Shape Optimizing Load Balancing for MPI-Parallel Adaptive Numerical Simulations

Henning Meyerhenke

Abstract. Load balancing is important for the efficient execution of numerical simulations on parallel computers. In particular when the simulation domain changes over time, the mapping of computational tasks to processors needs to be modified accordingly. Most state-of-the-art libraries addressing this problem are based on graph repartitioning with a parallel variant of the Kernighan-Lin (KL) heuristic. The KL approach has a number of drawbacks, including the optimized metric and solutions with undesirable properties.

Here we further explore the promising diffusion-based multilevel graph partitioning algorithm DibaP. We describe the evolution of the algorithm and report on its MPI implementation PDibaP for parallelism with distributed memory. PDibaP is targeted at small to medium scale parallelism with dozens of processors. The presented experiments use graph sequences that imitate adaptive numerical simulations. They demonstrate the applicability and quality of PDibaP for load balancing by repartitioning on this scale. Compared to the faster ParMETIS, PDibaP's solutions often have partitions with fewer external edges and a smaller communication volume in an underlying numerical simulation.

Keywords: Dynamic load balancing, graph partitioning and repartitioning, parallel adaptive numerical simulations, disturbed diffusion.

1. 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 on a computer, 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 [**10**].

2000 *Mathematics Subject Classification.* 68W10, 90C35.

A preliminary version of this article appeared in the proceedings of the 15th International Conference on Parallel and Distributed Systems 2009 [**21**].

©0000 (copyright holder)

The solutions of discretized PDEs are usually computed by iterative numerical solvers, which have become classical applications for parallel computers. For efficiency reasons the computational tasks, represented by the mesh elements, must be distributed onto the processors evenly. Moreover, neighboring elements of the mesh need to exchange their values in every iteration to update their own value. Due to the high cost of inter-processor communication, 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) [**34**]. The most common GPP formulation for an undirected graph $G = (V, E)$ asks for a division of V into k pairwise disjoint subsets (*parts*) such that all parts 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 vertices 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. Another application among many others is the simulation of the dynamic behavior of biomolecular systems [**3**]. To obtain an accurate solution, a high resolution of the mesh is required in the areas of interest. To use the available memory efficiently, 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 vertices 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 2) use local vertex-exchanging heuristics like Kernighan-Lin (KL) [**18**] within a multilevel improvement process to compute solutions with low edge cuts 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 [**37, 12**]. While the total communication volume can be minimized by hypergraph partitioning [**4**], synchronous parallel applications need to wait for the processor computing longest. Hence, the *maximum norm* (i. e., the worst part in a partition) of the simulation's communication costs is of higher importance. Moreover, for some applications, the *shape* of the subdomains plays a significant role. It can be assessed by various measures such as aspect ratio [**8**], maximum diameter [**26**], connectedness, or smooth boundaries. Optimizing partition shapes, however, requires additional techniques (e. g., [**8, 26, 22**]), which are far from being mature. Finally, due to their sequential nature, the most popular repartitioning heuristics are difficult to parallelize—although significant progress has been made (see Section 2).

Our previously developed partitioning algorithm DibaP aims at computing well-shaped partitions and uses disturbed diffusive schemes to decide not only *how many* vertices move to other parts, but also *which* ones. It contains inherent parallelism and overcomes many of the above mentioned difficulties, as could be shown experimentally for static graph partitioning [**22**]. While it is much slower than state-of-the-art partitioners, it often obtains better results.

Contribution. In this work we further explore the disturbed diffusive approach and focus on repartitioning for load balancing. First we present how the implementation of PDibaP has been improved and adapted for MPI-parallel repartitioning. With this implementation we perform various repartitioning experiments with benchmark graph sequences. These experiments are the first using PDibaP for repartitioning and show the suitability of the disturbed diffusive approach. The average quality of the partitions computed by PDibaP is clearly better than that of the state-of-the-art repartitioners ParMETIS and parallel Jostle, while PDibaP's migration volume is usually comparable. It is important to note that PDibaP's improvement concerning the partition quality for the graph sequences is even higher than in the case of static partitioning.

