DYNAMIC LOAD BALANCING FOR PARALLEL ADAPTIVE MULTIGRID METHODS ON UNSTRUCTURED MESHES

P. Bastian
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Universität Heidelberg, Im Neuenheimer Feld 368, 69120 Heidelberg, Federal Republic of Germany,
bastian@iwr.uni-heidelberg.de

Abstract

In this paper we consider parallel adaptive multigrid methods on unstructured meshes using MIMD computers with distributed memory. We present two load balancing algorithms designed for additive (BPX) and multiplicative multigrid. Both methods will be compared in terms of parallel and numerical efficiency and we also compare uniform with adaptive computation including the overhead introduced by load balancing and load migration.

1 Introduction

We consider solution adaptive strategies on parallel computers with distributed memory in order to solve linear elliptic boundary value problems of the form

$$\begin{aligned} \operatorname{div}(r(x,y)u - \varepsilon(x,y)\nabla u) &= f(x,y) \text{ in } \Omega \subset \mathbf{R}^2 && (1)\\ u &= g(x,y) \text{ on } \partial\Omega\ . && (2) \end{aligned}$$

Our adaptive strategy consists of the following steps:

1. Start with an intentionally coarse grid T_0, since the grid is coarse this can be done on a single processor. Set $j = 0$.
2. Discretize the b.v.p. on all grid levels $T_0, \ldots, T_j$, yielding systems of linear equations $A_k x_k = b_k$. The discretization is of finite volume type, based on the ideas of [12].
3. Solve the discrete equations with a multigrid procedure. Here we have the choice of an additive multigrid method (also called BPX method) or the usual multiplicative multigrid scheme. Both methods are of optimal complexity for arbitrary local grid refinement but differ in communication granularity and the way load balancing can be done. For details of the methods see [15],[4],[5],[6],[1],[2].
4. Estimate the discretization error. Here we employ either a residual based error estimator for the diffusion dominated case or the gradient of u scaled with local mesh size for the convection dominated case (see [13]). If the error criterion is met then STOP else flag elements for further refinement.

5. Determine a new mapping m of elements $T_0, \ldots, T_j$ to Processors P that will balance the load in each processor *after* grid refinement.
6. Migrate the elements to their new destination.
7. Refine the grid structure. Note that grid levels T_k, $k > 0$ do in general not cover the whole domain Ω, therefore more than one grid level may change during refinement. The result is a new grid hierarchy $T_0, T_1', \ldots, T_{j'}'$ with $j' \in \{j, j+1\}$. Set $j = j'$.
8. Goto step 2.

This concept is well suited for parallel computation since it avoids also serial bottlenecks in grid generation and post-processing. All steps of the method, except part of step 5, are carried out fully in parallel. Parallel graphical postprocessing (e.g. contour lines) has also been implemented. Note that the order of operations is important: Load balancing is done *before* grid refinement in order to move less data in step 6. The main objectives of the paper are the following:

- Presentation of two load balancing algorithms for additive and multiplicative multigrid.
- Evaluation of *parallel* and *numerical* efficiencies of the multigrid methods. Especially we answer the question whether the superior parallelizability of the additive method can outweigh the faster convergence of the multiplicative method.
- Evaluation of total efficiency for the fully adaptive procedure. This is the ultimate test since steps 5 and 6 of theadaptive procedure above must be considered as overhead compared to the single processor case.

2 Load Balancing

2.1 Single Grid Case

Let us first consider the case of a single grid level T. If there are no data dependencies between the grid points (e.g. as in defect computation or jacobi smoothing) the load balancing problem is to assign each $t \in T$ to a processor $m(t) \in P$ such that

1. Each processor has the same number of elements.
2. "Communication cost" $\sum_{t,t' \text{ are neighbors}} (1 - \delta_{m(t),m(t')})$ should be minimal.

This problem is known in various formulations as the *graph mapping problem* and many heuristc procedures have been developed for the approximate solution of this NP-hard problem. We want to mention variants of the recursive bisection method [9],[11], [14],[8] and the Kernighan-Lin algorithm ([7], [10]).

The situation gets more difficult when we consider multigrid methods on a hierarchy of grids $T_0, \ldots, T_j$. In addition to the communication in the smoother (intra grid communication) we have also communication in the grid transfer when the father of an element is not mapped to the same processor (inter grid communication). Secondly we cannot ignore the data dependencies between the grid levels (also in additive multigrid !). Especially the latter point changes the type of the problem completely: Instead of a graph mapping problem we now have to solve a *scheduling problem*. The first step towards the solution of these problems is the introduction of a clustering strategy.

