Implementation of Iterative Methods for Large Sparse Nonsymmetric Linear Systems On a Parallel Vector Machine

Sangback Ma and Anthony T. Chronopoulos

International Journal of High Performance Computing Applications 1990; 4; 9
DOI: 10.1177/109434209000400402

IMPLEMENTATION OF ITERATIVE METHODS FOR LARGE SPARSE NONSYMMETRIC LINEAR SYSTEMS ON A PARALLEL VECTOR MACHINE

Sangback Ma and Anthony T. Chronopoulos

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MINNESOTA
MINNEAPOLIS, MINNESOTA 55455

Summary

We restructure three outstanding iterative methods for large sparse nonsymmetric linear systems. These methods are CGS (conjugate gradient squared), CRS (conjugate residual squared), and Orthomin(k). The restructured methods are more suitable for vector and parallel processing. We implemented these methods on a parallel vector system. The linear systems for the numerical tests are obtained from discretizing four two-dimensional elliptic partial differential equations by finite difference and finite element methods. A vectorizable and parallelizable version of incomplete LU preconditioning is used. We restructured the subroutines to enhance the data locality in vector machines with storage hierarchy. Speedup was measured for multitasking by four processors.

The International Journal of Supercomputer Applications, Volume 4, No. 4, Winter 1990, pp. 9–24.

Introduction

The two most common ways to approximate the solution of partial differential equations are by discretization of the original problems, as by the finite difference method (FDM) and the finite element method (FEM). They both lead to sparse banded matrices. For many applications, a matrix dimension N greater than 10,000 is not uncommon. For this kind of problem a direct method, such as Gaussian elimination, cannot be used because of the prohibitive cost. A solution with reasonable cost is the use of iterative methods. For symmetric matrices, the conjugate gradient method with proper preconditioning can be successfully used. For nonsymmetric matrices, the conjugate gradient method does not apply. This difficulty can be overcome in several ways. We can solve $Ax = f$ by solving the normal equations: $A^T A\ x = A^T f$. However, the matrix $A^T A$ generally squares the condition number of the matrix A, which could lead to slower convergence. The generalized conjugate gradient method of Concus and Golub (1976) and Widlund (1978) solves the nonsymmetric linear system $Ax = b$ when the matrix A has positive definite symmetric part $M = (A + A^T)/2$. This requires the solution of an auxiliary linear system of equations, with M as the coefficient matrix. Also, the generalized conjugate residual works for real positive matrices (matrices with positive real part eigenvalues). However, the original algorithm requires a prohibitive amount of time and space. Vinsome proposed an algorithm, Orthomin(k), which requires storage of only k of previous direction vectors. Convergence can be obtained with any $k > 0$. The optimal k depends on the nonsymmetry of the problem. For symmetric problems, $k = 1$ yields the conjugate residual method.

The conjugate gradient squared method (CGS) was derived from the biconjugate gradients (BI-CG) method by simply squaring the residual and direction matrix polynomials (Sonneveld, 1989). CGS does not require multiplication by the transpose of a matrix. In CGS the residual is not biorthogonal and the directions are not biconjugate. However, it can be viewed as the result of polynomial preconditioning, with the polynomial varying from iteration to iteration. Thus, it turns out that CGS is in practice faster than BI-CG. CGS computes exactly the same parameters as BI-CG and so has exactly the same breakdown conditions. In fact, along the iteration of CGS one can superimpose a BI-CG iter-

"In this paper we restructure several iterative methods. The restructured versions have the vector updates, matrix-vector products, and dot products of each iteration grouped together, as far as possible. This provides better data locality. The two inner products are executed simultaneously, and this reduces by one the synchronization points of these methods.

ation with additional cost of one matrix-vector multiplication but without the need for multiplication by the transpose.

One important advantage of the CGS method over BI-CG is the absence of multiplication by the transpose. This is necessary when applying the linear iterative solver as an inner iteration of a Newton step to solve a nonlinear system of equations: $F(X) = 0$. If the iterative method only requires multiplication by the Jacobian matrix $A = (\partial F)/(\partial x)$, then we can approximate it by Taylor's expansion:

$$Av = \frac{F(x + \epsilon v) - F(x)}{\epsilon}.$$

This kind of approximation cannot be applied to approximate $A^T v$, and explicit evaluation of the Jacobian is then required. Another form of BI-CG gives rise to the conjugate residual squared (CRS) algorithm (Chronopoulos and Ma, 1989). The behavior of CRS is very similar to that of CGS.

Computers with a hierarchy of storage have been designed. They can be ranked, according to the processing speed, as follows: (1) scalar/vector registers, (2) cache memory (or local memory), (3) main memory, and (4) auxiliary disk. To make efficient use of the memory hierarchy, we need to maximize the data locality. In other words, the ratio (*Memory References*)/(*Floating Point Operations*) must be as low as possible. Practically, it means that we must keep the data either in cache or local memory or in vector registers as long as possible. Examples of vector computers with a memory hierarchy are the CRAY-2, Alliant FX/8, and IBM 3090. For these computers the numerical computations not only should be suitable for vectorization but also should have good data locality in order to achieve near-maximum performance. For parallel computer systems processor synchronization is a serious bottleneck in achieving peak performance. Thus, reducing the number of synchronization points in a numerical computation makes it more suitable for parallel processing.

For an iterative method, preconditioning the given linear system substantially reduces the number of iterations. We used incomplete LU factorization (ILU(0))

(Meijerink and Van Der Vorst, 1977) in this paper. However, ILU factorization has to solve $L\,y = z$, $U\,x = y$, which is a serial recurrence. For a vector machine, this causes serious slowdown. In this paper we adopted the von Neumann series approach by Van Der Vorst (1982). Also, the above-mentioned recurrences become a bottleneck in parallel execution. Radicati di Brozolo and Robert (1988) proposed a technique to handle this problem, and achieved a high speedup on an IBM 3090 with six processors. Our results on a CRAY-2 with four processors seem to confirm this.

In this paper we restructure the iterative methods Orthomin(k), CGS, and CRS. The restructured versions have the vector updates, matrix-vector products, and dot products of each iteration grouped together, as far as possible. This provides better data locality for these computations. The two inner products are executed simultaneously, and this reduces by one the synchronization points of these methods. We implemented the standard and restructured methods on a four-processor CRAY-2 supercomputer. We used the vectorizable ILU preconditioning on a single processor and an overlapped submatrix parallel ILU(0) version (Radicati di Brozolo and Robert, 1988) on the four-processor system. The numerical tests are two-dimensional elliptic partial differential equations discretized by finite difference and finite element methods.

Section 1 gives a general background for block-tridiagonal matrices; section 2 describes three iterative methods, their restructured versions, and the preconditioning schemes. Section 3 deals with vectorization and parallelization aspects of implementation on the CRAY-2, and section 4 describes the test problems. In section 5 numerical results are shown, and finally, in section 6, conclusions are drawn.

