Numerical Linear Algebra with Applications, Vol. 1(1), 1–1 (1996)

Parallel Adaptive Multigrid Methods in Plane Linear Elasticity Problems

Peter Bastian, Stefan Lang

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

and

Knut Eckstein

Graduiertenkolleg Kontinua und Strömungen, ISD, Universität Stuttgart, D-70569 Stuttgart, Federal Republic of Germany (eckstein@isd.uni-stuttgart.de)

In this paper we discuss the implementation of parallel multigrid methods on unstructured locally refined meshes for 2D linear elasticity calculations. The problem of rebalancing the workload of the processors during execution of the program is discussed in detail and a load balancing algorithm suited for hierarchical meshes is proposed. For large problems the efficiency per multigrid iteration ranges from 65% on 64 processors in the locally refined case to 85% on 256 processors in the uniformly refined case. All calculations were carried out on a CRAY T3D machine.

KEY WORDS multigrid, unstructured grids, adaptive local grid refinement, MIMD computer, dynamic load balancing, linear elasticity

1. Introduction

The efficient numerical solution of partial differential equations is an important field of active research. Over the last decades a number of techniques have been developed to reduce computer time. The first possibility to save time is in the discretization step. Here adaptive local grid refinement concentrates degrees of freedom in the critical parts of the solution domain. To that end an a-posteriori error estimator (or indicator) is applied to a given numerical solution and if the error exceeds a prescribed tolerance a modification strategy produces a refined mesh based on local quantities computed by the error estimator.

1070–5325/96/010001–01$5.50

Received 15 December 1995
Revised 30 May 1996

This procedure is repeated until the required tolerance has been reached. A review of these techniques is given in [1].

On each of the adaptively generated meshes a numerical solution has to be computed. Therefore a fast iterative solver is needed that can be stopped when the solution error has reached the discretization error. For elliptic partial differential equations multigrid methods are the fastest methods known so far. They have the important property that their convergence rate is independent of the mesh size. The optimality of the method for scalar elliptic problems on unstructured and locally refined meshes without assumptions on the regularity of the differential operator has been shown only very recently. An overview can be found in the paper by YSERENTANT [2]. The multigrid method fits nicely into the framework of adaptive local grid refinement which has been exploited already in a number of computer codes like PLTMG, [3], or KASKADE, [4].

Yet another method to save computer time is the use of modern computer architectures, i. e. parallel computers. But it has to be emphasized that the use of parallel computers will allow one to solve bigger problems than before only if the methods mentioned above are implemented in parallel. If non-optimal solvers are used on the parallel machine a large part of its capacity is wasted to compensate for the gain that could be achieved by using multigrid on a single processor.

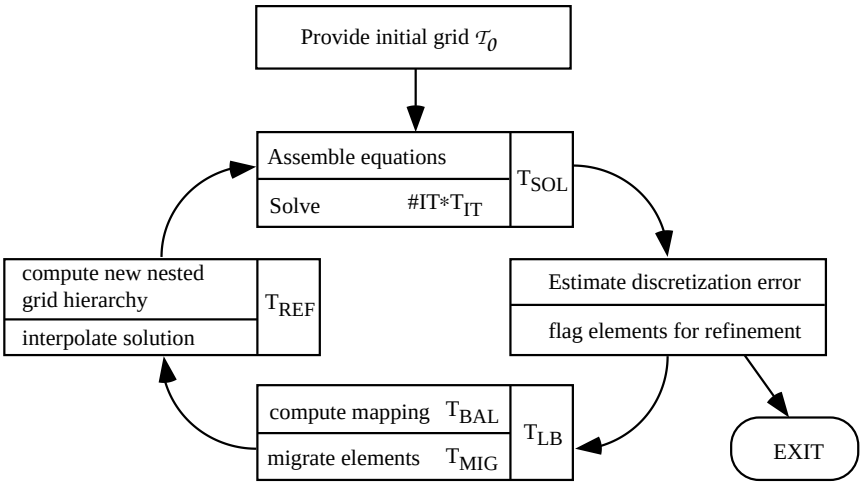

Figure 1. Basic adaptive solution strategy.

Therefore the aim of this paper is to combine adaptive local grid refinement and multigrid on parallel computers. This immediately leads to the problem of dynamic load balancing, since some processors will refine more elements than other processors. Since the location of refinement is in general not known in advance the load balancing problem must be solved during run-time. The load balancing problem consists of two parts: First a mapping of the data (e.g. elements) to the processors must be determined that balances execution time in the solver. Second the data must be migrated according to this new mapping. The computation of the optimal mapping is a NP-complete discrete optimization problem

which must be solved approximately by a heuristic procedure that does not dominate the execution time of the multigrid solver. Fig. 1 shows an outline of the parallel adaptive solution strategy.

This paper is organized as follows. In the next section we will describe shortly the equations of linear elasticity in two space dimensions and their discretization with the Finite-Element method. Section 3 covers the other components of the adaptive solution algorithm, especially the multigrid solver. Section 4 gives an overview of the parallelization approach based on data partitioning. Then section 5 describes in detail the load balancing algorithm that has been developed. Finally section 6 contains some speedup results for uniform and adaptive calculations.

2. Mathematical Model and its Discretization

2.1. Variational Formulation

Subject of the computations presented in this paper is the classical planar, first order theory of elasticity. The restriction to linear material laws is going to be lifted soon, as the routines allowing for nonlinear material laws are currently under implementation.

The calculation of the displacement response of an elastic body subjected to prescribed forces and displacements is equivalent to solving the following variational problem:

$$\Pi = \int_\Omega \left[\frac{1}{2} \epsilon : \sigma - f u \right] dx + \int_{\partial\Omega} g\, u\, dx = min. \tag{2.1}$$

This expression based on the mechanical axiom of static equilibrium represents the minimal energy criterion for the elastic body which covers the domain Ω. The physical interpretation is such that under the above condition of minimal elastic energy the body enters the equilibrium state with the sum of the internal forces being equivalent to the sum of the external forces.

The minimal energy criterion contains three different variables describing the state of the elastic body: The two dimensional displacement vector u, the deformation tensor ϵ and the stress tensor σ. The vectors f and g hold the external forces and the boundary stresses respectively.

As displacement, deformation and stress components of the elastic body are not independent of each other, several approaches to solving the variational problem exist. We choose the so-called "displacement formulation" which uses the dependencies between the description variables to eliminate ϵ as well as σ.

The linear material law

$$\epsilon = \frac{1+\nu}{E}\, \sigma \; - \; \frac{\nu}{E}\, tr(\sigma) \cdot I \tag{2.2}$$

is employed in the elimination of the stress tensor σ where E is the Young's modulus and ν the Poisson number. We choose E to $210,000$ and ν to 0.3 which is a common choice for steel material.

First order linearization of the kinematic coupling conditions between the displacement u and the deformation ϵ yields

$$\epsilon_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right). \tag{2.3}$$

Short-handing the above to $\epsilon = \epsilon(u) = Du$ finally results in

$$\Pi = \int_\Omega \left[\frac{1}{2} Du : CDu - fu \right] dx + \int_{\Gamma_1} g\, u\, dx = min \tag{2.4}$$

with the material tensor C containing expressions of E and ν. Γ_1 is usually being referred to as the Neumann boundary.

