ELSEVIER

Applied Numerical Mathematics 16 (1994) 157–182

APPLIED NUMERICAL MATHEMATICS

Load balancing for the parallel adaptive solution of partial differential equations

H.L. deCougny, K.D. Devine, J.E. Flaherty *, R.M. Loy, C. Özturan, M.S. Shephard

Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA

This paper is dedicated to Professor Robert Vichnevetsky to honor him on the occasion of his 65th birthday

Abstract

An adaptive technique for a partial differential system automatically adjusts a computational mesh or varies the order of a numerical procedure with a goal of obtaining a solution satisfying prescribed accuracy criteria in an optimal fashion. Processor load imbalances will, therefore, be introduced at adaptive enrichment steps during the course of a parallel computation. We develop and describe three procedures for retaining and restoring load balance that have low unit cost and are appropriate for use in an adaptive solution environment.

Tiling balances load by using local optimality criteria within overlapping processor neighborhoods. Elemental data are migrated between processors within the same neighborhoods to restore balance. Tiling is restricted to uniform two-dimensional meshes and provides limited control of communications volume by priority-based element selection criteria. These shortcomings can potentially be overcome by creating a dynamic partition graph connecting processors and their neighboring regions. After coloring the edges of the graph, elemental data are iteratively transferred between processors by pairwise exchange to permit a more global migration.

Octree decomposition of a spatial domain is a successful three-dimensional mesh generation strategy. The octree structure facilitates a rapid load balancing procedure by performing tree traversals that (i) appraise subtree costs and (ii) partition spatial regions accordingly.

Computational results are reported for two- and three-dimensional problems using nCUBE/2 hypercube, MasPar MP-2, and Thinking Machines CM-5 computers.

1. Introduction

Adaptive finite element methods that automatically refine or coarsen meshes (h-refinement) and/or vary the order of accuracy of a method (p-refinement) offer greater reliability, robustness, and computational efficiency than traditional numerical approaches for solving partial differential equations.

* Corresponding author. E-mail:flaherje@cs.rpi.edu.

0168-9274/94/$07.00 © 1994 Elsevier Science B.V. All rights reserved
SSDI 0168-9274(94)00039-5

High-order methods and the combination of mesh refinement and order variation (hp-refinement) can produce remarkably efficient methods with exponential convergence rates [2,4,5,11,12,27]. Like adaptivity, parallel computation is making it possible to solve previously intractable problems. With problems continuing to increase in complexity through the inclusion of more realistic effects in models, it seems natural to unite adaptivity and parallelism to achieve the highest gains in efficiency. Adaptivity, however, introduces complications that do not arise when simpler solution strategies are implemented on parallel computers. Adaptive algorithms utilize unstructured [2] or hierarchical [3,7] meshes that make the task of balancing processor loading much more difficult than with uniform structures. A balanced loading will, furthermore, become unbalanced as additional degrees of freedom are introduced or removed by adaptive h- and p-refinement.

Successful load partitioning strategies for unstructured-mesh computation on distributed-memory parallel computers employ recursive bisection to repeatedly split the discretized domain into two subdomains having balanced loading. Specific techniques use geometric [6], connective [15], or spectral [16,26] information. When applied to the entire mesh, recursive bisection methods require a complete remapping of the elements of the mesh and, thus, involve a substantial overhead. Some methods also require considerable computation [26]. Thus, global recursive bisection methods are too expensive for use with adaptive methods which, as noted, require repeated element redistribution. Recursive bisection may be of use with an adaptive strategy if applied locally to regions of the domain affected by adaptive enrichment [33].

Two partitioning strategies described herein use local migration to exchange elements between processors associated with neighboring spatial regions in order to achieve a global load balance. Local interchanges propagate incremental changes in the mesh or method between processors without solving an expensive global partitioning problem. Local computational cost metrics, such as the number of degrees of freedom, can be combined with similar information on partition boundaries to minimize the total workload including both the computational and communications efforts.

Our most mature partitioning strategy *tiling* [34] is a modification of a dynamic load balancing technique developed by Leiss and Reddy [25,28] who used local optimality criteria within overlapping processor neighborhoods to achieve a global load balance. A neighborhood consisted of a processor at the center of a circle of a given radius and all processors within that circle. With tiling, we extend the definition of a neighborhood to include all processors having finite elements that are neighbors of elements in the central processor (cf. Fig. 3). Every processor is the center of one neighborhood and, typically, belongs to many neighborhoods. Elements are migrated from a processor to others within the same neighborhood to obtain local optimality. Tiling is applicable to two-dimensional problems on structured meshes and we demonstrate its performance by using it with adaptive h- and p-refinement strategies to solve transient systems of conservation laws on a 256-processor nCUBE/2 hypercube (cf. Section 2).

At present, tiling is unsuitable for unstructured and three-dimensional meshes. With only local optimization, it can require several time or iterative steps to achieve a global balance. Finally, tiling provides limited capabilities for controlling the shape of partitions to reduce the communications volume. Partition shape control could utilize orthogonal recursive bisection [6] in, say, directions of principal axes of inertia of partitions. Redistribution through pairwise exchanges, our second migration strategy, builds upon tiling and can correct some of these deficiencies. Neighborhood adjacency is represented by a dynamic partition graph. Loading information is used to color edges of the partition graph so that work can be transferred between pairs of processors in a manner similar to

a pairwise heuristic strategy introduced by Hammond [19]. This strategy, described in Section 3, can be used with local partitioning strategies to provide better shape control when selecting elements for migration. Two-dimensional unstructured-mesh computations performed on a MasPar MP-2 SIMD system demonstrate some capabilities of this procedure.

Octree decomposition is a successful strategy for generating three-dimensional unstructured meshes [29] and we develop (cf. Section 4) a partitioning technique that exploits the properties of the underlying tree structure. Partitioning may be done locally or globally, but, in either case, it is inexpensive and, hence, may be used with adaptive procedures. Partitioning is based on two tree traversals that (i) calculate the processing costs of subtrees connected to each node and (ii) form the partitions. When used globally, partitions have approximately the same communications volume as other strategies [21,24,26], but their cost is far less. We demonstrate the performance of the tree-based partitioning technique on three-dimensional meshes that are associated with flight vehicle flows. Results computed on a Thinking Machines CM-5 computer are presented for an adaptive h-refinement solution of the Euler equations for a supersonic conical flow.

2. Tiling

2.1. Adaptive enrichment

We describe adaptive h- and p-refinement local time-stepping algorithms that are being used with the tiling partitioning scheme (Section 2.2) but which are typical of adaptive strategies. Applied to vector systems of conservation laws, finite element solutions $U(x,t)$ are obtained on a two-dimensional net of rectangular elements using a spatially discontinuous Galerkin method [5,8–10] and explicit Runge–Kutta integration [5]. A spatial discretization error estimate $E(t)$ in the L^1 norm is obtained by p-refinement [5,11] and used to control adaptive spatial enrichment so that $E(t) \leqslant \varepsilon$, for a prescribed tolerance ε.

With adaptive p-refinement (cf. Fig. 1), we initialize $U(x,0)$ to the lowest-degree polynomial satisfying $E_j(0) \leqslant \varepsilon/J$, $j = 1,2,\ldots,J$, where $E_j(t)$ is the restriction of $E(t)$ to element j and J is the number of elements in the mesh. After each time step, we compute E_j, $j = 1,2,\ldots,J$, and increase the polynomial degree of U on element j by one if $E_j > \varepsilon/J$ (=TOL). The solution U and the error estimate are recomputed on enriched elements, and further increases of degree occur until $E_j \leqslant TOL$ on all elements. The need for backtracking may be reduced by predicting the degree of the approximation needed to satisfy the accuracy requirements for the subsequent time step. After a time step is accepted, if $E_j > H_{\max}TOL$, $H_{\max} \in (0,1]$, we increase the degree of $U(t+\Delta t)$ on element j for the next time step. If $E_j < H_{\min}TOL$, $H_{\min} \in [0,1)$, we decrease the degree of $U(t+\Delta t)$ for the next time step.

In the adaptive h-refinement algorithm (cf. Fig. 2), we locally refine element j if $E_j > TOL/2^m$, where m is the level of refinement. Refinement involves dividing an element into four and initializing the solution through L^2 projection of the coarse-mesh data [5]. Elements neighboring high-error elements are also refined to provide a buffer between high- and low-error regions and maintain a difference of at most one level of refinement across element edges. For each time step, the local finite element method [5] is applied on successively finer meshes. To satisfy the Courant conditions, the time step is halved on each finer mesh.

```
void adaptive_p_refinement()
{
    while (t < t_final) {
        perform_runge_kutta_time_step(all_elements);
        do {
            Solution_Accepted = TRUE;
            for each element {
                error_estimate = calculate_estimate();
                if (error_estimate > TOL) {
                    mark_element_as_unacceptable();
                    increase_elements_polynomial_degree();
                    Solution_Accepted = FALSE;
                }
            }
            if (!Solution_Accepted) {
                recalculate_solution_on_unacceptable_elements();
            }
        } while (!Solution_Accepted);
        accept_solution(all_elements);

        predict_degrees_for_next_time_step(all_elements);
        t = t + Δt;
    }
}
```

Fig. 1. An adaptive p-refinement procedure.

2.2. *Dynamic load balancing via tiling*

The tiling algorithm consists of (i) a computation phase and (ii) a balancing phase, and is designed to be independent of the solution procedure. The computation phase corresponds to solution generation without load balancing. Each processor operates on its local data, exchanges inter-processor boundary data, and processes the boundary data. A balancing phase restores load balance following a given number of computation phases. Each balancing phase consists of the following operations:

(1) Each processor determines its work load as the time to process its local data since the previous balancing phase less the time to exchange inter-processor boundary data during the computation phase. Average work loads are also calculated for each overlapping neighborhood (cf. Fig. 3).
(2) Each processor compares its work load to the work load of the other processors in its neighborhood and determines those processors having loads greater than its own. If any are found, it selects the one with the greatest work load (ties are broken arbitrarily) and sends a request for work to that processor. Each processor may send only one work request, but a single processor may receive several work requests.

```
void adaptive_h_refinement(mesh, t_start, t_final, Δt, TOL)
{
    t = t_start;
    fine_mesh = mesh → nextmesh;
    while (t < t_final) {
            perform_runge_kutta_time_step(all elements of mesh);
            for each element of mesh {
                error_estimate = calculate_estimate();
                if ((error_estimate > TOL) && (element_not_refined_yet)) {
                    refine_element_into_four_fine_elements();
                    add_new_elements(fine_mesh);
                }
            }
            if (mesh is refined) {
                buffer(fine_mesh);
                project_coarse_data(mesh, fine_mesh);
                adaptive_h_refinement(fine_mesh, t, t + Δt, Δt/2, TOL/2);
                interpolate_fine_solution_to_coarse_mesh(fine_mesh, mesh);
            }
            t = t + Δt;
    }
}
```

Fig. 2. An adaptive h-refinement procedure.

