Portable Parallel Adaptation of Unstructured 3D Meshes

P.M. Selwood, M. Berzins, J.M. Nash and P.M. Dew

School of Computer Studies
The University of Leeds
Leeds LS2 9JT, West Yorkshire
United Kingdom

Abstract. The need to solve ever-larger transient CFD problems more efficiently and reliably has motivated the use of mesh adaptation on parallel computers. We will discuss issues arising in the portable parallelisation of a general-purpose, unstructured, tetrahedral adaptivity code for use on distributed memory computers. In particular, we will discuss the parallelisation of complex hierarchical data-structures and issues of partitioning and communication links. We will also consider algorithmic and implementation issues of the code such as the parallelisation of a depth-limited recursive search. Results from a range of parallel computers will be given for challenging transient shock problems which demonstrate the success of the approach. Future developments to make the parallel programming of such applications more high level will be considered.

1 Introduction

The execution of computations based around adaptive unstructured 3d meshes provides some non-trivial problems when being executed on a parallel machine. Typically, the computations are based around distinct phases of execution, marked by the timesteps of the updated solution values, the mesh adaption points, and the possible redistribution of the mesh elements. One of the main problems to overcome is to be able to efficiently support the communication of the required data within these phases, given that the mesh elements have been partitioned among the processors. Since the partitioning of the mesh and the redistribution of mesh elements are both carried out at run-time, a static compiler analysis is inappropriate. However, although a phase of execution has the characteristic that the data communicated is unpredictable, the communication patterns are repetitive. This may be taken advantage of by run-time protocols to improve efficiency.

The use of a shared address space for implementing parallel software eases the changeover from serial to parallel execution, and leads to clear modular code. However, most machines which support a shared memory enforce sequential consistency, which often results in poor performance for this class of problem (based around producer-consumer sharing patterns). Some weaker form of consistency or domain-specific knowledge of the problem needs to be employed in order to gain good parallel efficiency.

2 The Parallel Adaptation of Unstructured 3D Meshes

2.1 A parallel adaptive algorithm

The software outlined in this subsection is based upon a parallel implementation of a general purpose serial code, TETRAD (TETRahedral ADaptivity), for the adaptation of unstructured tetrahedral meshes [11]. The technique used is that of local refinements/derefinements of the mesh to ensure sufficient density of the approximation space throughout the spatial domain, Ω, at all times. A more complete discussion of the parallel algorithms and data structures may be found in [12, 13, 14].

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 a complete mesh hierarchy is maintained 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.

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.

1. There are two main options for partitioning a hierarchical mesh. The first is to partition the grid at the root or coarsest level, $\mathcal{T}_0$. This has a number of advantages. The local hierarchy is maintained on a processor and thus all parent/child interactions (such as refinement/derefinement) 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 may be difficult to get a good partitioning, both in terms of load balance and communication requirements.
   The other main 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. However, the data-structures have to be more complicated as hierarchical operations, such as multigrid V-cycles and derefinement for example, are no longer necessarily local to a processor (and are therefore likely to be slower).
   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.

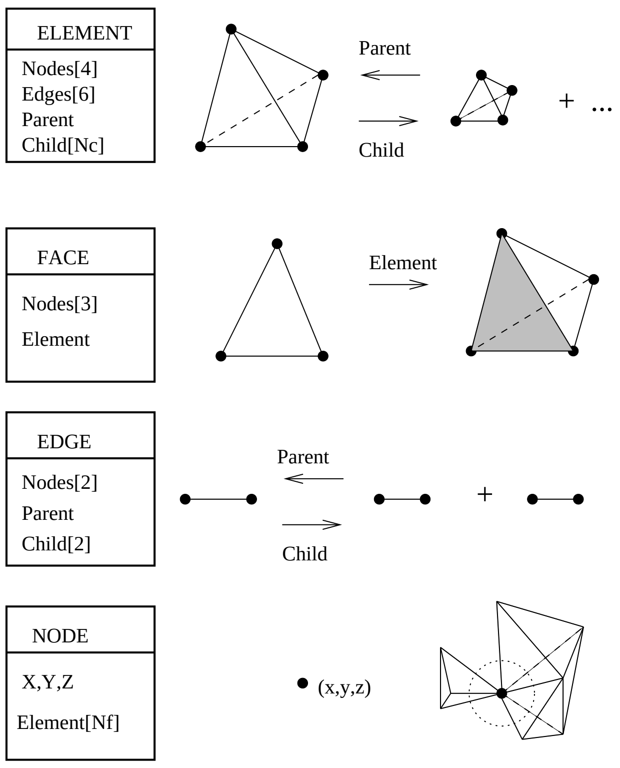

Fig. 1. Mesh Data-Structures in TETRAD

2. Given a partitioned mesh, we 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 [15] 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 (e.g. 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.

