Dynamic Load Balancing for Parallel Numerical Simulations based on Repartitioning with Disturbed Diffusion

Henning Meyerhenke
University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany
henningm@upb.de

***Abstract*—Load balancing is an important requirement for the efficient execution of parallel numerical simulations. In particular when the simulation domain changes over time, the mapping of computational tasks to processors needs to be modified accordingly. State-of-the-art libraries for this problem are based on graph repartitioning. They have a number of drawbacks, including the optimized metric and the difficulty of parallelizing the popular repartitioning heuristic Kernighan-Lin (KL).**

In this paper we further explore the very promising diffusion-based graph partitioning algorithm DIBAP [1] by adapting DIBAP to the related problem of load balancing and improving its implementation. The presented experiments with graph sequences that imitate adaptive numerical simulations are the first using DIBAP for load balancing. They demonstrate the applicability and high quality of DIBAP for load balancing by repartitioning. Compared to the faster state-of-the-art repartitioners PARMETIS and parallel JOSTLE, DIBAP's solutions have partitions with significantly fewer external edges *and* boundary nodes and the resulting average migration volume in the important maximum norm is also the best in most cases.

Keywords: Load balancing, graph partitioning, parallel adaptive numerical simulations, disturbed diffusion.

I. INTRODUCTION

Numerical simulations are very important tools in science and engineering for the analysis of physical processes modeled by partial differential equations (PDEs). To make the PDEs solvable, they are discretized within the simulation domain, e. g., by the finite element method (FEM). Such a discretization yields a mesh, which can be regarded as a graph with geometric (and possibly other) information. Application areas of such simulations are fluid dynamics, structural mechanics, nuclear physics, and many others [2].

The solutions of discretized PDEs are usually computed by iterative numerical solvers, which have become classical applications for massively parallel computers. To utilize all available processors in an efficient manner, the computational tasks, represented by the mesh elements, must be distributed onto the processors evenly. Moreover, the computational tasks of an iterative numerical solver depend on each other. Neighboring elements of the mesh need to exchange their values in every iteration to update their own value. Since inter-processor communication is much more expensive than local computation, neighboring mesh elements should reside on the same processor. A good initial assignment of subdomains to processors can be found by solving the graph partitioning problem (GPP) [3]. The most common GPP formulation for an undirected graph $G = (V, E)$ asks for a division of V into k pairwise disjoint subsets (*partitions*) such that all partitions are no larger than $(1 + \epsilon) \cdot \lceil \frac{|V|}{k} \rceil$ (for small $\epsilon \geq 0$) and the *edge-cut*, i.e., the total number of edges having their incident nodes in different subdomains, is minimized.

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

Motivation: The most popular graph partitioning and repartitioning libraries (for details see Section II) use local node-exchanging heuristics like Kernighan-Lin (KL) [4] within a multilevel improvement process to compute good solutions very quickly. Yet, their deployment can have certain drawbacks. First of all, minimizing the edge-cut with these tools does not necessarily mean to minimize the total running time of parallel numerical simulations [5]. The number of *boundary vertices* (vertices that have a neighbor in a different partition), for instance, models the communication volume between processors often more accurately than the edge-cut [6]. While the total number of boundary vertices can

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

be minimized by hypergraph partitioning [7], synchronous parallel applications need to wait for the processor computing longest. Hence, the *maximum norm* (i. e., the worst partition) of the load balancing costs and the simulation's communication costs is of higher importance. Finally, due to their sequential nature, the most popular repartitioning heuristics are difficult to parallelize – although significant progress has been made (see Section II).

Our previously developed partitioning algorithm DIBAP aims at computing well-shaped partitions and uses disturbed diffusive schemes to decide not only *how many* nodes move to other partitions, but also *which* ones. It is inherently parallel and overcomes many of the above mentioned difficulties, as could be shown experimentally for static graph partitioning [1]. While it is much slower than state-of-the-art partitioners, it often obtains better results.

Contribution: In this paper we further explore the disturbed diffusive approach and focus on repartitioning for load balancing. First we present how the implementation of DIBAP has been improved and adapted for the repartitioning setting. With this implementation we perform various experiments, in particular repartitioning of benchmark graph sequences. These latter experiments are the first using DIBAP for repartitioning and show the suitability of the disturbed diffusive approach. The average load balancing quality – measured by the quality of the partitions *and* the migration volume – of DIBAP is usually better than that of the state-of-the-art repartitioners PARMETIS and parallel JOSTLE. It is particularly important that the improvement achieved by DIBAP concerning the quality of the partitionings in the graph sequences is even higher than in the case of static partitioning.

II. RELATED WORK

In this section we give a short introduction to the state-of-the-art of practical graph repartitioning algorithms and libraries which only require the adjacency information about the graph and no additional problem-related information. We focus on implementations that are included in our experimental evaluation. For a broader overview the reader is referred to [3].

A. Graph partitioning

To employ local improvement heuristics effectively, they need to start with a reasonably good initial solution. If such a solution is not provided as input, the multilevel approach [8], [9] is a very powerful technique. It consists of three phases: First, one computes a hierarchy of graphs $G_0, \ldots, G_l$ by recursive coarsening in the first phase. G_l ought to be very small in size, but similar in structure to the input graph G_0. A very good initial solution for G_l is computed in the second phase. After that, the solution is extrapolated to the next-finer graph recursively. In this final phase each extrapolated solution is refined using the desired local improvement algorithm. A very common local improvement algorithm for the third phase of the multilevel process is based on the method by Fiduccia and Mattheyses (FM) [10], a running time optimized version of the Kernighan-Lin heuristic (KL) [4]. The main idea of both is to exchange nodes between partitions in the order of the cost reductions possible, while maintaining balanced partition sizes. After every node has been moved once, the solution with the best gain is chosen. This is repeated several times until no further improvements are found.