2. Related Work

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. For a broader overview the reader is referred to Schloegel et al. [**34**]. The most recent advances in graph partitioning are probably best covered in their entirety by the proceedings volume [**2**] the present article is part of.

2.1. 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 [**13**] is a very powerful technique. It consists of three phases: First, one computes a hierarchy of graphs $G_0, \dots, G_l$ 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 interpolated to the next-finer graph recursively. In this final phase each interpolated 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) [**9**], a variant of the well-known local search heuristic by Kernighan and Lin (KL) [**18**] with improved running time. The main idea of both is to exchange vertices between parts in the order of the cost reductions possible, while maintaining balanced partition sizes. After every vertex has been moved once, the solution with the best gain is chosen. This is repeated several times until no further improvements are found.

State-of-the-art graph partitioning libraries such as METIS [**16, 17**] and Jostle [**38**] use KL/FM for local improvement and edge-contractions based on matchings for coarsening. Recently, Holtgrewe et al. [**14**] presented a parallel library for static partitioning called KaPPa. It attains very good edge cut results, mainly by controlling the multilevel process using so-called edge ratings for approximate matchings. Recently Sanders and Osipov [**25**] and Sanders and Schulz [**27, 28**] have presented new sequential approaches for cut-based graph partitioning. They mainly employ a radical multilevel strategy, flow-based local improvement, and evolutionary algorithms, respectively.

2.2. Load Balancing by Repartitioning. To consider both a small edge-cut *and* small migration costs when repartitioning dynamic graphs, different strategies have been explored in the literature. To overcome the limitations of simple scratch-remap and rebalance approaches, Schloegel et al. [**30, 31**] 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 [**33, 39, 6**] have been developed. Also, the tools Parkway [**36**], a parallel hypergraph partitioner, and Zoltan [**5**], a suite of load balancing algorithms with focus on hypergraph partitioning, need to be mentioned although they concentrate (mostly) on hypergraphs. An efficient parallelization of the KL/FM heuristic that these parallel (hyper)graph partitioners use is 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 destroy data consistency. A coloring of the graph's vertices is used by the parallel libraries ParMETIS [**30**] and KaPPa [**14**] for this purpose.

2.3. Diffusive Methods 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 [**8**]. That is why in previous work [**24, 23**] 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 [**32**], not *which* elements should be migrated. Generally speaking, a diffusion problem consists of distributing load from some given seed vertex (or several seed 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 vertex has the same amount of load. This is one reason why diffusion has been studied extensively for load balancing [**40**]. Our algorithms Bubble-FOS/C [**23**] and the much faster DibaP [**22**] (also see Section 3) as well as a combination of KL/FM and diffusion by Pellegrini [**26**] exploit that diffusion sends load entities faster into densely connected subgraphs. This fact is used to distinguish dense from sparse graph regions. In the field of graph-based image segmentation, similar arguments are used to find well-shaped segments [**11**].

3. Diffusion-based Repartitioning with DibaP

The algorithm DibaP, which we have developed and implemented with shared memory parallelism previously [**22**], is a hybrid multilevel combination of the two (re)partitioning methods Bubble-FOS/C and TruncCons, which are both based on disturbed diffusion. We call a diffusion scheme *disturbed* if it is modified such that its steady state does not result in the balanced distribution. Disturbed diffusion schemes can be helpful to determine if two graph vertices or regions are densely connected to each other, i. e., if they are connected by many paths of small length. This property is due to the similarity of diffusion to random walks and the notion that a random walk is more likely to stay 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 components for (re-)partitioning in more detail.

3.1. Bubble-FOS/C. In contrast to Lloyd's related k-means algorithm [**19**], 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 parts. 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 [**23**] as distance measure (or, more precisely, as similarity measure). The similarity of a vertex v to a non-empty vertex 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.[1]

After the assignment step, each part 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 parts* 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 with the matrix $\mathbf{L}$, 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, the final partitions are very compact and have short boundaries. However, the repeated solution of linear systems makes Bubble-FOS/C slow.

