3D Parallel Mesh Adaptivity: Data-Structures and Algorithms *

P.M. Selwood † M.Berzins † P.M. Dew †

Abstract

The efficient solution of transient CFD problems on distributed memory computers requires the use of parallel adaptive meshing. We will discuss issues arising in the parallelisation of a general-purpose, unstructured, tetrahedral adaptivity code. In particular, we will consider data-structure issues such as communications links and the parallelisation of complex hierarchical data-structures. We will also discuss important algorithmic and implementation aspects of the code such as the parallelisation of a depth limited recursive search. Some results given demonstrate both the feasibility and problems of the approach discussed.

1 Introduction

The need to solve ever larger and more realistic CFD problems has made it necessary to use distributed memory parallel computers to achieve acceptable solution times. For transient problems, the present trend is to consider the use of mesh adaptation, both to retain the efficiency of the solver as the solution develops and to introduce a measure of reliability through the use of computed error estimates. The meshes used on parallel machines are often so large that serial adaptivity, as well as introducing a bottleneck, is not feasible due to single processor memory constraints. There is therefore a clear need for parallel adaptive procedures. In order to address this issue, we are parallelising TETRAD (TETRahedral ADaptivity), by Speares and Berzins [7] which is a general purpose serial code using unstructured tetrahedral meshes. The technique used is that of local refinements/de-refinements of the mesh to ensure sufficient density of the approximation space throughout the domain. For many problems, the use of such an adaptive scheme can result in a large saving in the number of elements used when compared to the uniform mesh required for similar accuracy, and thus reduced solution times.

2 Data-structures

One of the major issues involved in parallelising an adaptive code such as TETRAD is how to treat the existing data-structures. TETRAD utilises a complex tree-based hierarchical mesh structure, with a rich interconnection between mesh objects. Figure 1 indicates the mesh object structures used in TETRAD. In particular, note that the main connectivity information used is 'node to element' and that the mesh hierarchy is formed by both element and edge trees. Furthermore, as the meshes are unstructured, there is no way of knowing *a-priori* how many elements share any given edge or node.

*Work funded by UK EPSRC Grant No. GR/J84915

†Computational PDEs Unit, School of Computer Studies, University of Leeds, UK

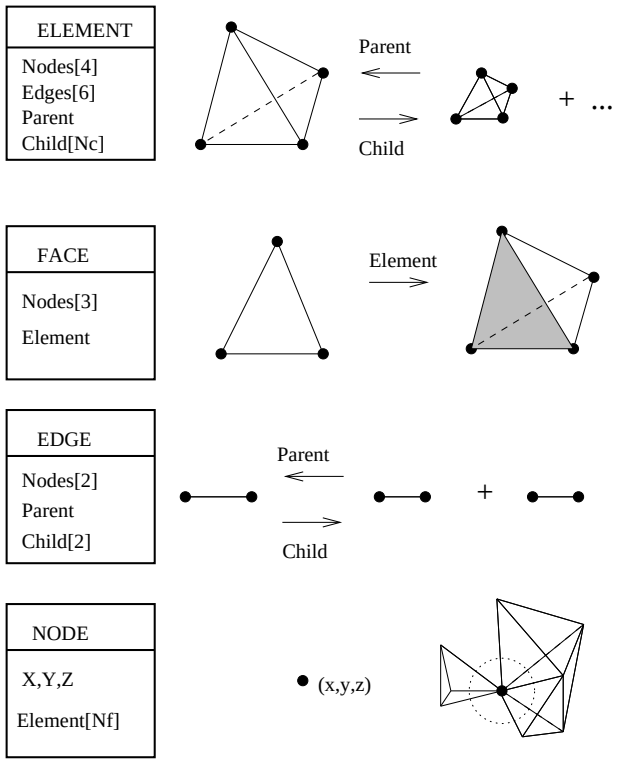

FIG. 1. *Mesh Data-Structures in TETRAD*

For parallelisation of TETRAD, there are two main data-structure issues. The first is how to partition a hierarchical mesh, the second is that we require new data-structures to support parallel partitioning of the mesh.