2.2 Clustering

Let a grid hierarchy $\mathcal{T} = T_0 \cup \ldots \cup T_j$ of depth $j+1$ be given. A cluster c is simply a subset of $\mathcal{T}$: $c \subseteq \mathcal{T}$. A clustering $C = \{c_1, \ldots, c_N\}$ is a decomposition of $\mathcal{T}$ into disjoint subsets:

$$\bigcup_{c \in C} c = \mathcal{T}, \quad c_i \cap c_j = \emptyset \ \forall i \neq j \ .$$

Instead of computing a destination $m(t)$ for each individual element $t \in \mathcal{T}$ we will compute a mapping of clusters to processors. $m : C \rightarrow P$. If t has been assigned to cluster $c(t)$ then t will be mapped to $m(t) = m(c(t))$. This has a number of advantages:

- The complexity of the load balancing problem is reduced. The clustering process itself will be cheap and highly parallelizable. If the number of clusters is large compared to the number of processors then the error in load balance will be negligible.
- Memory requirements for load balancing are reduced. This is a prerequisite in order to employ a central load balancing strategy (one processor would not be able to store information about each individual element).
- The construction of the clusters via the element hierarchy will result in a tradeoff between inter and intra grid communication.

Before explaining the clustering process we introduce some notation. For $c \in C$ define T_k^c as the level k elements mapped to cluster c:

$$T_k^c = \{t \in T_k | c(t) = c\}$$

and the functions

$$\begin{aligned} \text{bottom}(c) &= \min_k (T_k^c \neq \emptyset) \\ \text{top}(c) &= \max_k (T_k^c \neq \emptyset) \\ w_k(c) &= |T_k^c| \\ w(c) &= \sum_{k=0}^{j} w_k(c) \end{aligned}$$

For the individual element $t \in T_k$ itself we denote by $s(t) \subseteq T_{k+1} (k < j)$ the successors of t on the finer grid level and by $f(t) \in T_{k-1} (k > 0)$ the predecessor of t on the coarser level. The subtree $S(t) \subseteq \mathcal{T}$ defined by an element t is given by

$$S(t) = t \cup \{S(t') | t' \in s(t)\}$$

and $z(t) = |S(t)|$ is the size of this subtree. Subsequently we will use only clusters with the following properties:

(i) For $i = \text{bottom}(c)$ we have $w_i(c) = 1$. The element $T_i^c = \{t\}$ is called the *root element* of the cluster and can be obtained by $t = \text{root}(c)$.

(ii) $\forall k > i, t \in T_k^c : f(t) \in T_{k-1}^c$. This conditions ensures that all elements in a cluster form a tree (a subtree of the element tree).

The idea of the following clustering algorithm ist to start at some given grid level assigning each element to a seperate cluster. Then the successors of an element are assigned to the same cluster until a given depth is reached.

Algorithm 1 Clustering. The basic clustering algorithm is controlled by three parameters b, d and Z. The *baselevel* b determines the level where the clustering process starts. If the coarsest grid T_0 has very few elements we will refine it uniformely until T_b which is supposed to contain more than $Const|P|$ elements with $Const$ depending on the hardware used. All elements below level b are then only load balanced once at the beginning. The depth d is the maximum depth of a cluster, i.e. $\text{top}(c) - \text{bottom}(c) \leq d$ and Z is the minimal size of a cluster, i.e. $w(c) \geq Z$.

```
      cluster (C, T, j, b, d, Z) {
(1)       C = ∅;
(2)       for (k = b, ..., j)
(3)           for (t ∈ T_k) {
(4)               if ((z(t) ≥ Z) ∧ ((k − b) mod (d + 1) = 0)) ∨ (k = b) {
(5)                   create new c; C = C ∪ {c};
(6)                   bottom(c) = top(c) = k; root(c) = t;
(7)                   ∀i : w_i(c) = 0; w(c) = 0;
(8)               } else c = c(f(t));
(9)               c(t) = c; top(c) = max(top(c), k);
(10)              w_k(c) = w_k(c) + 1; w(c) = w(c) + 1;
              }
      }
```

Once the clusters have been constructed we can derive a clustering of finer granularity by assigning the subtrees defined by the sons of the root element to seperate clusters.