Results and Discussion

1. A MODEL PROBLEM

Let us consider the second-order elliptic partial differential equation problem in two dimensions in a rectangular domain Ω in R^2 with homogeneous boundary conditions:

$$-(au_x)_x - (bu_y)_y + (cu)_x + (du)_y + fu = g, \qquad (1)$$

where $u = 0$ on $\partial\Omega$ and $a(x,y)$, $b(x,y)$, $c(x,y)$, $d(x,y)$, and $g(x,y)$ are sufficiently smooth functions defined on Ω, and $a,b > 0$, $c,d,f \geq 0$ on Ω. Discretization of the above equation by FDM or FEM leads to a linear system of equations, where the matrix is sparse. Let Ω be $(0,1) \times (0,1)$, with n grid points and $h = 1/(n+1)$ as the mesh size in both x and y directions, and the unknowns be ordered in the natural ordering. For the finite difference solution we used central difference for the first-order terms and five-point difference for the second-order terms. Hence, the whole discretization has a truncation error of $O(h^2)$, where h is mesh size. This leads to a block-tridiagonal matrix with five diagonals. The FDM discretization gives a linear system of equations:

$$Ax = f$$

of order $N = n^2$. If $c(x,y)$ or $d(x,y)$ is nonzero, then resulting matrix A is a nonsymmetric, block-tridiagonal matrix of the form

$$A = [C_k, D_k, E_k], \qquad (2)$$

where $D_k, 1 \leq k \leq n$, C_k, E_k, $1 \leq k \leq n-1$ are matrices of order n. The blocks have the following form

$$C_k = \mathrm{Diag}[c_1^k, \ldots, c_n^k]$$
$$E_k = \mathrm{Diag}[e_1^k, \ldots, e_n^k]$$
$$D_k = [d1_i^k, d_i^k, d2_i^k],\ 1 \leq i \leq n,$$

with $d1_0^k = d2_n^k = 0$, and $d_i^k > 0$, $c_i^k < 0$, $e_i^k < 0$, $d1_i^k < 0$, $d2_i^k < 0$.

On the other hand, let $L(u)$ be the partial differential operator. The finite element method seeks an approximate solution of the form

$$u = \sum_{k=1}^{N} u_k \phi_k, \qquad (3)$$

where ϕ_k are the basis functions which are one at node k, and zero at other nodes. If we use Lagrangian basis functions with square elements, then the resulting matrix is also block-tridiagonal (see Johnson, 1987; Lapidus and Pinder, 1981). In this case the matrix has nine diagonals, rather than five for FDM. The form of the resulting matrix is

$$A = [C_k, D_k, E_k] \qquad 1 \leq k \leq n, \tag{4}$$

where D_k, $1 \leq k \leq n$, C_k, E_k, $1 \leq k \leq n-1$ are matrices of order n. The blocks have the following form

$$C_k = [c1_i^k, c_i^k, c2_i^k],\ 1 \leq i \leq n,$$
$$D_k = [d1_i^k, d_i^k, d2_i^k],\ 1 \leq i \leq n,$$
$$E_k = [e1_i^k, e_i^k, e2_i^k],\ 1 \leq i \leq n,$$

with $d1_0^k = d2_n^k = 0$, and $d_i^k > 0$, $c_i^k < 0$, $e_i^k < 0$, $d1_i^k < 0$, $d2_i^k < 0$.

2. ITERATIVE METHODS

We next describe three conjugate gradient-like iterative methods to solve $Ax = f$, where A is a nonsymmetric matrix of order N. The Orthomin(k) works for the symmetric part of A being positive definite, and the conjugate gradient squared, and conjugate residual squared converge for general matrices provided that the iteration does not break down. Here, we used right preconditioning, because in this form we can compute the same residual as in the nonpreconditioned case.

2.1. ORTHOMIN(k)

The generalized conjugate residual method (GCR) (Eisenstat, Elman, and Schultz, 1983) is a direct generalization of the conjugate residual method (CR) for symmetric and positive definite linear systems. In the absence of round-off error, GCR gives the exact solution in at most N iterations. The main difference between GCR for nonsymmetric matrix and CR for symmetric matrix is that in the ith iteration of GCR, we have to keep in storage all previous $i-1$ direction vectors, and compute p_i based on $A^T A$ orthogonality of p_i and p_j, $j < i$. Thus, as i gets larger, the costs and storage become prohibitive. The GCR method converges for A nonsymmetric with the symmetric part positive definite.

Vinsome (1976) proposed the Orthomin(k), as a practical version of GCR with storage requirement for k directional vectors. Eisenstat, Elman, and Schultz (1983) proved that Orthomin(k), $k > 0$ converges. Note that Orthomin(1) is the CR algorithm for a symmetric matrix. Here, Pr is the right preconditioner. We used right preconditioning, since it minimizes the residual norm rather than minimizing the norm of $Pr\ r_i$, where r_i is the ith residual vector.

2.1.1 ALGORITHM ORTHOMIN(k)

1. Choose x_0.
2. Compute $r_0 = f - A X_0$
3. $p_0 = r_0$

For $i = 0$ step 1 **Until** Convergence **Do**

4. $a_i = \dfrac{(r_i, A p_i)}{(A p_i, A p_i)}$
5. $x_{i+1} = x_i + a_i p_i$
6. $r_{i+1} = r_i - a_i A p_i$
7. Compute $A P_r r_{i+1}$.
8. $p_{i+1} = P_r r_{i+1} + \sum_{j=j_i}^{i} b_j^i p_j$, where
9. $b_j^i = -\dfrac{(A P_r r_{i+1}, A p_j)}{(A p_j, A p_j)}, j \leq i.$
10. $A p_{i+1} = A P_r r_{i+1} + \sum_{j=j_i}^{i} b_j^i A p_j.$

Endfor

In this algorithm $j_i = \max(0, i-k+1)$, $j_i = 0$ for the GCR method. The number of vector operations (i.e., operations on vectors of length N, including multiplication/division and addition/subtraction) per iteration is $(6k + 10)N + 1\ Mv + 1\ Wprec$, where Mv stands for matrix-vector product and $Wprec$ is the preconditioning work. We ignored isolated scalar operations.

In the following we present the restructured version of Orthomin(k).

2.1.2 ALGORITHM FOR RESTRUCTURED ORTHOMIN(k)

1. Choose x_0.
2. Compute $r_0 = f - Ax_0, P_r r_0, A r_0$
3. $p_0 = r_0, a_0 = \dfrac{(r_0, A r_0)}{(A r_0, A r_0)}$, $b_{-1} = 0$

For $i = 0$ step 1 **Until** Convergence **Do**

4. $p_i = P_r r_i + \sum_{j=j_{i-1}}^{i-1} b_j^{i-1} p_j$
5. $A p_i = A P_r r_i + \sum_{j=j_{i-1}}^{i-1} b_j^{i-1} A p_j.$
6. $x_{i+1} = x_i + a_i p_i$
7. $r_{i+1} = r_i - a_i A p_i$
8. Compute $P_r r_{i+1}, A P_r r_{i+1}$.

