C3P 715 February 1989

Distributed Irregular Finite Elements

R. D. Williams* and R. Glowinski**

*Concurrent Supercomputing Facility
California Institute of Technology
Pasadena, CA 91125 USA

**Mathematics Department
University of Houston
Houston, TX 77004 USA

Abstract

***DIME** is a programming environment for manipulation of irregular triangular meshes with a distributed memory machine. We have added a software layer **DIMEFEM** which manipulates Finite Element representations of functions and allows the stating and solving of variational statements. An application of **DIMEFEM** is an incompressible Navier-Stokes solver with a three-stage implicit time step, using quasi-optimally preconditioned conjugate gradient for the Stokes solver and Newtons' method in the space of quadratic Lagrangian elements for the nonlinear solver. The codes run on **NCUBE** hypercubes and transputer machines.*

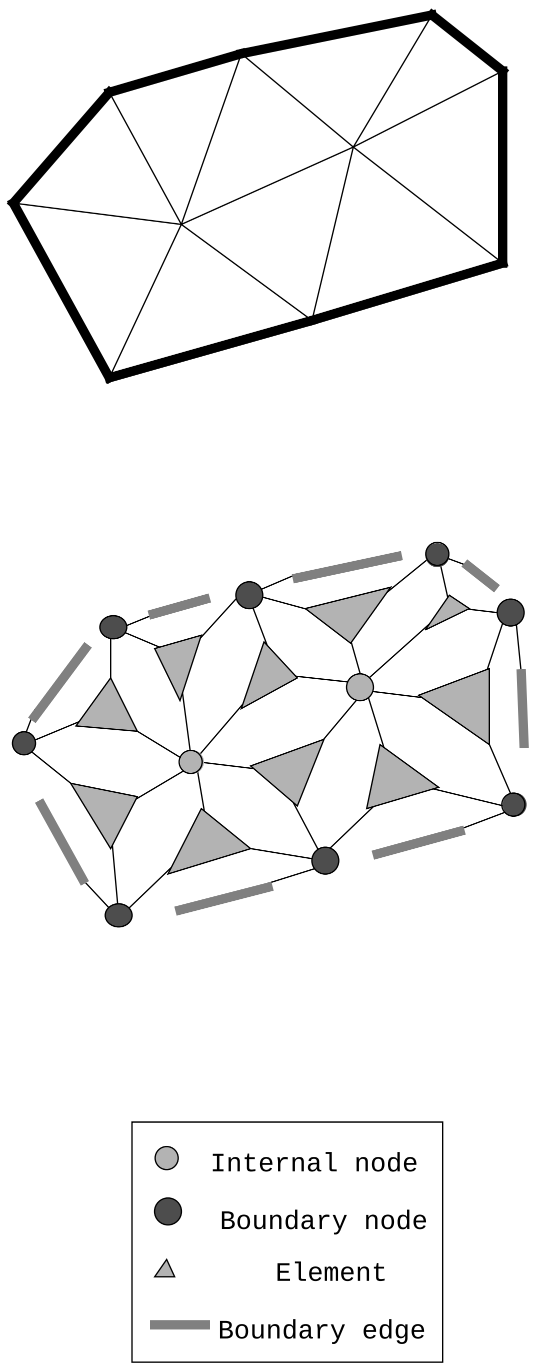

Figure 1: A triangular mesh and its logical structure. Nodes are connected to elements and elements are connected to exactly three nodes. Boundary nodes and boundary edges may carry extra data.

1. Introduction

Parallel computers offer potentially enormous increases in speed over sequential machines, and the greatest potential would seem to come from the seemingly infinitely extensible distributed memory machines. The achievement of this potential relies on efficient and portable software which takes advantage of all that the hardware has to offer. We can start either from a significant application code which has been debugged, tested, and most importantly accepted by a community of users, and attempt to ‘‘parallelize’’ it, or we can start from scratch.

