INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2004; **46**:37–57 (DOI: 10.1002/fld.674)

Parallel adaptive refinement for unsteady flow calculations on 3D unstructured grids

Jacob Waltz*,†

Applied Physics Division, Los Alamos National Laboratory, Los Alamos, NM 87545, U.S.A.

SUMMARY

A parallel adaptive refinement algorithm for three-dimensional unstructured grids is presented. The algorithm is based on an hierarchical *h*-refinement/derefinement scheme for tetrahedral elements. The algorithm has been fully parallelized for shared-memory platforms via a domain decomposition of the mesh at the algebraic level. The effectiveness of the procedure is demonstrated with applications which involve unsteady compressible fluid flow. A parallel speedup study of the algorithm also is included. Published in 2004 by John Wiley & Sons, Ltd.

KEY WORDS: adaptive refinement; unstructured grids; computational fluid dynamics; parallel computing

1. INTRODUCTION

1.1. Background and motivation

Adaptive refinement for unstructured grids has enjoyed a long and successful history, not just in Computational Fluid Dynamics (CFD) [1–4], but in Computational Physics and Engineering as a whole. That unstructured grids can be refined and derefined with no changes to the underlying solver is in fact one of their biggest advantages over structured grid approaches.

As both problem size and computational power have increased dramatically over the last decade, significant amounts of time and effort have been devoted to the parallelization of unstructured CFD solvers. Parallelization of such codes for either distributed- or shared-memory platforms is at this point rather straightforward conceptually, although the actual implementation may be somewhat involved.

In contrast, parallelization of *adaptive* unstructured CFD solvers suitable for 3D unsteady problems is at a less mature stage. Numerous efforts have been made over the last decade in the development of parallel adaptive 3D codes with varying results [5–11]. In some cases, the resultant algorithm is only partially parallelized. For example, the flow solver and domain

*Correspondence to: J. Waltz, Applied Physics Division, MST086, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545, U.S.A.
†E-mail: jwaltz@lanl.gov

Published online 14 July 2004

Received 2 July 2003
Revised 10 November 2003

decomposition may be parallel, but the actual mesh adaption is serial. In other cases, the algorithm is fully parallel but has been demonstrated only for test problems, rather than production–quality calculations, so that questions about the ultimate practicality of the algorithm remain. While a small number of these efforts have indeed been quite successful, application of fully parallel adaptive CFD codes to large unsteady 3D problems has yet to become commonplace.

The importance of parallel mesh adaption for this class of problems is best demonstrated with a simple example. First, recall Amdahl's law, which states that given N processors and a code with parallel fraction p, the speedup S is given by

$$S = \left[\frac{p}{N} + (1 - p)\right]^{-1} \tag{1}$$

For most unsteady 3D calculations, numerical experience indicates that the adaption process generally consumes $\approx 10\%$ of the computational time. If one assumes that the adaption algorithm is serial, and that the remaining 90% of the computational effort is perfectly parallel, then the maximum possible speedup for the entire calculation is

$$\lim_{N \to \infty} S = 10 \tag{2}$$

Given that machines with $O(10^3)$ processors are commonplace at modern high-performance computing centres, a maximum parallel speedup of $O(10)$ falls far short of optimal resource usage. Clearly, improvements can be gained only if the mesh adaption also is parallelized.

Any discussion of parallel adaptive refinement leads quite naturally into the discussion of distributed- versus shared-memory parallelization. The main advantage of shared-memory parallelization in the context of this work is that since parallelization is performed at the loop level, the user is freed from the task of dynamic load-balancing. A drawback of the shared-memory approach is that while the development and implementation process may initially seem much easier, a simple loop-by-loop parallelization of an unstructured grid code generally will lead to poor parallel performance due to memory contention. Specifically, if multiple processors try to simultaneously update identical pieces of data, delays will occur as processors wait for one another. In addition to the memory contention problem, some loops inevitably involve operations which are inherently non-parallel, such as recursion. Any serial loops will reduce the scalability of the overall adaptive solver.

With these observations in mind, a shared-memory parallelization strategy for unstructured mesh adaption should seek to achieve two primary goals: first, memory contention must be avoided as much as possible; and second, as many individual loops as possible must be parallelized, which in turn implies that inherently non-parallel algorithms must be rewritten with alternative procedures. A high level of single-processor performance is of course desirable.

The purpose of this paper, therefore, is to present a parallel adaptive refinement algorithm which successfully realizes both goals and thereby achieves a relatively high level of parallel efficiency. The more general goal is a fully parallel adaptive simulation capability suitable for large unsteady problems in 3D. No claim is made as to whether or not a shared-memory unstructured grid method is the optimal approach for such problems. Rather, the method is presented as simply one way to achieve the stated goal.

The adaption algorithm described herein has been implemented in a general purpose Finite Element CFD code [12] which also is parallelized for shared-memory platforms. Although

the algorithm is specifically applicable to unstructured tetrahedral meshes and shared-memory platforms, the challenges involved are, at a general level, applicable to structured grids and/or distributed-memory platforms. As was indicated, significant losses in parallel speedup will occur if the entire adaptive flow solver is not parallelized. Neither the mesh type nor the machine architecture change this fact.

A related issue is that after mesh adaption occurs, a number of secondary tasks must be performed before the flow solver can resume computation of the solution. These include evaluation of derived data structures; mesh renumbering; computation of Finite Element matrices and other geometrical parameters, etc. These secondary operations also should be parallelized to maximize the scalability of the entire adaptive flow solver. Since this discussion is restricted to the adaption algorithm proper, suffice it to say that the overall parallelization approach used here can be applied successfully to these secondary operations (see for example Reference [13]).

1.2. Summary of paper

The remainder of this paper is organized as follows. Section 2 discusses issues which, while not the primary focus of this work, are integral parts of the overall adaption algorithm. Specifically, the topics of error indication, algebraic domain decomposition, solution interpolation, allowable refinement cases, and implementation are addressed. The flow solver also is briefly described. Section 3 presents the adaption algorithm proper, beginning with an outline of the various substeps followed by more detailed descriptions. Parallel performance of the adaption algorithm for a typical unsteady compressible flow calculation is discussed in Section 4. Numerical examples, including a validation case, are presented in Section 5. Lastly, concluding remarks are given in Section 6.

