
ELSEVIER

Parallel Computing 26 (2000) 1555–1581

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM ☆

Ralf Diekmann [a,*], Robert Preis [b], Frank Schlimbach [c], Chris Walshaw [c]

[a] *Hilti AG, Corp. Research, Schaan, Liechtenstein*

[b] *Department of Mathematics and Computer Science, University of Paderborn, Fürstenallee 11, D-33102 Paderborn, Germany*

[c] *School of Computing and Mathematical Sciences, The University of Greenwich, 30 Park Row, Greenwich, London SE10 9LS, UK*

Received 15 February 1999; received in revised form 24 September 1999; accepted 24 September 1999

Abstract

We present a dynamic distributed load balancing algorithm for parallel, adaptive Finite Element simulations in which we use preconditioned Conjugate Gradient solvers based on domain-decomposition. The load balancing is designed to maintain good partition *aspect ratio* and we show that *cut size* is not always the appropriate measure in load balancing. Furthermore, we attempt to answer the question why the *aspect ratio* of partitions plays an important role for certain solvers. We define and rate different kinds of *aspect ratio* and present a new center-based partitioning method of calculating the initial distribution which implicitly optimizes this measure. During the adaptive simulation, the load balancer calculates a balancing flow using different versions of the diffusion algorithm and a variant of breadth first search. Elements to be migrated are chosen according to a cost function aiming at the optimization of subdomain shapes. Experimental results for Bramble's preconditioner and comparisons to state-of-the-art load balancers show the benefits of the construction.

Keywords: Mesh partitioning; Load balancing; Shape-optimization; Aspect ratio; Parallel; Adaptive; Finite element method

☆ Parts of the results appeared in the Proceedings of IRREGULAR'98 and EUROPAR'98 (Springer LNCS 1457 and 1470).

* Corresponding author.

E-mail addresses: diekral@hilti.com (R. Diekmann), robsy@upb.de (R. Preis), F.Schlimbach@gre.ac.uk (F. Schlimbach), C.Walshaw@gre.ac.uk (C. Walshaw).

0167-8191/00/$ - see front matter
PII: S0167-8191(00)00043-0

1. Introduction

Finite elements (or finite differences or finite volumes) can be used to numerically approximate the solutions of partial differential equations (PDEs). The PDEs describe, for example, the flow of air around a wing or the distribution of temperature on a plate which is partially heated [3,9]. The domain on which the PDE has to be solved is discretized into a mesh of *finite elements* (triangles or rectangles in 2D, tetrahedra or hexahedra in 3D) and the PDE is transformed into a set of linear equations defined on these elements [45]. The coupling between equations is given by the adjacencies in the mesh. Usually, iterative methods such as conjugate gradient (CG) or multigrid (MG) are used to solve the linear systems [3,35]. The quality of solutions obtained by such numerical approximation algorithms heavily depends on the accuracy of the discretization. In particular, in regions with steep solution gradients, the mesh has to be refined sufficiently, i.e., the elements have to be small in order to allow accurate approximation. Unfortunately, the regions with large gradients are usually not known in advance. Hence, the meshes either have to be refined regularly with the consequence that there is a large number of small elements in regions where they are not required, or the refinement takes place during the calculation based on error estimates of the current solution [41]. Obviously, the second variant is to be favored. Its solutions should have the same quality as if the mesh was refined regularly but the number of elements and, thus, the time needed to determine the solution is only a small fraction of the regular case.

The parallelization of numerical simulation algorithms usually follows the single-program multiple-data (SPMD) paradigm: The same code is executed on each processor but on different parts of the data. This means that the mesh is partitioned into P subdomains where P is the number of processors, and each subdomain is assigned to one processor [9,11]. Because iterative solution algorithms mainly perform local operations, i.e., data dependencies are defined by adjacencies in the mesh, [1] the parallel algorithms only require communication at the partition boundaries. The parallel efficiency depends on two factors: an equal distribution of data (computational load) on the processors, and a small communication overhead achieved by minimizing the boundary length.

Together with the additional constraint of minimizing the number of cut edges (the total interface length in the case of FE-mesh partitioning), the mesh partitioning problem turns out to be *NP*-complete [18], i.e. it is currently not solvable to optimality in a reasonable amount of time. Fortunately, a number of quite efficient graph (mesh) partitioning heuristics (approximation algorithms) have been developed [12,14,22,27,28,32,38]. Most of them optimize the *balance* of subdomains (their deviation from the mean number of elements) and the number of cut edges, the *cut size*. Optimizing for cut size is sufficient for many applications such as standard iterative

[1] Modulo some global dependencies in certain iterative schemes. We do not consider these since they are independent of the way the mesh is partitioned.

equational solvers for PDE problems. However, there are certain other cases where this is not true. If, for example, the decomposition is used to construct pre-conditioners [3,5], *cut* and *balance* may no longer be the only factors which determine the efficiency. In addition to a significant amount of time spent on the solution of interface problems, the *shape* of subdomains heavily influences the quality of preconditioning and, thus, the overall execution time [40]. First attempts at optimizing the aspect ratio (AR) of subdomains rate elements depending on their distance from the center of a subdomain [7,11] and include this into the cost function of a local iterative search heuristic such as the Kernighan–Lin algorithm (KL) [16]. In this paper we use a simple, center-based partitioning heuristic which optimizes subdomain shapes implicitly.

In the case of adaptive refinement, the distribution of data on the processors will become unbalanced if the number of newly generated elements is not the same on each processor (as is usually the case with solution adaptive refinement). Therefore, the partition has to be altered in order to re-establish a balanced distribution. A number of solutions to this load balancing problem are based on re-partitioning, where (sometimes even sequential) mesh partitioning algorithms are used [30]. We propose a two-phase distributed load balancing algorithm which takes the existing mesh partition into account. The first phase determines the amount of load that has to be moved between different subdomains in order to balance the distribution globally. The adjacencies between subdomains defines the quotient graph [11]. The algorithm determines a *balancing flow* on this graph. The flow tells the processors how many data items (i.e., elements) they have to move to each of their neighbors. This balancing flow calculation can optionally use different kinds of diffusive methods [4,8], in particular first and second order diffusion iterations [13,19], and a heuristic for convex non-linear min-cost flow [1].

In the second phase, the elements that have to be moved are identified. When choosing these elements, the load balancer tries to optimize partition AR in addition to load balance. The elements at partition boundaries are weighted by a cost function consisting of several components. The migration chooses elements according to their weight and moves them to neighbors. The element weight functions are used to "guide" the balancing flow calculation. We consider the existing partition and define weights for the edges of the quotient graph expressing a kind of "cost" for moving elements over the corresponding borders. The aim is to avoid a large flow of elements between parts with a small common border.

The main contributions of this paper are:

- We attempt to answer the question why cut size (interface length) might not always be the right measure in balancing adaptive meshes.
- We present numbers of iterations of parallel domain-decomposition preconditioned conjugate gradient solvers (DD–PCG) on irregular adaptive meshes.
- We investigate different graph partitioning algorithms with respect to AR and discuss a center-oriented method which optimizes this measure implicitly.
- We discuss the problem of dynamic load balancing for solvers using adaptive refinement and present new element migration strategies aimed at optimizing subdomain AR.

- We modify existing balancing flow algorithms to support the shape optimization task.

All results presented apply (so far) to 2D meshes. Some possible extensions to the 3D case are discussed in Section 5. The partitioning and load balancing algorithms are included in parallel adaptive FEM (PadFEM). This is an object-oriented environment which supports parallel adaptive numerical simulations [3,9]. It includes graphical user interfaces, mesh generators for 2D and 3D domains, mesh partitioning algorithms [12], triangular and tetrahedral element formulations for Poisson and Navier–Stokes problems [45], different solvers (especially DD–PCGs) [3,5], error estimators [41], mesh refinement algorithms [26], and load balancers [13].

The next section gives an introduction to the field of balancing adaptive meshes, DD-preconditioning, and shape of subdomains. A short overview of existing approaches is given too. Section 3 introduces us to a center-oriented method for the initial load distribution which implicitly optimizes the AR. Comparisons with results from the Chaco [21], Jostle [44] and Metis [27] partitioning tools are discussed. Section 4 considers the dynamic load balancing problem for parallel adaptive DD-preconditioners. It starts with the description of element migration strategies in Section 4.1. This second load balancing phase is presented first, since parts of it are used in Section 4.2 where the balancing flow calculation is described. Section 4.3 finishes the paper with a number of results of AR and iteration numbers of the DD–PCG solver of PadFEM. Comparisons are made to the partitioning tool Jostle [44].

2. The problem

2.1. Balancing adaptive meshes

In most cases, the computational load of a finite element problem can be measured in terms of numbers of elements. Thus a load balanced parallel execution requires a partition of the mesh. The mesh is partitioned into subdomains each of which contains the same number of elements. Such an initial partition can be generated by the use of any of the existing efficient graph partitioning algorithms. If the mesh is refined adaptively, the existing partition will usually become unbalanced. Several existing implementations of parallel adaptive grid applications solve this problem by repartitioning the mesh. This is done by the use of any of the aforementioned partitioning methods [30]. The drawback of such an approach is twofold: first, the mesh has to be routed to a single processor if the partitioning tool is sequential. Such an approach is obviously not scalable to large numbers of processors and to large meshes. Second, even if a parallel partitioning tool is available (such as ParMetis [37] or PJostle [44]), a new partition may differ greatly from the existing one. As a result, large amounts of data may have to be shifted between processors. Although there are attempts to minimize this data-movement [30], comparisons to approaches which take the existing distribution into account show that if the mesh adaption changes the mesh only slightly, only a small fraction of the data movement is really necessary [44].

