

Techniques for Parallel Adaptivity

P.K. Jimack
Computational PDE Unit
School of Computer Studies
University of Leeds
Leeds LS2 9JT, UK

Abstract: This paper considers the main algorithmic issues associated with developing parallel mesh adaptivity software for use with tetrahedral-based parallel finite element or finite volume solvers for transient computational mechanics problems in three space dimensions. Issues that are addressed include the use of different parallel data structures, the modification of these data structures when adaptivity occurs and the dynamic maintenance of load-balance in the parallel solver.

1 Introduction

Adaptive algorithms are an important feature of almost all state-of-the-art software for computational mechanics and scientific computation. These algorithms can take many forms, the most common being h-refinement (e.g. [25, 33]), p-refinement (e.g. [2, 42]) or r-refinement (e.g. [26, 27]), with various combinations of these also possible (e.g. [3, 10, 11]). The overall aim of any adaptive algorithm is to allow a balance to be obtained between accuracy and computational efficiency. In those regions of the domain where the solution is found to be most active there should be the most degrees of freedom, whereas many fewer degrees of freedom are required in those regions where the solution is smoothest. Ideally, a prescribed global error should be obtained with the smallest number of degrees of freedom possible by locating these degrees of freedom in an optimal manner.

In this paper attention is restricted to those adaptive algorithms which are based upon *mesh* adaptivity: h-refinement, r-refinement and combinations of these. It will be assumed that the solution scheme being used is based upon either finite elements or finite volumes and is of a fixed spatial order throughout. In addition to this, the paper will focus on the particular issues associated with the implementation of such adaptive algorithms on *distributed memory parallel computers* (or, at least, when programming under this paradigm). It will be shown that the combination of both parallelism *and* adaptivity introduces a number of important algorithmic problems which must be resolved; and these will be addressed here with reference to some particular implementations ([13, 29, 32]). These implementations also provide the main motivation for wishing to

understand parallel adaptivity since many transient 3-d problems, for example, are simply too complex to solve without the use of both adaptivity *and* a powerful parallel computational platform.

1.1 A Review of Some adaptive Algorithms

Any adaptive algorithm requires some form of error information to guide the positioning of the degrees of freedom. This is not an issue which will be addressed in this paper however, other than to note that such information may take the form of a formal estimate of the error in a computed solution (as in [1, 4, 34] for example), or else may simply be some form of error indicator based upon derivatives of dependent variables for example (as in [9, 28]). For steady problems we will assume that a local error estimate or indicator is available whenever a solution has been computed, and for transient problems (which are the main focus of this paper) it will be assumed that a local error estimate or indicator¹ is available at the end of each time-step.

Possibly the simplest adaptive strategy that can be used is to regenerate an entire mesh whenever a computational solution is not satisfactory (either locally or globally). This new mesh may have its local density controlled by the error estimates obtained from the previous solution so as to ensure that the degrees of freedom are located effectively. For steady-state problems this strategy is often quite effective (e.g. [35, chapter 5]), especially since it may easily be modified to only remesh certain regions within the domain rather than the entire domain when this is appropriate (as in [14] for example). Moreover, the parallel implementation of this approach relies only on the use of a robust parallel mesh generation tool such as that described in [16, 17, 31] for example. The cost of this approach can become prohibitive for transient problems however since the remeshes are often quite frequent and accurate interpolation is required from one mesh to the next.

An alternative strategy, which may be applied naturally to transient as well as steady problems, is based upon hierarchical refinement of an initial coarse mesh. In different regions of the spatial domain the depth of this refinement need not be the same, even if the coarse mesh is quite uniform. Moreover, by maintaining a hierarchical data structure, coarsening of an existing mesh may be achieved by the removal of one or more layers of refinement – either locally or globally. There are many examples of the application of this technique to transient problems in two dimensions, [6, 18, 25] being just a few. Furthermore, as in [6, 8, 21, 33] for example, local hierarchical refinement has also been applied to three-dimensional problems using tetrahedral elements. This is the approach that we concentrate on in the next section of the paper, where parallel implementations of such refinement strategies are considered (e.g. [13, 29, 32, 37]).