9. $b_j^i = -\dfrac{(A\,P_r\,r_{i+1}, Ap_j)}{(A\,p_j, A\,p_j)}, j \leq i.$

10. $a_{i+1} = \dfrac{(r_{i+1}, A\,P_r\,r_{i+1})}{(A\,P_r\,r_{i+1}, A\,P_r\,r_{i+1}) + \sum_{j=j_i}^{i} b_j^i (A\,P_r\,r_{i+1}, A\,p_j)}$

Endfor

The equation for a_i is proved as follows:

$$\begin{aligned}(Ap_{i+1}, Ap_{i+1}) &= (A\,r_{i+1} + \sum_{j=j_i}^{i} b_j^i\,A\,p_j, A\,p_{i+1}) \qquad (5)\\ &= (Ar_{i+1}, A\,p_{i+1})\\ &= (Ar_{i+1}, Ar_{i+1}) + \sum_{j=j_i}^{i} b_j^i (Ar_{i+1}, A\,p_j),\end{aligned}$$

and

$$\begin{aligned}(r_{i+1}, Ap_{i+1}) &= (r_{i+1}, Ar_{i+1} + \sum_{j=j_i}^{i} b_j^i\,A\,p_j), \qquad (6)\\ &= (r_{i+1}, A\,r_{i+1}).\end{aligned}$$

Here, we have used the fact that for $j < i$, $(r_i, A\,p_j) = (A\,p_j, A\,p_i) = 0$ (Eisenstat, Elman, and Schultz, 1983).

The difference between algorithms 2.1.1 and 2.1.2 is in the computation of a_i. In steps 4–7, four SAXPY (Single A times X Plus Y) or vector update operations can be grouped together in a DO-loop, reducing memory references using vector registers or local memory as the temporary storages. Also the two inner products are in steps 9 and 10, which are grouped together in a single DO-loop. This reduces the number of synchronization points by one per iteration, which would give better performance in parallel execution. We note that no increase in vector operations resulted from modifying the original algorithm.

2.2 CGS

The biconjugate gradient method was proposed by Fletcher (1976) to solve indefinite linear systems. CGS results from squaring the BI-CG matrix polynomials for r_i and p_i (Sonneveld, 1989). The CGS method has several advantages. First, it does not require multiplication of the transpose of a matrix times a vector. Second, in CGS the residual $r_n = \phi_n(A)^2 r_0$, while $r_n = \phi_n(A) r_0$ in BI-CG, where ϕ_n is a matrix polynomial. Hence, it

might have twice as fast the effect of reducing the residual error over BI-CG, which may lead to faster convergence than BI-CG. This may not be true in general (see Van Der Vorst (1989b). CGS breaks down whenever BI-CG does. In the absence of breakdown the CGS method converges in less than $N/2$ iterations. However, there are no easily checked conditions under which it converges. Each step requires about twice the amount of work necessary for the symmetric CG. Here, Pr is the right preconditioning operator.

2.2.1 ALGORITHM CGS

1. $r_0 = f - Ax_0$
2. $q_0 = p_{-1} = 0$
3. $\rho_{-1} = 1$

For $i = 0$ step 1 **Until** Convergence **Do**

4. $\rho_i = \tilde{r}_0^T r_{i,b_i} = \dfrac{\rho_i}{\rho_{i-1}}$
5. $u_i = r_i + b_i q_i$
6. $p_i = u_i + b_i(q_i + b_i p_{i-1})$
7. $v_i = A P_r p_i$.
8. $\sigma_i = \tilde{r}_0^T v_i$
9. $a_i = \dfrac{\rho_i}{\sigma_i}$
10. $q_{i+1} = u_i - a_i v_i$
11. $x_{i+1} = x_i + a_i P_r(u_i + q_{i+1})$
12. $r_{i+1} = r_i - a_i A P_r(u_i + q_{i+1})$.

Endfor

Vector operations per iteration are $19N + 2Mv + 2Wprec$, where Mv stands for matrix-vector product, and $Wprec$ for preconditioning work.

As we did for Orthomin(k), we can also restructure the computation in algorithm 2.2.1, to obtain a form that is more suitable for computers with a memory hierarchy.

2.2.2 ALGORITHM FOR RESTRUCTURED CGS

1. Compute $r_0 = f - Ax_0, P_r r_0, A P_r r_0$.
2. $a_0 = \dfrac{(r_0, r_0)}{(A P_r r_0, r_0)}$, $b_0 = 0, q_0 = P_r q_0 = A P_r q_0 = 0$.

For $i = 0$ step 1 **Until** Convergence **Do**

3. $u_i = r_i + b_i q_i$.

4. $P_r u_i = P_r r_i + b_i P_r q_i$.
5. $A P_r u_i = A P_r r_i + b_i A P_r q_i$.
6. $v_i = A P_r u_i + b_i(A P_r q_i + b_i v_{i-1})$.
7. $q_{i+1} = u_i - a_i v_i$.
8. Compute $P_r q_{i+1}$.
9. $x_{i+1} = x_i + a_i(P_r u_i + P_r q_{i+1})$.
10. Compute $A P_r q_{i+1}$.
11. $r_{i+1} = r_i - a_i(A P_r u_i + A P_r q_{i+1})$.
12. Compute $P_r r_{i+1}$, and $A P_r r_{i+1}$.
13. $b_{i+1} = \dfrac{(r_{i+1}, \tilde{r}_0)}{(r_i, \tilde{r}_0)}$.
14. $a_{i+1} = \dfrac{(r_{i+1}, \tilde{r}_0)}{(A P r r_{i+1}, \tilde{r}_0) - \dfrac{b_{i+1}}{a_i}(r_{i+1}, \tilde{r}_0)}$.

Endfor

In steps 13 and 14, a_i, b_i in CGS are the same as those in BI-CG (Chronopoulos and Ma, 1989). In BI-CG (Fletcher, 1976), we have

$$(Ap_i, \overline{p_i}) = (Ar_i + b_{i-1} A p_{i-1}, \overline{p_i}) = (Ar_i, \overline{p_i}) = (Ar_i, \overline{r_i} + b_{i-1}\overline{p}_{i-1}). \quad (7)$$

Here, we have used $(Ap_j, \overline{p_i}) = (r_i, \overline{r_j}) = 0, j < i$, from Eqs. (5.2) and (5.3) of Fletcher (1976). (Here, b_{i-1} in Fletcher [1976] is b_i in algorithms 2.1.1 and 2.1.2.). Thus, $(Ap_i, \overline{p_i})$ becomes

$$= (Ar_i, \overline{r_i}) + b_{i-1}(A r_i, \overline{p}_{i-1}) = (Ar_i, \overline{r_i}) + b_{i-1}(r_i, A^T \overline{p}_{i-1})$$
$$= (Ar_i, \overline{r_i}) + b_{i-1}\left(r_i, \frac{\overline{r}_{i-1} - \overline{r}_i}{a_{i-1}}\right)$$
$$= (Ar_i, \overline{r_i}) - \frac{b_{i-1}}{a_{i-1}}(r_i, \overline{r_i}).$$

Let $\phi_i(A)$, $\psi_i(A)$ be the polynomials, such that the residual vectors and direction vectors in BI-CG are $r_i = \phi_i(A)(r_0)$, $p_i = \psi_i(A)$, $\phi_0(A) = I$, $\psi_0(A) = I$. Now, using Eq. (7) and the fact that $r_i = \phi_i(A)^2 r_0$ in CGS, we get b_i, a_i in steps 13 and 14. Note that the dot products are collected in steps 13 and 14, and steps 3 through 7 consist of SAXPY or double SAXPY computations, which can be grouped in one DO-loop. Also both steps 8 and 9 and steps 9 and 10 could be grouped into one DO-loop, respectively. The differences between the standard and restructured algorithms are in computing the matrix-vector products and computing α_i. In algorithm 2.2.2, A times $P_r q_i$, and A times $P_r r_i$ were computed. In both cases, the number of the matrix-vector product is the same. For the computational costs, algorithm 2.2.2 has two more SAXPY operations (only one more in the nonpreconditioned case) in steps 4 and 5.