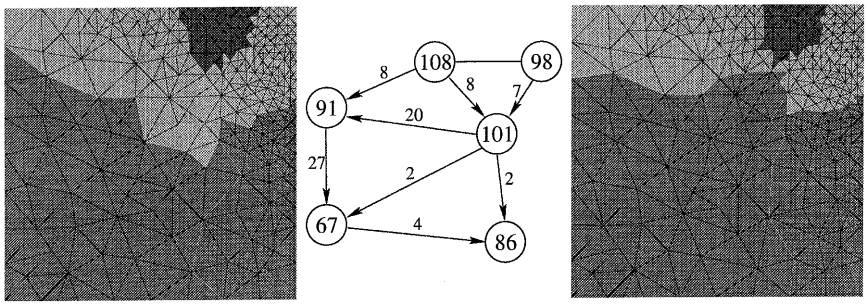


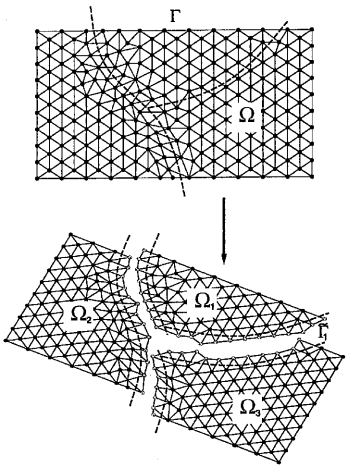
Fig. 1. An unbalanced partition (left), the corresponding quotient graph with balancing flow (center), and the resulting balanced partition after migration (right).

The load balancing module in PadFEM follows a diffusive strategy [10,13,37] applied to the quotient graph from a given partition. The graph is defined by the adjacencies between subdomains and contains one node for each subdomain. Edges denote common borders between the corresponding subdomains. Fig. 1 shows an unbalanced partition of a simple mesh (domain "square") into six subdomains (left) and the resulting quotient graph (center). Any reasonable load balancing which takes the existing distribution into account has to move data (elements) via the edges of this graph. We add weights to nodes of the graph according to their load and determine a balancing flow on the edges. This flow has the property that the partition is globally balanced after shifting a corresponding amount of load (elements) between adjacent subdomains. The flow is given as edge labels in Fig. 1. It can be determined by the use of network flow algorithms (which are – unfortunately – usually not parallel) or by local iterative methods such as diffusion or dimension exchange [4,8], which are parallel by definition. Second, in the element migration phase, the load is actually moved. Input to this phase is the flow on the quotient graph. The task is to choose element migrations in order to fulfill the flow demands. Their choice may consider additional cost criteria such as minimizing the boundary lengths or optimizing the shape of subdomains.

2.2. Preconditioning by domain decomposition

We briefly describe how preconditioning techniques based on domain decomposition work, based on our own implementation of the BPS algorithm of Bramble et al. [5]. There are several variants such as, e.g., the conjugate gradient boundary iteration method (CGBI) by Blazy et al. [3] which improve the BPS algorithm. From the algorithmic point of view though, they only slightly differ from the original method. Let us consider a Poisson problem with homogeneous Dirichlet boundary conditions (Fig. 2 (left)).

$$\begin{aligned} -\Delta u(x,y) &= f(x,y) \quad \forall (x,y) \in \Omega, \\ u(x,y) &= 0 \quad \forall (x,y) \in \Gamma. \end{aligned} \tag{1}$$



PCG

$x := 0,\ r := Ax - b,\ s := B^{-1}r,$
$p := -s,\ \delta_0 :=< r, s >$
if $\delta_0 < \epsilon \quad \Rightarrow$ STOP
while $(\delta_1 > \epsilon)$ {
$h := Ap$
$\tau := \delta_0 / < p, h >$
$x := x + \tau p$
$r := r + \tau h$
$s := B^{-1}r \qquad (*)$
$\delta_1 := < s, r >$
$\alpha := \delta_1 / \delta_0$
$p := -s + \alpha p$
}

Fig. 2. Domain Ω with boundary Γ, decomposition and the preconditioned CG algorithm (cf. e.g. [3]).

This kind of elliptic partial differential equation is derived from a discretization of the Navier–Stokes equation using splitting methods (pressure correction, [3]). Using standard finite element formulations, Eq. (1) is transformed into a system $Au = f$ of linear equations where the structure of A corresponds to the structure of the mesh. In order to increase the speed of convergence of the normal conjugate gradient iterative solver (CG), the linear system is preconditioned with a matrix B [45] (cf. Fig. 2):

$$B^{-1}Au = B^{-1}f. \tag{2}$$

The parallelization of the CG solver is usually based on domain decomposition (cf. Fig. 2) where the mesh (the domain) is split into a number of subdomains which are assigned to different processors.

The *domain decomposition preconditioning* is based on the BPS algorithm and solves the equation marked (*) in Fig. 2 in three steps:

(1) For each subdomain Ω_i it first computes

$$-\Delta v = r \quad \text{in } \Omega_i \quad \text{with } v = 0 \text{ on } \tilde{\Gamma},$$

where r is the residuum and $\tilde{\Gamma}$ are the artificial (inner) boundaries. This leads to a system of linear equations $Av = r$ within each subdomain which are solved using a sequential V-cycle multigrid.

(2) The next step is to compute a *Laplace Beltrami problem* [3]

$$(-\Delta)^{-1/4} w = r_\Gamma + [v]_{\tilde{\Gamma}}$$

with homogeneous boundary condition, where $[v]$ computes the jump over $\tilde{\Gamma}$ of v. This leads us to a system of linear equations with a dense matrix $\mathbf{C}w = [v]_{\tilde{\Gamma}}$. $\mathbf{C}$ can be approximated by a tridiagonal matrix $\tilde{\mathbf{C}}$ which simplifies the task of calculating the inverse $\tilde{\mathbf{C}}^{-1}$ (see [3]).

(3) The last step uses w to compute

$$-\Delta\hat{v} = 0 \text{ in } \Omega_i \quad \text{with } \hat{v} = w \text{ on } \tilde{\Gamma}.$$

This again leads us to a system of linear equations which can also be solved by the use of a V-cycle multigrid on each subdomain Ω_i.

The solution s of (*) (Fig. 2) is now given by $s = v + \hat{v}$ on all subdomains. For each iteration of the global CG method, Steps 1–3 have to be performed. Fig. 3 illustrates the DD–PCG algorithm on a simple square domain Ω split into two parts. The first picture shows the solutions on Ω_i with $\tilde{\Gamma} = 0$ (Step 1). In the second, the solution of the *Laplace Beltrami problem* on $\tilde{\Gamma}$ can be seen and the third presents the result after Step 3. The rightmost picture shows the final solution after four global iterations with a residual of $|r|_{\max} \leqslant 10^{-6}$.

The time that is needed by such a preconditioned CG solver is determined by two factors. First, the maximum time that is needed by any of the subdomain solutions and second, the number of iterations of the global CG. Both factors are at least partially determined by the shape of the subdomains. While the MG as solver on the Ω_i's is relatively resistant to shape, the number of global iterations is heavily influenced by the AR of the subdomains. In a way, the subdomains can be regarded as elements of the "interface" problem [16]. And just as with normal FEM, where the condition of A is influenced by the AR of elements, the condition of $B^{-1}A$ is influenced by the subdomains' AR in the preconditioning case. If rectangular shaped domains (and subdomains) are used, the relation between shape and number of global iterations can be expressed by the easiest definition of AR – the ratio between longest and shortest boundary edge. An example is presented in Fig. 4 where a Poisson problem is solved with DD–PCG on a regular mesh. The domain is split into an increasing number of slices with a resulting increase in AR. It becomes obvious that the number of global iterations grows heavily with the AR. To show that this is not caused by the increasing number of subdomains P, the lower curve gives the number of iterations when only P is growing and the AR is kept constant. It can be seen that in this case the number of iterations remains almost constant (although in fact the problem size is increasing).

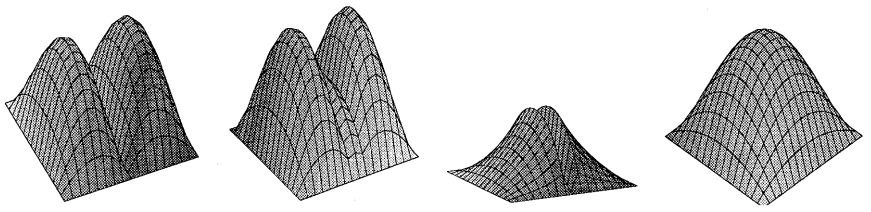


Fig. 3. The three steps of the DD–PCG algorithm (left to right): Subdomain solution with zero boundary conditions, interface solution, subdomain solutions with new boundary conditions. The rightmost illustration shows the solution after 4 iterations.

