A Parallel Shape Optimizing Load Balancer*

Henning Meyerhenke and Stefan Schamberger

Universität Paderborn,
Fakultät für Elektrotechnik, Informatik und Mathematik
Fürstenallee 11, D-33102 Paderborn
{henningm, schaum}@uni-paderborn.de

Abstract. Load balancing is an important issue in parallel numerical simulations. However, state-of-the-art libraries addressing this problem show several deficiencies: they are hard to parallelize, focus on small edge-cuts rather than few boundary vertices, and often produce disconnected partitions.

We present a distributed implementation of a load balancing heuristic for parallel adaptive FEM simulations. It is based on a disturbed diffusion scheme embedded in a learning framework. This approach incorporates a high degree of parallelism that can be exploited and it computes well-shaped partitions as shown in previous publications. Our focus lies on improving the condition of the involved matrix and solving the resulting linear systems with local accuracy. This helps to omit unnecessary computations as well as allows to replace the domain decomposition by an alternative data distribution scheme reducing the communication overhead, as shown by experiments with our new MPI based implementation.

Keywords: Load balancing, graph partitioning, parallel adaptive FEM computations.

1 Introduction

Finite Element Methods (FEM) play a very important role in engineering for analyzing a variety of physical processes that can be expressed via Partial Differential Equations (PDE). The domain on which the PDEs have to be solved is discretized into a mesh, and the PDEs are transformed into a set of equations defined on the mesh's elements (see e. g. [5]). Due to the sparseness of the discretization matrices these equations are typically solved by iterative methods such as Conjugate Gradient (CG) or multigrid.

Since an accurate approximation of the original problem requires a very large number of elements, this method has become a classical application for parallel computers. The parallelization of numerical simulation algorithms usually

* This work is supported by German Science Foundation (DFG) Research Training Group GK-693 of the Paderborn Institute for Scientific Computation (PaSCo) and DFG Collaborative Research Centre SFB-376.

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 232–242, 2006.

follows the Single-Program Multiple-Data paradigm: Each of the P processors executes the same code on a different part of the data. Thus, the mesh has to be split into sub-domains, each being assigned to one processor. To minimize the overall computation time, all processors should roughly contain the same number of elements. Furthermore, since iterative solution algorithms perform mainly local operations, the parallel algorithm mostly requires communication at the partition boundaries. Hence, these should be as small as possible due to the very high communication costs involved.

Depending on the application, some areas of the simulation space require higher resolutions and therefore more elements. Since in many cases the location of these areas varies over time, the mesh is refined and coarsened during the computation. Yet, this can cause imbalance between the processor loads and therefore delay the simulation. To avoid this, the element distribution needs to be rebalanced during runtime. For this, the application is interrupted and the repartitioning problem is solved. Although this interruption should be as short as possible, it is also important to find a new balanced partitioning with small boundaries that does not cause too many elements to change their processor. Migrating elements can be extremely costly since large amounts of data have to be sent over communication links and stored in complex data structures.

In previous work [19,15] we have shown that (re-)partitioning heuristics focusing on the shape of partitions are able to find solutions with a small number of boundary vertices while also causing little migration. There, we have compared our method to the state-of-the-art libraries Metis [11] and Jostle [21] regarding solution quality and runtime. It turns out that while the solution quality of the shape-optimizing approach is usually the best, its main drawback is its long runtime and high memory consumption. Therefore, in this paper we present a new parallel implementation based on the message-passing interface MPI that incorporates several improvements addressing these problems.

The remaining part of the paper is organized as follows. In the next Section we recapture related work and explain the shape optimizing bubble framework. Section 3 describes the diffusion scheme applied within this framework as its growth mechanism and an enhancement to the condition of the involved matrix. Our parallel MPI based implementation is presented in Section 4. The new concept of solving the linear systems with local accuracy reduces the computation time as well as the memory requirements. Additionally, it facilitates a new data distribution scheme decreasing communication. Subsequently, we present some of our experiments in Section 5 before we give a short conclusion.