3.2. TruncCons. The algorithm TruncCons [**22**] (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. Vertices of π_c receive an equal amount of initial load $|V|/|\pi_c|$, while the other vertices' 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) [**7**] 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 [**7**] of G. A common choice for α is $\alpha := \frac{1}{(1+\deg(G))}$. The computation $w = \mathbf{M}^{\psi} w^{(0)}$ could be realized by ψ matrix-vector products. A more localized view of its realization is given by iterative load exchanges on each vertex v with its neighbors. Then we get for $1 \leq t \leq \psi$:

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

After the load vectors have been computed this way independently for all k parts, each vertex v is assigned to the partition it has obtained the highest load from. This completes one TruncCons iteration, which can be repeated several

[1] In general $\mathbf{L}$ represents the whole graph. Yet, sparsifying the matrix in certain areas (also called *partial graph coarsening*) is possible and leads to a significant acceleration without sacrificing partitioning quality considerably [**23**]. While the influence of partial graph coarsening on the partitioning quality is low, the solutions of the linear systems become distorted and more difficult to analyze. Moreover, the programming overhead is immense. As the next section introduces a simpler and faster way of diffusive partitioning, we do not consider partial graph coarsening further here.

times (the total number is denoted by Λ subsequently) to facilitate sufficiently large movements of the parts.

A vertex 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, there are many such *inactive* vertices in the beginning. In fact, only vertices incident to the cut edges of the part under consideration are active initially. In principle each new FOS iteration adds a new layer of active vertices similar to BFS frontiers. We keep track which vertices are active and which are not. Thereby, it is possible to forego the inactive vertices when performing the local FOS calculations.

In our implementation the size of the matrix $\mathbf{M}$ for which we compute a matrix-vector product locally in each iteration is not changed. Instead, inner products involving inactive rows are not computed as we know their respective result does not change in the current iteration. That way the computational effort is restricted to areas close to the partition boundaries.

3.3. The Hybrid Algorithm PDibaP.

The main components of PDibaP, the MPI-parallel version of the original implementation of DibaP, are depicted in Figure 1. To build a multilevel hierarchy, the fine levels are coarsened (1) by approximate maximum weight matchings. Once the graphs are sufficiently small, the construction mechanism can be changed. In our sequential DibaP implementation, we switch the construction mechanism (2) to the more expensive coarsening based on algebraic multigrid (AMG)—for an overview on AMG cf. [**35**]. This is advantageous regarding running time because, after computing an initial partition (3), Bubble-FOS/C is used as local improvement algorithm on the coarse levels (4). Since AMG is well-suited as a linear solver within Bubble-FOS/C, such a hierarchy would be required for AMG anyway. In our parallel implementation PDibaP (cf. Section 4), however, due to a significant reduction of the parallel programming effort, we decided to coarsen by matchings, use a conjugate gradient solver, and leave AMG to future work.

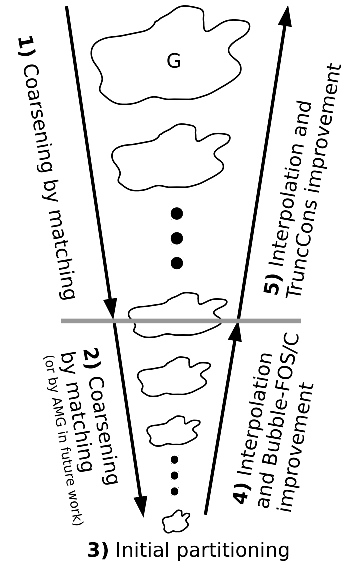
G
1) Coarsening by matching
5) Interpolation and TruncCons improvement
2) Coarsening by matching (or by AMG in future work)
4) Interpolation and Bubble-FOS/C improvement
3) Initial partitioning

Figure 1. Sketch of the combined multilevel hierarchy and the corresponding repartitioning algorithms used within PDibaP.

Eventually, the partitions on the fine levels are improved by the local improvement scheme TruncCons (5). PDibaP includes additional components, e. g., for balancing partition sizes and smoothing partition boundaries, see Section 4.3.