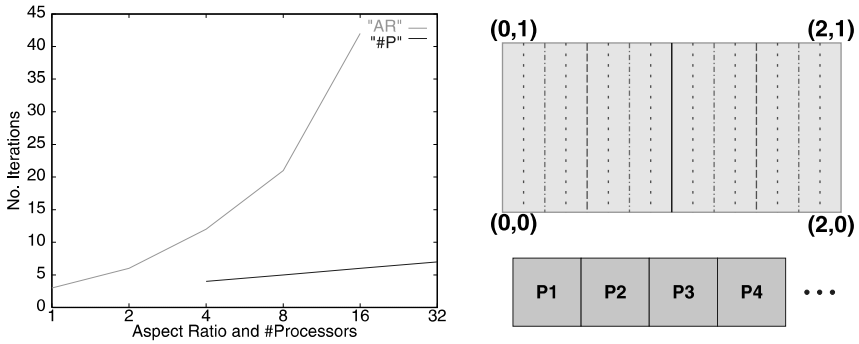


Fig. 4. # Iterations vs AR for DD–PCG. A rectangular domain recursively split into an increasing number of slices (top right). A growing rectangular domain with subdomains of equal shape for each processor (bottom right).

2.3. Aspect ratios

Possible definitions of AR can be found in Fig. 5. The first two are motivated by common measures in triangular mesh generation where the quality of triangles are expressed in either $L_{\max}/L_{\min}$ (longest to shortest boundary edge) or R_o^2/R_i^2 (the area of smallest circle containing the domain to the area of largest inscribed circle). The definition $\mathrm{AR} = R_o^2/R_i^2$ expresses the fact that circles are perfect shapes. Unfortunately, circles are quite expensive to find for arbitrary polygons: by the use of Voronoi-diagrams, we can determined both in $O(2n \log n)$ steps where n is the number of nodes of the polygon (and faster incremental update algorithms are not known). The definition $\mathrm{AR} = R_o^2/A$ (A being the area of the domain) is another measure that favors circle-like shapes. It still requires the determination of the smallest outer circle but turns out to be better in practice. We can do a further step and approximate R_o by the length B of the boundary of the domain (which can be determined fast and updated incrementally in O(1)). For a sub-domain with area A and perimeter B, $\mathrm{AR} = B^2/16A$ is the ratio between the area of a square with perimeter B and area A. This definition assumes that squares are perfect domains.

Circles offer a better perimeter/area ratio but force neighboring domains to become concave (Fig. 6 (left)). The example in Fig. 4 has already used the first definition of AR : $\mathrm{AR} = L_{\max}/L_{\min}$. This measure does not express the *shape* properly for

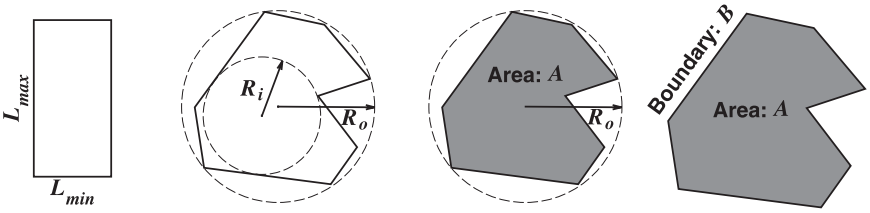


Fig. 5. Different definitions of AR: $L_{\max}/L_{\min}, R_o^2/R_i^2, R_o^2/A$ and $B^2/16A$.

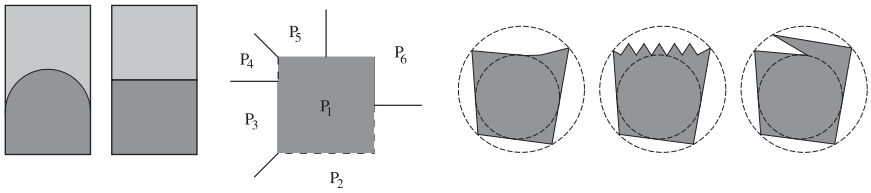


Fig. 6. Problems of several definitions of AR.

irregular meshes and partitions. Fig. 6 (center) presents an example. P_1 is perfectly shaped, but as the boundary towards P_4 is very short, $L_{\max}/L_{\min}$ is large. The circle-based measures usually fail to rate jagged boundaries or inscribed corners. Fig. 6 (right) displays examples each of which have the same AR but which are very different in shape. According to our experience (and also from the examples), $\mathrm{AR} = R_{\mathrm{o}}^2/A$ and $\mathrm{AR} = B^2/16A$ turn out to be the most robust measures, which best express the desired aims of producing compact domains.

2.4. Existing approaches

A large number of methods for the graph partitioning problem has been developed in different research and application fields. We briefly discuss some of the methods which are frequently applied for FE mesh partitioning.

The coordinate sorting (COO) method (see, e.g., [15]) is based on the vertex coordinates only and is very simple and easy to implement. The mesh is cut by straight lines perpendicular to the axis of the longest elongation of the mesh. The result is a stripe-wise partition. The method may also be applied recursively (COO_R), resulting in a more box-wise partition. Although this approach does not consider any connectivity information of the graph, for some application graphs like, e.g., FE meshes of simple domains, it results in parts with reasonable shapes. *Greedy* approaches are often based on the graph connectivity. Typically, the first part of a partition is initialized with one single element and further elements are added until the required size is reached. Then, a new part is initialized with an unassigned element and the new part is build up in the same greedy fashion. One possibility of choosing new elements is to repeatedly take all non-assigned elements adjacent to elements of the current subdomain, i.e. progressing in a breath-first manner (GBF, see e.g. [14]). Another possibility is to choose an element which reduces the cut most of all (GCF, see e.g. [12,38]). The greedy approach usually results initially in very compact subdomains, but often the last subdomain consists of all leftover elements and it is thus very unlikely that it has a smooth shape. More elaborate initial partitioners use connectivity measures based on the second smallest eigenvalue of the graph's Laplacian. These so called *spectral methods* [32] are quite expensive, but combined with fast multi-level contraction schemes they belong to the state-of-the-art in graph partitioning software [21].

Once a partition is calculated, one can use a local improvement method to further optimize the cut size. The KL heuristic [28] is the most frequently used local

improvement method. It uses a sequence of logical vertex pair exchanges to determine the sets that have to be exchanged physically. Fiduccia and Mattheyses [17] have modified the method. They use a sequence of single vertex moves to determine the sets. Meanwhile similar to the KL algorithm, the Helpful-Sets (HS) heuristic is based on local rearrangements [12]. It, too, has to search for two sets of equal size (one in each part), which will improve the cut if they are exchanged. Unlike KL, it does not only consider single vertices but also whole sets that take part in the exchange steps.

There are several mesh partitioning software libraries and most of them are freely available for academic research. The most popular examples are Chaco [21] by Hendrickson and Leland which includes *inertial* and spectral partitioning [22,32] as well as multilevel-strategies [2,23], Metis [27] by Karypis and Kumar, which includes fast multilevel strategies, Scotch [31] by Pellegrini and Roman, which includes mapping facilities, Top/Domdec [15] by Farhat et al., Jostle [43,44] by Walshaw et al. or Party [33] by Diekmann and Preis.

The tools Metis and Jostle are also designed to support partitioning and load balancing of adaptive mesh calculations in parallel. Both use the algorithm of [24] in order to determine the balancing flow. What is more, both use a multilevel strategy for shifting elements, where optionally the coarsening is only done inside subdomains. Jostle uses the concept of *relative gain* optimization at partition boundaries. Metis uses so called *enforcement levels* and greedy refinement. Both tools optimize according to the number of edges crossing the partition boundaries. The AR is not directly considered. Load balancers particularly designed to optimize the subdomain AR can be found in [11,16,39,42]. The tool PAR^2 was originally constructed as a parallel partitioner, but it can also be used in an adaptive environment [11]. The method described in [16] is an iterative partitioner which tries to improve the subdomain AR in a number of steps. In this work, DD-PCG solvers are used, but not in an adaptive environment. Other attempts are to optimize subdomain shapes by the use of meta-heuristics such as simulated annealing (SA) [25] or Tabu Search [39].

3. Partitioning the initial mesh

Mesh (graph) partitioning is an area of active research. As mentioned above, the problem of partitioning a graph into a number of equally sized parts such that the number of cut edge is minimized is NP-complete [18]. Nevertheless, the initial mesh of a parallel adaptive FEM simulation is not usually excessively large and will be refined throughout the simulation. In this section we propose a center-oriented method called bubble (BUB) for partitioning the initial mesh which implicitly optimizes the shape of subdomains. Some of the ideas this method is based on are very simple and natural and it has some similarities with different other approaches. Bubble generalizes some ideas of the bisection growing method of [38]. A similar center-based approach has been developed in [20] and a parallel center-based approach can be found in [43]. An anonymous referee pointed out the similarities to an

algorithm for vector quantizer design [29] where the vector entries are partitioned while considering a reproduction alphabet with as many elements as there are parts.

The idea of bubble (displayed in Fig. 7) is to represent a partition by a set of seed vertices, one for each part, from which the subdomains are grown simultaneously in a breadth-first manner until the whole mesh has been covered. Colliding parts form a common border and keep on growing along this border – just like soap bubbles in a bath. After the whole mesh is covered, the algorithm determines its "center" vertex for each part. This is defined as the new seed and the subdomain growing process starts again. The iteration will be stopped if the movement of all seeds is small enough, i.e., if the seed vertices are close to the centers for all parts. The algorithm is based on the observation that within "perfect" bubbles, the center and the seed vertex coincide. The distances in this method may either be chosen as the path length or as the Euclidean distances. In the case of path length the method works also on graphs without geometrical information.