2.3 CRS

CRS is a modification of CGS (Sonneveld, 1989). It is developed from the conjugate residual method, just as CGS is developed from the conjugate gradient method. Hence, its behavior and performance are expected to be similar to those of CGS. It turns out that CRS is a special case of CGS, with $\tilde{r}_0 = A^T r_0$. Hence, if A^T is available, then CRS takes the same number of operations per iteration as CGS. If A^T is not available, then we must introduce additional linear combinations over CGS. The following algorithm is then obtained.

2.3.1 ALGORITHM CRS

1. $r_0 = Ax_0 - f$
2. $q_0 = p_{-1} = 0$
3. $AP_r q_0 = P_r q_0 = AP_r p_{-1} = 0$
4. $\rho_{-1} = 1$

For $i = 0$ step 1 **Until** Convergence **Do**

5. $\rho_i = \tilde{r}_0^T AP_r r_i, b_i = \dfrac{\rho_i}{\rho_{i-1}}$
6. $u_i = r_i + b_i q_i, P_r u_i = P_r r_i + b_i P_r q_i, AP_r u_i$
   $= AP_r r_i + b_i AP_r q_i$
7. $p_i = u_i + b_i(q_i + b_i p_{i-1})$
8. $v_i = AP_r u_i + b_i AP_r q_i + b_i^2 AP_r p_{i-1}$
9. $\sigma_i = \tilde{r}_0^T AP_r v_i$
10. $a_i = \dfrac{\rho_i}{\sigma_i}$
11. $q_{i+1} = u_i - a_i v_i, AP_r q_{i+1} = AP_r u_i - a_i AP_r v_i, P_r q_{i+1}$
    $= P_r u_i - a_i P_r v_i$
12. $r_{i+1} = r_i - 2a_i AP_r u_i + a_i^2 AP_r v_i$
13. $x_{i+1} = x_i - a_i Pr(u_i + q_{i+1})$

Endfor

In step 8, $v_i = AP_r p_i = AP_r u_i + b_i AP_r q_i + b_i^2 AP_r p_{i-1}$ was used. The only difference from CGS is in steps 4 and 8 in computing ρ_i and σ_i. (In Polak et al. [1987] a similar algorithm appears but requires four matrix-vector products.)

Vector operations per iteration are $29N + 2Mv + 2W_{prec}$, where Mv stands for matrix-vector product,

and *Wprec* for preconditioning work. For the preconditioned case the costs are $34N + 2Mv + 2Wprec$; $5N$ for the additional costs comes from computing $P_r(u_i + q_{i+1})$ in steps 6 and 11.

We omit the restructured version, since it is identical to CGS, if $\tilde{r}_0 = A^T r_0$.

2.4 PRECONDITIONING

For the preconditioning matrix Pr, we look for matrices such that

$$P_r A \approx I$$

or $P_r A$ has the clustered eigenvalues, and the linear system $Pr\,x = y$ is easy to solve. One natural choice is the ILU factorization (Meijerink and Van Der Vorst, 1977), where $A = LU + E$, where $L_{i,j} = U_{i,j} = 0$, if $A_{i,j} = 0$, and $E_{i,j} = 0$ if $A_{i,j} \neq 0$. In other words, L,U have the same sparsity patterns as A. Let $NZ(A)$ denote the set of pairs of $[i,j]$ for which the entries a_{ij} of the matrix A are nonzero, the *nonzero pattern of A*.

2.4.1 ALGORITHM FOR THE ILU FACTORIZATION

For $i = 1$ step 1 **Until** N **Do**
  For $j = 1$ step 1 **Until** N **Do**
    If (i,j) belongs to $NZ(A)$) **Then**

$$s_{ij} = A_{ij} - \sum_{t=1}^{\min(i,j)-1} L_{it}U_{tj}$$

If $(i \geq j)$ Then $L_{ij} = s_{ij}$

If $(i < j)$ Then $U_{ij} = \dfrac{s_{ij}}{L_{ii}}$

    Endif
  Endfor
Endfor

Here, we set $U_{ii} = 1$, for $1 \leq i \leq N$. Denoting L_{ij} and U_{ij} by $\tilde{a}_{ij}$, the ILU(0) preconditioner matrix for the finite difference matrix is generated as follows.

For $i = 1$ step 1 **Until** N **Do**

$$\begin{aligned}
\tilde{a}_{i,i-1} &= a_{i,i-1} \\
\tilde{a}_{i,i} &= a_{ii} - a_{i-1,i}\tilde{a}_{i,i-1} - \tilde{a}_{i-nx,i}\tilde{a}_{i,i-nx}, 1 \leq i \leq N. \\
\tilde{a}_{i,i+1} &= \frac{a_{i,i+1}}{\tilde{a}_{i,i}}
\end{aligned}$$

Endfor

For the finite element case, the ILU(0) preconditioner matrix is generated as follows.

For $i = 1$ step 1 **Until** N **Do**

$$\begin{aligned}
\tilde{a}_{i,i-nx-1} &= a_{i,i-nx-1} \\
\tilde{a}_{i,i-nx} &= a_{i,i-nx} - \tilde{a}_{i,i-nx-1}\tilde{a}_{i-nx-1,i-nx} \\
\tilde{a}_{i,i-nx+1} &= a_{i,i-nx+1} - \tilde{a}_{i,i-nx}\tilde{a}_{i-nx,i-nx+1} \\
\tilde{a}_{i,i-1} &= a_{i,i-1} - \tilde{a}_{i,i-nx-1}\tilde{a}_{i-nx-1,i-1} - \tilde{a}_{i,i-nx}\tilde{a}_{i-nx,i-1} \\
\tilde{a}_{i,i} &= a_{i,i} - \tilde{a}_{i-1,i}\tilde{a}_{i,i-1} - \tilde{a}_{i-nx-1,i}\tilde{a}_{i,i-nx-1} \\
&\quad - \tilde{a}_{i-nx,i}\tilde{a}_{i,i-nx} - \tilde{a}_{i-nx+1,i}\tilde{a}_{i,i-nx+1} \\
\tilde{a}_{i,i+1} &= \frac{a_{i,i+1} - \tilde{a}_{i,i-nx}\tilde{a}_{i-nx,i+1} - \tilde{a}_{i-nx+1,i+1}\tilde{a}_{i,i-nx+1}}{\tilde{a}_{i,i}} \\
\tilde{a}_{i,i+nx-1} &= \frac{a_{i,i+nx-1} - \tilde{a}_{i,i-1}\tilde{a}_{i-1,i+nx-1}}{\tilde{a}_{i,i}} \\
\tilde{a}_{i,i+nx} &= \frac{a_{i,i+nx} - \tilde{a}_{i,i-1}\tilde{a}_{i-1,i+nx}}{\tilde{a}_{i,i}} \\
\tilde{a}_{i,i+nx+1} &= \frac{a_{i,i+nx+1}}{\tilde{a}_{i,i}}
\end{aligned}$$

