LOAD BALANCING FOR ADAPTIVE MULTIGRID METHODS

PETER BASTIAN*

Abstract. This paper presents two algorithms solving the load balancing problem arising in a data parallel implementation of multigrid methods on unstructured, locally refined meshes. The differences between additive and multiplicative multigrid and their influence on the load balancing procedure are discussed in detail. The quality of the proposed algorithms is assessed by numerical experiments on several parallel computers.

Key words. multigrid, unstructured grids, adaptive local grid refinement, MIMD computer, message passing, distributed memory, dynamic load balancing.

AMS(MOS) subject classifications. 65F10, 65N55

1. Introduction.

The self-adaptive solution of elliptic partial differential equations (pdes) showing local solution phenomena is becoming increasingly popular. The basic components of such a procedure - solver, error estimator and grid refinement - have been studied extensively in recent years and a number of implementations already exist [1], [8], [11]. The most efficient solvers used in this context are multigrid methods (see [9]) since they are optimal with respect to the number of iterations and work count per cycle. Specifically the BPX method [6] and its multiplicative counterpart [7] will be used in connection with various smoothers, see also [2],[3],[4] on this topic.

As a model problem, the pde

$$-\nabla \cdot (\epsilon(x)\nabla u) = f(x) \tag{1}$$

is solved in the open domain $\Omega \subset \mathbb{R}^2$ and $0 < \alpha \leq \epsilon(x) \leq M$ holds throughout the domain. Eq. (1) is supplemented with the boundary conditions

$$\begin{aligned} u(x) &= 0 \text{ on } \Gamma_1 &(2)\\ (\epsilon(x)\nabla u, n) &= g_2 \text{ on } \Gamma_2 = \partial\Omega \setminus \Gamma_1 \quad, &(3)\end{aligned}$$

where n is the unit outer normal and $\Gamma_1 \neq \emptyset$ (for simplicity of presentation homogeneous Dirichlet boundary conditions are imposed on Γ_1).

For the discretization of eq. (1) a standard conforming finite element method is used. The mesh consists of triangular elements and V_h denotes the space of continuous functions that are linear on each triangle and zero on Γ_1. The discrete solution $u_h \in V_h$ then solves

$$a(u_h, v) = f(v) = (f, v)_\Omega + (g_2, v)_{\Gamma_2} \qquad \forall v \in V_h \tag{4}$$

* Institut für Computeranwendungen III, Universität Stuttgart, Pfaffenwaldring 27, D-70569 Stuttgart, Federal Republic of Germany (peter@ica3.uni-stuttgart.de)

with

$$(5) \qquad a(u,v) = \int\limits_{\Omega} \epsilon(x)\nabla u \cdot \nabla v dx, \qquad (f,v)_{\Omega} = \int\limits_{\Omega} fvdx, \qquad (g,v)_{\Gamma_2} = \int\limits_{\Gamma_2} gvds.$$

The complete parallel adaptive solution strategy then reads as follows:

(1) Start with an initial unstructured and coarse mesh that resolves the basic details of the geometry.
(2) Discretize the pde and solve the resulting system of linear equations up to a certain accuracy. If the grid hierarchy contains more than one level a fixed number of multigrid cycles is sufficient (nested iteration). Additive and multiplicative multigrid variants as preconditioners in a conjugate gradient algorithm will be used.
(3) Estimate the discretization error. For our model problem eq. (1) the error is estimated in the H^1-norm. If the error is below a given tolerance then exit.
(4) Tag an element for refinement when its local error contribution is above some tolerance level derived from all local contributions (average or maximum).
(5) Compute a new mapping of the grid hierarchy to the processors of the parallel computer that minimizes computation time (approximately). Load balancing will be different for additive and multiplicative multigrid variants.
(6) Migrate the data according to the new mapping. It is important to do this before the subsequent refinement step since less data must be moved and load will already be balanced for the refinement step (at least in the multiplicative case).
(7) Refine the mesh in parallel according to the element tags.
(8) Interpolate the solution to the new mesh points. Goto step (2).

The rest of the paper concentrates on step (5) of the adaptive procedure and is organized as follows. The next section describes the construction of the multigrid hierarchy and the two multigrid variants. The subsequent section presents the load balancing problem for multigrid hierarchies in comparison to load balancing for single-grid methods. The following two sections contain the load balancing algorithms for multiplicative and additive multigrid variants. Then numerical results are presented in order to evaluate the algorithms and the last section draws some conclusions.

2. Local multigrid.

In this section a method for the construction of locally refined mesh hierarchies is described. This hierarchy will then be used in the additive and multiplicative multigrid methods.

2.1. Grid hierarchy.

The multigrid method works on a sequence of successively finer meshes. The initial mesh is intentionally coarse but should be fine enough to resolve important details of the geometry. The sequence of finer meshes is constructed with the refinement algorithm of BANK that is also used in the codes PLTMG (see [1]) and KASKADE (see [8]).

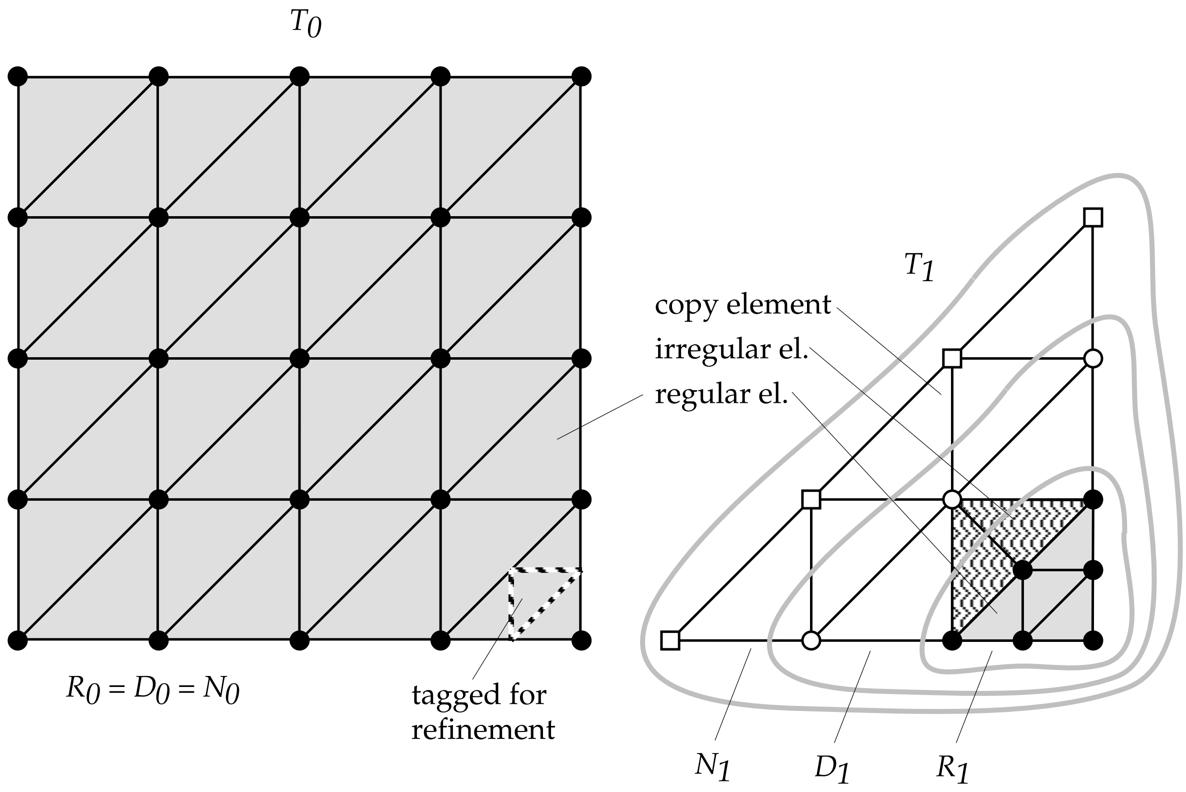

FIG. 1. *Nested local grid refinement*

The refinement algorithm is explained with the help of Fig. 1. The elements of the coarsest grid level T_0 are defined to be regular elements. Then a refinement rule can be applied to each element resulting in the generation of new elements on the next finer level. Each refinement rule is either of type regular, irregular or copy (see Fig. 2 for all possible rules), producing regular, irregular or copy elements on the next finer level. An irregular or copy element allows only the application of copy rules, whereas all types of rules can be applied to regular elements. This strategy generates meshes satisfying a minimum angle criterion since irregular refinement rules can only be applied once.

