C3P-856 January 1990

Performance of a Distributed Unstructured-Mesh Code for Transonic Flow

Roy D. Williams

Concurrent Supercomputing Facility

California Institute of Technology 356-48

Pasadena, CA 91125

Abstract

An algorithm has been implemented for calculation of steady solutions of the two-dimensional Euler equations using an unstructured 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.

In this paper I present execution times for the code using different numbers of processors on three distributed memory machines machines, being a Meiko computing Surface with 32 processors, an NCUBE with 256 processors and a Symult S2010 with 192 processors. For comparison purposes I also present timings for the CRAY YMP, using one processor.

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 may be 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 dynamically 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.

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[2,3] have used unstructured triangular meshes. Jameson and others[4,5] have used multigrid methods to accelerate convergence. In this paper I have used the two-dimensional explicit algorithm of Jameson and Mavriplis[2].

An explicit update procedure is local, and hence well matched to a massively parallel distributed machine, whereas an implicit algorithm is more difficult to parallelize. The implicit step consists of solving a sparse set of linear equations, where matrix elements are non-zero only for mesh-connected nodes. Matrix multiplication is easy to parallelize since it is also a local operation, and the solve may thus be accomplished by an iterative technique such as Conjugate Gradient, which consists of repeated matrix multiplications. If, however, the same solve is to be done repeatedly for the same mesh, the most efficient (sequential) method is first decomposing the matrix in some way, resulting in fill-in. In terms of the mesh, this fill-in represents non-local connection between nodes: indeed if the matrix were completely filled the communication time would be proportional to N^2 for N nodes.

DIME (Distributed Irregular Mesh Environment)[4] 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 graphics interface. A 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 1 shows a simple mesh covering a rectangle, and Figure 2 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. 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

Figure 1: A triangular mesh with boundary.

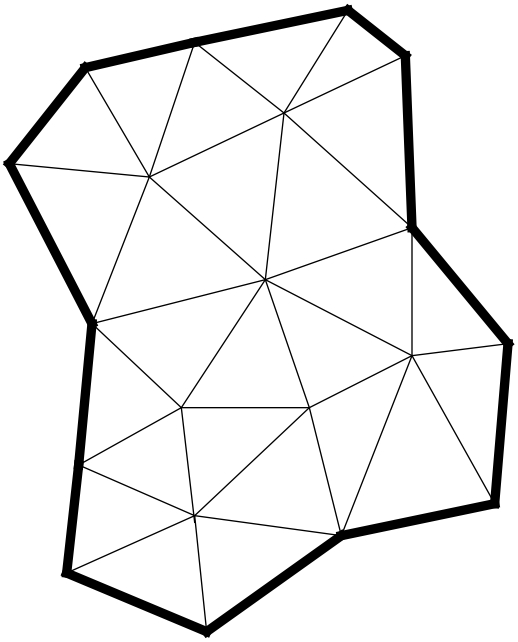
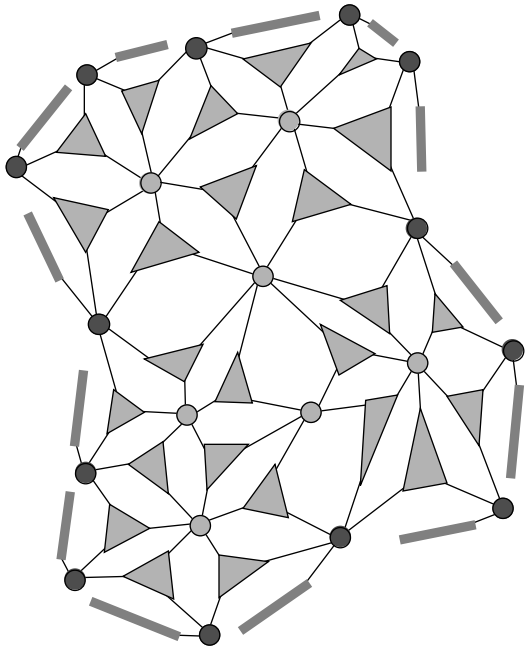