Endfor

3. VECTORIZATION AND PARALLELIZATION

3.1 VECTORIZATION

Implementation on a vector computer requires proper vectorization of the computations to take advantage of the full single-processor capacity. Vectorization for these methods is relatively straightforward. The matrix is stored in diagonals, so that matrix-vector multiplication is fully vectorized. Also the inner products and linear combinations are vector operations. The only computation that needs further attention is that of the preconditioning step,

$$LUx = y,$$

which consists of solving the two triangular systems, $L\,z = y$ and $U\,x = z$, where L and U are the incomplete LU factors, consisting of five diagonals (nine in FEM). Solution of these systems requires back-solving, which is a serial operation. We can solve this problem by using a

von Neumann series expansion, proposed by Van Der Vorst (1982), or by using the block preconditioning approach of G. Meurant (1984). In this paper we adopted the first method. Assume the given matrix A is block-tridiagonal. We can write $A = LU + \text{err}$. Assume further that the diagonal entries of L and U are 1, by scaling of the original problem. Then we can write $L = I + E + F$, where E is the matrix consisting of subdiagonal $L_{i,i-1}$, F is the matrix consisting of the rest of the subdiagonals L_{ij}, $j < i - 1$. Solving

$$(I + E + F)z_i = y_i$$

amounts to

$$(I + E)z_i = y_i - Fz_{i-1}.$$

Assuming E is small relative to I in norm, we expand in von Neumann series,

$$\begin{aligned} z_i &= (I + E)^{-1}(y_i - Fz_{i-1}) \\ &= (I - E + E^2 - E^3 + \ldots)(y_i - Fz_{i-1}), \end{aligned} \tag{8}$$

and we truncate the power series at the mth term. In the report of Van Der Vorst (1982) $m = 2$ was chosen for the incomplete Cholesky preconditioned conjugate gradient methods. We also chose $m = 2$. The assumption about the norm of E relative to I is satisfied if the matrix is diagonally dominant. None of our four test problems seems to violate this assumption. The backward solution for U is computed similarly. This scheme vectorizes the computation, with vector length = nx, where nx is the number of grid points in the x-direction using the natural ordering. The wavefront technique can also achieve vectorization by appropriate renumbering of the nodes (Filippone and Radicati di Brozolo, 1988; Van Der Vorst, 1989a).

3.2 PARALLELIZATION

In the codes of Orthomin, CGS, and CRS with or without preconditioning, the iteration loop cannot be executed in parallel, since the residual vectors and the directional vectors require the previous ones to be updated. Note that in these codes, most of the execution time is spent in the computational kernels, such as the matrix-vector product, the dot product, the SAXPY, or

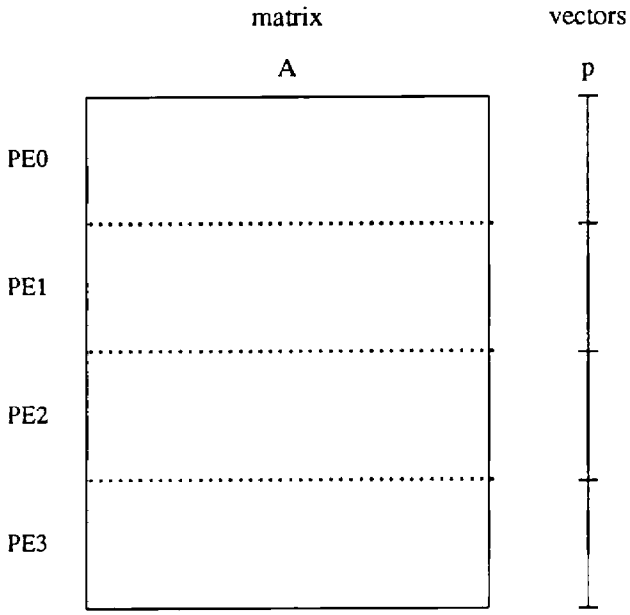

Fig. 1 Parallelization of matrix-vector product

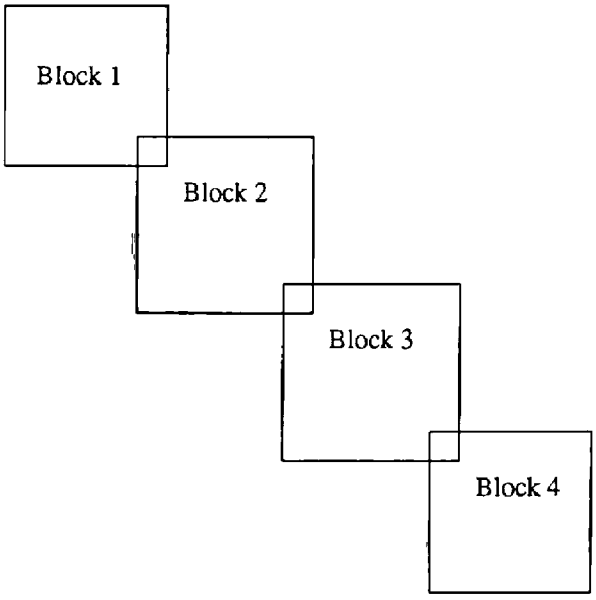

Fig. 2 Parallel execution of ILU matrix-vector product with overlapping

the double SAXPY. So we could realize reasonable speedup from parallelization of these kernels. For the preconditioning block, the recurrence in (8) poses difficulties to the parallelization. We adopted the technique of Radicati di Brozolo and Robert (1988). It solves the preconditioned matrix system by dividing into four submatrices, ignoring the original relation between those submatrices. However, we can make up for this loss of connection by introducing an overlapping region between consecutive submatrices. Then we take the average of the computed values by the subsystem in the overlapped regions. According to Radicati di Brozolo and Robert (1988) this overlapping strategy gives better performance than the nonoverlapping one.

The implementation of this parallelization is illustrated in Figures 1 and 2.

4. TEST PROBLEMS

Problem 1 (Elman, 1982):

$$-(b(x,y)u_x)_x - (c(x,y)u_y)_y + (d(x,y)u)_x + (e(x,y)u)_y + f(x,y)u = g(x,y),$$
$$\Omega = (0,1) \times (0,1)$$

where $b(x,y) = e^{-xy}, c(x,y) = \beta(x+y), d(x,y) = \beta(x+y)e^{-xy}$

$$e(x,y) = \gamma(x+y), f(x,y) = \frac{1}{(1+xy)},$$
$$u(x,y) = xe^{xy}\sin(\pi x)\sin(\pi y),$$

with Dirichlet boundary condition and $g(x,y)$ the corresponding right-hand side function. By changing γ and β, we could control the degree of nonsymmetry of the discretization matrix. In our paper, we set $\gamma = 50$, $\beta = 1$. We have used the five-point difference scheme for the second-order derivatives and the central difference scheme for the first-order derivatives. For an initial guess vector, we have chosen $x(i) = 0.5*\text{mod}(i,50)/10$.

