JOURNAL OF COMPUTATIONAL PHYSICS **94**, 352–381 (1991)

Finite Element Applications on a Shared-Memory Multiprocessor: Algorithms and Experimental Results

RAMESH NATARAJAN

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Received September 9, 1989; revised March 9, 1990

We describe strategies for parallelizing finite element applications on a shared-memory multiprocessor. The applications studied include the convection–diffusion equation, and the Stokes equations of low Reynolds number hydrodynamics. The overall approach that we use is standard, but with significant restructuring for efficient parallel computation. The primary focus is on parallel methods for solving the linear systems of algebraic equations generated by the finite element discretization using polynomial iterative solvers preconditioned by incomplete factorizations. The challenging issue is the parallelization of the preconditioner, especially for the unstructured matrices obtained from finite element discretizations, and we describe the algorithms developed for this purpose. Specific experimental results obtained with our programs on ACE, a prototype 8-way, bus-based, shared-memory multiprocessor are discussed in detail. © 1991 Academic Press, Inc.

1. INTRODUCTION

This paper describes parallelization strategies and experimental results obtained on a shared memory multiprocessor for the solution of problems in fluid flow and transport phenomena using finite element methods. Our intent in this research is to develop general-purpose methods for the complex, computationally-intensive problems in this application area. However, in order to emphasize parallel processing issues, we only discuss some relatively uncomplicated model problems in this paper, focusing primarily instead on the parallelization of iterative methods for solving the sparse linear systems of equations that arise from the finite element discretization of the underlying continuum problem. Strang and Fix [25] have remarked on the widespread use of direct solution methods based on LU decomposition in finite element programs. These methods have the following advantages: they are relatively robust, they can be used to solve for multiple right-hand sides at little additional cost, and they terminate with an answer in a fixed number of steps (event though the solution obtained may be inaccurate for poorly conditioned matrices). Although much progress has been made in developing direct solution methods for very large-scale problems (Duff [8]), the primary difficulty is the fill-in during elimination, which results in prohibitive computational and storage costs for many realistic applications.

0021-9991/91 $3.00

Iterative solution methods are useful in these large-scale applications because the matrix is never modified in them and very little additional storage is required beyond that for the original matrix itself. Recent research has led to effective iterative algorithms that are robust and rapidly convergent (Hageman and Young [13]; Kincaid, Oppe, and Joubert [16]). Notable among these are the so-called polynomial iterative algorithms which approximate the solution (or equivalently, the residual) from the Krylov subspace of the matrix, generating this subspace in such a way that the storage requirement and the work per iteration are kept small. These methods are generalizations of the well-known conjugate-gradient and Lanczos algorithms for symmetric, positive-definite matrices, and in practice are often coupled with effective preconditioning techniques that dramatically improve their convergence. Their attractiveness from the parallel processing point of view is due to the fact that the basic operations in them are highly regular and easily partitioned. The most computationally-intensive part is typically a matrix-vector multiplication, an operation that can be heavily optimized on parallel computers. The challenge in the parallel implementation of these methods is therefore primarily one of devising suitably optimized preconditioners, and in this context, we have insisted on the criterion that the uniprocessor performance of the parallel preconditioner be comparable to that of the best-known, equivalent sequential preconditioner (Saad and Schultz [22]; Ortega [19]). The determination of the best sequential preconditioner is itself a complex issue that is highly architecture and problem dependent. In this research, therefore, we only report our speedup measurements relative to the uniprocessor performance of the same program, with the previous criterion ensuring that our results are at least qualitatively consistent.

The outline of this paper is as follows. Section 2 contains the details of the model problems, their finite element discretization, and the iterative algorithms used in their solution. Section 3 discusses some general aspects of the parallel architecture and software environment, insofar as they influence the choice and efficiency of our finite element data structures. Sections 4 and 5 respectively describe the parallelization of the matrix assembly and of the iterative solution algorithms. Section 6 gives details of the experimental timings and speedups for our programs on the ACE parallel computer system. Section 7 concludes with a summary and indicates areas for future research.

2. Theoretical Details

A. *The Convection–Diffusion Equation*

I. The total concentration c of a passive scalar due to diffusion and convection within a fluid domain Ω with boundary $\mathscr{S}$ is given by

$$\nabla^2 c - Pe\,\mathbf{u} \cdot \nabla c = 0, \tag{1}$$

where $\mathbf{u}$ is the stationary velocity field in Ω, and Pe is the dimensionless Peclét number. We assume the following boundary conditions on non-intersecting, but possibly empty, subsets of $\mathcal{S}$,

$$c = c_0, \quad \text{on} \quad \Gamma_1, \tag{2a}$$

$$\nabla c \cdot \mathbf{n} = f, \quad \text{on} \quad \Gamma_2, \tag{2b}$$

$$\nabla c \cdot \mathbf{n} - hc = g, \quad \text{on} \quad \Gamma_3. \tag{2c}$$

Here $\mathbf{n}$ denotes the unit normal vector on $\mathcal{S} = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$. Conditions (2a) and (2b) arise respectively from specifying the concentration and the flux at the boundary surface. Condition (2c) arises when a phenomenological transfer model is used for the boundary flux (h is a dimensionless Biot number in heat-transfer applications).

II. Consider the function space $V = H_0^1(\Omega)$ of functions that along with their first derivatives are square-integrable in Ω and vanish on Γ_1. The weak form of (1) then corresponds to finding the function $c \in H^1(\Omega)$ such that

$$\int_\Omega [\nabla c \cdot \nabla v + Pe(\mathbf{u} \cdot \nabla c)\, v]\, dV = \int_S (\nabla c \cdot \mathbf{n})\, v\, dS, \qquad \forall v \in V. \tag{3}$$

III. We report on results obtained from a finite element discretization of (3) in two-dimensional geometries using two types of elements: (i) a quadrilateral element with 4-point bilinear basis functions; (ii) a quadrangular element with 9-point biquadratic basis functions. Both these elements are isoparametrically mapped to a square reference element where element integrals are evaluated using 4-point and 9-point Gaussian quadrature, respectively.

We expand for c in the terms of the finite element basis functions Φ_i in the form

$$c = \sum_{i=1}^{N} c_i \Phi_i, \tag{4}$$

and use Galerkin's method in (3) to obtain

$$\sum_{j=1}^{N} \left[\int_\Omega [\nabla \Phi_i \cdot \nabla \Phi_j + Pe(\mathbf{u} \cdot \nabla \Phi_j)\, \Phi_i]\, dV \right.$$
$$\left. - \int_S (\nabla \Phi_j \cdot \mathbf{n})\, \Phi_i\, dV \right] c_j = 0, \qquad i = 1, ..., N. \tag{5}$$

The integrals in (5) are evaluated element-by-element and the results assembled into a global matrix and right-hand side to yield a system of linear algebraic equations of the form

$$Ax = b, \tag{6}$$

where $x = [c_1, c_2, ..., c_N]^T$ is the vector of unknown nodal values. Essential boundary conditions are then imposed, following standard finite element practice, by setting the off-diagonal and diagonal entries in the corresponding row of A to zero and unity, respectively, and by appropriately modifying the entries in the right-hand side vector so that the known boundary values are trivially obtained in the solution process.

Remark 1. With the standard Galerkin discretization above, a very fine mesh is required for large values of *Pe*, to avoid unstable oscillations in the solution. The required stability criterion, and alternative Petrov–Galerkin discretization schemes that lead to a more stable formulation by using different basis functions for the expansion and test function sets, are discussed in Thomasset [26].

IV. The stiffness matrix A in (6) is non-symmetric and definite, but when *Pe* is moderate, there are variants of the standard conjugate gradient algorithm that seem to work well for it. One of these, which is very straightforward to implement and works especially well when preconditioned by a suitable matrix T, is the conjugate gradient squared (CGS) algorithm of Sonneveld [24], shown below.

```
conjugate gradient squared (CGS):
  $x_0 =$ initial solution guess
  $s_0 = \mathbf{T}^{-1}(b - \mathbf{A}x_0)$
  $p_0 = s_0$
  $q_0 = s_0$
  $\rho_0 = s_0^{\mathrm{T}} \cdot s_0$
**for** $k = 0, 1, \ldots,$ **until convergence do**
**begin**
  $\sigma_k = s_0^T \cdot \mathbf{T}^{-1}\mathbf{A}q_k$
  $\alpha_k = \rho_k / \sigma_k$
  $f_k = p_k - \alpha_k \mathbf{T}^{-1}\mathbf{A}q_k$
  $g_k = p_k + f_k$
  $x_{k+1} = x_k + \alpha_k g_k$
  $s_{k+1} = s_k - \alpha_k \mathbf{T}^{-1}\mathbf{A}g_k$
  $\rho_{k+1} = s_0^{\mathrm{T}} \cdot s_{k+1}$
  $\beta_k = \rho_{k+1} / \rho_k$
  $p_{k+1} = s_{k+1} + \beta_k f_k$
  $q_{k+1} = p_{k+1} + \beta_k(f_k + \beta_k q_k)$
**enddo**
```

The unpreconditioned version of the algorithm requires six vectors of length N, in addition to the storage for the matrix, solution vector, and right-hand side. The arithmetic work per iteration consists of two matrix-vector multiplications, seven saxpy's and two dot-product operations. It has the advantage over some other methods of not requiring a matrix–vector multiply with the transposed matrix, which can be useful, for example, when the matrix is stored in the sparse format of Section 4.

