>	<b>260.8</b>	<b>5020.0</b>	<b>1807.2</b>
fasttric	METIS	1.03	93.0	906.0	93.0	899.0	906.0	265.4	8554.0	3073.8
	JOSTLE	1.00	90.0	808.0	90.0	804.0	883.0	257.4	7162.0	2704.7
	BUBBLE-FOS/C	1.02	<b>75.0</b>	<b>792.0</b>	<b>75.0</b>	<b>783.0</b>	<b>818.0</b>	<b>237.3</b>	<b>7004.0</b>	<b>2682.3</b>
bubbles	METIS	1.03	91.0	744.0	88.0	725.0	795.0	231.0	4834.0	1605.8
	JOSTLE	1.00	69.0	624.0	<b>66.0</b>	614.0	724.0	209.7	2240.0	756.5
	BUBBLE-FOS/C	1.02	<b>66.0</b>	<b>604.0</b>	<b>66.0</b>	<b>599.0</b>	<b>709.0</b>	<b>204.9</b>	<b>1306.0</b>	<b>461.8</b>
trace	METIS	1.04	86.0	774.0	84.0	756.0	745.0	217.0	<b>338.0</b>	<b>162.3</b>
	JOSTLE	1.00	<b>66.0</b>	<b>586.0</b>	<b>64.0</b>	<b>570.0</b>	691.0	200.5	1084.0	370.8
	BUBBLE-FOS/C	1.01	70.0	590.0	70.0	585.0	<b>668.0</b>	<b>193.1</b>	970.0	382.7

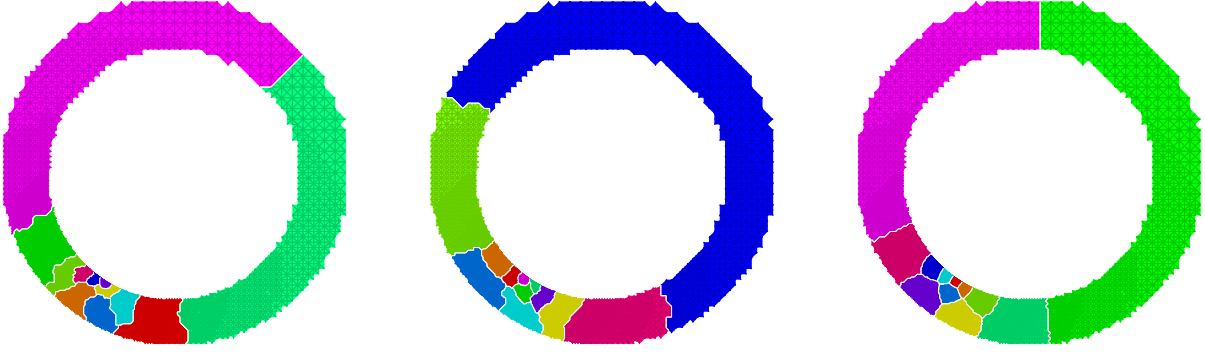


Figure 14: Frame 66 of the 'ringrot' benchmark. From left to right the solutions computed by METIS, JOSTLE and BUBBLE-FOS/C (4 iterations, 4 loops,  $\phi = 0.001$ ) are shown.

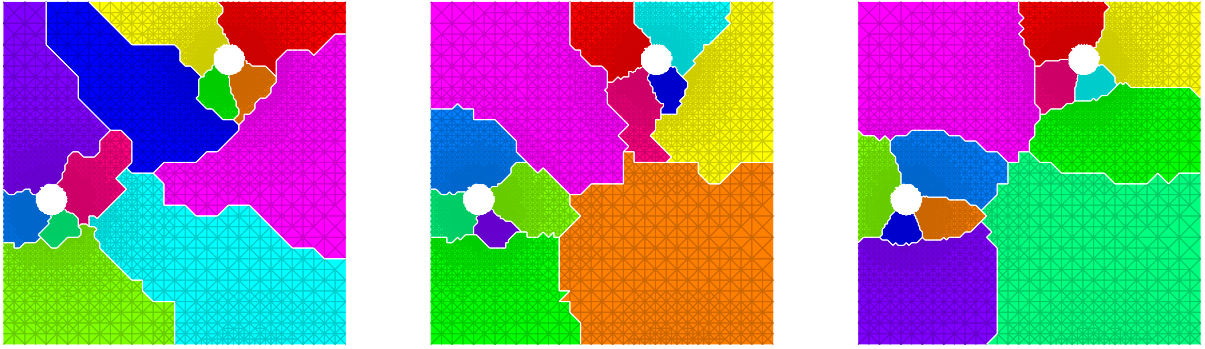


Figure 15: Frame 66 of the 'bubbles' benchmark. From left to right the solutions computed by METIS, JOSTLE and BUBBLE-FOS/C (4 iterations, 4 loops,  $\phi = 0.001$ ) are shown.

While JOSTLE's and PARMETIS's distributions are still reasonable, they tend to produce jagged shapes, explaining the often higher number of cut edges and boundary vertices.