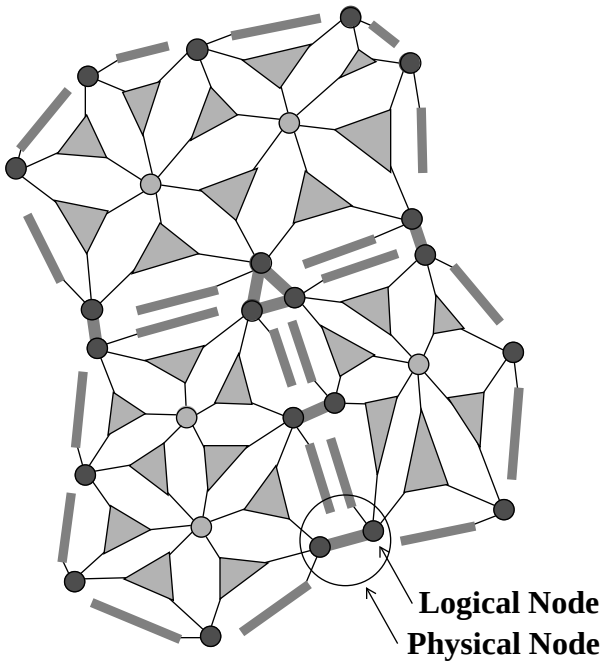
Figure 2: The same mesh as Figure 1, shown as separate nodes, elements, boundary nodes and boundary elements..



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 may be 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 the copies share data. These communication links between node-copies are shown by arrows in Figure 3, and each is a *foreign pointer*, which consists of a processor number and a pointer within that processor. Notice also in Figure 3 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 physical boundary or both.

Figure 3: The same mesh as Figure 2, decomposed between three processors. Nodes may be stored redundantly in several processors.



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 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[5] 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 bisected and loosely synchronously calling the function `REFINE`. Load balancing may be done by orthogonal recursive bisection[4] 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 governing equations are the Euler equations, which are of advective type with no diffusion,

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F}(\mathbf{U}) = 0$$

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, $w = (u, v)$ 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.

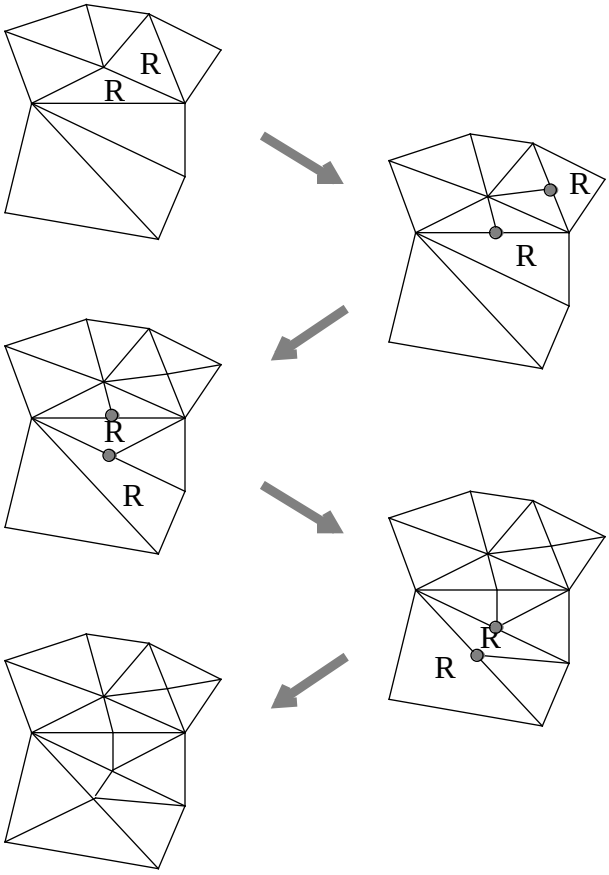
$$p = (\gamma - 1)\, \rho\, [\, \varepsilon - w^2/2]$$

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 Reference 2, so only an outline is given here. The method uses linear triangular elements to approximate the field. First a timestep is chosen for each node, which is constrained by a local Courant condition. The calculation consists of two parts:

Advection: At each node calculate **F** from **U**. Each element then averages **F** from its neighboring nodes, and

Figure 4: Refinement algorithm. Each chosen element is bisected along its longest side, then those elements opposite non-conforming nodes are bisected along their longest sides.



calculates the flux across each edge of the element, which is then added back into the node opposite the edge. This change in **U** is combined across the representations of the node in different processors. If the node is at a boundary which is a hard surface, a modification is made so that no flux of mass or energy occurs through the surface.

Artificial Dissipation: The artificial dissipation is calculated as a combination of approximations to the laplacian and the double laplacian of **U**, involving a combine step for each. The double laplacian is only used where the flow is smooth, to prevent dissipation of strong shocks.

The time-stepping is done with a five-stage Runge-Kutta scheme, where the advection step is done five times, and the dissipation step is done twice. Since advection takes one communiction stage and dissipation two, each full time step requires nine loosely synchronous communication stages.

Adaptive Refinement

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

$$R_e = \Delta_e \, | \nabla p |$$

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 pressure, 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.