B. *The Stokes Equations*

I. The creeping flow of an incompressible fluid in a domain Ω with bounding surface $\mathcal{S}$ is described by the following dimensionless equations

$$-\nabla p + \nabla \cdot \tau = \mathbf{f}, \tag{7}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{8}$$

where p, τ, and $\mathbf{u}$ denote the pressure, stress tensor, and velocity, respectively, and $\mathbf{f}$ is a volumetric body force. Equation (7) is also relevant when a Picard iterative scheme is used on the full steady Navier–Stokes equations by approximating it as a sequence of simpler Stokes problems. In that case $\mathbf{f}$ would also contain approximations to the nonlinear convective terms from the previous iteration. The stress tensor is given for a Newtonian fluid by

$$\tau = \nabla \mathbf{u} + (\nabla \mathbf{u})^{\mathrm{T}}. \tag{9}$$

This set of equations must satisfy the boundary conditions on $\mathcal{S} = \Gamma_1 \cup \Gamma_2$,

$$\begin{aligned} \mathbf{u} &= \mathbf{u}_0, && \text{on} \quad \Gamma_1, \\ -p\mathbf{n} + \tau \cdot \mathbf{n} &= \mathbf{g}, && \text{on} \quad \Gamma_2, \end{aligned}$$

where $\mathbf{n}$ denotes the unit normal vector on the boundary surface, and $\mathbf{u}_0$ and $\mathbf{g}$ are the velocity and traction boundary conditions. We make a few remarks on the boundary conditions.

Remark 2. If $\Gamma_1 = \varnothing$, then solutions $\mathbf{u}$ to (7)–(9) are unique only up to an arbitrary rigid motion. Then $\mathbf{f}$ must be consistent with the boundary condition on Γ_2.

Remark 3. If $\Gamma_2 \neq \varnothing$ and $\Gamma_1 \cap \Gamma_2 \neq \varnothing$, then we have a free-boundary on $\Gamma_1 \cap \Gamma_2$. This requires an additional boundary condition on Γ_2, essentially stating that this bounding surface is also a material surface of the fluid. We have not considered this case in the computations presented in this paper.

Remark 4. If $\Gamma_1 = \mathcal{S}$, then the boundary velocity $\mathbf{u}_0$ must be consistent with the global conservation of mass, i.e., $\int_{\mathcal{S}} \mathbf{u}_0 \cdot \mathbf{n}\, dS = 0$.

II. The weak version of this problem can be formulated as follows. Consider the function space $V = (H_0^1(\Omega))^n$ consisting of vector functions vanishing on Γ_1, whose components along with their first derivatives are square-integrable in Ω. Also consider the function space $Q = L^2(\Omega)$ consisting of square-integrable functions in Ω (*Note.* If the pressure is determined only up to an arbitrary constant, then the space Q is defined modulo the space of constant functions on Ω). The weak form of (7)–(9) then corresponds to finding a pair $(\mathbf{u}, p)$ in $(H^1(\Omega))^n \times Q$ such that

$$\int_\Omega [\tau : \nabla \mathbf{v} - p\nabla \cdot \mathbf{v}]\, dV = -\int_\Omega \mathbf{f} \cdot \mathbf{v}\, dV - \int_S [p\mathbf{n} - \tau \cdot \mathbf{n}] \cdot \mathbf{v}\, dS, \qquad \forall \mathbf{v} \in V, \tag{10}$$

$$\int_\Omega (\nabla \cdot \mathbf{u})\, q\, dV = 0, \qquad \forall q \in Q. \tag{11}$$

Remark 5. The normal stress condition appears as the natural boundary condition in (10), but this is not so useful when only velocity boundary conditions are imposed. An alternative formulation obtained by substituting the divergence condition (8) into (7), yields the weak form

$$\int_\Omega [\nabla \mathbf{u} : \nabla \mathbf{v} - p\nabla \cdot \mathbf{v}]\, dV$$
$$= -\int_\Omega \mathbf{f} \cdot \mathbf{v}\, dV - \int_S [p\mathbf{n} - \nabla \mathbf{u} \cdot \mathbf{n}] \cdot \mathbf{v}\, dS, \qquad \forall \mathbf{v} \in V, \tag{12}$$

which is useful in Cartesian coordinates where the integrand involving $\mathbf{u}$ and $\mathbf{v}$ simplifies to yield copies of the well-known weak form of the Laplacian operator for each scalar velocity component.

III. The discretization is carried out using the nine-point Crouzeix–Raviart quadrilateral element on which velocity and pressure are approximated by continuous biquadratic and piecewise-continuous linear basis functions, respectively. It has been shown that this basis yields discrete approximations to the spaces V and Q, that satisfy the Babuska–Brezzi condition required for the solvability of (10) and (11) (Fortin [10]; Girault and Raviart [12]).

For computational purposes, this quadrilateral is mapped to a square reference element on which the element integrals are evaluated using a 9-point Gaussian quadrature rule. There are nine nodal variables for each velocity component in this element—its values at the four vertices, at the four mid-edge nodes, and at the centroid. There are three nodal variables for the pressure in each element, which are its value and those of its derivatives at the centroid node (note that the pressure derivative is constant on each element). This leads to a total of 21 unknowns per element in two dimensions, which can be reduced to 17 unknowns per element by eliminating the unknown velocities and the pressure derivatives at the centroid node prior to element assembly. This procedure can be carried out stably, by eliminating the pressure gradient unknowns from the components of the momentum residual equation, and the velocity unknowns from the components of the continuity residual equation. The element-level substructuring is most useful since the number of unknowns and the bandwidth of the stiffness matrix is reduced without in any way affecting the solution accuracy. Thus, letting M and N denote the number of centroid and non-centroid nodes, respectively, in the finite element mesh, we expand for $\mathbf{u}$ and p in the form

$$\mathbf{u} = \sum_{i=1}^{N} \mathbf{u}_i \Phi_i, \qquad p = \sum_{i=1}^{M} p_i \Psi_i,$$

where Φ_i, Ψ_i are the basis functions that are obtained from the original Crouzeix–Raviart basis set after element-level substructuring.

The use of Galerkin's method then yields

$$\int_\Omega [\tau : \nabla \Phi_i \mathbf{e}_k - p \nabla \cdot (\Phi_i \mathbf{e}_k)]\, dV$$

$$= -\int_\Omega \mathbf{f} \cdot \mathbf{v}\, dV + \int_S [\tau \cdot \mathbf{n} \cdot \Phi_i \mathbf{e}_k - p \Phi_i \mathbf{e}_k \cdot \mathbf{n}]\, dS, \qquad i = 1, ..., N, \tag{13}$$

$$\int_\Omega (\nabla \cdot \mathbf{u})\, \Psi_i\, dV = 0, \qquad i = 1, ..., M. \tag{14}$$

The alternative form of (13), obtained from the weak formulation in (12), is given by

$$\int_\Omega [\nabla \mathbf{u} : \nabla \Phi_i \mathbf{e}_k - p \nabla \cdot (\Phi_i \mathbf{e}_k)]\, dV$$

$$= -\int_\Omega \mathbf{f} \cdot \mathbf{v}\, dV + \int_S [\nabla \mathbf{u} \cdot \mathbf{n} \cdot \Phi_i \mathbf{e}_k - p \Phi_i \mathbf{e}_k \cdot \mathbf{n}]\, dS, \qquad i = 1, ..., N. \tag{15}$$

IV. The weak form in (13)–(14) leads to a saddle-point variational problem and the standard iterative solution technique devised for minimization problems are not readily usable for it. We briefly comment on two alternative schemes that to lead to a minimization problem, but which have other difficulties associated with them especially in the context of iterative methods. The first scheme chooses an expansion basis set for $\mathbf{u}$ from the solenoidal subspace $V_0 \subset V$. The derivation of elements which satisfy this property is difficult and requires the introduction of new degrees of freedom for which appropriate boundary conditions must be obtained (see Cuvelier, Segal, and Steenhoven [7]). The difficulty here is in finding suitable good initial guesses that satisfy this divergence-free condition for use with linear iterative solvers.