2. PRELIMINARY CONSIDERATIONS

2.1. Flow solver

As mentioned in the introduction, the adaption algorithm described in this paper has been implemented in a general purpose CFD code [12] which also is fully parallelized for shared-memory architectures. The solver utilizes an edge-based Finite Element formulation for linear tetrahedra. The sample calculations presented in this paper, all of which involve unsteady compressible flow, utilized an exact Riemann solver to compute the numerical fluxes for the ideal-gas Euler equations. The numerical scheme is extended to higher-order accuracy via an edge-based form of the classic MUSCL procedure [14] combined with the Van Albada limiter. Time integration is performed with a fourth-order explicit Runge–Kutta method.

2.2. Error indication

The flow solver used in this work is based on linear tetrahedral elements. The interpolation error for a given unknown will therefore be dominated by second-order derivatives of the unknown or, in more general terms, the Hessian of the unknown. Along a single edge, this quantity is approximated in terms of a finite difference of first-order directional derivatives as

$$\varepsilon^{\alpha\beta} = \tfrac{1}{2}\,|\nabla u^{\beta} \cdot \underline{h} - \nabla u^{\alpha} \cdot \underline{h}| \tag{3}$$

where α and β represent nodal indices (so that the two together represent an edge in the mesh), $\underline{h}$ is the vector which defines the edge, and u is the field variable used to indicate error. The gradients are calculated with a simple lumped-mass Galerkin method.

The above quantity is normalized by the L_∞ norm of the error distribution to yield the final error indicator:

$$\varepsilon^{\alpha\beta} = \frac{|\nabla u^\beta \cdot \underline{h} - \nabla u^\alpha \cdot \underline{h}|}{\|\nabla u^\beta \cdot \underline{h} - \nabla u^\alpha \cdot \underline{h}\|_\infty} \tag{4}$$

This final error indicator is both dimensionless and bounded between zero and one. These two properties facilitate the use of multiple indicator variables: for each edge, the error estimate for each indicator variable is calculated, and the final error estimate assigned to the edge is the maximum of the individual values for the edge in question.

An alternative normalization approach is that advocated by Löhner [2], i.e. normalization by an average first-order derivative. Numerical experience with both approaches indicates that the gradient-normalization is superior to the L_∞-normalization when flow features of widely varying strength are present. In other cases, however, the L_∞-normalization has been found to be somewhat more reliable.

Given the above edge-based estimate of the interpolation error, the adaption criterion takes the form

$$\varepsilon^{\alpha\beta} \geqslant \varepsilon_{\mathrm{H}} \rightarrow \text{refine} \tag{5}$$

$$\varepsilon^{\alpha\beta} \leqslant \varepsilon_{\mathrm{L}} \rightarrow \text{derefine} \tag{6}$$

For steady-state calculations, the critical values ε_{H} and ε_{L} are defined in terms of the mean error value over all edges:

$$\varepsilon_{\mathrm{H}} = 4\bar{\varepsilon}; \quad \varepsilon_{\mathrm{L}} = \tfrac{1}{4}\,\bar{\varepsilon} \tag{7}$$

This adaption criterion, therefore, tends to equidistribute the error [15]. For unsteady calculations, which are the primary interest here, the critical values are fixed at

$$\varepsilon_{\mathrm{H}} \approx 0.15, \quad \varepsilon_{\mathrm{L}} \approx 0.05 \tag{8}$$

Although these values are somewhat ad hoc, experience indicates that they are in fact applicable to a wide range of problems. Similar behavior has been observed by other authors [2].

2.3. Algebraic domain decomposition

The basis of the algebraic domain decomposition is a colouring of the elements. Consider a mesh $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, with points $\mathscr{V}$ and elements $\mathscr{E}$, and denote by $V_{ij} = E_i \cap E_j$ the subset of points shared between elements E_i and E_j. A colouring group is defined as a subset G of $\mathscr{G}$ for which

$$V_{ij} = \emptyset \quad \forall E_i, E_j \in G \tag{9}$$

Since no two members of G share a point, they also cannot share a face or an edge.

When the mesh is modified, a new set of colouring groups are generated for the new mesh. As a result of the implementation, a set of colouring groups for a modified mesh will

contain only unrefined elements, i.e. elements in the original mesh which have not refined along with the children of elements which have refined. The refinement cases used in this work (discussed subsequently) have the property that when an element is refined, each child element will share at least one point with all of its siblings. An important consequence of this property is that no two elements within a colouring group can be siblings.

Simple colouring algorithms which yield $\approx$ 30–50 colours for a typical 3D unstructured grid can be devised fairly easily. The colouring itself can be performed in parallel if the elements are divided evenly among the processors and a separate set of colouring groups is generated independently on each processor. Whether or not the subset of elements assigned to a processor actually forms a topologically continuous submesh is irrelevant, and generally they will not. Numerical experience indicates that with this parallelization approach the number of colours typically increases by 10–15% with each doubling of the number of processors.

For certain substeps of the adaption procedure, all that is required for parallelization is a division of elements (or, alternatively, edges, faces, points, etc.) among processors. In these instances, the non-overlapping properties of the colouring groups are unnecessary and the colouring groups simply form a convenient means of partitioning the mesh. Alternatively, one can partition the mesh into equally sized portions (one per processor). In either case the domain decomposition is still performed at an algebraic level, i.e. via array indices, with no explicit 'handing off' of elements to processors. Again, the subset of elements assigned to a given processor will not in general form a topologically continuous submesh, nor does it need to.

2.4. Solution interpolation

When a new grid point is introduced along an edge, the solution at the new grid point must be interpolated from neighbouring points. The simplest and fastest way to interpolate the solution is via the Finite Element basis functions. This amounts to a simple node-based average on an edge, assuming that the new point is located at the centre of an edge. Other approaches also have been examined, such as a strictly conservative interpolation and a higher-order monotone approach similar to a MUSCL scheme. However, no differences in the results were observed. Therefore the nodal average was implemented in the interest of computational efficiency.

2.5. Refinement cases

The allowable refinement cases are the standard subdivisions of a tetrahedral element into two, four, or eight child elements. These three basic cases are referred to as the 1:2, 1:4, and 1:8 cases, respectively. Subdivision of 1:2 and 1:4 children is not allowed; instead, the parent element is fully refined into eight children. These cases are referred to as the 2:8 and 4:8 cases. The intermediate 2:4 case has been omitted for simplicity. The refinement cases, and the number of possible variations for each case, are illustrated in Figure 1. Note that the three basic cases can be classified according to the number of edges which are refined: one (1:2), three (1:4), or six (1:8). Refinement is limited to a user-specified maximum number of levels. Recall that for the 1:8 refinement case, three configurations are possible due to the choice of the inner diagonal. Only one of these configurations (chosen arbitrarily) has been implemented.