An existing code can easily be made to run on a small number of processors if the computationally intensive parts vectorize well and a good compiler is available. If this is not the case then part or all of the code must be rewritten, with the probability of introducing new bugs and the possibility of losing the confidence of the user community. Hopefully whatever algorithm is used in the sequential code is optimal, but that algorithm may not be optimal for parallel operation, and indeed may be impossible to effectively parallelize.

Writing a code from scratch is of course much more difficult than cannibalizing parts from an existing code, but has the distinct advantages that appropriate parallel algorithms may be chosen *ab initio*, and that debugging is done on the same set of parallel machines which are used for development and for the final application code.

There is much research still to be done on parallel algorithms, which suggests a third method of creating application software; to make a parallel application *language*, so that different algorithms may be tested and characterized. We have chosen to take this layered approach to developing Finite Element software with distributed data and computation, based on auto-adaptive irregular triangular meshes.

In Section 2 we discuss motivation for using irregular meshes. Section 3 deals with DIME (Distributed Irregular Mesh Environment)[1], which maintains the mesh itself; reading and writing it and its data to disk, refinement, load-balancing, memory management, defining the geometry and topology of the mesh, graphics and the menu-driven user interface.

Section 4 deals with DIMEFEM, which is concerned with Finite Element representations of scalar, vector and tensor-valued functions on the domain defined by DIME. These functions can be added and may be multiplied by a scalar. We may also create linear and multilinear operators, and solve variational equations involving these functions and operators.

In Section 5, we describe an incompressible Navier-Stokes solver which has been implemented with DIME and DIMEFEM. Sections 6 and 7 discuss portability of the code and future work.