The weak form corresponding to the variational problem (2.4) can be written as

$$\int_\Omega \left[\frac{1}{2} Du : CDv \right] dx = \int_\Omega f\, v\, dx + \int_{\Gamma_1} g\, v\, dx \tag{2.5}$$

where the desired solution u has to satisfy this expression for any valid choice of test function v.

2.2. Discretization

In order to find an approximate solution u_h on a given mesh denoted by $\mathcal{T}$ we define the Finite Element space V_h. We choose standard isoparametric triangles and quadrilaterals with piecewise linear/bilinear shape functions ϕ_i which result in 1 at the node n_i and in 0 at all other nodes. Then the discrete problem can be stated as follows: Find $u_h \in V_h \times V_h$ such that

$$\int_\Omega \left[\frac{1}{2} Du_h : CDv \right] dx = \int_\Omega f\, v\, dx + \int_{\Gamma_1} g\, v\, dx \qquad \forall v \in V_h \times V_h. \tag{2.6}$$

In the case of the quadrilateral elements we employ selective reduced integration (SRI) on the shear components in order to reduce the well known problems this element exhibits when subjected to bending or volumetric deformation modes.

By expressing the unknown (vector-valued) function u_h in the basis given by ϕ_i with coefficients x_i we obtain the linear, symmetric and positive definite system of equations

$$A\, x = b \tag{2.7}$$

that will be solved by the multigrid method. It should be noted however, that the application of the multigrid method is not restricted to symmetric matrices which is important for the extension of this work to elastoplastic problems.

A detailed discussion and analysis of the variational problem of plane elasticity and it's discretization outlined here can be found in [5] and [6]

3. The adaptive solution algorithm

3.1. Grid refinement

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 [3]) and KASKADE (see [4]). However, we allow quadrilateral elements and more refinement rules (see Fig. 2).

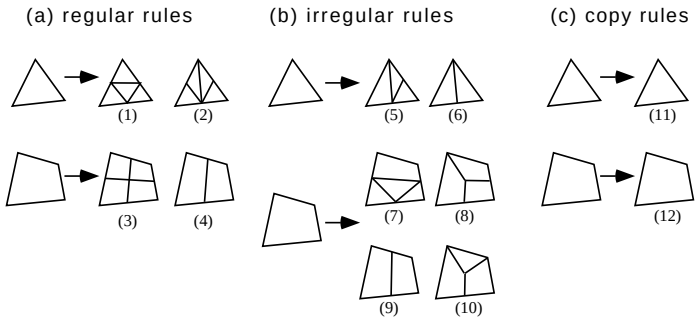

Figure 2. Complete set of refinement rules

The refinement algorithm is explained with the help of Fig. 3. The coarsest grid level T_0 is assumed to be generated by some mesh generator and all elements of this level 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. Note that the refinement is local in the sense that an element may not be refined at all. The copy elements are only needed for an efficient implementation of the local multigrid method and do not destroy the optimal complexity of the method.

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 irregular neighbour. In that case the irregular refinement is removed and replaced by a regular refinement rule. There exist other well known refinement strategies e.g. “hanging nodes” or transition elements. The subsequently explained load balancing strategies are applicable to any refinement method.

Let T_k denote the set of elements on level k and N_k the set of nodes on level k generated by the refinement algorithm described so far. Then the mesh $\mathcal{T}$ defined by

$$\mathcal{T} = \left\{ t \in \bigcup_{k=0}^{j} T_k \,\middle|\, t \text{ is not further refined} \right\} \tag{3.8}$$

is the mesh that defines the Finite-Element space for the discrete solution of our problem. The nodes of this mesh are given by

$$\mathcal{N} = \bigcup_{k=0}^{j} \{ n \in N_k | n \text{ did not exist in } N_{k-1} \} \tag{3.9}$$

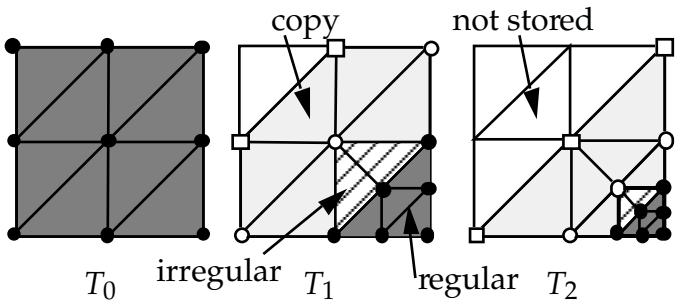

Figure 3. Nested local grid refinement

3.2. Local multigrid method

In order to solve the discrete problem

$$Ax = b \tag{3.10}$$

derived from a discretization on the mesh $\mathcal{T}$, a sequence of auxiliary problems

$$A_k x_k = b_k \tag{3.11}$$

is used. A_k corresponds to a discretization on the grid level T_k. In particular when we define

$$\begin{aligned} S_k &= \{n \in N_k | n \text{ is corner of a regular element in } T_k\}, \\ D_k &= \{n \in N_k | n \text{ is connected to a node } n' \in S_k\}, \end{aligned} \tag{3.12}$$

we need the stiffness matrix A_k and load vector b_k only at the nodes corresponding to the set D_k. The additional layer of copy elements in the mesh structure allows us to compute exactly this part of A_k.

The main conclusion of the papers [7] and [8] with respect to locally refined meshes was that it is sufficient to *smooth only the unknowns corresponding to the nodes in set S_k on level k* in order to achieve a convergence rate that is independent of the number of refinement steps and the space dimension for elliptic problems. The number of arithmetical operations for one iteration of the multigrid method can be shown to be proportional to the size of the set $\mathcal{N}$, i.e. the dimension of the system $Ax = b$.

ALGORITHM 3..1 One iteration of the multiplicative local multigrid method is given by algorithm `mlmg`. We only describe the V-cycle since the W-cycle is has no optimal complexity for arbitrary local refinement. Vectors x and b contain the current solution and the load vector. x and b live on the nodes $\mathcal{N}$.

`mlmg` (x, b) {
(1) $d_j = b - A_j x$; $v_j = 0$; on D_j

(2) for $(k = j; k > 0; k = k - 1)$ {
(3) $v_k = \mathcal{S}_k^{\nu_1}(v_k, d_k)$; on S_k
(4) $r_k = d_k - A_k v_k$; on D_k
(5) $d_{k-1} = \begin{cases} (I_{k-1}^k)^T r_k & \text{on } D_{k-1} \cap D_k \\ b - A_{k-1} x & \text{else} \end{cases}$
(6) $v_{k-1} = 0$; on N_k }
(7) $v_0 = A_0^{-1} d_0$; on N_0
(8) for $(k = 1; k \leq j; k = k + 1)$ {
(9) $x = x + v_{k-1}$; on $S_{k-1} \setminus S_k$
(10) $v_k = v_k + I_{k-1}^k v_{k-1}$; on N_k
(11) $v_k = \mathcal{S}_k^{\nu_2}(v_k, d_k)$; on S_k
}
(12) $x = x + v_j$; on S_j
}

The aim of algorithm `mlmg` is the computation of a correction v to the given iterate x by applying one V-cycle to the problem $Av = b - Ax$. The algorithm starts in line 1 with the computation of the defect on the finest level j (which does not necessarily cover the whole domain Ω !). The loop in line 2 goes from top to bottom and applies ν_1 pre-smoothing steps where only the values of v_k at the nodes corresponding to S_k are changed (line 3). This results in the change of the defect only at nodes in D_k (line 4). In order to obtain the defect on the next coarser level we restrict only to the nodes in D_{k-1} that are also in D_k on the fine mesh. For all other nodes the defect can be computed on the coarser grid level since the solution x has not been changed yet at those positions (line 5). Like in standard multigrid we start with a zero correction on the next coarser level in line 6. Line 7 contains the exact solve on the coarsest level. The loop in line 8 now realizes the upward part of the V-cycle. It starts with an update of the current iterate in line 9 with the correction at those nodes that are not changed any further on the higher levels of the grid. Note that each component of the global vector x is altered only once. Line 10 contains the interpolation step and line 11 the post-smoothing step. Finally line 12 updates the current iterate at the remaining nodes not yet corrected at the coarser levels.