The rationale behind PDibaP can be explained as follows. While Bubble-FOS/C computes high-quality graph partitions with good shapes, its similarity measure FOS/C is very expensive to compute compared to established partitioning heuristics. 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.

4. PDibaP: Parallel DibaP for Repartitioning

In this section we describe our parallel implementation of DibaP using MPI. In particular we highlight some differences to the sequential (and thread-parallel) version used for static partitioning [**22**].

4.1. Distributed Memory Parallelism. The foundation of our PDibaP implementation (data structure, linear algebra routines, auxiliary functions) is to a large extent based on the code described in more detail in our previous work [**23**] and in Schamberger's thesis [**29**]. PDibaP employs as graph data structure the standard distributed compressed sparse row (CSR) format with ghost (or halo) vertices. The linear systems within Bubble-FOS/C are solved with a conjugate gradient (CG) solver using the traditional domain decomposition approach for distributed parallelism. That means that each system is distributed over all processors and solved by all of them in parallel at the same time, which requires three communication operations per iteration within CG. The TruncCons process is executed in a similar manner. To account for the inactive vertices, however, we do not perform complete matrix-vector multiplications, but perform local load exchanges only if an active vertex is involved. Both CG and TruncCons require a halo update after each iteration. This communication routine is rather expensive, so that the number of iterations should be kept small. The linear algebra routines within PDibaP do not make use of external libraries. This is due to the fact that the solution process in Bubble-FOS/C is very specialized [**23, 29**].

4.2. Repartitioning. So far, PDibaP is targeted at repartitioning dynamic graphs. The option for parallel static partitioning is still in its infancy due to a limitation in the multilevel process, which we explain later on in this section.

When PDibaP is used for repartitioning instead of partitioning, one part of its input is an initial partition. Based on this partition, the graph is distributed onto the processors. We can assume that this partition 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. Moreover, a refinement limits the number of migrated vertices as well, an important feature of dynamic repartitioning methods.

In particular if the imbalance is higher than allowed, it is advisable to employ the multilevel paradigm. Local improvements on the input graph would not result in sufficiently large movements to a high quality solution. Therefore, a matching hierarchy is constructed until only a few thousand vertices remain in the coarsest graph. So far, only edges whose endpoints lie in the same part are considered to be part of the matching. This simplifies the parallel implementation and is a viable approach when repartitioning.

After constructing the hierarchy, 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 either Bubble-FOS/C or TruncCons, depending on the size of the level. After the refinement, the current solution is interpolated to the next level until the process stops at the input level. Sometimes the matching algorithm has hardly coarsened a level. This happens for example to avoid star-like subgraphs with strongly varying vertex degrees. Limited coarsening results in two very similar adjacent levels. Local improvement with TruncCons on both of these levels would result in similar solutions with an unnecessary running time investment. That is why in such a case TruncCons is skipped on the finer level of the two.

For static partitioning, which is still an ongoing effort, edges in the cut between parts on different processors should be considered as matching edges as well. Otherwise, the multilevel hierarchy contains only a few levels after which no more edges are found for the matching. The development and/or integration of such a more general matching is part of future work.

4.3. Balancing Procedures. In general the diffusive processes employed by PDibaP do not guarentee the nearly perfect balance required by numerical simulations (say, for example, no part should be larger than the average part size plus 3%). That is why we employ two balancing procedures within PDibaP. The first one called `ScaleBalance` is an iterative procedure that tries to determine for every part $1 \leq p \leq k$ a scalar β_p with which the diffusion load values are scaled. Suitable scalars are searched such that the assignment of vertices to parts based on the load vector entries $\beta_p w_p$ results in a balanced partition. More details can be found in Meyerhenke et al. [**23**, p. 554]. While `ScaleBalance` works surprisingly well in many cases within PDibaP, it also happens that it is not fully effective even after a fairly large number of iterations. Then we employ a second approach, called `FlowBalance`, whose basic idea is described in previous work as well [**23**, p. 554]. Here we highlight recent changes necessary to adapt the approach to the distributed parallelism in PDibaP.