Adaptivity algorithms Both TETRAD ([11]) and its parallel implementation, PTETRAD ([12]), use a similar strategy to that outlined in [16] to perform adaptivity. Edges are first marked for refinement/derefinement (or neither) according to some estimate or indicator (provided as part of the parallel solver: see

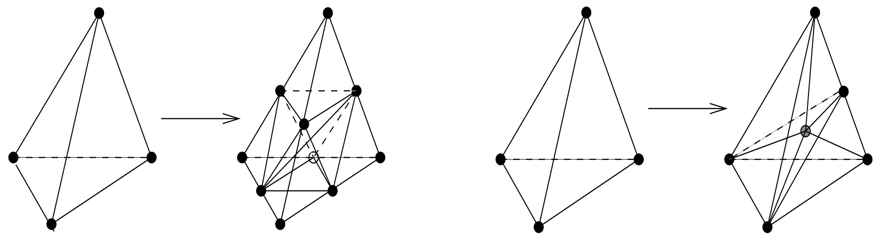

Fig. 2. (a) Regular Refinement dissecting interior diagonal; (b) Green Refinement by the addition of an interior node

2.2 below for example). Elements with all edges marked for refinement may then be refined regularly into eight children. To deal with the remaining elements which have one or more edge to be refined we use so-called "green" refinement. This places an extra node at the centroid of each element and is used to provide a link between regular elements of differing levels of refinement. The types of refinement are illustrated in Figure 2. An important restriction that is made is that green elements may not be further refined as this may adversely affect mesh quality ([17]). Instead, they are first removed and then uniform refinement applied to the parent element.

Immediately before the refinement of a mesh, the derefinement stage occurs. This may only take place when all edges of all children of an element are marked for derefinement and when none of the neighbours of an element to be deleted are green elements or have edges which have been marked for refinement. This is to prevent the deleted elements immediately being generated again at the refinement stage which follows. A further necessary constraint is that no edges or elements at the coarsest level, $\mathcal{T}_0$, may be derefined.

For further details of the implementation of these adaptive algorithms using MPI ([18]) please refer to [12]. This paper discusses important issues such as performing parallel searches in order to allow refinement of edges of green elements (which requires coarsening followed by regular refinement), maintaining mesh consistency and dealing with halo data in parallel.

2.2 A parallel finite volume solver

In order to apply the above adaptive algorithm to systems of PDEs of the form (??) a parallel solver is also required. The data structures supported by TETRAD have been used with both finite element and finite volume solvers (cell-centred and cell-vertex), however in this paper we restrict our numerical experiments to a cell-centred finite volume scheme.

The scheme that we use is applicable when (??) represents a system of hyperbolic conservation laws of the form

$$\frac{\partial \underline{u}}{\partial t} + \frac{\partial \underline{F}(\underline{u})}{\partial x} + \frac{\partial \underline{G}(\underline{u})}{\partial y} + \frac{\partial \underline{H}(\underline{u})}{\partial z} = 0 \, , \tag{1}$$

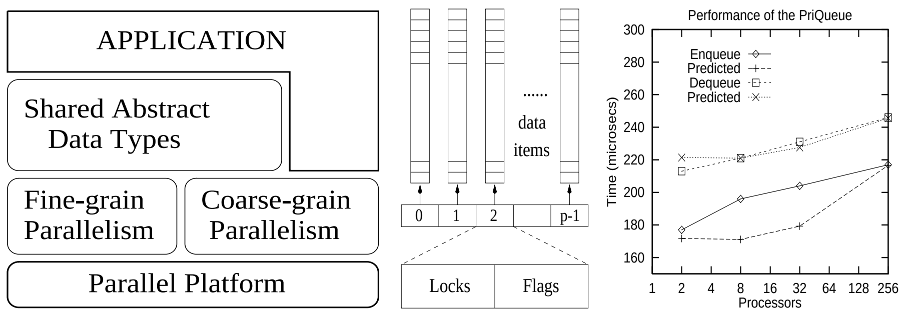

Fig. 3. (a) Support for portable applications; (b) A PriQueue implementation; (c) Performance on the Cray T3D