As smoothers we use point-block variants of Jacobi, Gauß-Seidel or ILU$_\beta$ (see [9]) iterations, i.e whenever a division by the diagonal element occurs, we multiply with the inverse of the 2×2 block matrix corresponding to the two unknowns at a node.

3.3. Refinement indicator

The adaptive grid refinement algorithm requires an indicator which decides on whether a finite element is to be further refined or not. We choose a very simplistic based on the idea of nodal stress comparison.

With the linear isoparametric triangles and quadrilaterals employed in our calculations the stress components typically exhibit discontinuities across element borders. In our case an element will be refined when the relative change in the value of the Von-Mises-stress

$$\sigma_{VM} = \sqrt{\sigma_x^2 - \sigma_x \sigma_y + \sigma_y^2 + 3\tau^2} \tag{3.13}$$

in any node towards adjacent elements surpasses a given value. The Von-Mises-stress was chosen as it takes into account all three components of σ and as it is invariant of the

coordinate system in use. In order to obtain a relative value the $\Delta\sigma_{VM}$ are divided by the maximum main-stress – which is the largest eigenvalue of the stress tensor – found on the entire problem domain.

The above indicator was chosen for reasons of simplicity and ease of implementation. Our primary aim in this context is to demonstrate the usability and scalability of our data structures, numerical algorithms and general concepts in the context of a parallel adaptive application from structural mechanics. The indicator is based on practitioners experience alone and does not claim a solid theoretical basis, yet the numerical results were satisfactory. This is due to the fact that the refinement algorithm presented above strictly avoids badly shaped elements with large aspect ratios, large differences in size between neighbouring elements and other grid deficiencies that would undoubtly inhibit this indicator.

4. Parallelization Approach

4.1. Grid Partitioning

The parallelization of all components of the adaptive multigrid method is based on a distribution of the data onto the set of processors. In our method the elements are assigned uniquely to the processors. This results in horizontal and vertical overlap as shown in Fig. 4.

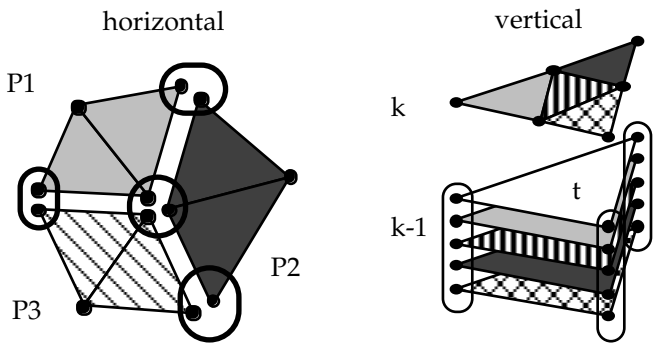

Figure 4. Horizontal and vertical overlap in data partitioning

The left part of the Fig. 4 shows the situation on one grid level (intra-grid). For instance 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. 4 shows the vertical or inter-grid situation. The white triangle t on level $k-1$ posesses four son triangles on level k each assigned to a different processor. Here the rule applies 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 $k-1$. Obviously the load balancer should avoid this situation whenever possible.

4.2. Assembling the stiffness matrix

In the Finite-Element method the sum of all element stiffness matrices yields the global stiffness matrix A_k on grid level k. In the parallel version the element stiffness matrices are only summed *per processor* to give a processor stiffness matrix A_k^p. The same is done with the right hand side b_k. Consequently we have the relations

$$A_k = \sum_{p \in P} A_k^p, \qquad b_k = \sum_{p \in P} b_k^p. \tag{4.14}$$

Note that A_k^p and b_k^p can be computed locally without any communication and that load balancing is optimal if each processor has the same number of elements. We say that vector b_k is stored *inconsistently* since at the nodes that are stored on more than one processor each processor knows only part of the global value. On the other hand if for some vector x_k we have $(x_k^p)_i = (x_k)_i$, i.e. each processor knows the global value also in the overlap nodes, we say that x_k is stored consistently. In order to transform an inconsistent vector into a consistent vector a communication operation over the interface nodes is required.

4.3. Restriction and Prolongation

The concept of inconsistent vectors is very useful for the parallelization of the grid transfer operators. The restriction operates on the defect $d_k = b_k - A_k x_k$. Now let us assume that A_k and b_k are stored inconsistently and x_k is stored consistently. Together with the linearity of all the operators involved we have

$$r_{k-1} d_k = r_{k-1} \left(\sum_{p \in P} b_k^p - \sum_{p \in P} A_k^p x_k \right) = \sum_{p \in P} r_{k-1} \underbrace{(b_k^p - A_k^p x_k)}_{d_k^p} = \sum_{p \in P} d_{k-1}^p, \tag{4.15}$$

which means that an inconsistent defect d_k^p on the fine grid is restricted to an inconsistent defect d_{k-1}^p on the coarse grid. This does not need any communication as long as the sons of an element e are mapped to the same processor as e. Under the same assumption a consistently stored correction v_{k-1} can be interpolated without any communication.

4.4. Smoother

For the smoother we either have the possibility of using Point-Jacobi or Block-Jacobi with inexact inner solvers. In the simple Point-Jacobi case we need $D_k = \mathrm{diag}(A_k)$ on each processor which requires one communication at the beginning of the solution cycle since A_k is stored inconsistently. But then we compute:

$$x_k^{n+1} = x_k^n + \omega D_k^{-1} \sum_{p \in P} d_k^p. \tag{4.16}$$

The sum corresponds to the fact that the inconsistent defect must be transformed into a consistent defect since only a consistent correction can be added to the consistently stored solution.

In the case of a Block-Jacobi smoother we assign also the nodes of the grid uniquely to the processors. For a node that is stored on more than one processor we define that the

processor with the smallest number is responsible for this node and all other processors compute a zero correction for it. Now the matrix D_k is a block diagonal matrix $D_k = \mathrm{diag}(M_{11}, \ldots, M_{PP})$ where M_{ii} corresponds to one step of Gauß-Seidel or ILU$_\beta$ for the unknowns assigned to processor i. Now the smoothing iteration is given by

$$x_k^{n+1} = x_k^n + \omega \sum_{p \in P} \left(D_k^{-1} \sum_{p \in P} d_k^p \right), \tag{4.17}$$

i.e. we need *two* communications over the interface nodes per smoothing step.

4.5. Refinement Algorithm

The most delicate part in the parallelization of the refinement algorithm is the "green" closure, where a conforming mesh has to be constructed from the refinement tags produced by the error indicator. Usually this step involves an iteration that can be implemented with optimal complexity on a serial machine. On the parallel machine this approach is not very useful, therefore we implemented a complete set of refinement rules (see Fig. 2) in the sense that for each possible pattern of refined edges (8 for triangles, 16 for quadrilaterals) there is a refinement rule that fits to the given edge pattern. With this approach an iteration is completely avoided and the conforming mesh can be computed with only one communication over the interface elements. All these additional elements are considered as irregular elements and cannot be refined further.