The inverses of the refinement cases form the allowable derefinement cases: 2:1, 4:1, 8:1, 8:2, and 8:4. For derefinement the 4:2 case also is included. The parent element and its children are collectively referred to as a derefinement family, and the derefinement cases can

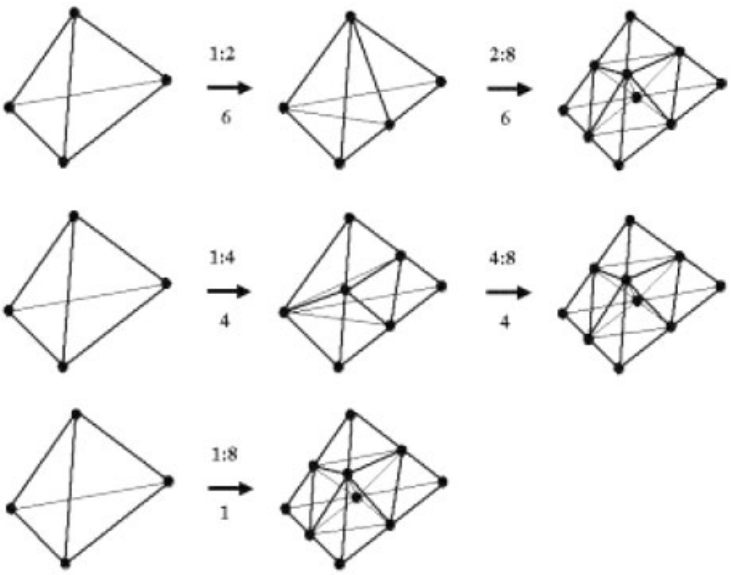

Figure 1. Allowed refinement cases.

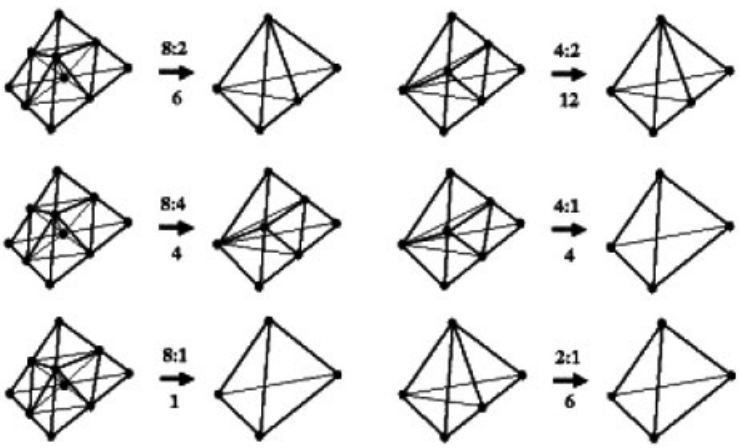

Figure 2. Allowed derefinement cases.

be classified according to the number of points which are removed from the derefinement family. No restriction is made on the order in which derefinements can occur, and elements in the original mesh are not allowed to coarsen. The derefinement cases are shown in Figure 2.

2.6. Adaption data structures

The fundamental adaption data structure is a 'master' list of elements. This list contains the nodes of all elements, including refined elements. A separate list contains the 'active' elements. The active list represents the current mesh at any stage in the simulation.

For each master element, the following information is stored:

- active element number;
- refinement case;
- number of children;
- child elements;
- refinement level;
- child number;
- parent element.

For the active elements, an additional pointer-like array stores the corresponding master element numbers.

The flow solver (which includes the routines for the aforementioned secondary operations) operates only on the active list of elements; the master list of elements and all other adaption arrays are contained in a separate module which is accessible only by the adaption subroutines. To reduce memory usage, no information for a refined element is retained other than that contained in the adaption data structures.

3. ADAPTION ALGORITHM

3.1. Overview

An adaptive refinement algorithm can roughly be broken down into the following substeps:

1. Compute error estimate at the edges.
2. Identify edges for refinement.
3. Identify edges for derefinement.
4. Add points and elements to the mesh.
5. Remove points and elements from the mesh.
6. Reinitialize solver.

Variations on the above are of course possible. The point of this breakdown, however, is to illustrate that for maximum parallel efficiency, each of the above tasks must be parallelized such that the two desired requirements outlined in Section 1 are satisfied.

Note that the first of the above substeps consists of a single loop over the edges and is trivial to parallelize once the gradients of the indicator variable are known. Therefore, no further discussion of this step is necessary. Additionally, the last substep consists of the aforementioned secondary operations and is not treated here. The remaining substeps will be addressed in the following subsections. A complete description of the algorithmic procedure for each substep is not practical. Instead, the goal is to provide a general sense of the parallel implementation with a moderate amount of low-level detail; the remainder can hopefully be deduced from the text.

3.2. Refinement compatibility

The identification of edges which satisfy the refinement criterion is on the surface a relatively straightforward task: any edge which satisfies the refinement criterion is simply flagged for refinement. However, due to the fact that the adaption criteria are applied at the edge-level,

while adaption decisions are ultimately applied at the element level, this initial marking of edges will not in general yield a compatible refinement pattern for all elements. For example, an element may have four edges marked for refinement, a situation which does not correspond to any allowed refinement case. Therefore, after the initial marking, additional edges must be marked to ensure that any element with marked edges complies with one of the allowed refinement cases. Additionally, if an edge is marked for refinement, all elements which contain that edge will be affected. Therefore, the refinement pattern for a given element also must be compatible with the refinement patterns of all neighbouring elements. The general procedure used to ensure consistent refinement patterns is referred to as *compatibility*, and a parallel compatibility algorithm for refinement is described in what follows. Compatibility for derefinement is discussed in a subsequent section.

The following terminology will be useful in what follows:

- Intermediate points: points which belong to the children of 1:2 and 1:4 refinements, but not to the parent element.
- Locked edges: edges which are not allowed to refine.
- Intermediate elements: elements which contain at least one intermediate point.

Initially, any edge which contains at least one intermediate point is flagged as a locked edge. The requirement that the 1:2 and 1:4 cases refine only into 2:8 and 4:8 cases is therefore equivalent to prohibiting refinement of locked edges. This stipulation can be explained as follows. As shown in Figure 1, when a 1:2 or 1:4 case is created, a subset of the edges of the original parent element is refined. Subsequently, a 2:8 or 4:8 case requires that all edges of the parent element which were not originally refined be refined, while all other edges in the refinement family be held fixed (some of these edges will be removed during the internal reconnection process needed to create the new child elements). Inspection of the refinement cases indicates that this latter category of edges consists of edges in the refinement family which contain one or more intermediate points. Therefore, these edges are flagged as locked edges and are not allowed to refine.