There are two main options for partitioning a hierarchical mesh. The first is to partition the grid at the root or coarsest level. In this case, to achieve good load balance, the graph of the coarse mesh need to be weighted by the number of leaf elements that are children of the coarse mesh element. This approach has a number of advantages. The local hierarchy is maintained on a processor and thus all parent/child interactions (such as refinement/de-refinement) are local to a processor. The partitioning cost will also be low, as the coarse mesh is generally quite small. The main disadvantage of this approach however, is that for comparatively small coarse meshes with large amounts of refinement, it is difficult to get a good partitioning, both in terms of load balance and communication requirements.

The second approach is to partition the leaf level mesh, i.e. the actual computational grid. The pros and cons of this approach are the opposite of those with the coarse level partitioning. In particular, the quality in terms of load balance and cut-weight of the partition is likely to be better, albeit at the expense of a longer partitioning time. Also, the data-structures have to be more complicated as hierarchical operations (such as multigrid V-cycles) are no longer necessarily local to a processor. This may, for example, lead to slower de-refinement as communication would now be required.

One may consider other partitioning strategies other than these, such as partitioning at all levels of the mesh, but in many cases, this will not bring any advantage and will introduce unnecessary complication. The approach taken for parallelising TETRAD is that of partitioning the coarse mesh. The only disadvantage of this, that of possible suboptimal partition quality, can be avoided if the initial, coarse mesh is scaled as one adds more

processors.

Given a partitioned mesh, we will also need new data-structures in order to support inter-processor communication and to ensure data consistency. Data consistency is handled by assigning ownership of mesh objects (elements, faces, edges and nodes). As is common in many solvers such as those used by [1] we use halo elements, a copy of inter-processor boundary elements (with their associated data) used to reduce communication overheads. In order to have complete data-structures (eg elements have locally held nodes) on each processor, we also have halo copies of edge,node and face objects. If a mesh object shares a boundary with many processors, it may have a halo copy on each of these. All halos have the same owner as the original mesh object. In situations where halos may have different data than the original, the original is used to overwrite the halo copies and thus is definitive. This is used to help prevent inconsistency between the various copies of data held.

The updating of halo data (eg at the end of a time-step) requires communication between a mesh object and its copies. We store the remote locations of halos in a one way linked list. This is used (rather than an array) as it is not known a-priori how many many halos any mesh object might have. Each link in the list consists of a processor-pointer pair precisely locating a halo on a remote processor. This enables updates to proceed without searching on the remote processor. When new elements are created in the inter-partition boundary regions, these linked lists for fast communication need to be set up for the new elements. This requires a pointer to be sent to the original mesh object from its halo to be stored in the relevant linked list. This communication is in the opposite direction from the links already set up. Without adding a communication link in this direction (from halo to original), we would have to search for the mesh object for which we need to add the link, because of the lack of inherent structure in the mesh. This is clearly not a scalable approach and thus we insist that the links between boundary mesh objects and their halos are two-way.

3 Parallel Adaptivity Algorithms

To understand the algorithms required for parallel adaptivity, we first need to outline those required in the serial case. TETRAD uses a similar strategy to that described in [2]. Edges are marked for refinement/de-refinement based on some criteria. This may be an error estimate or may simply utilise gradients. Elements with all edges marked for refinement are refined regularly into eight children. To remove the remaining hanging nodes, we use so-called 'green' refinement. This places an extra node at the centroid of the element. All nodes in the element are joined to this new node creating a varying number of child elements. The types of refinement are illustrated in Figures 2 and 3. Green elements therefore provide a link between regular elements of differing levels of refinement. One restriction we make however is that green elements may not be refined further. This is because so doing can adversely affect the mesh quality [4].