Recently, a number of authors have considered the effects of complementing h-refinement algorithms, such as those mentioned above, with the use of

¹From now on we will use the single term *error estimate* to represent either a formal *a posteriori* error estimate *or* an error indicator, which may be obtained with considerably less rigour.

node movement (r-refinement). On their own, r-refinement algorithms, including those described and analyzed in [19, 26, 27] for example, have never really proved to be sufficiently robust to be the basis of reliable and efficient adaptive software. Nevertheless, in combination with local h-refinement a number of practical advantages may be observed ([3, 10]). This is because the aspect ratio of the elements is permitted to alter substantially as a calculation progresses, thus allowing higher resolution in some directions than in others when this is appropriate. Moreover, r-refinement has a number of theoretical advantages when it comes to producing parallel software since distorting an existing partitioned mesh does not affect its load-balance in any way (see Subsection 1.2 below).

Unfortunately, the concepts of local node relocation, r-refinement, and of hierarchical refinement for transient problems do not fit together in a natural manner. The whole purpose of maintaining a mesh hierarchy (as well as allowing the use of efficient parallel multilevel solvers (e.g. [3, 6])) is to allow simple mesh coarsening through de-refinement. However, this feature will not generally remain once relocation of vertices is permitted. The way in which different 3-d parallel adaptive algorithms (e.g. [13, 29, 32]) cope with this conflict is discussed in some detail in Section 2.

1.2 The Load-Balancing Issue

In order for a parallel finite element or finite volume solver to perform efficiently when solving a large problem on a tetrahedral mesh it is necessary that the workload on each processor should be approximately equal (assuming that a homogeneous parallel system is being used). This is generally achieved by partitioning the mesh across the processors so that each processor *owns* about the same number of elements. All of the computational work associated with a given element is undertaken by the processor which owns it, with a certain amount of data communication being required when neighbouring elements are owned by different processors (see Section 2 for further details and some examples). This means that an additional constraint on a partition of the computational mesh is that, in order to minimize the amount of inter-processor communication, the number of elements with neighbours owned by different processors should be kept as low as possible. This load-balancing problem is well-known for the parallel solution of steady problems using finite element or finite volume schemes and there are many efficient heuristics for obtaining good partitions of meshes and their dual graphs ([5, 15, 22, 30, 39, 41] being just a few examples).

When mesh adaptivity is used an extra dimension is added to this load-balancing problem however. It is clear that the adaptivity must be carried out in parallel since serial adaptivity would require a copy of the entire mesh to be held on a single processor and would lead to a major serial bottleneck. However, once parallel adaptivity has occurred it is likely that, even if a mesh was well-partitioned before, the quality of the partition will have deteriorated significantly afterwards. For the same reasons that it is undesirable to perform the adaptivity on a single processor it is also undesirable to re-partition the mesh using just one processor: it would carry a large communications overhead, become a serial

bottleneck and would be constrained by the amount of memory available to a single processor. Hence a parallel load-balancing algorithm is required which is capable of modifying an existing partition in a distributed manner so as to improve the quality of the partition whilst keeping the amount of data relocation as small as possible. Whilst this particular *dynamic* load-balancing issue is not considered explicitly in this paper (see [20] or [36] for a detailed discussion of this) its importance is such that, as may be seen from the next section, any parallel adaptivity algorithm must pay significant attention to it.

2 Parallel Algorithms

This is the major section of the paper in which we discuss the algorithmic details behind a number of practical implementations of parallel adaptivity for transient problems in three space dimensions. In each of the cases considered in detail ([13, 29, 32]) the underlying parallel solver works on a partitioned mesh of tetrahedra and so the discussion of adaptivity will be restricted to the refinement and coarsening of such meshes. The section is broken down into four related subsections, beginning with an introduction to some common distributed data structures. This subsection also discusses some of the advantages and disadvantages associated with maintaining a complete hierarchy of meshes. The following subsection continues with this theme by describing some of the techniques actually used for refining and coarsening meshes and shows how these may differ when a hierarchy is or is not maintained. The final two subsections concentrate almost entirely on parallel issues by considering the important problems of ensuring data consistency between processors and maintaining a good load-balance respectively.

2.1 Data Structures

Clearly the exact data structures that are used for a parallel (or sequential) implementation of a complex algorithm will vary from code to code. It is not the intention of this short review article to consider such technicalities in detail here, however it is necessary to discuss some data structure issues so that mesh partitioning and adaptivity may then be considered.