such as the three-dimensional Euler equations for example, and is a parallel version of the algorithm described in detail [11]. This is a conservative cell-centred scheme which is a second-order extension of Gudunov's Riemann problem-based scheme ([19]), using MUSCL-type piecewise linear reconstructions of the primitive variables within each element ([20]). Although the time-stepping is explicit it is executed in two distinct phases: a non-conservative predictor-type update (referred to in [20] as the "Hancock step") followed by a second half-time-step based upon the application of the underlying conservation law. Implicit in this numerical method is the need to solve a Riemann problem at each element interface at each time-step – although this is only done approximately using a modified form of the approximate solver described in [21].

The parallel version of the solver is straightforward to implement due to the face data structure that exists within the adaptivity software (see Figure 1 for example). To avoid any conflicts at the boundary between two sub-domains a standard "owner computes" rule is used for each of the faces when solving the approximate Riemann problems to determine fluxes. The use of halo elements ensures that the owner of each face has a copy of all of the data required to complete these flux calculations provided the halo data is updated twice for each time-step (i.e. immediately before the Hancock step and then again before the second half-time-step).

3 Shared Abstract Data Types for Portable Performance

The use of MPI allows the PTetrad code to be readily ported between parallel platforms. However, this does not imply that the same solution is the most optimal one on all platforms, or that using some other form of communications mechanism might give significant performance improvements (see Section 4.3). This section describes the application of *Shared Abstract Data Types* (SADTs) [4] to support such performance requirements.

3.1 Background

SADTs are an extension of ADTs to include concurrency. A serial ADT supports a well-defined interface, for example, the Enqueue and Dequeue of elements from a Priority Queue (PriQueue), hiding the internal implementation from the programmer. An SADT instance may be made visible to multiple processors, which may then concurrently invoke operations. The abstraction barrier enables implementations to take advantage of parallelism where appropriate, while shielding the user from details such as communication and synchronisation.

The Leeds studies have used the generic software structure as shown in Figure 3(a). The SADTs are used to hide the often complex concurrency issues involved in the use of fine-grain parallelism to support irregular forms of parallelism [?], as well as making use of coarse-grain parallelism. The application is written using a combination of coarse-grain parallelism, to structure the main phases of the code, and the SADTs, to support more dynamic forms of parallelism.

Weakened forms of data consistency can also be used to maximise performance, where this will not affect the correctness of the application using it [4, 1]. For example, the PriQueue may not be seen in the same state by all processors, such that a Dequeue may not strictly receive the highest priority element. This enables more efficient parallelisation methods which can segment the PriQueue elements across the processors, making effective use of high degrees of parallelism, and reducing communications and synchronisation overheads.

There are many related research efforts in this area, including *Distributed Shared Abstractions (DSAs)* [1], *Information Sharing Abstractions* [6] and *Parallel Abstract Data Types* [2], all of which aim to provide the applications programmer with high level abstractions of sharing in parallel systems.

3.2 Example: A Priority Queue

An example implementation of the PriQueue on the Cray T3D machine is shown in Figure 3(b). Each processor holds a segment of the PriQueue elements in its local memory, which are ordered by their priority. An associated local data structure holds associated information to govern the access of the elements. The implementation makes use of the SHMEM library to provide high bandwidth and low latency access to the local memories by other processors. The cyclic access of the segments by the processors distributes the priorities approximately evenly. A processor is then guaranteed to remove one of the p highest priorities, when there are p processors. This weaker form of semantics allows the highly concurrent implementation given above, resulting in scalable performance characteristics as the number of processors grow.

The graph in Figure 3(c) shows that the time to complete a Dequeue and Enqueue (for 1000 single word elements per processor under continuous access) grows only slowly as the number of processors increase. This results in an increase in throughput from $11,400$ Enqueues per second for 2 processors, up to $1,190,700$ Enqueues per second for 256 processors, due to the lack of any significant serial bottlenecks in the implementation. The use of the PriQueue to support the travelling salesman problem [3] has demonstrated that its weaker data consistency semantics does not have an impact on the quality of the load balancing.