B. Load balancing by repartitioning

In order to consider both a small edge-cut and small migration costs, different strategies have been explored in the literature. To overcome the limitations of simple scratch-remap and rebalance approaches, Schloegel et al. [11], [12] combine both methods. They propose a multilevel algorithm with three main features. In the local improvement phase, two algorithms are used. On the coarse hierarchy levels, a diffusive scheme takes care of balancing the subdomain sizes. Since this might affect the partition quality negatively, a refinement algorithm is employed on the finer levels. It aims at edge-cut minimization by profitable swaps of boundary vertices.

To address the load balancing problem in parallel applications, distributed versions of the partitioners METIS, JOSTLE, and SCOTCH [13], [14], [15] (and the parallel hypergraph partitioners ZOLTAN [16] and PARKWAY [17]) have been developed. An efficient parallelization of the KL/FM heuristic that these tools use is very complex due to inherently sequential parts in this heuristic. For example, one needs to ensure that during the KL/FM improvement no two neighboring vertices change their partition simultaneously and therefore destroy the consistency of the data structures.

C. Diffusion for shape optimization

Some applications profit from good partition shapes. As an example, the convergence rate of certain iterative linear solvers can depend on the geometric shape of a partition [18]. That is why in our previous work [19], [20] we have developed shape-optimizing algorithms based on diffusion. Before that repartitioning methods employed diffusion mostly for computing *how much* load needs to be migrated between subdomains [21], not *which* elements should be migrated. Generally speaking, a diffusion problem consists of distributing load from some given seed vertex (or vertices) into the whole graph by iterative load exchanges between neighbor vertices. Typical diffusion schemes have the property to result in the balanced load distribution, in which every node has the same amount of load. This is one reason why diffusion has been studied extensively for load balancing [22]. In order to distinguish dense from sparse graph regions, our algorithms BUBBLE-FOS/C [20] and the much faster DIBAP [1] (also see Section III) as well as a combination of KL/FM and diffusion by Pellegrini [23] exploit that diffusion sends load entities faster into densely connected subgraphs. In the field of graph-based image segmentation similar arguments based on the isoperimetric constant are used to find well-shaped segments [24].

III. DIFFUSION-BASED REPARTITIONING WITH DIBAP

The algorithm DIBAP, which we have developed in our previous work [1], is a hybrid multilevel combination of the two (re)partitioning methods BUBBLE-FOS/C and TRUNC-CONS, which are both based on disturbed diffusion. We call

a diffusion scheme *disturbed* if it is modified such that its convergence state does not result in the balanced distribution. Disturbed diffusion schemes can be helpful to determine if two graph nodes or regions are densely connected to each other. This property is due to the similarity of diffusion to random walks and the notion that a random walk stays in a dense region for a long time before leaving it via one of the few external edges.

Before we explain the whole algorithm DIBAP, we describe its two main (re-)partitioning components in more detail.

A. BUBBLE-FOS/C

In contrast to Lloyd's related k-means algorithm [25], BUBBLE-FOS/C partitions or clusters graphs instead of geometric inputs. Given a graph $G = (V, E)$ and $k \geq 2$, initial partition representatives (centers) are chosen in the first step of the algorithm, one center for each of the k partitions. All remaining vertices are assigned to their closest center vertex. While for k-means one usually uses Euclidean distance, BUBBLE-FOS/C employs the disturbed diffusion scheme FOS/C [20] as distance measure (or, more precisely, as similarity measure). The similarity of a node v to a non-empty node subset S is computed by solving the linear system $\mathbf{L}w = d$ for w, where $\mathbf{L}$ is the Laplacian matrix of the graph and d a suitably chosen vector that disturbs the underlying diffusion system. After the assignment step each partition computes its new center for the next iteration – again using FOS/C, but with a different right-hand side vector d. The two operations *assigning vertices to partitions* and *computing new centers* are repeated alternately a fixed number of times or until a stable state is reached. Each operation requires the solution of k linear systems, one for each partition.

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

B. TRUNCCONS

The algorithm TRUNCCONS (for *truncated consolidations*) is also an iterative method for the diffusion-based local improvement of partitions, but it is much faster than BUBBLE-FOS/C. Within each TRUNCCONS iteration the following is performed independently for each partition π_c: First, the initial load vector $w^{(0)}$ is set. Nodes of π_c receive an equal amount of initial load $|V|/|\pi_c|$, while the other nodes' initial load is set to 0. Then, this load is distributed within the graph by performing a small number ψ of FOS (first order diffusion scheme) [26] iterations. The final load vector w is computed as $w = \mathbf{M}^{\psi} w^{(0)}$, where $\mathbf{M} = \mathbf{I} - \alpha\mathbf{L}$ denotes the diffusion matrix [26] of G. A common choice for α is $\alpha := \frac{1}{(1+\deg(G))}$. One way to compute the final load of a node v is by iterative load exchanges for $1 \leq t \leq \psi$:

$$w_v^{(t)} = w_v^{(t-1)} - \alpha \sum\nolimits_{\{u,v\} \in E} (w_v^{(t-1)} - w_u^{(t-1)})$$