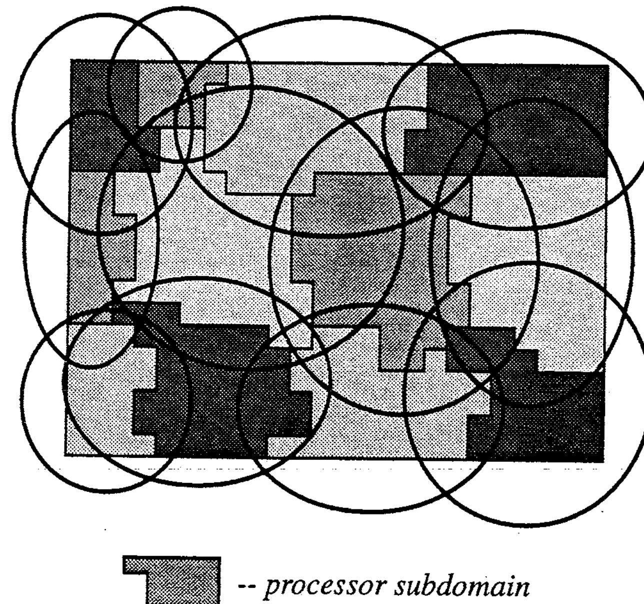

Fig. 3. Example of 12 processors in 12 neighborhoods using tiling.

(3) Each processor prioritizes the work requests it receives based on the request size, and determines which elements to export to the requesting processor. Details of the selection algorithm are given below.
(4) Once elements to be exported have been selected, the importing processors and processors containing neighbors of the exported elements are notified. Importing processors allocate space for the incoming elements, and the elements are transferred.

Each processor knows the number of computation phases to perform before entering the balancing phase. Synchronization guarantees that all processors enter the balancing phase at the same time.

The technique for selecting elements gives priority to elements with neighbors in the importing processor to prevent the creation of "narrow, deep holes" in the element structures. Elements are assigned priorities (initially zero) based upon the locality of their neighbors. An element's priority is decreased by one for each neighbor in its own processor, increased by two for each neighbor in the importing processor, and decreased by two for each neighbor in some other processor. Thus, elements whose neighbors are already in the importing processor are more likely to be exported to that processor than elements whose neighbors are in the exporting processor or some other processor. When an element has no neighboring elements in its local processor, it is advantageous to export it to any processor having its neighbors. Thus, "orphaned" elements are given the highest export priority.

Because individual elements' processing costs can vary widely in the adaptive p-refinement method, elemental processing costs are computed and used so that the minimum number of elements satisfying the work request are exported. This approach differs from that of Wheat [34], who uses the average cost per element to determine the number of export elements. When two or more elements have the same priority, the processor selects the element with the largest work load that does not cause the exported work to exceed the work request or the work available for export.

In the adaptive h-refinement method, the local time-stepping scheme, outlined in Fig. 2, requires that each mesh level be distributed evenly over the processor array to avoid idle time. Communication costs are increased since offspring elements may be on different processors than their parent elements; however, the increase in communication time is outweighed by a decrease in processor idle time. Memory overhead for the tiling algorithm is also increased, as processor location information for parent and offspring elements must be maintained and "ghost elements" near subdomain boundaries must be allocated for non-local coarse elements along coarse–fine mesh interfaces. Selection priority schemes that account for the interconnections between mesh levels could reduce the additional communication and storage needed; such schemes are the subject of future study.

Example 2.1. We solve

$$u_t + 2u_x + 2u_y = 0, \quad t > 0, \tag{1a}$$

on $0 < x, y < 1$ with initial and boundary conditions specified so that

$$u(x, y, t) = \tfrac{1}{2}[1 - \tanh(20x - 10y - 20t + 5)], \tag{1b}$$

using adaptive p-refinement on a (32×32)-element mesh with $TOL = 3.5 \times 10^{-5}$ and tiling on 16 processors. In Fig. 4, we show the processor domain decomposition after 20 time steps. The shaded elements have higher-degree approximations and, thus, higher work loads. The tiling algorithm redistributes the work so that processors with high-order approximations have fewer elements than those processors with low-order approximations. The total processing time for the adaptive p-refinement

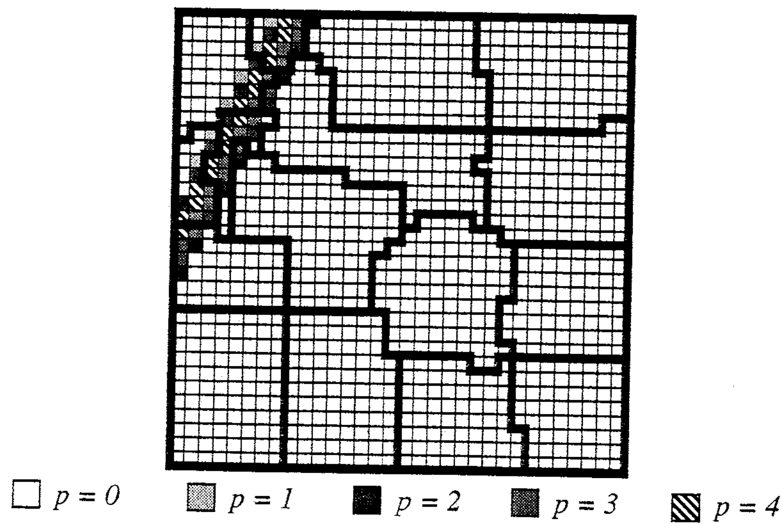

Fig. 4. Processor domain decomposition after 20 time steps for Example 2.1 using adaptive p-refinement and tiling. Dark lines represent processor subdomain boundaries.

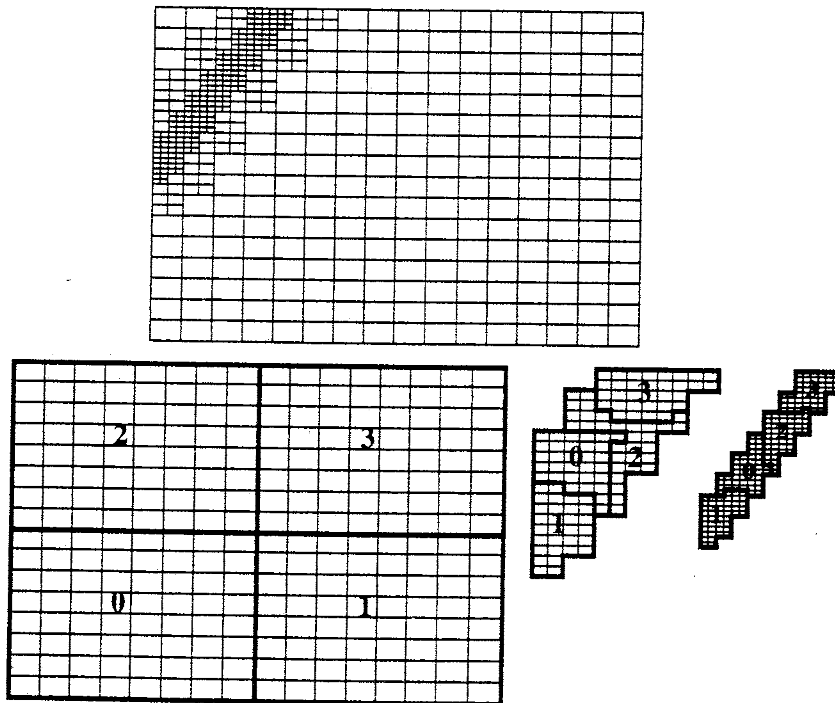

Fig. 5. Processor domain decomposition after 10 time steps for Example 2.1 using adaptive h-refinement and tiling. The decomposition on each mesh level is shown.

method was reduced 41.98% from 63.94 seconds to 37.10 seconds by balancing once each time step. The average/maximum processor work ratios without and with balancing are 0.362 and 0.695, respectively. Parallel efficiency is increased from 35.10% without balancing to 60.51% with tiling.

We also solve (1) using the adaptive h-refinement method on a (16×16)-element base mesh with $TOL = 10^{-3}$ and tiling on 4 processors. In Fig. 5, we show the processor domain decomposition for each mesh level after 10 times steps. The total processing time for the adaptive h-refinement method was reduced 58.0% from 104.89 seconds to 44.01 seconds by balancing a mesh after each time

Table 1
Performance comparison for Example 2.2 using adaptive p-refinement without balancing and with balancing at each time step, and a fixed-order method yielding comparable accuracy

	Adaptive p-refinement		Fixed-order
	Without tiling	With tiling	Without tiling
Total execution time (seconds)	3636.53	1088.36	10,590.87
Maximum computation time (seconds)	3557.77	778.57	10,570.07
Average/maximum work ratio	0.118	0.543	0.999
Average communication time (seconds)	22.31	31.38	19.03
Maximum balancing time (seconds)	0.00	28.98	0.00
Parallel efficiency	11.62%	38.84%	99.71%

step. The average/maximum processor work ratios without and with balancing are 0.271 and 0.747, respectively. Parallel efficiency is increased from 26.7% without balancing to 63.8% with tiling.

Example 2.2. Again, we solve (1) on $\Omega = (0, 16) \times (-7.5, 8.5)$ with a fixed-order ($p = 2$) method and with adaptive p-refinement for 86 time steps to $t = 0.3$ using a (160×160)-element mesh on 256 processors of the nCUBE/2 without balancing and with balancing once each time step. In the adaptive p-refinement method, polynomial degrees of the elements varied from 0 to 2, and computation time per element varied from 0.02 to 1.2 seconds per time step, indicating a great deal of imbalance along the front. Even without balancing, the adaptive p-refinement method required 65.7% less execution time than the fixed-order method to achieve comparable accuracy (cf. Table 1). With balancing, the maximum computation time of the adaptive method (excluding communication or balancing time) was further reduced by 78.1%. The irregular subdomain boundaries created by the tiling algorithm increased the average communication time by 40.7%. Despite the extra communication time and the load balancing time, there is a 70.1% improvement in the total execution time relative to the non-balanced adaptive method, and an 89.7% improvement relative to the fixed-order method.

Example 2.3. We solve

$$u_t + 2u_x + 2u_y = 0, \quad t > 0, \tag{2a}$$

on $\Omega = (0, 16) \times (-7.5, 8.5)$ with initial and boundary conditions specified so that

$$u(x, y, t) = \tfrac{1}{2}[1 - \tanh(100x - 10y - 20t + 5)], \tag{2b}$$

with $p = 0$ on a uniform (640×640)-element mesh and on a (160×160)-element base mesh with adaptive h-refinement for 60 time steps on 256 nCUBE/2 processors without balancing and with balancing once per time step on each mesh level. Two levels of refinement were used along the steep front. We compare the adaptive h-refinement computation with a uniform mesh computation of similar accuracy. The adaptive solution required 46.9% less total execution time than the non-adaptive solution, despite the load imbalances created by the adaptive method (cf. Table 2). With balancing, the maximum computation time (excluding communication or balancing time) was reduced by 86.1% relative to the adaptive method without balancing. The average communication time with balancing is nearly doubled, due to the communication between coarse and fine mesh elements that have been

Table 2
Performance comparison for Example 2.3 using adaptive h-refinement without balancing and with balancing at each time step on each mesh level, and a uniform mesh yielding comparable accuracy

	Adaptive h-refinement		Uniform mesh
	Without tiling	With tiling	Without tiling
Total execution time (seconds)	3455.07	681.48	6508.16
Maximum computation time (seconds)	3430.17	476.00	6491.07
Average/maximum work ratio	0.0743	0.535	1.000
Average communication time (seconds)	7.68	15.33	14.82
Maximum balancing time (seconds)	0.00	42.14	0.00
Parallel efficiency	7.38%	37.36%	99.70%

migrated to different processors. The total balancing time is larger than the balancing time for p-refinement, since many more balancing phases are used as each mesh level is load balanced. Despite the tiling overhead, we see an 80.3% improvement in the total execution time of the h-refinement method.