2 Related Work

2.1 Graph Partitioning and Load Balancing Heuristics

Balancing an FEM mesh can be expressed as a graph (re-)partitioning problem. The mesh is transformed into a graph whose vertices represent the computational

work and the edges their interdependencies. Due to its complexity existing libraries for this problem are based on heuristics. State-of-the-art implementations like Metis [11], Jostle [21] or Party [17] follow the multilevel scheme [7] with a local improvement heuristic based on exchanging vertices between partitions. This heuristic reduces the number of cut-edges or the boundary size as well as balances the partition sizes. Hence, the final solution quality mainly depends on this method. Implementations are mostly based on the Kernighan-Lin (KL) heuristic [12], while the local refinement in Party is derived from theoretical analysis with Helpful-Sets (HS) [8]. To address the load balancing problem during parallel computations, distributed versions of the libraries Metis [20] and Jostle [22] have been developed. However, due to the sequential nature of the KL heuristic, their parallelization is difficult. This situation is even worse with the HS heuristic in Party due to the large overhead for exchanging large vertex sets.

While the global edge-cut is the classical metric that most graph partitioners optimize, it is not necessarily the best metric to follow [6] because it does not model the real communication and runtime costs of FEM computations. Hence, different metrics have been implemented to model the real objectives more closely [16,11]. As an example, since the convergence rate of the CGBI solver in the PadFEM environment depends on the geometric shape of a partition, its load balancer iteratively decreases the partitions' aspect ratios by applying the algorithm "Bubble" [3], whose basic idea appeared already in [23]. Yet, its implementation contains a strictly sequential part and suffers from some other difficulties described in [18]. Details about this algorithm and how to overcome its issues are discussed in the following.

2.2 The Bubble Framework

The bubble framework is related to the k-means algorithm well-known in cluster analysis [13] and transfers its ideas to graphs: First, one chooses randomly for each partition one vertex as its center vertex. With this initial set of seed vertices at hand, all remaining vertices are assigned to their closest seed based on some distance measure. (This resembles the simultaneous growth of soap bubbles starting at the seed vertices and colliding at common borders.) After all vertices of the graph have been assigned this way, each sub-domain computes its new center, which acts as the seed in the next iteration. This can be repeated until a stable state is reached. Fig. 1 illustrates the three main operations. This framework can be implemented in various ways, but many approaches show some major disadvantages for our given problem (cf. [14] for a broader discussion). They can be overcome by the growth mechanism explained next.

3 The Diffusion Based Growth Mechanism

Our implementation of the bubble operations is based on solving diffusion problems on the input graph. This is due to the fact that diffusion prefers densely

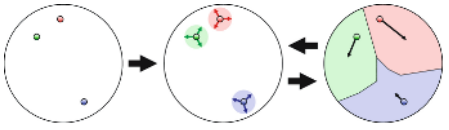

Fig. 1. The main bubble framework operations: Determine initial seeds for each partition (left), grow partitions around the seeds (middle), move seeds to the partition centers (right)

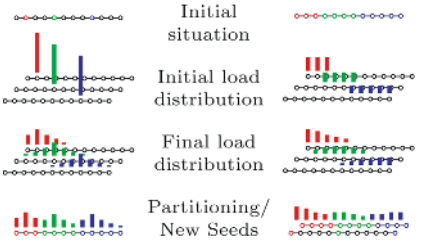

Fig. 2. Schematic view: Placing load on single vertices (left) or a partition (right), the diffusion process, and the mapping of vertices to the partitions according to the load

connected regions of the graph. Thus, one can expect to identify vertex sets that tend to possess a small number of boundary vertices.

3.1 The FOS/C Diffusion Scheme

Generally speaking, a diffusion problem consists of distributing load from some given seed vertex (or vertices) into the whole graph by load exchanges between neighbor vertices. Standard diffusion schemes like FOS [1] converge to fully balanced load distributions. This is undesired here because the amount of load should represent a distance between vertices. Hence, we disturb FOS to obtain a hill-like load distribution with meaningful diffusion distances between vertices.

How this hill-like distribution is interpreted as distance values is illustrated in figure 2. Given a seed vertex for each partition (left), we place load on the respective seed and use a diffusive process to have it spread into the graph. This is performed independently for every partition. After the load is distributed, we assign each vertex to that partition it has obtained the highest load amount from (highest load means shortest distance). The next step (right) does not place load on a single seed vertex only, but distributes it evenly among all vertices of the given partition. After performing the diffusion process, the resulting load distribution can either be used as an optional consolidation or for contracting the partitions to the seed vertices of the next iteration. A consolidation again assigns the vertices to partitions according to the highest load as in the previous step. This further improves the partition shapes. During a contraction, for each partition the vertex containing the highest load becomes its new seed.