The refinement (Figure 4) is done by the algorithm of Rivara[6], and is also explained in detail in Reference 4. Each nominated element (marked 'R' in the Figure) 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 4. In general some of these new nodes will be 'non-conforming' (marked with circles in Figure 4). 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. Notice that the actual number of elements refined may be greater than the number initially nominated for refinement.

After refinement the mesh may be 'topologically relaxed'[4]. 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 a Delaunay triangulation[7](dual of a Voronoi tesselation). For elliptic problems this is important because the stiffness matrix of a Delaunay mesh now has the desirable characteristic of diagonal dominance[7], making the iterative solution of the Finite-Element equations robust.

Example

Figure 5 shows the flow-field resulting from Mach 0.8 flow over a NACA0012 airfoil at 1.25 degrees angle of attack, computed with a 32-node machine. This problem is that used by the AGARD working group[8] in their benchmarking of compressible-flow algorithms. At the top left is the mesh used for the computation, with 5135 elements, after four stages of adaptive refinement. At top right is the logical structure of the mesh, showing elements and nodes separately. Notice that each processor owns about the same number of elements. At bottom left are Mach-number contours from this work, and at lower right is the same data from the AGARD benchmark group. The sonic line is shown by a heavy line in both plots.

Note the shock above the airfoil, and the corresponding increase in mesh density there and at the leading edge.

Timing

The efficiency of any parallel algorithm increases as the computational load dominates the communication load. In the case of a domain-decomposed mesh, the computational time depends on the number of elements per processor, and the communication time on the number of nodes at the boundary of the processor domain. If there are N elements in total, distributed among n processors, we expect the computation to go as N/n and the communication as the square root of this, so that the efficiency should approach unity as the square root of n/N.

I have run the example described above starting with a mesh of 525 elements, and refining 50% of the elements. In fact more than 50% will be refined because of the nature of the refinement algorithm: in practice it is about 70%. The refinement continues until the memory of the machine runs out.

Figure 6 shows timing results for 1, 4, 16, 64 and 256 NCUBE processors. The time taken per simulation 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 only about 120Kb available for the mesh.

Figure 7 shows the same curves for 1, 4, 16, 64, and 128 Symult processors, and Figure 8 the results for 1, 4, 16 and 32 processors of the Meiko computing surface. For comparison, Figure 9 shows the results for one head of the CRAY YMP, and also for the SUN Sparcstation.

Each Figure has diagonal lines to guide the eye; these are lines of constant time *per element.* We expect the curves for the sequential machines to be parallel to these because the code is completely local and the time should be proportional to the number of elements. For the parallel machines we expect the discrepancy from parallel lines to indicate the importance of communication inefficiency.

Conclusions

For the Symult machine we can extract speedup by measuring the distance between the curve and that for one processor, the results being:
4 processors: 3.2
16 processors: 7.9
64 processors: 19.5

The curved part of the curves for small numbers of elements indicates that communication is dominating; for example the 16-processor Symult is faster than the 64-processor with 525 elements because there are only 525/64 = 8 elements per processor, and communication dominates.

The NCUBE curves did not emerge from the communication-dominated phase due to lack of memory, so it is difficult to calculate a speedup.

The Symult curves show a slight non-linearity. This is because of the primitive load balancing method used. In Figure 5 we see several processor domains which are very long and thin, which becomes more common for larger number of processors, and is exacerbated by inhomogeneities in refinement, such as those caused by the shock. This means that the boundary length of some processors is more like the number of elements than its square root. Even one of these difficult domains slows the code because the code moves at the pace of the

slowest.

Another problem is that as the number of processors increases, there may be widely separated processors (in terms of the machine architecture) which must communicate. Again, one such message slows the whole machine.

Acknowledgement

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, eds. F.G. Zhang and Y.L. Zhu, Springer-Verlag Lecture Notes in Physics **264**, page 334;

A. Jameson, T.J. Baker, and N.P. Weatherill, *Calculation of Inviscid Transonic Flow over a Complete Aircraft*, AIAA Paper 86-0103;

A. Jameson and T. J. Baker, *Improvements to the Aircraft Euler Method*, AIAA Paper 87-0452;

A. Jameson, *Successes and Challenges in Computational Aerodynamics*, AIAA Paper 87-1184.

2 D. J. Mavriplis, *Accurate Multigrid Solution of the Euler Equations on Unstructured and Adaptive Meshes*, NASA-CR 181679 also ICASE Report 88-40;

3 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, eds. F.G. Zhang and Y.L. Zhu, Springer-Verlag Lecture Notes in Physics **264**, page 523;