First, we solve a load balancing problem on the quotient graph of the partition Π. The quotient graph Q contains a vertex for each part in Π and two vertices are connected by an edge in Q if and only if their corresponding parts share a common boundary in Π. The load balancing problem can be solved with diffusion [**15**]. The solution yields the migrating flow that balances the partition. Hence, we know *how many* vertices have to be moved from π_i to π_j, let us call this number n_{ij}. It remains to be determined *which* vertices take this move. For quality reasons, this decision should be based on the diffusion values in the respective load vectors computed by Bubble-FOS/C or TruncCons. That is why we want to migrate the n_{ij} vertices with the highest values in the load vector w_j.

In our sequential and thread-parallel version of DibaP, we use a binary heap as priority queue to perform the necessary selection, migration, and resulting updates to the partition. Since parallel priority queues require a considerable effort to obtain good scalability, we opt for a different approach in PDibaP. For ease of implementation (and because the amount of computation and communication is relatively small), each processor preselects its local vertices with the highest n_{ij} load values in w_j. These preselected load values are sent to processor p_j, which performs a sequential selection. The threshold value found this way is broadcast back to all processors. Finally, all processors assign their vertices whose diffusion loads in w_j is higher than the threshold to part π_j.

This approach might experience problems when the selected threshold value occurs multiple times among the preselected candidate values. In such a case, the next larger candidate value is chosen as threshold. Another problem could be the scheduled order in which migration takes place. It could happen that a processor needs to move a number of vertices that it is about to obtain by a later move. To address this, we employ a conservative approach and move rather fewer vertices than too many. As a compensation, the whole procedure is repeated iteratively until a balanced partition is found.

5. Experiments

Here we present some of our experimental results comparing our PDibaP implementation to the KL/FM-based load balancers ParMETIS and parallel Jostle.

5.1. Benchmark Data. Our benchmark set comprises two types of graph sequences. The first one consists of three smaller graph sequences with 51 frames each, having between approximately $1M$ and $3M$ vertices, respectively. The second group contains two larger sequences of 36 frames each. Each frame in this group has approximately $4.5M$ to $16M$ vertices. These sequences result in 50 and 35 repartitioning steps, respectively. We choose to (re)partition the smaller sequences into $k = 36$ and $k = 60$ parts, while the larger ones are divided into $k = 60$ and $k = 84$ parts. These values have been chosen as multiples of 12 because one of our main test machines has 12 cores per node.

All graphs of these five sequences have a two-dimensional geometry and have been generated to resemble adaptive numerical simulations such as those occurring in computational fluid dynamics. A visual impression of some of the data (in smaller versions) is available in previous work [**23**, p. 562f.]. 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 result in unbalanced subdomain sizes. For more details the reader is referred to Marquardt and Schamberger [**20**], who have provided the generator for the sequence data.[2] Some of these frames are also part of the archive of the 10th DIMACS Implementation Challenge [**1**].

5.2. Hardware and Software Settings. We have conducted our experiments on a cluster with 60 Fujitsu RX200S6 nodes each having 2 Intel Xeon X5650 processors at 2.66 GHz (results in 12 compute cores per node). Moreover, each node has 36 GB of main memory. The interconnect is InfiniBand HCA 4x SDR HCA PCI-e, the operating system Cent OS 5.4. PDibaP is implemented in C/C++.

[2] Some of the input data can be downloaded from the website `http://www.upb.de/cs/henningm/graph.html`.

PDibaP as well as ParMETIS and parallel Jostle have been compiled with Intel C/C++ compiler 11.1 and MVAPICH2 1.5.1 as MPI library. The number of MPI processes always equals the number of parts k in the partition to be computed.

The main parameters controlling the running time and quality of the DibaP algorithm are the number of iterations in the (re)partitioning algorithms Bubble-FOS/C and TruncCons. For our experiments we perform 3 iterations within Bubble-FOS/C, with one `AssignPartition` and one `ComputeCenters` operation, respectively. The faster local approach TruncCons is used on all multilevel hierarchy levels with graph sizes above 12,000 vertices. For TruncCons, the parameter settings $\Lambda = 9$ and $\psi = 14$ for the outer and inner iteration, respectively. These settings provide a good trade-off between running time and quality. The allowed imbalance is set to the default value 3% for all tools.

