CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 1(1), 51–62 (SEPTEMBER 1989)

Supersonic fluid flow in parallel with an unstructured mesh

ROY D. WILLIAMS

Concurrent Computation Project
California Institute of Technology 356-48
Pasadena, CA 91125, USA

SUMMARY

An algorithm has been implemented for time-accurate solutions of the two-dimensional compressible Euler equations using an irregular triangular mesh. The code runs on distributed or shared memory or sequential machines, and is written using the Distributed Irregular Mesh Environment (DIME). DIME is a programming environment for calculations with such meshes, with adaptive mesh refinement and dynamic load balancing. Results are presented for an example of a Mach 3 flow over a step, computed with a 32-processor NCUBE hypercube.

INTRODUCTION

The potential of a multiprocessor can only be realized if the problem at hand can be split into many small pieces which can run in parallel. The most cost-effective and scaleable multiprocessors are distributed-memory machines, in which not only the computation but also the data must be partitioned. In many cases, such as solving Laplace's equation on a regular mesh, this decomposition is natural and straightforward.

Many important problems cannot be easily decomposed, and one such example is the computation of high-speed compressible flows. The difficulty is that the solution is very inhomogeneous, containing shocks where the mesh and computational effort should be concentrated and also regions of near-uniform flow where the opposite is true. Additional meshing problems are posed by complex domain boundaries. Since the position of shocks is not known in advance, an adaptive mesh is called for, and a consequent need for dynamic load-balancing of the computation.

In addition to the ease of programming a regular mesh calculation, another advantage is that computations are efficiently performed with a vector machine. Unfortunately it is difficult to adaptively refine a regular mesh or fit complex boundaries. Rather than try to use such a regular or piecewise regular mesh, I have chosen to use a completely unstructured irregular mesh. Such meshes have the flexibility required for dynamic adaptive meshing of complex domains, but at the cost of more memory utilized in storing the logical structure of the mesh, and slower computation due to gather/scatter operations. But since the unstructured mesh can fit itself accurately to the boundaries and near-singularities of the problem domain, it must be the most efficient for sufficiently inhomogeneous problems.

1040-3108/89/010051-12$06.00

Received June 1989

Unstructured meshes have been widely used for calculations with conventional sequential machines. Jameson[1] uses explicit finite-element based schemes on fully unstructured tetrahedral meshes to solve for the flow around a complete aircraft, and other workers[2] have used unstructured triangular meshes. Jameson and others[1–4] have used multigrid methods to accelerate convergence.

The solution algorithms employed in the papers[5] use explicit time stepping and require the use of artificial dissipation to stabilize the computations. The discrete equations used in the numerical procedures result from coupling different time-stepping schemes for the governing equations with the Galerkin finite element method[6] over a mesh of triangular elements. Although the algorithms can be used in a time-accurate mode, the emphasis here is on the solution of steady-state problems by stepping through the false transient with a relatively coarse mesh, then adaptively refining as the solution equilibrates.

DIME (Distributed Irregular Mesh Environment)[7] is a programming environment under development at Caltech for calculations with unstructured triangular meshes using distributed-memory machines. The environment provides adaptive refinement, dynamic load-balancing, and a tool for specifying and coarsely triangulating a domain. This coarse triangulation is loaded into a single processor of the machine, then refinement and balancing are used to create a computational mesh, domain-decomposed among all the processors of the machine. The user specifies a data structure to be associated with each node, with each element (triangle), and with each boundary node of the mesh; these data are manipulated by a user-program written in C using constructs such as `FORALLNODES...NEXTNODE`, which is a loop over all the nodes of the mesh.

THE DIME STRUCTURES

Figure 1a shows a simple mesh covering a rectangle, and Figure 1b shows its representation as nodes pointing to elements and elements pointing to nodes. There are extra data structures attached to boundary nodes which point to the next boundary node clockwise. Each of these three DIME structures also points to its respective user-data, which for the purposes of this paper contain the fluid simulation data such as velocity and density.

The mesh is connected locally, so that DIME is good for problems which are also local. Many scientific problems can be expressed as a set of local equations. When the mesh is split among the processors of the machine, the physical locality is preserved, in the sense that communication links are set up between processors only when the domains controlled by those processors have a common border. Physical locality does not necessarily mean that processors are locally connected in the machine; the communcation protocol used by DIME is a general message-passing system and the programmer does not need to know the connectivity of the machine.