Another point is that the partitions delivered by BUBBLE-FOS/C in these experiments are connected, whereas PARMETIS and JOSTLE sometimes produce disconnected domains.

#### 7.4.2 Parallel Running Times

The previous experiments show that the BUBBLE-FOS/C library outperforms state-of-the-art repartitioning libraries w. r. t. the solution quality on graphs derived from numerical simulations. However, its running time is much higher due to the involved numerical computations. In the following we present experimental results that study the parallel execution time in more detail.

In a distributed simulation each of the processors already contains one part of the mesh when the load balancing algorithm starts. In the experiments we measure the parallel running times that are required by 8, 16, 32 and 64 processors to repartition the graph. Note that in the current implementation the number of processors must match the number of subdomains.

Due to similarities, we limit the presentation to one experiment. The mesh in the included

Table 5: Running time results in seconds and memory consumption in MB of the distributed BUBBLE-FOS/C library repartitioning frame 100 of the 'bigtrace' sequence on the ARMINIUS cluster in parallel.

$P$	full graph		partly coarsened graph			
	DD		DD		DS	
	CG	PCG	CG	PCG	CG	PCG
8	398	425	1108	242	868	181
16	301	322	526	119	357	83
32	423	439	395	110	265	81
64	426	449	189	66	132	49

example is taken from the 'bigtrace' benchmark and represents the last transition of that sequence. The considered graph contains 832,779 vertices and 1,248,312 edges. The parameters of our distributed implementation are 2 iterations with 8 loops and  $\phi = 0.001$ . The experiments are run on a 200-node Linux cluster called ARMINIUS, where each of the nodes is equipped with two 64 bit Xeon 3.2 GHz processors and 4 GB of memory. The available interconnection network is Infiniband, communication is realized through the Scampi MPI implementation.

The main difference between the runs are the parameters for the CG solver, the most costly part of the computation. We can choose to precondition with the inverse of the matrix main diagonal (PCG) and decide to disable or enable the partial graph coarsening presented in Section 6.2.3. If partial graph coarsening is enabled, the data of the linear systems can either be distributed via the domain decomposition (DD) or the domain sharing (DS) approach introduced in Section 6.2.4.

The running time results in seconds of the experiment are listed in Table 5. Comparing the domain decomposition and the domain sharing approaches, we conclude that domain sharing is always superior. Sending only a few large data packages is more efficient than communicating many small ones due to the latency. PARMETIS and JOSTLE can be estimated to require only approximately one second for the presented experiment on the hardware employed. Hence, BUBBLE-FOS/C's running time gap to its competitors is still significant with about two orders of magnitude. Yet, the gap is often lower than in the sequential case. Moreover, recall that, compared to the sequential setting, the quality difference in favor of BUBBLE-FOS/C is generally higher.

Regarding the preconditioning with the inverse of the matrix diagonal, a benefit can only be expected when using different hierarchy levels since the input graph is unweighted and almost 3-regular. Accordingly, when solving the linear systems on the full graph only, the running times are shorter when disabling the preconditioner. However, computing a solution on the partly

coarsened graphs, preconditioning reduces the running time by a factor between 3 and 5.

So far we have looked into values that are computed with the same number of processing nodes. For the following vertical comparison, one should have in mind that the number of linear systems that have to be solved equals the number of domains. Hence, a linear speedup results in a constant execution time for full graphs.

When computing solutions on the full graph, more partitions lead to a longer running time. However, in case of the partly coarsened approach the opposite can be observed and a larger partition number requires less execution time. This can be explained with the reduction of the linear system size which occurs because more and more parts can be replaced with lower levels of the hierarchy. Yet, since the graph size is constant, the partitions become smaller. Hence, fewer local computations have to be executed and the fraction of the communication overhead rises, which slows down and eventually limits the benefit. Hence, we suppose that BUBBLE-FOS/C scales best if the problem size grows together with the number of processing nodes.

## 8 Conclusions

With this paper we have presented the first comprehensive description of our new graph partitioning and repartitioning heuristic BUBBLE-FOS/C, which delivers high-quality solutions. Although it does not contain any explicit objectives other than balancing the partition sizes, it is able to compute connected well-shaped domains with short and straight boundaries, a small diameter and few cut edges. Hence, it satisfies the demands of a range of applications very well, like the given example of balancing parallel adaptive numerical simulations.

We impute the superior solution quality to the diffusion process that gathers global information about the graph structure, screens relevant information due to the properties of the load distribution, and directs the learning framework in order to find good partition placements. Node-exchanging heuristics that are implemented in METIS, JOSTLE, and related libraries are missing this kind of information. Hence, although these libraries contain very good refinement algorithms, local changes only cannot eliminate the effects of wrong global conditions.

Furthermore, the proposed shape optimizing algorithm mainly consists of easily parallelizable computations, which might be an important advantage over commonly used vertex exchange refinement heuristics, considering the upcoming multi-core processors. The major drawback of the presented approach is its high running time, which can be reduced somewhat by techniques described in this paper and by new algorithmic features [20]. Thus, as it is a very promising approach, both practical and theoretical aspects of diffusive graph partitioning should be explored further.