A second scheme is to introduce a penalty parameter ε^{-1}, $\varepsilon \ll 1$ into the problem formulation; i.e., instead of (11) we write

$$\int_\Omega (\nabla \cdot \mathbf{u} + \varepsilon p)\, q\, dV = 0, \qquad \forall q \in Q. \tag{16}$$

Note that this is equivalent to artificially introducing a small amount of compressibility into the problem. Using Galerkin's method, with the basis function expansions for **u** and p in (12) and (13), and carrying out the finite element assembly, we obtain a set of equations of the form

$$AU + B^{\mathrm{T}}P = b_1, \tag{17a}$$

$$BU + \varepsilon DP = b_2, \tag{17b}$$

in which A is a $N \times N$ symmetric, positive-definite matrix, B is a $M \times N$ matrix, and D is a $M \times M$ diagonal matrix. Eliminating P by block Gaussian elimination, we obtain

$$\hat{A}U = (A + \varepsilon^{-1}B^{\mathrm{T}}D^{-1}B)\, U = b_1 - B^{\mathrm{T}}D^{-1}b_2. \tag{18}$$

Note that $\hat{A}$ is also symmetric, positive-definite, and when piecewise-constant basis functions are used for the pressure, this matrix can be directly assembled element-by-element. For small ε, $\hat{A}$ is very ill-conditioned, but this is not a serious problem when direct solution methods are used, since the magnitude of ε can be adjusted to obtain a reasonable compromise between the round-off error growth and the approximation of the incompressibility condition. For iterative methods, however, this will result in the number of iterations required for convergence becoming very large and our experience has been that even with preconditioning it is difficult to obtain an effective algorithm with this approach.

V. Performing the block Gaussian-elimination on (17) in an alternative way (and setting $\varepsilon = 0$) leads to a formulation that avoids the difficulties associated with matrix ill-conditioning

$$AU = b_1 - B^{\mathrm{T}}P, \tag{19a}$$

$$BA^{-1}B^{\mathrm{T}}P = BA^{-1}b_1 - b_2. \tag{19b}$$

This yields two uncoupled positive-definite systems that can be solved successively for P and U in that order. An inner–outer iterative approach is used in solving (19b). The outer loop requiring the inversion of $(BA^{-1}B^{\mathrm{T}})$ is carried out by a standard conjugate-gradient algorithm. Each iteration in this algorithm requires the action of $(BA^{-1}B^{\mathrm{T}})$ on a vector and the inversion of A required in this step is carried out using preconditioned conjugate-gradient in an inner loop. The algorithm for the outer iteration is as follows

```
inner-outer conjugate gradient algorithm (IOCG)
  initialize outer iteration
  $x_1 = \mathbf{A}^{-1}b_1$
  $r_2 = \mathbf{B}x_1 - b_2$
  $x_{20}$ = initial solution guess
  $r_{10} = \mathbf{B}^{\mathrm{T}}x_{20}$
```

```
    $x_1 = \mathbf{A}^{-1} r_{10}$
    $s_0 = r_2 - \mathbf{B} x_1$
    $p_0 = s_0$
    $\gamma_0 = s_0^{\mathrm{T}} \cdot s_0$
for $k = 0, 1, \ldots,$ until convergence do
    begin
    $r_{1k} = \mathbf{B}^{\mathrm{T}} p_k$
    $x_1 = \mathbf{A}^{-1} r_{1k}$
    $q_k = \mathbf{B} x_1$
    $\tau_k = p_k^{\mathrm{T}} \cdot q_k$
    $\alpha_k = \gamma_k / \tau_k$
    $x_{2k+1} = x_{2k} + \alpha_k p_k$
    $s_{k+1} = s_k - \alpha_k q_k$
    $\gamma_{k+1} = s_{k+1}^{\mathrm{T}} \cdot s_{k+1}$
    $\beta_{k+1} = \gamma_{k+1} / \gamma_k$
    $p_{k+1} = s_{k+1} + \beta_{k+1} p_k$
enddo
    finish up by solving for $x_1$
    $r_1 = b_1 - \mathbf{B}^{\mathrm{T}} x_{2k}$
    $x_1 = \mathbf{A}^{-1} r_1$
```

This implementation of IOCG requires four vectors of length $M + N$ (assuming that the inner conjugate gradient is unpreconditioned), besides the storage that is used for A, B, B^{T}, and the solution and right-hand side vectors. We store B^{T} explicitly in order to obtain an efficient matrix–vector multiply with it. Each iteration of the unpreconditioned inner conjugate gradient requires one matrix–vector multiply, and two dot-products, and three saxpy's on vectors of length N. The requirements of the outer conjugate gradient iteration are roughly similar, two matrix-vector multiplys with B and B^{T} are required, and two dot-products and four saxpy's on vectors of length M.

One nice feature of the formulation given here is that the inner conjugate gradient benefits from the progress of the outer loop, so that especially in the later stages, when good initial guesses are available, very few inner iterations are required to reduce the residual to a prescribed tolerance. Thus, even though sparse Cholesky factorization methods are also attractive for the inversion of A, the present scheme requires less storage, better exploits the information in good initial guesses, and is more amenable to a parallel implementation. However, care must be taken to ensure that the inner iteration is converged to sufficient accuracy so that the matrix that is inverted in each outer iteration is unchanged, since this property is essential for the convergence of the outer conjugate gradient iteration. Although in practice we have not found this to be a problem, we note that Bramble and Pasciak [6] have proposed an alternate algorithm for (19) that avoids this difficulty by not using the inner–outer formulation at all; however, their approach does require the estimation of an additional "scaling" parameter.

3. Parallel Architecture and Software Considerations

Our work is directed towards the asynchronous, shared-memory class of parallel computers, specifically the ACE workstation multiprocessor developed at the IBM T. J. Watson Research Center, which has recently been the focus of our experimental efforts. The ACE architecture is comprised of eight 32-bit processors rated at one MIPS, with each processor having about 8 Mb of local memory. These processors are connected by a 80 Mb/s bus to each other and to a global memory that can be extended to 64 MB. The various design aspects of the memory configuration, such as the interleaving used, and the nominal relative latencies of local and global memory, can be found along with the other architectural details, in the paper by Garcia, Foster, and Freitas [11]. Since ACE is not a production scientific computer but an evolving experimental prototype, we have not made absolute performance a crucial issue in this research. In addition, we have avoided using any machine-dependent parameterizations that might affect the generality and efficiency of our programs on other shared memory parallel computers or on future versions of the ACE architecture itself.

Our parallel programs are written using the Preface/Mach/EPEX environment (Bernstein and So [4]; Bolmarcich [5]). This environment provides a set of parallel constructs that can be embedded by the programmer in otherwise standard, sequential FORTRAN code to allow the specification of shared and private data, and to indicate the parallel flow of control through the code. These constructs are identified by a preprocessor and replaced in-line by appropriate subroutine calls to a run-time synchronization library. The parallel control constructs used include:

1. *serial sections*, that identify code executed by only one processor
2. *parallel loops*, for the self-scheduled execution of independent iterations of a FORTRAN DO-loop
3. *critical sections*, that identify code executed by only one processor at a time
4. *barriers*, at which all processes must synchronize before proceeding with further execution.

Since these are all standard parallel programming constructs, we expect their functionality to be reproducible in other environments.

We illustrate the style, syntax, and efficiency of our parallel programs with a simple example. Consider some of the operations on shared vector operands that might be repeatedly used in iterative algorithms, such as dot-products, saxpy's, scalar normalization, vector-to-vector copy, and so on. These are all parallelized by a run-time "strip-mining" of the operands among the processors, an approach that should yield close to perfect speedups with sufficiently long operands. An example of a routine written in this style is shown in Appendix 1, where the stride one dot-product of two shared vectors is computed, and the result is returned as a private value to each process. All the parallel constructs mentioned earlier are used in this example (*Note.* the @ symbol identifies keywords for the preprocessor).

The various factors that affect the parallel efficiency of this routine are described below, assuming throughout that the vector lengths are large relative to the number of processors. First, we have the extra work due to simply inserting the code for the parallel constructs, which can be measured by comparing the uniprocessor execution time of the parallel program with that of the original sequential code. Since each parallel construct adds only a few synchronization instructions to the original sequential program, it is expected and confirmed by experiments that this overhead is negligible.

Second, we have the overhead due to uneven partitioning of the work during multiprocessor execution, which manifests itself as the time spent by processes waiting at synchronization points, such as barriers, serial sections, and critical sections. This overhead is small in the dot-product example for a variety of reasons; few instructions are executed in explicitly-protected mode, and the number of accesses to implicitly-protected regions of code, such as the parallel DO-loop scheduler are reduced by chunking iterations. Finally, in the interval between the two global synchronizations in the routine where the largest amount of work is performed, with each process computing its portion of the parallel dot-product, there is a very even division of work (especially when the vector length is a multiple of the number of processors).

Third, random or non-reproducible environmental effects in an asynchronous MIMD environment can lead to uneven execution times between global synchronizations. These effects generally have their origins in hardware resource contention, such as bus and memory bank conflicts, and process multiplexing. Their impact in a shared memory environment can be reduced by using a smaller task granularity. For example, the use of a smaller chunksize in the parallel DO-loop enables faster processors to take up more of the work load and compensate for variations in the individual processing speeds. These overheads are similar to those described in the previous paragraph in terms of their effect being manifested as a work-load imbalance; however, they differ in that they depend more intimately on the architectural organization and balance, and on the run-time environment, rather than on the programmer-induced partitioning of work. This makes it very difficult to obtain an adequate general characterization for this class of overheads. However, our measurements are always carried out in single-user mode with fewer processes than processors, and a "master" processor is always left free to perform the various time-shared operating system chores unrelated to the application. In addition, various local memory optimizations essential to obtaining good performance also have the desirable side-effect of reducing memory and bus contention. For these reasons, the effect of this class of overheads on our program performance is small, and our results are always well reproducible.