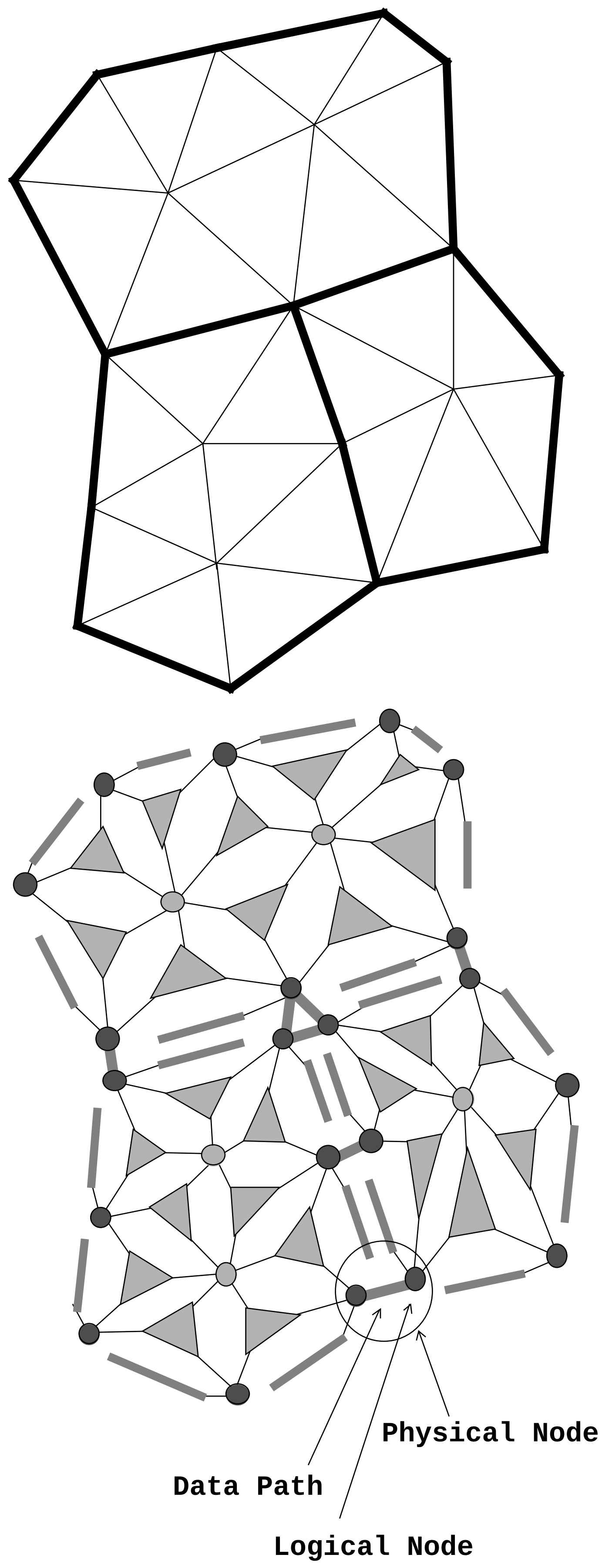

Figure 2: A triangular mesh with internal boundaries, shown as a physical mesh (top) and as a logical mesh (right). The internal boundary may be between different materials, or it may be a boundary between processor domains.

In addition to this Finite-Element layer and the fluid solver, DIME has also been used for supersonic flow calculations[2], free-lagrange fluid flow[3] and Monte-Carlo simulation of randomly triangulated surfaces for superstring theory. Another layer, DIME3D, describes two-dimensional oriented manifolds embedded in three-dimensional space, and has been used for simulation of the sensory system of an electric fish by the Boundary Element Method, and for modeling surfaces for computer graphics.

2. Irregular Meshes and Parallelism

Mesh generation can be the most time-consuming part of a finite-element calculation, since often the experimenter requires different mesh resolution in different regions of the problem domain. In parallel the problem is more difficult, since the mesh is to be split up among many processors so that each processor has about the same amount of work to do, and so that communication overhead is minimized.

We have chosen to use irregular meshes and have them refine and load-balance themselves as the solution emerges, rather than try to make the load-balanced mesh at the start. Instead of directly choosing the places at which the mesh should be finer, the experimenter chooses the criterion governing refinement. An irregular mesh has more overhead than a regular mesh and cannot be vectorized, but hopefully its flexibility and convenience will make up for this, especially with highly inhomogeneous problems.

3. DIME

The Distributed Irregular Mesh Environment[1] is a programming environment for creating and manipulating irregular triangular meshes. A user of DIME writes a set of subprograms and associates these with menu items. When the code is compiled with DIME the user's menu items and many other items provided by DIME appear in a graphics window with a tree of menus, or the code can be run in batch mode. Domains and meshes can be created, written to or read from disk, and a PostScript hard-copy facility is available.

The mesh defines an orientable sheet which may be planar or embedded in three-dimensional space. The mesh may be considered as a collection of nodes and elements (triangles), with each node connected to several elements and each element connected to exactly three nodes, as shown in Figure 1. Each node and element structure contains data, and DIME knows only the amount of such data, so that when a new node or element is created the correct amount of memory is allocated and the appropriate user-defined function is called to put something sensible in this new space. Thus there is a distinction between the data owned by DIME, which is mainly pointers to other data structures; and user-data, such as fluid pressure or velocity.

In a similar fashion data may also be associated with the boundaries of the mesh, either as boundary-node data or boundary-edge data.

In parallel the mesh is stored by domain-decomposing it, so that each processor ‘‘owns’’ some of the elements. It would be possible to make this parallelism completely invisible to the user, by sending a request asynchronously to another processor whenever off-processor data is needed, but this approach may be very inefficient. Instead DIME deals with these processor-processor borders as shown in Figure 2. At a border between several processor domains, each of those processors keeps a copy of the single ‘physical’ node at the junction with all its data. Thus each processor may access the data in the shared node by looking at its own copy. The user must remember that whenever the node data is changed, it should be changed in the same way in the copies in other processors. Several function calls are provided to do this in various ways (see below) which ensure that the logical nodes which constitute a single physical node always have the same data.