The Bubble-algorithm is shown in Fig. 8. In order to find the initial seeds, we start a breadth-first search (BFS) from a vertex with minimal degree (in the case of FE-meshes, this is usually an element at a domain corner) and search for the vertex which is farthest from this starting point. This vertex is chosen as the first seed. We

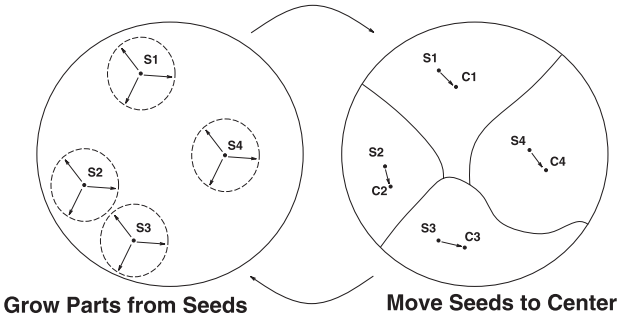


Fig. 7. The iterative Bubble method.

perform breath-first search from an element v of minimal degree;
take an element with furthest distance to v as seed of part 1;
FOR i from 2 to P DO
    perform breath-first search from seeds 1,...,$i-1$ simultaneously;
    take an element with furthest distance as seed of part i;
ENDFOR
DO Grow parts from seeds in breath-first manner;
    Calculate centers of parts and assign them as new seeds;
UNTIL (the seeds do not change OR the AR did not improve for 10 iter.)

Fig. 8. The Bubble algorithm.

then repeat to perform simultaneous BFS from all seeds that have been found so far to determine a vertex which is farthest from all seeds. It becomes the next seed. Altogether, P BFSs are performed with P being the number of parts. With this approach of each new seed having the maximum distance from all previous ones we distribute the seeds fairly evenly over the graph. The path length is used as distance measure in the first loop. The main loop of Bubble is started by growing the parts from each seed in a breadth-first manner, i.e., each part checks if any of its elements is adjacent to an uncovered element and the smallest part with at least one such adjacent element grabs the one with the shortest Euclidean distance to its seed and assigns it to that part. Only vertices are added which are adjacent to vertices previously assigned to the same part, ensuring connected parts. The ordering of smallest parts tries to keep the final load difference small and the choice of an adjacent element with shortest Euclidean distance benefits a low AR of that part. This is repeated until all vertices (elements) are covered. Afterwards, new seeds are calculated by each part independently by searching for the center vertices. This part could be executed in parallel. We define the center of a part to be the vertex for which the sum of Euclidean distances to all other vertices in that part (we call this the *distance-value*) is minimal. One may find the centers by calculating the distance values for all vertices, but this would have a time consumption of $O(\#\text{elements}^2)$. To avoid this, we calculate the distance-values for the seed as the initial center and all its adjacent vertices and move the center to the neighbor with smallest value; this process is repeated until a local minimum is found. The Bubble algorithm will terminate if in an iteration none of the seeds move any more. To avoid cyclic movements of seeds (which sometimes occur), we stop the algorithm if the AR does not improve for 10 consecutive iterations.

Bubble produces connected parts which are, in general, very compact and have a smooth shape. A major drawback of Bubble is the lack of a guarantee for balanced partitions. Although at the end the seeds are spread out evenly over the whole graph, the parts do not have to contain the same number of elements. To repair this, one may add a local partitioning method to balance the load, trying to further optimize either the cut or the AR.

We investigate the performance of different types of mesh partitioning strategies with respect to the number of global iterations of the DD–PCG, the cut size and the AR. We include the simple coordinate methods COO and COO_R and the greedy methods GBF and GCF as described in Section 2.4. Bubble is used without any load balancing, as well as with additional load balancing minimizing either the cut or the AR, where load balancing methods as described in Section 4 are used. In addition, we use the default settings of the Party, Jostle, K-Metis and P-Metis graph-partitioning libraries, as well as a Simulated Annealing code which is designed to optimize the AR. Fig. 9 displays the results of the partitioning of the meshes *turm* (with 531 elements) and *cooler* (with 749 elements, Fig. 11) into 8 parts. The test case for the numerical solver is a Poisson problem with Dirichlet-0 boundary conditions. The number of iterations for the tested methods differ significantly. It can be observed that the AR $L_{\max}/L_{\min}$ and R_o/R_i do not follow the line of the iteration numbers, whereas the values of R_o/A and $B^2/16A$ roughly do so. Unlike in the simple example

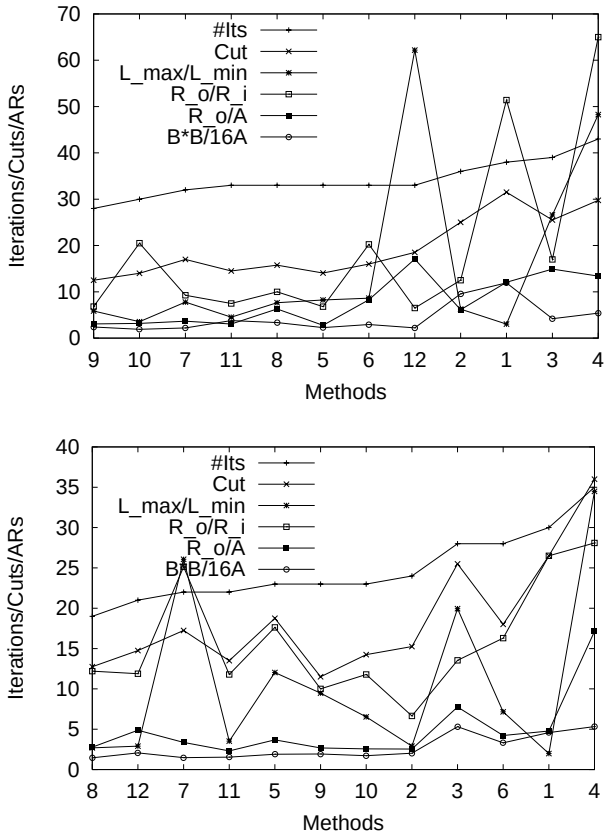


Fig. 9. Results of example *turm* (531 elements, top) and example *cooler* (749 elements, bottom). The methods are listed with increasing numbers of global PCG iterations. Compared Methods are 1:COO, 2:COO R, 3:GBF, 4:GCF, 5:BUB, 6:BUB+CUT, 7:BUB+AR, 8:Party, 9:Jostle, 10:K-Metis, 11:P-Metis, 12:SA.

of Section 2, the results additionally indicate that a low *cut* also leads to a fairly low number of iterations. A comparison of the tested partitioners shows that the simple coordinate and the greedy methods usually result in large iteration numbers. For the bubble variations, the balance by improving the AR $B^2/16A$ leads to lower iteration numbers than balancing by improving the cut. Furthermore, the partitions calculated by the partitioning libraries and the Simulated Annealing approach also lead to similarly low iterations numbers.

Table 1 shows additional results using the mesh crack with 20,141 elements as example. The mesh is partitioned into 16 parts and the Bubble method with its variations is compared with the default settings of the partitioning libraries Chaco (Multilevel approach), K-Metis, P-Metis and Jostle. We list the cut, the average AR$B^2/16A$ of all parts, and the number of iterations of the DD–PCG as measures The results reveal that the AR is a better measure for the number of iterations than

Table 1
Comparisons for mesh crack (20,141 elements) partitioned into 16 parts

	Chaco	Metis		Jostle	BUB		
		K–	P–			+CUT	+AR
Cut	856	800	760	768	920	799	813
AR	1.93	1.85	1.98	1.92	1.92	1.96	1.87
Iter.	46	46	53	44	50	50	44

the cut. To give a telling example, the partition calculated by Pmetis has the lowest cut, but needs the largest number of iterations. The partition calculated by bubble has a high cut size, but if it is load-balanced by further optimizing the AR, it finally succeeds in achieving the overall goal: minimizing the number of global iterations of the DD–PCG algorithm. If the bubble partition is load-balanced minimizing the cut instead, the cut becomes smaller. However, the AR and the number of iterations are not as low as for minimizing the AR. Combined with shape optimizing load balancing methods which are described in the following, bubble serves as reasonable initial partitioner. The times to calculate the partitions was fairly small for all methods due to the medium sized examples. In an adaptive environment, the partitioning only has to be performed on the initial mesh which is usually small. For larger meshes, bubble could be used as partitioning method for the coarse graph in the multilevel paradigm after coarsening the large initial mesh into a much smaller one of similar structure. The Chaco, Jostle and Metis tools have already applied this multilevel strategy.

4. Shape optimized dynamic load balancing

We now consider the problem of dynamic load balancing in the case of the mesh being refined adaptively. Our solution to this problem is a two-step approach: first, a balancing flow is calculated on the edges of the partition quotient graph. Afterwards, elements at partition boundaries are chosen and migrated in order to fulfill the demands of the flow. We first discuss different functions for choosing elements which aim at considering the shape of the partition. This second step is described first because some of the element rating functions introduced here are used in Section 4.2 to guide the balancing flow calculation. Results in Section 4.3 compare different functions for rating elements with regard to partition shape. The benefits of the different steps are investigated and the final algorithm is compared to Jostle.