For asymptotically long vector operands, our measurements on ACE show parallel efficiencies of over 0.95 for the dot-product programmed as shown in Appendix 1. This is quite satisfactory, but it raises the question as to whether it is possible to achieve an even better efficiency than this by sectioning array operands and distributing them among the processors as private data. The relative merits of these two approaches on the ACE architecture leads to some subtle and interesting issues that we have discussed in great detail in a companion paper (Natarajan [18]), from which we summarize briefly. The ACE architecture has been consciously designed to avoid the high development cost of hardware synchronization for maintaining the "cache-consistency" of shared data in the individual local memories. Instead, certain operating system assists, akin to demand paging, are used in order to reduce the latency by having shared data paged into the local memories of individual processors. The effectiveness of these assists, however, can be rapidly degraded if the shared data is not very carefully managed. In particular, in a straightforward dynamic task-scheduling environment, without further work, a low-latency access can only be guaranteed only for the private data. It is our observation that in a relatively complicated program, after an initial transient, the ACE memory model moves to a state in which the local memories contain only code and private data, while the global memory contains only shared data. The relative efficiency of global/shared versus distributed/private data structures in this memory model is again not obvious. Briefly, global data structures provide programming simplicity and generality, and lead to a better processor utilization when the workload is uneven, while distributed data structures lead to much lower latencies in accessing data.

In our current program implementations we make the most appropriate choice for storing the data structures consistent with these programming trade-offs. For example, the stiffness matrix and the various arrays in the linear solvers are declared as shared variables for programming convenience, while element matrices, whose scope is restricted to a single parallel task are declared as private variables. This means that memory accesses to the primary data structures might possibly take place at the global memory latency, although in practice, this will be partially compensated by the greater generality of usage and the improved load balancing that is obtained. In addition, in the present instance, there are some difficulties associate with the use of distributed data structures for some of the larger problems, since on the ACE machine the size of local memory can become a limitation (particularly without the full implementation of the proposed use of global memory as an automatic paged store for private-data pages).

4. Matrix Storage Formats and Finite Element Assembly

The primary targets for speedup in a finite element program are: (i) the computation of element matrices and their assembly into the stiffness matrix; and (ii) the solution of linear systems of algebraic equations. On parallel computers, the efficiency of both these steps depends crucially on the storage format used for the stiffness matrix. We have used the standard storage format for non-symmetric sparse matrices in wich the nonzero entries are stored row-by-row in a single array. An associated array holds the corresponding column index of each entry in the previous array, and a third array holds pointers to the location of the first element of each row of the matrix in the first and second arrays. Symmetric matrices are also stored in the same format, although it would be sufficient to just store the upper or lower triangle. The computational efficiency obtained with the redundant representation, particularly in the matrix–vector multiply operation, more than compensates for the extra storage used.

The parallelization of the finite element assembly process is straightforward. Each element matrix is computed in parallel and the assembly into the stiffness matrix, which is a shared array, is performed in a critical section, thereby ensuring that multiple processes do not simultaneously modify the same shared memory location.

```
for each element pardo
   call routines to generate element matrix and right-hand sides
   begin critical section
   assemble into stiffness matrix and right-hand side
   end critical section
endpardo
```

The bottleneck here occurs when processes have to wait while another process is executing within the critical section. We show here qualitatively that this waiting time can be small, particularly in those applications in which the cost of generating the element matrices is greater than the cost of the assembly into the global matrix. Specifically, consider the case where in each independent iteration in the parallel loop, the ratio of the number of instructions executed outside the critical section to that executed inside it, exceeds the total number of active processes. Then, during execution, each process in its first pass through the loop will be released in a staggered manner from the critical section. After this preliminary transient, therefore, processes are synchronized so that they encounter no contention on subsequent arrivals at the critical section, and the overall execution then proceeds practically as though the loop were fully parallel.

From this argument it is clear that the primary optimization in the assembly algorithm is to reduce the transit time for each process through the critical section. For example, in the sparse storage format for the stiffness matrix described earlier, the row pointer index directly gives the range of storage locations into which each entry from the element matrix will be assembled. The exact location in this range is then determined by searching the column index list for a match. Since, for each row, the column–index array is an ordered list, a binary search algorithm can be used and may be faster than the usual linear search. Let α denote the order of the element matrix, and β the number of nonzeroes in a given row of the global matrix. The asymptotic cost of the linear and binary search algorithms for the assembly of each row of the element matrix are given by $O(\alpha\beta)$ and $O(\alpha \log_2 \beta)$, respectively. However, if these two algorithms are calibrated for ordered lists of various lengths, then a simple check of the number of non-zeroes of the appropriate row of the global matrix, can provide a runtime determination of the appropriate algorithm to be used. Yet another alternative is to use an $O(\alpha + \beta)$ linear-time algorithm in which no searching is performed, but which requires an extra temporary vector of length equal to the order of the global matrix; as described by Pissanetzky [20], this temporary vector is used to set up pointers for directly performing the assembly of each row of the element matrix.

We note that the use of a single critical section in this program is unduly conservative, and in particular, processes that update mutually-exclusive regions of the stiffness matrix are unnecessarily blocked from simultaneous execution. One approach around this is to partition the stiffness matrix, say in a block–row fashion, and use separate locks for the updates into each block, which will clearly reduce the synchronization–waiting at each individual lock. However, this modification does not change the overall number of instructions that are executed in a protected mode, but the granularity of work in each individual access to a critical section is reduced by having it moved to an inner loop of the matrix assembly. For example, in the extreme case when each row of the global matrix has its own lock, the storage required for these locks and the cost of merely executing the locking instruction itself, can both become significant overheads.

A different approach is suggested by the underlying problem formulation, which indicates that the simultaneous assembly of two element matrices needs to be protected by a critical section only if these two elements have common nodes in the finite element mesh. An element interference graph may be constructed whose vertices correspond to elements, with edges connecting vertices whose corresponding elements have common nodes in underlying mesh. By coloring this interference graph so that no two adjacent vertices have the same color, we obtain an algorithm that dispenses with the critical section, and instead loops sequentially over the color classes as follows:

```
for each color class seqdo
  for all vertices (elements) in this color class pardo
    call routines to generate element matrix and right-hand sides
    assemble into stiffness matrix and right-hand side
  endpardo
endseqdo
```

This algorithm will be efficient if the number of sequential steps is kept small, equivalently, by coloring the interference graph with the fewest possible colors. For general graphs, the minimum coloring problem is NP-complete, but fast heuristics are known that will generate a reasonably good minimum coloring. These approximate algorithms are usually sufficient, since the increased complexity of finding a more optimal minimum coloring must be balanced against the incremental speedup that is obtained in the assembly algorithm. We have not implemented this approach in our software at the present time, but we consider it to be promising especially in nonlinear and time-dependent problems where the overhead of generating the coloring can be amortized over repeated assemblies on the same mesh.

5. Implementation Details for Iterative Methods

As mentioned earlier, the only part of the iterative method that is not straightforward to parallelize is the preconditioning, and this section is therefore devoted to this aspect. In particular, there are two rather special issues that arise in selecting an appropriate preconditioner for the stiffness matrices generated by a finite element discretization. First, the data structures describing the mesh are element-oriented rather than node-oriented, and especially for irregular domains, these nodes may be numbered in a fairly arbitrary way. Second, the essential boundary conditions are enforced directly in the global matrix after assembly, by modifying the appropriate rows so as to trivially yield the correct boundary values upon solution. These two considerations make it difficult to identify a sparsity pattern in the global stiffness matrix, even though this structure would be evident in the invariably carefully-chosen ordering of the equivalent finite-difference formulation. Therefore, without further analysis, it is not directly possible to use some of the block-based preconditioners that have proved so attractive in the finite difference setting, especially for structured problems on regular meshes. Our feeling is that the selection of the preconditioner should not detract from the flexibility of the finite element formulation, and should be carried out without any a priori assumptions on the sparsity structure of the matrix, especially if this can be achieved without sacrificing efficiency in any way.

One class of well-known, general-purpose preconditioners can be derived from an incomplete LU factorization of the iteration matrix (assuming this factorization exists, see Meijerink and Van der Vorst [17]; Kershaw [15]). The advantage of these preconditioners is that they can be economically computed by limiting the amount of "fill-in" to a fixed sparsity pattern, which is often simply that of the original matrix itself. Furthermore, their use in the inner loop of the iterative method requires just a sparse triangular forward and backward solve, and this has

node orderings generated from finite-difference discretizations, that a careful data dependency analysis can be carried out to obtain a stage-wise partitioning of the relevant computations in the preconditioner. In each such stage, there is substantial medium-grain concurrency on the average, while successive stages have to be executed sequentially. Their study was directed towards vectorization, but the basic issues on a parallel computer are the same.

We show how this "wavefront" approach to parallel preconditioning can be extended to general sparse matrices through an automatic data-dependency analysis of the underlying computation graph. This analysis is carried out for the incomplete factorization as well as for the sparse triangular solves in a separate preprocessing step, and the results are saved so that the actual numerics can subsequently be repeatedly performed with very little overhead. The schedule for the sparse triangular solves, for example, is reused in each step of the iterative algorithm when the preconditioner is applied. The preprocessing for the incomplete factorization is useful in time-dependent or nonlinear problems, where this computation must be carried out repeatedly for a matrix with the same sparsity pattern but with different data in it, so that the same schedule can be reused.