5. Load Balancing

5.1. Goals

The purpose of the load balancing algorithm is to assign the data to the processors in such a way that execution time is minimal. In this generality it is very hard to solve this problem (even approximately). Therefore we make the assumption that the computation time between any two synchronization points is reasonably large compared to communication time (coarse granularity). This means that in the first place one should assign the data in such a way that the computation time is equally balanced between the synchronization points and that minimizing the communication time is only the second goal.

The overall solution algorithm described so far consists of the different parts assembly, multigrid solver, error estimator and grid refinement. Fortunately it turns out that assigning an equal number of *elements* to each processor will balance the work load very well for all parts of the algorithm, especially in the element assembly phase which can be the most time consuming part in case of nonlinear problems.

The mapping of all other data objects, e.g. the nodes, is completely determined by the assignment of the elements. The most time consuming operation in the multigrid solver is a matrix–vector product. If the mesh consists of either triangles or quadrilaterals it can be shown that the time needed for a matrix vector product is proportional to the number of *elements* plus a term proportional to the number of boundary nodes. As long as the number of boundary nodes is negligible compared to the number of interior nodes the assignment of an equal number of elements to each processor will therefore also balance the work load in the multigrid solver.

The second goal of the load balancer is now to minimize communication requirements. The assembly part requires no communication at all therefore we concentrate on the communication in the multigrid solver. Since the processors will be synchronized at least once on each grid level in the smoother it is tempting to require that the elements of *each* grid level should be assigned to *all* processors in such a way that communication in the smoother is (approximately) minimized. The assignment of two consecutive grid levels should, however, be related in order to minimize also the communication in the grid transfers. In the case of local refinement even simple one-dimensional examples show that these two requirements are contradictory. Therefore the assignment must be done such that a compromise between low communication in the smoother and low communication in the grid transfers is achieved.

In the following we propose a clustering strategy based on the multigrid hierarchy that will achieve this compromise. The outline of the load balancing strategy is as follows:

(i) Combine elements into clusters using the multigrid hierarchy. This step can be done completely 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.

5.2. Clustering

For the clustering algorithm we require a tree-based (local) refinement strategy as was described in section 3.1. above. By $T_k = \{t_1^k, \ldots, t_{n_k}^k\}$ we denote all elements of level k and by $T = \cup_{k=0}^{j} T_k$ we denote the set of all elements. Since the refinement is based on subdividing individual elements we have the *father* relationship:

$$F_k \subseteq T_k \times T_{k+1}, \qquad (t, t') \in F_k \Leftrightarrow t' \text{ is generated by refinement of } t. \tag{5.18}$$

If j is the highest level then we set $F = \cup_{k<j} F_k$. Since the father of an element is unique we 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

$$S(t) = \{t' \in T | (t, t') \in F\}. \tag{5.19}$$

A simple recursive formula gives us the number of elements in the subtree that has a given element as its root:

$$z(t) = \begin{cases} 1 & S(t) = \emptyset \\ 1 + \sum\limits_{t' \in S(t)} z(t') & \text{else} \end{cases} \tag{5.20}$$

In order to quantify the communication cost (approximately) we also need a neighbour relation on the elements:

$$NB_k \subseteq T_k \times T_k, \qquad (t,t') \in NB_k \Leftrightarrow t' \text{ is neighbouring element of } t. \tag{5.21}$$

The union over all levels gives $NB = \cup_{k \le j} NB_k$.

Now we are in a position to define the clusters. In general a clustering is a partitioning of the set T, i.e.

$$C = \{c_1, \ldots, c_m\}, \qquad c_i \subseteq T \tag{5.22}$$

such that

$$\bigcup_{c \in C} c = T, \qquad c_i \cap c_j = \emptyset \Leftrightarrow i \neq j.$$

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

$$bot(c) = \min_k \{c \cap T_k \neq \emptyset\}, \qquad top(c) = \max_k \{c \cap T_k \neq \emptyset\}. \tag{5.23}$$

We also need the number of elements in a cluster on each level:

$$w_k(c) = |\{t \in T_k | c(t) = c\}|, \qquad w(c) = |\{t \in T | c(t) = c\}|, \tag{5.24}$$

where $|A|$ denotes the number of elements in set A. In the following we require 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 $c \in C$ we require: $((bot(c) < k \le top(c)) \wedge (t \in c \cap T_k)) \Rightarrow f(t) \in c$.

This definition ensures that the elements in a cluster form a *subtree* of the element tree structure. Therefore the relation F implies a relation F^C on the cluster set C by

$$((t,t') \in F \wedge c(t) \neq c(t')) \Rightarrow (c(t), c(t')) \in F^C. \tag{5.25}$$

The neighbour relation NB also implies a neighbour relation NB^C on the clusters via

$$((t,t') \in NB \wedge c(t) \neq c(t')) \Rightarrow (c(t), c(t')) \in NB^C. \tag{5.26}$$

The following algorithm constructs a clustering with the desired properties.

ALGORITHM 5..1 **Clustering of an element set.** The following algorithm `cluster` receives a multigrid hierarchy T (with highest level $j(T)$) as input and delivers a partitioning into clusters C as output. The parameters b, d, Z control the algorithm. b is the $baselevel$ since in practice we start partitioning on a level higher than zero if the coarse grids are very coarse or in the dynamic situation where we do 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 (C, T, b, d, Z) {
    C = ∅;
    for (k = b, ..., j(T))
        for (t ∈ T_k) {
            if ((z(t) ≥ Z) ∧ ((k − b) mod (d + 1) == 0)) {
                create new c; C = C ∪ {c};
                bot(c) = top(c) = k; root(c) = t;
                ∀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 our implementation the parallel grid refinement algorithm imposes an additional constraint that excludes some elements from becoming the root of a new cluster, since this constrained will be removed in a new version of our code we refer to [10] for details.

5.3. Balancing the Clusters

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

ALGORITHM 5..2 Algorithm `mg_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. Algorithm `mg_assign` uses another algorithm `assign` that solves the smaller assignment problems. `Assign` is a modification of standard graph partitioning algorithms and several variants will be discussed below.

```
mg_assign (C, P, j, b, M) {
    for (p ∈ P, k = b, ..., j) load[k, p] = 0;
(1) for (k = j, j − 1, ..., b) {
(2)     C_k = {c ∈ C | top(c) = k};
        if (C_k == ∅) continue;
(3)     l_k = Σ_{p∈P} load[k, p] + Σ_{c∈C} w_k(c);
(4)     Determine P' ⊆ P with |P'| ≤ max(1, l_k/M);
(5)     assign(k, C_k, P', load);
```

```
(6)             for (c ∈ C_k)
(7)                 for (i=bot(c),...,top(c)) load[i,m(c)] = load[i,m(c)] + w_i(c);
            }
        }
```

Algorithm `mg_assign` proceeds as follows. It uses a two-dimensional array $load[k,p]$ to store the number of level-k-elements that have been assigned to processor p. Then it proceeds from top to bottom (loop in line 1) and selects the clusters with the currently highest level that has 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 coarse grid agglomeration 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` gets also the array $load$ to take this into account. Finally lines 6 and 7 update the $load$ array.