3.1 Green Tetrahedra and Recursive Searching

In the situation where we require further refinement of a green element, we will first de-refine the green elements and re-refine regularly. Not surprisingly, this has a knock-on effect. Consider Figure 4. If edge a is marked for refinement, the element T_1 will have to be refined. As T_1 is green, we will have to de-refine it and re-refine. This leads to a need to refine edge b which is shared with element T_2. This will therefore need to be refined in a similar manner. To check for situations like this, we use a depth-limited recursive search. This search runs

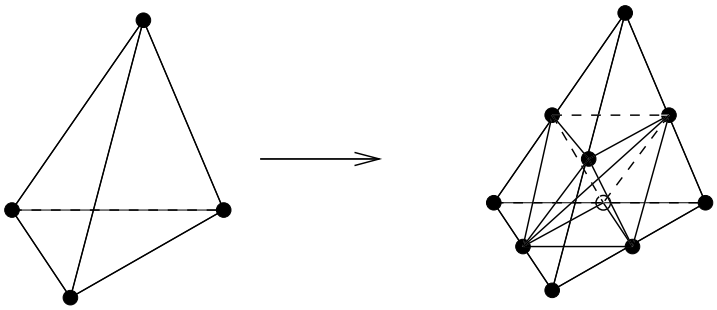

FIG. 2. *Regular Refinement dissecting interior diagonal*

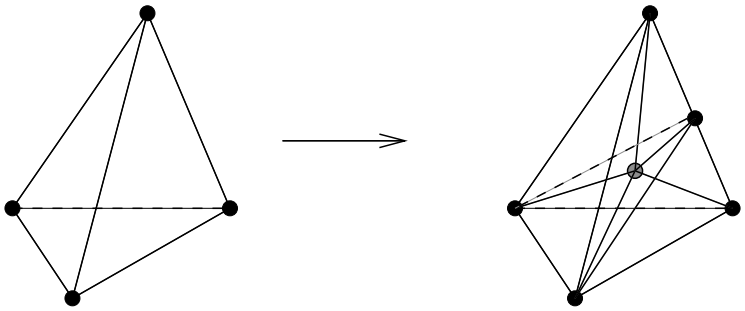

FIG. 3. *Green Refinement by the addition of an interior node*

through the list of green elements for de-refinement and subsequent refinement. For each of these elements, we check all the surrounding elements with which we share an edge. Any of these elements which are green and one level higher in the mesh hierarchy (the situation in Figure 4) will also be marked for de-refinement and subsequent refinement. We now recurse, using the newly found green as the base of the search. This search is clearly limited in depth of recursion by the maximum depth of refinement in the mesh and termination is therefore ensured.

The problem with parallelising this search is that the search progresses through the domain spatially and may well leave the the subdomain held on any given processor. Therefore, as we search, we construct a list of those elements that are found in the search and have halos. We communicate globally to check whether all the lists are empty (the termination condition) and if not, pass a message to each of the halos of the elements on the list. The recursive search starts again on these halos. This not only ensures consistency of