In general a mesh may be considered to be made up of a number of different entities. In three dimensions these are typically elements, faces, edges and vertices. Numerous data structures can then be defined to relate these different entities to each other. For example, each element may contain a pointer to its four faces, each face to its three edges, each edge to its two vertices and each vertex to its three coordinates. Other data structures may also be defined, such as each element pointing to its four vertices or to the four neighbouring elements with which it shares a common face. It is these data structures which (provided they are consistent) define an unstructured tetrahedral mesh. Hence, when we refer to a partition of a mesh this really means a partition of these data structures.

In order to distribute the data structures as efficiently as possible it is usual to partition the elements first, then the faces and then the edges and vertices. This way it may be ensured that each vertex is owned by a processor which owns at least one of the edges connected to it, each edge is owned by a processor which owns at least one of the faces connected to it, etc. There are essentially two choices about how to deal with a data structure which points to an entity which is owned by a different processor. (For example, an element data structure may exist which maintains a pointer from each element to its four vertices. Whilst each vertex is guaranteed to be owned by a processor which owns at least one of the (many) elements around it, there is no guarantee that all (or any) of the four vertices of an element are owned by the processor which owns the element.) The first option is to use the notion of *halo* data. This involves the processor which owns the data structure having its own *copy* of those entities that are pointed to by this structure which are owned by a different processor. The second is to make use of pointers from one processor to another in order to allow the data structure to be resolved. These two approaches are considered further in Subsection 2.3. In the remainder of this subsection however we discuss the issue of the leanness of a code's data structures against their generality, and the issue of whether or not to maintain a hierarchy in these data structures.

In [29] Oliker *et al.* tie their parallel adaptivity very closely with their edge-based parallel flow solver. The edge-based solver that they use has been selected because the number of edges in a tetrahedral mesh is much smaller than the number of faces and so an edge-based scheme is likely to be more efficient than an element-based finite volume one. The mesh data structures used in this work reflect this close coupling with their particular solution scheme by being very lean at the expense of allowing generality in the parallel solver. At a given level of refinement only the following structures are present.

- A vertex list with each entry containing its coordinates and solution values and a list of those edges radiating from it.
- An edge list with each entry containing the pair of vertices at the ends of the edge, a list of elements around the edge, a colour (used to group edges with no common vertex) and a (possibly empty) list of boundary faces to which the edge belongs.
- An element list with each entry containing the six edges defining the element and a subdivision type.
- A list of boundary faces with each entry containing the three edges which make up the face, a boundary type, a colour and a pointer to the element to which the face belongs.

In addition to this data however, three parent-child hierarchies are also maintained for the edges, boundary faces and elements. This means that, in effect, a number of different meshes are being stored simultaneously at different levels in the hierarchy. As we will see in the next subsection, this has a significant effect on the way in which coarsening of the mesh may be achieved.

In contrast, the work of Flaherty *et al.* [13] makes use of more general data structures to describe a given mesh, however they do not maintain a hierarchy of meshes as with [29]. Instead of just defining boundary faces, Flaherty *et al.* define a face entity to exist throughout the mesh and then define two-way data structures to link elements, faces, edges and vertices as illustrated:

$$\text{Elements} \longleftrightarrow \text{Faces} \longleftrightarrow \text{Edges} \longleftrightarrow \text{Vertices}.$$

Moreover, each of these entities are explicitly classified relative to a geometric model so as to allow an appropriate geometric representation of the spatial domain as the mesh is refined. The use of face entities throughout the mesh means that the memory required for these structures will be greater than that required in [29], however the advantage is that a much wider variety of parallel solvers may now be used (both finite element and cell-centred finite volume for example). The lack of a mesh hierarchy means that parallel multilevel solvers are not practical however.

The third approach that we consider in detail here is that of Selwood *et al.* ([32]) which uses an even heavier data structure by effectively combining the hierarchy of meshes used in [29] with the use of a face entity as in [13]. The motivation behind this approach is to build a mesh adaptivity module which can stand alone and which therefore needs to be able to support the data structures that may be required by a wide variety of parallel solvers (both edge-based and face-based for example, as well as multilevel solvers). The consequence however is that this algorithm requires the most memory of the three that we consider here; and when data needs to be migrated in order to improve load-balance this will be a greater task than with the other two algorithms.