Figure 2 shows the same mesh as in Figure 1 but split up among three processors. Where a processor–processor boundary passes through a node, copies of the node are kept, one for each participating processor. The user-data in each of these copies is identical. Thus each physical node of the mesh is represented by several copies, with the copies connected by communication links. DIME has only two forms of communication: either the processors share global data, or for each physical node

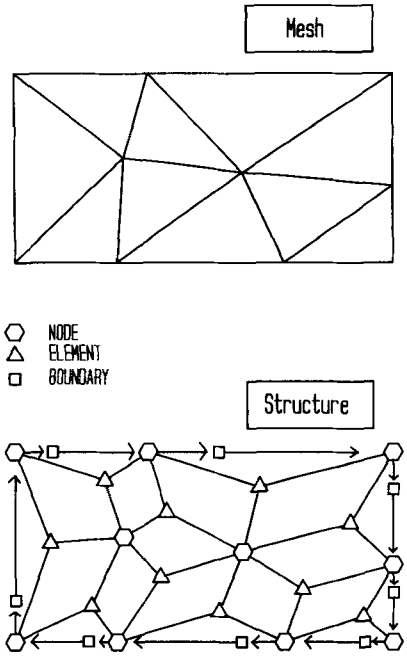

Figure 1. (a) an unstructured mesh (b) the DIME structure corresponding to the mesh, consisting of nodes connected to elements and elements connected to nodes, plus boundary structures

the copies share data. These communication links between node-copies are shown by arrows in Figure 2, and each is a *foreign pointer*, which consists of a processor number and a pointer within that processor. Notice also in Figure 2 that new boundary data structures have been set up for processor–processor boundaries in addition to those for physical boundaries. Thus a boundary structure can refer to a processor–processor or a physical boundary or both.

An application code for DIME consists of two parts: the DIME environment itself, which sets up these communication links, does refinement and generally keeps track of the mesh structure; and a user-program, which manipulates application-specific data attached to the DIME structures using the macros and functions provided by DIME.

The user-program written for a particular application will generally consist of each node gathering information from its neighboring elements, and each element gathering from its neighboring nodes, plus of course manipulation of the element and node data without reference to the neighbors.

DIME runs *loosely synchronously*, meaning that the program for each processor consists of alternate cycles of calculation and communication. A particular

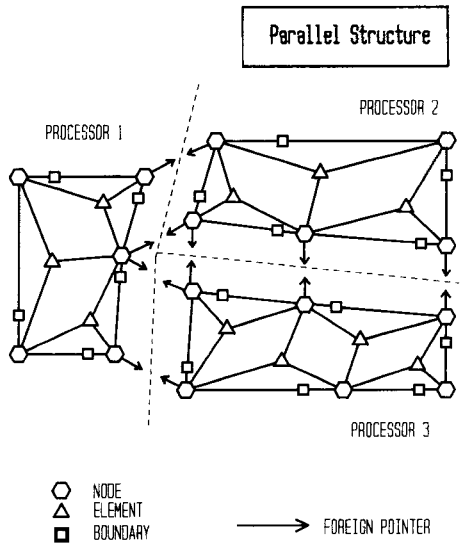

Figure 2. DIME structure for the mesh of Figure 1, split among three processors. Foreign pointers are shown as arrows and consist of a processor number and an address in that processor

communication section of one processor must correspond to the same section in the processors with which it communicates. A processor waits when another processor is not ready to communicate. The machine does not run completely synchronously, in the sense that during a computation cycle processors may run different parts of the code, but when communication occurs the receiving processor will be looking to the sender and expecting a message, and if that message does not arrive the machine deadlocks. The refinement process, for example, involves communication, so it must be done in all processors even if only one processor actually has any elements to refine. Another example might be choosing a timestep for the simulation, where the global maximum fluid speed might be required. Each processor calculates the maximum for its portion of the mesh then the function call `COMBINE` obtains the global maximum. The `COMBINE` must be called in all processors.

Just as `COMBINE` combines information from each processor and passes the result back to each processor, so the DIME macro `NODE_COMBINE` combines information from the copies of a node and passes the result back to each copy. Thus, for example, if each node is summing the data from each of its neighboring elements, there should be a `NODE_COMBINE` after the sum so that the part of the sum in neighboring processors is included.