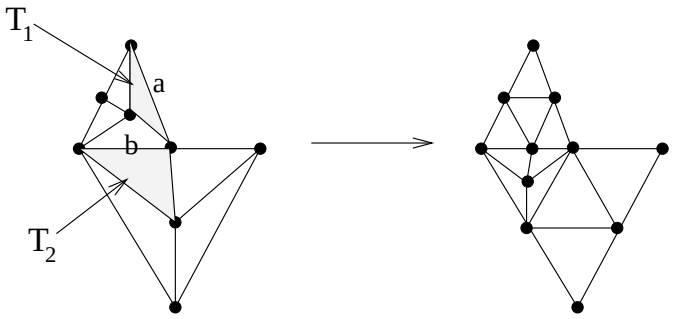

FIG. 4. *Knock on effect of refining green elements*

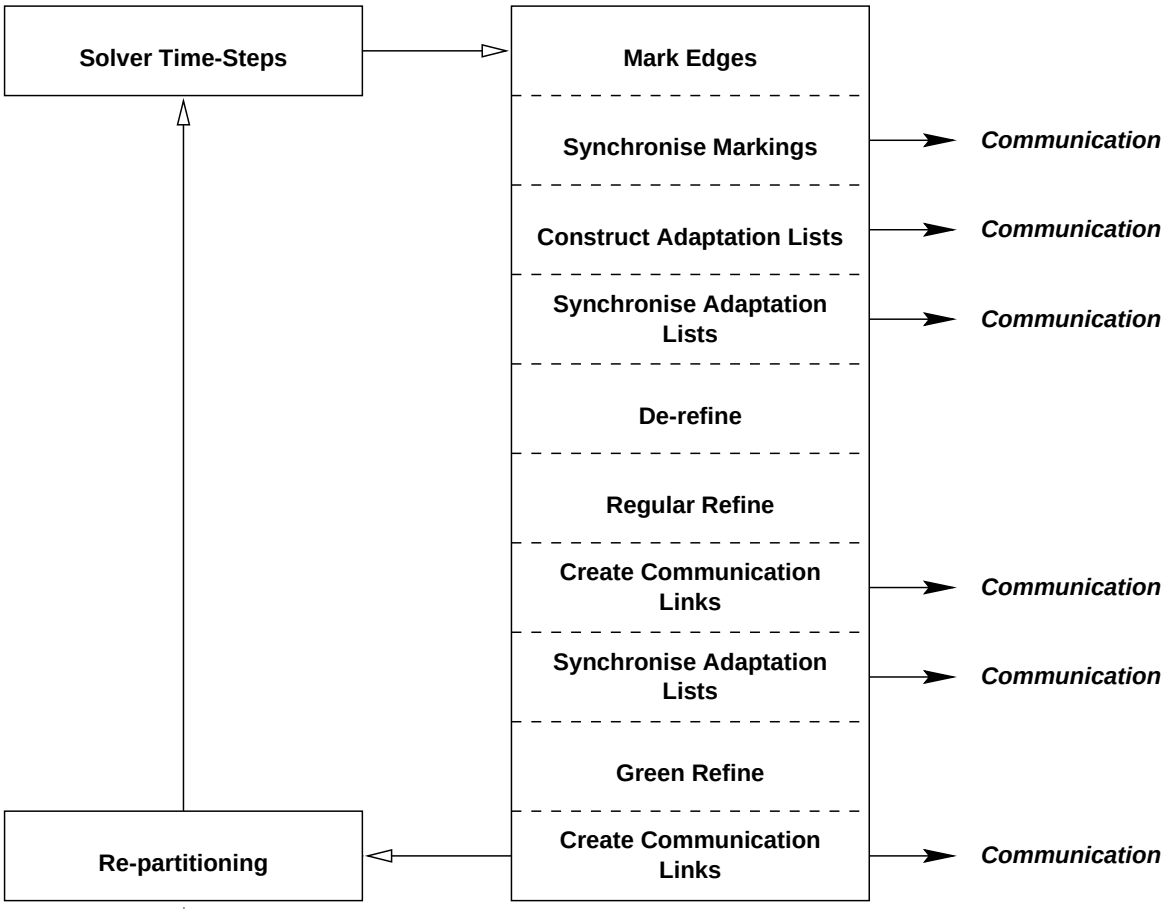

FIG. 5. *The Parallel TETRAD algorithm*

the mesh, as all elements marked for de-refinement and subsequent refinement will have their halos similarly marked, but also ensures that the mesh matches across partition boundaries. Again, the depth of the mesh provides a limit, this time on the number of exchanges of halos that take place. However, not all processors will necessarily have elements on which to start a search, and these processors will be idle until a global check of communication list sizes is made. These processors are not permitted to continue with further parts of the refinement process as the search may enter their sub-domain at a later stage.

The algorithm for the parallel version of TETRAD is shown in Figure 5. In general, the parallel version follows the same pattern as the serial code, but with many extra communication steps which ensure consistency of the distributed data. Note that we use a BSP style of programming [5]. Each processor works in 'supersteps' where no communication is done. These are separated by communications where the data is updated across processors.

3.2 Parallel Mesh Consistency

The structure of the parallel version of TETRAD is shown in Figure 5. In general, the parallel version follows the same pattern as the serial code, but with many extra communication steps which ensure consistency of the distributed data. Note that we use a BSP style of programming [5]. Each processor works in 'supersteps' where no

communication is done. These are separated by communications where the data is updated across processors.

Unlike the BSP paradigm, however we do not have all the data held consistently at each of the superstep boundaries. For example, in the de-refine phase, an edge may be de-refined on one processor, whilst its halo copies may be left intact on others. This inconsistency may persist throughout the regular refinement phase, together with the related creation of communication links. Green refinement is in some sense a 'patching-up' of the regular mesh: making sure we have no hanging nodes. It is therefore appropriate that it is in this phase we ensure all our original and halo edges match up.