As well as permitting the use of parallel multilevel solvers and the ability to coarsen a mesh through de-refinement (i.e. undoing previous refinement), there is another important consequence of maintaining hierarchies of meshes at different refinement levels as in [29, 32]. In addition to the extra memory overhead, there is also the problem of how the hierarchical data structures should be partitioned. Should all children always be owned by the same processor as their parent (so-called *vertical* partitioning), or should some *horizontal* partitioning be allowed, whereby not all entities have the same owner as their parents or siblings? In both [29] and [32] the former strategy is chosen, however there are sound arguments to suggest that this could become a significant constraint on the load-balancing problem when the coarsest mesh in the hierarchy is not sufficiently fine. In this case some horizontal partitioning, as attempted in two dimensions in [6] for example, may be appropriate.

2.2 Mesh Refinement and Coarsening

All of the algorithms that we are considering here base their adaptivity on the bisection or removal of edges. In the case of [29] this is dependent upon specific edge-based error estimates whereas [13] and [32] permit more general error estimates to be used as the basis for marking edges for refinement or removal.

Again we begin with a detailed look at the algorithm employed by Olikier *et al.* [29]. They permit three basic element subdivision types: an isotropic $1 \rightarrow 8$

refinement where all of the edges are bisected, an isotropic $1 \rightarrow 4$ refinement where just three edges are bisected, and an anisotropic $1 \rightarrow 2$ refinement where just one edge is bisected. These latter two stencils are used both as buffers between different levels of isotropic ($1 \rightarrow 8$) refinement and also to achieve a degree of anisotropic refinement when this is deemed appropriate. An edge may only be removed if it is the child of another edge (i.e. if it was created by bisecting a parent edge) and if its sibling is also marked for removal. In this case the pair of edges are replaced by their parent and then local reconnection is done to either replace all sibling elements by their parent (if all children of edges of the parent element have been removed) or to replace an isotropic refinement pattern with an anisotropic pattern (if only some of the edges have been coarsened). Because parent edges and elements are retained in the data structure when refinement occurs, coarsening requires no new edges or elements to be created from scratch.

A similar strategy is used by Selwood *et al.* in [32]. The main difference is in the way that the $1 \rightarrow 4$ and $1 \rightarrow 2$ refinement stencils are implemented and used. Unlike [29], Selwood *et al.* do not permit the refinement of these *green* elements once they have been introduced: instead they should be removed and their parent refined isotropically ($1 \rightarrow 8$) before continuing. In addition to this, there is also a small difference in the stencils that are actually used for anisotropically refined elements in [32], where a temporary interior node is introduced at the centroid of each element being refined into 4 or 2 children. Otherwise the basic procedure is almost the same as for [29]. In each case the coarsening step consists of undoing previous refinements and is undertaken before the refinement step. In [32] coarsening is only usually undertaken when at least one edge in the mesh has been marked for refinement (i.e. when refinement is required as well).

The approach used for refining and coarsening elements in [13] is somewhat different however. As well as not being able to coarsen through the use of de-refinement (due to the lack of a hierarchy of meshes), the algorithm of Flaherty *et al.* also differs in that it is explicitly coupled to a geometric model of the problem domain so as to improve the geometric accuracy when refining or coarsening on the boundary. Moreover, their algorithm also incorporates a form of r-refinement through the use of a mesh smoothing procedure. As in [29, 32] mesh coarsening is completed first, however in [13] this is achieved by simply collapsing marked edges down to one of their vertices. Following this all entities which contain the deleted vertex are removed and local reconnection is performed in the polyhedral cavity around the remaining vertex. This whole process is then followed by an application of the mesh smoothing algorithm. Such an approach is completely different from the coarsening algorithms of [29, 32] in that an entirely new mesh has been created at this point: it need not be related to any previous mesh and, in some regions, it may even be coarser than the initial mesh. The refinement algorithm used in [13] is much closer to those of [29, 32] however and also makes use of predefined stencils. When all of the edges of an element are marked for refinement the same $1 \rightarrow 8$ isotropic stencil is used for example, with similar anisotropic stencils to [29, 32] being used when only some of the edges of an element are marked for refinement. The use of mesh smoothing ensures that,

unlike in [32], no distinction needs to be made between any of these stencils when it comes to further refinement.