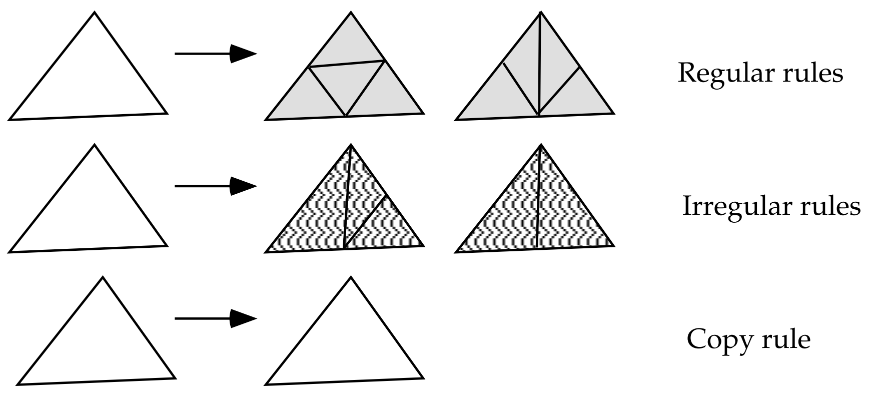

FIG. 2. *Complete set of refinement rules*

The refinement algorithm is responsible for generating a conforming mesh on each level, i.e. the intersection of two different elements is either empty, a node or an edge. In practice it will also happen frequently that the error estimator decides to refine irregular elements or a regular element with an irregular neighbor. In that case the irregular refinement is removed and replaced by a regular refinement rule.

2.2. Subspace decomposition. Both multigrid methods are based on the same decomposition of V_h into suitable subspaces V_k. In order to define V_h and its decomposition from a given grid hierarchy let us denote by

$$T_k = \{t_1^k, t_2^k, \ldots\}, \quad N_k = \{n_1^k, n_2^k, \ldots\} \tag{6}$$

the set of triangles and the set of nodes on level k. We will also need the following two subsets of the nodes N_k:

$$\begin{aligned} R_k &= \{n \in N_k \mid n \text{ is a corner of a } \textit{regular} \text{ element } t \in T_k\}\,, \\ D_k &= \{n \in N_k \mid n \text{ is a corner of } t \in T_k \text{ and } t \text{ has at least one corner } m \in D_k\}\,. \end{aligned} \tag{7}$$

By construction we have $R_k \subseteq D_k \subseteq N_k$. Fig. 1 illustrates these definitions for a two level hierarchy. On level 0 we have $R_0 = D_0 = N_0$ since all elements are regular.

Assume now that a grid hierarchy $T_0, T_1, \ldots, T_J$ with $J+1$ levels is given. For the definition of the finite element space V_h we introduce the “finest mesh”:

$$T_h = \bigcup_{k=0}^{J} \{t \in T_k | \quad t \text{ is } \textit{not} \text{ refined further}\} \tag{8}$$

with the corresponding set of nodes

$$N_h = \{n_1^h, n_2^h, \ldots\} \quad . \tag{9}$$

V_h is then generated by the standard nodal basis

$$\Phi_h = \{\varphi_1^h, \varphi_2^h, \ldots\}, \quad \varphi_i^h(n_j^h) = \delta_{ij} \tag{10}$$

where φ_i^h is linear on each $t \in T_h$ and $n_i^k, n_j^k \notin \Gamma_1$.

Each grid level T_k defines a subspace $V_k \subseteq V_h$. A basis Φ_k of V_k is given by

$$\Phi_k = \left\{ \varphi_i^k \,\middle|\, \begin{array}{l} \varphi_i^k \text{ is the standard nodal basis function corresponding} \\ \text{to node } n_i^k \in R_k \text{ and } n_i^k \notin \Gamma_1 \end{array} \right\}. \tag{11}$$

Note that only the nodes corresponding to R_k are used in the definition of Φ_k. By construction we have that $V_h = \cup_{k=0}^{J} V_k$ and the subspaces are overlapping, i.e. the representation $v_h = \sum_{k=0}^{J} v_k$, $v_k \in V_k$ is, in general, not unique.

For the implementation (see below) also the basis functions corresponding to nodes in N_k are occasionally needed. They are denoted by Φ_k^N.

2.3. Additive multigrid. Expanding all functions in equation (4) with respect to the basis Φ_h of V_h results in a system of linear equations

$$A_h x_h = b_h \tag{12}$$

with $(A_h)_{ij} = a(\varphi_i^h, \varphi_j^h)$, $(b_h)_i = f(\varphi_i^h)$. Eq. (12) is solved with the help of the linear systems

$$A_k x_k = b_k, \quad k \in \{0, 1, \ldots, J\} \tag{13}$$

that are defined with respect to the subspaces V_k. In fact the system (12) is never assembled explicitely, only the systems with respect to the subspaces are needed.

Given a current iterate x_h^n, the iteration error is defined as $e_h^n = x_h - x_h^n$. The iteration error is the solution of the defect equation

$$A_h e_h^n = d_h^n = b_h - A_h x_h^n \quad . \tag{14}$$

The additive multigrid method computes approximations v_k to e_h^n *on each grid level simultaneously* by solving a linear system of the form $A_k v_k = d_k^n$ *approximately* with a linear iterative method. The right hand side d_k^n in this equation can only be computed efficiently by a recursive process from level J down to level 0. Similarly the final update step $x_h^{n+1} = x_h^n + v_h$ will require a recursive interpolation procedure as in standard multigrid. The complete algorithm for additive multigrid is now given:

ALGORITHM 2.1. **Additive multigrid method.** *Uses vectors x_k, b_k, v_k and d_k, $k \in \{0, 1, \ldots, J\}$, of dimension $|N_k|$. The matrix A_k is needed in all rows corresponding to nodes in D_k. On entry x_k contains the current iterate on all levels. Further we have $(b_k)_i = f(\varphi_i^k)$, and $(A_k)_{ij} = a(\varphi_i^k, \varphi_j^k)$, $\varphi_i^k, \varphi_j^k \in \Phi_k^N$.*

```
        amg ($J$) {
(1)         for ($k \in \{0,1,\ldots,J\}$) {
(2)             $d_k = b_k - A_k x_k$; /* on $D_k$ */
(3)             $v_k = 0$; /* on $N_k$ */
            }
(4)         for ($k = J; k \geq 1; k = k-1$)
(5)             $d_{k-1} = r_{k-1}^k d_k$; /* only on $N_{k-1} \cap D_k$ */
(6)         Solve $A_0 v_0 = d_0$;
(7)         for ($k \in \{1,2,\ldots,J\}$)
(8)             $v_k = \mathcal{S}_k^\nu(v_k, d_k)$; /* only on $R_k$ ! */
(9)         for ($k = 1; k \leq J; k = k+1$)
(10)            $v_k = v_k + p_{k-1}^k v_{k-1}$; /* on $N_k$ */
(11)        for ($k \in \{0,1,\ldots,J\}$) $x_k = x_k + v_k$; /* on $N_k$ */
        }
```