Problem 2 (The Convection Diffusion; Sonneveld, 1989):

$$-\epsilon(u_{xx} + u_{yy}) + \cos(\alpha)u_x + \sin(\alpha)u_y = 0$$
$$u(x,y) = x^2 + y^2 \text{ on } \partial\Omega$$
$$\Omega = (0,1) \times (0,1)$$

We have used the five-point difference scheme for the second-order derivatives and the central difference scheme for the first-order derivatives. For small ϵ values, we might need to use the upwind difference scheme for the first derivative to maintain diagonal dominance. In our experiment of $\epsilon = 0.1$ we used central difference. We used $\alpha = 0.5$. For initial value, we have chosen $x(i) = 0.5*\text{mod}(i,50)/10$.

Problem 3 (Berkeley; Sonneveld, 1989):

$$-\epsilon(u_{xx} + u_{yy}) + v_x u_x + v_y u_y = 0$$
$$v_x = 2y(1-x^2), v_y = -2x(1-y^2),$$
$$\Omega = (-1,1) * (0,1)$$
$$u(x,y) = 0, x = -1$$
$$u(x,y) = 0, x = 1$$
$$u(x,y) = 0, y = 1$$
$$u(x,0) = 1 + \tanh(10(2x+1)), y = 0, -1 \leq x \leq 0$$
$$\frac{\partial u}{\partial n} = 0, y = 0, 0 \leq x \leq 1$$

The boundary conditions are Dirichlet on all sides of the rectangle, except Neumann boundary on the half side [0,1] of the x-axis. For small ϵ values, we might need to use the upwind difference scheme for the first derivative to maintain diagonal dominance, but for this problem we used $\epsilon = 0.1$, and central differences. We have used the five-point difference scheme for the second-order derivatives and the central difference scheme for the first-order derivatives. For an initial guess vector, we have chosen $x(i) = 0.5*\text{mod}(i,50)/10$.

Problem 4 (Sonneveld, 1989):

$$-u_{xx} + u_x + (1+y^2)(-u_{yy} + u_y) = f(x,y)$$
$$\Omega = (0,1) \times (0,1)$$

where

$$u(x,y) = e^{(x+y)} + x^2(1-x)^2\ln(1+y^2)$$

with Dirichlet boundary conditions, $f(x,y)$ the corresponding right-hand side is computed to have the solution $u(x,y)$. We have used the five-point difference scheme for the second-order derivatives and the central difference scheme for the first-order derivatives. For an initial guess vector, we have chosen $x(i) = 0.5*\text{mod}(i,50)/10$.

5. NUMERICAL RESULTS

The CRAY-2 is a four-processor machine. Each processor can execute independent tasks concurrently. All processors have equal access to the large central memory. The CRAY-2 at Minnesota Supercomputer Center has 512 megawords of central memory. Each CRAY-2 processor has eight vector registers (each 64 words long) and has data access through a single path between its vector registers and main memory. Each processor has 16K words of local memory with no direct path to central memory but with a separate data path between local memory and its vector registers. Also, there are six parallel pipelines: common memory to vector register, load/store vector register to local memory, load/store floating addition/subtraction, floating multiplication/division, integer addition/subtraction, and logical pipelines. The central memory is divided into four quadrants, and assignment of four quadrants to four processors takes place and changes at each memory cycle. Hence, if more than one processor requests an access to the same quadrant, then a memory conflict will occur.

First, we ran experiments with $Nx = Ny = 128$ for problems 1, 2, and 4 and $Nx = 128$, $Ny = 64$ for problem 3, where Nx is the number of nodes in x-direction, and Ny in y-direction. For both FDM and FEM we used the vectorizable ILU(0) preconditioning, as described in sections 2 and 3.1. For Radicati's technique, we let the two consecutive submatrices overlap in region of two Nx nodes. For both FDM and FEM, we used natural ordering. We have not implemented Eisenstat's trick (Eisenstat, 1981). Table 1 contains the number of iterations needed for residual error norm at 10^{-6} for Orthomin(4), CGS, and CRS with ILU(0) preconditioning; Table 2 shows the number of vector FLOPS per iteration, where vector FLOP means componentwise addition or multiplication of length N, where N is the dimension of the matrix. Figures 3 through 6 show plots of the total FLOPS needed for each error norm. Our tests confirm the results reported by Sonneveld (1989). CRS turns out to be very similar to CGS.

Table 3 shows the speedups of Orthomin(4) and CGS for the Berkeley problem. (For one CPU we repeat

Table 1
Number of Iterations with Error Tolerance = 1.e − 6

Problem	Finite Difference	Finite Element
1: $Nx = 128$, $Ny = 128$, $\gamma = 50$, $\beta = 1$		
Orthomin (4)	373/111/112	341/106/110
CGS	253/56/57	246/44/50
CRS	234/55/61	216/45/52
2: $\epsilon = 0.1$, $\alpha = 0.5$, $Nx = 128$, $Ny = 128$		
Orthomin (4)	707/167/160	705/122/230
CGS	212/73/74	178/110/109
CRS	212/72/75	176/110/111
3: $\epsilon = 0.1$, $Nx = 128$, $Ny = 64$		
Orthomin (4)	324/99/107	263/72/77
CGS	205/72/67	139/33/39
CRS	207/77/75	140/34/36
4: $Nx = 128$, $Ny = 128$		
Orthomin (4)	378/112/125	312/98/109
CGS	222/78/80	180/57/59
CRS	208/65/78	181/55/63

The number after the first slash (/) is obtained by preconditioning; the number after the second slash is obtained by Radicati's parallel preconditioning.

Table 2
Number of Vector Floating Point Operations per Iteration

	Vector FLOPS	
Method	Finite Difference	Finite Element
Orthomin (4)	43/56	51/72
CGS	37/62	53/94
CRS	37(47)/62(77)	53(63)/94(109)

The number in the parentheses for CRS is the one for the case when A^T is not available.

the experiment until we get the CPU utilization ratio of 96%. We believe this CPU time is close enough to the real CPU time in a dedicated mode. For Tables 4 and 5, we repeated the experiments until we achieved the CPU utilization ratio of 85%. Considering the serial portion we believe this is close enough to the maximum.) We did not include CRS, because it is expected to be quite similar to CGS. For FDM with $Nx = 124$, $Ny = 64$, preconditioned CGS achieves an unusually high speedup of 3.66, which is due to the decrease in the number of iterations in the parallel case. On the other hand, $Nx = 128$, $Ny = 64$, with FEM, preconditioned CGS gives an unusually low speedup of 2.41, which is due to the unusual increase in the number of iterations. It seems that this unusual decrease or increase sometimes happens with Radicati's technique, and this phenomenon is also reported by Radicati di Brozolo and Robert (1988). We used the microtasking library for the parallelization of the inner DO-loops. Since the experiments were not performed in dedicated mode, we believe that the maximum speedup possible may be higher than shown here. For $Nx = 128$ or 256, the average speedup of the nonpreconditioned case is around 3.1, while for Radicati preconditioning cases it is around 2.7. These are somewhat lower than values reported by Radicati di Brozolo and Robert (1988).