We now restate some important properties of the diffusion scheme applied (for details cf. [14]). Let $\mathbf{L}$ be the Laplacian matrix of the unweighted, connected input graph $G = (V, E)$. Shifting a small load amount δ (drain) from each vertex back to the seed vertex/vertices (comprised in the set $S \subset V$) in each iteration by the drain vector d leads to the desired disturbed diffusion scheme called FOS/C

with the matrix/vector notation $w^{(i+1)} = \mathbf{M}w^{(i)} + d$, where $\mathbf{M} = \mathbf{I} - \alpha\mathbf{L}$ is the diffusion matrix with some suitable constant $\alpha > 0$, $w^{(i)}$ is the load vector at iteration i and d the drain vector, whose vector sum is 0, so that $d \in range(\mathbf{L})$.

Theorem 1 (Convergence of FOS/C). *[14] The FOS/C scheme converges for any arbitrary initial load vector $w^{(0)}$.*

Corollary 1. *[14] The convergence state $w^{(*)}$ of FOS/C can be characterized as $w^{(*)} = \mathbf{M}w^{(*)} + d \Leftrightarrow (\mathbf{I} - \mathbf{M})w^{(*)} = d \Leftrightarrow \alpha\mathbf{L}w^{(*)} = d$. Hence, the convergence state can be determined by solving the linear system $\mathbf{L}w = d$, where $w = \alpha w^{(*)}$.*

The resulting load vector w represents the hill-like distribution we need in order to compute diffusion distances between a seed and some other vertex. Therefore, we have the choice to compute this vector by local operations (e. g. the second order diffusion scheme [4]) or by generally faster solvers using global knowledge (such as CG or multigrid), whichever is more appropriate.

3.2 Improving the Matrix Condition

Observe that if load diffuses faster into dedicated regions, then the flow over the edges directing there must be higher than the flow over edges pointing elsewhere. Due to [9] and [2] we know that the solution of the FOS/C diffusion problem is equivalent to a $\|\cdot\|_2$-minimal flow over the edges of the graph. The diffusion problem can therefore be regarded as a flow problem, too. To make the sink of the flow unique, we insert an extra vertex into the input graph G of n nodes as in [15]. This new vertex is connected to every other vertex in G by an edge of weight $\phi > 0$, which leads to a modified Laplacian matrix $\mathbf{L}_\phi$ having one additional row and column whose off-diagonal entries are all $-\phi$. The diagonal of $\mathbf{L}_\phi$ contains for each row the weighted degree of the corresponding vertex, so that it is symmetric positive-semidefinite (spsd) and of rank n. The resulting linear system is denoted by $\mathbf{L}_\phi w_\phi = d_\phi$ with the following drain vector d_ϕ:

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

Solving this spsd system by iterative methods can be made faster and more robust to numerical imprecision by fixing entries (as many as the dimension of the null space of $\mathbf{L}_\phi$) of w_ϕ and deleting their corresponding rows and columns from the matrix [10]. Hence, we improve on previous work [15,14] by fixing the value of the extra vertex to be zero and delete the row and column appended to $\mathbf{L}$ before. What remains is the addition of ϕ to the diagonal values of $\mathbf{L}$. This results in a symmetric positive-definite (spd) matrix whose condition can

be controlled by the parameter ϕ (therefore we actually solve $\mathbf{L}'w' = d'$, where $\mathbf{L}' = \mathbf{L} + \phi\mathbf{I}$ and d' (resp. w') equals d_ϕ (resp. w_ϕ) without the entry for the extra vertex). Note that this simple preconditioning is well-defined by the notion of the extra vertex.

Using $\mathbf{L}'$ has even more advantages than improving the convergence and robustness of iterative solvers: the distributions of different seeds are comparable without post-processing because the extra vertex acts as a common reference point. Moreover, unlike in [15], the extra vertex is eliminated from the actual solution process, which makes the use of multigrid/multilevel methods easier and further speeds up computations.

4 Parallel Implementation of Bubble-FOS/C

In this section we present the Bubble-FOS/C algorithm, its new MPI based implementation, and show improvements to the algorithm in terms of runtime and memory consumption. For sake of simplicity we denote the linear system of our diffusion/flow problem from now on $\mathbf{L}w = d$, although it has the structure of $\mathbf{L}'w' = d'$ from the previous Section. A specific system corresponding to partition p is denoted by $\mathbf{L}w_p = d_p, p \in \{1, ..., P\}$.