These internal boundaries in the mesh may be used for other purposes than separating processor domains. In an elasticity code, for example, there may be different materials separated by an internal boundaries. The boundary condition to be imposed is that the displacement be the same for each material, which is exactly the same as the condition above that each logical node constituting a physical node have the same data. Periodic boundary conditions may be imposed by imagining the mesh to cover the surface of a cylinder, with an internal boundary cutting the cylinder so that it can be unrolled. Similarly spherical and toroidal meshes may be made.

3.1 Communication

DIME uses two communication protocols, `combine` and `node_combine`, for global and local communication respectively. The function `combine` is provided by the underlying operating system Express[4,5], and is used for example to decide if a convergence criterion has been reached, which might be the maximum of a quantity over all the processors:

```
    maxcrit = local_maximum(....);
    combine(&maxcrit, max, DOUBLE);
```

First the value of the quantity `maxcrit` is evaluated by all the processors in parallel, then after the call to `combine`, all the processors have the global maximum in the variable `maxcrit`. The function `max` could be any associative and commutative binary function such as `sum` for getting the global sum of all the individual processor values. `node_combine` is implemented with a general message-passing and routing system.

All the communication in DIME is loosely synchronous[4,5], meaning that when a function involving communication (such as `combine`) is called, all the processors must call it in the same logical place, but not necessarily at the same time. The processors block on reading and writing, so that if a processor is late, the others wait for it. Conversely, if only some processors communicate deadlock results. This is Loosely Synchronous communication, which is not like the lockstep of soldiers marching, nor is it asynchronous like people communicating by telephone. Instead it is like a group of people working on their own projects but with scheduled meetings every so often which everybody must attend. In this fashion

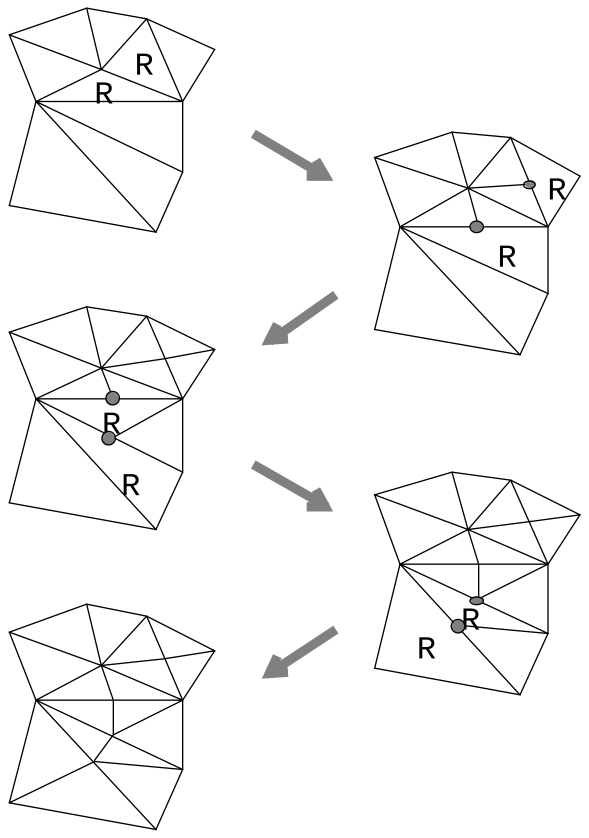

Figure 3: Refinement; some elements are selected to be refined (marked R at top left), which are bisected on their longest sides, creating non-conforming nodes, marked with a blob. Each of the triangles opposite the non-conforming nodes is marked for refinement, which are then bisected along their longest sides, and so on to completion.

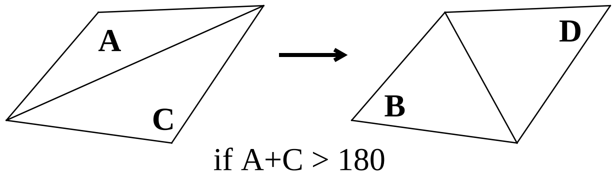

Figure 4: Topological relaxation; on each non-boundary edge, there is a triangle on each side with the edge being a diagonal: if the sum of the opposite angles A+C is greater than 180 degrees, the diagonal is moved to its other position. Now B+D is less than 180 degrees.

the code runs deterministically without need for locks, and a natural synchronization is imposed on the calculation.