5.3. Results. In addition to the graph partitioning metrics edge-cut and communication volume (of the underlying application based on the computed partition), we are also interested in migration costs. These costs result from data changing their processor after repartitioning. We count the number of vertices 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 PDibaP to the state-of-the-art repartitioning tools ParMETIS and parallel Jostle. Both competitors are mainly based on the vertex-exchanging KL heuristic for local improvement. The load balancing toolkit Zoltan [**5**], whose integrated KL/FM partitioner is based on the hypergraph concept, is not included in the detailed presentation. Our experiments with it indicate that it is not as suitable for our benchmark set of FEM graphs, in particular because it yields disconnected parts which propagate and worsen in the course of the sequence. We conclude that currently the dedicated graph (as opposed to hypergraph) partitioners seem more suitable for this problem type.

The partitioning quality is measured in our experiments by the edge cut (EC, a summation norm) and the maximum communication volume ($\mathrm{CV}_{\max}$). $\mathrm{CV}_{\max}$ is the sum of the maximum incoming communication volume and the maximum outgoing communication volume, taken over all parts, respectively. The values are displayed in Table 1, averaged over the whole sequence and aggregated by the different k. Very similar results are obtained for the geometric mean in nearly all cases, which is why we do not show these data as well. The migration costs are recorded in both norms and shown for each sequence (again aggregated) in Table 2. Missing values for parallel Jostle (—) indicate program crashes on the corresponding instance(s).

The aggregated graph partitioning metrics show that PDibaP is able to compute the best partitions consistently. PDibaP's advance is highest for the communication volume. With about 12–19% on parallel Jostle and about 34–53% on ParMETIS these improvements are clearly higher than the approximately 7% obtained for static partitioning [**22**], which is due to the fact that parallel KL (re)partitioners often compute worse solutions than their serial counterparts for static partitioning.

Table 1. Average edge cut and communication volume (max norm) for repartitionings computed by ParMETIS, Jostle, and PDibaP. Lower values are better, best values per instance are written in bold.

Sequence	ParMETIS		Par. Jostle		PDibaP	
	EC	$\mathrm{CV}_{\max}$	EC	$\mathrm{CV}_{\max}$	EC	$\mathrm{CV}_{\max}$
biggerslowtric	11873.5	1486.7	9875.1	1131.9	**8985.5**	**981.8**
biggerbubbles	16956.8	2205.3	14113.2	1638.7	**12768.3**	**1443.5**
biggertrace	17795.6	2391.1	14121.3	1687.0	**12229.2**	**1367.5**
hugetric	34168.5	2903.0	28208.3	2117.6	**24974.4**	**1766.2**
hugetrace	54045.8	5239.7	–	–	**34147.4**	**2459.4**

Table 2. Average migration volume in the ℓ_1- and ℓ_∞-norm for repartitionings computed by ParMETIS, Jostle, and PDibaP. Lower values are better, best values per instance are written in bold.

Sequence	ParMETIS		Par. Jostle		PDibaP	
	ℓ_∞	ℓ_1	ℓ_∞	ℓ_1	ℓ_∞	ℓ_1
biggerslowtric	**60314.3**	606419.1	64252.2	557608.7	65376.1	**550427.0**
biggerbubbles	77420.0	1249424.3	**68865.1**	**791723.6**	93767.5	1328116.1
biggertrace	54131.2	733750.4	49997.8	**533809.2**	**46620.4**	613071.2
hugetric	**231072.8**	2877441.8	244082.5	2932607.6	232382.6	**2875302.5**
hugetrace	**175795.8**	**3235984.1**	–	–	189085.3	3308461.4

The results for the migration volume are not consistent. All tools have a similar amount of best values. The fact that ParMETIS is competetitive is slightly surprising when compared to previous results [**21**], where it compared worse. Also unexpectedly, PDibaP shows significantly higher migration costs for the instance biggerbubbles. Our experiments indicate that PDibaP has a more constant migration volume, while the values for parallel Jostle and ParMETIS show a higher amplitude. It depends on the instance which strategy pays off. This behavior is shown in Figure 2. It displays the migration volumes in the ℓ_∞-norm for each frame within the benchmark sequence called *slowrot*, which is smaller but similar to the ones used in our main experimental study.