4.1 The Bubble-FOS/C Heuristic

```
Algorithm Bubble-FOS/C(G, π, l, i)
01 in each loop l
02   if π is undefined π = determine-seeds(G)
03   else parallel for each partition p
04     centers = Contraction(G, π)
05   parallel for each partition p
06     π = AssignPartition(G, centers)
07   in each iteration i
08     parallel for each partition p
09       π = Consolidation(G, π)
10     π = scale-balance(π)
11   π = greedy-balance(π)
12 return smooth(π)
```

Fig. 3. Sketch of the algorithm

Incorporating FOS/C into the bubble framework results in the algorithm sketched in Figure 3. It can be invoked with or without a valid partitioning π. In the latter case, we determine initial seeds randomly (line 2). Otherwise, we contract the given partitions (lines 3-4) by applying the proposed mechanism based on solving the P linear systems $\mathbf{L}w_p = d_p$. Then, we determine a partitioning (lines 5-6) before performing optional consolidations (lines 7-9). These consolidations can also be used for balancing by scaling the vectors w_p (line 10). This approach can quickly find almost balanced solutions in most cases. If necessary, we perform an additional greedy balancing operation (line 11) to guarantee a certain partition size.

Depending on the quality of the initial solution, it is advisable to repeat the learning process several times. (A multilevel scheme can help to keep the number of repetitions small [14].) Before returning the partitioning π, vertices can be migrated optionally if the number of their neighbors in another partition is larger than the number in their own partition (line 12). This further smooths the partition boundaries but might lead to a slightly higher imbalance.

4.2 Partial Graph Coarsening

As explained above, one needs to solve P linear systems $\mathbf{L}w_p = d_p$ with the same matrix $\mathbf{L}$ and a different right-hand side d_p for each bubble operation based on FOS/C. However, since a vertex is assigned according to the maximum load value, we notice that only a part of the solution is relevant to the vertex assignment due to its hill-like manner. Hence, it is not necessary to compute the exact solution for all vertices of the graph, but only in the important areas surrounding the respective partition, and an approximation elsewhere. This observation can be exploited to both speed up the computations and reduce the memory requirements in a parallel implementation:

Before the first computation, each domain creates a local level hierarchy. Similar as in state-of the-art graph partitioning libraries, this is achieved by calculating a 2-approximation of a maximum weighted matching restricted to edges connecting local vertices. These are then combined to form the vertices of the next level. After that, the implemented data structure allows us to solve linear systems that are composed of different levels of the hierarchy, reflecting the different solution accuracies on the domains.

To solve a linear system, we project the drain vector onto the respective vertices of the lowest hierarchy level and first compute load values there. Figure 4 (left) illustrates a solution for one partition on the lowest levels. One can see that the highest solution values can be found close to the originating domain. Since the matching process preserves the graph structure, the solution on the lowest level is similar to the expected load distribution in the original graph. Hence, we are able to use it to determine the most relevant parts of the solution. Important domains will be switched to a higher hierarchy level while the unimportant ones remain on the lowest one.

The approximate solution is then interpolated to higher levels where necessary and the system is solved again more accurately. Figure 4 (middle) gives an example of a load distribution that has been calculated with varying accuracy. In the important regions of the graph, the linear system is solved on the highest hierarchy level, that is the original graph, while in areas further away from the respective domain lower levels are used.

Although we now have to solve two linear systems per partition, a small one on the lowest hierarchy levels and the second one on the mixed levels, less runtime

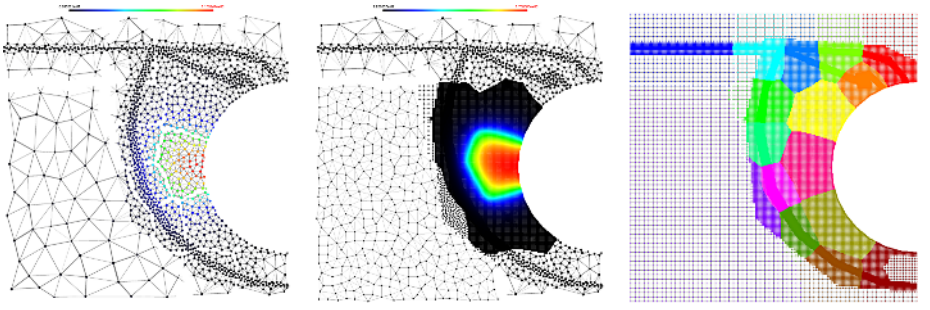

Fig. 4. Vertex loads on the lowest levels (left) and the final solution with local accuracy on the respective levels (middle). The shown solution has been computed for the pink domain leading to the displayed partitioning (right). Edges between vertices of different domains (the initial partitioning) are cut.