We now give a generic version of algorithm `assign` that is used in algorithm `mg_assign` above. `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 5..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) {
(1)         if (P == {p}) {∀c ∈ C : set map(c) = p; return;}
(2)         Divide P into P_0, P_1;
(3)         for (i = 0, 1) l_i = Σ_{p∈P_i} load[k,p];
(4)         W = l_0 + l_1 + Σ_{c∈C} w_k(c);
(5)         Determine C_0, C_1 ⊆ C, C_0 ∪ C_1 = C, C_0 ∩ C_1 = ∅ such that
(6)             | (|P_0|/(|P_0|+|P_1|)) W − ( l_0 + Σ_{c∈C_0} w_k(c) ) | → min;
(7)         assign(k, C_0, P_0, load);
(8)         assign(k, C_1, P_1, load);
        }
```

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 elements on level k are equal in the corresponding 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 elements. Finally lines 7 and 8 contain the recursive calls that subdivide the new cluster sets again.

Several strategies are available to bisect the cluster sets in lines 6 and 7 of the algorithm above.

- **Orthogonal coordinate bisection.** In this variant each cluster is assigned a coordinate (x, y) by taking the center of mass of the root element of the cluster. Then the clusters are ordered by their x (or y) coordinate. A given position x_{cut} (respectively y_{cut} now defines the bisection $C_0 = \{c \in C | x(c) \leq x_{cut}\}$ and $C_1 = \{c \in C | x(c) > x_{cut}\}$ (or alternatively in y direction). The cut position x_{cut} is determined such that the expression in line 6 of algorithm `assign` is minimized. The bisection directions are chosen in an alternating fashion.
- **Inertial bisection.** This is a method similar to coordinate bisection but using a rotated coordinate system derived from the inertial moments. For details see [11] and [12].
- **Spectral bisection.** All subsequent methods do not require any coordinate information but use only graph connectivity information. As a graph we consider $G_k = (C_k, E_k), E_k = (C_k \times C_k) \cap NB^C$ where C_k is the cluster subset constructed in algorithm `mg_assign` and NB^C is the cluster neighbourship relation defined in Eq. (5.26). Note that the neighbourship relation is restricted to the subsets C_k. The connectivity between the sets C_i and C_k, $i \neq k$ is not considered.
  The spectral bisection method then derives the so-called *Laplace-matrix* L_k from the graph G_k as follows:

$$(L_k)_{i,j} = \begin{cases} \text{degree}(c_i) & i = j \\ -1 & (c_i, c_j) \in E_k \\ 0 & \text{else} \end{cases} . \tag{5.27}$$

  Matrix L_k is symmetric and positive semi-definite. Now each cluster $c_i \in C_k$, $1 \leq i \leq m_k$ is assigned a number x_i and the x_i define the vector $x = (x_1, \ldots, x_{m_k})^T$. According to [13] the graph bisection problem can be formulated as:

$$\begin{array}{ll} \text{Minimize} & \frac{1}{4} x^T L_k x \\ \text{under the constraint} & \sum_{i=1}^{m_k} x_i = 0, \qquad x_i \in \{-1, 1\}. \end{array} \tag{5.28}$$

  This optimization problem is now solved with $x \in \mathbf{R}^{m_k}$ instead of the constrained. If the graph G_k is connected the continuous optimization problem has the solution $x_{opt} = \sqrt{m_k} e_2$ where e_2 is the eigenvector corresponding to the second smallest eigenvalue of L_k. The components of x_{opt} are now used to define the bisection by setting $C_0 = \{c_i \in C | (x_{opt})_i \leq x_{cut}\}$ and $C_1 = \{c_i \in C | (x_{opt})_i > x_{cut}\}$ for a given x_{cut}. The position x_{cut} is chosen in order to minimize the expression in line 6 of algorithm `assign`.
- **Kernighan-Lin bisection.** In this method one starts with a random bisection into C_0 and C_1 that minimizes the expression in line 6 of algorithm `assign`. Then subsets of C_0 and C_1 are swapped repeatedly until a local minimum of the number of edges connecting C_0 and C_1 is found. Note that each swapping step does not change the load balance. By using the output of any of the other partitioning schemes as a starting partitioning instead of the random one an improved method can often been obtained. For details we refer to [14], [12], [10].
- **Multi-level bisection.** This method tries to apply ideas from multigrid to the solution of the graph bisection problem. This method is especially useful if the graph to be partitioned is very large. In the so-called coarsening phase neighbouring nodes of the

given graph are assembled into clusters (since our nodes are already clusters we have clusters of clusters now). These clusters together with edges defined in the canonical way form a new coarser graph which is coarsened repeatedly until a given minimal size is reached. For the coarsest graph a high quality bisection is determined using e.g. spectral bisection. Then the result interpolated to the next finer graph in the canonical way. On the finer levels the Kernighan-Lin heuristic is used to improve the interpolated coarse grid solution. This process is repeated until the finest level is reached. For details we refer to [12], [15].

All methods except coordinate bisection have been implemented in the load balancing software CHACO by HENDRICKSON and LELAND, see [12]. CHACO has been adapted to our code so that it can be used as algorithm `assign` above, for details see [16].

The load balancing scheme discussed in this section has been designed for the standard multiplicative multigrid algorithm. For the additive variants, like BPX (see [7]), load balancing can be done differently since the synchronization behavior is different. For a solution of the load balancing problem for additive multigrid (including local refinement) we refer to [10]. Finally we remark that the load balancing scheme can be extended immediately to the three-dimensional situation.

5.4. A simple parallel model

In this section we derive a simple model for parallel efficiency that shows the influence of the different load balancing schemes. Suppose we have a load balancing scheme A and that parallel efficiency is determined by the following formula:

$$E_A(P) = \frac{T_1}{\left(T_S + \frac{T_1 - T_S}{P} + X_A\right) P}, \tag{5.29}$$