The present effort is also motivated by the work of Baxter *et al.* [3], who present experimental speedup results for the parallelization of the sparse triangular solves on an Encore Multimax. After the results in this paper had been obtained, we also became aware of other related work. Anderson and Saad [1] have described level-scheduling algorithms similar to that used here for the sparse triangular solves, and have obtained experimental results on an Alliant FX/8 for various test matrices in the Boeing–Harwell test collection (see also Saad [21]). Hammond and Schreiber [14] have also considered similar experiments on an Encore Multimax, and in particular, have discussed two algorithms, which they term static and dynamic scheduling, respectively, showing that the former has less runtime synchronization overhead while the latter provides better processor utilization. The approach for the triangular solves that we have outlined in the previous paragraph, is intermediate to these two approaches, and it tries to simultaneously provide the performance benefits of both these alternatives.

Our present work also complements and extends the earlier work in some other directions. First, we show the relevance of automatically performing the parallelization analysis so that it is not necessary to sacrifice the flexibility in mesh generation and assembly that is intrinsic to the finite element method. Second, we show that the preprocessing algorithms can be also parallelized, although in practice this may not be so critical since the sequential algorithms are both quite efficient and have their cost amortized. Nevertheless, it does lead to some interesting parallel programming issues that are discussed below. Third, we also consider the parallelization of the incomplete factorization algorithm, an aspect that also has not been explicitly treated in the previous work; again perhaps the cost of this phase is also amortized. However, our measurements do show the importance of achieving good speedups here, for the overall efficiency of the application. The primary novelty of this aspect of our implementation is the simple and inexpensive scheme that is used to avoid race conditions on the shared data, in order to make it suitable for parallel environments without cache-coherence synchronization.

The algorithms given below are described in terms of a sparse matrix T of order n, with m non-zero entries. For the stiffness matrices generated from finite element discretizations, we usually have the case that $n < m \ll n^2$. We assume that T is stored in the standard row pointer–column index format, but note that in this format, the entries of a given row of T can be accessed in constant time, while access to the entries of a given column will require $O(m)$ operations. Since efficient access to the column entries of T is essential to our algorithms, we maintain, in addition, a separate representation of the sparsity pattern of T in a column pointer–row index format. This representation requires an additional $m+n$ integers beyond the $m+n$ integers and m reals that are used for storing the original matrix, and is very efficiently computed from the original representation using a linear $O(m)$ transposition sort algorithm (Gustavson [9]).

The basic primitive tasks in the parallel implementation are the following routines in which certain operations on the rows of T are performed, and which we assume to be efficiently coded to exploit sparsity:

1. *rowscl*(j), performs the scaling operation $T_{j,i} \leftarrow T_{j,i}/T_{j,j}$ for $j < i \leqslant n, j$ being given.
2. *rowupd*(j, k), performs the update operation $T_{k,i} \leftarrow T_{k,i} - T_{k,j} \times T_{j,i}$ for all $j < i \leqslant n$, j and k are given. We term j and k as the pivot and the target row indices, respectively.
3. *dot*(i, js, je, b), computes the dot-product of row i of T and a vector b; i.e., it returns $\sum_j T_{i,j} b_j$ for $js \leqslant j \leqslant je$, and if $js > je$ it returns 0.

These primitive routines are assumed to be indivisible, since in the usual case, we expect the granularity of work in them to be insufficient for amortizing parallelization overheads. This assumption may not true either if the target multiprocessor can exploit very fine-grain nested parallelism, or if the number of entries in a sparse row is relatively large, but in this case, these primitive routines can also be trivially parallelized without affecting the present discussion in any way.

The sequential row-oriented algorithms for the incomplete factorization, and the forward and backward sparse triangular solves using these routines are given below:

```
incomplete factorization:
for $j = 1, n - 1$ do
  call rowscl($j$)
  for $k = j + 1, n$ do
  call rowupd($j, k$)
  enddo
enddo
```

```
forward triangular solve:
for $i = 1, n$ do
  $t = dot(i, 1, i - 1, b)$
  $b_i \leftarrow (b_i - t)/T_{i,i}$
enddo
```

```
backward triangular solve:
for $i = n - 1, 1$ step $-1$ do
  $b_i \leftarrow b_i - dot(i, i + 1, n, b)$
enddo
```

The row-update operation in the inner loop of the incomplete factorization algorithm is performed only if $T_{k,j}$ is non-zero, and as such the implementation shown above is inefficient and would not be used in practice; it has complexity $O(n^2)$ due to program looping alone, irrespective of how many inner-loop updates are actually performed. In fact, algorithms whose complexity is $O(m)$ can be obtained; a simplified uniprocessor version of our parallel implementation is one such algorithm.

The parallelization of the forward solve algorithm follows directly from the observation that for a sparse triangular matrix it is possible to concurrently schedule some of the outer loop iterations, taking care to maintain the storage-dependencies implied by the sequential algorithm. Consider the directed graph $G_1 = (V_1, E_1)$, where the nodes V_1 correspond to rows of T, and there is an edge from node i to node j in G_1 whenever $T_{j,i} \neq 0$, $i < j \leqslant n$. We denote the cardinalities, $|E_1| = m_1 \leqslant m$ and $|V_1| = n$. The directed graph G_1 merely denotes the sparsity structure of the *lower-triangle* of T, and is therefore directly represented by the sparse matrix data structure used for it.

The algorithm for determining a level partitioning of the outer loop iterations for concurrent scheduling requires the following shared integer arrays and counters as workspace (*Note.* All arrays are dimensioned for the worst-case storage requirement):

1. *InEdge*(i), length $n+1$, for the predecessor counts of each vertex in G_1.

2. *Level*(i), length $n+1$, which eventually holds a list of vertices ordered in the sequence in which they will be executed.

3. *LevPt*(j), length $n+1$, holds pointers into the array *Level*(i), so that all vertices from *Level*(*LevPt*(j)) to *Level*(*LevPt*($j+1$) $- 1$) can be concurrently scheduled in the jth sequential step.

4. *nverts*, counter, for the number of vertices listed in the array *Level*.

5. *nlev*, counter, for the number of levels listed in the array *LevPt*.

The algorithm simply carries out a topological sort of the vertices of the directed graph G_1. It proceeds by recursively visiting the successors of all the vertices at a given level and decreasing their predecessor counts. When the predecessor count of a successor vertex becomes zero, it is added to the list in *Level*:

1. For each vertex $i \in V_1$, determine the predecessor vertex count and store the value in *InEdge*(i)

2. Locate the source vertex (we assume only one such vertex exists) and place it in *Level*(1). Also set *LevPt*(1) $= 1$ and initialize *nlev* and *nverts* to 0 and 1, respectively

3. Until all the nodes have been placed in the list do the following
   (a) Set *nlev* $\leftarrow$ *nlev* $+ 1$, *LevPt*(*nlev* $+ 1$) $\leftarrow$ *nverts* $+ 1$.
   (b) If *nverts* $= n$, then we are done. Exit.
   (c) Visit all the successor vertices of the vertices listed in *Level*(*LevPt*(*nlev*)) to *Level*(*LevPt*(*nlev* $+ 1$) $- 1$) and decrement their predecessor counts in the array *InEdge* by 1. If the predecessor count becomes 0, then add the vertex to the list in *Level* and set *nverts* $\leftarrow$ *nverts* $+ 1$.
   (d) Go to Step (a).

4. End.

A fully parallel implementation of this algorithm is possible by noting that Step (3c) above can be executed as a parallel DO-loop if a non-blocking memory instruction such as the Fetch-and-Add is used to modify the locations in *InEdge* and the counter *nverts*, respectively. On ACE, the Fetch-and-Add instruction is not supported in hardware, but is available through software emulation and the run-time library defaults to a critical-section implementation. The overhead of this critical section, is exactly why we compute the schedule as a preprocessing step, rather than performing the scheduling directly during the numerical computation of the triangular solve.

The actual numerical computation of the forward triangular solve with this schedule is carried out as follows

```
for $ilev = 1$, $nlev$ step 1 seqdo
  for $il = LevPt(ilev)$, $LevPt(ilev + 1) - 1$ pardo
    $i \leftarrow Level(il)$
    $b_i \leftarrow (b_i - dot(i, 1, i-1, b))/T_{i,i}$
  endpardo
endseqdo
```

The algorithm for the back triangular solve is very similar, except that a directed graph $G_2 = (V_2, E_2)$ which is generated by the non-zero entries in the *upper-triangle* of T is used. The procedure closely follows the description for the parallelization of the forward solve given above.