These results lead to the conclusion that PDibaP's implicit optimization with the iterative algorithms Bubble-FOS/C and TruncCons focusses more on good partitions than on small migration costs. In some 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 PDibaP.

It is interesting to note that further experiments indicate a multilevel approach to be indeed necessary in order to produce sufficiently large partition movements

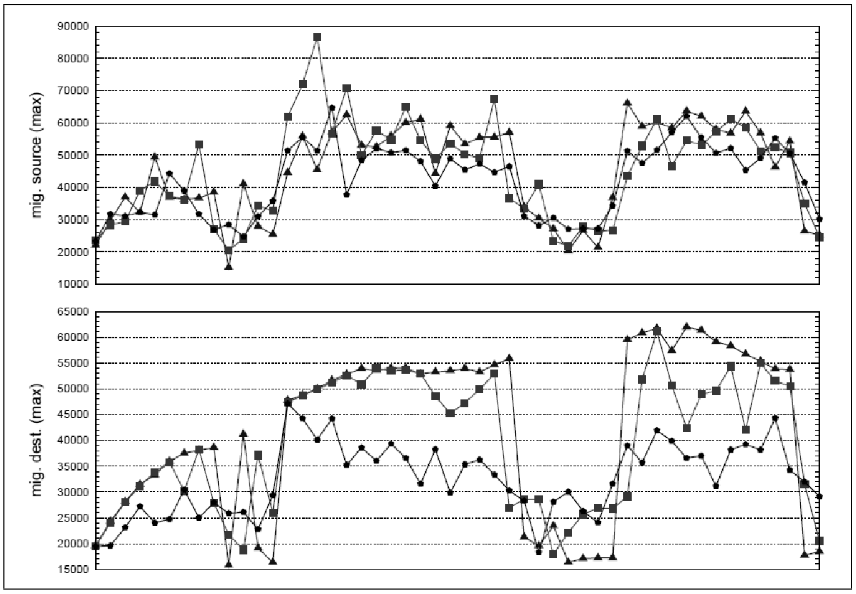

Figure 2. Number of migrating vertices (ℓ_∞-norm) in each frame of the biggertrace sequence for PDibaP (circles), METIS (triangles), and Jostle (squares). Lower values are better.

Table 3. Average running times in seconds for the benchmark problems. Lower values are better, best values per instance are written in bold. The values marked by * denote averaged times (or, in case of –, incomparable values) where parallel Jostle did not finish the whole sequence due to a premature crash.

Sequence	ParMETIS		Par. Jostle		PDibaP	
	$k = 36$	$k = 60$	$k = 36$	$k = 60$	$k = 36$	$k = 60$
biggerslowtric	**0.27**	**0.22**	0.50	0.88	8.71	10.38
biggerbubbles	**0.38**	**0.30**	0.79	1.24	15.02	19.19
biggertrace	**0.33**	**0.26**	0.56	0.59	9.27	10.77
	$k = 60$	$k = 84$	$k = 60$	$k = 84$	$k = 60$	$k = 84$
hugetric	**0.68**	**0.64**	2.41*	4.68*	55.36	62.37
hugetrace	**0.85**	**0.76**	–	–	50.56	56.69

that keep up with the movements of the simulation. Partitions generated by multilevel PDibaP are of a noticeably higher quality regarding the graph partitioning metrics than those computed by TruncCons without multilevel approach. Also, maybe surprisingly, using a multilevel hierarchy results in steadier migration costs.

The running time of the tools, depicted in Table 3, for the dynamic graph instances used in this study can be characterized as follows. ParMETIS is the fastest, taking from a fraction of a second up to a few seconds for each frame, with the average always being below one second. Parallel JOSTLE is approximately a factor of 2-4 slower than PARMETIS (without counting sequences where parallel JOSTLE crashed prematurely). PDIBAP, however, is significantly slower than both tools, with an average slowdown of about 28-97 compared to PARMETIS. It requires from a few seconds up to a few minutes for each frame, with the average being 10-20 seconds for the small benchmarks and about a minute for the large ones.