3. Element redistribution by pairwise exchange

3.1. Redistribution

The element redistribution algorithm and its similarities and differences to the tiling procedure of Section 2 are described through an example. Consider the unbalanced mesh distribution on eleven processors as shown in Fig. 6(a). Let $G_P(V, E)$ be a *partition graph* with each vertex in V representing a partition assigned to a processor and E representing the set of edges between partitions. Two partitions u and v are connected by an edge $(u, v) \in E$ if they share a mesh edge, and not just a vertex. By excluding vertex adjacency in G_P, the number of edges in E and, hence, the time to communicate with adjacent processors is kept minimal during the load balancing phase. Additionally, transferring elements between partitions that share only a vertex results in a higher surface to volume ratio which increases communication cost. Fig. 6(b) shows the partition graph obtained from the mesh distribution in Fig. 6(a).

Following Leiss and Reddy [25], a work load deficient processor will request work from its most heavily loaded neighbor. As a result, a processor can receive multiple requests but can only request load from one processor. This pattern of requests produces a load hierarchy and forms a forest of trees T_i, consisting of subgraphs of G_P, as shown in Fig. 6(c).

The proposed redistribution algorithm pairs the processors on each tree T_i and transfers load from the heavily loaded to the lightly loaded processor of the pair. The pairing of processors is equivalent to coloring the edges of each tree T_i with colors representing separate load transfer (communication) cycles. The edge coloring approach synchronizes the load transfer between neighboring processors and differs from tiling (cf. Section 2). With pairing, processors P_2 and P_7 of Fig. 6(b) would be engaged with load transfer with only one neighbor at a single transfer step. With tiling, processors P_2 and P_7 would receive and send work during the same transfer step. Tiling would be more efficient since load transfer occurs in one direction only, i.e., from a heavily loaded to less loaded processor. Processors

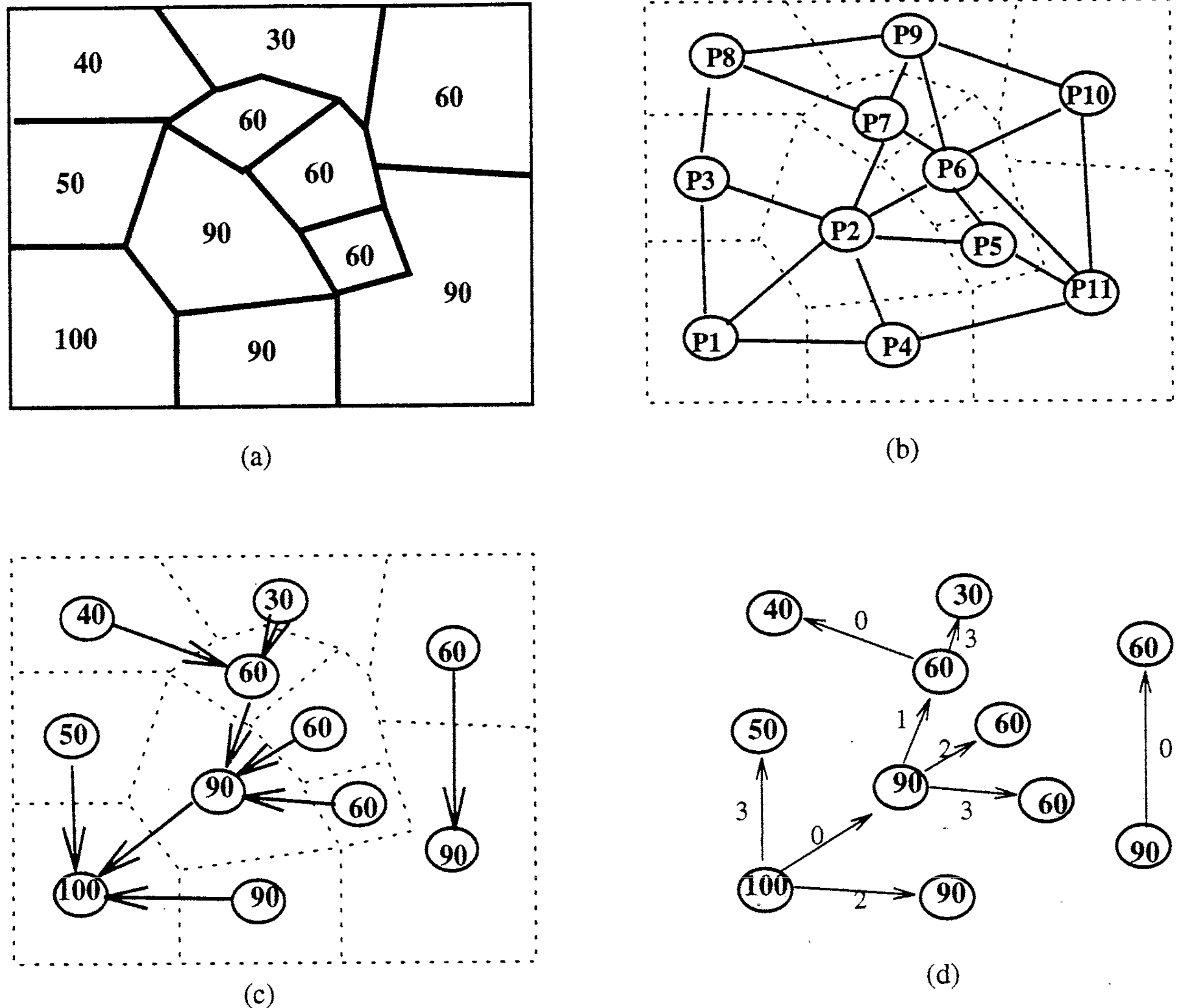

Fig. 6. Unbalanced load in each partition (a), partition graph G_P (b), load request (c), load transfer between pairs in steps 0,. . .,3 (d).

P_2 and P_7 would be doing useful work packing elements to be transferred to their offspring while their parents pack elements to be transferred to them. With pairwise exchange, coloring determines the ordering of processor load transfers. Therefore, in this example, P_7 would first exchange load with offspring P_8 corresponding to color 0. It would then exchange load with parent P_2 corresponding to color 1 and during the third color cycle, exchange with P_9. The color cycles cause processors to remain idle and, hence, reduce efficiency. However, this disadvantage is overshadowed by a number of advantages. First, if unidirectional load transfers are done, the fewer messages per color cycle may reduce network congestion and synchronous exchange of messages will increase communication performance. Second, since processors are synchronized by pairs, a greater repertoire of selection criteria can be used to decide which elements to transfer. Unlike tiling, where elements can only be transferred from heavily loaded to less loaded processors, pairing allows transfers in the opposite direction. This can be used to improve the surface to volume ratio (or communications volume) of partitions. Since there is no explicit synchronization by edge coloring with tiling, a bidirectional transfer of load would be extremely difficult.

Fig. 6(d) shows the coloring phase used to pair processors. If $\Delta(G)$ denotes the maximum vertex

```
void redistribute(mesh, tol_imbalance, max_iters)
{
    mypid = get_my_processor_id();
    iter = 0 ;
    while (imbalance(mesh) > tol_imbalance && iter < max_iters) {
        compute_neighboring_load_differences(mesh);
        proc = neighbor_having_largest_load_difference(mesh);
        T = request_load_from_neighbor(proc, mesh);
        determine_amount_of_load_to_send_or_receive(T, mesh);
        set_up_links_to_linearize_tree(T);
        color_tree(T);

        for each color C {
            if (processor_owns_ color(C, mypid) &&
                is_a_neighbor_of_color_pair(C, mypid, pair_processor))
              transfer_load_between_pair(mesh, mypid, pair_processor);
            }
        iter = iter + 1;
        }
}
```

Fig. 7. Redistribution algorithm.

degree (number of edges incident on a vertex) in a graph G, then Vizing's theorem [36] indicates that G can be edge-colored using C colors with $\Delta(G) \leqslant C \leqslant \Delta(G)+1$. Special graphs, such as trees, only need $\Delta(G)$ colors; therefore, $\Delta(T_i)$ colors are required to color T_i.

The main steps of the *redistribute* algorithm are illustrated in Fig. 7 and the detailed steps follow:

(1) The transfer of work between paired processors is iterated until the load on each processor converges to a value close to the optimal balance (cf. Section 3.2).
(2) Load differences are computed in $\Delta(G_P)$ time by having each processor send and receive load values to and from its neighbors.
(3) The Leiss and Reddy [25] load request process is used to construct the forest of trees T_i. An edge of G_P is marked when a request has been made to indicate whether or not it is a tree edge. Since the incoming requests for load should be sorted, this step takes $O(\max_i\{\Delta(T_i)\log\Delta(T_i)\})$ time.
(4) Deciding the load to transfer to requesting processors is critically related to the convergence of the redistribution algorithm (cf. Section 3.2).
(5) To facilitate efficient parallel scan operations on T_i, each tree is linearized by establishing links between neighboring processors. The links can be constructed by either an Euler Tour [22] or a depth-first traversal [31] of the tree.
(6) The linearized tree is edge-colored by employing a parallel scan operation. Since there can be as many as $\Delta(T_i)$ links on a processor, the scan operation using the Euler Tour links takes $O(\max_i\{\Delta(T_i)\log|V_i|\})$, where $|V_i|$ is the number of vertices in T_i. A depth-first traversal

stores two links per processor and, hence, enables a more efficient scanning step with time complexity $O(\max_i\{\log|V_i|\})$.

(7) One iteration of the edge-synchronized redistribution algorithm involves C steps corresponding to the $\max_i(\Delta(T_i))$ colors.

(8) The elements to be transferred are selected. As with tiling, a cost reflecting the communication and computational effort is associated with partition-boundary elements. The element yielding the smallest increase in communication cost is transferred.

3.2. Load transfer and convergence

Suppose a parent processor with load value L_0 has m load requesting offspring with load values L_i, $i = 1, 2, \ldots, m$, as shown in Fig. 8(a). Each offspring requests an amount r_i which is equal to the difference from its current load to the average of its load and that of its parent, i.e.,

$$r_i = \lceil (L_0 - L_i)/2 \rceil. \tag{3a}$$

The parent processor will send a total load to make its load become the average of the loads L_i, $i = 0, 1, \ldots, m$; thus,

$$to_send_{\text{tot}} = L_0 - \left\lfloor \frac{\sum_{i=0}^{m} L_i}{m+1} \right\rfloor. \tag{3b}$$

The parent determines the individual amounts to_send_i to transfer to children in proportion to the their load requests truncated to the nearest integer with the remainder distributed evenly, i.e., for $i = 1, \ldots, m$,

$$to_send_i = min\left\{r_i, \left\lfloor to_send_{\text{tot}} \cdot \frac{r_i}{\sum_{j=1}^{m} r_j} \right\rfloor + \delta_i\right\}, \tag{3c}$$

where

$$\delta_i = \begin{cases} 1, & \text{if } i \leqslant to_send_{\text{tot}} - \sum_{k=1}^{m} \left\lfloor to_send_{\text{tot}} \cdot \frac{r_k}{\sum_{j=1}^{m} r_j} \right\rfloor, \\ 0, & \text{otherwise.} \end{cases} \tag{3d}$$

The minimum prevents transferring loads greater than the requested load. In Fig. 8(b), we show the load requests and grants for a sample subtree.