2.3 Data Consistency

One of the main difficulties associated with computing mesh refinements and coarsenings in parallel is that of maintaining consistent data at the partition boundary. It is very simple for each processor to adapt the interior of its mesh (i.e. those entities with no neighbours owned by another processor) in parallel provided these modifications have no effect on any entities which are not in the interior. In practice this situation is very unlikely to occur however, and so one of the major overheads in parallel adaptivity turns out to be that associated with communicating alterations to a data structure on one processor to any other processors which need to know about them.

The exact nature of this communications overhead is clearly highly dependent upon the data structures that are actually in use and on how they have been partitioned. As has already been mentioned in Section 1, an important concept in the partitioning of mesh data structures is that of the halo copy. For example, in [13] any bounding faces, edges or vertices of elements along the partition boundary are duplicated on each processor which needs to use that boundary entity. Only one processor is the owner of a given entity however, and so each of the processors with a halo copy need to have this copy updated by the owner whenever it is modified (due to adaptivity for example).

A similar situation holds for the vertically partitioned hierarchical meshes used in [29, 32]. In [32] for example, there is a knock-on effect associated with refining green elements which will often be passed across the partition boundary. Recall from Subsection 2.2 that when an edge of a green element in [32] is marked for refinement, the green elements must first be removed so that their parent may then be isotropically refined. This can then lead to the bisection of an edge that would otherwise not be marked for refinement, and if this edge is shared by an element owned by a different processor, then a message will need to be passed to that processor so as to allow green refinement to take place on that element. (Note that if this is already a green element then the knock-on effect can go on for another level and so this search is best implemented recursively.) In practice, all of these local communications may be sent together at a small number of fixed points in the parallel adaptive algorithm, however their total overhead is often quite significant.

2.4 Dynamic Load-Balancing

As well as the problem of maintaining consistency of the dynamic distributed data structures when parallel adaptivity occurs the other major overhead associated with parallel refinement and coarsening of meshes is that of maintaining a good load-balance for the parallel solver. This has already been introduced briefly in Subsection 1.2 where it is observed that a parallel dynamic load-balancing algorithm is required which is capable of modifying an existing partition of a mesh

so that:

- the total load on each processor is about the same,
- the partition boundary is as short as possible,
- the amount of data which needs to be migrated is as small as possible.

Such algorithms do exist (see [23, 36, 37, 38] for example) and so the purpose of this subsection is to discuss *how* they should be used in order to maximize the overall efficiency of a parallel adaptive solver. There is clearly a cost associated not only with applying such an algorithm in order to determine what a new partition of the mesh should be but also, and more significantly, with actually migrating data between processors in order to achieve this new partition. In order for re-partitioning to be worthwhile at any stage in the solution process it is necessary that this cost should be offset by the reduction in the solver time that will result from using the improved partition.

In the case of the hierarchical refinement algorithms of [29, 32] it is important to observe that the use of vertical partitioning implies that only elements (or edges) of the coarsest mesh may be migrated between processors. When such a migration occurs the entire hierarchy of elements (edges) beneath this coarse mesh entity must be transferred with it. This clearly means that obtaining a perfect load-balance is unlikely to be possible when some coarse mesh entities have been heavily refined. Moreover, a significant amount of data communication will be associated with the migration of such entities. The algorithm of Flaherty *et al.* [13] has no such constraints on the load-balancing phase since no mesh hierarchy needs to be maintained. In this approach a diffusive strategy known as iterative tree balancing (ITB) is employed (see [12, 24, 40] for details), which allows layers of elements on interprocessor boundaries to be migrated from heavily-loaded to lightly-loaded processors.

In all three algorithms considered in detail here ([13, 29, 32]) the parallel mesh adaptivity is based upon coarsening followed by refinement. Hence, in order to minimize data migration, the natural place for the re-partitioning step to come is between these two phases of the adaptivity algorithm – when the total number of mesh entities is at a minimum. This requires an accurate predictor for the size of the mesh on each processor after refinement has occurred, based upon knowledge of which edges have been marked for bisection. Such a predictor is proposed in [13].