DIME is written in C and runs with the Express operating system[8] on distributed or shared memory machines or sequentially. Express is the descendent of CrOS/Cubix, developed at Caltech. Having the code run sequentially is very useful for program

development and debugging. Since compiling and machine access are so much easier; there is the best of both worlds, with the ease of use of a sequential machine, and the speed of a parallel machine.

The user may adaptively refine the DIME mesh by selecting a set of elements to be refined and loosely synchronously calling the function `REFINE`. Load-balancing may be done by orthogonal recursive bisection[7] or the user may directly specify which elements are to be given to which processor. DIME provides many graphics functions, including contouring, vector plots, examining the logical mesh in detail, zooming and panning, and a Postscript hard-copy interface.

COMPRESSIBLE FLOW ALGORITHM

The algorithm of Löhner *et al.*[5], which is an explicit method of the Taylor–Galerkin family[6], was used, with an artificial viscosity to stabilize strong shocks. The governing equations are of advective type with no diffusion, where **U** is a vector containing the information about the fluid at a point. I have used bold symbols to indicate an *information* vector, or a set of fields describing the state of the fluid. In this implementation, **U** consists of density, velocity, and specific total energy (or equivalently pressure); it could also include other information about the state of the fluid such as chemical mixture or ionization data. **F** is the flux vector, and has the same structure as **U** in each of the two coordinate directions. In this paper,

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho \varepsilon \end{bmatrix} \quad \mathbf{F}_x = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho v u \\ u(\rho \varepsilon + p) \end{bmatrix} \quad \mathbf{F}_y = \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ v(\rho \varepsilon + p) \end{bmatrix}$$

where ρ is the density, $\mathbf{w} = (u, v)$ is the velocity, ϵ is the specific total energy, and p is the pressure. The equation set is completed by the addition of an equation of state: the fluid is assumed to be an ideal gas.

$$p = (\gamma - 1)\rho[\varepsilon - \mathbf{w}^2/2]$$

where γ is the ratio of specific heats, taken to be 1.4. The local sound speed is then given by $c^2 = \gamma p / \rho$.

The numerical algorithm is explained in detail in papers[5], so only an outline is given for advancing the solution from t to $t + \delta t$. The method uses linear triangular elements to approximate the fields and alternates between node-based and element-based representations of the fields. First a timestep is chosen. This is constrained by a Courant condition with a safety factor.

Step 1:

Assuming the node-based information vector **U** is known at time t, the node-based flux **F** can be found. Each element then gathers **U** and **F** from its neighboring nodes to obtain a first order element-based approximation to the information vector at time $t + \delta t/2$. The element-based flux can then be calculated.

Step 2a:

The nodal solution is advanced from t to $t+\delta t$ using the time- and space-staggered values from the first step. Each node gathers information from its neighboring elements, and a system of sparse linear equations is generated whose sparsity graph is the same as that of the mesh. These equations can typically be efficiently solved with three iterations of a relaxation solver.

Step 2b:

Boundary conditions are applied. This is somewhat delicate for hyperbolic flows[9]. There are three types of boundary for the simulation of a wind-tunnel; inflow, outflow, and solid wall. If the flow is supersonic, the outflow condition is irrelevant, since no information can propagate back into the bulk of the tunnel. For the inflow conditions, **U** is completely specified—density, velocity and pressure. At a solid wall, just one of the four components of **U** must be specified, which is that the velocity normal to the wall should be zero. After steps 1 and 2a, this will not be exactly true, so the velocity is projected to be parallel to the wall, and the pressure and density extrapolations are taken from neighboring internal nodes of the mesh.

Step 2c:

Artificial viscosity is applied. This is necessary to make the calculation stable. The underlying assumption of a discrete approximation to a continuum is that fields change slowly from mesh-point to mesh-point; without artificial viscosity even the continuum model develops discontinuities, so clearly this assumption would fail to be true for the discretized version. The artificial viscosity model is that of Lapidus[11]. A change $\delta\mathbf{U}$ in **U** is made which is diffusive:

$$\delta\mathbf{U} = \delta t \frac{\partial}{\partial \mathbf{l}}\left[k(x)\frac{\partial \mathbf{U}}{\partial \mathbf{l}}\right]$$

where the unit vector **l** is the local gradient of fluid speed:

$$\mathbf{l} = \nabla|\mathbf{w}|/|\nabla|\mathbf{w}||$$

and the diffusion coefficient k is given by

$$k = C\Delta_e \frac{\partial}{\partial \mathbf{l}}(\mathbf{w} \cdot \mathbf{l})$$

where C is a constant and Δ_e is the element area.