In line (2) the defect is computed at all nodes D_k and v_k is set to zero everywhere. The restriction operator r^k_{k-1} from level k to $k-1$ (line 5) changes only values in nodes $N_{k-1} \cap D_k$ on the coarse grid. Besides that, r^k_{k-1} is transposed to the prolongation p^k_{k-1} defined below. In line (8) ν iterations of a standard iterative method like Jacobi or Gauß-Seidel are applied to the system $A_k v_k = d_k$ with the exception that only values in nodes R_k are allowed to be changed. This corresponds to an approximation of the iteration error in the space V_k. In line (10) standard linear interpolation p^k_{k-1} is used to prolongate the correction in all nodes N_k. The update step (11) is also applied to all nodes N_k. This ensures that the correct values in x_k are available on *all* grid levels at the beginning of the next iteration.

2.4. Multiplicative multigrid. In the multiplicative multigrid method the calculation of correction v_k on level k takes into account the corrections calculated on previously visited levels. The algorithm uses the same ingredients as the additive multigrid method:

ALGORITHM 2.2. **Multiplicative multigrid method.** *Vectors x_k, b_k, v_k, d_k and matrix A_k, $k \in \{0, 1, \ldots, J\}$, are the same as in additive multigrid.*

```
        mmg ($J$) {
(1)         for ($k \in \{0,1,\ldots,J\}$) {
(2)             $d_k = b_k - A_k x_k$; /* on $D_k$ */
(3)             $v_k = 0$; /* on $N_k$ */
            }
(4)         for ($k = J; k \geq 1; k = k-1$) {
(5)             $v_k = \mathcal{S}_k^{\nu_1}(v_k, d_k)$; /* only on $R_k$ ! */
(6)             $d_{k-1} = r^k_{k-1}(d_k - A_k v_k)$; /* only on $N_{k-1} \cap D_k$ */
            }
(7)         Solve $A_0 v_0 = d_0$;
(8)         for ($k = 1; k \leq J; k = k+1$) {
(9)             $v_k = v_k + p^k_{k-1} v_{k-1}$; /* on $N_k$ */
(10)            $v_k = \mathcal{S}_k^{\nu_2}(v_k, d_k)$; /* only on $R_k$ */
            }
(11)        for ($k \in \{0,1,\ldots,J\}$) $x_k = x_k + v_k$; /* on $N_k$ */
        }
```

In difference to algorithm 2.1 smoothing (step 5) is done *before* restriction (step 6). Also the restriction takes into account the previously computed correction v_k. As usual in standard multigrid a post–smoothing step has been added (step (10)). Note that algorithm 2.2 is a standard multigrid V–cycle adjusted for the locally refined grid hierarchy. A W–cycle is not used here since it would not have optimal work count for arbitrarily refined grids.

A remarkable difference between algorithms 2.1 and 2.2 is that multiplicative multigrid can be made robust for singular perturbation problems by using a smoother that is an exact solver in the limit case. This strategy is not applicable to additive multigrid which is shown in [4]. For additive multigrid it does usually not pay off to use a good smoother or more than one smoothing step. In the experiments reported below both methods are used as preconditioners in a conjugate gradient algorithm.

The computational effort of `amg` and `mmg` is approximately equal if the same smoother is used but the multiplicative method has better convergence properties. Therefore on a serial computer the additive method has no advantage. However on a parallel machine the algorithms show different synchronization behavior and allow for different load balancing strategies which will be explained in the subsequent sections. For a mathematical analysis of both algorithms we refer to the excellent papers by Xu, [16], and Yserentant, [17].

3. Data partitioning and the load balancing problem. The parallelization of all components of the adaptive multigrid method is based on a distribution of the data onto the set of processors. The assignment of *elements to processors* is used to determine also the mapping of the remaining parts of the data structure (nodes, edges, etc.) to the processors. Overlapping storage of objects on processor boundaries, as shown in Fig. 3, is used for an efficient implementation.

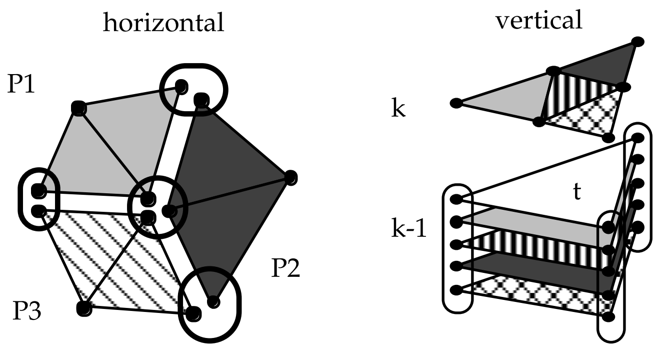

Fig. 3. *Horizontal and vertical overlap in data partitioning*

The left part of Fig. 3 shows the situation on one grid level (intra-grid). The node in the center is stored in three copies on the three processors p_1, p_2 and p_3. The right part of Fig. 3 shows the vertical or inter-grid situation. The white triangle t on level $k-1$ has four son triangles on level k which are assigned to 4 other processors. Then the rule is that for every element assigned to a processor also a copy of the father element must be stored on that processor including all its nodes. Consequently 4 additional copies of triangle t will exist on level $k-1$. Of course it is the aim of the load balancer to avoid this situation as often as possible.

The task is now to determine the mapping of elements to processors such that work is equally balanced between any two synchronization points and communication is minimized. This problem is (even after some abstractions) in the class of NP-complete problems and therefore only heuristic methods are acceptable (remember that multigrid has linear complexity!). Load balancing for single-grid methods like preconditioned conjugate gradient or explicit time-stepping schemes can be abstracted as a *graph partitioning problem*, where the nodes of a graph must be partitioned into equal sized subsets with a minimum number of inter-partition edges. A number of algorithms and software packages are available to solve this problem, see [13], [10], [12]. Load balancing for multigrid methods, however, is more complicated since one iteration consists of several alternating computation and communication steps. Moreover there is communication in the smoother and in the coarse grid correction whose minimization is usually contradictory.

In the following a clustering strategy based on the multigrid hierarchy is proposed that will achieve a compromise between communication in the smoother and in the coarse grid transfer. The outline of the load balancing strategy is as follows:

(i) Combine elements into clusters using the multigrid hierarchy. This step can be done in parallel.
(ii) Transfer all cluster information to the master processor.
(iii) Assign clusters to processors such that each processor has (approximately) the same number of elements on each grid level and communication on each level is low. This is done on a single processor.
(iv) Transfer mapping information back to cluster owners.
(v) Redistribute the data structure in parallel.

Note that the optimization problem of mapping clusters to processors is solved on a single processor. As will be shown below, this is acceptable for todays parallel machines consisting of a moderate number of powerful processors (e.g. up to 256 processors of the CRAY T3D have been used in [3]).