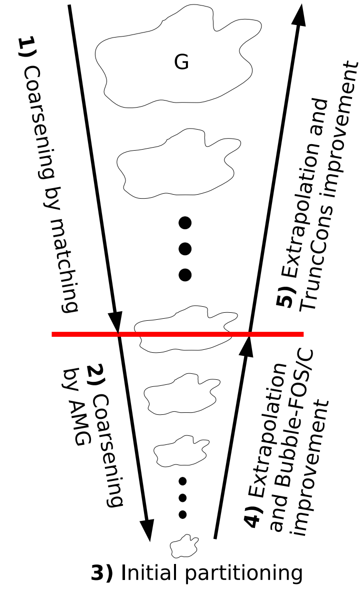

Fig. 1. Sketch of the combined multilevel hierarchy and the corresponding repartitioning algorithms used within DIBAP.

After the load vectors have been computed this way independently for all k partitions, each node v is assigned to the partition it has obtained the highest load from. This completes one TRUNCCONS iteration, which can be repeated several times (the total number is denoted by Λ subsequently) to facilitate sufficiently large movements of the partitions.

A node with the same amount of load as all its neighbors does not change its load in the next FOS iteration. Due to the choice of initial loads, such an *inactive* node is a certain distance away from the partition boundary. By avoiding load computations for inactive nodes, we restrict the computational effort to areas close to the partition boundaries.

C. *The hybrid algorithm* DIBAP

The main components of DIBAP are depicted in Figure 1. To build a multilevel hierarchy, the fine levels are coarsened (1) by approximate maximum weight matchings [27]. Once the graphs are sufficiently small, we switch the construction mechanism (2) to the more expensive coarsening based on algebraic multigrid (AMG) – for an overview on AMG cf. [28]. This is advantageous regarding running time because, after computing an initial partitioning (3), BUBBLE-FOS/C is used as local improvement algorithm on the coarse levels (4). Since BUBBLE-FOS/C uses AMG as a linear solver, such a hierarchy needs to be built anyway. Eventually, the partitionings on the fine levels are improved by the local improvement scheme TRUNCCONS. DIBAP includes additional components, e. g., for balancing partition sizes and smoothing partition boundaries. Their description is outside the scope of this paper and can be found in [29].

The rationale behind DIBAP can be explained as follows. While BUBBLE-FOS/C computes high-quality graph partitions with good shapes, its similarity measure FOS/C is quite expensive to compute compared to established partitioning heuristics. Consequently, BUBBLE-FOS/C's running time is

too high for real practical value. To overcome this problem, we use the simpler process TRUNCCONS, a truly local algorithm to improve partitions generated in a multilevel process. It exploits the observation that, once a reasonably good solution has been found, alterations during a local improvement step take place mostly at the partition boundaries. The disturbing truncation within TRUNCCONS allows for a concentration of the computations around the partition boundaries, where the changes in subdomain affiliation occur. Moreover, since TRUNCCONS is also based on disturbed diffusion, the good properties of the partitions generated by BUBBLE-FOS/C are mostly preserved.

IV. IMPROVING AND ADAPTING THE DIBAP IMPLEMENTATION FOR REPARTITIONING

In this section we describe improvements and adaptations to the DIBAP implementation compared to the version used for static partitioning [1]. Although our changes are not very drastic, they usually allow for a faster repartitioning with higher quality.

A. Multiple coarse solutions

A natural way to improve the solution quality would be to restart the whole algorithm multiple times with different initial partition assignments. However, as DIBAP is already slower than the state-of-the-art, such a computationally expensive method is not advisable. Instead of restarting the complete algorithm, we only call BUBBLE-FOS/C a number of times and keep only the best of the solutions, before starting a single multilevel repartitioning step with TRUNCCONS. Since the graph on the coarsest TRUNCCONS level (the finest BUBBLE-FOS/C level) is relatively small, BUBBLE-FOS/C returns a solution quite fast.

B. Repartitioning

When DIBAP is used for repartitioning instead of partitioning, one part of its input is an initial partitioning. We can assume that this partitioning is probably more unbalanced than advisable. It might also contain some undesirable artifacts. Nevertheless, its quality is not likely to be extremely bad. It is therefore reasonable to improve the initial partition instead of starting from scratch.

If DIBAP is called although the imbalance criterion is not violated by the input, we perform TRUNCCONS with very small Λ and ψ (e. g., both are set to 3) on the input graph only. This is relatively inexpensive and eliminates possible artifacts. It also improves the quality of the subdomains somewhat, while it generates hardly any migration costs. Note that this fast approach is not feasible in case of static partitioning where no initial solution is provided.

In case the imbalance is higher than allowed, the usual multilevel paradigm comes into play again. A matching hierarchy is constructed until only a few thousand nodes remain in the coarsest graph. The initial partition is projected downwards the hierarchy onto the coarsest level. On the coarsest level the graph is repartitioned with BUBBLE-FOS/C, starting with the projected initial solution. Going up the multilevel hierarchy recursively, the result is then improved with TRUNCCONS on each level and extrapolated to the next one.