3.3 The Application of SADTs to the PTetrad Software

As described above, the two key ideas behind the support for high performance SADTs are concurrency and weak data consistency. The segmentation of the PriQueue elements across the processors allows for highly concurrent access, and thus scalable performance. This form of segmented access is also clearly visible in the PTetrad code when partitioning the mesh elements. The weak consistency of the PriQueue elements (meaning that there is not a struct global ordering on the element priorities) reduces the overheads of communication and synchronisation, supporting good practical performance. This is also visible in the PTetrad code, in that the shared halo elements only need be fetched from their home location at specific points in the execution of the code, and then subsequently accessed locally.

4 Supporting Portable Parallel Adaptation

4.1 Related Work

Support at the lowest level comes from the use of adaptive paging protocols, which can take some advantage of the repetitive communications patterns to reduce the overall network traffic. The Reactive NUMA (R-NUMA) system [?] is a cache coherency protocol which combines the advantages of a conventional Cache Coherent NUMA (CC-NUMA) protocol with the Simple COMA (S-COMA). CC-NUMA improve data locality by caching shared data accesses (invoking the appropriate coherency protocol). S-COMA additionally allows the local memory of the processor to be used to store pages of shared data (using the same coherency protocol). This can potentially improve data locality by utilising the larger size of the main memory, but operating system overheads make this a more expensive option. An R-NUMA system combines the two approaches. *Reuse* pages contain data which is frequently accessed locally and *communication* pages are mainly used to exchange data between processors. The former can use the S-COMA protocol to reduce the network traffic caused due to cache capacity misses, and the latter can use the CC-NUMA approach to allow the sharing of sparse data at the cache-line level. The system can dynamically decide when a page switches between reuse and communication by noting the number of capacity and conflict misses. Applying the system to a partitioned mesh allows internal mesh elements to be located on reuse pages and the shared (halo) elements to use communication pages. This is under the assumption that these two distinct types of mesh elements can be arranged to lie on distinct pages, which would imply some form of domain-specific knowledge about the application being executed.

Support at the intermediate level is characterised by the provision for a generic framework for expressing irregular data structures and unstructured computations. This is typically through the use of both static compiler analysis, and cache coherency protocols which can take advantage of unpredictable but repetitive communications patterns. As an example, the C** language uses data-parallelism to perform parallel operations on a data collection within a global namespace. The support for data-parallelism allows a static compiler analysis to identify distinct phases of execution which require communications. However, the compiler does not attempt to identify the communications patterns in each phase - this is predicted by a run-time cache protocol. A sequentially consistent shared memory is the standard model used by parallel machines, but unfortunately introduces significant inefficiencies for typical producer-consumer sharing patterns [?]. The C** coherency protocol is divided into two stages. The first stage incrementally builds a schedule to support the given communications, by using a handler at each directory controller to note the incoming requests. The second stage uses this schedule at the beginning of the subsequent phases of execution in order to prefetch the expected data, with any unsatisfied requests being incrementally added to the schedule. The protocol includes coalescing neighbouring cache blocks, to be transferred using bulk messages. Limitations of the approach include an initially empty schedule for each phase, resulting in no cache blocks being requested, and schedule deletions only being supported by clearing the current schedule. The protocol approach is similar to the Chaos runtime system [?]. However, Chaos requires the protocol stages to be explicitly defined within the application, and does not provide the facility for the incremental update of a schedule, which is more appropriate to an adaptive code.

Support at the high level is typified by the use of the skeleton/template approach, where domain-specific knowledge of the type of application to be executed can be used to support high performance in a portable manner. An example is the M-Tree abstract data type [10], which aims to capture the data structure and computational structure of adaptive numerical problems, using a regional mesh tree (in which each node represents a region of the domain and children specify sub-domains). Example applications are in the area of the adaptive-mesh heat flow problem, Adaptive Multigrid and Barnes-Hut. A number of first-order functions can be specified by the programmer to support basic queries and updates on a tree node. Higher-order functions can specify particular computational patterns for the given application, such as performing a reduction on all nodes at a given level using some user-defined operator. This structured abstraction provides the opportunity to investigate detailed issues which are common to the application domain, such as the optimal dynamic load balancing method and caching technique. Related studies have used the terms *Distributed Shared Abstractions* (DSAs) [1] and *Information Sharing Mechanisms* [6], and *Shared Abstract Data Types*, as described in Section 3. The common characteristic is the representation of shared abstractions as structures that may be internally distributed across the nodes of a machine, so that their implementations can be altered which maintaining software portability.