The physics which we try to simulate with a finite element code is local. Differential equations relate field values at a point to field values in the immediate vicinity of the point. For this reason DIME allows a node to reference only its neighboring elements, and an element to reference only its neighboring nodes. When node data are changed, the other logical nodes (which may be in other processors) should also be changed.

The striped lines in Figure 2 show data paths, by which the logical nodes making up a physical node communicate. Communication along these paths is achieved by the construction `node_combine`.

`node_combine` is for combining not globally over all the processors, but for each physical node combining over all the logical nodes constituting it. Suppose we wish each node to compute the sum over its neighboring elements of a quantity `x` and put the result in the node-data-value `result`:

```
FORALLNODES(node)
    FORALLNEIGH(node, elmt)
        node->user->result +=
                           elmt->user->x;
    NEXTNEIGH(node, elmt)
NEXTNODE(node)
node_combine(result, sum, DOUBLE);
```

In this code, `node` is a pointer to a node structure and `node->user` is a pointer to the user-data associated with that node. `elmt` points to an element (triangle), and `elmt->user` points to the user data for the element. `FORALLNODES` loops over all the nodes, and `FORALLNEIGH` loops over all the neighboring elements of a node. In sequential operation the call to `node_combine` is not necessary, but in parallel it sums the values of the node-data-value `result` over all the copies of the node in different processors.

3.2 Manipulating the Mesh

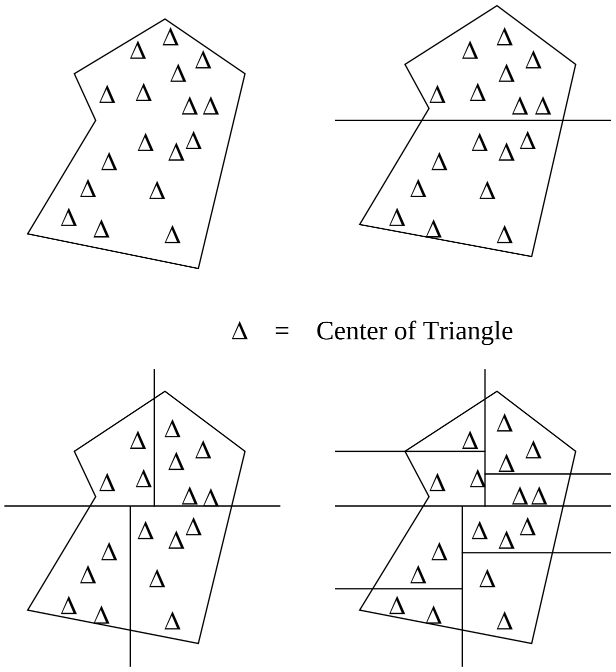

Figure 5: Load Balancing by orthogonal recursive bisection. Each D represents the center of a triangle, and the set is first divided into two sets of equal size by a horizontal median, then each set is divided by a vertical median, etc. The balancing is done in parallel and takes time log n, where n is the number of processors.

The mesh is adaptive in the sense that it can be refined either automatically or by hand, and load balancing is available for distributing the mesh among any number of local-memory processors. Topological changes may be made to improve the condition number of the mesh for Finite-Element calculations, or for other purposes.

Refinement[6] consists of picking some elements to be refined, and applying the algorithm shown in Figure 3. Topological relaxation is shown in Figure 4, and has the property that when the mesh is completely topologically relaxed, it is a Delaunay triangulation, or equivalently the stiffness matrix for linear elements is diagonally dominant[7].

If heavy refinement occurs in one place, the calculation is likely to become unbalanced, in the sense that one processor will have many more elements to deal with than the others. Since the calculation moves at the pace of its slowest processor, the processors with less elements to compte are idle much of the time and the efficiency drops. load balancing is the process of sending elements to other processors, complete with all their data, and reestablishing their internal and external communication links. We wish for each processor to have about the same number of elements, and for elements which communicate heavily to be in processors which are close together in the machine architecture. The algorithm used at present is orthogonal recursive bisection, illustrated in Figure 5, which is particularly suitable for a hypercube architecture.