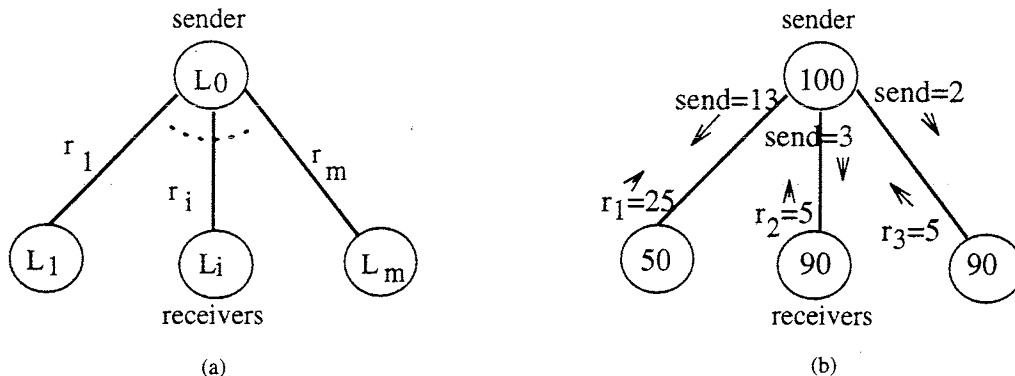

Fig. 8. Load request $r_i = 1, \ldots, m$ from sender (a) and a transfer example (b).

Leiss and Reddy [25] investigated convergence of the iterative load balancing algorithm without limit cycles (indefinite repeated load transfer patterns). Let an *H-neighborhood* denote the neighbors of a processor within a distance of H and τ denote a load threshold value. If elements are taken as load units, then $\tau = 1$. Also, let d be the *diameter* (the maximum of all the shortest paths between any two nodes) of the processor graph, which, in the present case, is G_P. Finally, define an *H-neighborhood imbalance* at time t as the variance

$$GIMB_H^t = \sum_{j \in S_p} (L_j - \alpha)^2.$$

Here S_p is the processor set for the H-neighborhood, and α is the H-neighborhood average load value. Leiss and Reddy show that

(1) after a rebalancing iteration $GIMB_H^{t+1} < GIMB_H^t$ and
(2) after balancing terminates, the maximum system imbalance is bounded by $\lceil d\tau/2H \rceil$.

Their first result implies that the imbalance in the *neighborhood* and *not* necessarily in the whole system reaches a minimum. The second result states that loading can be severely imbalanced even if it is neighborhood-balanced. A worst-case example involves a configuration with P processors forming a one-dimensional chain with each having a load that differs from its neighbor by unity, i.e., a load ramp. If $H = \tau = 1$, then, since $d = P$, the imbalance after termination of the algorithm will be $\frac{1}{2}P$. Increasing H to $\frac{1}{2}P$ produces a global system balance; however, it requires *each* of the P processors to send messages to the $\frac{1}{2}P$ H-neighbors and is, thus, impractical. In general, choosing $H > 1$ will increase the communication volume and reduce the efficiency of the iterative balancing algorithm.

To avoid this problem with Leiss and Reddy's [25] approach while keeping $H = 1$, two modifications are made to handle the case when the load difference between neighboring processors is τ. First, instead of considering $L_0 - L_i = \tau$ as balanced, this procedure (and tiling) exchanges the excess load (3a) even when $GIMB_H^{t+1} \leqslant GIMB_H^t$. Second, the previous exchange is stored to ensure that excess load is not transferred back to the original processor, thus, preventing period two limit cycles. This modification does not, however, avoid cycles of period greater than two.

Example 3.1. The iterative redistribution heuristic was tested on a MasPar MP-2 system which has a torus-connected architecture and an SIMD style of computation. Up to 2048 processors were used in the test cases with each processor having 64K bytes of memory. The MasPar system provides two types of communication mechanisms. The *xnet* mechanism provides a fast eight-way communication between processors arranged in a mesh topology. The *router* provides a slower general-purpose communication among any pair of processors. Since our applications involve unstructured meshes with curved boundaries, the communication requirements between mapped partitions are irregular. Hence, the slower router communication had to be used.

Four test cases involving meshes on square and irregular regions were run. Starting with a coarse mesh, orthogonal recursive bisection [6] was used to get an initial partition. The partitioned mesh was mapped onto the processors and refined selectively in parallel to create imbalanced processor loads. The square mesh was refined in one corner to create a "plateau" of high loading. As neighboring load transfers progress, the plateau evolves to the difficult ramp load distribution. In Table 3, we show various statistics for the test cases. In square1, a small mesh with 16 processors was employed (cf.

Table 3
Test cases for Example 3.1 and statistics before and after convergence to load balance

Test	No. of elements	No. of processors	Avg. elements per processor	Load (Min, Max)		Max. boundary edges		No. of iterations	Time (secs)
				Before	After	Before	After		
square1	164	16	10.25	2, 32	7, 11	16	12	25	9.7
square2	32904	2048	16.06	16, 52	16, 18	24	21	63	33.6
square3	33300	2048	16.25	16, 52	16, 18	24	25	398	214.9
curved	1008	32	31.5	18, 47	31, 32	22	25	25	12.3

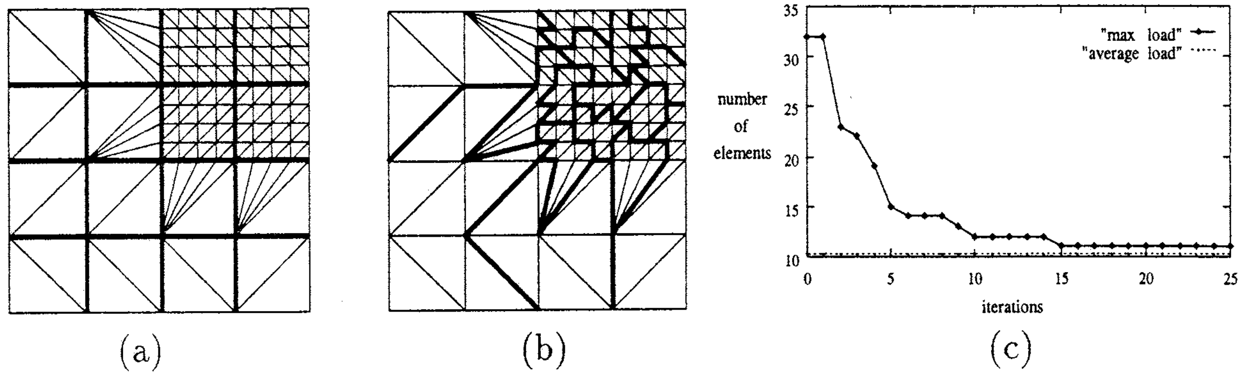

Fig. 9. Test square1 of Example 3.1: unbalanced load after mesh refinement (a), after redistribution (b), and convergence history (c).

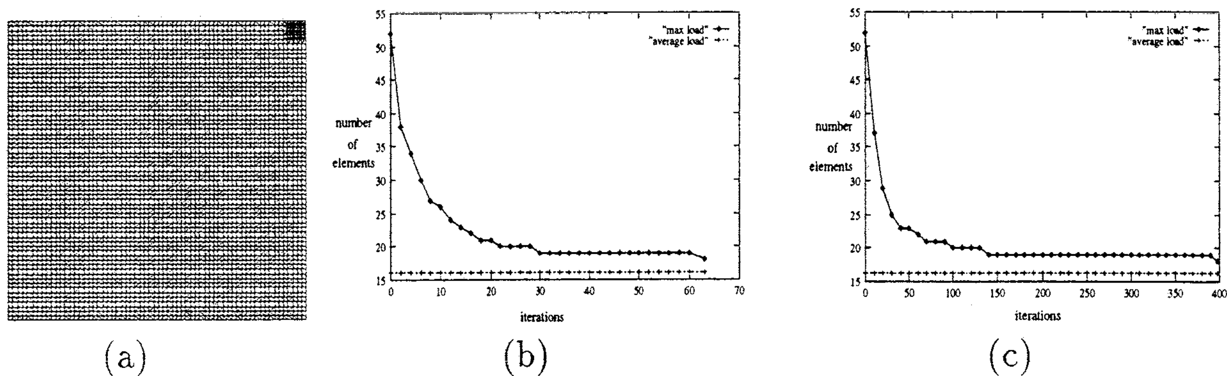

Fig. 10. The mesh for tests square2 and square3 (a) of Example 3.1, and convergence histories (b) and (c), respectively.

Fig. 9). In square2 and square3, 2048 processors were employed with refinement occurring in 4 and 16 processors, respectively, of the upper right corner of the mesh as shown in Fig. 10(a). The final test involved a highly unstructured mesh with a curved boundary (Fig. 11).

In Table 3, we list the average number of elements per processor, the maximum and minimum loads, and the maximum number of edges located on the partition boundaries before and after redistribution has been run until it converged to an optimal balance. Tests square1, square2, and curved show rapid convergence. Test square3, on the other hand, shows slow convergence even though the number of elements and the maximum imbalance is similar to the square2 test. Since a ramp evolves during redistribution, only a small load can be transferred from the highly loaded to the less loaded processors. In the worst case of a one-dimensional ramp with a unit

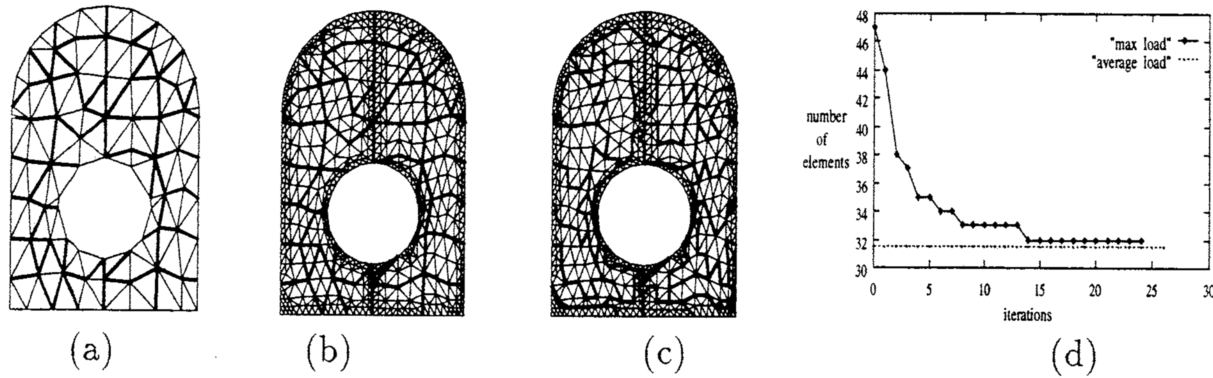

Fig. 11. Test curved of Example 3.1: initial coarse mesh partitioned by orthogonal recursive bisection (a), unbalanced mesh after refinement (b), balanced mesh after redistribution (c), and convergence history(d).