4. Load balancing for multiplicative multigrid. Multiplicative multigrid requires smoothing on each grid level before going to the next grid level. This effectively synchronizes all processors on each level, except if the grid on some level breaks up into several disconnected parts. Therefore the aim of the load balancing scheme for multiplicative multigrid must be to assign the elements of each level equally to the processors. In addition the mapping of two subsequent grid levels must somehow be related in order to reduce inter-grid communication. This is achieved by the following clustering strategy.

4.1. Clustering. For the clustering algorithm a tree-based (local) refinement strategy, as was described above, is required. Let $T = \cup_{k=0}^{J} T_k$ denote the set of all elements (J is the finest level). Since the refinement is based on subdividing individual elements there exists a natural *father* relationship:

$$F_k \subseteq T_k \times T_{k+1}, \quad (t, t') \in F_k \Leftrightarrow t' \text{ is generated by refinement of } t. \tag{15}$$

If J is the highest level then set $F = \cup_{k<J} F_k$. Since the father of an element is unique one can express the relation also in a functional form, i.e. $f(t') = t \Leftrightarrow (t, t') \in F$. The *sons* of an element are defined by

$$s(t) = \{t' \in T | (t, t') \in F\}. \tag{16}$$

All successors of a given element are given recursively by

$$S(t) = \begin{cases} \{t\} & s(t) = \emptyset \\ \{t\} \cup \bigcup\limits_{t' \in s(t)} S(t') & \text{else} \end{cases} \tag{17}$$

and $z(t) = |S(t)|$ defines the number of elements in $S(t)$.

In the mesh T_k two triangles sharing a common edge are said to be neighbors. This relation is formalized by:

$$NB_k \subseteq T_k \times T_k, \quad (t,t') \in NB_k \Leftrightarrow t, t' \text{ share common edge.} \tag{18}$$

Again we sum over all grid levels, which gives $NB = \cup_{k \leq J} NB_k$.

Now the clusters can be defined. In general a clustering $C = \{c_1, \ldots, c_m\}$ is a partitioning of the set T, i.e.

$$T = \bigcup_{c_j \in C} c_j, \quad c_i \cap c_j = \emptyset \Leftrightarrow i \neq j, \quad c_i, c_j \subseteq T. \tag{19}$$

The partitioning defines a mapping $c : T \to C$ from elements to clusters that will also be denoted by c: $c(t) = c \Leftrightarrow t \in c$. Some additional quantities can be derived from the partitioning. First the lowest and highest level of any element in a cluster are given by:

$$bot(c) = \min_k \{k | c \cap T_k \neq \emptyset\}, \quad top(c) = \max_k \{k | c \cap T_k \neq \emptyset\}. \tag{20}$$

Also the number of elements in a cluster on each level is needed:

$$w_k(c) = |\{t \in T_k | t \in c\}|, \quad w(c) = |\{t \in T | t \in c\}|, \tag{21}$$

where $|A|$ denotes the number of elements in set A. In the following it is required that the clustering has the following important properties:

(i) $w_{bot(c)}(c) = 1$ for all clusters $c \in C$. The *unique* $t \in T_{bot(c)} \cap c$ is called the *root* element of the cluster and is denoted by $root(c)$.
(ii) For all clusters $c \in C$ and $c \ni t \neq root(c)$ we require that $f(t) \in c$.

This definition ensures that the elements in a cluster form a *subtree* of the element tree structure. The following algorithm constructs a clustering with the desired properties.

ALGORITHM 4.1. **Clustering of an element set.** The following algorithm `cluster` receives a multigrid hierarchy T (with highest level J) as input and delivers a partitioning into clusters C as output. The parameters b, d, Z control the algorithm. Parameter

b is the *baselevel* since in practice partitioning is started on a level higher than zero if the coarse grids are very coarse or in the dynamic situation where one does not like to rebalance the coarsest levels. Parameter d is the desired depth of the clusters and Z is the minimal size of the clusters.

```
cluster $(J, C, T, b, d, Z)$ {
    $C = \emptyset$;
    for $(k = b, \ldots, J)$
        for $(t \in T_k)$ {
            if $((z(t) \geq Z) \wedge ((k-b) \bmod (d+1) == 0))$ {
                create new $c$; $C = C \cup \{c\}$;
                $bot(c) = top(c) = k$; $root(c) = t$;
                $\forall i : w_i(c) = 0$; $w(c) = 0$;
            } else $c = c(f(t))$;
            $c(t) = c$; $top(c) = \max(top(c), k)$;
            $w_k(c) = w_k(c) + 1$; $w(c) = w(c) + 1$;
        }
}
```

The algorithm proceeds as follows: It runs over all levels from b to J and over all elements within each level. If the subtree defined by the current element is large enough and the level relative to b is a multiple of $d+1$ the current element will be the root of a new cluster else it will be in the cluster of its father element. In the dynamic situation, when the multigrid structure is already distributed over the processors, algorithm cluster can be run in parallel. If the parameters b, d, Z are not changed within one run then only the computation of $z(t)$ requires communication (comparable to a restriction from the finest to the coarsest mesh). In the current implementation the parallel grid refinement algorithm imposes an additional constraint that excludes some elements from becoming the root of a new cluster. Since this constraint will be removed in a new version of the program see [5] for details.

4.2. Balancing the Clusters. After the clustering step, the clusters have to be assigned to processors. This assignement problem is solved on a *single* processor in the current implementation. The assignment heuristic is given by the following two algorithms `mmg_assign` and `assign`.

ALGORITHM 4.2. Algorithm `mmg_assign` maps a set of clusters C to a set of processors P by repeatedly solving smaller assignment problems with particular subsets of C. As parameters it receives b the baselevel used in the clustering algorithm, J the highest level of the multigrid hierarchy and M the minimum number of elements desired per processor. The number M usually depends on the hardware. In parameter *map* the resulting mapping of clusters to processors will be returned. Algorithm `mmg_assign` uses another algorithm `assign` that solves the smaller assignment problems.

```
mmg_assign $(C, P, J, b, M, map)$ {
    for $(p \in P, k = b, \ldots, J)$ $load[k, p] = 0$;
(1)     for ($k = J, J-1, \dots, b$) {
(2)         $C_k = \{c \in C | top(c) = k\}$;
            if ($C_k == \emptyset$) continue;
(3)         $l_k = \sum_{p \in P} load[k,p] + \sum_{c \in C_k} w_k(c)$;
(4)         Determine $P' \subseteq P$ with $|P'| \le \max(1, l_k/M)$;
(5)         assign($k, C_k, P', load, map$);
(6)         for ($c \in C_k$)
(7)             for (i=$bot(c), \dots, top(c)$) $load[i, map(c)] = load[i, map(c)] + w_i(c)$;
        }
    }
```

Algorithm `mmg_assign` proceeds as follows. It uses a two-dimensinal array $load[k,p]$ to store the number of level-k-elements that have been assigned to processor p. Then it proceeds from level J to level b (loop in line 1) and selects the clusters with the currently highest level that have not yet been assigned (line 2). Line 3 computes the number of elements on this level and line 4 determines the number of processors that will be used for that level. Lines 3 and 4 implement a strategy that uses fewer processor when the grids get coarser (controlled by the parameter M). In line 5 algorithm `assign` is called to assign the clusters C_k to the (sub-) set of processors P'. Since some level-k-elements have already been assigned in previous iterations, algorithm `assign` receives also the array *load* to take this into account. Finally lines 6 and 7 update the *load* array.

The following algorithm `assign` is a modification of the recursive bisection idea that is able to take into account that some elements already have been assigned to some processors.