Table 4 shows the total CPU time for the original algorithms and restructured ones for one CPU. Table 5 shows the same thing for four CPUs. The test problem for Tables 3–5 was the Berkeley problem with $Nx = 256$, $Ny = 128$, and $\epsilon = 0.1$. Since we have relied on the compiler to make efficient use of the local memory and the vector registers, the saving through restructuring is minimal, especially for CGS and CRS, which need more operations in the restructured version. But for Orthomin, restructuring is indeed faster, since restructuring versions have the same number of operations.

6. CONCLUSIONS

We implemented and tested three CG-like methods for both five-diagonal and nine-diagonal block-tridiagonal matrices arising from two-dimensional discretization of elliptic partial differential equations by finite difference

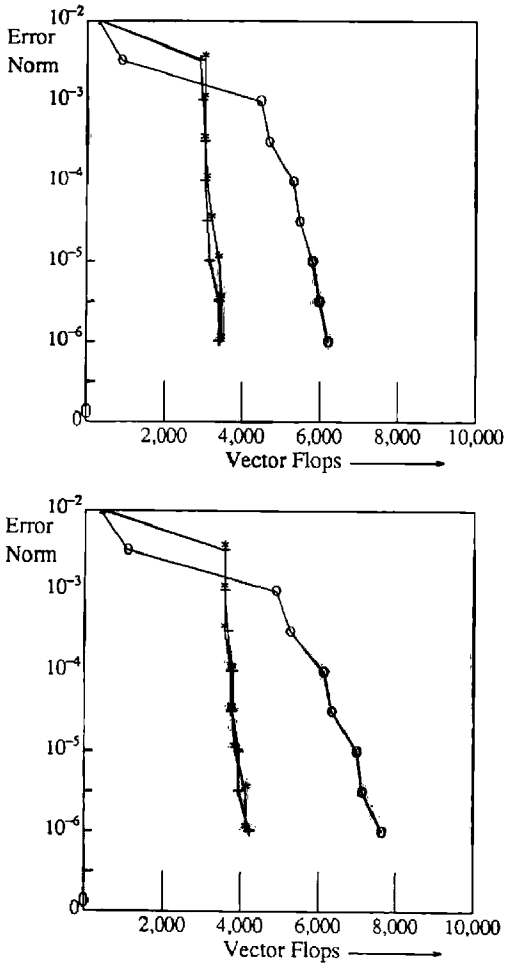

Fig. 3 Problem 1 Finite difference (top) and finite element (bottom) methods. (○) Orthomin(4); (*) CGS; (+) CRS.

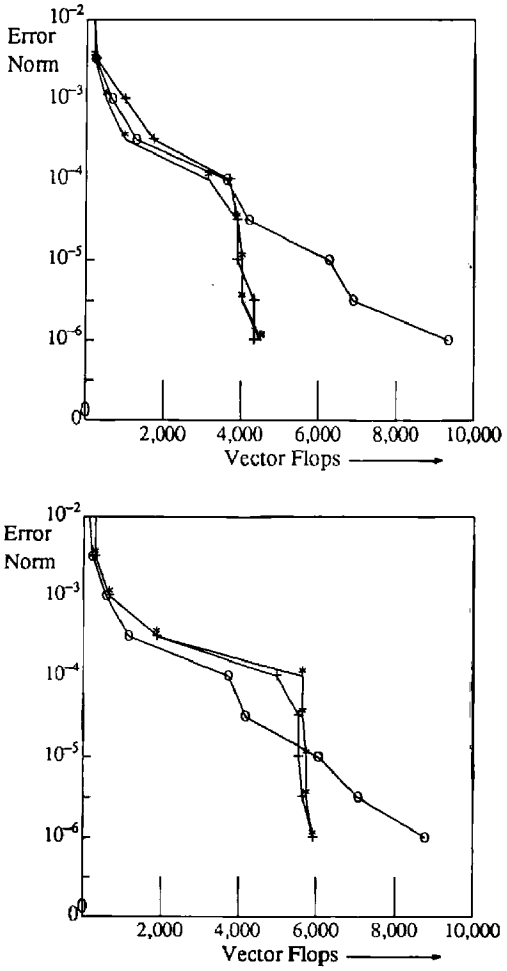

Fig. 4 Problem 2 Finite difference (top) and finite element (bottom) methods. (○) Orthomin(4); (*) CGS; (+) CRS.

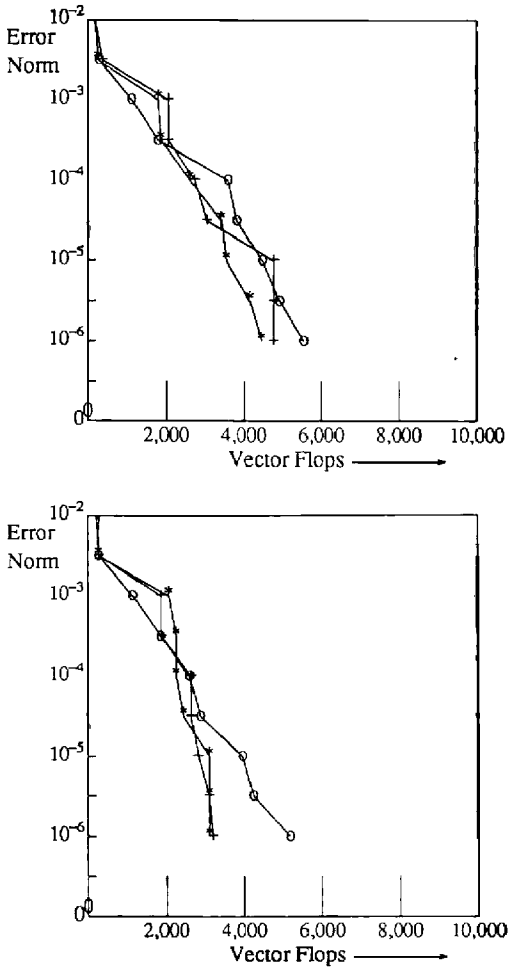

Fig. 5 Problem 3 Finite difference (top) and finite element (bottom) methods. (○) Orthomin(4); (*) CGS; (+) CRS.

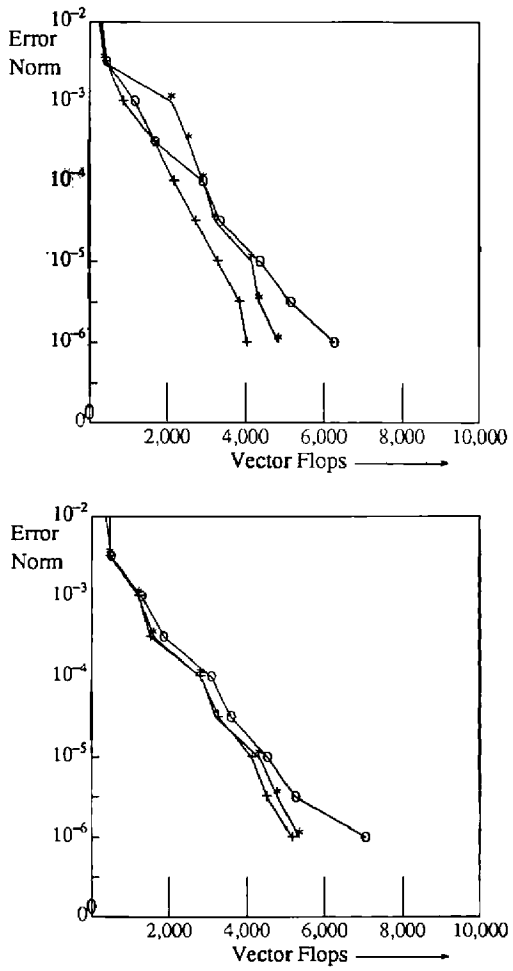

Fig. 6 Problem 4 Finite difference (top) and finite element (bottom) methods. (○) Orthomin(4); (*) CGS; (+) CRS.