Sometimes the matching algorithm has hardly coarsened a level, in order to avoid star-like subgraphs with strongly varying node degrees. This limited coarsening results in two very similar adjacent levels. Local improvement with TRUNCCONS on both of these levels would more or less result in the same work being done twice. That is why in such a case TRUNCCONS is skipped on the finer level of the two.

Graph	$\|E\|$	Time / $\|E\|$ (µs)
mrng1	505,048	19.07
mrng2	2,015,714	24.52
mrng3	8,016,848	15.63
mrng4	14,991,280	17.28

k	Time / $k^{0.8}$ (s)
4	2.19
8	2.19
12	2.19
16	2.20
20	2.21
32	2.11

TABLE I
NORMALIZED RUNNING TIMES OF DIBAP ($\Lambda = 10$, $\psi = 14$) ON GRAPHS FROM THE *mrng* SERIES WITH $k = 8$ (LEFT) AND ON EIGHT BENCHMARK GRAPHS FROM [30], [31] (RIGHT).

V. EXPERIMENTAL RESULTS

This section presents some of our experimental results obtained with our new DIBAP implementation. Sections V-A and V-B use the partitioning functionality of DIBAP to draw conclusions about the scaling of the running time and the effect of using multiple coarse solutions. The major experimental part is comprised of the comparison to the state-of-the-art load balancing tools PARMETIS and parallel JOSTLE (Section V-C).

A. Running time

For sufficiently large graphs the running time of DIBAP is dominated by that of TRUNCCONS, because the hierarchy level on which the algorithm switch takes place is fixed with a constant. As most expensive step within TRUNCCONS, one performs for each subdomain Λ times ψ FOS iterations (with small Λ and ψ). Seeing Λ and ψ as constants, the asymptotic running time can be bounded by $\mathcal{O}(k \cdot |E|)$.

To estimate experimentally how the graph size enters into the running time, we have conducted experiments on four graphs from the *mrng* series (dual graphs of 3D FEM meshes). Since these graphs have different sizes, but a similar structure, the results are not structurally biased. The smallest graph *mrng1* has 257,000 nodes, the largest one *mrng4* around 7.5 million. All four graphs have a very similar average degree of just below four. As a representative example, normalized running times for partitioning them into $k = 8$ subdomains are shown in Table I (left). These values indicate that an increase in the number of edges results in a similar increase in running time. The primary reasons for variations in the data are the partition placement and the resulting number of inactive nodes. Similar results can be seen in other experiments with FEM graphs. Table I (right) contains the average running times of DIBAP divided by $k^{0.8}$ for partitioning the eight largest

Setting	Λ	ψ	#coarse	Time (s)
1	10	14	1	16.52
2	10	14	3	18.73
3	14	19	1	36.51
4	14	19	3	39.21

TABLE II
PARAMETER SETTINGS AND RESULTING RUNNING TIMES WITH DIBAP ON AN INTEL CORE 2 DUO 6600 PROCESSOR FOR PARTITIONING THE "AVERAGE BENCHMARK GRAPH".

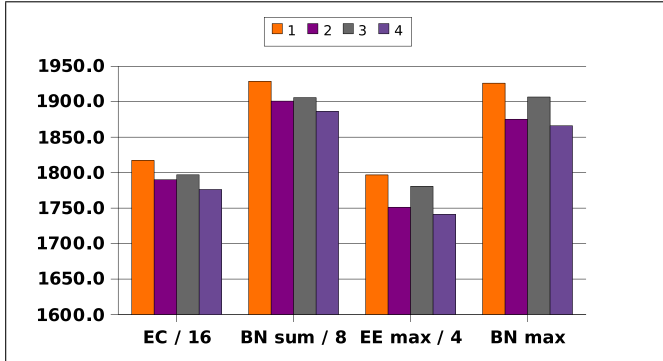

Fig. 2. Solution quality obtained by DIBAP in the four parameter settings (1 to 4) of Table II. Lower values are better.

graphs of Walshaw's popular archive [30], [31]. With this particular divisor the results are nearly constant. This sublinear behavior in k is due to the fact that the number of inactive nodes per partition increases when k becomes larger. While an extrapolation of these data to asymptotic behavior may be shaky, our expectations drawn from theoretical considerations are essentially confirmed.

B. Influence of multiple coarse solutions

To evaluate the influence of using the best of multiple coarse solutions provided by BUBBLE-FOS/C, we have conducted a set of experiments where the TRUNCCONS loop parameters Λ and ψ are fixed (they determine primarily the running time and are also very important for the solution quality), but the number of coarse solutions are varied.

Table II displays the settings for Λ, ψ, and the number of coarse initial solutions as well as the resulting running times for partitioning the benchmark set of the eight largest graphs in Walshaw's archive with DIBAP. The data are highly aggregated, as they are averaged over all graphs, different numbers of k ($k \in \{4, 8, 12, 16, 20, 32\}$), and ten randomized runs on each graph. The quality obtained by DIBAP in the four settings is shown in Figure 2. The first two data columns show the aggregated values for the edge-cut (EC) and the boundary nodes (BN sum) in the summation norm, respectively. In the next two data columns, the data for the maximum norm follow (EE: external edges). Note that for presentation reasons the values in the first three data columns are divided by the value shown in the x-axis.