where T_1 is the time needed by one processor, T_S is the serial part of the algorithm, the rest $T_1 - T_S$ is perfectly parallelizable and X_A are the communication costs. In more detail X_A is only that part of communication cost that can be influenced by the load balancing schemes, e.g. message setup-time would not be part of X_A since the number of messages is almost not influenced by the choice of the load balancing scheme. Then we want to see how the efficiency is influenced when the interface length is reduced by a better load balancing scheme B. The efficiency for load balancing scheme B is modeled by

$$E_B(P) = \frac{T_1}{\left(T_S + \frac{T_1 - T_S}{P} + \eta X_A\right) P}, \tag{5.30}$$

where η is a factor that describes the improvement in the communication cost. Typically the values of η are in the range of $\frac{1}{3}$ to $\frac{3}{4}$.

After some algebraic manipulations we obtain the following formula that expresses the (better) efficiency E_B in terms of the (worse) efficiency E_A:

$$E_B(P) = \frac{E_A(P)}{1 - (1-\eta)(1 - E_A(P)) \left(\frac{1}{1+\mu\frac{P-1}{P}}\right)}, \tag{5.31}$$

where μ is the cost of the serial part relative to the communication cost:

$$\mu = \frac{T_S}{X_A}. \tag{5.32}$$

Formula (5.31) says how much the efficiency of load balancing scheme A is improved by shortening the interface length of the partitions. The improvement however depends also on E_A and the factor μ. E.g. if E_A is already close to one, then the variation of η has not much influence. The same is true when the serial part is not negligible, i.e. when $\mu > 1$. Figure 5 shows an example with η set to 0.5 and μ set to 0, 0.5, 1.0, and 2.0. It depicts the dependency of the efficiency of the better load-balancing scheme B on the original efficiency of the worse scheme A. In our case where the efficiency of scheme A is already very good (about 0.9) the efficiency of B differs only slightly.

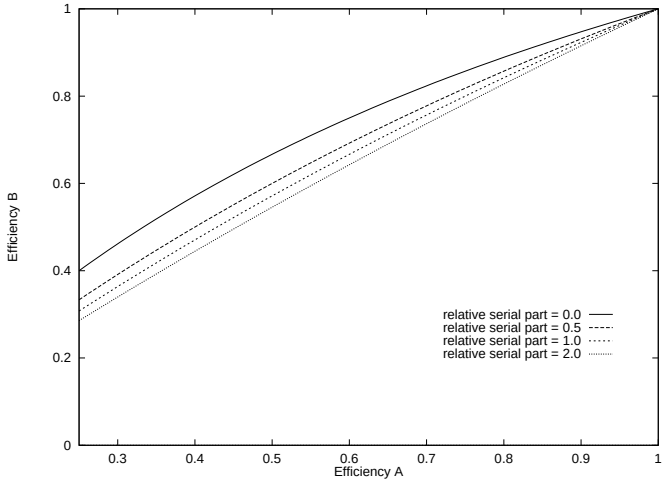

Figure 5. Dependency between efficiency A and B

6. Numerical Results

6.1. Uniformly refined test case

The uniformly refined test case is being evaluated with respect to the influence of the load-balancing scheme on the efficiency of our solution procedure. Therefore the equations of linear elasticity with parameters given in section 2 are solved in a domain given by Fig. 6a which is nonsymmetric and not single-connected. By chosing such a domain which resembles a seal ring and which has mostly curved boundaries we tried to eliminate initial advantages of specific load balancing schemes. Dirichlet boundary conditions have been used on the inner side of the ring and von Neumann conditions on the outer side. The initial mesh consisting of 24 quadrilateral elements is shown in Fig. 6b. Part c depicts the first refinement stage with 96 elements.

6.1.1. Fixed-size problem In this subsection we examine a problem with fixed size, i.e. the mesh from Fig. 6b has been refined uniformly 5 times, resulting in a hierarchy with

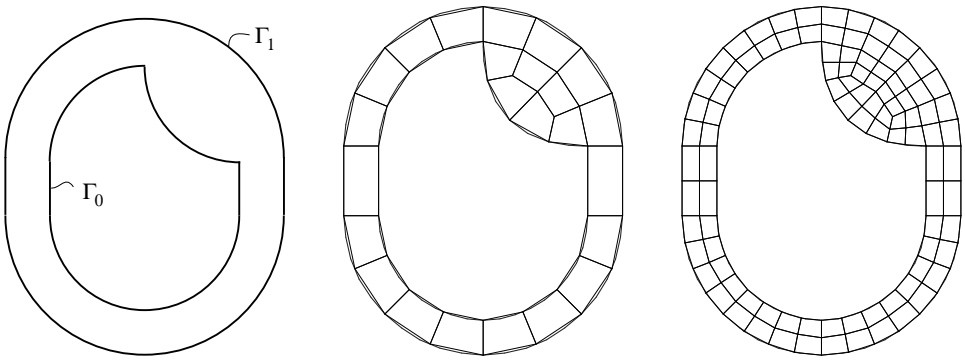

Figure 6. a) problem domain b) grid 0 c) grid 1

6 levels and 24576 elements on the finest level. This problem is then solved on 1,4,16 and 64 processors of the CRAY T3D. The parameters of the multigrid method were: V-cycle, 1 pre and post-smoothing steps with Block-Jacobi-Smoother and ILU$_{\beta=0.6}$ as inexact inner solver. The parameters for the clustering algorithm `cluster` are set to $b = 2$, $d = 1$ and $Z = 1$, i.e. levels 0 and 1 are treated on processor 0.

Table 1 shows the results for various mapping schemes within algorithm `assign`. The various methods were: rcb (recursive coordinate bisection), rib (recursive inertial bisection), ribkl (recursive inertial bisection with Kernighan-Lin optimization), rsbkl (recursive spectral bisection with Kernighan-Lin optimization), mk50 (multi-level bisection with Kernighan-Lin optimization and a coarse graph size of 50 nodes), mk200 (like mk50, but coarse graph size was 200 nodes). All methods except rcb were taken from the CHACO library. The column labeled E_{It} contains the efficiency for *one* multigrid iteration defined by $E_{IT}(P) = T_{IT}(1)/(T_{IT}(P)P)$, where $T_{IT}(P)$ is the time per iteration on P processors. The column labeled E_{Sol} gives the efficiency in the multigrid solver for obtaining a reduction of 10^{-6} in the euclidean norm of the residual, i.e. this number includes also the convergence rate of the method (numerical efficiency). The column labeled E_{Max} shows an upper bound for E_{It} computed as: Total number of nodes on all levels in serial calculation divided by maximum number of nodes on all levels in one processor in the parallel calculation. The number E_{Max} therefore accounts for load imbalance and overlap but *not* for communication cost and idle times. The column labeled Min/Max IF shows the minimum and maximum number of overlap nodes per partition on the finest level. The column labeled Min/Max Prod is similar to Min/Max IF but each individual overlap node is weighted with the distance to the destination processor.

- Table 1 shows that the efficiency per iteration, E_{It}, does not vary much with the different load balancing schemes. We explain this with the help of formula (5.31). In the case of very many unknowns per processor ($P = 4, 16$), the efficiency E_{rcb} obtained with simple recursive coordinate bisection is already very good and cannot be improved much by reducing interface length. E.g. if E_A in formula (5.31) is already 0.9 and the interface length is halved ($\eta = 0.5$) the efficiency E_B will be 0.947 according to this simple model. In the case of very few unknowns per processor ($P = 64$) the serial part of the algorithm due to the coarse grids is not negligible compared to communication

cost, i.e. $\mu > 1$ weakens the influence of η in formula (5.31).

- A detailed look at table 1 reveals practically no correlation of interface length and measured parallel efficiency E_{It}. E.g. for $P = 4$ ribkl has much smaller interface length than rib alone and also E_{Max} is better for ribkl, but measured efficiency E_{It} is better for the rib method. An additional view on mapping data reveals that all schemes showing an efficiency of approx. 92% in the case of $P = 4$ kept the maximum workload on each level on the same processor. Thus idle times were hidden by computation. This is not the case when the maximum workload is mapped to different processors on different levels. In the latter cases only 87% efficiency are achieved. Furthermore cache usage may be influenced by the shape of the partitions as shown in [10].