Algorithm 2 Splitting of clusters. The following algorithm splits a cluster set C into smaller clusters. The result is placed in two cluster sets L' and C'. L' are clusters that can not be subdived further without violating the minimum size condition $w(c) \geq Z$. C' are clusters that may be subdivided further.

```
      split(C, C', L', Z) {
(1)       C' = L' = ∅;
(2)       for each (c ∈ C) {
(3)           r = root(c);
(4)           if (s(r) ≠ ∅) ∧ (∀s ∈ s(r) : |S(s) ∩ c| ≥ Z) {
(5)               Assume s(r) = {s_1, s_2, ..., s_n};
(6)               create new c_1; c_1 = c; root(c_1) = s_1;
(7)               for (i = 2, ..., n) {
(8)                   create new c_i;
(9)                   c_i = S(s_i) ∩ c_1; c_1 = c_1 \ c_i;
```

```
(10)                root(c_i) = s_i;
                }
(11)            for (i = 1, ..., n)
(12)                if (s(s_i) ≠ ∅) ∧ (∀s ∈ s(s_i) : |S(s) ∩ c_i| ≥ Z)
(13)                    C' = C' ∪ c_i;
                    else
(14)                    L' = L' ∪ c_i;
(15)        } else L' = L' ∪ c;
        }
    }
```

2.3 Load Balancing for Multiplicative Multigrid

The basic idea of load balancing for multiplicative multigrid is to map each grid level T_k evenly onto all processors. This is necessary since the processors will synchronize (locally) on each level before going to the next corser grid. Algorithm cluster will provide a compromise between inter and intra grid communication since communication may occur at most at the "surface" of clusters.