Comparing the solution quality obtained with the different parameter settings, it is of course not surprising that the average solution quality is improved by choosing the best out of more than one initial solutions, as can be seen from the data. Similarly, that the most time-consuming parameter setting achieves the best quality meets our expectations. Yet, it is remarkable that setting 2 is consistently better than setting 3, although much less computational effort has been invested. This indicates that selecting a good initial partition is very helpful for saving running time. It is much cheaper than additional TRUNCCONS operations, as can be seen by the large running time difference between settings 2 and 3.

The additional gain derived from multiple solutions declines when their number is further increased. While three might not be the optimal choice, our experiments are only meant to show the general trend rather than the best value. But another positive influence of having multiple choices is the reduced variance in the data. For $k = 12$ selecting the best of three initial solutions reduces the variance in the edge-cut by an average factor of more than six. Hence, the results are not only better on average, but also more reliable as they deviate less from the mean.

C. Load balancing of graph sequences

Unlike the parallel versions of METIS and JOSTLE, the implementation of our load balancer is not prepared yet for a distributed-memory parallelization.[1] That is why we concentrate in the following on the quality of the experiments and neglect their running time. Comparing the latter is part of future work with an MPI parallel version of DIBAP. Preliminary results suggest that the average slowdown factor for using DIBAP instead of PARMETIS (which is faster than parallel JOSTLE) depends very much on the graph size and k and lies approximately between 30 and 60.

The parameter settings $\Lambda = 10$, $\psi = 14$, and three coarse solutions computed by BUBBLE-FOS/C are chosen as they provide a good trade-off between running time and quality. The `thrsh` parameter, which controls the size of the hierarchy level on which the switch between TRUNCCONS and BUBBLE-FOS/C takes place, is set to $8,000$ nodes.

Our benchmark set comprises two types of graph sequences. Twelve sequences are made of 101 frames of small graphs (around 10,000 to 15,000 nodes each), which are repartitioned into $k = 12$ subdomains. The second set consists of three sequences of larger graphs (between 110,000 and 1,100,000 nodes each), which are repartitioned into $k = 16$ subdomains. While the sequence *bigtric* has 101 frames, the sequences *bigbubbles* and *bigtrace* have only 46 frames. All graphs of these 15 sequences have a two-dimensional geometry and are generated to resemble adaptive numerical simulations such as fluid dynamics. The graph of frame $i+1$ in a given sequence is obtained from the graph of frame i by changes restricted to local areas. As an example, some areas are coarsened, whereas others are refined. These changes are in most cases due to the movement of an object in the simulation domain and often

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

result in unbalanced subdomain sizes.[2]

In addition to the graph partitioning metrics edge-cut and boundary nodes, we are here also interested in migration costs. These costs result from data changing their processor after repartitioning. We count the number of nodes that change their subdomain from one frame to the next as a measure of these costs. One could also assign cost weights to the partitioning objectives and the migration volume to evaluate the linear combination of both. Since these weights depend both on the underlying application and the parallel architecture, we have not pursued this here. We compare our new algorithm DIBAP to the state-of-the-art repartitioning tools PARMETIS and parallel JOSTLE, which are mainly based on the node-exchanging KL heuristic for local improvement. The partitioning quality (EC: edge-cut, BN: boundary nodes, EE: external edges) measured in our experiments in the summation (sum) and maximum norm (max) are displayed in Figure 3, where the values are shown relative to PARMETIS. Moreover, the values are averaged over all small sequences on the left and over all large sequences on the right. The corresponding non-aggregated data for the large sequences can be found in Tables III and IV and for the small sequences in the appendix. Additionally, we are interested in the migration costs, which are recorded in both norms and detailed for each sequence in Table V.

Sequence / Norm	PARMETIS	Par. JOSTLE	DIBAP
bigtric (ℓ_1)	1866.3	1569.4	**1436.8**
bigbubbles (ℓ_1)	4716.2	3974.8	**3527.1**
bigtrace (ℓ_1)	4124.9	3349.1	**2919.7**
bigtric (ℓ_∞)	321.3	267.0	**230.9**
bigbubbles (ℓ_∞)	845.7	740.4	**615.4**
bigtrace (ℓ_∞)	718.7	584.5	**463.9**

TABLE III
AVERAGE NUMBER OF *external edges* IN THE ℓ_1- AND ℓ_∞-NORM FOR REPARTITIONINGS COMPUTED BY PARMETIS, JOSTLE, AND DIBAP ON THREE LARGE GRAPH SEQUENCES. LOWER VALUES ARE BETTER, BEST VALUES PER INSTANCE ARE WRITTEN IN BOLD.

Sequence / Norm	PARMETIS	Par. JOSTLE	DIBAP
bigtric (ℓ_1)	3717.7	3121.9	**2865.9**
bigbubbles (ℓ_1)	9387.2	7873.7	**7041.7**
bigtrace (ℓ_1)	8212.2	6630.9	**5815.5**
bigtric (ℓ_∞)	319.6	265.5	**229.7**
bigbubbles (ℓ_∞)	840.3	729.0	**613.5**
bigtrace (ℓ_∞)	713.2	577.8	**461.7**

TABLE IV
AVERAGE NUMBER OF *boundary nodes* IN THE ℓ_1- AND ℓ_∞-NORM FOR REPARTITIONINGS COMPUTED BY PARMETIS, JOSTLE, AND DIBAP ON THREE LARGE GRAPH SEQUENCES. LOWER VALUES ARE BETTER, BEST VALUES PER INSTANCE ARE WRITTEN IN BOLD.