D. G. Holmes and S.H. Lamson, *Adaptive Triangular Meshes for Compressible Flow Solutions*, Proc. Int. Conf. Numerical Grid Generation, Landshut, 1986, ed J. Hauser and C. Taylor, Pineridge, 1986, p 413;

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;

W. J. Usab and E. M. Murman, *Embedded Mesh Solution of the Euler Equations using a Multiple Grid Method*, AIAA Paper 83-1946-CP;

R. 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** (1985) 441;

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** (1984) 1043;

R. Löhner, K. Morgan, J. Peraire, O.C. Zienkowicz and L. Kong, in *Numerical Methods for Fluid Mechanics II*, eds. K.W. Morton and M.J. Baines, Clarendon Press, Oxford, 1986;

R. Löhner, *Finite Elements in CFD: What lies Ahead*, Int. J. Num. Meth. in Eng., **24** (1987) 1741.

4 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, ed. G.C. Fox, Pasadena, CA, January 1988; also Caltech Concurrent Computation Project Report C3P-502.

5 *EXPRESS: An Operating System for Parallel Computers*, ParaSoft Corporation, Pasadena, California;

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

6 M-C. Rivara, *Design and Data Structure of Fully Adaptive Multigrid, Finite-Element Software*, ACM Trans. in Math. Software, **10**, (1984) 242.

7 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.

8 AGARD Subcommittee C. *Test Cases for Steady Inviscid Transonic or Supersonic Flows*. AGARD FDP WG-07, Advisory Group for Aerospace Research or Development, January 1983.

Figure 5: Top left: Mesh used for calculation of flow field; Top right, logical structure of the mesh, split amongst 32 processors. Bottom left, contours of Mach number with spacing 0.05, and the sonic line marked with a heavy line; Bottom right, contours of Mach number from AGARD benchmark.