4.1. Element migration

The basic idea of the element migration phase is to rate elements at partition boundaries according to certain cost functions and to greedily choose elements that have to be moved to neighboring processors based on this rating, until the necessary load has been moved. The migration phase starts with a valid balancing flow

determined by the phase of flow calculation which will be de scribed in Section 4.2. For now, assume that the flow is represented as values x_{ij} on edges (i,j) of the quotient graph (i.e., x_{ij} elements have to be moved from partition i to partition j. The migration phase aims at balancing the load such that the AR of subdomains is maintained, improved, or deteriorates as little as possible.

As discussed in Section 2.3 we use the perimeter-to-area ratio as a definition of AR. For a subdomain P_i with area A_i and perimeter B_i (its boundary length) its AR_i is given by

$$\mathrm{AR}_i = \frac{B_i^2}{16A_i}. \tag{3}$$

The simplest way to rate elements at partition boundaries is to count the change in cut size (of the element graph) when moved to other partitions. For an element e, let $\mathrm{ed}(e,i)$ be its number of edges adjacent to elements in P_i. The change in cut when e is moved from P_i to P_j is given by

$$\Gamma_{\mathrm{cut}}(e,i,j) = \mathrm{ed}(e,i) - \mathrm{ed}(e,j) \tag{4}$$

For many FEM meshes constructed of triangles, the resulting element graph has a maximal degree of three and the change in Γ_{cut} is at most 1. It is known [6] that local graph partitioning algorithms perform badly on degree-3 graphs, so it is no surprise that this element rating function does not produce satisfying results. We will see comparisons later on. Nevertheless, it is used in several applications and, if combined with other sophisticated methods such as multilevel schemes, it can still work very well [44].

We can also take the change in AR to rate elements at partition boundaries. First, define $\mathrm{AR} = \Sigma_i \mathrm{AR}_i$ as the AR of a partitioned mesh. Second, let a_e be the area of element e, b_e the length of its border, and $b_e(i)$ the length of its border to elements in subdomain P_i. Then

$$\Gamma_{\mathrm{ar}}(e,i,j) = \frac{B_i^2}{16A_i} + \frac{B_j^2}{16A_j} - \frac{(B_i - b_e + 2b_e(i))^2}{16(A_i - a_e)} - \frac{(B_j + b_e - b_e(j))^2}{16(A_j + a_e)} \tag{5}$$

will be the change in AR, if e is moved from P_i to P_j. We can simplify the calculations of (5) by just taking the change in total border length:

$$\Gamma_B(e,i,j) = b_e(i) - b_e(j). \tag{6}$$

The drawback of the functions described so far is their "local" nature. They rate elements only based on local changes in a cost function, but they have no idea of which element will have to be preferred if two of them have the same value. And the case that two elements have the same value does not mean they are equally attractive for the task of optimizing the overall AR. It just means that they have the same number of external edges (4), the same size and border length (5) or the same border length (6). A "global" view, whether it is favorable for a subdomain to own certain elements or not, is lacking.

A first step towards a kind of "global" view is to consider the distances of elements from the center of their subdomain. If all boundary-elements have the same

distance to this center, the domain will be a circle and will have a good AR. Elements lying far away from the center might be good candidates to move. Let C_i be the center of gravity of subdomain i. If we define the center c_e of element e to be the average of its node coordinates, C_i will be defined as

$$C_i = \frac{1}{A_i} \sum_{e \in P_i} c_e \cdot a_e. \tag{7}$$

Let $\mathrm{dist}(e,i)$ be the (Euclidean) distance of element e to the center of subdomain P_i. The function

$$\Gamma_{\mathrm{rd}}(e,i,j) = \mathrm{dist}(e,i) - \mathrm{dist}(e,j) \tag{8}$$

is called the *relative distance* of element e. The idea behind this function is to prefer elements which are far away from the center of their own subdomain and, additionally, near to the center of the target partition. Such function of element rating can be found in some implementations [7,11,36]. The problem is that they do not account for the real physical size of a domain. A large partition might easily move elements to a small one, even if this does not improve the AR of any of them. We can define D_i as the average distance of the centers of all elements of subdomain P_i from its center C_i.D_i can be used to normalize the distance function (8) [11].

$$\Gamma_{\mathrm{srd}}(e,i,j) = \frac{\mathrm{dist}(e,i)}{D_i} - \frac{\mathrm{dist}(e,j)}{D_j} \tag{9}$$

is called the *scaled relative distance function*. Fig. 10 (left) reveals the advantage of Γ_{srd} in contrast to Γ_{rd}. Without any scaling, P_3 prefers to move elements to P_2, whereas a move towards P_1 improves the AR of P_1 and P_3.

If we consider the shape of a common border between two subdomains, the ideal would be to have a straight line perpendicular to the interconnection of the centers of both. If the border is circular in any direction, one of the domains will become concave which is not desirable. If elements are moved between subdomains which are very different in size, the influence of the smaller one on Γ_{srd} will become larger with increasing distance from the line connecting both centers. Thus, elements far away from this line are more likely to be moved than those lying on or near to this line.

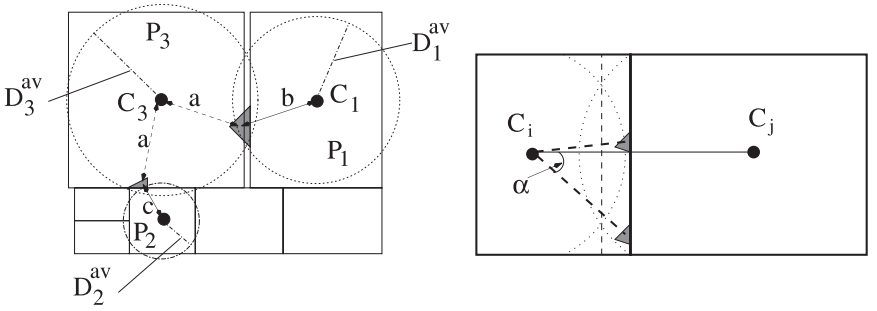


Fig. 10. Scaled distances (left). Angle (right).

This effect can be observed in Fig. 11 (left) where the domain "Cooler" is refined in 10 steps from initially 749 elements to finally 7023. The refinement mainly takes place at the upper part of the domain and the subdomains lying in the corresponding region give up elements according to Γ_{srd}. To avoid this effect, we superimpose Γ_{srd} by another function $\Gamma_{\alpha}(e,i,j) = \cos\alpha$, where α is the angle between the lines $C_i \leftrightarrow C_j$ and $C_i \leftrightarrow c_e$. Fig. 10 (right) presents the construction. The value of $\cos\alpha$ will be large, if angle α is small, i.e., the superposition slightly increases the values of those elements which lie in the direction of the target partition. Fig. 11 (right) shows the effect on the same example of Fig. 11 (left). As intended, the borders between subdomains are much straighter and the target domains are less concave.

A first comparison of the different rating functions is to be found in Table 2. It shows the maximum and average AR as well as the Cuts, if the domain "Square" is refined in 10 steps from initially 115 elements to finally 10,410. The values for AR given here and in the rest of the paper are scaled to 1.0 being an "optimal" shape. Average values of >1.6 can be considered as unacceptable. It can be observed that Γ_{cut} is not only bad in optimizing AR but also in finding partitions with a good cut. The explanation has already been given: As the graph has a maximal degree of 3, local methods are likely to fail. Interestingly, Γ_{ar} and Γ_B do not perform well either. The purely local rating of this functions does not allow a good AR optimization. Γ_{rd} and Γ_{srd} are quite similar to each other because the difference in size of the subdomains is not very large in this example. The effects of Γ_{α} are not directly visible in terms of maximum or average AR. This kind of "cosmetic" operation becomes more obvious in Fig. 11 (left) and (right).

Table 3 shows similar results for combinations of the different Γ's. It can be observed that additional improvements will be possible, if such combinations are used. Unfortunately, the possibilities of different combinations enlarge the space of

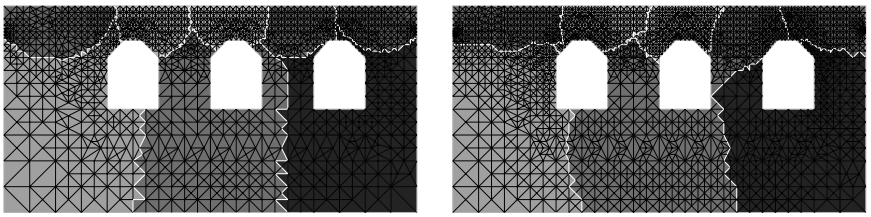


Fig. 11. Migration according to Γ_{srd} (left) and to $\Gamma_{\mathrm{srd}} + \Gamma_{\alpha}$ (right).