ALGORITHM 4.3. Algorithm `assign` assigns a given set of clusters C to a given set of processors P such that the work on level k of the multigrid hierarchy is balanced. In order to take into account that the processors are already loaded with some elements on level k it receives the array *load*. The output of the algorithm is given by the mapping $map : C \to P$.

```
        assign ($k, C, P, load, map$) {
(1)         if ($P == \{p\}$) {$\forall c \in C$ : set $map(c) = p$; return;}
(2)         Divide $P$ into $P_0, P_1$;
(3)         $l_0 = \sum_{p \in P_0} load[k,p]$; $l_1 = \sum_{p \in P_1} load[k,p]$;
(4)         $W = l_0 + l_1 + \sum_{c \in C} w_k(c)$;
(5)         Determine $C_0, C_1 \subseteq C, C_0 \cup C_1 = C, C_0 \cap C_1 = \emptyset$ such that
(6)             $\left| \frac{|P_0|}{|P_0|+|P_1|} W - \left( l_0 + \sum_{c \in C_0} w_k(c) \right) \right| \to \min$;
(7)         assign($k, C_0, P_0, load, map$);
(8)         assign($k, C_1, P_1, load, map$);
        }
```

Algorithm `assign` proceeds as follows. If P contains only one processor the recursion ends and all clusters in C are assigned to this processor (line 1). Else the set of processors is divided into two halves P_0 and P_1 (line 2). Line 3 then computes the load that has already been assigned to the two processor sets (on level k) and line 4 computes the total load that is available on level k. Now the cluster set C must be divided into two halves C_0 and C_1 such that the number of level-k-elements is equal in both processor sets P_0 and P_1 (lines 5 and 6). Note that P_0 and P_1 are not required to contain the same number of processors. Finally lines 7 and 8 contain the recursive calls that subdivide the new cluster sets again. In this paper only *orthogonal coordinate bisection* (see e.g. [13]) is used for the bisection step in lines (5-6). The coordinate of a cluster is the center of mass of its root element. In [3] results with more elaborate graph partitioning schemes are reported.

5. Load balancing for additive multigrid.

In the case of additive multigrid the communication in smoother and coarse grid correction is not interleaved. Therefore, if a complete subtree $S(t)$ is assigned to one processor, communication during restriction and prolongation is only necessary for the root element t if its father $f(t)$ is assigned to a different processor. Furthermore the communication in the smoother can be delayed after all nodes on all levels have been processed and at most one message has to be exchanged between any pair of processors. Additive multigrid (with one smoothing step) thus can be implemented with the same communication requirements as non-overlapping domain decomposition (but slightly larger messages).

The idea of the additive load balancing algorithm is to find a set of root elements RT such that $\bigcup_{t\in RT} S(t)$ contains most of the total work and the assignment of subtrees $S(t), t \in RT$ to processors allows good load balancing. Note that the elements in RT need not be of the same grid level!

In order to find the set RT one starts with subtrees (which are clusters in the sense of the previous section) that are as large as possible. This is accomplished by algorithm `cluster` when $d \geq J$ is used (then one has $RT = T_b$, b being the baselevel). In a second step the subtrees are divided into smaller clusters when needed (see below). A simple way to break up a cluster into smaller clusters is to define the sons $s(t)$ of the root element as roots of new clusters. This is done by the following algorithm `split`.

ALGORITHM 5.1. **Split clusters into smaller clusters.** Algorithm `split` takes all clusters in C and uses the sons of the root elements as new cluster roots. Since this will be done on the master processor a minimal cluster size Z is required due to memory restrictions. Therefore the results are two cluster sets: C' the set of (smaller) clusters that still can be subdivided further and L' the set of clusters that cannot be subdivided further. In order to avoid clusters containing less than Z elements the definition of a cluster is generalized such that also elements below the root element are allowed to be in a cluster (see explanation of algorithm below).

```
        split $(C, C', L', Z)$ {
(1)         $C' = L' = \emptyset$;
(2)         for each $(c \in C)$ {
(3)             $r = root(c)$; $i = 0$;
(4)             for $(s \in s(r))$ {
```

```
(5)                 if ($|S(s) \cap c| \geq Z$) {
(6)                     $i = i + 1$;
(7)                     create new $c_i$;
(8)                     $c_i = S(s) \cap c$; $c = c \setminus c_i$; $root(c_i) = s$;
(9)                     $C' = C' \cup c_i$;
                    }
                }
(10)            if ($|c| = 1 \wedge i > 0$)
(11)                $c_1 = c_1 \cup c$;
(12)            else $C' = C' \cup c$;
            }
(13)        for each ($c \in C'$) {
(14)            $r = root(c)$; $i = 0$;
(15)            for ($s \in s(r)$)
(16)                if ($|S(s) \cap c| < Z$) $i = i + 1$;
(17)            if ($i = |s(r)|$) { $C' = C' \setminus c$; $L' = L' \cup c$};
            }
        }
```

Lines (1-9) of algorithm `split` subdivide each cluster c of C at its root by assigning a son (and all successors of it) to a new cluster if the new cluster is large enough. Lines (10-12) assigns the old root element to the first new cluster if it would be the only remaining element in cluster c. Note that the old root element will not be the root element of cluster c_1 (the root of c_1 is defined in line (8))! In a second loop through all new clusters in lines (13-17) those clusters are detected that cannot be subdivided further.

The following algorithm assigns clusters to processors for additive multigrid. It uses algorithm `split` in order to subdivide clusters when necessary.

ALGORITHM 5.2. **Load balancing for additive multigrid.** The input of algorithm `amg_assign` consists of C, the set of divisible clusters, L the set of indivisible clusters, P the set of processors available to compute elements in C and L, Z the minimum size of a single cluster, *tol* a parameter controling the load balancing quality and k the current subdivision level. Algorithm `amg_assign` is called recursively. In the initial call C is assumed to be a clustering obtained with algorithm `cluster` using an appropriate baselevel b and a depth $d \geq J$. In addition all clusters in C are assumed to be divisible. The output of the algorithm is given by the mapping $map : C \rightarrow P$.

```
        amg_assign ($C, L, P, Z, tol, map$) {
(1)         if ($P = \{p\}$) {$\forall c \in C : map(c) = p$; $\forall l \in L : map(l) = p$; return; }
(2)         Divide $P$ into $P_0, P_1$;
            while (true) {
(3)             $W_C = \sum_{c \in C} w(c)$; $W_L = \sum_{l \in L} w(l)$;
(4)             $W_{opt} = \frac{|P_0|}{|P_0|+|P_1|}(W_C + W_L)$;
(5)             if ($W_C < W_{opt}$) {
```

```
(6)             $C_0 = C$; $C_1 = \emptyset$;
(7)             Find order for $L$: $a_L : \{1, \ldots, |L|\} \to L$;
(8)             Determine $i \in \{0, \ldots, |L|\}$ such that
(9)                 $\left|W_{opt} - \left(W_C + \sum_{n=1}^{i} w(a_L(n))\right)\right| \to \min$;
(10)            $L_0 = \bigcup_{n=1}^{i} \{a_L(n)\}$; $L_1 = \bigcup_{n=i+1}^{|L|} \{a_L(n)\}$;
            } else {
(11)            $L_0 = \emptyset$; $L_1 = L$;
(12)            Find order for $C$: $a_C : \{1, \ldots, |C|\} \to C$;
(13)            Determine $i \in \{0, \ldots, |C|\}$ such that
(14)                $\left|W_{opt} - \sum_{n=1}^{i} w(a_C(n))\right| \to \min$;
(15)            $C_0 = \bigcup_{n=1}^{i} \{a_C(n)\}$; $C_1 = \bigcup_{n=i+1}^{|C|} \{a_C(n)\}$;
            }
(16)        for $(i = 0, 1)$ $W_i = \sum_{c \in C_i} w(c) + \sum_{l \in L_i} w(l)$;
(17)        if $(\max(W_0, W_1) \le (1 + tol)W_{opt}) \vee (C = \emptyset)$ break;
(18)        split$(C, C', L', Z)$;
(19)        $C = C'$; $L = L \cup L'$;
        }
(20)    amg_assign$(C_0, L_0, P_0, Z, tol \cdot \sigma, map)$;
(21)    amg_assign$(C_1, L_1, P_1, Z, tol \cdot \sigma, map)$;
    }
```