## References

- [1] S. Arora and S. Kale. A combinatorial, primal-dual approach to semidefinite programs. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC'07)*, pages 227–236. ACM, 2007.
- [2] 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.
- [3] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Conference on Design automation (DAC'82)*, pages 175–181. IEEE Press, 1982.
- [4] F. Fouss, A. Pirotte, J.-M. Renders, and M. Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):355–369, 2007.
- [5] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing (STOC'74)*, pages 47–63. ACM Press, 1974.
- [6] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins Univ. Press, 3rd edition, 1996.
- [7] D. J. Harvey, S. K. Das, and R. Biswas. Design and performance of a heterogeneous grid partitioner. *Algorithmica*, 45(3):509–530, 2006.
- [8] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Proceedings of Irregular'98*, volume 1457 of *Lecture Notes in Computer Science*, pages 218–225. Springer-Verlag, 1998.
- [9] B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. In *Proceedings Supercomputing '95*, page 28 (CD). ACM Press, 1995.
- [10] V. E. Henson and U. Meier-Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(1):155–177, 2002.
- [11] J. Hromkovič and B. Monien. The bisection problem for graphs of degree 4 (configuring transputer systems). In *Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science (MFCS'91)*, volume 520 of *Lecture Notes in Computer Science*, pages 211–220, 1991.
- [12] Y. F. Hu and R. F. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25(4):417–444, 1999.
- [13] E. F. Kaasschieter. Preconditioned conjugate gradients for solving singular systems. *Journal of Computational and Applied Mathematics*, 24(1-2):265–275, 1988.

- [14] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, [...], Version 4.0*, 1998.
- [15] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.
- [16] B. W. Kernighan and S. Lin. An efficient heuristic for partitioning graphs. *Bell Systems Technical Journal*, 49:291–308, 1970.
- [17] R. Khandekar, S. Rao, and U. Vazirani. Graph partitioning using single commodity flows. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC'06)*, pages 385–390. ACM, 2006.
- [18] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–136, 1982.
- [19] H. Meyerhenke. *Disturbed Diffusive Processes for Solving Partitioning Problems on Graphs*. PhD thesis, Universität Paderborn, 2008.
- [20] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS'08)*. IEEE Computer Society, 2008. Best Algorithms Paper Award.
- [21] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, page 57 (CD). IEEE Computer Society, 2006.
- [22] H. Meyerhenke and S. Schamberger. Balancing parallel adaptive FEM computations by solving systems of linear equations. In *Proceedings of the 11th International Euro-Par Conference*, volume 3648 of *Lecture Notes in Computer Science*, pages 209–219. Springer-Verlag, 2005.
- [23] B. Monien and S. Schamberger. Graph partitioning with the party library: Helpful-sets in practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society, 2004.
- [24] L. Oliker and R. Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.
- [25] F. Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *Proceedings of the 13th International Euro-Par Conference*, volume 4641 of *Lecture Notes in Computer Science*, pages 195–204. Springer-Verlag, 2007.
- [26] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, April 2003.
- [27] I. Safro, D. Ron, and A. Brandt. Graph minimum linear arrangement by multilevel weighted edge contractions. *Journal of Algorithms*, 60(1):24–41, 2006.

- [28] S. Schamberger. Repartitioning benchmarks. <http://wwwcs.upb.de/cs/schaum/benchmark.html>, 2005.
- [29] S. Schamberger. *Shape Optimized Graph Partitioning*. PhD thesis, Universität Paderborn, 2006.
- [30] S. Schamberger and J. M. Wierum. Graph partitioning in scientific simulations: Multilevel schemes vs. space-filling curves. In *7th International Conference on Parallel Computing Technologies (PaCT'03)*, number 2763 in Lecture Notes in Computer Science, pages 165–179. Springer-Verlag, 2003.
- [31] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.
- [32] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proceedings of Supercomputing 2000*, page 59 (CD). IEEE Computer Society, 2000.
- [33] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997.
- [34] H. D. Sterck, U. M. Yang, and J. J. Heys. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM Journal on Matrix Analysis and Applications*, 27(4):1019–1039, 2006.
- [35] K. Stüben. An introduction to algebraic multigrid. In U. Trottenberg, C. W. Oosterlee, and A. Schüller, editors, *Multigrid*, pages 413–532. Academic Press, 2000. Appendix A.
- [36] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1-2):281–309, 2001.
- [37] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, June 1997.
- [38] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2000.
- [39] C. Walshaw. *The parallel JOSTLE library user guide: Version 3.0*, 2002.
- [40] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.
- [41] C. Walshaw and M. Cross. Jostle: Parallel multilevel graph-partitioning software – an overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).
- [42] C. Walshaw, M. Cross, and M. G. Everett. A Localised Algorithm for Optimising Unstructured Mesh Partitions. *International Journal of High Performance Computing Applications*, 9(4):280–295, 1995.
- [43] G. Zumbusch. *Parallel Multilevel Methods: Adaptive Mesh Refinement and Loadbalancing*. Teubner, 2003.