- More important than the efficiency of a single iteration step is the performance of the complete solution process in order to obtain a fixed accuracy. Figure 7 shows the result of three different load balancing schemes: rcb (a) takes 11 iterations to complete, rib (b) shows the best parallel efficiency but needs the largest number of iterations, therefore it achieves the worst numerical efficiency E_{Sol}. The best numerical efficiency is measured with rsbkl (c) which needs the least number of iterations. In general the more expensive load balancing algorithms yield a lower iteration count. The variation of the number of iterations is due to the block Jacobi smoothing where the number, length and position of the partition interfaces possesses significant influence.

Table 1. Results for uniformly refined, fixed size test case.

P	LB	# It.	E_{It}	E_{Sol}	E_{Max}	Min/Max IF	Min/Max Prod
1	–	7	–	–	–	–	–
4	rcb	11	91.9	58.5	96.6	76 / 266	76 / 266
4	rib	9	91.9	71.5	97.7	84 / 255	89 / 260
4	ribkl	8	86.7	75.9	98.2	66 / 140	66 / 140
4	rsbkl	8	88.3	77.3	98.4	66 / 158	66 / 158
4	mk50	9	92.3	71.8	96.5	76 / 266	76 / 266
4	mk200	8	87.0	76.1	97.0	66 / 134	66 / 134
16	rcb	11	81.6	51.9	88.6	72 / 221	72 / 299
16	rib	13	82.7	44.6	89.2	70 / 212	70 / 323
16	ribkl	9	76.4	59.4	89.9	66 / 164	99 / 362
16	rsbkl	9	78.1	60.7	89.5	66 / 160	66 / 294
16	mk50	10	75.1	52.6	88.8	72 / 221	72 / 299
16	mk200	10	78.8	55.1	89.4	66 / 161	66 / 304
64	rcb	13	57.9	31.2	67.6	62 / 179	63 / 245
64	rib	13	57.6	31.0	69.4	62 / 119	69 / 315
64	ribkl	11	55.9	35.6	68.9	59 / 94	65 / 282
64	rsbkl	11	56.0	35.6	69.1	60 / 116	65 / 390
64	mk50	12	54.0	31.5	67.8	62 / 179	63 / 245
64	mk200	12	54.4	31.7	69.4	60 / 96	63 / 384

6.1.2. Scaled-size problem In this section the same problem as in the previous section is solved but the problem size *per processor* remains constant. This means that with a fourfold increase in the number of processors the multigrid hierarchy is extended by one level. Each processor is assigned 24576 elements on the finest level leading to a total of 6291456 elements and 12601344 degrees of freedom on the finest level in the case of 256

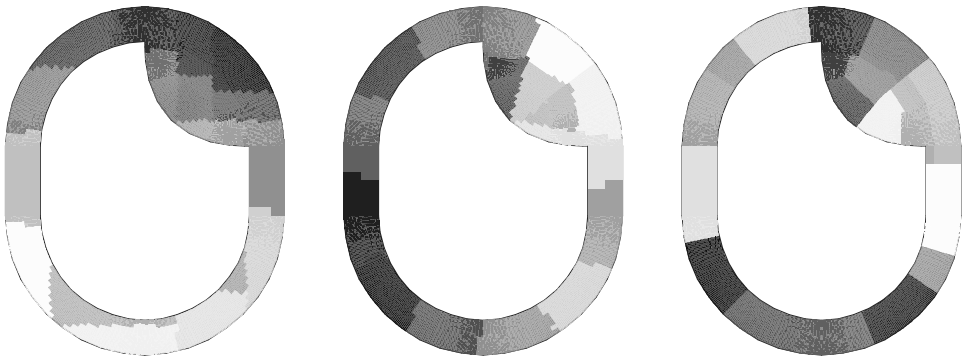

Figure 7. a) rcb b) rib c) rsbkl

processors. The clustering parameters are set to $b = 3$, $d = \infty$ and $Z = 1$, i.e. levels 0, 1 and 2 are always treated on one processor whereas the remaining levels are divided into 1536 clusters distributed among all processors. Experiments with a progressively smaller number of processors for the coarse grids showed no improvement in the parallel efficiency.

Table 2. Results for uniformly refined, scaled size test case

P	LB	# It.	E_{It}	E_{Sol}	E_{Max}	Min/Max IF	Min/Max Prod
1	–	7	–	–	–	–	–
4	rcb	9	89.6	69.7	97.7	146 / 538	146 / 538
4	rib	9	89.9	69.9	98.2	146 / 498	146 / 498
4	ribkl	8	92.2	80.6	98.6	130 / 266	130 / 266
4	rsbkl	8	92.3	80.9	97.7	130 / 290	195 / 355
16	rcb	11	87.7	55.9	96.9	290 / 869	290 / 1177
16	rib	11	87.2	55.6	97.2	290 / 820	290 / 1239
16	ribkl	9	88.6	69.0	97.2	258 / 692	378 / 1418
16	rsbkl	10	88.1	61.7	97.2	258 / 629	258 / 1161
64	rcb	12	85.4	49.8	96.0	452 / 1286	483 / 2025
64	rib	11	86.0	54.8	96.5	484 / 903	485 / 2349
64	ribkl	10	86.0	60.2	96.5	451 / 774	455 / 2131
64	rsbkl	10	86.0	60.2	96.7	451 / 969	453 / 3555
256	rcb	12	84.5	49.3	96.1	451 / 1158	451 / 2326
256	rib	12	84.4	49.3	96.5	451 / 1033	451 / 6446
256	ribkl	13	84.8	45.7	96.4	451 / 903	453 / 7058
256	rsbkl	12	84.9	49.5	96.5	451 / 903	455 / 2650

Table 2 shows again E_{It} and E_{Sol} as the most important results. Scaled efficiency per multigrid iteration reaches 85% on 256 processors due to the fast communication of the Cray T3D and low surface to volume ratio. For the smaller processor numbers a good correlation of interface length and efficiency per iteration is visible. For increasing number of processors the variations in interface length become smaller since the number of clusters per processor decreases. In the 256 processor case only 6 clusters are assigned to one processor. Using a smaller cluster depth would lead to an increasing number of clusters

yielding better partitioning on the one hand but additional inter-grid communication on the other hand. In practice the best efficiency has been obtained with the above-mentioned parameters.

6.2. Adaptively refined test case

In order to investigate the parallel performance in the case of adaptive local grid refinement the problem in Fig. 8 has been solved. The domain with 16 reentrant corners was chosen not to test the efficiency of the refinement indicator but the efficiency of the parallel multigrid solver in a hierarchically refined set of grids. Dirichlet boundary conditions have been applied at Γ_0 and Neumann boundary conditions at Γ_1. The refinement is concentrated at the reentrant corners. A locally refined mesh is shown in Fig. 9. Again a fixed-size and a scaled size computation are presented.

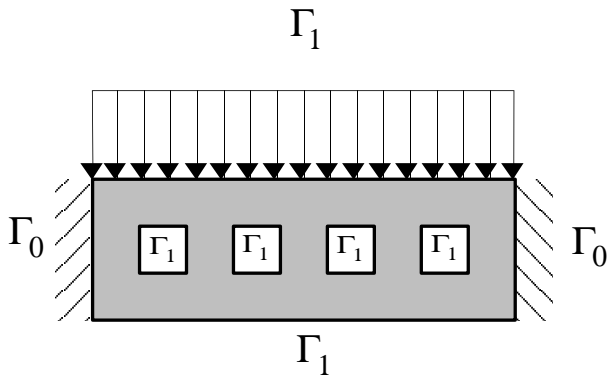

Figure 8. Problem setup for the adaptively refined example.

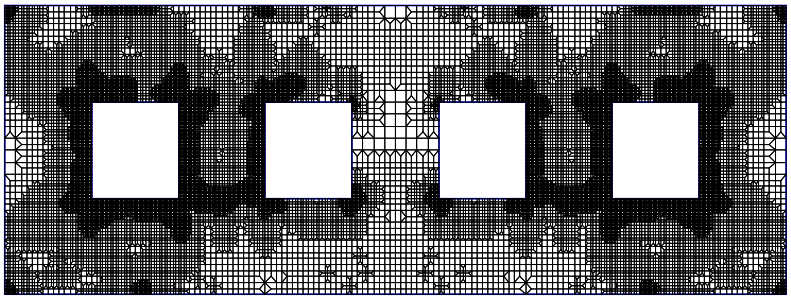

Figure 9. Adaptively refined mesh.

6.2.3. Fixed-size problem The mesh for the fixed-size calculation contained 23314 nodes (46628 degrees of freedom) and 7 grid levels. Table 3 shows the results for par-

allel and numerical efficiency. In the 64 processor calculation each computing node was assigned only about 200 elements on the finest level. The decrease in parallel efficiency compared to the uniformly refined case is due to the following reasons:

- In contrast to the uniformly refined case we have no geometric growth in the number of nodes per level. The number of nodes on level 3,4,5 and 6 were 3282, 8523, 18444 and 17249 in this example. This leads to a worse calculation to communication ratio.