ADAPTIVE REFINEMENT

After the initial transients have dispersed and the flow has settled, the mesh may be refined. The criterion used for deciding which elements are to be refined is based on the gradient of the density. The user specifies a percentage of elements which are to be refined, and a criterion

$$R_e = \Delta_e \nabla \rho$$

is calculated for each element. A value R_{crit} of this criterion is found such that the given percentage of elements have a value of R_e greater than R_{crit}, and those elements are refined. The criterion is not simply the gradient of the density, because the strongest shock in the simulation would soak up all the refinement, leaving weaker shocks unresolved. With the element area, Δ_e, in the criterion, regions will 'saturate' after sufficient refinement, allowing weaker shocks to be refined. Only when the artificial viscosity coefficient C is reduced will the shocks become narrower, allowing further refinement.

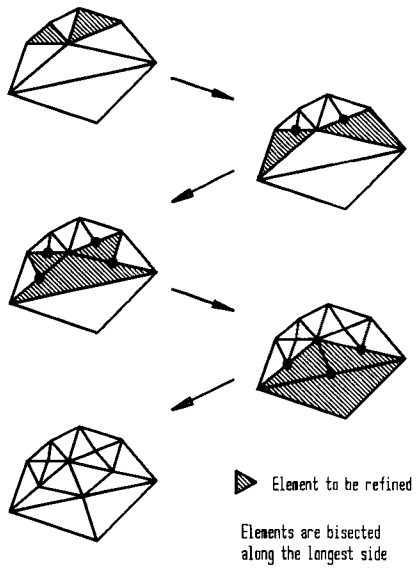

Figure 3 Refinement algorithm for DIME; elements are bisected on the longest side and non-conforming nodes (circles) made. The elements opposite these are refined and so on until all non-conforming nodes are eliminated

The refinement is done by the algorithm of Rivara[9], and is also explained in detail in Reference 7. Each nominated element is bisected by creating a new node at the midpoint of the longest edge and joining the new node to the opposite vertex of the element. This first stage is shown at the top of Figure 3. In general some of these new nodes will be 'non-conforming' (marked with circles in Figure 3). A non-conforming node is one which causes a geometrical triangle to be a logical quadrilateral, since on the other side of the refined element is an angle of 180 degrees. The elements opposite the non-conforming nodes (the logical quadrilateral elements) are marked for refinement and bisected along the longest side. The process continues until there are no non-conforming nodes.

After refinement the mesh may be 'topologically relaxed'[7]. For each non-boundary edge of the mesh there is an element on each side of the edge, forming a quadrilateral with the edge being one of the diagonals. If the sum of the two angles opposite the diagonal is greater than 180 degrees, then the diagonal is switched to the other possibility. Now the sum of the opposite angles must be less than 180 degrees since the sum of all four angles of a quadrilateral is 360 degrees. This process is continued until no further relaxations are possible. The reason for this is that after topological relaxation the mesh is now a Delaunay triangulation (dual of a Voronoi tesselation), so that each node is logically connected to its nearest neighbors. For elliptic problems this is important because the stiffness matrix of a Delaunay mesh now has the desirable characteristic of diagonal dominance[12], making the iterative solution of the finite element equations robust.

EXAMPLE

Figure 4 shows the flow-field resulting from Mach 3 flow over a step, computed with a 32-node NCUBE hypercube. This problem is that used by Woodward and Colella[13] in their benchmarking of compressible-flow algorithms. At the top is the mesh used for the computation, where the heavy lines mark splits between processors. Notice that each processor owns about the same number of elements. The lower three panels show velocity, density and pressure. There is a primary detached shock upstream, and part way down this primary shock, from the top of the wind-tunnel, a diffuse secondary shock splits from it.

In Figure 4 the contour lines are somewhat ragged, especially toward the downstream (right) end of the wind-tunnel where the mesh is coarse. In doing the simulation the artificial viscosity coefficient must be chosen: if it is too small the contour lines become ragged and eventually the whole calculation becomes unstable; if too large the shocks widen and diffuse and detail is lost.