We now turn to the parallelization of the incomplete factorization. One straightforward approach starting from the sequential algorithm is to note that the individual row-updates in the inner loop are independent, and can therefore be performed in parallel. For sparse matrices, however, additional parallelism can be obtained by scheduling some of the outer loop iterations concurrently. The data dependency in the sequential algorithm that must be respected is that a given row j becomes an eligible pivot row and can begin updating other rows as soon as it is updated by all rows k with $T_{j,k} \neq 0$, $k < j$. It is easy to see that the earliest opportunity at which a given row j is an eligible pivot row is determined exactly by the level-partitioning computed for the graph in the parallel forward solve algorithm, and we therefore assume all the required information to be available in the arrays *Level* and *LevPt*, and in the shared counter *nlev*. The only additional difficulty is that two different pivot rows, with indices say j_1 and j_2, may simultaneously try to update the same target row, leading to multiple-writers on the same segment of shared data.

The first approach to resolve this difficulty is to synchronize memory accesses to ensure mutually-exclusive, atomic writes on the relevant shared memory locations. This approach will clearly only be effective if a fast (preferably hardware-implemented) synchronization primitive is available. A variant of this approach is to use a critical section for an entire row update operation, which leads to a somewhat larger granularity of work per synchronization, but this, however, can be offset by the fact that the synchronization–waiting due to process contention at such a critical section is also correspondingly increased.

The second approach that may be used requires no additional synchronization for the memory updates. It works by ensuring that if at all there is a possibility of a similtaneous update on the same target row by two different source rows, then these updates are performed on the same processor. This enforces the sequentiality of these updates and takes care of the multiple-writers problem. The most conservative algorithm for implementing this approach is one which we term *lazy-evaluation*, where updates to a given target row are delayed until all the source rows that will modify it have become eligible pivot rows; all updates to this target row are then scheduled together for execution on a single processor. The disadvantage of this approach can easily be seen with a few test execution graphs; by delaying updates in this way we obtain a longer critical path so that the execution time is increased irrespective of the number of processors that are used. At the same time, a critical path analysis alone is inconclusive, given the reduction in the synchronization cost, the better local memory utilization, and finally, the smaller gather–scatter overhead that is realized when multiple updates on a target row are performed sequentially on a single processor.

The appropriate algorithm in this situation of competing efficiency trade-offs therefore depends on several issues, including the sparsity structure of matrix, the relative latencies to local and global memory, and the cost of synchronization. The approach that we have taken here is a novel hybrid of the two possible schemes outlined above. In it we use the aggressive scheduling of target row updates to obtain the shorter critical paths characteristic of the first method. We also perform a preprocessing analysis in which the possible simultaneous update of the same target row are identified and scheduled for execution on the same processor, in order to obtain the low synchronization costs and better local memory utilization of the second method. This algorithm which is described in further detail below requires a list of edge-pairs (*Source*, *Target*) which identify the source and target row indices for each row-update operation, along with an associated list *Group*, which identifies edge-pairs with the same target row index for grouped execution on the same processor. The following integer arrays and counters are required as shared workspace:

1. *Source*(i), length m, for the list of source row indices

2. *Target*(i), length m, for the list of corresponding target row indices

3. *Group*(i), length m, contains flags for each pair in the *Source* and *Target* lists; if $\geqslant 1$, it indicates the number of succeeding edge-pairs that have the same target row index; if 0 it indicates that the associated edge-pair has the same target row index as a preceding edge-pair.

4. *LisPt*(i), length $n+1$, pointers into the *Source* and *Target* lists identifying edge-pairs that can be scheduled for concurrent execution of the corresponding row updates.

5. *nedge*, contains the number of edges that have been placed on the *Source* and *Target* lists, respectively.

The algorithm then proceeds as follows:

1. Set $nedge = 0$, $ilev = 0$, and $LisPt(1) = 1$. Then do the following:
   (a) Set $ilev \leftarrow ilev + 1$. If $ilev = nlev$, then we are done. Exit.
   (b) Visit each vertex j in the list $Level(LevPt(ilev))$ and $Level(LevPt(ilev+1)-1)$. For each edge emanating from vertex j, identify the target vertex k. Set $nedge \leftarrow nedge + 1$, and $Source(nedge) = j$, $Target(nedge) = k$.
   (c) Set $LisPt(ilev+1)$ to the number of edges added in Step (b) above.
   (d) Go to Step (a).
2. For each *ilev* from 1 to $nlev - 1$ do the following:
   (a) Sort the entries in *Target* between the locations $LisPt(ilev)$ and $LisPt(ilev+1)-1)$. Passively move the corresponding entries in *Source*. This step is designed to move all the edges with the same target vertex within the same level into contiguous locations for easy identification.
   (b) Scan the sorted entries of *Target* between the locations $LisPt(ilev)$ and $LisPt(ilev+1)-1$ and count multiple entries. If the count is 1, then set the corresponding entry in *Group* with the value 1. Otherwise, if the count is $nn > 1$, then set the first corresponding entry in *Group* equal to *nn* and the next $nn - 1$ entries to 0.

In the uniprocessor implementation, step (2) is unnecessary since then the issue of write conflicts on shared data does not arise. Again, note that the use of the Fetch-and-Add primitive (or a critical section) would allow Step 1(b) above to be implemented using a parallel DO-loop. Step (2), on the other hand, is seen to be fully parallel over each level.

The actual numerical evaluation of the incomplete factors can now be carried out using the following procedure:

```
parallel incomplete factorization:
for $ilev = 1, nlev - 1$ seqdo
  for $i = LevPt(ilev), LevPt(ilev+1) - 1$ pardo
    $j \leftarrow LevPt(i)$
    call $rowscl(j)$
  endpardo
  barrier
  for $i = LisPt(ilev), LisPt(ilev+1) - 1$ pardo
    if $(Group(j) \neq 0)$ then
```

```
            for $ig = i,\ i + Group(j) - 1$ seqdo
              $j \leftarrow Source(ig)$
              $k \leftarrow Target(ig)$
              call $rowupd(j, k)$
            endseqdo
          endif
        endpardo
      endseqdo
```

In this algorithm, whenever multiple updates are performed on the same target row in the innermost loop, i.e., when the value of *Group*(j) is greater than one, the target row may be copied into local memory and held in expanded form between updates to reduce the latency and the number of gather–scatter operations that are required in this update.

The black-box approach to parallel preconditioning described here can be generalized further, particularly towards internally restructuring the stiffness matrix so that a preconditioner with either more parallelism or better convergence properties can be obtained. These two considerations can often conflict, as is well known for the matrix generated from the finite-difference discretization of the two-dimensional Laplacian operator with a 5-point stencil on a rectangular domain. For the SSOR and point-incomplete-factorization preconditioners, the lexicographic nodal ordering provides much less parallelism than the red–black nodal ordering in the application of the preconditioner. However, the red–black preconditioner has a much slower rate of convergence, and this trade-off has to be taken into account in assessing its overall effectiveness. Schrieber and Tang [23] have suggested the use of mesh coloring algorithms to automatically generate suitable parallel incomplete-factorization preconditioners for an arbitrary finite element mesh, but they do not implement a program. In any event, the effectiveness of the parallel preconditioners obtained by restructuring in this way is an interesting open issue.

6. Discussion of Experimental Results

In this section we present experimental timing measurements on ACE, for our programs with some model problems specialized from the general formulation of Section 2. These model problems use regular domains and discretizations only for simplicity of programming and clarity of exposition; the methods are clearly applicable to more general situations.

A. *Model Problem for Convection–Diffusion Equation*

The following problem was solved on the domain $\{(x, y) : 0 < x, y < 1\}$

$$\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}\right) - Pe\,\frac{\partial c}{\partial x} = f(x, y), \tag{20}$$

TABLE I

Timings (in Seconds) for Various Parts of the Program in the Test Problems of Section 6A

	Bilinear elements			Biquadratic elements		
Procs.	*Assem*	*Setup*	*Consq*	*Assem*	*Setup*	*Consq*
1	50.0	9.6	110.7	103.2	20.6	160.9
2	25.1	7.7	60.7	52.1	15.2	89.5
3	16.9	6.5	43.1	35.1	12.0	64.0
4	12.7	6.1	34.6	26.4	10.7	51.9
5	10.4	6.0	29.7	21.5	10.2	43.9
6	9.0	6.2	26.8	18.7	10.0	39.8
7	7.8	6.3	24.0	15.7	10.0	35.7

where the function $f(x, y)$ and Dirichlet boundary conditions were chosen to obtain a known exact solution for c, so that the correctness of the computation could be verified; these details are not essential to the rest of discussion here.

Two test problems using meshes with 40×40 bilinear elements and 20×20 biquadratic elements, respectively, were generated, and in each case this leads to [illegible] initial guess, 16 iterations of the preconditioned CGS method were sufficient to reduce the relative 2-norm of the residual below 10^{-6}. The timings obtained for some of the important phases of the computation are shown in Table I. These include timings for the assembly of the stiffness matrix (*assem*), the setting up of the preconditioner (*setup*), and the iterative part of the CGS algorithm (*consq*). The setup time includes all the preprocessing, as well as the numerical computation of the incomplete factorization. The timings for *consq* will depend on the number of iterations, and hence on the desired solution accuracy. However, since these iterations are all executed sequentially, the speedups computed from these timings will be independent of the number of iterations.