is required in total compared to solving one system on the original graph. The lowest levels of the hierarchy are very small and can be processed quickly. The additional time spent in this computation is compensated by the reduction of the system sizes in the second computation.

Note that for each of the P linear systems a different part of the graph is important. Hence, on each domain a number of hierarchy levels contribute to the respective solutions. In our implementation, all systems are solved simultaneously with a standard CG solver. Therefore, we are able to combine the data sent by all P instances and reduce the number of necessary messages.

4.3 Domain Decomposition vs Domain Sharing

Usually a domain decomposition is applied to distribute a graph on a parallel computer. Following this practice, the implemented CG solver requires three communications per iteration, one matrix communication that updates the halo values and two scalar products. Hence, the number of messages is proportional to the number of iterations, which typically grows with the system size.

Since we solve P linear systems concurrently, a second possibility to distribute the computations onto the processing nodes exists. Instead of letting every node process the chosen hierarchy level of its own domain for each of the P systems, it is possible to assemble one complete linear system on each processor. The systems are then solved locally without any communication, and finally the solution is sent back to the domains. We call this approach *domain sharing*.

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

5 Experimental Results

In this Section we present some of our experiments executed on a Fujitsu Siemens hpcLine2. This system consists of 200 computing nodes, each of which has two Intel Xeon 3.2 GHz EM64T processors and 4 GB RAM. In our tests, we only use a single processor per node. We apply the Intel Compiler 8.1 and the Scali MPI implementation via the Infiniband interconnection. The test set comprises a number of two- and three-dimensional FEM graphs of different sizes. Since the results are similar, we only include five of them here.

As mentioned, it has already been shown [15,14] that the Bubble-FOS/C algorithm is able to produce partitionings with few boundary vertices. Since the solution quality varies only little in the settings, we focus our attention on the run-time improvements here.

Table 1 displays the recorded run-times for the five selected graphs. The first column contains the values for the classical domain decomposition approach without constructing a level hierarchy. Note that with the number of partitions the number of linear systems doubles, as well as the number of CPUs. Hence, in the optimal case of this setting all run-times were roughly the same, slightly varying due to the different right hand sides of the linear systems and the resulting number of CG iterations. Of course, the communication overhead prohibits this.

The middle column lists the run-times applying the level approach and domain decomposition. Though we construct the hierarchy and solve the additional small systems on the lowest levels, usually comparable run-times for 8 processors can be achieved. Note that with 8 processors it is very likely that every part of the hierarchy has to be solved on the highest level, especially for three-dimensional graphs, since almost all domains share a common border. If the number of partitions increases, we notice some run-time reduction in contrast to the approach

Table 1. Running times (s) for Bubble-FOS/C algorithm without the level approach (nolevel, domain decomposition by default), with hierarchy and domain decomposition (DD), and with hierarchy and domain sharing (DS). The shock9 ($|V| = 36476, |E| = 71290, \phi = 0.008$), ocean ($|V| = 143437, |E| = 409593, \phi = 0.06$), wave ($|V| = 156317, |E| = 1059331, \phi = 0.07$), auto ($|V| = 448695, |E| = 3314611, \phi = 0.125$), and hermes ($|V| = 320194, |E| = 3722641, \phi = 0.12$) graphs have been repartitioned on 8, 16, 32, and 64 processors respectively.

	nolevel (DD)				DD				DS			
Graph	8	16	32	64	8	16	32	64	8	16	32	64
shock9	2.68	2.97	3.53	4.17	2.49	2.26	1.53	1.74	1.96	1.34	0.99	1.01
ocean	7.27	8.43	8.90	9.61	8.79	6.67	4.90	8.17	8.84	4.75	3.17	3.02
wave	17.93	18.48	20.75	37.02	31.90	25.76	26.79	25.26	43.67	27.40	18.16	10.22
auto	37.92	40.84	48.39	53.08	126.15	106.04	79.08	52.19	141.93	77.88	42.30	24.53
hermes	70.89	74.35	77.63	112.93	191.71	144.11	95.17	87.26	165.75	105.31	54.08	29.08

without levels. However, for 64 processors the run-times increase again for the two small graphs, which can be explained with the increasing communication overhead.