It is difficult to know how to run this wind-tunnel simulation automatically; either with a predefined schedule of artificial viscosity, time-stepping and refinement, or some control process to manage these operating parameters. In particular it is difficult to know how far to run the simulation before refining the mesh, and to what extent to do so.

TIMING

Figure 5 shows timing results for 1, 4, 16, 64 and 256 NCUBE processors. The time per element per time-step is shown for the compressible flow algorithm against number of elements in the simulation. The curves end when the processor memory is full. Each processor offers a nominal 512Kb memory, but when all the software and communication buffers are accounted for, there is about 120Kb available for the mesh.

The time taken is a sum of computation and communication contributions. For N elements divided among n processors we expect the number of elements to be N/n per processor so that the computation time per element is proportional to $1/n$. The communication time is dependent on the number and size of the messages. The size of the messages is proportional to the number of nodes at

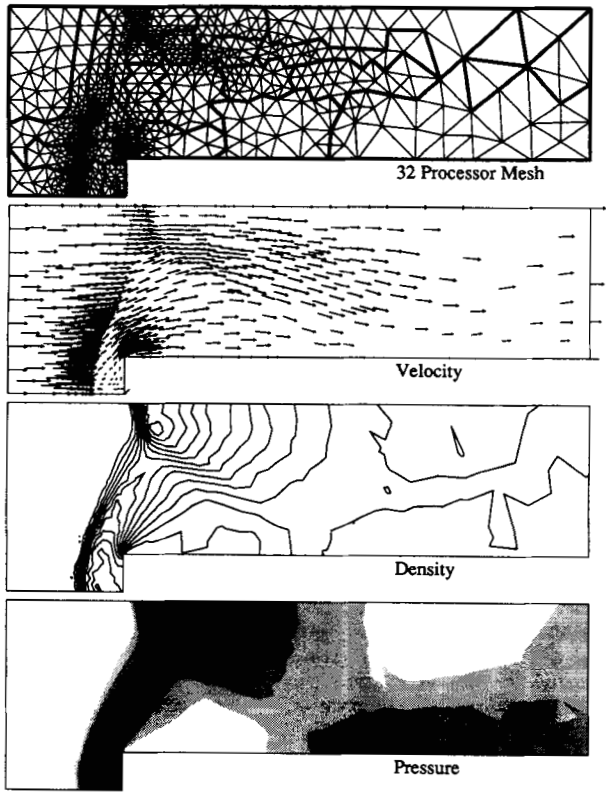

Figure 4. Mach 3 flow over a step, coming in at the left of the picture. The 32-processor mesh is heavily refined in the vicinity of the shocks

processor–processor boundaries, which is proportional to $(N/n)^{\frac{1}{2}}$. The communication time is also proportional to the greatest Hamming distance between the processors of the machine, which for a hypercube is $\log_2 n$, plus some constant for latency. I shall use the term speed-up in the sense of constant grain-size rather than constant problem size; thus speed-ups can be extracted by comparing the greatest speed from a given number of processors with the asymptotic speed of the single processor, the results being,