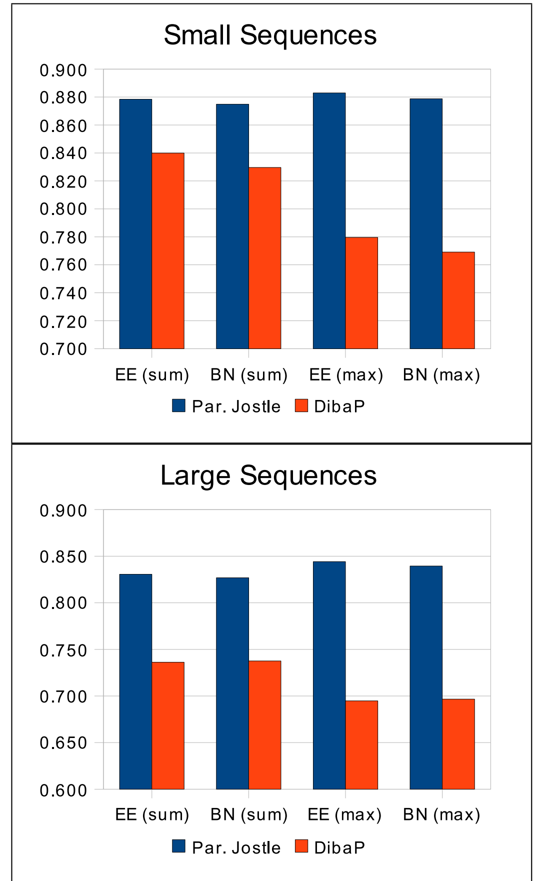

Fig. 3. Average partition quality (external edges, boundary nodes) in the ℓ_1-norm (sum) and ℓ_∞-norm (max) for repartitionings *relative to* PARMETIS (which has normalized value 1.0) on twelve small (top) and three large (bottom) graph sequences. Lower values are better.

	PARMETIS		JOSTLE		DIBAP	
Sequence	mig_1	mig_∞	mig_1	mig_∞	mig_1	mig_∞
bubbles	2460.7	558.0	**1723.9**	367.9	1775.0	**367.7**
change	284.4	74.9	**330.9**	67.8	352.2	**64.7**
circles	3200.5	676.4	**2128.6**	505.3	2164.1	**490.2**
fastrot	4314.3	934.5	3229.6	860.2	**3094.8**	**811.6**
fasttric	4648.1	1003.0	3466.6	943.8	**2940.0**	**785.8**
heat	299.8	66.8	**286.5**	**56.8**	561.1	98.7
refine	**1.5**	**1.2**	114.2	19.0	30.6	6.7
ring	3369.8	730.2	2684.0	524.7	**2584.1**	**508.0**
rotation	2914.9	**721.3**	**2281.1**	722.8	2421.6	724.2
slowrot	3928.4	806.7	2774.7	622.1	**2511.4**	**593.7**
slowtric	3094.9	718.3	2322.1	591.9	**2165.1**	**559.7**
trace	977.5	223.7	896.0	182.3	**781.9**	**169.9**

TABLE V
AVERAGE MIGRATION VOLUME IN THE ℓ_1- AND ℓ_∞-NORM FOR REPARTITIONINGS COMPUTED BY PARMETIS, JOSTLE, AND DIBAP FOR TWELVE SMALL GRAPH SEQUENCES. LOWER VALUES ARE BETTER, BEST VALUES PER INSTANCE ARE WRITTEN IN BOLD.

The averaged graph partitioning metrics show that DIBAP is able to compute the best partitions on average. DIBAP's advance is highest for the boundary nodes in the maximum norm, which can be considered a more accurate measure for the communication costs of typical numerical solvers than the edge-cut. With about 12–15 % on parallel JOSTLE and about 23–30% on PARMETIS these improvements are clearly higher than the approximately 7% obtained for static partitioning [1], which is due to the fact that parallel KL (re)partitioners often

[2]For more details the reader is referred to Marquardt and Schamberger [32], who have provided the sequence data. The input data can also be downloaded from the project website http://wwwcs.upb.de/cs/henningm/graph.html.

Sequence / Norm	PARMETIS	Par. JOSTLE	DIBAP
bigtric (ℓ_1)	27563.2	**20170.0**	22248.3
bigbubbles (ℓ_1)	197449.2	**157475.0**	182205.9
bigtrace (ℓ_1)	71934.6	**61294.1**	90358.2
bigtric (ℓ_∞)	5257.8	4507.9	**4275.5**
bigbubbles (ℓ_∞)	39601.7	29448.9	**28422.0**
bigtrace (ℓ_∞)	15117.4	**11881.8**	13975.4

TABLE VI
AVERAGE MIGRATION VOLUME IN THE ℓ_1- AND ℓ_∞-NORM FOR REPARTITIONINGS COMPUTED BY PARMETIS, JOSTLE, AND DIBAP FOR THREE LARGE GRAPH SEQUENCES. LOWER VALUES ARE BETTER, BEST VALUES PER INSTANCE ARE WRITTEN IN BOLD.