As the compatibility procedure progresses, additional edges which do not contain intermediate points will be flagged as locked edges. Elements which contain such edges will generally be adjacent to elements which undergo a 2:8 or 4:8 refinement.

Each element can be therefore be placed into one of three classes:

1. Normal elements without locked edges.
2. Normal elements with locked edges.
3. Intermediate elements with at least one edge marked for refinement.

A separate compatibility algorithm must be applied to each of the above classes. The algorithm for the first class is presented first. For a given element, denote by `nrefine` the number of edges marked for refinement. The phrase ‘activate an edge’ is taken to mean ‘mark the edge for refinement’ (and the opposite for the phrase ‘deactivate an edge’).

Algorithm I: Refinement Compatibility, Class 1

```
(1) If nrefine = 1
(2)   Accept as a 1:2 refinement
(3) Else If nrefine = 2 OR nrefine = 3
(4)   If active edges are on the same face
```

```
(5)      Activate any inactive edges of the face
(6)      Accept as a 1:4 refinement
(7)    Else If active edges are not on the same face
(8)      Activate all edges
(9)      Accept as a 1:8 refinement
(10)   End If
(11)  Else If nrefine > 3
(12)    Activate any inactive edges
(13)    Accept as a 1:8 refinement
(14) End If
```

The above algorithm takes a conservative approach which errs on the side of over-resolution rather than under-resolution.

For the second class, normal elements with locked edges, the number of possible refinement cases is reduced. Any element with locked edges will have at least one edge for which refinement is prohibited. Therefore a 1:8 refinement cannot occur for this class of elements and the final value of `nrefine` cannot exceed three.

Algorithm II: Refinement Compatibility, Class 2

```
(1)  Deactivate all locked edges
(2)  Count number of active edges
(3)  If nrefine = 1
(4)    Accept as a 1:2 refinement
(5)  Else If any face has nrefine >= 2 AND no locked edges
(6)      Activate any inactive edges of the face
(7)      Accept as a 1:4 refinement
(8)  Else
(9)    Deactivate all edges
(10)   Mark all edges as locked
(11) End If
```

Last is compatibility for intermediate elements. In this case, compatibility must be enforced at the parent level: for a 2:8 or 4:8 refinement to occur on a particular parent element, all non-locked edges of all associated child elements must be refined. If, after activation of edges, any child is found to contain an invalid pattern of marked edges, no refinement is allowed for any of the child elements. This situation can occur, for example, if an initially non-locked edge of a particular child element is subsequently locked by an adjacent non-sibling element.

Algorithm III: Refinement Compatibility, Class 3

```
(1)  Identify parent element iparent
(2)  Do for each child element ielement
(3)    Activate all non-locked edges
(4)    Deactivate all locked edges
(5)  End Do
(6)  Set compatible = TRUE
(7)  Do for each child element ielement
```

```
 (8)    If ielement is not a valid refinement case
 (9)      compatible = FALSE
(10)    End If
(11)  End Do
(12)  If compatible = FALSE
(13)    Do for each child element ielement
(14)      Deactivate all edges of ielement
(15)      Mark all edges of ielement as locked
(16)      Mark ielement as normal
(17)    End Do
(18)  End If
```

These three basic algorithms define the possible compatibility steps required for a single element. To perform a compatibility pass for the entire mesh, the three algorithms must be placed within a loop over all elements. Furthermore, the entire procedure is iterative: compatibility passes must be repeated until no additional changes occur. The number of compatibility passes is 5–15 in most cases, but occasionally increases beyond this amount. This can occur, for example, when, in a certain region of the mesh, the distribution of $\varepsilon^{\alpha\beta}$ has a mean value close to ε_{H}. Once the compatibility process is complete, all elements with marked edges can be assigned a refinement case based on the pattern of marked edges.

Since each edge belongs to more than one element, a strong possibility for memory contention exists if the compatibility process were parallelized as-is. Recall, however, that no two elements within a colouring group share a point, and if two elements do not share a point, they cannot share an edge. Therefore, elements within each colouring group can update edge-based data in parallel. The parallel compatibility procedure therefore takes the form of an outer serial loop over the colouring groups, and an inner parallel loop over the elements within each colour. The inner loop contains the three basic compatibility algorithms.

3.3. Derefinement compatibility

The derefinement compatibility procedure is qualitatively similar to the refinement compatibility procedure. The primary difference is that it operates on points rather than edges. Initially, any edge which satisfies the derefinement criterion is marked for derefinement. An element is then considered a candidate for derefinement if and only if all of its edges are marked for derefinement and all edges of its sibling elements are marked for derefinement. Points which belong to elements marked for derefinement are themselves marked for derefinement if and only if they do not belong to the corresponding parent element. As with the refinement compatibility, the derefinement compatibility also is conservative: as long as at least one edge of a given derefinement family fails to satisfy the derefinement criterion, no derefinement will occur.

When the parent element is originally refined, points are introduced on refined edges in a specific manner so as to produce a valid refinement case. This list of introduced points must be reconstructed for each derefinement family. Construction of the point list is dependent on implementation, but in general terms involves examination of the nodes of the child elements and therefore can be parallelized via the colouring approach.

Once the list of introduced points is reconstructed, derefinement involves the removal of some or all of these points in a manner consistent with the adaption cases. The compatibility

for a single parent element is as follows, where `nderefine` denotes the number of points marked for removal (i.e. the number of active points) and `icase` denotes the refinement case of the original parent. Note that for 1:2 cases, `nderefine` cannot exceed 1, and for 1:4 cases, `nderefine` cannot exceed 3.

Algorithm IV: Derefinement compatibility

```
(1)  If nderefine = 1
(2)    If icase =  1:2
(3)      Accept as a 2:1 derefinement
(4)    Else
(5)      Deactivate all points
(6)    End If
(7)  Else If nderefine = 2
(8)    If icase =  1:4
(9)      Accept as a 4:2 derefinement
(10)   Else
(11)     Deactivate all points
(12)   End If
(13) Else If nderefine = 3
(14)   If icase = 1:4
(15)     Accept as a 4:1 derefinement
(16)   Else If icase = 1:8
(17)     If inactive points lie on same face
(18)       Accept as an 8:4 derefinement
(19)     Else
(20)       Deactivate all points
(21)     End If
(22)   End If
(23) Else If nderefine = 4
(24)   If inactive points lie on same face
(25)     Deactivate third point of face
(26)     Accept as an 8:4 derefinement
(27)   Else
(28)     Deactivate all points
(29)   End If
(30) Else If nderefine = 5
(31)   Accept as an 8:2 derefinement
(32) Else If nderefine = 6
(33)   Accept as an 8:1 derefinement
(34) End If
```

For the derefinement compatibility, consistency with the allowed derefinement cases must be enforced both at the child level as well as between neighbouring parent elements. The derefinement compatibility is therefore parallelized the same way as the refinement compatibility (recall that sibling elements cannot be in the same colouring group). To accelerate the process, a given derefinement family is evaluated for derefinement compatibility only when

the first child of that family is accessed by the compatibility algorithm. As with the refinement compatibility, the derefinement compatibility also must be performed as an iterative process.