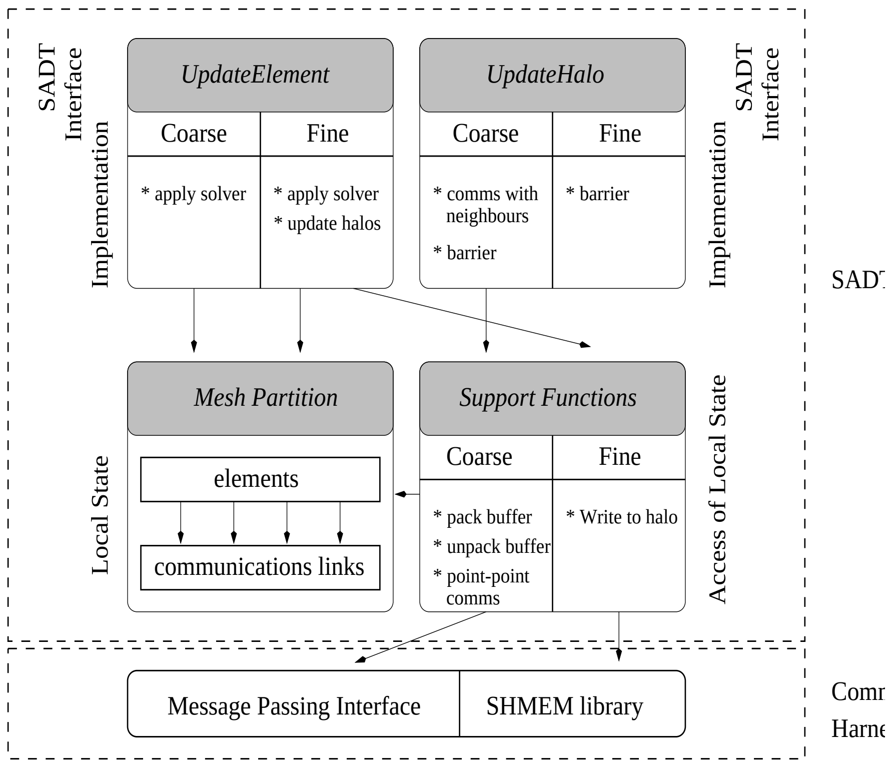

Fig. 4. The support of the PTetrad code using an SADT

4.2 The SOPHIA Interface

....

```
    for each local element e, execute UpdateElement(e);
    UpdateHalo();
```

4.3 Implementation Issues

Figure 4 demonstrates the current approach which is being taken to the support of the PTetrad code in an efficient and portable manner. The Sophia interface is reflected in the SADT interface, shown here by the *UpdateElement()* and *UpdateHalo()* functions. The figure shows two example implementations of these functions, using either coarse-grain or fine-grain parallelism. In the coarse-grain approach, all elements will be updated by the solver, followed by a communications phase in which a message containing updated halo information is sent to each neighbouring mesh partition. The fine-grain approach updates the halo information as the solver is applied to the local elements. In both cases, weak data consistency is applied such that the halo updates are only guaranteed to have completed after the processors barrier synchronise. The former approach is typically used in conjunction with a message passing library, such as MPI, where the overheads in accessing the network are substantial. The latter approach, using the SHMEM library in this case, can be used to overlap the network access with subsequent local computation by the solver, given sufficiently low software overheads for network access. Employing weak data consistency allows the blocking of communications in the case of MPI, and the potential to pipeline communications for SHMEM, supporting efficient network access.

4.4 Experiments

....

5 Concluding Remarks

....