These ideas have been developed further still however by Biswas and Oliker in [7], and applied to the parallel adaptive algorithm of Oliker *et al.* [29]. The key to their approach is the use of two weighted dual graphs of the coarse level mesh (such a graph has a node for each coarse element and an edge connecting each pair of nodes which correspond to adjacent elements). In one of these duals each node has a weight equal to the total number of elements in the mesh hierarchy which are descendents of the corresponding coarse element, known as the *remap* weight, and in the other dual each node has a weight equal to the total number of *leaf* elements in the refinement tree beneath the corresponding coarse element – this is known as the *computational* weight. For the purposes of

re-partitioning a mesh immediately prior to the refinement stage of the adaptive algorithm, estimates of the computational weights should be used which correspond to the total number of leaf elements that are predicted to be present *after* mesh refinement². A dynamic load-balancing algorithm, such as [23, 36, 38], may then be used to determine a new partition in parallel. Associated with this partition there will be a computational gain, which is proportional to the decrease in the load imbalance over the original partition, and a re-balancing overhead, which is proportional to the amount of data that must be migrated when using the optimal remapping strategy. For the calculation of the computational gain it is necessary to make use of the computational weights in the dual graph of the coarse mesh along with an appropriate cost model for the parallel solver that is being used. When estimating the communication overhead however, the remap weights should be used in a cost model which takes into account the current latency and bandwidth characteristics of the parallel architecture. Whenever mesh adaptivity occurs a modified partition should be calculated, however the data remapping required to actually obtain this new partition should only be undertaken when the computational gain associated with it clearly exceeds the cost of performing this remapping.

As suggested by the above, the dynamic load-balancing issues associated with the use of parallel adaptivity are extremely complex. Moreover, research in this area is still developing quite rapidly and it is likely that a significant amount of further work will be required before the production of truly scalable *and* portable software which makes optimal use of adaptivity on parallel architectures is possible.

3 Discussion

In this paper we have presented a brief overview and comparison of a number of state-of-the-art algorithms for applying adaptivity in parallel when solving transient problems in three space dimensions. The major issues that these algorithms have had to address centre on the modification and partitioning of the distributed data structures which represent the computational mesh that is used by a parallel solver. In all of the cases considered here the parallel solvers are built upon the use of a spatial discretization based upon a partitioned mesh of tetrahedra, either finite element or finite volume, along with an appropriate time-stepping scheme (which may be either implicit or explicit).

The modification of distributed data structures in parallel (when adaptivity occurs) presents a number of challenges. In particular, it is vital that the data structures are kept consistent across partition boundaries. This can require a significant amount of inter-processor communication. Further communication is required to both calculate and implement the data migration that is needed in order to maintain a balanced computational load after local refinement and/or coarsening has taken place. Due to the potentially high cost of re-partitioning,

²In fact, in [7] the load-balancing algorithm is only actually applied after refinement has taken place and so the computational weights are exact rather than predicted, however this is almost certainly less efficient than re-balancing before the refinement stage.

computation and communication cost models have recently been investigated which seek to assess the net value of mapping an existing partition to a modified one.

This paper specifically avoids discussion of different parallel solvers, techniques for parallel error estimation and algorithms for parallel dynamic load-balancing – other than where this has a direct impact on the parallel adaptivity. In practice however, all of these issues are extremely closely related and no parallel adaptive software can be successfully implemented without due consideration of all of these aspects.