3.4. Point addition

An integer help array `edge_flag(nedge)` is used to mark edges for refinement. A value of 1 indicates a refined edge, and a value of 0 indicates an unrefined edge. The number of refined edges is therefore obtained by summing the entries of the help array.

When an edge is refined, a new point is placed at the centre of the edge. Each new point must be incorporated into the global point data structures, and therefore a global point number must be assigned to each refined edge. A simple serial algorithm to count new points and assign global numbers is the following, where `npoint_new` denotes the number of new points. Note that for each new point, the help array `point_flag(npoint)` stores the edge.

Algorithm V: Point Numbers

```
(1) npoint_new = 0
(2) Do for each edge iedge
(3)   If edge_flag(iedge) = 1
(4)     npoint_new = npoint_new + 1
(5)     edge_flag(iedge) = npoint_new
(6)     point_flag(npoint_new) = iedge
(7)   End If
(8) End Do
```

Given an array A of length N, and a second array B, with entries of 0 or 1 and also of length N, the above algorithm essentially applies the semi-recursive mapping

$$A(i) = \sum_{j=2,i} A(j)B(j) \tag{10}$$

In the context of the Point Numbers algorithm, A would be the point number stored on each refined edge and B would be the flag which indicates whether or not an edge is refined. Clearly the above relation is inherently non-parallel, and therefore an alternative approach is needed.

The above procedure can be parallelized via a two-pass approach over processors. The edges are divided up among the processors, and an auxiliary help array `npoint_proc(nproc+1)` (initialized to zero) is used to count the number of new points in each processor's group of edges. In the parallel version, `npoint_new` is a local variable private to each parallel thread.

Algorithm VI: Parallel Point Numbers

```
(1) nedge_base = nedge / nproc
(2) Parallel Do for each processor iproc
(3)   npoint_new = 0
(4)   iedge1 = (iproc - 1) * nedge_base + 1
(5)   If iproc = nproc
(6)     iedge2 = nedge
(7)   Else
```

```
 (8)     iedge2 = iproc * nedge_base
 (9)    End If
(10)    Do for each edge iedge in iedge1:iedge2
(11)     npoint_new = npoint_new + edge_flag(iedge)
(12)    End Do
(13)    npoint_proc(iproc+1) = npoint_new
(14)  End Parallel Do

(15)  Do for each processor iproc
(16)    npoint_proc(iproc+1) = npoint_proc(iproc+1) + npoint_proc(iproc)
(17)  End Do

(18)  Parallel Do for each processor iproc
(19)    npoint_new = npoint_proc(iproc)
(20)    Compute iedge1, iedge2 as before
(21)    Do for each edge iedge in iedge1:iedge2
(22)      If edge_flag(iedge) = 1
(23)        npoint_new = npoint_new + 1
(24)        edge_flag(iedge) = npoint_new
(25)        point_flag(npoint_new) = iedge
(26)      End If
(27)    End Do
(28)  End Parallel Do
```

Since this basic procedure is used extensively in what follows, a brief description is warranted. The first parallel loop can be interpreted as counting the number of non-zero entries in the array B for each subset of the array, i.e. the number of refined edges within each processor's list of edges is counted and stored with the processor in the array `npoint_proc`. In the serial loop, an offset is created for each processor: the offset consists of the total number of refined edges in all previous processor's lists of edges, i.e. mapping (10) is applied to the help array `npoint_proc` with $B = 1 \forall j$. In the final parallel loop, mapping (10) is applied locally within each processor's list of edges, and the offset for each processor is added in to give the mapping a global context.

Note that although the second loop is serial, the loss in parallel efficiency is minimal due to the short length of the loop. Also note that the parallel version requires two passes over the mesh, compared to just one for the serial version. Therefore, a certain amount of overhead is incurred. Variations on this basic parallel counting procedure will be used in what follows.

At the end of the counting procedure, the new global point number for a refined edge `iedge` is `edge_flag(iedge) + npoint`, and the edge number for new point `ipoint` is `point_flag (ipoint)` (the global point number for `ipoint` is simply `ipoint + npoint`).

Once the global point numbers are known, the global point data structures can be reallocated to the proper sizes. In Fortran 90, this task is accomplished by first copying, in parallel, the desired data to temporary storage. The arrays in question are destroyed, reallocated, and then refilled by coping, also in parallel, back from temporary storage. This approach is used in what follows whenever an array needs to be resized.

Recall that the array point_flag stores the parent edge for each new point; the co-ordinates and the interpolated unknowns at the new point are calculated from the values associated with the nodes of the parent edge. Since the storage locations for the new points are already known, the co-ordinates and unknowns can be computed and stored via a single parallel loop over new points.

New points on the boundary of the mesh are incorporated in the exact same fashion. Boundary conditions for new boundary points are determined by querying the boundary conditions of the nodes of the parent edge.

The new point number associated with each refined edge will be needed in what follows. For convenience, the new point numbers are converted to global point numbers, i.e. npoint is added to each non-zero entry of the edge_flag array.

3.5. Element addition

All elements marked for refinement are transcribed into a separate list via a counting procedure similar to that used to assign new point numbers. For each refinement case, a look-up table specifies which local edges of the parent element are refined and defines the connectivity of the child elements in terms of a local point list. The local point list for each refined element can be constructed as follows, where my_points(10) denotes the list of local points for an element, element_nodes denotes the nodes of an element, element_edges denotes the edges of an element, iedge denotes a local edge, and jedge denotes a global edge.

Algorithm VII: Point List

```
(1) Parallel Do for each refined element ielement
(2)   my_points(1:4) = element_nodes(1:4,ielement)
(3)   Look up icase and nrefine
(4)   Do for each refined edge iedge
(5)     jedge = element_edges(iedge, ielement)
(6)     ipoint = iedge + 4
(7)     my_points(ipoint) = edge_flag(jedge)
(8)   End Do
(9) End Parallel Do
```

For the 2:8 and 4:8 cases, the procedure must be slightly modified as follows. First, the local point list from the initial refinement is reconstructed using look-up tables. Second, the remaining entries in the local point list are obtained via the above procedure.