Table 2
AR and cut obtained with different rating functions

	AR-max	AR-avrg	Cut-max	Cut-avrg
Γ_{cut}	2.03	1.66	190	128
Γ_{ar}	1.85	1.55	191	130
Γ_B	1.88	1.56	187	129
Γ_{rd}	1.53	1.33	142	92
Γ_{srd}	1.47	1.29	144	90

Table 3
Combined cost functions

	AR-max	AR-avrg	Cut-max	Cut-avrg
$\Gamma_{\mathrm{rd}} + 0.2\Gamma_B$	1.365	1.222	119	80
$\Gamma_{\mathrm{srd}} + 0.2\Gamma_B$	1.342	1.201	131	84
$\Gamma_{\mathrm{rd}} + 0.4\Gamma_\alpha$	1.572	1.359	142	93
$\Gamma_{\mathrm{srd}} + 0.6\Gamma_\alpha$	1.496	1.322	140	90
$\Gamma_{\mathrm{rd}} + 0.2\Gamma_B + 0.2\Gamma_\alpha$	1.405	1.246	125	83
$\Gamma_{\mathrm{srd}} + 0.2\Gamma_B + 0.4\Gamma_\alpha$	1.392	1.249	134	85

possible parameter settings. However, generally speaking, Γ_{srd} combined with Γ_α is a good choice for an element rating function.

4.2. Balancing flow calculation

In Section 2.1 we have defined the quotient graph and balancing flows on this graph. We will now briefly review possible algorithms that are suitable to calculate balancing flows. Furthermore we will discuss modifications to support the element migration phase.

4.2.1. Possible solutions

Let $G = (V, E, w)$ be the quotient graph of a load balancing problem (Fig. 1). G has $n = |V|$ nodes, $m = |E|$ edges and there are load values w_i attached to each node $i \in V$ expressing the number of elements of the partition, i is representing. The edges in G express the adjacencies between subdomains (i.e., if two subdomains are not adjacent, then the corresponding edge does not exist either). The task of the balancing problem is to find a flow x_{ij} of load on edges $(i, j) \in E$ of the graph such that after having moved the desired numbers of elements, the load is globally balanced, i.e. $w_i = \bar{w}\ \forall i \in V$. Since the flow is directed, we need to define (implicit) directions for edges. We may assume that edges are directed from nodes with smaller numbers to nodes with larger numbers. A flow of $x_{ij} < 0$ on edge $(i, j) \in E$ then means that $-x_{ij}$ load has to be moved from node j to i.

Let $A \in \{-1, 0, 1\}^{n \times m}$ be the *node-edge incidence matrix* of G. The edge directions are expressed by the signs (from $+1$ to -1). The matrix $L = AA^{\mathrm{T}}$ is called the *Laplacian matrix* of G. If we define $x \in \mathbb{R}^m$ to be the vector of flow values, and if we like to minimize the total amount of flow, we will search for an x satisfying

$$\text{Minimize} \sum_{(i,j)\in E} |x_{ij}| \quad \text{subject to } Ax = w - \bar{w}. \tag{10}$$

Eq. (10) is known to be the *min-cost flow problem* [1]. The question of what are good cost criteria for such a flow is not easy to answer. It is fairly obvious that any solution to (10) shifts the flow only via shortest paths in the network. Thus, it often uses only a small fraction of the available edges, a quality that might not be desirable. This is especially the case when afterwards the shape of subdomains has to be considered.

A compromise between the conflicting goals of not shifting too much load but using all edges more or less equally might be the Euclidean norm of the flow defined as $\|x\|_2^2 = \sum x_{ij}^2$. This measure turns (10) into a non-linear, but convex optimization problem which is still solvable in polynomial time. For this special type of problem, solutions can be found easily. Hu and Blake propose the use of Lagrange multipliers for a solution of $Ax = w - \bar{w}$ [24]. The method requires the solution of $L\lambda = w - \bar{w}$ and the flow is afterwards given by $x_{ij} = \lambda_i - \lambda_j$. $L\lambda = w - \bar{w}$ is easy to solve in parallel, because L directly relates to the quotient graph G (in fact, L is topologically equivalent to G).

Other local iterative methods determine x directly by performing

$$x^t = \xi A^{\mathrm{T}} w^t \quad \text{and} \quad w^{t+1} = w^t - Ax^t. \tag{11}$$

Iteration (11) is similar to $w^{t+1} = Mw^t$ with iteration matrix $M = (I - \xi L)$ and parameter $\xi \leqslant 1/\max\_\deg(G)$. It is known as the *diffusion method* [4,8]. Some schemes try to speed up the convergence of this local iterative method by using non-homogeneous iteration matrices M, optimal values of ξ [13], and over-relaxations such as $w^{t+1} = \beta M w^t + (1 - \beta) w^{t-1}$ [19]. It can be shown that all of these local iterative methods determine flows x with minimal $\|x\|_2$ [10].

4.2.2. Guidance of flow: Local iterative methods

We can use any of the above mentioned methods to calculate the balancing flow in PadFEM. The drawback here is the fact that they do not take into account the "quality" of elements at partition boundaries, the length of the boundaries, and their position relative to the centers of their partitions. The edges in G just denote neighborhoods between subdomains, regardless of the number of elements along this border, and the shapes of the corresponding partitions.

As an initial idea, we can put weights on the edges of G expressing the cost (or benefit) of moving elements via certain edges to consider the task of shape optimization within the balancing flow calculation. We can use the element rating functions defined in Section 4.1 to perform this task. For a subdomain P_i, let Bo_i be its set of border elements, and $\mathrm{Bo}_i(j)$ the elements at the border adjacent to subdomain P_j. We define the weight ω_{ij} on edge $(i,j) \in E$ by

$$\omega_{ij} = \frac{1}{|\mathrm{Bo}_i(j)|} \sum_{\substack{e \in \mathrm{Bo}_i(j) \\ e \notin \mathrm{Bo}_i(k) \forall k \neq j}} \Gamma(e,i,j), \tag{12}$$

where Γ can be any of the cost functions and even combinations of Section 4.1. In the iteration process of (11), the parameter ξ defines the amount of load to shift between neighboring processors per step, depending on their load difference. If we define ξ to be

$$\xi ij = \frac{\omega_{ij}}{\sum_k \omega_{ik}}, \tag{13}$$

edges with large weights are preferred during the balancing (recall that the flow calculation is only a pre-step to the real load movement; so the borders $\mathrm{Bo}_i(j)$ do not change in this step).

Table 4
Guided diffusion: mean values of AR using different edge-weighting functions

	Square	Cooler	Smiley	Tower
Γ_u	1.351	1.289	1.387	1.341
Γ_{srd}	1.364	1.302	1.393	1.344
Γ_α	1.353	1.294	1.387	1.340
$\Gamma_{\mathrm{rd}} + \Gamma_\alpha$	1.361	1.296	1.390	1.765

Table 4 shows results of the diffusion method if ξ is chosen according to (13) and the ω_{ij}'s use different element rating functions Γ. The values given in Table 4 are mean values of experimental results with the given edge weighting function and several different combinations of the Γ's as migration rating functions. Γ_u denotes the case of no edge weighting. We can see that there are slight improvements in AR for certain examples, but that this approach is generally not helpful. A reason might be its very local structure. If there is a large load difference between two processors, a large number of elements will be shifted over their common border regardless of how short it is and regardless of whether the receiving processor is itself able to shift (parts of) the load to any of its neighbors properly. In the next section we will describe a method which considers weights along paths of load movement. We will see that such an approach can support the shape optimization phase in a better way.

4.2.3. Average weighted flow

The average weighted flow (AWF) algorithm proceeds in two steps. It first determines (source/sink/amount) triples giving pairs of processors and the amount of load that has to be moved between them. Afterwards, these flows are routed via paths in the graphs.

The first step starts from each source node and searches for sinks nearby which are reachable via "good" paths in the graphs. For a path $p = v_1, \ldots, v_k$, we define the *path quality*

$$q(p) = \frac{1}{k} \sum_{i=1}^{k-1} \omega_{v_i v_{i+1}} \tag{14}$$

as the average quality of edges on the path. Since we are searching for paths with high quality, the task is to maximize $q(p)$. Unfortunately, this makes the problem intractable. If $q(p)$ were just the sum of the ω's, we would search for longest paths without loops, an *NP* complete problem [18]. Additionally, the averaging in (14) causes the loss of transitivity. A "best" path from i to j via k must no longer include the best paths from i to k and from k to j.

We use a variance of breadth first search (BFS) as a heuristic to find approximately "best" paths. The algorithm starts from a source and searches – in normal BFS style – for paths to sinks. A sink node has a "distance" from the source according to the quality of the best path via which it has been reached during the BFS. If a node is found again (due to circle closing edges), its "distance" is updated. The algorithm visits each edge at most once and, thus, has a running time of $O(m)$. For

Table 5
AWF in comparison to diffusion

	Square		Cooler		Smiley		Tower	
	Diff	AWF	Diff	AWF	Diff	AWF	Diff	AWF
Γ_u	1.351	1.340	**1.289**	**1.289**	1.387	1.411	1.341	1.320
Γ_{srd}	1.364	**1.320**	1.302	1.289	1.393	**1.340**	1.344	1.311
Γ_α	1.353	1.344	1.294	1.296	1.387	1.368	1.340	1.336
$\Gamma_{\text{rd}} + \Gamma_\alpha$	1.361	1.328	1.296	**1.289**	1.390	1.374	1.765	**1.300**

each sink node, all the paths from the source found during the search are stored together with their quality. The load $w_i - \bar{w}$ of a source i is distributed (logically) to its "best" sinks, each of them filling up to $\bar{w}$ in order to build the (source/sink/amount) triples. The time for this phase is dominated by the search. As there are at most n sources, the total running time is $O(n \cdot m)$.