References

- [1] M. Ainsworth and J.T. Oden (1993), *A Unified Approach to A Posteriori Error Estimation Using Element Residual Methods*. Numer. Math., 65, pp. 23–50.
- [2] I. Babuska, B.A. Szabo and I.N. Katz (1981), *The p-Version of the Finite Element Method*. SIAM J. on Numer. Anal., 18, pp. 515–545.
- [3] R.E. Bank (1998), *PLTMG Users' Guide 8.0*. SIAM.
- [4] R.E. Bank and A. Weiser (1985), *Some A Posteriori Error Estimates for Partial Differential Equations*. Math. Comp., 44, pp. 283–301.
- [5] S.T. Barnard and H.D. Simon (1993), *A Fast Multilevel Implementation of Recursive Bisection*. In Proc. of the Sixth SIAM Conf. on Parallel Processing for Scientific Computing (ed. R.F. Sincovec *et al*), SIAM.
- [6] P. Bastian, K. Birken, K. Johannsen, S. Lang, K. Eckstein, N. Neuss, H. Rentz-Reichert and C. Wieners (1998), *UG - A Flexible Software Toolbox for Solving Partial Differential Equations*. To appear in Computing and Visualization in Science.
- [7] R. Biswas and L. Oliker (1997), *Load Balancing Unstructured Adaptive Grids for CFD Problems*. In Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing (ed. M. Heath *et al.*), SIAM.
- [8] R. Biswas and R.C. Strawn (1994), *A New Procedure for Dynamic Adaption of Three-Dimensional Unstructured Grids*. Appl. Numer. Math., 13, pp. 437–452.
- [9] M.O. Bristeau, R. Glowinski, L. Dutto, J. Periaux and G. Roge (1990), *Compressible Viscous Flow Calculations Using Compatible Finite Element Approximations*. Int. J. for Numer. Meth. in Fluids, 11, pp. 719–749.
- [10] P.J. Capon and P.K. Jimack (1995), *An Adaptive Finite Element Method for the Compressible Navier-Stokes Equations*. In Numerical Methods for Fluid Dynamics 5 (ed. M.J. Baines and K.W. Morton), OUP, pp. 327–334.

- [11] L. Demkowicz, J.T. Oden, W. Rachwicz and O. Hardy (1991), *An h-p Taylor-Galerkin Finite Element Method for the Compressible Euler Equations*. Comp. Meth. in Appl. Mech. and Eng., 88, pp. 363–396.
- [12] K.D. Devine, J.E. Flaherty, R. Loy and S. Wheat (1996), *Parallel Partitioning Strategies for the Adaptive Solution of Conservation Laws*. In Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations (ed. I. Babuska *et al.*), Springer.
- [13] J.E. Flaherty, R.M. Loy, C. Ozturan, M.S. Shephard, B.K. Szymanski, J.D. Teresco and L.H. Ziantz (1998), *Parallel Structures and Dynamic Load Balancing for Adaptive Finite Element Computation*. Appl. Numer. Math., 26, pp. 241–263.
- [14] O. Hassan, L.B. Bayne, K. Morgan, E.J. Probert and N.P. Weatherill (1998), *Unstructured Mesh Methods for 3-D Transient Flows Involving Moving Boundaries* To appear in Proceedings of the 6th ICFD Conference on Numerical Methods for Fluids, Oxford, UK.
- [15] B. Hendrickson and R. Leland (1993), *A Multilevel Algorithm for Partitioning Graphs*. Technical Report SAND 93-1301, Sandia National Laboratories.
- [16] D.C. Hodgson and P.K. Jimack (1996), *Efficient Parallel Generation of Partitioned, Unstructured Meshes*. Advances in Eng. Software, 27, pp. 59–70.
- [17] D.C. Hodgson and P.K. Jimack (1998), *A Domain Decomposition Preconditioner for a Parallel Finite Element Solver on Distributed Unstructured Grids*. Parallel Computing, 23, pp. 1157–1181.
- [18] P.K. Jimack (1993), *A New Approach to Finite Element Error Control for Time-Dependent Problems*. In Numerical Methods for Fluid Dynamics 4 (ed. M.J. Baines and K.W. Morton), OUP, pp. 567–573.
- [19] P.K. Jimack (1996), *A Best Approximation Property of the Moving Finite Element Method*. SIAM J. on Numer. Anal., 33, pp. 2206–2232.
- [20] P.K. Jimack (1997), *An Overview of Dynamic Load-Balancing for Parallel Adaptive Computational Mechanics Codes*. In Parallel and Distributed Processing for Computational Mechanics I (ed. B.H.V. Topping), Saxe-Coburg Publications).
- [21] Y. Kallinderis and P. Vijayan (1993), *An Adaptive Refinement/Coarsening Scheme for 3-D Unstructured Meshes*. AIAA Journal, 31, pp. 1440–1447.
- [22] G. Karypis and V. Kumar (1995), *A Fast High Quality Multilevel Scheme for Partitioning Irregular Graphs*. Technical Report TR 95-035, Department of Computer Science, University of Minnesota.
- [23] G. Karypis and V. Kumar (1997), *A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm*. In Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing (ed. M. Heath *et al.*), SIAM.