The scalability of PDIBAP is not good due to the linear dependence on k in the running time. PARMETIS is able to profit somewhat from more processors regarding execution time. PDIBAP and parallel JOSTLE, however, become slower with increasing k. Neglecting communication, the running time of PDIBAP should remain nearly constant for growing k when it computes a k-partitioning with k processors. However, in this parallel setting the communication overhead yields growing running times. Therefore, one can conclude that PDIBAP is more suitable for simulations with a small number of processors.

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. Nevertheless, a further acceleration of shape-optimizing load balaincing is of utmost importance. Minutes for each repartitioning step might be problematic for some targeted applications.

6. Conclusions

With this work we have demonstrated that the shape-optimizing repartitioning algorithm DIBAP based on disturbed diffusion can be a good alternative to traditional KL-based methods for balancing the load in parallel adaptive numerical simulations. In particular, the parallel implementation PDIBAP is very suitable for simulations of small to medium scale, i. e., when the number of vertices and edges in the dynamic graphs are on the order of several millions. While PDIBAP is still significantly slower than the state-of-the-art, it usually computes considerably better solutions w. r. t. edge cut and communication volume. 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 PDIBAP is expected to pay off.

As part of future work, we aim at an improved multilevel process and faster partitioning methods. It would also be worthwhile to investigate if BUBBLE-FOS/C and TRUNCCONS can be further adapted algorithmically, for example to reduce the dependence on k in the running time.

References

[1] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. 10th DIMACS implementation challenge. http://www.cc.gatech.edu/dimacs10/, 2012.

[2] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Proceedings of the 10th DIMACS Implementation Challenge*, Contemporary Mathematics. American Mathematical Society, 2012.

[3] N. A. Baker, D. Sept, M. J. Holst, and J. A. McCammon. The adaptive multilevel finite element solution of the Poisson-Boltzmann equation on massively parallel computers. *IBM J. of Research and Development*, 45(3.4):427 –438, May 2001.

[4] U. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed System*, 10(7):673–693, 1999.

[5] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.*, 69(8):711–724, Aug. 2009.

[6] C. Chevalier and F. Pellegrini. PT-scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, 2008.

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

[8] 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.

[9] 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.

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

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

[12] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, 2000.

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

[14] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *IPDPS*, pages 1–12. IEEE, 2010.

[15] Y. F. Hu and R. F. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25(4):417–444, 1999.

[16] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*. Univ. of Minnesota, Minneapolis, MN, 1998.

[17] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

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

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

[20] 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)*, pages 685–691. CSREA Press, 2005.

[21] H. Meyerhenke. Dynamic load balancing for parallel numerical simulations based on repartitioning with disturbed diffusion. In *Proc. Internatl. Conference on Parallel and Distributed Systems (ICPADS'09)*, pages 150–157. IEEE Computer Society, 2009.

[22] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009. Best Paper Awards and Panel Summary: IPDPS 2008.

[23] H. Meyerhenke, B. Monien, and S. Schamberger. Graph partitioning and disturbed diffusion. *Parallel Computing*, 35(10–11):544–569, 2009.

[24] 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.

[25] V. Osipov and P. Sanders. *n*-level graph partitioning. In *Proc. 18th Annual European Symposium on Algorithms (ESA'10)*, pages 278–289, 2010.

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

[27] P. Sanders and C. Schulz. Engineering multilevel graph partitioning algorithms. In *Proc. 19th Annual European Symposium on Algorithms (ESA'11)*, pages 469–480, 2011.

[28] P. Sanders and C. Schulz. Distributed evolutionary graph partitioning. In *Meeting on Algorithm Engineering & Experiments (ALENEX'12)*. SIAM, 2012.

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

[30] 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.

[31] 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.

[32] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001.

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

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

[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] A. Trifunović and W. J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.*, 68(5):563–581, 2008.

[37] 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)*, pages 611–614. SIAM, 1995.

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

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

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

Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany

E-mail address: meyerhenke@kit.edu