The second phase routes the flow demands between source/sink pairs via paths in the network. The task here is to distribute the flow evenly among (a subset of) possible paths from the source to the sink. The first phase has already calculated possible paths. If a flow of x_{st} has to be shifted from a source s to a sink t and if there are two possible paths p_1 and p_2 between them, the fraction of $q(p_1)/(q(p_1) + q_(p_2))$ will be shifted via p_1. The rest will be shifted via p_2.

Table 5 shows a comparison of AWF and Diffusion. Again, several different functions for rating element are chosen to guide the migration. The values in Table 5 are mean values of all results for a given problem and a given flow algorithm and edge weighting function. It can be observed that an improvement in the achievable AR of around 5% is possible by weighting the edges properly (Γ_u is again the non-weighted case). AWF behaves generally better than Diffusion. Sometime the improvements are large ($\approx$25%), in most cases they are around 5%. Γ_{srd} turns out to be a good edge weighting function, and also the use of Γ_α sometimes improves the situation.

4.3. Results

Figs. 12 and 13 present the development of AR, cut and number of iterations of the DD-PCG solver over a number of refinement levels of the Domain "Square" (Fig. 1) refined in a number of steps from 736 to 15421 elements. Experiments using regular meshes and regular domains show an increase in the number of iterations by one when the number of elements are doubled. For irregular meshes such as those used here, one can not hope for such good convergence behavior. Fig. 12 displays the behavior if Γ_{cut} is used as element rating function, Fig. 13 if $\Gamma_{\text{srd}} + \Gamma_\alpha$ is used. It can be observed that the number of iterations will be very unstable and generally larger if cut is the optimization goal. If the shape is optimized, the solver behaves much better.

Generally, AR expresses the development of the number of iterations much better than Cut. We conclude from the experiments that an optimization according to AR is reasonable.

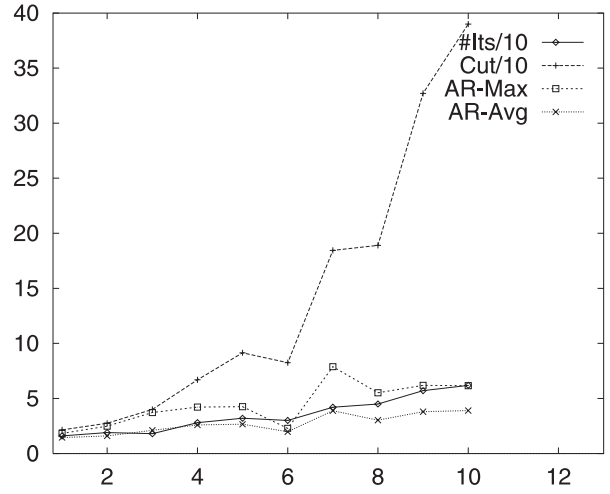


Fig. 12. Development of AR, cut and # Iterations for domain "square" and increasing level of refinment with Γ_{cut} as element rating function.

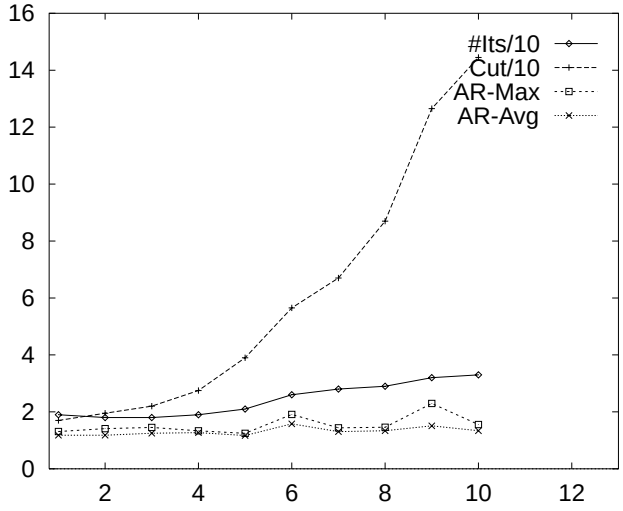


Fig. 13. Development of AR, cut and # Iterations for domain "square" and increasing level of refinment with $\Gamma_{\text{srd}} + \Gamma_{\alpha}$ as element rating function.

Fig. 14 (left) shows the results of the AR-optimization using Γ_{srd} as rating function and AWF with $\Gamma_{\text{srd}} + 0.2\Gamma_{\alpha} + 1.2\Gamma_{B}$ as balancing flow calculation. In the tests, the domain "Smiley" is refined in 6 steps from 2500 elements to 5000 elements using Rivara's refinement algorithm [34]. For comparisons, the right picture shows the same domain, refined in the same way, but now with Γ_{cut} as element

Fig. 14. Optimization according to AR (left) and cut (right).

rating and standard Diffusion for balancing flow calculation. The differences are directly visible.

We compare the results of the AR-optimization in PadFEM with those of Jostle. The element migration decision in Jostle uses Γ_{cut} as the rating function. Together with a special border optimization based on KL and with sophisticated tie-breaking methods, Jostle is able to generate and maintain well shaped partitions, even if the AR is not directly considered.

Figs. 15 and 16 show the development of AR over typical runs of (solve/ refine/ balance ...) for our four test examples. Fig. 15 shows the results if Jostle (without multi-level coarsening-the Jostle-D configuration described in [44]) is used as load balancer. Fig. 16 gives the results if the AR-optimization of PadFEM is used (with

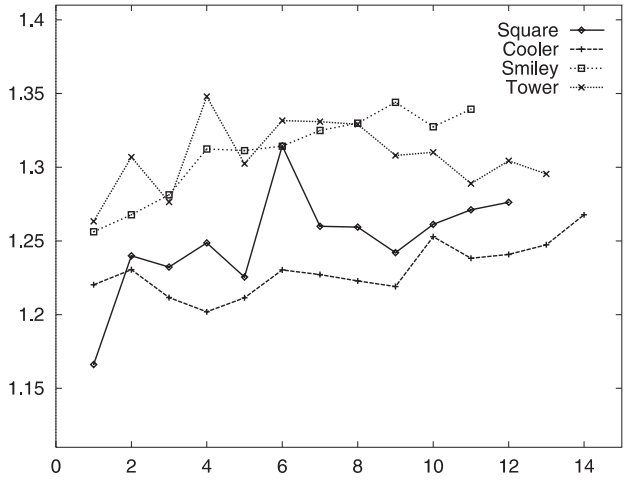


Fig. 15. The development of the AR for increasing levels of refinement with load balancer of Jostle.

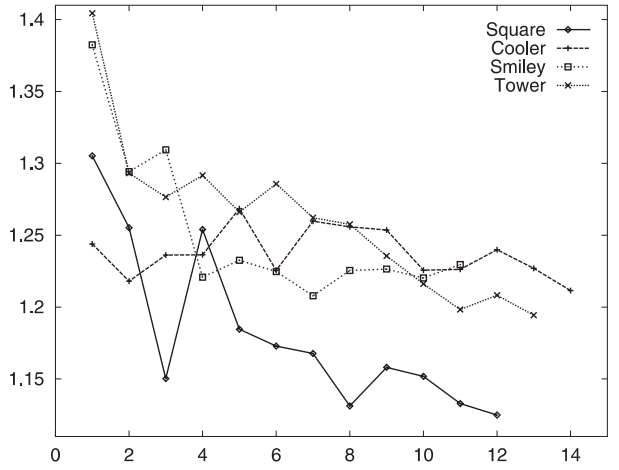


Fig. 16. The development of the AR for increasing levels of refinement with AR-optimization of PadFEM.

best possible parameter combinations). It can be observed that the AR is decreasing with increasing numbers of elements if the PadFEM-balancer is used. The opposite is true if Jostle determines the elements to be moved. This shows that our strategy can operate as high-quality load balancer in adaptive environments maintaining well shaped subdomains over a long period of time. Standard load balancers like Jostle (and with Metis, it would be just the same) cannot directly be used over a longer period without complete repartitionings from time to time. At least this is the case, if the partition AR is a measure of importance.

To show that the results given in Fig. 15 and 16 are not special cases, Table 6 presents values of AR averaged over several steps of refinements and different parameter settings. The unweighted results (columns "UW") are just normal average values, the others show ARs weighted with the number of elements of the individual meshes. The idea behind this weighting is to increase the importance of finer meshes, where the solvers take longer times and where it is much more important to achieve good AR. The results show that PadFEM can improve the AR of mesh partitions by around 10% over a state-of-the-art general purpose load balancing tool. This sort of improvement is confirmed in [42].

Table 6
Comparison, Jostle ↔ PadFEM

	Jostle		PadFEM	
	UW	W	UW	W
Square	1.250	1.267	**1.182**	**1.140**
Cooler	**1.230**	1.245	1.238	**1.229**
Smiley	1.310	1.326	**1.252**	**1.234**
Tower	1.307	1.303	**1.261**	**1.219**

5. Extensions to 3D and parallelism

All methods discussed have been applied to examples in 2D. The graph partitioning algorithms either work on graph measures or on the Euclidean distance, which can easily be extended to 3D. The balancing flow calculation works only on graph measures, so the dimension of the mesh does not play a role. Differencing between the dimensions is a proper definition of AR. In 3D, borders change to surfaces, areas to volumes and circles to spheres. The main consequence is an increase in calculation time for the different shape cost measures, but in principle, all definitions can be extended easily.