Table 3
Speedup with Four Processors for Problem 3 with Error Tolerance = 10^{-2}

	Finite Difference			Finite Element		
Method	**64 × 32**	**128 × 64**	**256 × 128**	**64 × 32**	**128 × 64**	**256 × 128**
Orth (4)	2.29	3.01	3.15	3.14	3.15	3.05
CGS	2.11	3.19	3.26	2.19	3.19	3.05
Orth (4)-ILU (0)	3.05	3.17	2.57	2.30	3.05	2.54
CGS-ILU (0)	2.47	3.66	2.85	3.09	2.41	2.71

Table 4

Comparison of Restructured Algorithm vs. Original One: Total Elapsed Time (μsec) for One CPU with Error Tolerance = 10^{-6}

	Finite Difference		Finite Element	
Method	**Original**	**Restructured**	**Original**	**Restructured**
Orthomin (4)	5.77	5.36	6.31	6.21
CGS	4.11	3.98	3.48	3.44

No preconditioning was used.

and finite element methods. As with a serial machine, CGS and CRS turn out to be more efficient over Orthomin(4), as was reported by Sonneveld (1989). CGS and CRS seem to behave very similarly. Restructuring of the code for CRAY-2 reduced the CPU time by 5% on the average for one CPU. At least for Orthomin, this restructuring is guaranteed to be faster, since it has the same costs. For CGS and CRS, the overall saving may depend on the relative costs of memory references and synchronization of the particular machine. If we use the cache or local memory between main memory and vector registers, or the main memory and disk storage, the restructuring version is expected to be faster than the standard one.

For Radicati's parallel preconditioning with overlap size of 2 *Nx*, the number of total iterations needed is about the same as that in the serial case, but the speedup ranges from 2.6 to 3.2, in general. Since the experiments were performed in a nondedicated mode, the maximum speedup will be higher.

Table 5

Comparison of Restructured Algorithm vs. Original One: Total Elapsed Time (μsec) for Four CPU with Error Tolerance = 10^{-6}

	Finite Difference		Finite Element	
Method	**Original**	**Restructured**	**Original**	**Restructured**
Orthomin (4)	1.83	1.75	2.07	2.02
CGS	1.26	1.22	1.14	1.07

No preconditioning was used.

ACKNOWLEDGMENT

We thank the referees, whose comments helped enhance the quality of presentation of this paper. The research was partially supported by University of Minnesota graduate school grant-in-aid 0350-2104-07, and NSF grants CER DCR-8420935 and CCR-8722260. We acknowledge the Minnesota Supercomputing Institute for providing time on the CRAY-2.

BIOGRAPHIES

Anthony T. Chronopoulos obtained his B.Sc. and M.S. degrees in mathematics from the University of Athens (Greece) and the University of Minnesota in 1979 and 1981, respectively. He received his Ph.D. in computer science from the University of Illinois at Urbana-Champaign in 1986. Since 1986 he has been an assistant professor in the Department of Computer Science at the University of Minnesota. His interests include numerical algorithms and parallel processing.

Sangback Ma received his B.S. degree from Seoul National University of the Republic of Korea in 1978, and his M.S. degree in mathematics from University of Minnesota. Since 1987 he has been working toward a Ph.D. in computer science. His interests lie in the numerical solution of partial differential equations and its implementation on parallel vector computers.

SUBJECT AREA EDITOR

Iain Duff

REFERENCES

Chronopoulos, A., and Ma, S. 1989. On squaring Krylov subspace iterative methods for nonsymmetric linear systems. TR 89-67. Minneapolis: University of Minnesota, Computer Science Department.

Chronopoulos, A., and Gear, C. 1989a. S-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.* 25:153–168.

Chronopoulos, A., and Gear, C. 1989b. On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy. *Parallel Comput.* 11:37–53.

Concus, P., and Golub, G. 1976. A generalized conjugate gradient method for unsymmetric systems of linear equations. *Lecture Notes in Econom. and Math. Systems* 139:56–65.

Eisenstat, S. 1981. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Statist. Comput.* 2:1–4.

Eisenstat, S., Elman, H., and Schultz, M. 1983. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.* 20:345–357.

Elman, H. 1982. Iterative methods for large, sparse, nonsymmetric systems of linear equations. Ph.D. thesis, Yale University, Department of Computer Science.

Filippone, S., and Radicati di Brozolo, G. 1988. Vectorized ILU preconditioners for general sparsity patterns. Technical report. IBM ECSEC, Italy.

Fletcher, R. 1976. Conjugate gradient methods for indefinite systems. *Lecture Notes in Math.* 506:73–89.

Johnson, C. 1987. *Numerical solutions of partial differential equations by the finite element method.* Cambridge, U.K.: Cambridge University Press.

Koniges, A., and Anderson, D. 1987. ILUBCG2: A preconditioned bi-conjugate gradient routine for the solution of linear asymmetric matrix equations arising from 9-point discretizations. *Comp. Phys. Commun.* 43:297–302.

Lapidus, L., and Pinder, G. 1981. *Numerical solution of partial differential equations in engineering and science.* New York: Wiley.

Meijerink, J., and Van Der Vorst, H. 1977. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.* 31:148–162.

Meurant, G. 1984. The block preconditioned conjugate gradient method on vector computers. *BIT* 24:623–633.

Polak, S., Heijer, C., Schilders, W., and Markowich, P. 1987. Semiconductor device modeling from the numerical point of view. *Internat. J. Numer. Methods Engrg.* 24:763–838.

Radicati di Brozolo, G., and Robert, Y. 1988. Parallel and vector conjugate gradient-like algorithms for sparse nonsymmetric systems. Technical report. IBM ECSEC, Italy.

Sonneveld, P. 1989. CGS, a fast Lanczos-type solver for nonsymmetric systems. *SIAM J. Sci. Statist. Comput.* 10:36–52.

Van Der Vorst, H. 1982. A vectorizable variant of some ICCG methods. *SIAM J. Sci. Statist. Comput.* 3:350–356.

Van Der Vorst, H. 1989a. High performance preconditioning. *SIAM J. Sci. Statist. Comput.* 10(6):1174–1185.

Van Der Vorst, H. 1989b. The convergence behaviour of some iterative solution methods. Report 89-19. Delft University of Technology, The Netherlands.

Vinsome, P. W. 1976. Orthomin, an iterative method for solving sparse sets of simultaneous linear equations. SPE 5729. Society of Petroleum Engineers of AIME.

Widlund, O. 1978. A Lanczos method for a class of nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.* 15:801–812.