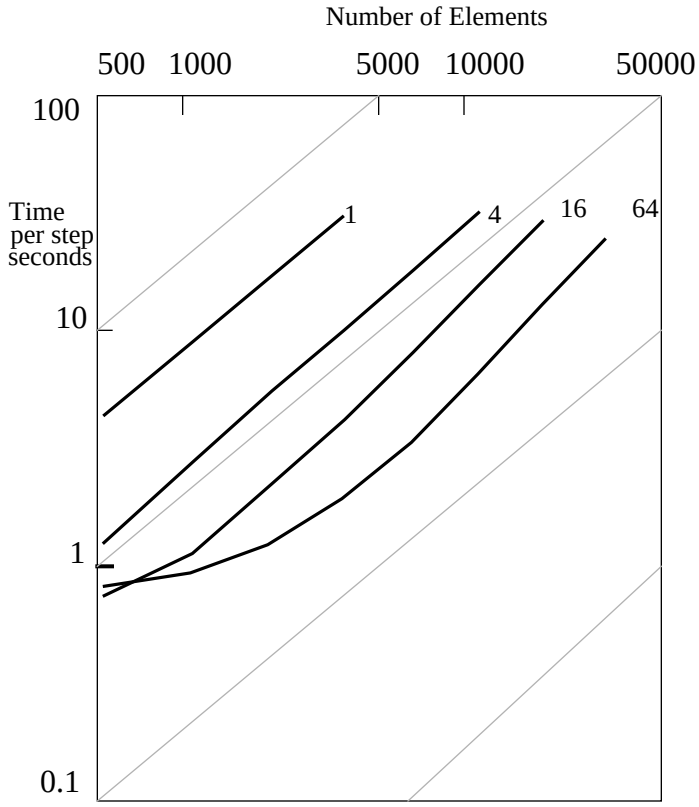


Figure 6: **Symult S2010**

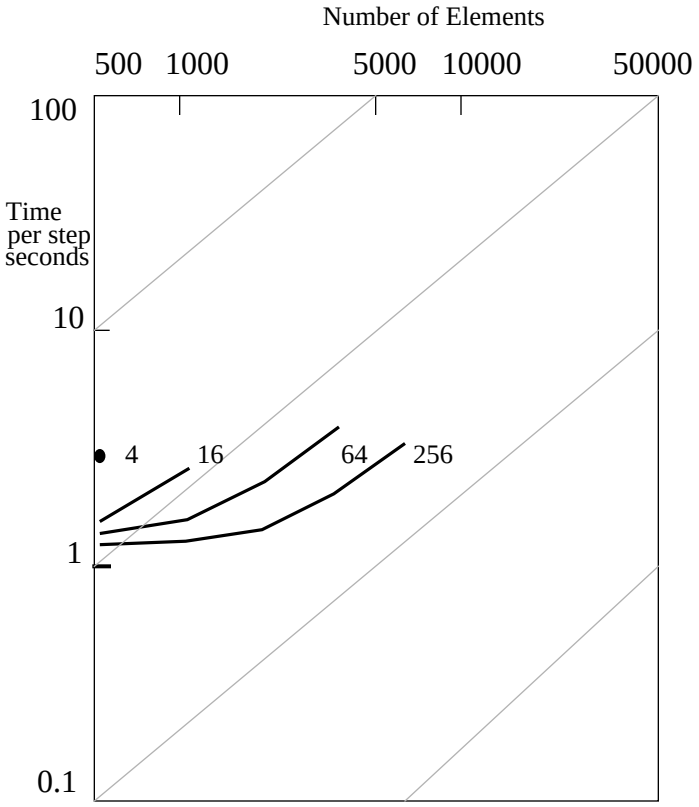


Figure 7: **NCUBE/1**

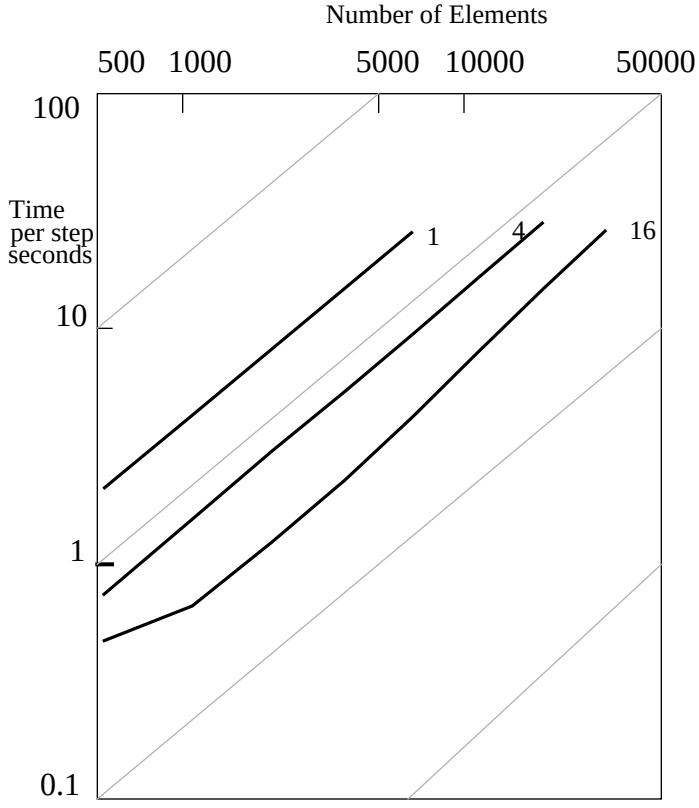


Figure 8: **Transputers**

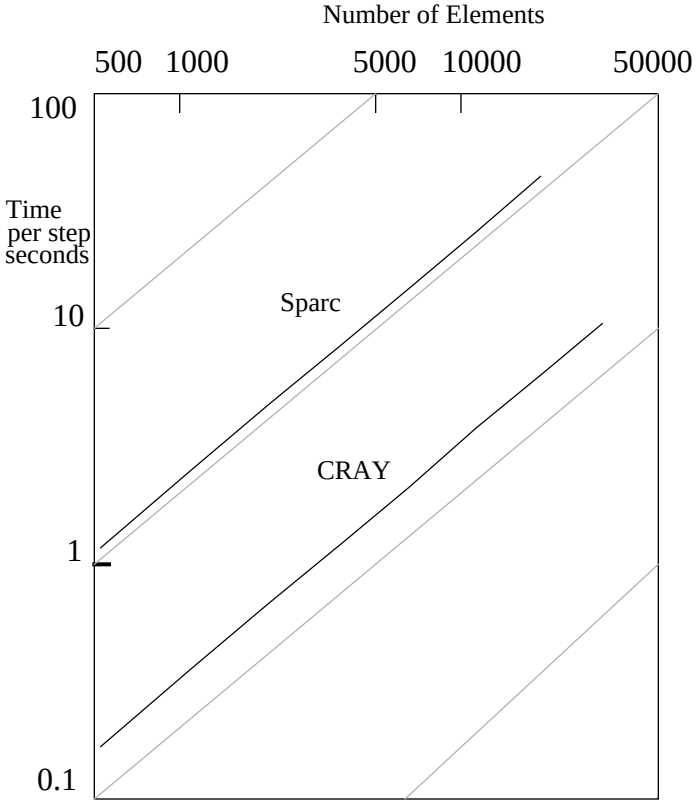


Figure 9: **CRAY & SUN Sparc**