In the second step, global element numbers must be assigned to all newly created child elements. This step also is performed via the parallel counting algorithm used previously. For a given parent element, the element number for each new child is stored in the list of children for the parent element. Once the total number of new elements is known, the master element data structures are reallocated to the appropriate sizes. New elements are added to the master list of elements via a single parallel loop over the list of refined elements which makes use of the local point lists and look-up tables to define the connectivity.

3.6. Point removal

Once points have been identified for removal, elimination of these points from the data structures is fairly straightforward. The first step is to count the number of points which remain

in the mesh and to assign a new global point number to each remaining point. This step can be accomplished with the same parallel counting algorithm (the points are divided up among the processors). Next, the data structures are reallocated to their new smaller sizes. A more efficient approach would be to reallocate the data structures once, accounting for both refinement and derefinement, but this has not been incorporated. Removal of boundary points is accomplished in the same fashion.

3.7. Element removal

An element is marked for derefinement if any of its points are marked for removal. All parent elements which have children marked for removal are transcribed into a separate list with the usual counting procedure. These parents are then processed, and the child elements removed as needed. For the 2:1, 4:1, and 8:1 derefinement cases, the child elements are simply marked for removal from the master list of elements. In the other derefinement cases, the connectivity of child elements which remain must be redefined, and any excess children must be removed.

To facilitate the redefinition of remaining child elements, the local point list for refinement of the parent can be reconstructed. The remaining child elements are subsequently redefined with simple table look-ups. The entire redefinition and flagging procedure contains no possibility for memory contention and therefore can be performed with a single parallel pass over the list of parent elements.

Once child elements have been flagged for removal and/or redefined, the master element list and other adaption data structures are updated in a similar fashion to the point data structures. The only added difficulty is that new child/parent numbers must be tracked and updated.

3.8. Active mesh update

The final step in an adaption pass is to generate the new list of active elements from the master element list. The algorithm for this step is as follows, where `nelement_href` is the number of elements in the master list and `nelement_proc(nproc)` is a help array. The array `master_flag(nelement_href)` is initially 1 for active elements and 0 for inactive elements. The master list of nodes is contained in the array `master_nodes`.

Algorithm VIII: Active Mesh Update

```
(1)  nelement_base = nelement_href / nproc
(2)  Parallel Do for each processor iproc
(3)    nelement_new = 0
(4)    ielement1 = (iproc - 1) * nelement_base + 1
(5)    If iproc = nproc
(6)      ielement2 = nelement_href
(7)    Else
(8)      ielement2 = iproc * nelement_base
(9)    End If
(10)   Do for each element ielement in ielement1:ielement2
(11)     nelement_new = nelement_new + master_flag(ielement)
(12)   End Do
(13)   nelement_proc(iproc+1) = nelement_new
```

```
(14)  End Parallel Do

(15)  Do for each processor iproc
(16)    nelement_proc(iproc+1) = nelement_proc(iproc+1) + nelement_proc(iproc)
(17)  End Do

(18)  Parallel Do for each processor iproc
(19)    nelement_new = nelement_proc(iproc)
(20)    Compute ielement1, ielement2 as before
(21)    Do for each element ielement in ielement1:ielement2
(22)      If master_flag(ielement) = 1
(23)        nelement_new = nelement_new + 1
(24)        element_flag(nelement_new) = ielement
(25)        master_flag(ielement) = nelement_new
(26)        element_nodes(1:4,nelement_new) = master_nodes(1:4,ielement)
(27)      End If
(28)    End Do
(29)  End Parallel Do
```

At the end of this procedure, for each active element `ielement`, the corresponding master element number is `element_flag(ielement)`, and the final number of active elements is `nelement = nelement_proc(nproc+1)`.

After the active element list is finalized, the refinement module passes control back to the flow solver. All secondary operations are performed at this point.

4. PARALLEL PERFORMANCE

An unsteady compressible flow calculation was used to evaluate the parallel performance of the adaption algorithm. The test problem consisted of a 3D shocktube with Sod initial conditions. The initial mesh contained approximately 10^6 elements. The calculation was performed for 30 time steps, with mesh adaption on the first time step (based on the initial conditions) and every five time steps thereafter; therefore, mesh adaption occurred a total of seven times. Two levels of refinement were allowed, and density was used as the error indicator variable. The mesh size reached approximately 7×10^6 elements by the second adaption pass, and approximately 10^7 elements by the fourth adaption pass. The mesh size remained at approximately this level for the remainder of the calculation. The calculation was repeated with up to 64 processors, and for each case the total amount of CPU time spent in the mesh adaption module was measured via the system profiler.

The results of this study are shown in Figure 3: the solid line indicates ideal parallel speedup, and the dashed line indicates the measured parallel speedup. The initial super-linear speedup for two processors is due to cache effects. Application of Amdahl's Law to the data (with the speedup for two processors limited to two) yields an average parallel fraction of 0.987. Note that this value has been achieved with a fixed problem of moderate size; a well-known caveat of Amdahl's Law is that for a given code, p generally increases with problem size.

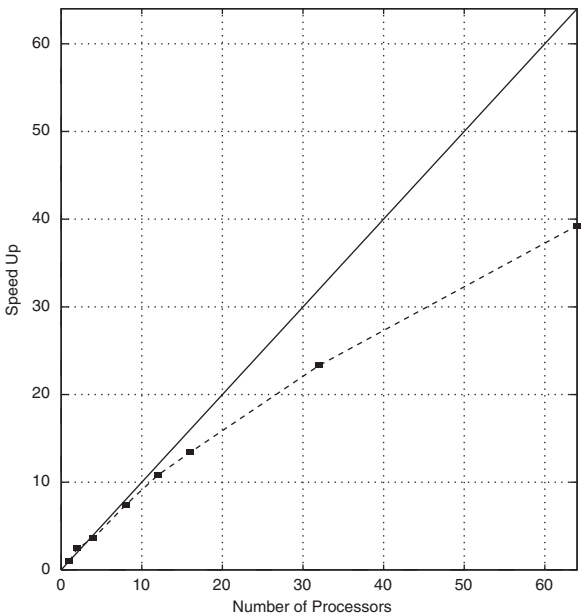

Figure 3. Parallel speedup of adaption algorithm.