The results of the domain sharing approach can be found in the right column. Although large messages are sent before and after solving the linear systems, it turns out that avoiding communication inside the CG solver speeds up the calculation significantly. This advantage becomes larger with a growing number of partitions, because the fraction of vertices where no exact solution is required increases as well. Hence, for larger number of processors (32 and more) this new scheme shows a clear improvement to the original method regarding run-time.

6 Conclusion

We have presented the parallel load balancing heuristic Bubble-FOS/C and significant improvements concerning a parallel implementation. By introducing an extra vertex, we are able to improve the condition of the involved matrices and therefore the numerical stability and complexity, without changing the matrix structure. Constructing local hierarchies and solving the linear systems with partial accuracy reduces the problem size and therefore the memory requirements. This allows us to solve the linear systems locally and avoid high latency communication inside the solver, which leads to a significant run-time reduction in case of a larger number of partitions.

In the future, it would be interesting to replace the Conjugate Gradient method and combine the presented hierarchical approach with a faster algebraic multigrid solver instead. Note that the latter is based on hierarchy levels by default.

References

[1] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Parallel and Distributed Computing*, 7(2):279–301, 1989.

[2] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25(7):789–812, 1999.

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

[4] R. Elsässer, B. Monien, and R. Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systems*, 35:305–320, 2002.

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

[6] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Irregular'98*, number 1457 in LNCS, pages 218–225, 1998.

[7] B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing'95*, 1995.

[8] J. Hromkovic and B. Monien. The bisection problem for graphs of degree 4. In *Math. Found. Comp. Sci. (MFCS '91)*, volume 520 of *LNCS*, pages 211–220, 1991.
[9] Y. F. Hu and R. F. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25(4):417–444, 1999.
[10] E. F. Kaasschieter. Preconditioned conjugate gradients for solving singular systems. *J. of Computational and Applied Mathematics*, 24(1-2):265–275, 1988.
[11] G. Karypis and V. Kumar. *MeTis: A Software Package for Partitioning Unstrctured Graphs, Partitioning Meshes, [...], Version 4.0*, 1998.
[12] B. W. Kernighan and S. Lin. An efficient heuristic for partitioning graphs. *Bell Systems Technical Journal*, 49:291–308, 1970.
[13] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. Berkeley, University of California Press, 1967.
[14] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid. In *Proc. 20th IEEE Intern. Parallel and Distributed Processing Symposium, (IPDPS'06)*, page 57 (CD). IEEE Computer Society, 2006.
[15] H. Meyerhenke and S. Schamberger. Balancing parallel adaptive fem computations by solving systems of linear equations. In *Proc. Euro-Par 2005*, number 3648 in LNCS, pages 209–219, 2005.
[16] L. Oliker and R. Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *J. Par. Dist. Comp.*, 52(2):150–177, 1998.
[17] S. Schamberger. Graph partitioning with the Party library: Helpful-sets in practice. In *Comp. Arch. and High Perf. Comp.*, pages 198–205, 2004.
[18] S. Schamberger. On partitioning FEM graphs using diffusion. In *HPGC, Intern. Par. and Dist. Processing Symposium, IPDPS'04*, page 277 (CD), 2004.
[19] S. Schamberger. A shape optimizing load distribution heuristic for parallel adaptive FEM computations. In *Parallel Computing Technologies, PACT'05*, number 2763 in LNCS, pages 263–277, 2005.
[20] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Par. Dist. Comp.*, 47(2):109–124, 1997.
[21] C. Walshaw. *The parallel JOSTLE library user guide: Version 3.0*, 2002.
[22] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *J. Parallel Computing*, 26(12):1635–1660, 2000.
[23] C. Walshaw, M. Cross, and M. G. Everett. A Localised Algorithm for Optimising Unstructured Mesh Partitions. *Intl. J. Supercomputer Appl.*, 9(4):280–295, 1995.