Table 4
Iterations and CPU times to achieve various percent reductions in imbalance for Example 3.1

	Percent reduction in imbalance					
	Iterations			Time (secs)		
Test	50%	75%	90%	50%	75%	90%
square1	4	5	10	5.2	5.7	7.6
square2	4	11	30	7.1	12.7	21.9
square3	13	29	98	28.6	47.7	92.8
curved	2	4	9	4.4	6.4	8.9

load difference between neighboring processors, only one element can be transferred per iteration. Hence, the larger the excess load to be migrated from the plateau, the slower the convergence of the redistribution algorithm. The average load and the distance the elements have to travel is approximately the same in both the square2 and square3 cases. Since square3 has four times the excess load of square2, we expect the number of iterations of square3 to be approximately four times that of square2. The convergence histories, shown in Figs. 10(b) and 10(c), indicate this.

The number of boundary edges, which represents the communications volume of a partition, does not necessarily decrease after redistribution. Whereas tests square1 and square2 show a slight reduction, square2 and curved show an increase in the number of boundary edges after balancing. Hence, even though the redistribution algorithm reduces the load imbalance, the selection criteria used for element migration does not guarantee a reduction of communication costs.

The convergence histories, shown in Figs. 9–11, exhibit a sharp decline in the imbalance within the first few iterations and a slower rate in subsequent iterations. We further demonstrate the performance of the procedure by listing, in Table 4, the number of iterations and the CPU time required for a 50%, 75%, and 90% reduction in the original imbalance. Since higher percentage reductions require far more iterations, a trade-off can be established between the time to do a computation with an imbalanced load and the time needed to achieve further reductions in imbalance. As a result, the redistribution process can be halted prior to convergence.

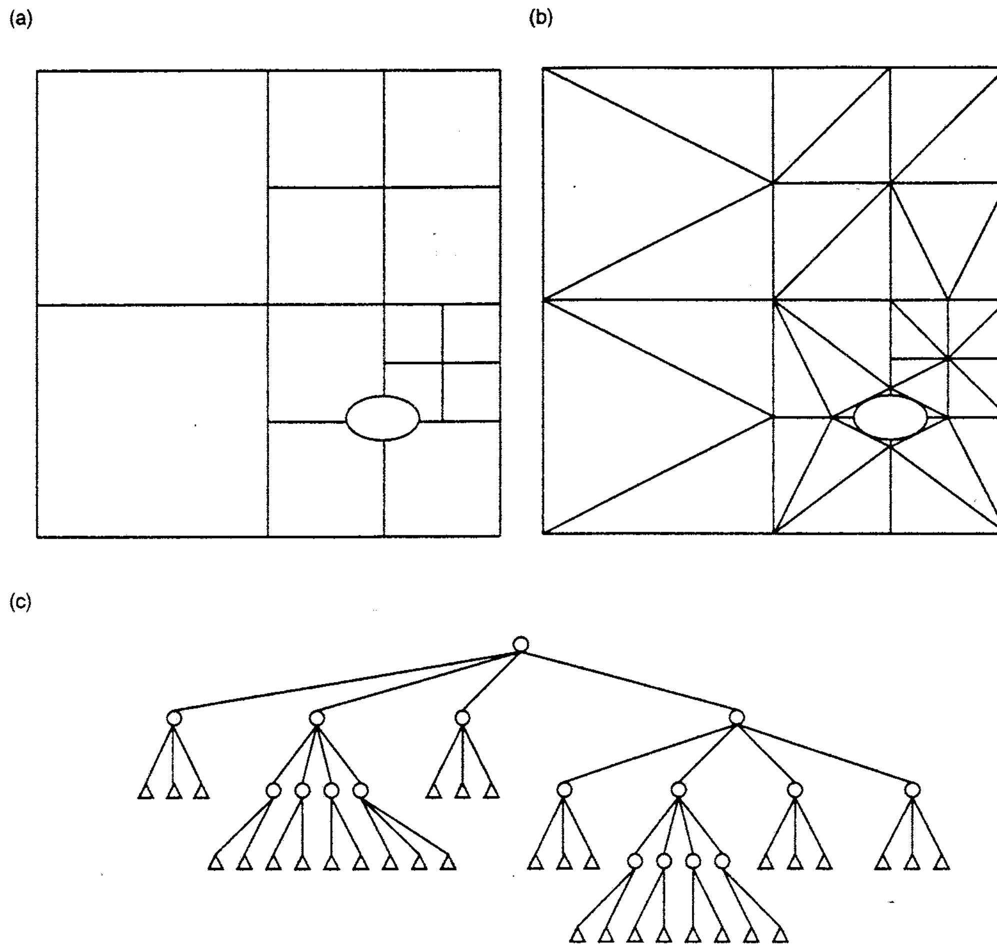

Fig. 12. A quadtree representation of the flow field surrounding an object (a), division of terminal quadrants into triangular elements (b), and the quadtree structure (c).

4. Octree-based partitioning

Special data structures used by the adaptive solution strategy may be exploited to improve parallel performance relative to the general-purpose partitioning techniques described in the previous two sections. In particular, we propose a tree-based partitioning algorithm that uses the hierarchical structure of octree-derived unstructured meshes to distribute elemental data while reducing information exchange between processors. An octree-based mesh generator [29] recursively subdivides an embedding of the problem domain in a cubic universe into eight octants wherever more resolution is required. Octant bisection is initially based on geometric features of the domain but solution-based criteria are introduced during adaptive h-refinement. Finite element meshes of tetrahedral elements are generated from the octree by subdividing terminal octants.

In Fig. 12, we illustrate a tree and mesh for a two-dimensional flow domain containing a small object. The root of the tree represents the entire domain (Fig. 12(c)). The domain is recursively quartered until an adequate resolution of the object is obtained (Fig. 12(a)). A smooth gradation is maintained by enforcing a one-level maximum difference between adjacent quadrants. After obtaining adequate resolution, leaf quadrants are subdivided into triangular elements that are pointed to by leaf nodes of the tree (Figs. 12(b) and (c)). Quadrants containing the object are decomposed

using the geometry of the object. Smoothing [29], which normally follows element creation, is not shown.

Our tree-based procedure creates a one-dimensional ordering of the octree and partitions it into nearly equal-sized segments based on tree topology. Initially, a cost metric is determined for all subtrees. Cost is currently defined as the number of elements within a subtree. For a leaf octant, this would simply be the number of tetrahedra associated with it. P-refinement would likely use the elemental degrees of freedom as a cost metric. If the solution algorithm employs spatially-dependent time steps then, typically, a greater number of smaller time steps must be taken on smaller elements and this must also be reflected in the subtree cost. In any event, appropriate costs may be determined by a traversal of the octree.

With the total cost available from the initial phase and the number of partitions prescribed, the optimal partition size is also known. Partitions, consisting of a set of octants that are each the root of a subtree, are determined by a truncated depth-first search. Thus, octree nodes are visited and subtrees are accumulated into successive partitions. The subtree rooted at a visited node is added to the current partition if it fits. If it exceeds the optimal partition size, a decision must be made to add it or to continue the traversal. In the latter case, the traversal recursively visits the offspring of the node and may divide the subtree among two or more partitions. The decision to add the subtree or continue the traversal is based on the amount by which the optimal partition size is exceeded. A small excess may not justify an extensive search and may be used to balance another partition that is slightly undersized. When the excess is too large to justify inclusion in the current partition, and the node is either terminal or sufficiently deep in the tree, the partition is closed and subsequent nodes are added to the next partition.

This partitioning method requires storage for nonterminal nodes of the tree that would normally not be necessary since they contain no solution data. However, only minimal storage costs are incurred since information is only required for tree connectivity and the cost metric. For this modest investment, we obtain a partitioning algorithm that only requires $O(J)$ serial steps.

Partitions formed by this procedure do not necessarily form a single connected component; however, the octree decomposition and the orderly tree traversal tend to group neighboring subtrees together. Furthermore, a single connected component is added to the partition whenever a subtree fits within the partition.

A tree partitioning example is illustrated in Fig. 13. All subtree costs are determined by a post-order traversal of the tree. The partition creation traversal starts at the root, Node 0 (Fig. 13(a)). The node currently under investigation is identified by a double circle. The cost of the root exceeds the optimal partition cost, so the traversal descends to Node 1 (Fig. 13(b)). As shown, the cost of the subtree rooted at Node 1 is smaller than the optimal partition size and, hence, this subtree is added to the current partition, P_0, and the traversal continues at Node 2 (Fig. 13(c)). The cost of the subtree rooted at Node 2 is too large to add to P_0, so the algorithm descends to an offspring of Node 2 (Fig. 13(d)). Assuming Node 4 fits in P_0, the traversal continues with the next offspring of Node 2 (Fig. 13(e)). Node 5 is a terminal node whose cost is larger than the available space in P_0, so the decision is made to close P_0 and begin a new partition, P_1. As shown (Fig. 13(f)), Node 5 is very expensive, and when the traversal is continued at Node 3, P_1 must be closed and work continues with partition P_2.

Our partitioning algorithm is similar in spirit to that of Farhat's automatic finite element decomposer [15]. Farhat essentially performs a breadth-first search of the mesh, accumulating elements into

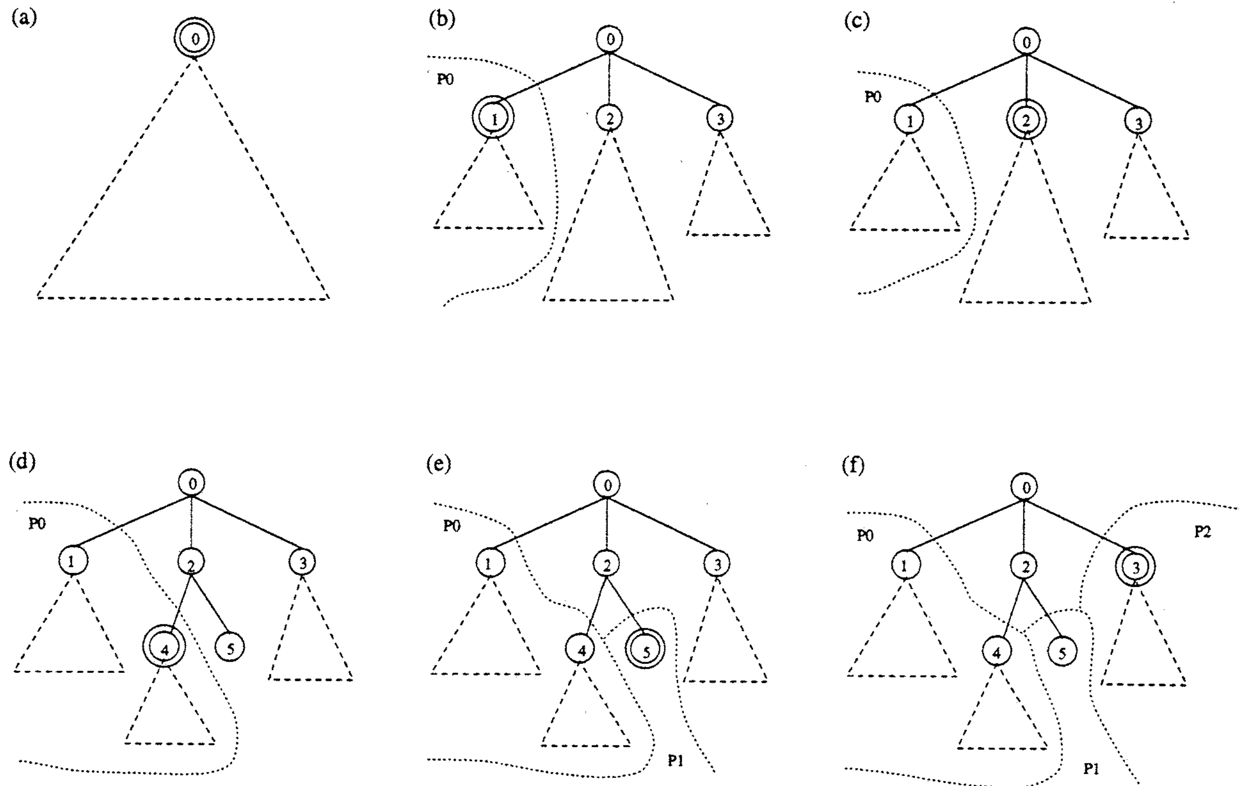

Fig. 13. A tree partitioning example. The partition creation traversal starts at the root (a). Nodes are visited and added to the current partition if their subtree fits (b). When a subtree is too large to fit (c), the traversal descends into the subtree (d). Alternatively, the partition is closed and work begins on a new partition (e). The process continues until the traversal is complete (f).