- Algorithm `mg_assign` requires that clusters with different top level are balanced separately (cluster sets C_k). This leads to elements on the same grid level being assigned to processors in several independent steps which leads in turn to higher communication overhead and idle times.

Table 3. Parallel adaptive problem

P	LB	# It.	E_{It}	E_{Sol}	E_{Max}	Min/Max HIF	Min/Max Prod
1	–	16	–	–	–	–	–
4	rcb	16	88.6	88.6	97.4	25 / 28	25 / 28
4	rib	18	86.2	76.5	96.2	46 / 46	46 / 46
4	ribkl	18	79.7	70.8	96.2	20 / 20	20 / 20
4	rsbkl	17	76.9	72.3	91.5	22 / 50	22 / 70
16	rcb	20	58.0	46.4	85.5	63 / 131	63 / 140
16	rib	19	55.8	47.0	85.0	13 / 56	13 / 104
16	ribkl	19	53.6	45.1	82.9	6 / 47	6 / 91
16	rsbkl	19	54.5	45.9	81.2	8 / 60	8 / 161
64	rcb	22	24.7	18.0	65.8	30 / 111	32 / 138
64	rib	20	24.1	19.3	61.9	22 / 84	22 / 184
64	ribkl	20	24.5	19.6	63.2	8 / 69	8 / 155
64	rsbkl	20	21.6	17.3	53.3	16 / 81	16 / 238

6.2.4. Scaled-size problem The same problem as in section 6.2.3. is solved but now the number of nodes per processor was held approximately constant by varying the tolerance in the error indicator. Since the efficiency results did not depend much on the load balancing scheme, only rcb was used for this experiment.

Table 4 shows E_{It} and E_{Sol} on up to 64 processors. In order to judge the cost of the load balancing procedure the execution times for the mapping (T_{LB}), for the load migration (T_{MIG}) in the last balancing step and the multigrid solution time (T_{Sol}) have been included in Table 4.

Table 4. Scaled parallel adaptive problem

P	Unknowns	# It.	T_{Sol}	T_{LB}	T_{MIG}	E_{It}	E_{Sol}	E_{Max}
1	23,314	16	49.9	–	–	–	–	–
4	96,538	16	51.2	0.66	1.15	92.4	92.4	97.6
16	299,587	18	49.1	3.57	4.52	80.7	71.7	93.6
64	691,309	19	29.5	3.36	2.50	65.4	55.1	85.4

7. Conclusions

In this paper we showed that adaptive multigrid methods can be effectively implemented on modern parallel computer architectures and applied to linear elasticity calculations. The efficiencies per multigrid iteration reached 85% on 256 processors of the CRAY T3D for a uniformly refined test case and 65% on 64 processors for an adaptively refined test case.

The problem of dynamic load balancing has been discussed in detail and a central scheme with a clustering strategy based on the mesh hierarchy has been proposed. Within this algorithm standard graph partitioning algorithms are used to partition special subsets of the clusters. A public domain library (CHACO) has been adapted to our code in order to compare different partitioning schemes. It has been found that parallel efficiency is only slightly influenced by the different schemes because of the following reasons:

- For large problems computation time is dominating completely, for small problems the coarse grids in the multigrid process are a non-negligible serial part. In both cases the influence of partition interface length on efficiency is only weak.
- Mapping the maximum workload onto different processors on different levels prohibits that some of the communication time can be hidden behind computation time.
- Cache hit rate and therefore effective computation speed may depend on the shape of the partitions.

On the other hand it has been found that the number of mg iterations to reach a certain accuracy may vary greatly with the different load balancing schemes. Obviously the quality of the block Jacobi smoother depends on the number, shape and position of the partitions, especially if the elements not isotropic.

The investigation shows that many competing effects are influencing the observed *numerical* efficiency and it is felt that partition interface length, the standard measure for load balancing algorithms, is not the most important one. Most of the effects are problem and/or machine dependent and therefore difficult to consider in an improved balancing scheme.

Further work will include the extension to nonlinear material laws and the construction of new load balancing schemes that also take the synchronization between multigrid levels and load migration time into account.

Acknowledgments

The authors would like to thank the Konrad-Zuse-Center Berlin (ZIB) for the possibility of using their 256 processor CRAY T3D. Also the use of the CHACO load balancing library is greatly acknowledged.

REFERENCES

1. K. Eriksson, D. Estep, P. Hansbo, and C. Johnson, 'Introduction to adaptive methods for differential equations', *Acta Numerica*, (1995).
2. H. Yserentant, 'Old and new convergence proofs for multigrid methods', *Acta Numerica*, (1993).
3. R. Bank, *PLTMG Users Guide Version 6.0*, SIAM, Philadelphia, 1990.
4. P. Deuflhard, P. Leinen, and H. Yserentant, 'Concepts of an adaptive hierarchical finite element code', *IMPACT of Computing in Science and Engineering*, **1**, 3–35, (1989).

5. D. Braess, *Finite Elemente*, Springer, 1991.
6. T. J. R. Hughes, *The Finite Element Method*, Prentice Hall, 1987.
7. J. H. Bramble, J. E. Pasciak, J. Wang, and J. Xu, 'Parallel multilevel preconditioners', *Math. Comp.*, **55**, 1–22, (1990).
8. H. Yserentant, 'Two preconditioners based on multi-level splitting of finite element spaces', *Numer. Math.*, **58**, 163–184, (1990).
9. G. Wittum, 'On the robustness of ilu smoothing', *SIAM J. Sci. Statist. Comput.*, **10**, 699–717, (1989).
10. P. Bastian, *Parallele adaptive Mehrgitterverfahren*, Teubner Skripten zur Numerik, Teubner-Verlag, 1996.
11. B. Nour-Omid, A. Raefsky, and G. Lyzenga, 'Solving finite-element equations on concurrent computers', in *Parallel computations and their impact on mechanics*, ed., A. K. Noor, 209–227, American Soc. Mech. Eng., New York, (1986).
12. Hendrickson and R. Leland, 'The chaco user's guide version 1.0', Technical Report SAND93-2339, Sandia National Laboratory, (October 1993).
13. B. Hendrickson and R. Leland, 'Multidimensional spectral load balancing', Technical Report SAND93-0074, Sandia National Laboratory, (January 1993).
14. B. W. Kernighan and S. Lin, 'An efficient heuristic procedure for partitioning graphs', *The Bell System Technical Journal*, **49**, 291–307, (1970).
15. G. Karypis and 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).
16. S. Lang, *Lastverteilung für paralle adaptive Mehrgitterberechnungen*, Master's thesis, Universität Erlangen-Nürnberg, IMMD III, 1994.