Another method of manipulating the mesh provided by DIME is relaxation, in which the position of each node is moved to the average position of its neighboring nodes. This process improves the condition number of the mesh stiffness matrix at little cost.

4. Finite Element Calculations

DIMEFEM is a software layer which enables finite element calculations to be done with the irregular mesh maintained by DIME. The data objects dealt with by DIMEFEM are Finite Element Functions (FEFs) which may be scalar or have several components (vector fields), and also linear, multilinear and nonlinear operators which map these FEFs to numbers. The guiding principle is that interesting physical problems may be expressed in variational terms involving FEFs and operators on them[8,9]. We shall use as example a Poisson solver.

Poisson's equation is $\nabla^2 u = f$, which may also be expressed variationally as:

find u such that for all v

$$a(u, v) \equiv \int \nabla u \cdot \nabla v \, d\Omega = \int f\, v \, d\Omega \equiv L(v)$$

where the unknown u and the dummy variable v are taken to have the correct boundary conditions. To implement this with DIMEFEM, we first allocate space in each element for the FEFs u and f, then explicitly set f to the desired function. We now define the linear operator L and bilinear operator a as above, and call the linear solver to evaluate u.

4.1 Memory Allocation

When DIME creates a new element by refinement, it comes equipped with a pointer to a block of memory of user-specified size which DIMEFEM uses to store the data representing FEFs and corresponding linear operators. A template is kept of this element memory containing information about which is already allocated and which is free. When a FEF is to be created the memory allocator is called, which decides how much memory is needed per element, and returns an offset from the start of the element-data-space for storing the new FEF. Thus a function in DIMEFEM typically consists of allocating a stack of work space, doing calculations, then freeing the work space.

A FEF thus consists of a specification of an element type, a number of fields (1 for scalar, 2 or more for vector), and an offset into the element data for the nodal values.

4.2 Operations and Elements

Finite element approximations to functions form a finite-dimensional vector space, and as such may be multiplied by a scalar and added, and functions are provided to do these operations. If the function is expressed as Lagrangian elements[10] it may also be differentiated, which changes the order of representation, for example differentiating a quadratic element produces a linear element.

At present DIMEFEM provides two kinds of element, Lagrangian and Gaussian, although strictly speaking the latter is not a Finite Element because it possesses no interpolation functions. The Gaussian element is simply a collection of function values at points within each triangle and a set of weights, so that integrals may be done by summing the function values multiplied by the weights. As with one-dimensional Gaussian integration, integrals are exact up to some polynomial order. We cannot differentiate Gaussian FEF's, but can apply pointwise operators such as multiplication and function evaluation that cannot be done in the Lagrangian representation.

Consider the nonlinear operator L defined by

$$L(u) = \int \exp[\, du/dx \,] \, d\Omega$$

The most accurate way to evaluate this is to start with u in Lagrangian form, differentiate, convert to Gaussian representation, exponentiate, then multiply by the weights and sum. This can be done explicitly with DIMEFEM, but in the future we hope to create an environment which 'knows' about representations, linearity etc., and can parse an expression such as the above and evaluate it correctly.

4.3 Linear Solver

The computational kernel of any finite element software is the linear solver. We have implemented this with preconditioned conjugate gradient, so that the user supplies a linear operator L, an elliptic bilinear operator a, and a scalar product S (a strongly elliptic symmetric bilinear operator which satisfies the triangle inequality), and an initial guess for the solution. The conjugate gradient solver replaces the guess by the solution u of the standard variational equation

$$a(u, v) = L(v) \qquad \forall v$$

using the preconditioner S.