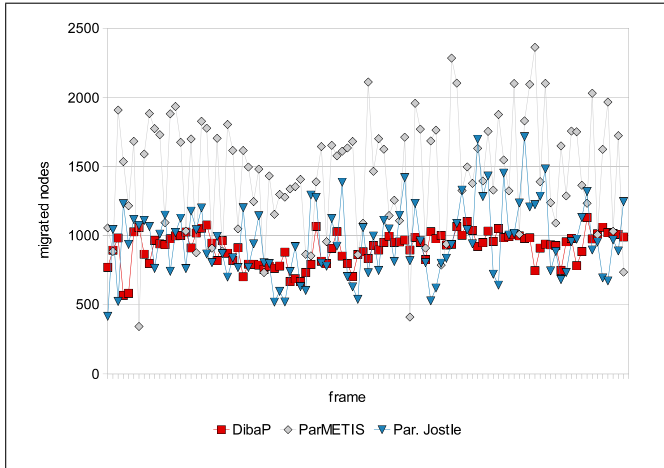

Fig. 4. Number of migrating nodes (ℓ_∞-norm) in each frame of the slowrot sequence for DIBAP (red square), METIS (grey diamond), and JOSTLE (blue triangle). Lower values are better.

compute worse solutions than their serial counterparts for static partitioning.

DIBAP's migration volume is best for six (out of 15) sequences in the summation norm. Compared to this, only parallel JOSTLE is competitive. Its partitions have a lower quality, but its migration volume is best for eight sequences in the summation norm. In the more important maximum norm the results are not fundamentally different, but DIBAP performs even better than before. Again, it attains the best partitions. Moreover, the migration volume is best for eleven sequences. As a representative example, Figure 4 shows the migration volumes for each frame within the *slowrot* sequence in the ℓ_∞-norm. One can see the different strategies of the three programs. While JOSTLE and DIBAP have a relatively constant migration volume, the values for PARMETIS fluctuate extremely.

These results lead to the conclusion that DIBAP's implicit optimization with the iterative algorithms BUBBLE-FOS/C and TRUNCCONS focusses more on good partitions than on small migration costs. In some special cases the latter objective should receive more attention. As currently no explicit mechanisms for migration optimization are integrated, such mechanisms could be implemented if one finds in other experiments that the migration costs become too high with DIBAP.

It is interesting to note that further experiments indicate a multilevel approach to be indeed necessary in order to produce sufficiently large partition movements in order to keep up with the movements of the simulation. Partitions generated by multilevel DIBAP are of a noticeably higher quality regarding the graph partitioning metrics than those computed by TRUNCCONS without multilevel approach. Also, using a multilevel hierarchy results in very steady migration costs, which rarely deviate much from the mean.

In summary one can say that, in almost all cases, DIBAP computes the best repartitioning results w. r. t. to the graph partitioning metrics. Concerning the migration volume, the results are not as clear. The ℓ_1-norm values are slightly in favor of JOSTLE (eight times best) compared to DIBAP (six times best). Yet, in the ℓ_∞-norm of the migration volume, DIBAP is the clear winner again. The strategy of PARMETIS to migrate either very few or very many nodes does not seem to pay off on average since PARMETIS computes in most cases the worst solutions.

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

VI. CONCLUSIONS

With this paper we have demonstrated that the disturbed diffusive repartitioning algorithm DIBAP is a clear alternative to traditional KL-based methods for balancing the load in parallel adaptive numerical simulations. While DIBAP is still significantly slower than the state-of-the-art, it usually computes considerably better solutions. In situations where the quality of the load balancing phase is more important than its running time – e. g., when the computation time between the load balancing phases is relatively high – the use of DIBAP is expected to pay off. As part of future work we develop an MPI parallel version of DIBAP. Improvements to the algorithms and their implementation aim at a further acceleration of the disturbed diffusive approach.

REFERENCES

[1] 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, pp. 1–13, best Algorithms Paper Award.

[2] G. Fox, R. Williams, and P. Messina, *Parallel Computing Works!* Morgan Kaufmann, 1994.

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

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

[5] D. Vanderstraeten, R. Keunings, and C. Farhat, "Beyond conventional mesh partitioning algorithms and the minimum edge cut criterion: Impact on realistic applications," in *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing (PPSC'95)*. SIAM, 1995, pp. 611–614.

[6] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Comput.*, vol. 26, no. 12, pp. 1519–1534, 2000.

[7] U. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed System*, vol. 10, no. 7, pp. 673–693, 1999.

[8] T. Bui and C. Jones, "A heuristic for reducing fill in sparse matrix factorization," in 6*th SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*, 1993, pp. 445–452.

[9] B. Hendrickson and R. Leland, "A multi-level algorithm for partitioning graphs," in *Proceedings Supercomputing '95*. ACM Press, 1995, p. 28 (CD).

[10] 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)*. IEEE Press, 1982, pp. 175–181.

[11] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel diffusion schemes for repartitioning of adaptive meshes," *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, pp. 109–124, 1997.

[12] ——, "A unified algorithm for load-balancing adaptive scientific simulations," in *Proceedings of Supercomputing 2000*. IEEE Computer Society, 2000, p. 59 (CD).

[13] ——, "Parallel static and dynamic multi-constraint graph partitioning," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 3, pp. 219–240, 2002.

[14] C. Walshaw and M. Cross, "Parallel optimisation algorithms for multilevel mesh partitioning," *Parallel Computing*, vol. 26, no. 12, pp. 1635–1660, 2000.

[15] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, no. 6-8, pp. 318–331, 2008.