- [24] E. Leiss and H. Reddy (1989), *Distributed Load-Balancing: Design and Performance Analysis*. W.M. Kuck Research Computation Laboratory, 5, pp. 205–270.
- [25] R. Lohner (1987), *An Adaptive Finite Element Scheme for Transient Problems in CFD*. Comp. Meth. in Appl. Mech. and Eng., 61, pp. 323–338.
- [26] K. Miller and R. Miller (1981), *Moving Finite Elements, Part I*. SIAM J. on Numer. Anal., 18, pp. 1019–1032.
- [27] M.C. Mosher (1985), *A Variable Node Finite Element Method*. J. of Comp. Phys., 57, pp. 157–187.
- [28] J.T. Oden, T. Strouboulis and P.H. Devloo (1987), *Adaptive Finite Elements for High-Speed Compressible Flow*. Int. J. for Numer. Meth. in Fluids, 7, pp. 1211–1228.
- [29] L. Oliker, R. Biswas and R.C. Strawn (1996), *Parallel Implementation of an Adaptive Scheme for 3D Unstructured Grids on the SP2*. In Parallel Algorithms for Irregularly Structured Problems, LNCS 1117 (Springer-Verlag).
- [30] R. Preis and R. Diekmann (1996), *The PARTY Partitioning Library User Guide*. Technical Report rsfb-96-024, Department of Computer Science, University of Paderborn.
- [31] R. Said, N.P. Weatherill and K. Morgan (1998), *Unstructured Mesh Generation on Parallel and Distributed Computers*. To appear in these proceedings.
- [32] P.M. Selwood, M. Berzins and P.M. Dew (1997), *Parallel Mesh Adaptivity: Data-Structures and Algorithms*. In Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing (ed. M. Heath *et al.*), SIAM.
- [33] W. Speares and M. Berzins (1997), *A 3-D Unstructured Mesh Adaptation Algorithm for Time-Dependent Shock Dominated Problems*. Int. J. for Numer. Meth. in Fluids, 25, pp. 81–104.
- [34] E. Suli and P. Houston (1997), *Finite Element Methods for Hyperbolic Problems: A Posteriori Error Analysis and Adaptivity*. In State of the Art in Numerical Analysis (ed. I. Duff and G.A. Watson), OUP, pp. 441–471.
- [35] B.H.V. Topping and A.I. Khan (1996), *Parallel Finite Element Computations*. Saxe-Coburg Publications.
- [36] N. Touheed, P. Selwood, P.K. Jimack and M. Berzins (1998), *A Comparison of Some Dynamic Load-Balancing Algorithms for a Parallel Adaptive Flow Solver*. To appear in these proceedings.
- [37] V. Vidwans, Y. Kallinderis and V. Venkatakrisnan (1994), *Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids*. AIAA Journal, 32, pp. 497–505.

- [38] C. Walshaw, M. Cross and M.G. Everett (1997), *Dynamic Load-Balancing for Parallel Adaptive Unstructured Meshes*. In Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing (ed. M. Heath *et al.*), SIAM.
- [39] C. Walshaw, M. Cross, M.G. Everett, S. Johnson and K. McManus (1995), *Partitioning and Mapping of Unstructured Meshes to Parallel Machine Topologies*. In Irregular '95: Parallel Algorithms for Irregularly Structured Problems (ed. A. Ferreira and J. Rolim), Springer.
- [40] S.R. Wheat, K.D. Devine and A.B. MacCabe (1994), *Experience with Automatic, Dynamic Load Balancing and Adaptive Finite Element Computation*. In Proc. of 27th Hawaii Int. Conf. on System Sciences, Kihei.
- [41] R.D. Williams (1991), *Performance of Dynamic Load Balancing for Unstructured Mesh Calculations*. Concurrency: Practice and Experience, 3, pp. 457–481.
- [42] O.C. Zienkiewicz, D.W. Kelly and J.P. Gago (1983), *The Hierarchical Concept in Finite Element Analysis*. Int. J. of Comp. and Structures, 16, pp. 53–65.