5. Navier-Stokes Solver

We have implemented an incompressible flow solver using DIME and DIMEFEM. The algorithm is described more completely in Reference 9. The evolution equation for an incompressible Newtonian fluid of viscosity ν is

$$du/dt + \nu \nabla^2 u + (u \cdot \nabla)\, u + \nabla p = f$$

$$\nabla \cdot u = 0$$

We use a three-stage operator-split scheme whereby for each timestep of length dt the equation is integrated

1. from t to t + ϑdt with incompressibility and no convection, then

2. from t + ϑdt to t + (1-ϑ) dt with convection and no incompressibility condition, then

3. to t + dt as in stage 1 with incompressibility and no convection.

The parameter ϑ is $1 - 1/\sqrt{2} \approx 0.29$.

Each of these three implicit steps involves the solution of either a Stokes problem:

$$\alpha u - \nu \nabla^2 u + \nabla p = f$$

$$\nabla \cdot u = 0$$

or the nonlinear problem:

$$\alpha u - \nu \nabla^2 u + (u \cdot \nabla) u = f$$

where α is a parameter inversely proportional to the timestep. We solve the Navier-Stokes equation, and consequently also these subsidiary problems, with given velocity at the boundary (Dirichlet boundary conditions).

5.1 Stokes Problem

The Stokes problem is solved by defining u_0 such that $\alpha u_0 - \nu \nabla^2 u_0 = f$, and $w = u - u_0$, so that

$$\alpha w - \nu \nabla^2 w + \nabla p = 0$$

$$\nabla \cdot w = -\nabla \cdot u_0$$

For any pressure field p we can solve for w, which is linearly related to p. Thus we can define the bilinear operator $A(p, q) = \int q \, \nabla \cdot w \, d\Omega$ for any dummy fields p and q. The Stokes problem reduces to the solution of the variational statement in standard form

$$A(p, q) = L(q) \equiv -\int q \, \nabla \cdot u_0 \, d\Omega$$

This equation is solved by conjugate gradient, using a quasi-optimal preconditioner which involves the solution of a further Poisson equation with Neumann boundary conditions.

5.2 Nonlinear Problem

Given a velocity field u, we can define

$$y(u) = \alpha u - \nu \nabla^2 u + (u \cdot \nabla) u - f$$

so that the solution of the nonlinear problem is equivalent to the nonlinear minimization of the functional $J(u) = \int [\, y(u) \,]^2 d\Omega$. Such minimization problems can be solved by nonlinear conjugate gradient: the kernel of the algorithm is doing a one-dimensional minimization along a particular search direction w, that is minimizing $J(u-\rho w)$ with respect to ρ. Since y is quadratic in u, J is quartic in ρ, so this minimization can be accomplished analytically.

In fact rather better definitions of y and J are from

$$\alpha y - \nu \nabla^2 y = \alpha u - \nu \nabla^2 u + (u \cdot \nabla) u - f$$

$$\text{with} \quad J(u) = \alpha \int y^2 \, d\Omega + \nu \int [\nabla y]^2 \, d\Omega$$

Intuitively, these definitions would seem to provide a good preconditioner because the linear part of the nonlinear problem is solved by a *linear* conjugate gradient algorithm, and we don't waste the expensive nonlinear conjugate gradient method on this linear part of the problem.

The minimization can be accomplished by noting that if u is changed to $u-\rho w$, then y is changed to $y-\rho y_1+\rho^2 y_2$, where y is as above, and y_1 and y_2 are defined by

$$\alpha y_1 - \nu \nabla^2 y_1 = (w \cdot \nabla) u + (u \cdot \nabla) w$$

$$\alpha y_2 - \nu \nabla^2 y_2 = (w \cdot \nabla) w$$

Notice that solving for y_1 involves definition of the trilinear operator L

$$L(u, v, w) = \int [(u \cdot \nabla) v] \cdot w \, d\Omega$$

and L partially evaluated to a linear operator.