References

1. C. Clemencon, B. Mukherjee and K. Schwan, *Distributed Shared Abstractions (DSA) on Multiprocessors*, IEEE Transactions on Software Engineering, vol 22(2), pp 132-152, February 1996.
2. J. Darlington and H. W. To, *Building Parallel Applications Without Programming*, Abstract Machine Models for Highly Parallel Computers, (eds J.R. Davy and P.M. Dew) Oxford University Press, pp 140-154, 1995.
3. P. M. Dew and J. M. Nash, *The High Performance Solution of Irregular Problems*, MPPM'97: Massively Parallel Programming Models Workshop, Royal Society of Arts, London, November 1997 (to be published by IEEE Press).
4. D. M. Goodeve, S. A. Dobson and J. R. Davy, *Programming with Shared Data Abstractions*, Irregular'97, Paderborn, Germany, June 1997.
5. K. Gharachorloo, S. V. Adve, A. Gupta and J. H. Hennessy, *Programming for Different Memory Consistency Models*, Journal of Parallel and Distributed Computing, vol 15, pp 399-407, 1992.
6. L. V. Kale and A. B. Sinha, *Information sharing mechanisms in parallel programs*, Proceedings of the 8th International Parallel Processing Symposium, pp 461-468, April 1994.
7. W. F. McColl, *An Architecture Independent Programming Model For Scalable Parallel Computing*, Portability and Performance for Parallel Processing, J. Ferrante and A. J. G. Hey eds, John Wiley and Sons, 1993.
8. J. M. Nash, P. M. Dew and M. E. Dyer, *A Scalable Concurrent Queue on a Message Passing Machine*, The Computer Journal 39(6), pp 483-495, 1996.
9. J. M. Nash, *Scalable and Portable Performance for Irregular Problems Using the WPRAM Computational Model*, To appear in Information Processing Letters: Special Issue on Models for Parallel Computation.
10. Q. Wu, A. J. Field and P. H. J. Kelly, *Data Abstraction for Parallel Adaptive Computation*, in M. Kara, J. R. Davy, D. Goodeve and J. Nash eds., Abstract Machine Models for Parallel and Distributed Computing, IOS Press, pp 105-118, 1996.

11. W. Speares and M. Berzins, *"A 3-D Unstructured Mesh Adaptation Algorithm for Time-Dependent Shock Dominated Problems"*, Int. J. Num. Meth. in Fluids, 25, 81–104, 1997.
12. P.M. Selwood, M. Berzins and P.M. Dew, *"3D Parallel Mesh Adaptivity: Data-Structures and Algorithms"*, Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing, SIAM, 1997.
13. P. Selwood, N. Touheed, P.K. Jimack, M. Berzins and P.M. Dew, *"Parallel Dynamic Load-Balancing for the Solution of Transient CFD Problems Using Adaptive Tetrahedral Meshes"*, To appear in Proc. of Parallel CFD 97 Conference, May, 1997, Manchester, UK.
14. P. 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"*, Parallel CFD – Proc. of Parallel CFD 96 Conference (ed. A.Ecer, J.Periaux, N.Satufoka and P.Schiano), Elesvier Science BV, 1997.
15. J. Cabello, *"Parallel Explicit Unstructured Grid Solvers on Distributed Memory Computers"*, Advances in Eng. Software, 23, 189–200, 1996.
16. R. Löhner, R. Camberos and M. Merriam, *"Parallel Unstructured Grid Generation"*, Comp. Meth. in Apl. Mech. Eng., 95, 343–357, 1992.
17. M.E.G. Ong, *"Uniform Refinement of Tetrahedron"*, SIAM J. Sci. Comp., 15, 1134–1144, 1994.
18. Message passing Interface Forum, *"MPI: A Message Passing Interface Standard"*, Int. J. of Supercomputer Applications, 8, no. 3/4, 1994.
19. S.K. Godunov, *"A Finite Difference Method for the Numerical Computation of Discontinuous Solutions of the Equations of Fluid dynamics"*, Mat. Sb., 47, 357–393, 1959.
20. B. van Leer, *"On the Relation Between Upwind Difference Schemes"*, SIAM J. Sci. Stat. Comp., 5, 1–20, 1984.
21. A. Harten, P.D. Lax and B. van Leer, *"On Upstream Differencing and Godunov Type Schemes for Hyperbolic Conservation Laws"*, SIAM Rev., 25, 36–61, 1983.