Another problem with mesh consistency is that of making sure we adapt all halos in the same manner as the original elements. There are two approaches to tackling this problem. The first proceeds by adapting the original elements and later having an update phase that adapts the halos in a similar manner. The second, which we use is to note that as the serial code creates lists of elements and edges to be adapted, we only have to ensure that if these lists contain a boundary element, its counterpart on neighbouring processors contain the halos. Once both these adaptations have taken place, we need to put in place the communication links between the original element and its halos. As there is no way of directly locating the new halos from a remote processor, we utilise the existing links for the parent elements in the hierarchy. Despite this however, the cost of creating these communication links is high.

A further issue is that of load balancing. Any partition should be chosen to optimise the solver's load balance and to minimise the solver's communication. An adaptive scheme will therefore inherit this partition. It would be inappropriate to repartition the mesh for adaptation as in general, the partitioning required for balanced adaptivity would be quite different from that required for a balanced solver. Adaptivity tends to concentrate on regions where the solution is developing rapidly, with the rest of the mesh unaffected. Furthermore, the cost of re-partitioning would far outweigh any gain in adaptivity performance. The use of adaptivity will destroy existing load balance. It is therefore necessary to re-partition the mesh after an adaptive step to ensure solver efficiency. This would ideally utilise a tool such as Jostle [8, 9] which is designed for distributed meshes.

The communications patterns used in adaptivity are of vital importance given the high proportion of communication to computation. In order to retain portability we use MPI [3] with non-blocking communications to perform our message passing. Nonblocking communication is known to be vital for efficient communication and is also necessary to avoid deadlock. When we construct new inter-processor links, we do so for many elements at the same time. For most current machines, the communications cost model in [6] would indicate that the latency involved in sending each element link separately would be overwhelming. We therefore use packing to send one large message per processor rather than many small ones. This maximises the used bandwidth and minimises the incurred latency.

4 Results

In order to demonstrate both the feasibility and some of the problems of the approach described above we report on some early computational studies. The adaptivity is driven by marking edges to some prescribed formula rather than by error estimates for a given solution. This allows us to concentrate on the adaptivity without the need to consider solver specifics. We also do not include interpolations. Interpolation from the old solution

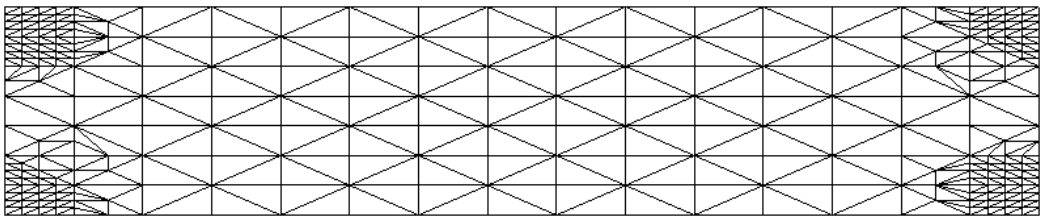

FIG. 6. *Problem 1 - slice through mesh at* $z = 0.17$

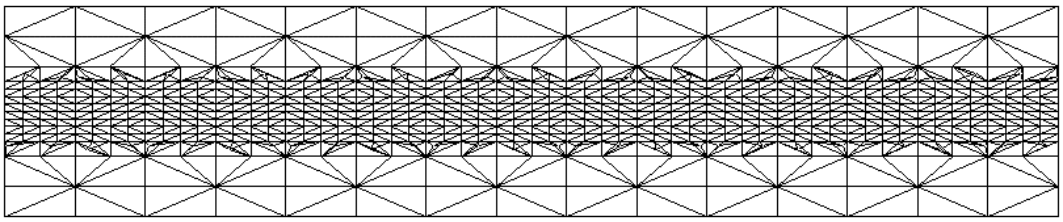

FIG. 7. *Problem 2 - slice through mesh at* $z = 0.1$

to the solution on the new mesh is one of the most computationally intensive parts of the adaptation procedure. Thus the results obtained are just dependent on the core adaptivity and communication routines.