Algorithm 3 Multiplicative Load Balancing. Assume a clustering C has been constructed with algorithm cluster. Then procedure lbmul calculates a mapping $m : C \to P$ using procedure m_partition. Parameter M in lbmul is used to determine the number of processors to use on a certain level of the grid structure.

```
    m_partition (k, C, P, load) {
(1)     if (P = {p}) {for (c ∈ C) m(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)     Find order for C: a : {1, ..., |C|} → C;
(6)     Determine i ∈ {0, ..., |C|} such that
(7)         | (|P_0| / (|P_0|+|P_1|)) W − ( l_0 + Σ_{n=1}^{i} w_k(a(n)) ) | → min;
(8)     C_0 = ⋃_{n=1}^{i} {a(n)}; C_1 = ⋃_{n=i+1}^{|C|} {a(n)};
(9)     m_partition(k, C_0, P_0, load);
(10)    m_partition(k, C_1, P_1, load);
    }

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

(7) m_partition($k, C_k, P', load$);
(8) for ($c \in C_k$)
(9) for (i=bottom(c), ..., top(c)) $load_{i,m(c)} = load_{i,m(c)} + w_i(c)$;
}
}

Procedure lbmul works from level j down to baselevel b (line 2). On each level C_k are the clusters that have no elements above level k. These clusters are now mapped as evenly as possible to the processors by procedure m_partition. Lines 5 and 6 of lbmul determine a subconfiguration that is suitable for the number of elements available on level k. After the clusters have been mapped, the load in each processor on each level is updated in line 9. Note that since $k = \text{top}(c)$ and usually $\text{bottom}(c) < \text{top}(c)$ we already assign some elements of levels below k in step (7). This has to be considered in procedure m_partition.

Procedure m_partition is a general recursive bisection procedure which has been extended to handle arbitrary processor numbers and to take into account that some load has already been assigned to the processors. The type of bisection procedure is determined by the ordering computed in line 5 of algorithm m_partition. Currently we sort alternatingely by x and y coordinates of the cluster root element. It is also possible to impose a neighborship relation on the clusters and use more sophisticated bisection methods at this point, which is currently beeing investigated.

2.4 Load Balancing For Additive Multigrid

In the additive multigrid method we have no synchronization in the smoother before going to the next coarser level, the defect form the finest grid is simply restricted to all coarser levels. This has the consequence that only the number of elements in each processor must be equal no matter how the elements are distributed over the levels. So the ideal case would be to take the elements t of some level b and define the subtrees $S(t)$ as clusters. In the case of adaptive refinement it may happen, however, that some clusters become to large to get a load balance of sufficient quality. The idea is now to subdivide the clusters adaptively in this case with procedure split from above. The algorithm reads as follows:

Algorithm 4 Additive Load Balancing. We assume that a clustering C has been computed by calling cluster($C, \mathcal{T}, j, b, j, Z$) (Note that depth $d = j$ is used). The set L is initially empty, $k = 0$ is used in the first call and tol is the maximum relative error in load balance at the first subdivision level (a typical value is 0.15) and is reduced by a factor σ (typically 0.5) in each recursive call.

lbadd (C, L, P, b, Z, tol, k) {
(1) if ($P = \{p\}$) {$\forall c \in C : m(c) = p$; $\forall l \in L : m(l) = p$; return; }
(2) Divide P into P_0, P_1;
while (true) {
(3) $W_C = \sum_{c \in C} w(c)$; $W_L = \sum_{l \in L} w(c)$;
(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) $C_0 = \bigcup_{n=1}^{i} \{a_C(n)\}$; $C_1 = \bigcup_{n=i+1}^{|L|} \{a_C(n)\}$;
}
(16) for $(i = 0, 1)$ $W_i = \sum_{c \in C_i} w(c) + \sum_{l \in L_i} w(l)$;
(17) if $(\max(W_0, W_1) \leq (1 + tol) W_{opt}) \vee (C = \emptyset)$ break;
(18) split(C, C', L', Z);
(19) $C = C'$; $L = L \cup L'$; $k = k + 1$;
}
(20) lbadd$(C_0, L_0, P_0, b, Z, tol \cdot \sigma, k)$;
(21) lbadd$(C_1, L_1, P_1, b, Z, tol \cdot \sigma, k)$;
}