partitions. Subdomains are accumulated during the search, and each is closed in turn when its cardinality reaches the number of elements divided by the number of processors. This is directly analogous to closing partitions in the tree algorithm. Likewise, subdomains (partitions) may be multiply connected. The similarity, however, ends with the hierarchical nature of the tree traversal. With the large-scale information available at each tree node, larger and more compact spatial regions may be added to a partition; thereby, reducing the likelihood of creating thin partitions having large surface areas.

The tree traversal partitioning algorithm may easily be extended for use in a parallel adaptive environment. An initial partitioning is made using the serial algorithm described above. When a new partitioning is needed due to adaptive enrichment, each processor computes its subtree costs using the serial traversal algorithm within its domain. This step requires no interprocessor communication. An inexpensive parallel prefix operation may be performed on the processor-subtree totals to obtain a global cost structure. This information enables a processor to determine where its local tree traversal is located in the global traversal.

Now, following the serial procedure, each processor may traverse its subtrees to create partitions. A processor determines the partition number to start working on based on the total cost of processors preceding it. Each processor starts counting with this prefix cost and traverses its subtrees adding the cost of each visited node to this value. Partitions end near cost multiples of N/P, where N is the total cost and P is the number of processors. Exceeding a multiple of N/P during the traversal is analogous to exceeding the optimal partition size in the serial case and the same criteria may be

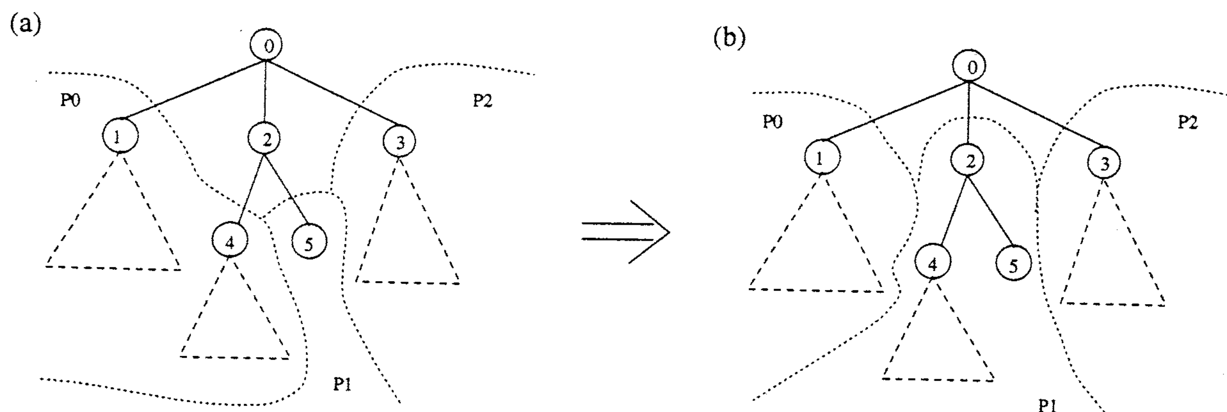

Fig. 14. Iterative rebalancing of tree-based partitions. The subtree rooted at Node 4 (a) has been shifted from P_0 to P_1 (b) to relieve a load imbalance. The new root of P_1 is Node 2, the common parent of Nodes 4 and 5.

used to determine where to end partitions. When all processors finish their traversals, each subtree (and its associated data) is assigned to a new partition and may be migrated to its new location. Migration may be done using global communication; however, on some architectures, it may be more efficient to move data via simultaneous processor shift operations. This linear communication pattern is possible due to the one-dimensional nature of the partitioning traversal.

While the cost of computing the new partition is small, the cost of data movement is likely to be high and it would be desirable to amortize this by tolerating small imbalances. A strategy to delay the need for complete repartitioning would simply shift partition boundaries, thus, migrating subtrees from a processor P_n to its neighbors P_{n-1} and P_{n+1}. If, for example, processor P_n seeks to transfer cost r to P_{n-1}, it simply traverses its subtrees accumulating their costs until it reaches r. The nodes visited comprise a subtree which may be transferred to P_{n-1} and which is contiguous in the traversal with the subtrees in P_{n-1}. Likewise, if P_n desires to transfer work to P_{n+1}, the reverse traversal could remove a subtree from the trailing part of P_n. Consider, as an example, the subtree rooted at Node 4 of Fig. 14(a) and suppose that its cost has increased through enrichment. In Fig. 14(b), we show how the partition boundary may be shifted to move the subtree rooted at Node 4 to partition P_1. The amount of data to be moved between processors may utilize the tiling or pairwise exchange procedures discussed in Sections 2 and 3, respectively.

Example 4.1. Performance results obtained by applying the tree-based partitioning algorithm to various three-dimensional irregular meshes are presented in Figs. 15 and 16. The meshes were generated by the finite octree mesh generator [29]. "Airplane" is a 182K-element mesh of the volume surrounding a simple airplane [13]. "Copter" is a 242K-element mesh of the body of a helicopter [13]. "Onera", "Onera2" and "Onera3" are 16K-, 70K-, and 293K-element meshes, respectively, of the space surrounding a swept, untwisted Onera-M6 wing which has been refined to resolve a bow shock [14]. "Cone" is a 139K-element mesh of the space around a cone having a 10° half-angle and which also has been refined to resolve a shock.

The quality of a partition has been measured in Fig. 15 as the percent of element faces lying on inter-partition boundaries relative to the total number of faces of the mesh. The graph displays these percentages as a function of the optimal partition size. In all cases the cost variance between the partitions is very small (about as small as the maximum cost of a leaf octant). The proportion

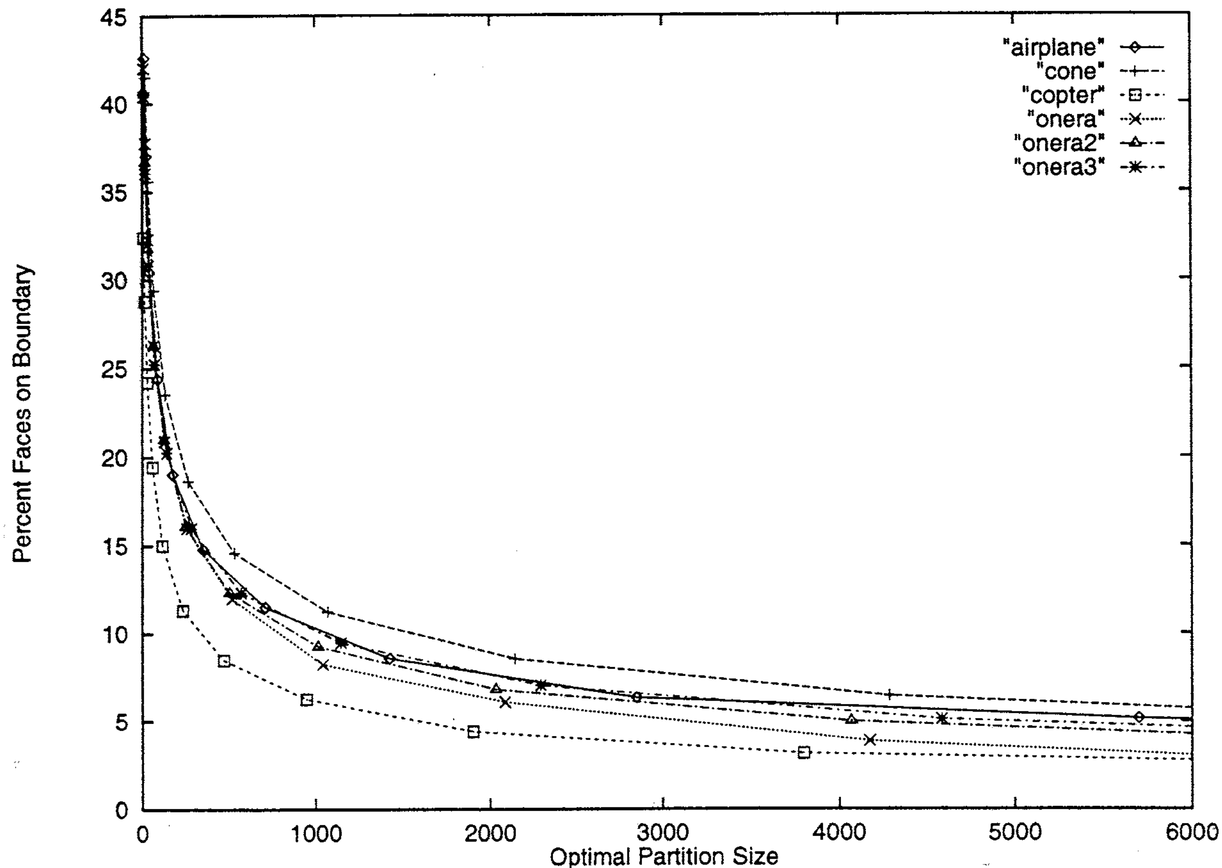

Fig. 15. Global performance measure of the tree partitioning algorithm on the five meshes of Example 4.1.

is, in a sense, the total surface area that partitions hold in common. Smaller ratios require less communication relative to the amount of local data access. This measure is closely related to the number of "cuts" that the partition creates [20,24,30]; however, we have chosen to normalize by the total number of faces in order to compare partition quality over a wide range of mesh sizes and number of partitions.

The data of Fig. 15 show the expected behavior that the interface proportion approaches zero as the partition size increases (due to the number of partitions approaching unity). Conversely, as the optimal partition size approaches unity (due to number of partitions approaching the number of elements), the interface proportion goes to unity. The interface proportion is less than 12% when the partition size exceeds 1000 for these meshes. Interfaces drop to below 9% and 8%, respectively, for partition sizes of 2000 and 3000. This performance is comparable to recursive spectral bisection [23] but requires much less computation ($O(J)$ as opposed to $O(J^2)$ [26]).