We have not shown any quantitative results concerning run-times or parallel efficiencies. The main reason is the difficulty of getting complex applications such as parallel adaptive numerical codes using DD-preconditioned CG solvers to run with competitive performance. The balancing flow calculation is parallel by definition and can be calculated together with the CG solver iterations (for the AWF method, parallelization turns out to be a lot more complex). Concerning the element migration it is necessary to find independent sets of processor pairings to avoid conflicts during the data movement.

6. Conclusions

We have presented a two-step load balancing algorithm for adaptive mesh calculations on distributed memory machines. Our new center-oriented method called Bubble naturally produces very low AR and can successfully be used as a starting point for a balancing strategy. The balancing algorithm is distributed and takes existing partitions into account. The algorithm first determines a balancing flow and then moves elements. The element migration is particularly designed to optimize the AR of subdomains. Experiments which make use of a preconditioned Conjugate Gradient solver based on Domain Decomposition and comparisons to existing balancing tools show the benefit of our approach.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows – Theory, Algorithms, and Applications, Prentice-Hall, Eaglewood, Cliffs, NJ, 1993.

[2] S.T. Barnard, H.D. Simon, Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, Concurrency: Practice and Experience 6 (2) (1994) 101–117.

[3] S. Blazy, W. Borchers, U. Dralle, Parallelization methods for a characteristic's pressure correction scheme, in: E.H. Hirschel (Ed.), Flow Simulation with High-Performance Computers II Notes on Numerical Fluid Mechanics, 1995.

[4] J.E. Boillat, Load balancing and poisson equation in a graph, Concurrency: Practice and Experience 2 (4) (1990) 289–313.

[5] J.H. Bramble, J.E. Pasciac, A.H. Schatz, The construction of preconditioners for elliptic problems by substructuring i. and ii., Math. Comput. 47+49, (1986+1987).

[6] T.N. Bui, S. Chaudhuri, F.T. Leighton, M. Sisper, Graph bisection algorithms with good average case behaviour, Combinatorica 7 (2) (1987) 171–191.
[7] N. Chrisochoides, C.E. Houstis, E.N. Houstis, S.K. Kortesis, J.R. Rice, Automatic load balanced partitioning strategies for PDE computations, in: Proceedings of the ACM International Conference on Supercomputing, 1989, pp. 99–107.
[8] G. Cybenko, Load balancing for distributed memory multiprocessors, J. Par. Distr. Comput. 7 (1989) 279–301.
[9] R. Diekmann, U. Dralle, F. Neugebauer, T. Römke. Padfem: A portable parallel FEM-tool, in: HPCN, LNCS 1067, 1996, pp. 580–585.
[10] R. Diekmann, A. Frommer, B. Monien, Efficient schemes for nearest neighbor load balancing, in: G. Bilardi et al., (Ed.), 6th European Symposium an Algorithms (ESA'98), LNCS 1461, 1998, pp. 429–440.
[11] R. Diekmann, D. Meyer, B. Monien, Parallel decomposition of unstructured FEM-meshes, Concurrency: Practice and Experience 10 (1) (1998) 53–72.
[12] R. Diekmann, B. Monien, R. Preis, Using helpful sets to improve graph bisections, in: Hsu et al. (Ed.), Interconnection Networks and Mapping and Scheduling Parallel Computations, DIMACS Disc. Math. Theory Com. Sci. 21 (1995) 57–73AMS.
[13] R. Diekmann, S. Muthukrishnan, M.V. Nayakkankuppam, Engineering diffusive load balancing algorithms using experiments, In: G. Bilardi et al. (Ed.), IRREGULAR'97, LNCS 1253, 1997, pp. 111–122.
[14] C. Farhat, A simple and efficient automatic FEM domain decomposer, Computers and Structures 28 (5) (1988) 579–602.
[15] C. Farhat, S. Lanteri, H.D. Simon, Top/domdec – a software tool for mesh partitioning and parallel processing, J. Comput. Syst. Engrg. 6 (1) (1995) 13–26.
[16] C. Farhat, N. Maman, G. Brown, Mesh partitioning for implicit computations via iterative domain decomposition: Impact and optimization of the subdomain aspect ratio, Int. J. Numer. Methods Engrg. 38 (1995) 989–1000.
[17] C.M. Fiduccia, R.M. Mattheyses, A linear-time heuristic for improving network partitions, in: Proceedings of the 19th IEEE Design Automation Conference, 1982, pp. 175–181.
[18] M.R. Garey, D.S. Johnson, Computers and Intractability, W.H. Freeman, San Francisco, 1979.
[19] B. Ghosh, S. Muthukrishnan, M.H. Schultz, First and second order diffusive methods for rapid, coarse, distributed load balancing, In: Proc. ACM-SPAA'96, 1996, pp. 72–81.
[20] T. Goehring, Y. Saad. Heuristic algorithms for automatic graph partitioning, Technical Report UMSI 94-29, University of Minnesota Supercomputer Institute, 1994.
[21] B. Hendrickson, R. Leland, The Chaco user's guide: Version 2.0. Technical Report SAND94-2692, SNL, Albuquerque, NM, Oct 1994.
[22] B. Hendrickson, R. Leland, An improved spectral graph partitioning algorithm for mapping parallel computations, SIAM J. Sci. Comput. 16 (2) (1995) 452–469.
[23] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proc. Supercomputing '95. ACM, Dec 1995.
[24] Y.F. Hu, R.J. Blake, D.R. Emerson, An optimal migration algorithm for dynamic load balancing, Concurrency: Practice and Experience 10 (6) (1998) 467–483.
[25] D.S. Johnson, C.R. Aragon, L.A. McGeoch, C. Schevon, Optimization by simulated annealing: An experimental evaluation; part 1, graph partitioning, Operations Research 37 (6) (1989) 865–893.
[26] M.T. Jones, P.E. Plassmann, Parallel algorithms for adaptive mesh refinement, SIAM J. Sci. Comput. 18 (1997) 686–708.
[27] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, Technical Report 95-035, CS Dept., University of Minnesota, 1995 (to appear in SIAM J. Sci. Comput.).
[28] B.W. Kernighan, S. Lin, An effective heuristic procedure for partitioning graphs, The Bell Syst. Tech. J. Feb 1970, 291–308.
[29] Y. Linde, A. Buzo, R.M. Gray, An algorithm for vector quantizer design, IEEE Trans. Communications COM-28 (1980) 84–95.

[30] L. Oliker, R. Biswas, Plum: Parallel load balancing for adaptive unstructured meshes, J. Par. Dist. Comput. 52 (2) (1998) 150–177.
[31] F. Pellegrini, J. Roman, Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: HPCN, Apr 1996, pp. 493–498.
[32] A. Pothen, H.D. Simon, K.P. Liu, Partitioning sparse matrices with eigenvectors of graphs, SIAM J. Matrix Anal. Appl. 11 (3) (1990) 430–452.
[33] R. Preis and R. Diekmann, Party – a software library for graph partitioning, in: B.H.V. Topping (Ed.), Advances in Computational Mechanics with Parallel and Distributed Processing, 1997, pp. 63–71.
[34] M.-C. Rivara, Mesh refinement processes based on the generalized bisection of simplices, SIAM J. Numer. Anal. 21 (3) (1984) 604–613.
[35] Youcef Saad, Iterative Methods for Sparse Linear Systems. PWS Publ. Co., 1996.
[36] Frank Schlimbach, Load-Balancing Heuristics optimising Subdomain Aspect Ratios for Adaptive Finite Element Simulations, Ph.D. Thesis, School of Computing and Math. Sciences, The University of Greenwich, London, 1999.
[37] K. Schloegel, G. Karypis, V. Kumar, Multilevel diffusion schemes for repartitioning of adaptive meshes, J. Par. Dist. Comput. 47(2) (1997) 109–124.
[38] H.D. Simon, Partitioning of unstructured problems for parallel processing, Comput. Syst. Engrg. 2 (1991) 135–148.
[39] D. Vanderstraeten, C. Farhat, P.S. Chen, R. Keunings, O. Zone, A retrofit based methodology for the fast generation and optimization of large-scale mesh partitions: Beyond the minimum interface size criterion, Comput. Meth. Appl. Mech. Engrg. 133 (1996) 25–45.
[40] D. Vanderstraeten, R. Keunings, C. Farhat, Beyond conventional mesh partitioning algorithms and the minimum edge cut criterion: Impact on realistic applications, in: Sixth SIAM Conference on Parallel Processing for Scientific Computing, 1995, pp. 611–614.
[41] R. Verfürth, A Review of a posteriori Error Estimation and Adaptive Mesh-Refinement, Wiley, Chichester, 1996.
[42] C. Walshaw, M. Cross, R. Diekmann, F. Schlimbach, Multilevel mesh partitioning for optimising domain shape. Tech. Rep. 98/IM/38, Univ. Greenwich, London SE18 6PF, UK, July 1998 (to appear in Int. J. High Performance Comput. Appl.).
[43] C. Walshaw, M. Cross, M.G. Everett, A localised algorithm for optimising unstructured mesh partitions, Int. J. Supercomput. Appl. 9 (4) (1995) 280–295.
[44] C. Walshaw, M. Cross, M.G. Everet, Parallel dynamic graph partitioning for adaptive unstructured meshes, J. Par. Dist. Comput. 47 (2) (1997) 102–108.
[45] O.C. Zienkiewicz, The Finite Element Method, McGraw-Hill, New York, 1989.