We give results for a small network of SGI Indys with R4000 processors, connected by ethernet. We have also run the same experiments on a 4 node (R8000) SGI Power Challenge for which we obtain qualitatively similar results as for the workstation network. This is not surprising as although the Power Challenge has significantly better communications, it also has similarly improved computational speeds. We are currently porting the code to other machines (e.g. the Cray T3D).

We consider two simple examples, both based on a simple cuboid domain with an initial mesh of $3,675$ elements at a uniform density. We use an initial partition formed by recursive co-ordinate bisection. Problem 1 involves refinement in regions near the corners of the domain. We refine the mesh twice in these areas. This provides a test where most (but not all) of the adaptation is away from inter-processor boundaries. A 2D slice through the mesh is shown in Figure 6.

The second example is that of two levels of refinement in an area around a line which goes through the middle of the domain. This corresponds with a partition boundary given by co-ordinate bisection and thus many of the elements created will be in the halo region. This case would therefore not be expected to perform well. A 2D slice through the mesh is shown in Figure 7.

As would be expected, the algorithm does quite well for the first problem, even on this rather limited hardware setup. Two processors takes 3.3 seconds which is reduced to 1.8 seconds by using four processors. This quite surprising as two processor job does no refinement in the inter-partition boundary regions whereas there is moderate refinement in these areas for the four processor job. The refinement for this problem is also quite well balanced for both cases, the imbalance after adaptivity being less than 1%.

The algorithm does rather worse for the second problem. Two processors take 8.04 secs whilst four take 10.24 secs. Note that in this case adding more processors slows down the adaptivity. It is easy to see why this is the case. For this second example, the partition

boundary introduced when we move from two to four processors corresponds with the region being refined. Therefore most of the boundary elements in the four processor case are being refined and need to communicate.

These two examples represent the two extremes likely to be encountered in real situations. One has hardly any refinement of halo elements and the other has a large proportion of halo to normal elements refined. One would expect adaptivity in a normal solution process to lie somewhere between these two extremes. Furthermore, there is little de-refinement in the examples shown. Whilst de-refinement is computationally inexpensive, it also involves no communication, due to the hierarchical nature of our data-structures. We can thus only gain performance for problems with more de-refinement. Similarly, the addition of interpolation would also increase the computation to communication ratio and give more improvement.

5 Conclusion

The use of adaptive unstructured mesh algorithms on distributed memory machines requires the use of complex data-structures with rich inter-processor communication links. The algorithms used involve many communications to ensure consistency of the distributed mesh as it evolves. Despite this, initial results are encouraging.

References

[1] J. Cabello, *Parallel Explicit Unstructured Grid Solvers on Distributed Memory Computers*, Advances in Eng. Software, Vol. 23, No. 3, 1996, pp 189–200

[2] R. Lohner and J. D. Baum, *Adaptive H-Refinement on 3D Unstructured Grids for Transient Problems*, J. Num. Meth. Fluids, Vol. 14, 1992, pp 1407–1419

[3] The Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Int. J. Supercomputing App., Vol. 8, No. 3/4, 1994

[4] M. E. G. Ong, *Uniform Refinement of Tetrahedron*, SIAM J. Sci. Comp., Vol 15, No. 4, 1994

[5] L. Valiant, *A Bridging Model for Parallel Computation*, Communications ACM, Vol. 33, No. 8, 1990

[6] P. M. Selwood, N. A. Verhoeven, J. M. Nash, M. Berzins, N. P. Weatherill, P. M. Dew and K. Morgan, *Parallel Mesh Generation and Adaptivity: Partitioning and Analysis*, Proc. Parallel CFD '96, 1996 (in press).

[7] W. Speares and M. Berzins, *A 3D Unstructured Mesh Adaptation Algorithm for Time-Dependent Shock Dominated Problems*, Submitted to Int. J. of Num. Meth. in Fluids.

[8] C. Walshaw, M. Cross and M. Everett, *A Localised Algorithm for Optimising Unstructured Mesh Partitions*, Int. J. Supercomputing Appl, Vol. 9, No. 4, 1995.

[9] C. Walshaw, M. Cross and M. Everett, *Parallel Partitioning of Unstructured Meshes*, Proc. Parallel CFD '96, 1996 (in press)