Table 1

Processors	Speed-up	Efficiency
4	2.9	72%
16	12	75%
64	36	56%
256	112	44%

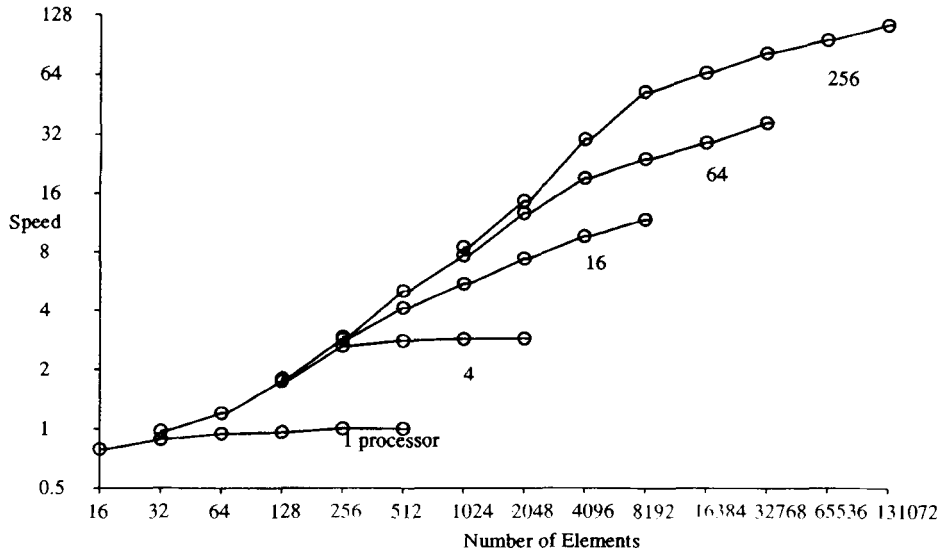

Figure 5. Timings for the compressible flow code with an NCUBE hypercube. This is a log–log plot of speed (elements processed per unit time) against the number of elements, with various numbers of processors. The speed is normalized to the asymptotic speed of a single processor. The curve for a particular number of processors should become flat as the communication time becomes small compared with computation time

It is clear from Figure 5 that boundary effects are still very important even when the memory of each processor is full; these speed-up values would improve greatly with faster communication. The efficiency is poorer for larger numbers of processors, because the communication protocol is designed to send messages from any processor to any other, so that the communication times have a factor $\log_2 n$. If the load-balancer and the communication protocol were designed to take advantage of the architecture of the machine, so that the mesh-neighbor processors were also machine neighbors, this factor could be eliminated, but at the expense of portability.

FUTURE WORK

As mentioned above a pressing need is for robust automatic control of the simulation; deciding when the false transients have passed and thus that it is time to refine, how much to refine, and what the artificial viscosity coefficient should be.

The discretization of the boundary conditions leaves much to be desired, since the extrapolation from interior to boundary mentioned above is only a first-order process compared to the second-order advection solver used here, creating a large diffusion at the boundaries, possibly explaining the diffuseness of the secondary shock in Figure 4.

In addition to load-balancing the spatial mesh, the simulation should be load-balanced in time; meaning that different parts of the mesh should run at different time-steps. This is called domain-splitting[14]. The local time-step is controlled by the Courant condition, so for roughly uniform velocity, the time-step should be proportional to the element size. In the present implementation the time-step is a global minimum of this Courant condition; in the future each processor will have its own time-step. To keep synchronization, each time-step must be an integer quotient of some global maximum step, and it would be convenient if each time-step were to be forced to be a power of two multiplied by a global minimum time-step.

A more accurate solver for compressible flow is flux-corrected transport[15]. In this case, a low-order scheme, such as that implemented here, is used for a predictor step, and a higher-order scheme also used for the step; the higher-order scheme adds non-physical oscillations which are removed using information from the lower-order scheme.

The most desirable extension of this work would be a full three-dimensional solver. Unfortunately maintaining a fully unstructured mesh in 3D is much more difficult than in 2D, partly because it is so hard to visualize a 3D unstructured mesh. Some work has been done with partially unstructured meshes in 3D[1]. As noted above, the 512Kb memory of an NCUBE processor is mostly taken with code and buffers, leaving little room for actual mesh, so that the boundary part is not negligible compared to the internal part. In 3D this problem is exacerbated, because the surface area to volume ratio becomes $N^{\frac{2}{3}}$ rather than the $N^{\frac{1}{2}}$ in 2D. One solution is of course, a larger memory: the MarkIIIfp hypercube offers 4096Kb per processor. Another solution is faster communication, which is currently being installed.

ACKNOWLEDGEMENTS

This work was supported in part by Department of Energy Grant DE-FG03-85ER25009.

REFERENCES

1. A. Jameson and T.J. Baker, 'Euler calculations for a complete aircraft', in *10th Int. Conf. on Num. Meth. in Fluid Mech.*, Beijing 1986, F.G. Zhang and Y.L. Zhu eds., Springer-Verlag Lecture Notes in Physics 264, 1986, p 334.
   A. Jameson, T.J. Baker, and N.P. Weatherill, *Calculation of Inviscid Transonic Flow over a Complete Aircraft*, AIAA Paper 86-0103, 1986.
   A. Jameson and T. J. Baker, *Improvements to the Aircraft Euler Method*, AIAA Paper 87-0452, 1987.
   A. Jameson, *Successes and Challenges in Computational Aerodynamics*, AIAA Paper 87-1184, 1987.
2. D. J. Mavriplis, *Accurate Multigrid Solution of the Euler Equations on Unstructured and Adaptive Meshes*, NASA-CR 181679 also ICASE Report 88-40, 1988.
   E. Perez *et al.*, 'Adaptive full-multigrid finite element methods for solving the two-dimensional Euler equations, in *10th Int. Conf. on Num. Meth. in Fluid Mech.*, Beijing 1986, F.G. Zhang and Y.L. Zhu eds., Springer-Verlag Lecture Notes in Physics 264, 1986, p 523.
   D. G. Holmes and S.H. Lamson, 'Adaptive triangular meshes for compressible flow solutions', *Proc. Int. Conf. Numerical Grid Generation*, Landshut, 1986, J. Hauser and C. Taylor eds., Pineridge, 1986, p 413.
3. D.J. Dannenhoffer and R. L. Davis, 'Adaptive grid computations for complex flows', *Proc. 4th Int. Conf. Supercomputing*, Santa Clara, CA, 1989, vol II p 206.