Algorithm `amg_assign` is a recursive bisection strategy. If P contains more than one processor it subdivides P, C and L into two parts each and calls itself recursively. Line (2) bisects the processor set P. The while loop in lines (3-19) bisects C and L until the load balancing tolerance is satisfied or the clusters can not be subdivided further. It starts by computing the number of elements in C and L (line (3)) and the optimal number of elements that should be assigned to processor set P_0 (line (4)). In the following cluster bisection step either C *or* L is bisected, depending on line (5). The bisection is implemented such that the clusters are ordered by their x-coordinate (or y alternatingly) and then a linear search is performed that optimizes load balance. Note that more complicated methods, e.g. spectral bisection, could be used in the ordering step as well. Now the total load for processor sets P_0 and P_1 is computed in line (16). If load balance is good enough or the clusters can not be subdivided further the while loop is exited (line (17)). If the load balancing criterion is not fulfilled the cluster size is decreased by calling algorithm `split` and a new bisection is computed. Lines (20,21) contain the recursive call that now maps C_0, L_0 to P_0 and C_1, L_1 to P_1. Note that the load balancing tolerance *tol* is multiplied with $\sigma \le 1$ (usually $\sigma = 0.5$ is used). This allows one to start with a relatively large tolerance (e.g. $tol = 0.2$) and avoids excessive cluster splitting in the first bisection steps.

6. Numerical results.

6.1. Uniform refinement of an unstructured mesh. The first example solves the Laplace equation ($\epsilon = 1$ in Eq. (1)) on the mesh given in Fig. 4. The coarse mesh (level 0) consists of 210 triangular elements and the finer meshes are obtained by uniform refinement (i.e. no local refinement is employed). Grid level 0 is assigned to one processor, for grid level 1 (840 elements) up to 16 processors are used and grid levels 2 (3360 elements) and higher are allowed to use all processors. The mapping of levels 1 and 2 is done only once before further refinement (with orthogonal coordinate bisection), which means that communication in the coarse grid correction is only possible between levels 0/1 and 1/2. The same mapping is used for multiplicative *and* additive multigrid.

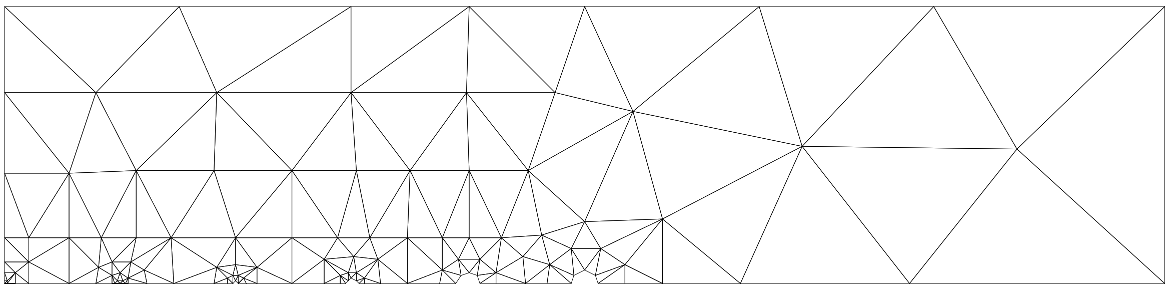

FIG. 4. *Unstructured mesh used in the uniformely refined example.*

Table 1 shows iteration times $T_{it}(p)$ (in seconds), efficiency per iteration defined as $E_{it} = \frac{T_{it}(1)}{T_{it}(p) \cdot p}$ and iteration numbers for a 10^{-6} reduction of the residual in the euclidean norm. The iteration times $T_{it}(1)$ for the larger problems have been obtained by simple multiplication with a factor 4. Method mmg was a multiplicative multigrid V-cycle with one symmetric Gauß-Seidel step used for pre- and post-smoothing. Method amg was an additive multigrid cycle with two symmetric Gauß-Seidel sweeps used as a smoother. Both multigrid cycles were used as preconditioner in a conjugate gradient iteration. The Gauß-Seidel smoother has been parallelized by ignoring data dependencies between processors, i.e. it is better described as a Block-Jacobi-Smoother with inexact inner iteration. The nodes on processor boundaries have been assigned arbitrarily to one of the processors. On the coarsest level the residual is always reduced by a factor of 10^{-4} with an ILU-iteration.

The table shows that additive multigrid needs about twice the number of iterations of the multiplicative multigrid method (a result expected from theory). The time per iteration is about the same for both methods (The results of [4] suggest that using only one smoothing step for amg would be more efficient but this would not change the overall conlusion). The efficiency per iteration is only slightly better for additive multigrid although less messages have to be sent. The bad efficiencies for 27457 unknowns on 64 processors are due to high surface to volume ratio (less than 500 unknowns per processor) and large serial overhead (level 0 is solved on one processor). If the number of unknowns per processor is constant (scalability) the efficiencies are very good and reach almost 90% on 64 processors. Since iteration times and parallel efficiency are comparable for additive and multiplicative multigrid the overall solution time is less for multiplicative multigrid in this example.

TABLE 1
Efficiency per iteration and iteration count for uniformely refined example on Cray T3D.

Levels		5		6		7		8	
Unknowns		27457		108673		432385		1724929	
Method		mmg	amg	mmg	amg	mmg	amg	mmg	amg
$\|P\| = 1$	T_{it}	3.39	2.9	13.56	11.6	54.24	46.4	216.96	185.6
	#IT	10	21						
$\|P\| = 4$	T_{it}	0.952	0.802	3.56	3.03				
	E_{it}	89	90	95	96				
	#IT	10	21	10	23				
$\|P\| = 16$	T_{it}	0.318	0.266	1.00	0.844	3.74	3.18		
	E_{it}	67	68	85	86	91	91		
	#IT	11	24	12	25	12	26		
$\|P\| = 64$	T_{it}	0.150	0.113	0.338	0.278	1.03	0.862	3.85	3.26
	E_{it}	35	40	63	65	82	84	88	89
	#IT	13	26	12	26	12	27	12	28

TABLE 2
Comparison of time per multigrid iteration (in seconds) on different machines for constant load per processor in the uniformely refined example (about 27500 unknowns per processor).

Levels	5	6	7	8	
Processors	1	4	16	64	
Cray T3D	3.39	3.56	3.74	3.85	(1.14)
Intel Paragon	5.26	5.73	6.10	6.27	(1.19)
Parsytec GC PowerPlus	4.46	4.93	5.37	5.56	(1.25)

Table 2 compares scaled iteration time per multigrid iteration (only multiplicative multigrid, same parameters as above) on different parallel machines. The last column of the table shows the scalability factor from one to 64 processors. In conclusion the Cray T3D has the fastest processor *and* the best scalability due to its fast communication network.