Procedure lbadd is recursive in the number of processors. In each call the processor configuration is divided into two halves (which need not be of the same size) in step (2). In the while loop we try to map the clusters C and L onto the processors. The mapping step explicitly considers whether the clusters are divisible (large) or indivisible (small). The smaller of the two sets C and L is mapped to one part of the configuration and the larger set is subdivided onto both halves. If the relative error in load balance is below *tol* or no clusters can be subdivided further, the mapping is accepted (step (17)). If load balance is inacceptable procedure split is used to generate smaller clusters allowing better load balance. Steps (20,21) now call the algorithm recursively for each of the two subconfigurations. Note that the tolerance is reduced by a factor σ in each recursive call. The reason is that we want to keep large clusters at the beginning and refine the clustering only in the higher stages where we have smaller subconfigurations.

3 Experimental Results

All computations reported been done on a Parsytec Supercluster SC-128 with 25MHz T800 processors and 4MB of memory per processor. The operating system was PARIX Version 1.2.

3.1 A Model Example

The first test has been designed to study the influence of different locality of refinement on the load balancing quality and parallel efficiencies of the multigrid methods. Therefore we solve

$$\begin{array}{rcll} -\Delta u & = & 0, & \text{in } \Omega = (0,1)^2 \\ u(x,0) & = & \frac{1}{4}x, & x \in [0,1[\\ u(1,y) & = & \frac{1}{4} + \frac{1}{4}y, & y \in [0,1[\\ u(x,1) & = & \frac{1}{2} + \frac{1}{4}(1-x), & x \in [0,1[\\ u(0,y) & = & \frac{3}{4} + \frac{1}{4}(1-y), & y \in]0,1[\end{array} \quad . \tag{3}$$

The initial triangulation T_0 consists of 4 quadrilaterals with one unknown. Refinement is uniform up to some prescribed baselevel b and beginning with level b refinement is such that the grid on level k is restricted to the rectangle $[0, s_k]^2$ with $s_k = (\sqrt{w}/2)^{k-b}, (k > b)$. The factor w is called the "growth factor", since the number of elements on level $k+1$ is defined recursively by $|T_{k+1}| = w|T_k|$. With $w = 4$ one gets uniform refinement and $w = 1$ indicates a case where there is no geometric growth in the number of unknowns. Figure 1(a) shows the grid and solution for $w = 2$ and $j = 8$. Table 1 shows the results for $w = 1, 2, 3, 4$ and additive and multiplicative multigrid. Note that the problem size is increased with the number of processors. However it is not possible to get always the same number of unknowns N per processor for different values of w. This makes it difficult to compare results for different w and the same number of processors.

The conclusions drawn from this test are:

- The additive method has always better or equal (only one case) efficiencies than the multiplicative method. This is due to coarser grained parallelism and the possibility of using decompositions with smaller interfaces. The latter is the more important point which can be seen by considering the results for $P = 64$ and comparing efficiencies for different w. For $w = 4$ both load balancing methods yield the same decomposition and efficiencies differ not much. For w getting smaller the differences become greater. Figure 1(b) shows the load distribution on 8 processors for the additive method and (c) for the multiplicative method. The figures show only the position of the "topmost" elements.
- The solution time is always *smaller* for the multiplicative method than for the additive method, except the case of $w = 1, P = 64$. This is not a result of the small number of unknowns per processor. In contrary to the case $w > 1$, one can observe for $w = 1$ that efficiencies do not increase with problem size above level 10 for a fixed number of processors. This is due to the fact that the unknowns do not grow geometrically with the number of levels (no decrease of the surface to volume ratio).
- In the case where additive multigrid is equal or better than multiplicative multigrid in terms of total computation time, both methods have to be considered as inefficient. Since additive multigrid needs twice as many iterations as multiplicative multigrid, parallel efficiency for multiplicative multigrid must be below 50% in order to allow additive multigrid to be better. Since there are also losses in the latter method, the break even point happens to be at 25% efficiency for multiplicative and 50% efficieny for additive multigrid in this example.