4. W. J. Usab and E. M. Murman, *Embedded Mesh Solution of the Euler Equations using a Multiple Grid Method*, AIAA Paper 83-1946-CP, 1983.
5. Löhner, K. Morgan and O.C. Zienkowicz, 'An adaptive finite element procedure for compressible high speed flows', *Comp. Meth. in Appl. Mech. and Eng.*, **51**, 441 (1985).
R. Löhner, K. Morgan and O.C. Zienkowicz, 'The solution of non-linear hyperbolic equation systems by the finite element method', *Int. J. Num. Meth. in Eng.*, **4**, 1043 (1984).
R. Löhner, K. Morgan, J. Peraire, O.C. Zienkowicz and L. Kong, *Numerical Methods for Fluid Mechanics II*, K.W. Morton and M.J. Baines, eds., Clarendon Press, Oxford, 1986.
R. Löhner, 'Finite elements in CFD: what lies ahead?', *Int. j. numer. methods eng.*, **24**, 1741 (1987).
6. J. Donea, 'A Taylor–Galerkin method for convective transport problems', *Int. j. numer. methods eng.*, **20**, 101 (1984).
7. R.D. Williams, 'DIME: a programming environment for unstructured triangular meshes on a distributed-memory parallel processor', *Proc. 3rd Int. Conf. on Hypercube Parallel Processors and Applications*, G.C. Fox ed., Pasadena, CA, January 1988; also Caltech Concurrent Computation Project Report C3P-502, 1987.
8. *EXPRESS: An Operating System for Parallel Computers*, ParaSoft Corporation, Pasadena, California, 1987.
G.C. Fox, M. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon and D.W. Walker, *Solving Problems on Concurrent Processors*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.
9. M.-C. Rivara, 'Design and data structure of fully adaptive multigrid, finite-element software', *ACM Trans. in Math. Software*, **10**, 242 (1984).
10. M.J. Abbett, 'Boundary condition calculation procedures for inviscid supersonic flow fields', *Proc. AIAA Compuational Fluid Dynamics Conf.*, 1973.
C.C. Lytton, 'Solution of the Euler equations for transonic flow over a lifting aerofoil', *J. Comp. Phys.* **73**, 395 (1987).
T.H. Pulliam, 'Characteristic boundary conditions for the Euler equations', in *Numerical Boundary Condition Procedures*, NASA Conf. Pub. 2201, NASA Ames, Moffett Field, CA, P. Kutler ed.
11. A. Lapidus, 'A detached shock calculated by second order finite differences', *J. Comput. Phys.*, **2**, 154 (1967).
12. D.M. Young, *Iterative Solution of Large Linear Systems*, Academic Press, New York, 1971.
G.H. Golub and C.F. van Loan, *Matrix Computations*, Johns Hopkins UP, Baltimore, 1983.
13. P. Woodward and P. Colella, 'The numerical simulation of two-dimensional fluid flow with strong shocks', *J. Comp. Phys.*, **54**, 115 (1984).
14. R. Löhner, K. Morgan and O.C. Zienkowicz, 'The use of domain splitting with an explicit hyperbolic solver', *Comp. Meth. in Appl. Mech. and Eng.*, **45**, 313 (1984).
15. E. S. Oran and J. P. Boris, *Numerical Simulation of Reactive Flow*, Elsevier, New York, 1987.
R. Löhner, K. Morgan, J. Peraire and M. Vahdati, 'Finite-element flux-corrected transport for the Euler and Navier–Stokes equations', *Int. J. Num. Meth. in Fluids*, **7**, 1093 (1987).