The best performance occurred with the helicopter mesh, which was the only mesh of a solid object (as opposed to a flow field surrounding an object). The solid can easily be cut along its major axis to produce partitions with small inter-partition boundaries, and was included for generality. While still reasonable, the lowest performance occurred with the cone mesh. This is most likely due to the model and shock region being conically shaped, which is somewhat at odds with the cubic octree decomposition.

In general, inter-partition boundaries should be less than 10%, indicating partition sizes of 2000 or more. This minimum partition size is not an excessive constraint, since a typical three-dimensional

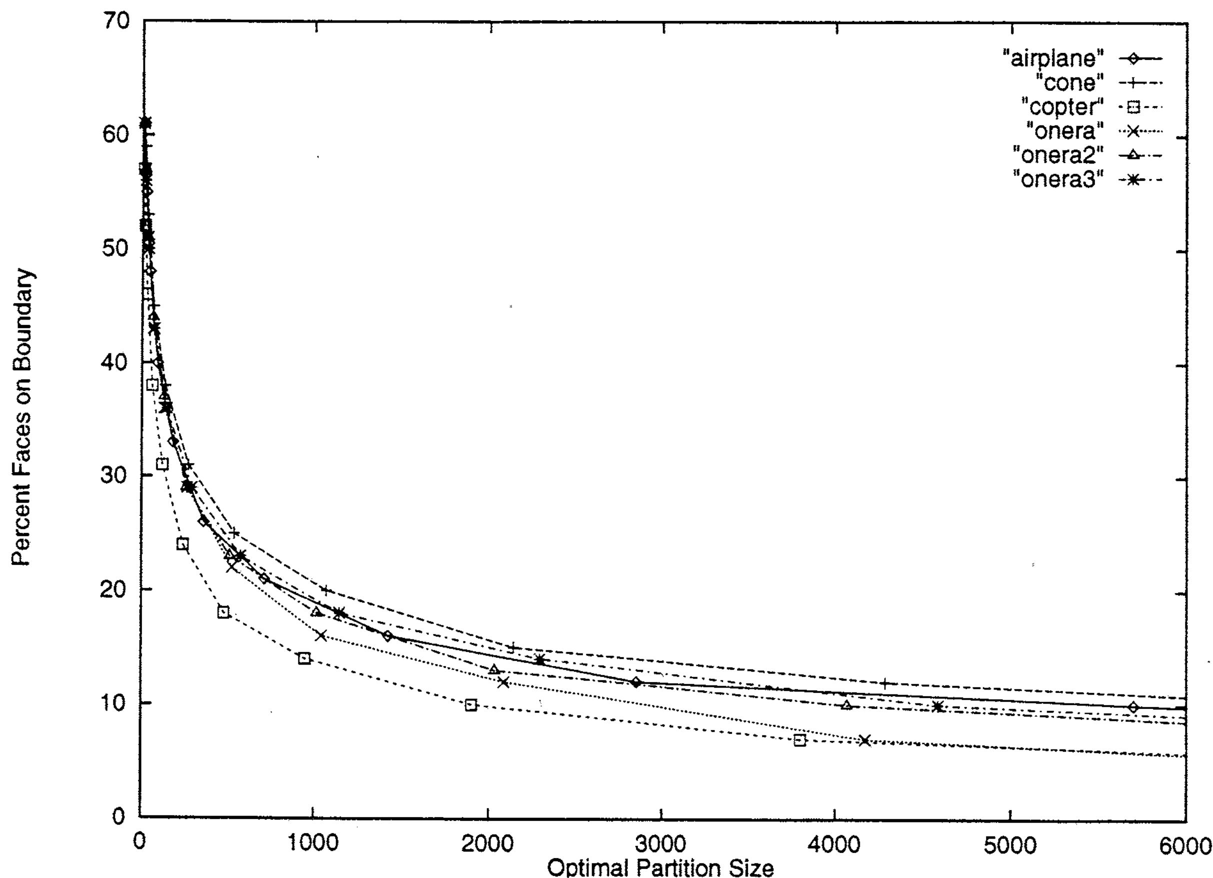

Fig. 16. Local performance of the tree partitioning algorithm on the five meshes of Example 4.1.

problem employing a two-million-element mesh being solved on a 1024-processor computer would have about 2000 elements per processing element.

Another measure of partition quality is the percent of a partition's element faces lying on inter-partition boundaries relative to the total number of faces in that partition. This is shown in Fig. 16. This number is, in a sense, the ratio of surface area to volume of a partition. For our example meshes, this measure was below 22% and 18%, respectively, for partition sizes of 1000 and 1500.

In Fig. 17, we show partitions of several of the meshes introduced in this example. The partitions exhibit a blocked structure; however, several partitions of the airplane mesh appear to be made up of disconnected components. It is possible, though unlikely, that a partition not be connected. However, in this case the partitions only appear to be disconnected because the display is a two-dimensional slice through the three-dimensional domain.

Example 4.2. In Fig. 18, we show the pressure contours of a Mach 2 Euler flow past the "Cone" mesh of Example 4.1. The solution employs the discontinuous finite element scheme [5, 8–10] with van Leer's flux vector splitting [32] and was computed on a Thinking Machines CM-5 computer with 128 processors. Several h-refinement steps were required to yield this mesh. At each iteration, elements were marked with the desired tree level (either larger for refinement, or smaller for coarsening), and a new global mesh created to satisfy these constraints. The shock surface and pressure contours are shown above; below are examples of how the mesh may be partitioned for 16 and 32 processor machines.

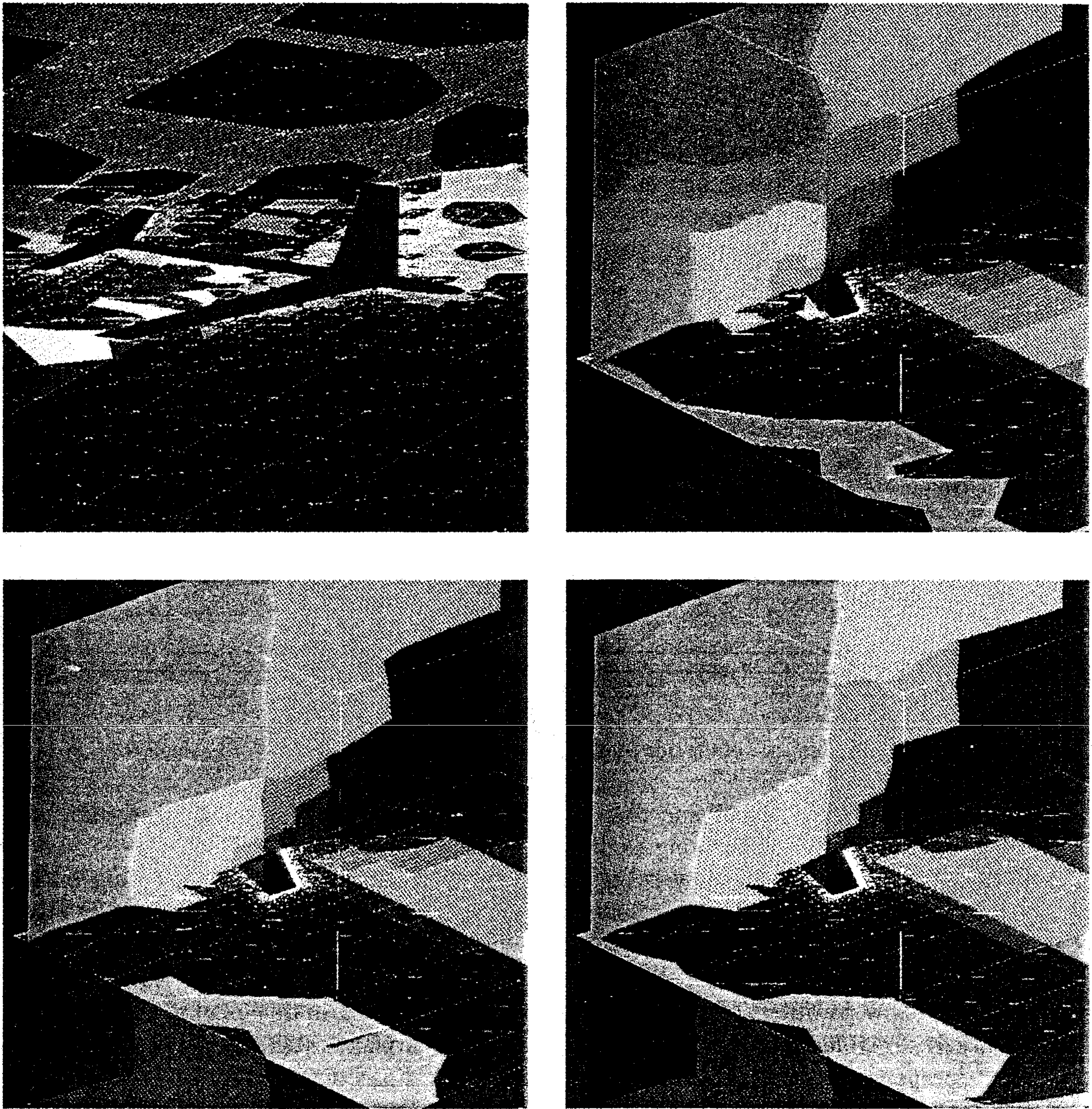

Fig. 17. The airplane mesh, and three refinements of the Onera M6 wing mesh, all divided into 32 partitions. Colors denote partition membership.

5. Discussion

We have described partitioning strategies that are appropriate for load balancing parallel distributed-memory computation with adaptive h- and p-refinement techniques for partial differential equations. Tiling performs local balancing within overlapping neighborhoods and we demonstrate its effective performance by using it with a local finite element technique [1, 8–11] to solve two-dimensional