6.2. A fixed-size adaptive example. As a first adaptive example Laplace's equation $-\Delta u = 0$ is solved in the L-shaped domain $\Omega = (0,1)^2 \setminus \{(x,y)|x \geq 0.5 \wedge y \leq 0.5\}$ with Dirichlet boundary conditions taken from the exact solution

$$u(r,\phi) = \left(\frac{r}{4}\right)^{\frac{2}{3}} \sin(\frac{2}{3}\phi). \tag{22}$$

The error is estimated in the H^1-norm:

$$\|u - u_h\|_1 \leq C \left(\sum_{t\in\mathcal{T}} \eta^2(t)\right)^{\frac{1}{2}} \tag{23}$$

with

$$\begin{aligned}\eta^2(t) &= h_t^2\|f + \nabla\epsilon \cdot \nabla u_h\|_{0,t}^2 \\ &+ \sum_{e\in E(t)\cap E_N} h_e\|g_2 - e\nabla u_h \cdot n_e\|_{0,e}^2 \\ &+ \sum_{e\in E(t)\cap E_\Omega} \frac{1}{2}h_e\|[\epsilon\nabla u_h]_e\|_{0,e}^2,\end{aligned}$$

where $E(t)$ is the set of edges of triangle t, E_N are the Neumann boundary edges, E_Ω are the interior edges, h_t, h_e is the size of triangle t and edge e and $\|\cdot\|_{0,t}, \|\cdot\|_{0,e}$ are the L_2 norms over triangle t and edge e. A triangle is flagged for refinement if

$$\eta^2(t) \leq 0.27 \max_{t'\in\mathcal{T}} \eta^2(t'). \tag{24}$$

The adaptive procedure is run until

$$\left(\sum_{t\in\mathcal{T}} \eta^2(t)\right)^{\frac{1}{2}} \leq 0.006 \tag{25}$$

is reached. The final finite element space has 25375 degrees of freedom and the multigrid hierarchy contains 70327 triangles distributed over 17 grid levels. The finest grid levels contain only several hundred triangles each in the vicinity of the reentrant corner.

Fig. 5 shows the load distribution for additive and multiplicative multigrid on 16 processors. The picture clearly shows the smaller surface to volume ratio for the additive multigrid load balancer.

Table 3 shows the efficiency per multigrid iteration on the final mesh, E_{it}, the time spent in the multigrid solver for a 10^{-4} reduction of the residual, T_{sol}, and the total computation time T_{total} starting from the initial mesh with 6 triangles. On a single processor the multiplicative multigrid method with 2 Gauß-Seidel post-smoothings is clearly faster than additive multigrid with Jacobi smoothing. Starting with 16 processors the time T_{sol} is smaller for additive multigrid and for 64 processors even total computation time is better for additive multigrid. The overall efficiency however is very poor for both methods in this extreme example with less than 400 unknowns per processor on 17 grid levels in the 64 processor case. It is not advisable to use more than 16 processors in this example which still runs on a small workstation (20 MB storage).

6.3. A scaled-size adaptive example. In the example of this section the problem size is increased with the number of processors by choosing an appropriate tolerance in the error estimator. Eq. (1) is solved in the unit square with the coefficient function ϵ given by the left picture in Fig. 6. This coefficient function varies over four orders of magnitude. The right part of Fig. 6 shows the adaptively refined mesh after 10 levels of refinement. The same error estimator as in the previous example has been used.

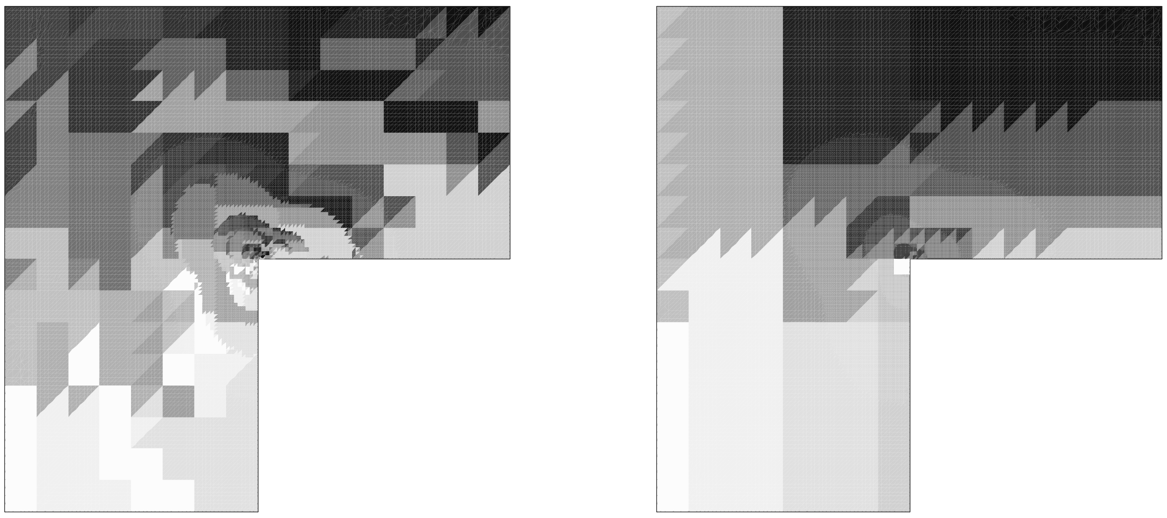

FIG. 5. *Load mapping on 16 processors using multiplicative (left) and additive multigrid (right) for the fixed-size example.*

TABLE 3
Efficiency, solution time and total computation time for fixed-size adaptive example (Intel Paragon).

Processors	cg+mmg+gs(0,2)			cg+amg+djac(1)		
	E_{it}	T_{sol}	T_{total}	E_{it}	T_{sol}	T_{total}
1	100	14.0	123	100	23.5	154
2	88	7.97	81	97	12.1	88
4	65	5.34	56	90	6.56	57
16	38	2.28	39	65	2.25	25
32	26	1.66	24	55	1.34	25
64	15	1.44	24	38	0.97	18

Table 4 shows efficiencies per multigrid iteration, E_{it}, and time spend in the solver, T_{sol}, on the finest mesh with the given number of unknowns. Symmetric Gauß-Seidel was used as a smoother for both multigrid variants. The table shows that multiplicative multigrid is preferable in all cases, although the efficiency per iteration is again better for additive multigrid.

Table 5 now compares total computation time and estimated global errors for uniform and adaptive computations. In the given example a solution with comparable quality can be computed on 48 processors of the Intel Paragon in 159 seconds with uniform refinement or with only 4 processors in 135 seconds with adaptive refinement.

7. Conclusions.

In this paper two load balancing algorithms for adaptive multigrid methods have been proposed, one for multiplicative multigrid and one for additive multigrid. Both methods use the grid hierarchy information in order to achieve a compromise in communication in the coarse grid transfer and in the smoother.