The speedups for the assembly and the iterative part computed from Table I are shown plotted in Fig. 1. For the assembly, the departure of the curves from linearity reflects the increased contention at the critical section when more active processes are used. The alternative assembly algorithm described in Section 4 might be preferred in this case, but we have no experimental results for it as yet. The iterations of the CGS algorithm also parallelize well, reflecting the generally highly parallel nature of the computations in it. As the detailed timings below show, the drop-off in the efficiency with increasing numbers of processors is due to the rather small granularity of parallel work for this problem size and, to a certain extent, due to the limited speedup obtained in the application of the preconditioner. The setup part has the poorest speedup characteristics, which levels off with more than three–four processors; this is due to the relatively large synchronization overhead in this portion of the computation, and it is precisely here that hardware support for a Fetch-and-Add instruction might have led to greater efficiency.

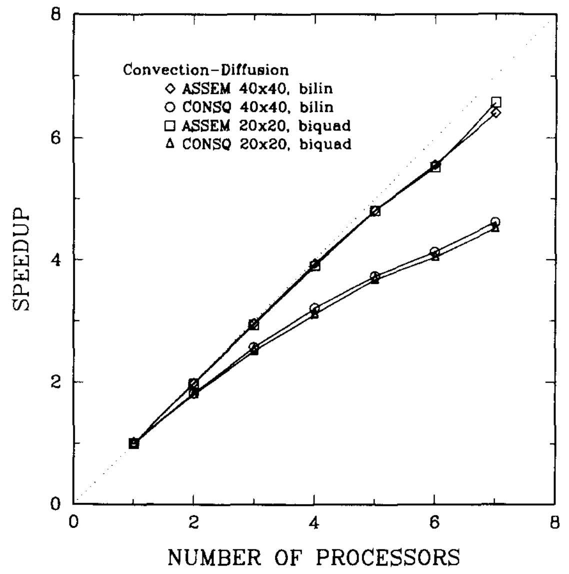

FIG. 1. Speedups for the matrix assembly and the iterative part of the CGS solver for the convection–diffusion problem.

A low-level breakdown of the execution times for some of the individual routines in CGS for the biquadratic elements test problem is given in Table II, and speedups are shown plotted in Fig. 2. These timings include the symbolic preprocessing for the sparse triangular solves (*pssol1*) and the actual numerical computation (*pssol2*), the symbolic preprocessing for the incomplete factorization (*pilu1*) and the actual numerical computation (*pilu2*), and the matrix-vector multiplication (*mvmul*). The measurements had to be carried out rather carefully to obtain the two significant decimal places reported, because of the rather coarse granularity (60 Hz) of the timer.

TABLE II

Timings (in Seconds) for Some Routines in the CGS Algorithm for the Biquadratic Elements Test Problem in Section 6A

Procs.	*pssol1*	*pssol2*	*pilu1*	*pilu2*	*mvmul*
1	2.49	2.85	3.55	12.7	1.84
2	1.50	1.64	2.16	6.90	0.96
3	1.21	1.22	1.54	4.74	0.66
4	1.05	1.02	1.25	3.69	0.50
5	0.96	0.90	1.08	3.07	0.42
6	0.90	0.84	0.98	2.68	0.37
7	0.88	0.81	0.90	2.42	0.31

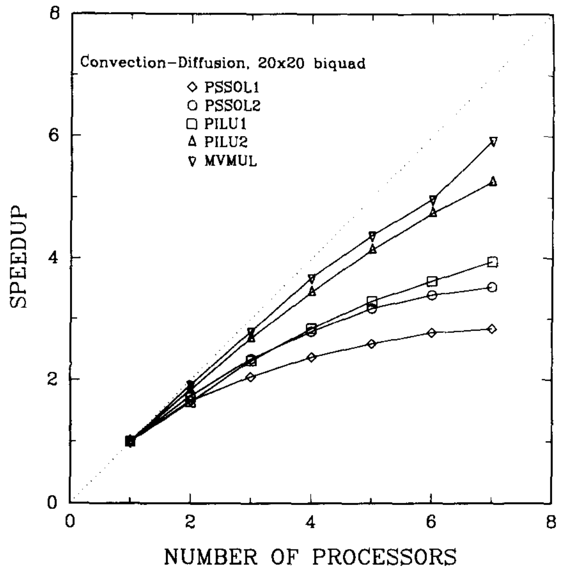

FIG. 2. Speedups for the preconditioning routines used in the CGS solver for the convection-diffusion problem for a 20×20 biquadratic elements mesh.

From Fig. 2, we see that the speedups obtained for the preprocessing routines *pssol1* and *pilu1* in the biquadratic elements case are 2.8 and 3.6, respectively, on six processors. As expected, the performance in the numerical parts, i.e., in *pssol2* and *pilu2* is somewhat better; with six processors, speedups of 3.4 and 4.7, respectively, are obtained. The parallel efficiency obtained for *pssol2* is especially critical, since this routine is invoked twice in each iteration of CGS to perform the preconditioning, and the results that we obtain for it are comparable to that reported in Baxter *et al.* [3] for problems of equivalent size. The speedup for *mvmul* is 5.0 on six processors, and we attribute the less than perfect speedup for this fully parallel routine to the small granularity of parallelization.

B. *Model Problem for Stokes Equation*

In this case, (7)–(9) was solved on the domain $\{(x, y): 0 < x, y < 1\}$, with zero velocity boundary conditions throughout, except for the top surface $y = 1$, where the x-component of the boundary velocity was set to unity. This is the classic "driven-cavity" problem and the discretization (15) is applicable. Two meshes were chosen for detailed experimental study. The first mesh consists of 8×8 elements and for it the order of the linear systems solved in the inner and outer iterations was 450 and 64, respectively. The second mesh consists of 16×16 elements, and required linear systems of 1666 and 256 to be solved in the inner and outer iterations, respectively. An absolute convergence criterion of 10^{-5} for the outer iteration and 10^{-8} for the inner iteration was used, and these values are appropriate in view of the discretization error and the scaling of the problem variables. As an

TABLE III

The Residual Reduction and the Number of Inner Iterations for Each Outer Iteration of the IOCG Algorithm in Section 6B

Outer	8 × 8 elements		16 × 16 elements	
iter. no	Residual norm	No. of inner iters.	Residual norm	No. of inner iters.
1	0.67e-1	12	0.45e-1	19
2	0.65e-2	12	0.50e-2	19
3	0.13e-2	12	0.10e-2	19
4	0.42e-3	10	0.33e-3	15
5	0.12e-3	9	0.10e-3	12
6	0.48e-4	8	0.28e-4	11
7	0.10e-4	7	0.13e-4	10
8	0.16e-5	7	0.43e-5	9

implementation issue, we mention that the sparse matrix A is actually two separate copies of the equivalent discrete Laplacian for each velocity component, except for the different boundary conditions on the top surface. However, the matrix was not stored in this fashion, but rather the ordering obtained by numbering the unknowns at each node consecutively was used, which is consistent with the usual finite element practice of trying to obtain a stiffness matrix of low bandwidth for direct methods. Note that if the two velocity components are coupled, then the ordering will not affect the storage requirement in the sparse format, but the incomplete factorization preconditioner that is obtained will be different in each case.

The convergence history of the outer iteration of the IOCG algorithm is shown in Tablr III. It is seen that the number of inner iterations required to reduce the inner residual to a fixed tolerance decreases by almost a factor of two as better initial guesses are available during the progress of the outer iteration to convergence. In Table IV, we show the timings and parallel efficiencies for the entire IOCG algorithm on these two test problems. The corresponding speedups are

TABLE IV

Timings (in Seconds) and Parallel Efficiencies for the Entire IOCG Algorithm of Section 6B

	8 × 8 elements		16 × 16 elements	
Procs.	Time	Eff.	Time	Eff.
1	301.2	1.00	1796.5	1.00
2	183.2	0.82	1007.0	0.89
3	136.7	0.73	726.1	0.82
4	120.5	0.62	601.1	0.74
5	105.6	0.57	541.1	0.66
6	99.9	0.50	437.0	0.69
7	97.3	0.44	—	—

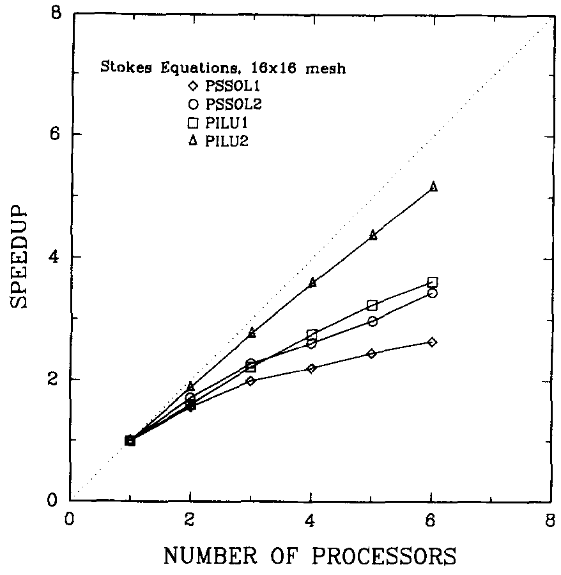

FIG. 3. Speedups for the entire IOCG algorithm for the Stokes equations. Results for two different mesh sizes are shown.

shown plotted in Fig. 3. The results show, as expected, that the overheads are reduced for the larger problem size, resulting in higher parallel efficiencies for them. In this algorithm it is especially critical that the inner conjugate gradient procedure be implemented efficiently, since it accounts for the dominant cost in the computation, so that a substantial payoff is obtained from a careful implementation and an effective preconditioner for it.