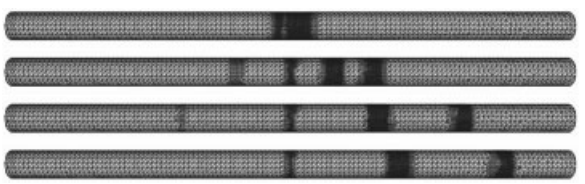

Figure 4. From top to bottom: adapted surface mesh at $t = 1.8$ s; $t = 8.0$ s; $t = 16.3$ s; and $t = 20$ s.

5. NUMERICAL EXAMPLES

5.1. Shocktube

The first numerical example is identical to that used to evaluate the parallel performance, except that the initial mesh contained only 2×10^4 elements. The mesh size increased by approximately one order of magnitude. The results are shown in Figures 4 and 5: Figure 4 shows the adapted mesh at different times during the calculation, and Figure 5 shows the density and pressure along a centreline through the shocktube after an elapsed time of 20 s. In the latter two plots, the data represent the solution at interpolation points evenly spaced along the centreline, and not the solution at mesh points. The solid line indicates the exact one-dimensional solution. The results shown were obtained using four processors. The calculation was repeated using between one and 32 processors, and in all cases the results were numerically identical.

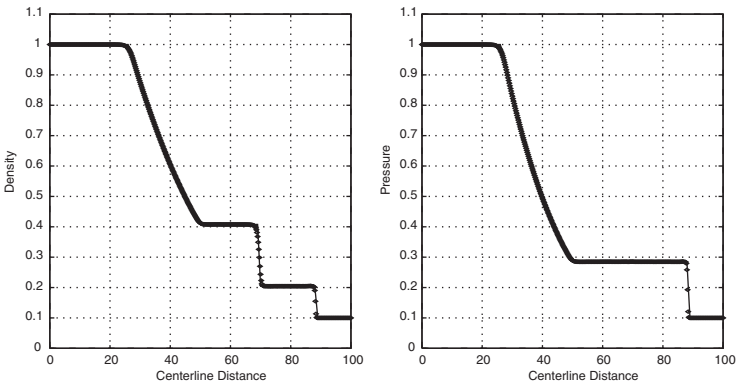

Figure 5. Computed density (left) and pressure (right) along centreline at $t = 20$ s.

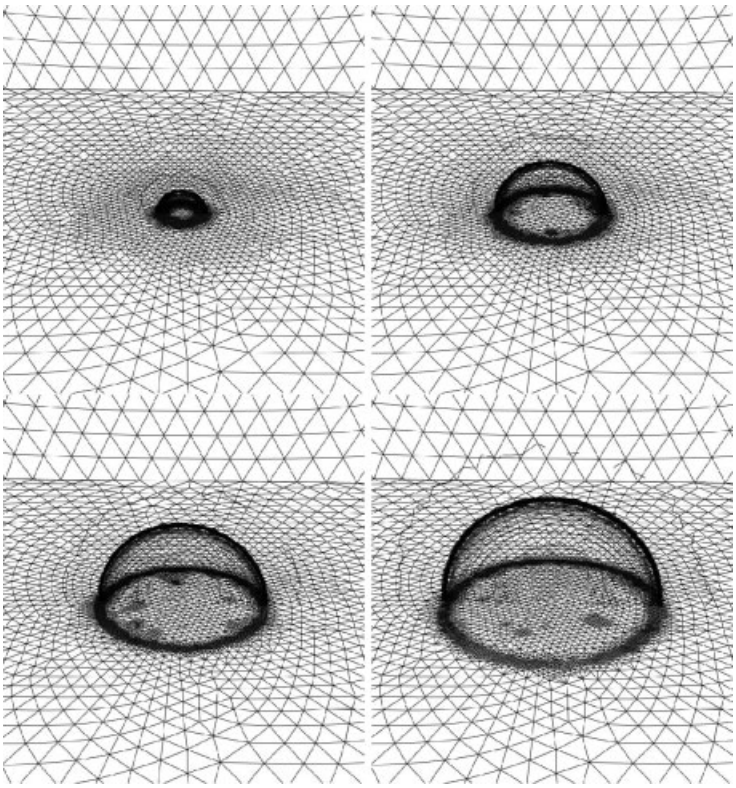

Figure 6. Adapted surface mesh and pressure contours after 100, 400, 700, and 1000 time steps.

5.2. *Taylor–Sedov problem*

The second numerical example consists of a three-dimensional Taylor–Sedov problem. The blast was initialized as a hemi-spherical region with a radius of approximately 10 cm, with

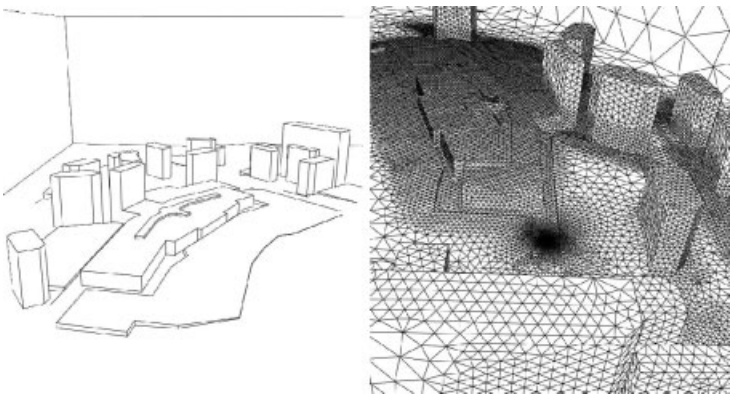

Figure 7. Geometry definition for shopping complex and blast location.

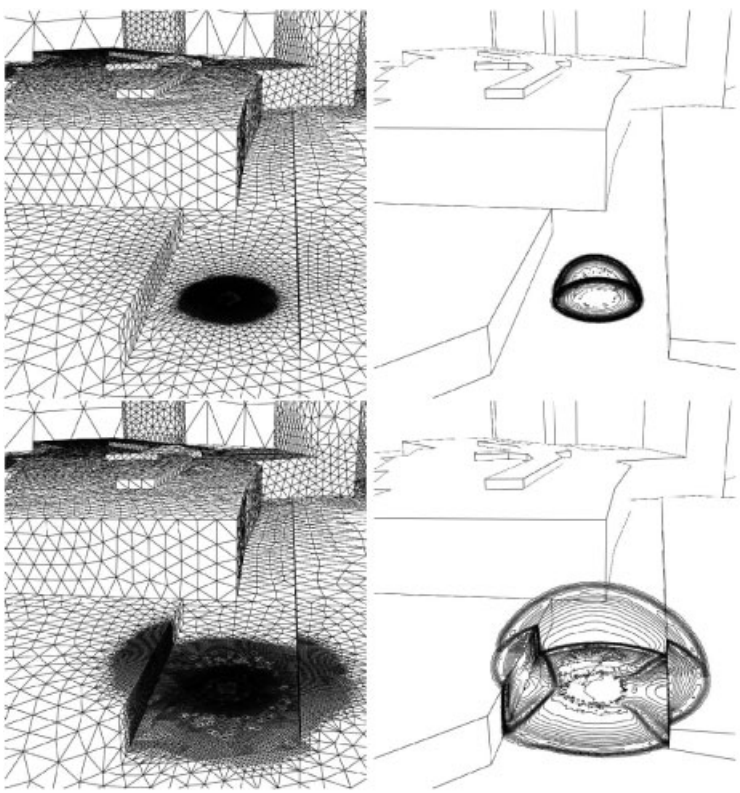

Figure 8. Adapted mesh and pressure contours at different times.