5.3 Time Step Limit

During the nonlinear step, the fluid is modeled by an infinitely compressible fluid with viscosity. Such a fluid is liable to develop shocks if evolved in time for long enough. Thus the time step is constrained to be less than the velocity gradient in the direction of the velocity:

$$dt\ \ |(u \cdot \nabla)\, u| \,/\, |u| << 1$$

5.4 Results

With velocity approximated by quadratic, and pressure by linear Lagrangian elements, we found that both the Stokes and nonlinear solvers converged in 3 to 5 iterations. We ran the square cavity problem as a benchmark.

To reach a steady state solution, we adopted the following strategy; with a coarse mesh keep advancing simulation time until the velocity field no longer changes, then refine the mesh, iterate until the velocity stabilizes, refine, and so on. The refinement strategy was as follows. The velocity is approximated with quadratic elements with discontinuous derivatives, so we can calculate the maximum of this derivative discontinuity for each element, then refine those elements above the 70th percentile of this quantity.

Figure 6 shows the mesh after 4 cycles of this refinement and convergence, at Reynolds number 1000. We note heavy refinement at the top left and top right, where the boundary conditions force a discontinuity in velocity, and also along the right side where the near discontinuous vorticity field is being transported aound the primary vortex. Figure 7 shows the logical structure, split amongst 4 transputers, with notation similar to that of Figure 3. Figures 8 and 9 show streamlines and vorticity respectively. The results accord well with the benchmark of Ref 11.

6. Portability

DIME, DIMEFEM and the fluid solver are written in C, and run under the Express operating system[4]. The codes were developed on NCUBE hypercubes, small Transputer boards and on a SUN workstation. Exactly the same code compiles and runs on all of these. We expect to port with little effort to a 64-node Intel iPSC/2 machine, a 256-node Ametek 2010 machine and a 40-node Meiko transputer machine in the near future.

7. Future Work

Future work includes improved methods for self-adaptive refinement of the mesh, free surfaces, and algorithm improvement. The latter includes adding some compressiblity to the nonlinear part of the timestep to reduce the tendency of the fluid to shock[8] and thus increase the allowable time step. We would like to implement a compressible flow algorithm, since a self-adaptive mesh is perhaps more suited to problems with shocks than incompressible flow.

We would also like to develop a language for a more natural way of programming problems that can be expressed as variational principles, as mentioned in Section 4.2.

References

1. R. D. Williams, ‘‘*DIME: A programming environment for unstructured triangular meshes on a distributed memory parallel processor*’’, Proc. 3rd Hypercube Conference, Pasadena, CA, 1988, ed G. C. Fox.

2. R. D. Williams, ‘‘*Supersonic Flow with an Unstructured Mesh*’’, Caltech Concurrent Computation Report C3P-636.

3. R. D. Williams, ‘‘*Free-Lagrange hydrodynamics with a distributed-memory parallel processor*’’, Parallel Computing, **7** (1988) 439.

4. Express operating system, ParaSoft Corp., Pasadena, California.

5. 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 solutions of Large Linear Systems*’’ Academic Press, New York, 1971.

8. R. Glowinski, ‘‘*Numerical Methods for Nonlinear Variational Problems*’’, Springer-Verlag, New York, 1984.

9. M. O. Bristeau, R. Glowinski and J. Periaux, ‘‘*Numerical Methods for the Navier-Stokes Equations. Applications to the Simulation of Compressible and Incompressible Viscous Flows*’’, Comp. Phys. Rep., **6** (1987) 73.

10. C. Johnson, ‘‘*Numerical solutions of partial differential equations by the finite element method*’’, Cambridge UP, Cambridge, England, 1987.

11. R Schreiber and H. B. Keller, ‘‘*Driven Cavity Problems by Efficient Numerical Techniques*’’, J. Comp. Phys., **49**

Figure 6: Auto-refined mesh for square cavity problem. Refinement is based on velocity gradient in the direction of the velocity.

Figure 7: Logical structure of the mesh in Figure 6 split among 4 processors.

Figure 8: Steady state streamlines for square cavity problem with R = 1000.

Figure 9: Steady state vorticity for square cavity problem with R = 1000.

(1983) 310.