The synchronization behavior of additive multigrid allows a load distribution with

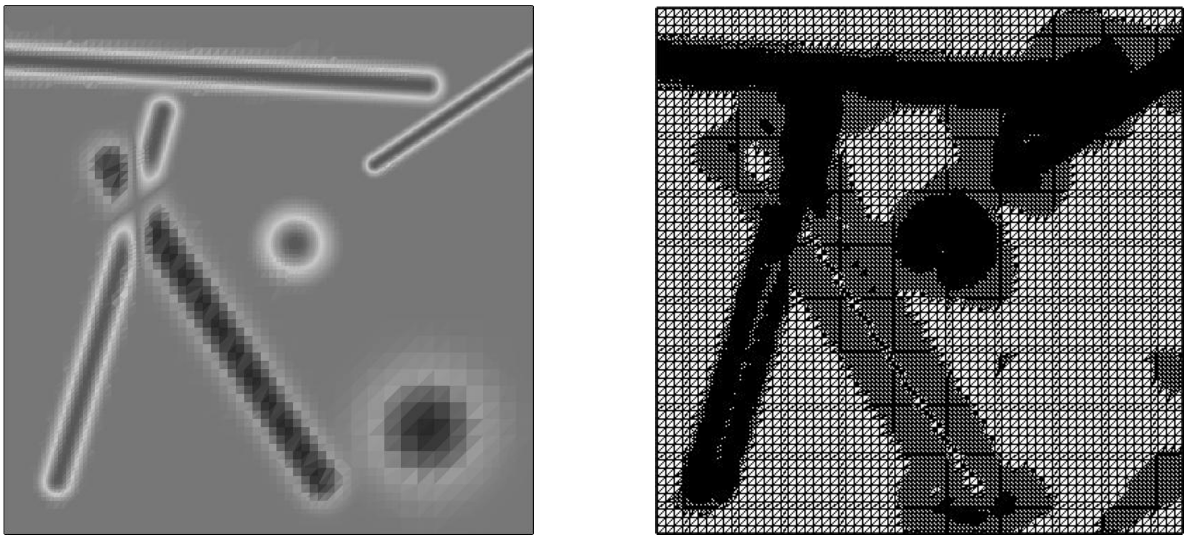

FIG. 6. *Coefficient function $\epsilon(x)$ (left) and adaptively refined mesh (right) for the scaled-size adaptive example.*

TABLE 4
Efficiency per iteration and solution times (in seconds) for the scaled-size adaptive example on Intel Paragon.

Unknowns		18174	48448	155999	551162
Est. Error		0.681	0.36	0.19	0.095
Processors		1	4	12	48
cg+mmg+sgs(2,2)	E_{it}	100	79	73	67
	T_{sol}	36.3	29.4	33.2	26.5
cg+amg+sgs(2)	E_{it}	100	88	82	76
	T_{sol}	102.0	74.3	83.0	72.9

much smaller surface to volume ratio. This results in superior parallel efficiencies per iteration for small problems. However, additive multigrid needs more iterations than multiplicative multigrid. The experiments show that if the problems are reasonably large, multiplicative multigrid has the smaller overall computation time.

The question whether parallel adaptive computation pays off compared to parallel uniform computation is of course very problem dependent. In general one can conclude that if adaptive computation pays off on a sequential computer then it will also be efficient on a parallel computer as long as the problems are reasonably large.

Finally it should be noted that both load balancing schemes can be extended immediately to the three-dimensional situation. The techniques presented in this paper can also be used to solve more complicated equations. They have, e.g., been used successfully for computations in nonlinear structural mechanics.

A. Notation. This appendix recapitulates the most frequently used symbols.

TABLE 5
Comparison of total computation time for uniform and adaptive calculation (Intel Paragon).

Processors	uniform		adaptive	
	time	est. error	time	est. error
1	78	2.76	183	0.681
4	91	1.29	135	0.36
12	116	0.63	135	0.19
48	159	0.31	135	0.095

Ω	open domain in $\mathbb{R}^2$
V_h	fine grid finite element space
Φ_h	nodal basis generating V_h
T_h	set smallest triangles covering Ω ("finest mesh")
N_h	set of nodes of T_h
J	finest level in multigrid hierarchy ($J+1$ levels)
T_k	set of triangles on level k
N_k	set of nodes on level k
D_k, R_k	subsets of N_k defined in (7)
V_k	finite element space on level k
Φ_k	nodal basis for V_k, corresponds to nodes D_k
Φ_k^N	nodal basis corresponding to nodes N_k on level k
T	union of all grid levels T_k
$F_k \subseteq T_k \times T_{k+1}$	father relationship from level $k+1$ to k
$F \subseteq T \times T$	union of all F_k
$f(t)$	father element of $t \in T$
$s(t)$	set of elements having t as father (the sons)
$S(t)$	set of descendants of t
$z(t)$	number of elements in $S(t)$
$NB_k \subseteq T_k \times T_k$	neighbor relationship on level–k–elements
$NB \subseteq T \times T$	union of all NB_k
C	cluster set, i.e. a partitioning of T
$c(t)$	cluster of element t
$bot(c)$	smallest k such that $c \cap T_k \neq \emptyset$
$top(c)$	largest k such that $c \cap T_k \neq \emptyset$
$w_k(c)$	number of level–k–elements in cluster c
$w(c)$	number of elements in cluster c
$root(c)$	unique element in c whose father is not in c
P	processor set

REFERENCES

[1] R. BANK: *PLTMG Users Guide Version 7.0.* SIAM, Philadelphia, 1994.

[2] P. Bastian, G. Wittum: *On Robust and Adaptive Multigrid Methods*, Proceedings of the 4^{th} European Multigrid Conference, Amsterdam, July 1993.

[3] P. Bastian, K. Eckstein, S. Lang: *Parallel Adaptive Multigrid Methods in Structural Mechanics*, submitted to Numerical Linear Algebra with Applications.

[4] P. Bastian, W. Hackbusch, G. Wittum: *Additive and Multiplicative Multi-Grid: a Comparison*, submitted to Numerische Mathematik.

[5] P. Bastian: *Parallele adaptive Mehrgitterverfahren*, Teubner Skripten zur Numerik, Teubner-Verlag, Stuttgart, 1996.

[6] J. H. Bramble, J. E. Pasciak, J. Xu: *Parallel Multilevel Preconditioners*, Math. Comput., **55**, 1-22 (1990).

[7] J. H. Bramble, J. E. Pasciak, J. Wang, and J. Xu, *Convergence estimates for multigrid algorithms without regularity assumptions*, Math. Comp., **57**, (1991), pp. 23–45.

[8] P. Deuflhard, P. Leinen, H. Yserentant: *Concepts of an Adaptive Hierarchical Finite Element Code*, IMPACT of Computing in Science and Engineering, **1**, 3-35 (1989).

[9] W. Hackbusch: *Multi-Grid Methods and Applications*, Springer, Berlin, Heidelberg 1985.

[10] B. Hendrickson, R. Leland: *The Chaco User's Guide Version 1.0*, Technical Report SAND93-2339, Sandia National Laboratory, October 1993.

[11] K. Eriksson, D. Estep, P. Hansbo and C. Johnson: *Introduction to adaptive methods for partial differential equations.* Acta Numerica, (1995).

[12] G. Karypis, V. Kumar: *A fast and high quality multilevel scheme for partitioning irregular graphs.* Technical Report 95-035, University of Minnesota, Department of Computer Science, (1995).

[13] H. Simon: *Partitioning of Unstructured Problems for Parallel Processing*, Computing Systems in Engineering, **2**(2/3), 135-148, (1991).

[14] G. Wittum: *On the Robustness of ILU Smoothing*, SIAM J. Sci. Statist. Comput., **10**, 699-717 (1989).

[15] G. Wittum: *Linear Iterations as Smoothers in Multigrid Methods: Theory with Applications to Incomplete Decompositions*, IMPACT of Computing in Science and Engineering, **1**, 180-215 (1989).

[16] J. Xu: *Iterative Methods by Space Decomposition and Subspace Correction: A Unifying Approach*, SIAM Review, **34**(4), 581-613, (1992).

[17] H. Yserentant: *Old and new convergence proofs for multigrid methods*, Acta Numerica, (1993).