[16] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE Computer Society, 2007, best Algorithms Paper Award.

[17] A. Trifunović and W. J. Knottenbelt, "Parallel multilevel algorithms for hypergraph partitioning," *J. Parallel Distrib. Comput.*, vol. 68, no. 5, pp. 563–581, 2008.

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

[19] 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*, ser. Lecture Notes in Computer Science, vol. 3648. Springer-Verlag, 2005, pp. 209–219.

[20] 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)*. IEEE Computer Society, 2006, p. 57 (CD).

[21] K. Schloegel, G. Karypis, and V. Kumar, "Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 5, pp. 451–466, 2001.

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

[23] 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*, ser. Lecture Notes in Computer Science, vol. 4641. Springer-Verlag, 2007, pp. 195–204.

[24] L. Grady and E. L. Schwartz, "Isoperimetric graph partitioning for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 3, pp. 469–475, 2006.

[25] S. P. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–136, 1982.

[26] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Parallel and Distributed Computing*, vol. 7, pp. 279–301, 1989.

[27] R. Preis, "Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs," in *Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science (STACS'99)*, ser. Lecture Notes in Computer Science, vol. 1563. Springer-Verlag, 1999, pp. 259–269.

[28] K. Stüben, "An introduction to algebraic multigrid," in *Multigrid*, U. Trottenberg, C. W. Oosterlee, and A. Schüller, Eds. Academic Press, 2000, pp. 413–532, appendix A.

[29] S. Schamberger, "Shape optimized graph partitioning," Ph.D. dissertation, Universität Paderborn, 2006.

[30] A. J. Soper, C. Walshaw, and M. Cross, "A combined evolutionary search and multilevel optimisation approach to graph partitioning," *Journal of Global Optimization*, vol. 29, no. 2, pp. 225–241, 2004.

[31] C. Walshaw, "The graph partitioning archive," http://staffweb.cms.gre.ac.uk/$\sim$c.walshaw/partition/, 2009.

[32] O. Marquardt and S. Schamberger, "Open benchmarks for load balancing heuristics in parallel adaptive finite element computations," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, (PDPTA'05)*. CSREA Press, 2005, pp. 685–691.

APPENDIX

A. Additional experimental results

	PARMETIS		JOSTLE		DIBAP	
Sequence	EC	bnd_1	EC	bnd_1	EC	bnd_1
bubbles	366.7	723.8	323.8	638.6	**312.8**	**612.1**
change	357.9	706.5	308.4	607.6	**297.9**	**588.8**
circles	371.2	733.0	328.8	646.4	**314.2**	**610.0**
fastrot	433.6	857.4	385.2	757.9	**362.2**	**705.9**
fasttric	455.0	900.1	407.5	803.2	**376.3**	**741.8**
heat	182.2	360.2	**154.5**	**304.2**	159.5	306.4
refine	225.9	448.6	199.9	389.1	**191.3**	**377.6**
ring	274.4	541.1	238.0	471.2	**231.0**	**446.5**
rotation	387.9	767.7	342.6	675.3	**341.0**	**662.7**
slowrot	431.7	853.5	383.5	754.9	**359.3**	**703.9**
slowtric	502.1	994.0	434.2	856.5	**406.7**	**796.5**
trace	328.1	644.1	285.4	557.9	**273.6**	**524.8**

TABLE VII
AVERAGE NUMBER OF EXTERNAL EDGES (EDGE-CUT, EC) AND BOUNDARY NODES (BND) IN THE ℓ_1-NORM FOR REPARTITIONINGS COMPUTED BY PARMETIS, JOSTLE, AND DIBAP ON TWELVE SMALL GRAPH SEQUENCES. LOWER VALUES ARE BETTER, BEST VALUES PER INSTANCE ARE WRITTEN IN BOLD.

	PARMETIS		JOSTLE		DIBAP	
Sequence	ext_∞	bnd_∞	ext_∞	bnd_∞	ext_∞	bnd_∞
bubbles	86.0	84.8	75.4	74.1	**66.0**	**64.3**
change	83.0	81.5	70.9	69.6	**64.6**	**63.6**
circles	81.1	80.1	74.4	72.9	**66.7**	**63.7**
fastrot	96.1	94.8	86.4	84.7	**73.6**	**71.4**
fasttric	98.4	97.1	92.1	90.4	**77.6**	**76.3**
heat	56.9	56.2	**45.1**	**44.4**	48.1	46.6
refine	47.6	46.6	42.9	41.2	**36.2**	**35.7**
ring	71.3	69.9	61.1	60.5	**59.9**	**57.1**
rotation	90.0	88.7	80.3	78.8	**71.3**	**70.0**
slowrot	93.2	91.9	82.2	80.7	**70.4**	**68.7**
slowtric	112.9	111.6	97.6	95.8	**78.5**	**76.4**
trace	73.8	71.9	65.9	63.8	**58.9**	**56.3**

TABLE VIII
AVERAGE NUMBER OF EXTERNAL EDGES (EXT) AND BOUNDARY NODES (BND) IN THE ℓ_∞-NORM FOR REPARTITIONINGS COMPUTED BY PARMETIS, JOSTLE, AND DIBAP ON TWELVE SMALL GRAPH SEQUENCES. LOWER VALUES ARE BETTER, BEST VALUES PER INSTANCE ARE WRITTEN IN BOLD.