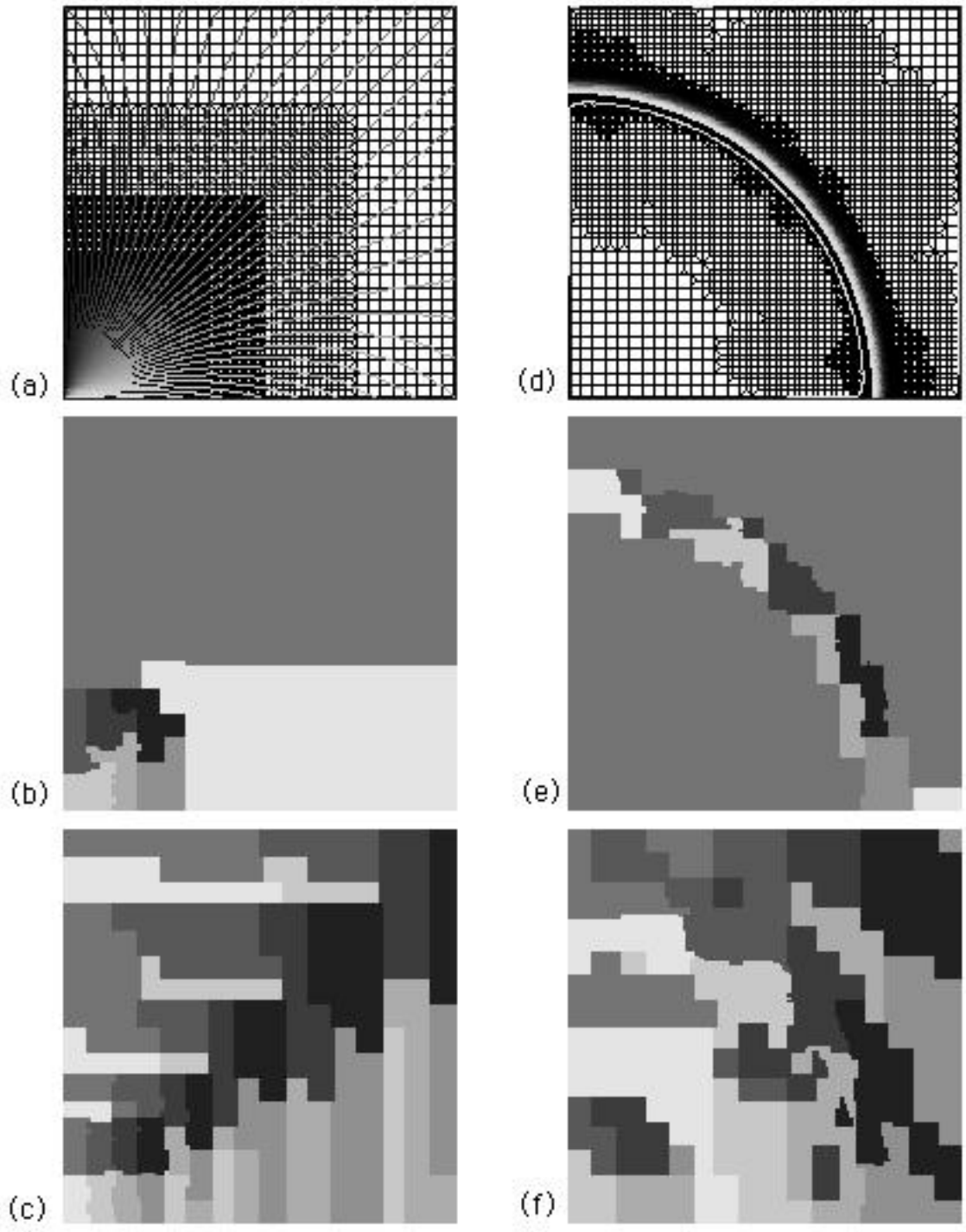

Figure 1: *Grids and load distribution for the two numerical examples.*

The parallel efficiency E_{IT} is identical to the total efficiency in the solver for this example since the number of iterations needed are the same for serial and parallel computation. An important question is where the losses come from. Considering one multigrid cycleone can identify three sources for inefficencies:

- Idle times: Nonoptimal load balance results in idle times where a processor must wait for data from another processor.

Table 1: *Results for different locality of refinement (w, see text) and a varying number of processors. T_{SOL} is total time for a 10^{-6} reduction in residual norm on the finest level j after a nested iteration. E_{IT} is the parallel efficiency of one iteration, including the cg method in the additive case. Multigrid data: $\nu_1 = \nu_2 = 1$, V-cycle for multiplicative multigrid, one smoothing step for additive multigrid, smoother was point-jacobi in both cases. Refinement was uniform up to level 4 ($h = 1/32$) except for cases $w = 1$ and $P > 1$ where refinement was uniform up to level 5.*

w		multiplicative mg +jac					cg+ additive mg +jac				
	P	1	4	16	32	64	1	4	16	32	64
4	j	4	5	6	7	7	4	5	6	7	7
	N	1089	4225	16641	66049	66049	1089	4225	16641	66049	66049
	T_{SOL}	5.95	5.87	6.55	12.49	6.83	9.33	10.17	11.04	20.98	11.43
	E_{IT}		85	75	77	71		88	79	82	76
3	j	5	6	7	7	8	5	6	7	7	8
	N	3553	10657	31656	31656	94440	3553	10657	31656	31656	94440
	T_{SOL}	18.14	15.59	13.09	7.99	15.46	31.38	25.81	21.44	11.97	17.85
	E_{IT}		86	76	71	47		90	80	71	71
2	j	5	7	8	9	10	5	7	8	9	10
	N	2768	12223	24767	49974	99638	2768	12223	24767	49974	99638
	T_{SOL}	15.59	19.05	11.77	12.60	13.44	26.00	28.43	18.68	19.77	20.92
	E_{IT}		78	64	59	55		86	71	74	63
1	j	6	6	10	13	15	6	6	10	13	15
	N	2753	7425	20225	29825	36225	2753	7425	20225	29825	36225
	T_{SOL}	15.39	11.13	10.17	10.69	11.14	24.37	18.49	15.68	13.93	9.96
	E_{IT}		78	60	41	24		85	64	53	45

- Double computations: The unique mapping of elements to processors results in multiple computations on the overlapping nodes.
- Cost of communication: There may be considerable non nearest-neighbor asynchronous communication.

The obtainable efficiency when counting only the double computations can be easily computed. For the case $P = 64, w = 1$ we determined a theoretical efficiency of 44% for the multiplicative method and 75% for the additive method. The differences to the measured values must be due to communication cost and idle times.

3.2 Comparison of Uniform and Adaptive Computation

The purpose of this test is to compare parallel adaptive computation with parallel uniform computation. To that end, an exact solution showing local behaviour is prescribed in order to be able to compare the discretization error on different grids. The b.v.p. is again Laplace's equation in the unit square, here with the exact solution

$$u = \frac{1}{1 + e^{-200(r-0.8)}}, \qquad r = \sqrt{x^2 + y^2} \quad . \tag{4}$$

Figure 1(d) shows the locally refined grid (residual based error estimator) and load distribution on 8 processors for the additive (e) and multiplicative case (f). Table 2

Table 2: *Results for example with prescribed exact solution. Reported are total computation times for reaching a solution with the same discretization error in the L_∞ norm. Multigrid data: multiplicative multigrid with symmetric Gauß-Seidel smoother, $\nu_1 = \nu_2 = 1$, 10^{-4} reduction per level in the adaptive scheme. The initial grid used 4 square elements to cover the unit sqare.*

	uniform				adaptive			
	j=8,N=263169,$\|u-u_h\|_\infty = 1.54 \cdot 10^{-3}$				j=8,N=29262,$\|u-u_h\|_\infty = 1.65 \cdot 10^{-3}$			
	CRIMSON	P=80	P=96	P=128	CRIMSON	P=8	P=16	P=20
T_{JOB}	468	109	98	84	77	176	123	107
T_{LB}		17.9	17.3	17.8		21.9	24.2	22.6

shows total computation times for the full adaptive method for a number of parallel configurations and a large workstation (Silicon Graphics Crimson). The results can be summarized in:

- The gain in the number of unknowns was a factor of 9 in the final grid hierarchy. Since savings were smaller on coarser grids the total gain was 6 on the (single processor) workstation.
- In the parallel case the gain was a factor of 4 in the number of processors, i.e. 20 processors needed the same time with adaptive computation as 80 processors with uniform computation. However it should be noted that the uniform version also spent a relatively large part of computation time for load balancing. This is due to load balancing required when grids get finer in the nested iteration and more processors can be used efficiently (The coarsest grid was $h = 1/2$ in all tests!).
- Of course the comparison above used the unstructured code for the uniform calculations although the grid was structured and rectangular. A traditional structured code would be about 2 to 4 times faster than the unstructured code but this comparison would not be fair since there is big difference in functionality.

4 Conclusions

It has been shown in the present work that the parallelization of adaptive multigrid methods on unstructured grids is possible with acceptable efficiencies. Parallel efficiency usually ranges from 50% on 64 processors to 75% on 16 processors over several examples with different locality of refinement and shape of the refinement region (see [3] for more examples).

Emphasis has been put on a comparison of multiplicative and additive multigrid methods, since they differ in granularity and especially in the way the grid hierarchy can be mapped onto the processors. Additive multigrid allows partitions with much lower inter- and intra-grid communication. Therefore parallel efficiencies are usually higher and scalability is better. However, the gain is usually not sufficient to compensate the worse numerical efficiency for the processor numbers considered.

Acceptable efficiencies for the full adaptive computation are currently limited to not more than 16 processors. This is due to the load migration algorithm, which is comparable in complexity to the solution of the simple linear b.v.p. considered (but

does not scale as well). The problem will be less important as soon as the time between two load balancing steps becomes larger, i.e. when more complicated equations are to be solved. In this case it will be also advantageous to use more sophisticated (and more time consuming) mapping algorithms to increase the quality of the load balance.

References

[1] P. Bastian: *Locally Refined Solution of Unsymmetric and Nonlinear Problems*, Proceedings of the 8^{th} GAMM Seminar, Kiel, NNFM, Vieweg Verlag, Braunschweig, 12-21, 1993.

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

[3] P. Bastian: *Parallel Adaptive Multigrid Methods*, IWR Report 93-60, Universität Heidelberg, October 1993.

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

[5] 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.

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

[7] B. W. Kernighan, S. Lin: *An Efficient Heuristic Procedure for Partitioning Graphs.* The Bell System Technical Journal, **49**, 291-307, 1970.

[8] B. Hendrickson, R. Leland: *An Improved Spectral Load Balancing Method*, Proc. of the 6^{th} SIAM Conference on Parallel Processing for Scientific Computing, March 1993.

[9] A. Pothen, H. Simon, K. Liou: *Partitioning Sparse Matrices with Eigenvectors of Graphs*, SIAM J. Matrix Anal. Appl., **11**, 430-452, 1990.

[10] P. Sadayappan, F. Ercal, J. Ramanujam: *Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube*, Parallel Computing, **13**, 1-16, (1990).

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

[12] G. E. Schneider, M. J. Raw: *A Skewed, Positive Influence Coefficient Upwinding Procedure for Control-Volume-Based Finite-Element Convection-Diffusion Computation*, Numerical Heat Transfer, **9**, 1-26 (1986).

[13] T. Sonar: Strong and Weak Norm Refinement Indicators Based on the Finite Element Residual for Compressible Flow Computation, IMPACT of Computing in Science and Engineering, **5**, 111-127 (1993).

[14] R. D. Williams: *Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations*, Report C3P 913, California Institute of Technology, Pasadena, CA., (1990).

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