an initial pressure and energy in this region 10^{15} times higher than the ambient values. Two levels of mesh refinement were allowed, with pressure as the indicator variable. The initial mesh contained approximately 10^5 tetrahedra. Figure 6 shows the adapted mesh on the bottom

surface and pressure contours on a vertical cut-plane through the blast origin at various times during the calculation. The calculation was performed with two processors, and at the time of the final image the mesh contained approximately 10^6 tetrahedra.

5.3. *Urban blast*

The final numerical example is a blast in an urban shopping complex. Figure 7 shows the geometry and blast location (indicated by the area of high mesh-resolution in the approximate centre of the image). Three levels of mesh refinement were allowed, with pressure as the indicator variable. The initial mesh contained approximately 7×10^5 tetrahedra and 24 processors were used. Figure 8 shows the adapted surface mesh and pressure contours at different times during the calculation. The pressure contours are shown on the surface and on a vertical cut-plane through the blast origin. At the time of the final image the mesh contained approximately 2×10^7 tetrahedra.

The calculation was repeated on the same number of processors, but with the mesh adaption operating serially and the remainder of the calculation operating in parallel. Comparison of the two calculations (based on system profiler measurements) indicates a parallel speedup of approximately 20 in the mesh adaption for this problem. This speedup corresponds to a parallel fraction of 0.991, which is slightly higher than that measured in the speedup study of Section 4. The slightly higher parallel fraction is most likely due to the larger problem size.

6. CONCLUSIONS

A parallel adaptive refinement algorithm for three-dimensional unstructured grids has been described. The algorithm is based on hierarchical *h*-refinement/derefinement for tetrahedral elements, and is parallelized via a domain decomposition of the mesh at the algebraic level. The algorithm has been implemented in a general purpose Computational Fluid Dynamics code, and the effectiveness of the overall parallel adaptive flow solver has been demonstrated. Scaling studies indicate that a parallel fraction on the order of 0.99 is achieved for the mesh adaption procedure.

A reexamination of the hypothetical calculation discussed in Section 1 is appropriate as a closing point. Recall that in this calculation, a perfectly parallel flow solver was assumed to consume 90% of the computational time, and mesh adaption was assumed to consume 10% of the computational time. In the case of a purely serial mesh adaption procedure, the maximum possible speedup was limited to 10. However, given a parallel fraction of 0.99 for the mesh adaption, the maximum possible speedup increases to 1000. Although this example is somewhat contrived, it nonetheless illustrates the impact of parallel mesh adaption on the overall parallel efficiency of an adaptive unstructured flow solver. More generally, it reinforces the notion that if truly scalable performance is desired from a code, all aspects of the code must be parallelized.

ACKNOWLEDGEMENTS

This work was completed while the author held a National Research Council Postdoctoral Research Associateship at the Naval Research Laboratory, Washington, DC. The shopping complex geometry was provided by Dr. F. Camelli of George Mason University, Fairfax, VA.

REFERENCES

1. Mavriplis DJ. Accurate multigrid solution of the Euler equations on unstructured and adaptive meshes. *AIAA Journal* 1990; **28**:213–221.
2. Löhner R, Baum J. Adaptive *h*-refinement on 3-D unstructured grids for transient problems. *International Journal for Numerical Methods and Fluids* 1992; **14**:1407–1419.
3. Kallinderis Y, Vijayan P. Adaptive refinement-coarsening scheme for three-dimensional unstructured meshes. *AIAA Journal* 1993; **31**:1440.
4. Peraire J, Piero J, Morgan K. Adaptive remeshing for three-dimensional compressible flow calculations. *Journal of Computational Physics* 1992; **103**:269–285.
5. Minyard T, Kallinderis Y. A parallel Navier–Stokes method and grid adapter with hybrid prismatic/tetrahedral grids. *AIAA Technical Paper* 1995–0222, 1995.
6. Shephard MS, Flaherty JE, de Cougny HL, Ozturan C, Bottasso CL, Beall MW. Parallel automated adaptive procedures for unstructured meshes. *Parallel Computing in CFD*, AGARD-R-807, 1995.
7. Shostko A, Löhner R. Parallel 3-D H-refinement. *AIAA Technical Paper* 1995–1662, 1995.
8. Oliker L, Biswas R, Gabow HN. Parallel tetrahedral mesh adaption with dynamic load balancing. *Parallel Computing* 2000; **26**:1583–1608.
9. Leyland P, Richter R. Completely parallel compressible flow simulation using adaptive unstructured meshes. *Computer Methods in Applied Mechanics and Engineering* 2000; **184**:467–483.
10. Aftosmis M, Berger MJ, Adomavicius G. A parallel multilevel method for adaptively refined Cartesian grids with embedded boundaries. *AIAA Technical Paper* 2000–0808, 2000.
11. Zhang SJ, Wang TS, Liu J, Chen YS, Godavarty D, Mallapragada P. A parallelized, adaptive, multi-grid hybrid unstructured solver for all-speed flows. *AIAA Technical Paper* 2002–0109, 2002.
12. Waltz J. Parallel adaptive unstructured finite element schemes for 3D compressible and incompressible flows. *AIAA Technical Paper* 2002–2978, 2002.
13. Waltz J. Derived data structure algorithms for unstructured finite element meshes. *International Journal for Numerical Methods and Engineering* 2002; **54**:945–963.
14. Hirsch C. *Numerical Computation of Internal and External Flows*, vols. 1 and 2. Wiley: New York, 1988.
15. Aftosmis MJ, Berger MJ. Multilevel error estimation and adaptive *h*-refinement for Cartesian meshes with embedded boundaries. *AIAA Technical Paper* 2002–0863, 2002.