One straightforward optimization in the IOCG algorithm results from the fact that the matrix to be inverted in the inner loop is unchanged between outer iterations, so that its incomplete factorization may be computed once initially, and then reused in subsequent iterations. Our informal measurements in the specific case of

TABLE V

Timings (in Seconds) for the Incomplete Factorization Preconditioner Routines in the Inner Loop of IOCG for the 16 × 16 Elements Test Problem of Section 6B

Procs.	*pssol1*	*pssol2*	*pilu1*	*pilu2*
1	3.74	4.74	6.74	37.97
2	2.40	2.77	4.21	20.15
3	1.88	2.08	3.03	13.72
4	1.69	1.81	2.44	10.57
5	1.52	1.59	2.08	8.69
6	1.41	1.38	1.86	7.36

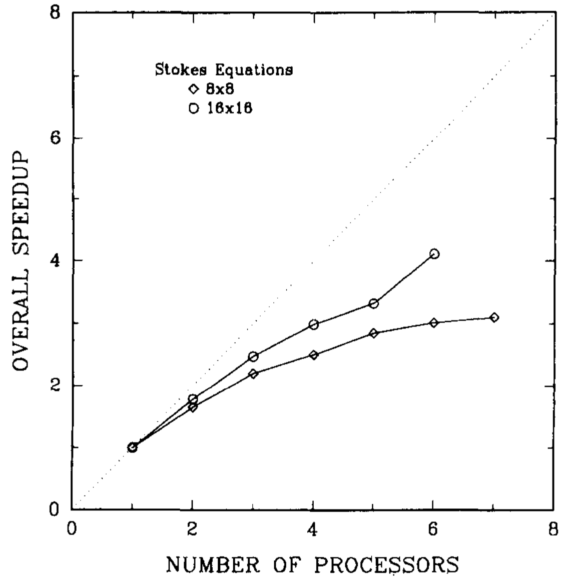

FIG. 4. Speedup performance of the preconditioning routines in the inner conjugate gradient loop of the IOCG algorithm for the Stokes equations. Results are for the 16×16 mesh.

the 16×16 element mesh problem shows the overhead of computing the incomplete factorization to amount to roughly about 24% of the time of the inner iteration in the parallel algorithm, so that the potential savings can be quite substantial.

In Table V, we show the timings obtained for the various preconditioning routines. These results, and the speedups shown in Fig. 4, are similar to those of the previous subsection. In particular, a continuous decrease in the execution time is seen as the number of processors is increased in the range studied. For example, with six processors we obtain speedups of 2.65 and 3.62 respectively for the two symbolic routines *pssol1* and *pilu1*, and speedups of 3.43 and 5.16 respectively for the numerical routines *pssol2* and *pilu2*.

7. SUMMARY AND FUTURE WORK

We have described implementation techniques and experimental results for finite element applications on a small shared-memory parallel computer. The primary focus was on the use of polynomial iterative methods with incomplete factorization preconditioners for solving linear systems of algebraic equations. A strategy for parallelizing these algorithms while maintaining the intrinsic flexibility of the finite element method was discussed, and shown experimentally to lead to reasonable parallel efficiencies on the ACE multiprocessor system. Overall, the speedup trends for the rather small problems that we have studied are encouraging and seem to demonstrate the viability of the approach developed here for realistic, large-scale applications on moderately parallel, shared memory computers. Since the architectural parameters of various parallel computers will differ widely, it is difficult to obtain a truly general implementation that is equally efficient across the range of target architectures. Nevertheless, we believe that our approach will be efficient without further modification on many of these architectures, and we are currently exploring this aspect for more realistic "production-type" applications on commercial supercomputers. In addition, the model problems of this paper may be generalized by using a building-block approach to develop parallel implementation strategies for more complicated applications. For example, the Stokes equation solver can be used in the inner loop of a Picard-type nonlinear iteration for solving the full Navier–Stokes equations. Similarly, the methods for the convection–diffusion equation may be extended to multicomponent transport problems or even to problems that involve both flow and transport (such as te Boussinesq equations) by suitably combining the various algorithms described in this paper.

APPENDIX 1

```
      function ddot(n,dx,dy)  /* inner-product of two shared arrays */
      @SHARED /ddotxx/ dglb
      real dglb                      /* shared variable declaration */
      @ENDSHARED
      real dx(1),dy(1),dloc
      integer i,n,ichnk
      @SERBEG
      dglb = 0.0e0          /* one process initializes global variable */
      @SEREND
      dloc = 0.0e0        /* all processes initialize private variable */
      ichnk = ((n - 1)/ @NPROCS) + 1        /* compute chunksize */
      @DO 30 i = 1, n, 1 CHUNK = ichnk
      dloc = dloc + dx(i) * dy(i)         /* dot-products in parallel */
30    continue
      @ENDDO 30
      @LOCK
      dglb = dglb + dloc     /* sum private values in critical section */
      @ENDLOCK
      @BARRIER   /* ensure all processes have updated global counter */
      ddot = dglb   /* make private copy of dot-product and return it */
      return
      end
```

ACKNOWLEDGMENTS

I thank the members of the ACE development group, especially Armando Garcia, for making this research possible, and Professor S. C. Eisenstat (Yale) for helpful comments on a draft of this paper.

REFERENCES

1. E. ANDERSON AND Y. SAAD, *Int. J. High Speed Comput.* **1**, 73 (1989).
2. C. C. ASHCROFT AND R. G. GRIMES, *SIAM J. Sci. Statist. Comput.* **9**, 122 (1988).
3. D. BAXTER, J. SALTZ, M. SCHULTZ, S. EISENSTAT, AND K. CROWLEY, Yale University Research Report YALEU/DCS/RR-629, 1988 (unpublished).
4. R. BERNSTEIN AND K. SO, IBM T. J. Watson Research Center Technical Report RC 13600, 1988 (unpublished).
5. A. S. BOLMARCICH, IBM T. J. Watson Research Center Technical Report RC 14369, 1989 (unpublished).
6. J. H. BRAMBLE AND J. E. PASCIAK, *Math. Comput.* **50**, 1 (1988).
7. C. CUVELIER, A. SEGAL, AND A. A. VAN STEENHOVEN, *Finite Element Methods and Navier–Stokes Equations* (Reidel, Dordrecht, 1986).
8. I. S. DUFF, *SIAM J. Sci. Statist. Comput.* **5**, 270 (1984).
9. F. G. GUSTAVSON, *IBM Tech. Discuss. Bull.* **16**, 357 (1973).
10. M. FORTIN, *Int. J. Numer. Methods Fluids* **1**, 347 (1981).
11. A. GARCIA, D. FOSTER, AND R. FREITAS, IBM T. J. Watson Research Center Technical Report RC 14491, 1989 (unpublished).
12. V. GIRAULT AND P. A. RAVIART, *Finite Element Approximation of the Navier–Stokes Equations*, Lecture Notes in Mathematics, Vol. 749 (Springer-Verlag, Berlin, 1979).
13. L. A. HAGEMAN AND D. M. YOUNG, *Applied Iterative Methods* (Academic Press, New York, 1981).
14. S. W. HAMMOND AND R. SCHREIBER, RIACS Technical Report 89.24, 1989 (unpublished).
15. D. S. KERSHAW, *J. Comp. Phys.* **26**, 43 (1978).
16. D. R. KINCAID, T. C. OPPE, AND W. D. JOUBERT, University of Texas at Austin Research Report CNA-228, 1988 (unpublished).
17. J. A. MEIJERINK AND H. A. VAN DER VORST, *J. Comput. Phys.* **44**, 134 (1981).
18. R. NATARAJAN, IBM T. J. Watson Research Center Technical Report RC 68072, 1989 (unpublished).
19. J. M. ORTEGA, *Introduction to Parallel and Vector solution of Linear Systems* (Plenum, New York, 1988).
20. S. PISSANETZKY, *Sparse Matrix Technology* (Academic Press, New York, 1984).
21. Y. SAAD, *SIAM J. Sci. Statist. Comput.* **10**, 1200 (1989).
22. Y. SAAD AND M. H. SCHULTZ, Yale University Research Report YALEU/DCS/TR-425, (1985).
23. R. SCHREIBER AND W. P. TANG, in *Proceedings, Symp. Cyber 205 Appl., 1982.*
24. P. SONNEVELD, *SIAM J. Sci. Statist. Comput.* **10**, 36 (1989).
25. G. STRANG AND G. J. FIX, *An Analysis of the Finite Element Method* (Prentice–Hall, Englewood Cliffs, NJ, 1973).
26. F. THOMASSET, *Implementation of Finite Element Methods for the Navier–Stokes Equations* (Springer-Verlag, Berlin, 1981).