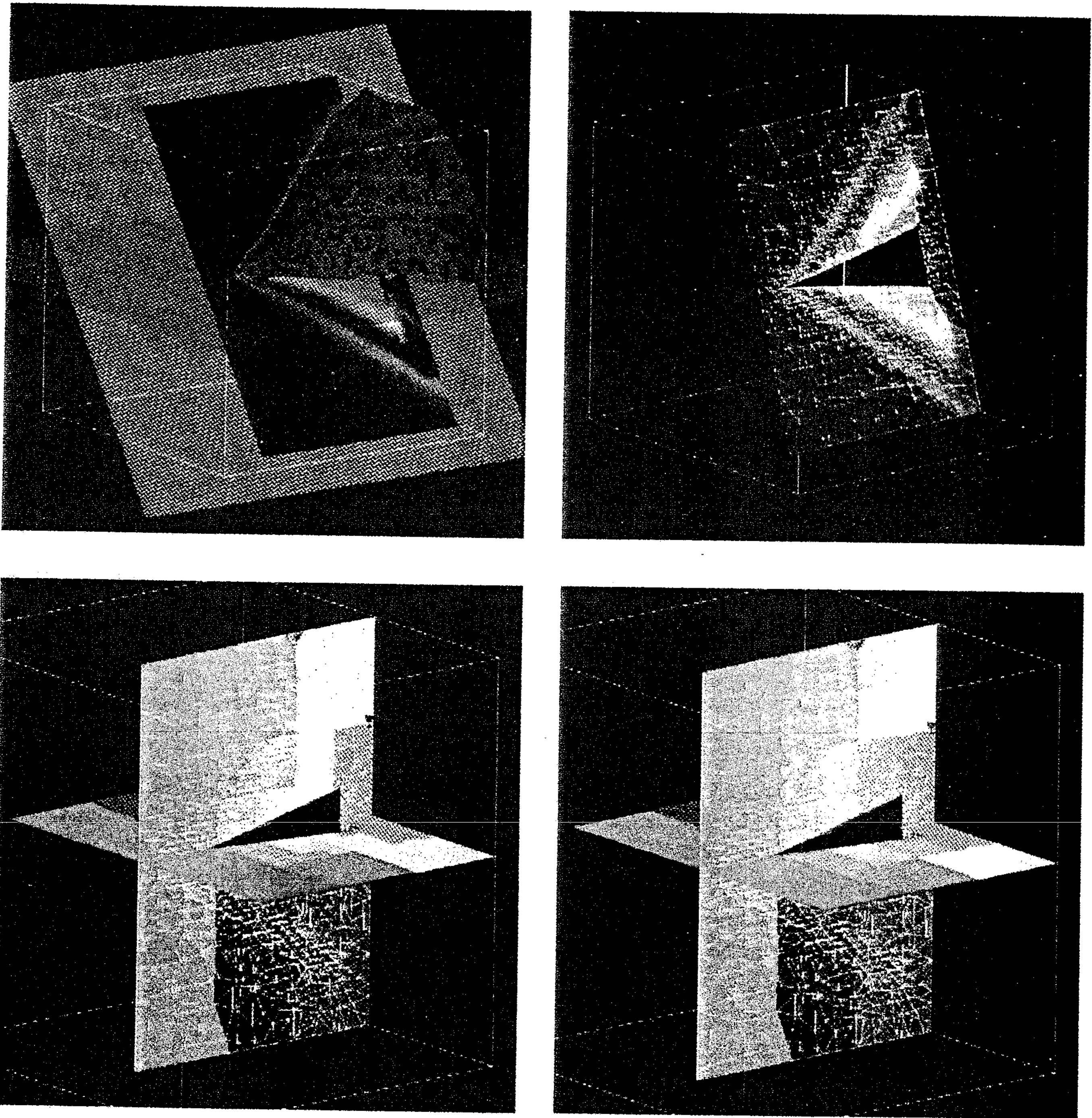

Fig. 18. Shock surface and pressure contours found when computing the Mach 2 flow past a cone having a half-angle of 10° (top). Partitions of the mesh into 16 (left) and 32 (right) pieces (bottom). Colors in the top figures denote pressure levels while those of the bottom figures denote partition membership.

systems of conservation laws by adaptive h- and p-refinement. The next step involves combining the h- and p-refinement procedures to develop an hp-refinement algorithm. When used with hyperbolic systems, h-refinement would be used near discontinuities while p-refinement would be used in regions of smooth flow. Appropriate combinations of h- and p-refinement at discontinuities, such as the 1:15 ratio used with elliptic problems [18], may, however, provide superior performance.

Pairwise exchange extends tiling to unstructured meshes and provides a means for controlling communications volume through bidirectional migration. This procedure is being implemented on an MIMD computer and development and testing will continue using three-dimensional problems in biomechanics and compressible flow as examples.

Octree-based partitioning provides an effective and efficient partitioning strategy that may either be used in conjunction with octree mesh generation [29] or on its own, by the inexpensive construction of an octree from an existing mesh. Parallel partitioning techniques and incremental migration strategies for use with adaptivity are being developed. It should also be possible to combine octree partitioning with other strategies to provide additional control of communications volume. For example, octree decomposition could be used to provide a course partition that could be continued with recursive spectral bisection [26]. Recursive spectral bisection at terminal tree nodes may be parallelized [24], it costs less than a global application because of the smaller partition domains and its nonlinear complexity, and it is more effective on smaller regions [23].

Theoretical issues associated with each algorithm must be investigated. Convergence under iteration of either the tiling or pairwise exchange migration strategies must be established, as must the avoidance of limit cycles.

Comparisons between methods are essential; however, the three techniques described herein are under development and are not finished products. Software is being developed with portability in mind and, with aims of unifying our research effort and performing explicit comparisons, we find ourselves heading for a message passing environment using the Chameleon protocol [17].

Acknowledgements

This research was supported by the Air Force Office of Scientific Research under Grant F49620-94-1-0200; the U.S. Army Research Office under Contracts DAAL03-91-G-0215 and DAALO3-89-C-0038 with the University of Minnesota Army High Performance Computing Research Center (AHPCRC) and the DoD Shared Resource Center at the AHPCRC; the Massively Parallel Computation Research Laboratory at Sandia National Laboratories, operated for the U.S. Department of Energy under Contract DE-AC04-76DP00789, Research Agreement AD-9585; an ARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland; and the Grumman Corporate Research Center.

We also wish to thank Thinking Machines Corporation, and in particular Zdeněk Johan and Kapil Mathur, for their assistance with the CM-5.

References

[1] S. Adjerid, M. Aiffa and J. E. Flaherty, High-order finite element methods for singularly perturbed elliptic and parabolic systems, *SIAM J. Appl. Math.* (to appear).

[2] S. Adjerid, J.E. Flaherty, P. Moore and Y. Wang, High-order adaptive methods for parabolic systems, *Phys. D* 60 (1992) 94–111.

[3] D.C. Arney and J.E. Flaherty, An adaptive mesh moving and local refinement method for time-dependent partial differential equations, *ACM Trans. Math. Software* 16 (1990) 48–71.

[4] I. Babuška, The *p*- and *hp*-versions of the finite element method. The state of the art, in: *Finite Elements: Theory and Applications* (Springer-Verlag, New York, 1988).

[5] R. Biswas, K.D. Devine and J.E. Flaherty, Parallel, adaptive finite element methods for conservation laws, *Appl. Numer. Math.* 14 (1994) 255–284.
[6] M.J. Berger and S.H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, *IEEE Trans. Comput.* 36 (5) (1987) 570–580.
[7] M.J. Berger and J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, *J. Comput. Phys.* 53 (1984) 484–512.
[8] B. Cockburn, S.-Y. Lin and C.-W. Shu, TVB Runge–Kutta local projection discontinuous Galerkin finite element method for conservation laws III: one-dimensional systems, *J. Comput. Phys.* 84 (1989) 90–113.
[9] B. Cockburn, S.-Y. Lin and C.-W. Shu, TVB Runge–Kutta local projection discontinuous Galerkin finite element method for conservation laws IV: the multidimensional case, *Math. Comp.* 54 (1990) 545–581.
[10] B. Cockburn and C.-W. Shu, TVB Runge–Kutta local projection discontinuous Galerkin finite element method for conservation laws II: general framework, *Math. Comp.* 52 (1989) 411–435.
[11] K.D. Devine, J.E. Flaherty, R.M. Loy and S.R. Wheat, Parallel partitioning strategies for the adaptive solution of conservation laws, Tech. Report 94-1, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY (1994).
[12] P. Devloo, J.T. Oden and P. Pattani, An *h-p* adaptive finite element method for the numerical simulation of compressible flow, *Comput. Methods Appl. Mech. Engrg.* 70 (1988) 203–235.
[13] S. Dey, Personal communication (1993).
[14] M. Dindar, Personal communication (1993).
[15] C. Farhat, A simple and efficient automatic FEM domain decomposer, *Comput. & Structures* 28 (5) (1988) 579–602.
[16] M. Fiedler, Algebraic connectivity of graphs, *Czechoslovak Math. J.* 23 (1973) 298–305.
[17] W. Gropp and B. Smith, Users manual for the Chameleon parallel programming tools, Tech. Report ANL-93/23, Argonne National Laboratories, Argonne, IL (1993).
[18] W. Gui and I. Babuška, The *h*-, *p*- and *hp*-versions of the finite element method in one dimension, Part I: the error analysis of the *p* version; Part II: the error analysis of the *h* and *hp* versions; Part III: the adaptive *hp* version, *Numer. Math.* (to appear).
[19] S.W. Hammond, Mapping unstructured grid computations to massively parallel computers, Ph.D. Dissertation, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY (1991).
[20] B. Hendrickson and R. Leland, An improved spectral graph partitioning algorithm for mapping parallel computations, Tech. Report SAND92-1460, Sandia National Laboratories, Albuquerque, NM (1992).
[21] B. Hendrickson and R. Leland, Multidimensional spectral load balancing, Tech. Report SAND93-0074, Sandia National Laboratories, Albuquerque, NM (1993).
[22] J. Jaja, *An Introduction to Parallel Algorithms* (Addison-Wesley, Reading, MA, 1992).
[23] Z. Johan, Personal communication (1993).
[24] Z. Johan, K. Mathur and S.L. Johnsson, An efficient communication strategy for finite element methods on the Connection Machine CM-5 system, Tech. Report No. 256, Thinking Machines Corp., Cambridge, MA (1993).
[25] E. Leiss and H.N. Reddy, Distributed load balancing: design and performance analysis, *W.M. Keck Research Computation Laboratory* 5 (1989) 205–270.
[26] A. Pothen, H.D. Simon and K.-P. Liou, Partitioning sparse matrices with eigenvectors of graphs, *SIAM J. Matrix Anal. Appl.* 11 (1990) 430–452.
[27] E. Rank and I. Babuška, An expert system for the optimal mesh design in the *hp*-version of the finite element method, *Internat. J. Numer. Methods Engrg.* 24 (1987) 2087–2106.
[28] H.N. Reddy, On Load Balancing, Ph.D. Dissertation, Department Computer Science, University of Houston, Houston, TX (1989).
[29] M.S. Shephard and M.K. Georges, Automatic three-dimensional mesh generation by the finite octree technique, *Internat. J. Numer. Methods Engrg.* 32 (4) (1991) 709–749.
[30] H.D. Simon, Partitioning of unstructured problems for parallel processing, *Comput. Systems Engrg.* 2 (1991) 135–148.
[31] B.K. Szymanski and A. Minczuk, A representation of a distribution power network graph, *Archiwum Elektrotechniki* 27 (2) (1978) 367–380.
[32] B. Van Leer, Flux vector splitting for the Euler equations, ICASE Report No. 82-30, ICASE, NASA Langley Research Center, Hampton, VA (1982).

[33] C. Walshaw and M. Berzins, Dynamic load-balancing for PDE solvers on adaptive unstructured meshes, Preprint, School of Computer Studies, Tech. Report, University of Leeds, Leeds (1992).

[34] S.R. Wheat, A fine grained data migration approach to application load balancing on MP MIMD machines, Ph.D. Dissertation, Department Computer Science, University of New Mexico, Albuquerque, NM (1992).

[35] S.R. Wheat, K.D. Devine and A.B. Maccabe, Experience with automatic, dynamic load balancing and adaptive finite element computation, in: H. El-Rewini and B.D. Shriver, eds., *Proceedings 27th Hawaii International Conference on System Sciences* 2 (1994) 463–472.

[36] V.G. Vizing, On an estimate of a chromatic class of a multigraph, in: *Proceedings Third Siberian Conference on Mathematics and Mechanics*, Tomsk (1964).