Solution of the Pressure Equation Using Krylov Methods On The CM-5

by

Danesh Tafti
NCSA &
Dept. of Mechanical and Industrial Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801

and

Mahadevan Ramaswamy
NCSA &
Dept. of Aeronautical and Astronautical Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

Several Krylov based linear solvers are tested for solving representative linear systems resulting from the discretization of the pressure-Poisson equation on the CM-5. Case studies with both symmetric and nonsymmetric diagonally dominant system matrices are conducted for orders N = 262144 and 3145728. The initial pressure fields, source term, and the coefficient matrices are generated from a Navier-Stokes solver. Two types of preconditioners, one based on matrix diagonalization and the other based on Jacobi iterations are tested. For symmetric cases, CG without any preconditioning was the most efficient. In the case of non-symmetric systems, BiCGSTAB, BiCGSTAB2 and their QMR versions were the most effective. In general, the Jacobi iterative preconditioner was quite effective with these methods and reduced the CPU time by a factor of 1.5-3.0. The Jacobi preconditioner was particularly effective with GMRES for the large system in which it reduced the CPU time by a factor of 8.0. Performance studies on 64 nodes of the CM-5 gave an estimated performance between 800 and 1000 MFLOPS for these methods.

1. Introduction

The efficient solution of large sparse system of equations is critical to many applications in science and engineering. In most cases for large general sparse systems, direct solution techniques are not feasible and/or are very difficult to implement effectively. On the other hand iterative techniques [1] offer a range of options which are often low in memory utilization and quite effective for diagonally dominant systems. A class of iterative methods which have gained wide acceptance are Krylov subspace methods [2]. These methods like most other iterative methods do not require the assembly of the whole system matrix in any form, but deal with a localized or elemental system. The basic operations in their implementation are vector inner products, matrix vector products and vector-vector and vector-scalar algebraic operations. Their attractiveness to application scientists stems from their relative programming simplicity and the fact that they can be effectively implemented on parallel architectures [3,4]. These class of methods have been widely researched as evidenced by the large volume of work available in the literature. It is our intent only to give a brief overview of the methods which have been used in this work.

The Conjugate Gradient (CG) method, first developed by Hestenes and Stiefel [5], proved to be an attractive method to solve symmetric positive definite (SPD) systems. The CG method is based on the generation of a series of orthogonal vectors (residual vectors $\mathbf{r}_n$) and search directions $(\boldsymbol{p}_n)$ which are used to successively update the solution. The residual vectors satisfy a three term recurrence relationship which is fundamental to the efficient implementation of the method. An important property of the CG method is for an SPD system it minimizes the function $\boldsymbol{r}_n^T \boldsymbol{A}^{-1} \boldsymbol{r}_n$ at each iterate. It theoretically converges to the exact solution within N iterations, where N is the order of the system. CG requires one matrix-vector multiply and two inner products per iteration. Derivatives of this method like Bi-Conjugate Gradient (BiCG) [6], Conjugate Gradient Squared (CGS) [7], Bi-Conjugate Gradient-Stabilized (BiCGSTAB [8] and BiCGSTAB2 [9]), have been developed to solve linear systems which are nonsymmetric. However, these methods no longer have the minimization property exhibited by CG and can have irregular convergence of the residual norm and can often break down. In this context it should be pointed out that, even though the CG method was developed and its theoretical convergence proved only for a SPD system, it has been successfully applied to solve nonsymmetric systems through suitable preconditioners [10]. BiCG is a variation of the CG method for nonsymmetric systems and generates two sequences of bi-orthogonal vectors based on the system matrix $\boldsymbol{A}$ and its transpose $\boldsymbol{A}^T$. It requires two matrix vector multiplies (one with matrix $\boldsymbol{A}$ and the other with $\boldsymbol{A}^T$) and two inner products. CGS is a variant of BiCG in that it eliminates the use of $\boldsymbol{A}^T$ and applies the updating operations for $\boldsymbol{A}$ and $\boldsymbol{A}^T$ to the same vectors. Each iterate consists of two matrix vector products and two inner products. Although in many situations, CGS exhibits faster convergence than BiCG, it has been noted to have very irregular convergence, especially when the initial guess is very near the solution [8]. The BiCGSTAB method combines the faster convergence of CGS and the stability of BiCG. It uses a local minimization to smooth the convergence behavior. It has two matrix vector multiplications and four inner products per iteration. Unlike BiCG and its variants mentioned above, GMRES generates and stores the whole sequence of orthogonal vectors by using the Gram-Schmidt [11] orthogonalization. The GMRES method is the most robust of all these methods for ill-conditioned or nonsymmetric non-diagonally dominant systems.

However, it requires the storage of the full orthogonal basis to the Krylov sequence and can become prohibitively expensive in terms of storage and work for large systems. To overcome this problem, a restarted version of GMRES called GMRES*(m)* [11] is used, where only m orthogonal vectors are computed and stored to give an approximate solution before the procedure is started again. There is no set criterion for the choice of m which is mostly governed by the trade off between memory utilization and CPU requirements.

Recently Freund and Nachtigal [12] proposed the quasi-minimal residual (QMR) method based on the look ahead version of the nonsymmetric Lanczos algorithm [13]. The QMR method still uses only short recurrences and the quasi-minimal residual property smoothes the convergence. However like BiCG, the QMR algorithm requires the transpose of $\mathbf{A}$. By squaring the QMR method, Freund and Szeto [14] derived the transpose free QMR squared method. The QMR method has also been used in conjunction with other methods to obtain smoother convergence. Freund [15] obtained the QMR version of CGS. Also Chan et al. [16] derived a QMR version of BiCGSTAB [8] . These methods provide smoother convergence over the original methods with improved CPU time.

Preconditioning is important for the efficient utilization of these methods. The number of iterations required to solve a system ($\boldsymbol{Ax} = \boldsymbol{b}$) varies as the square root of the condition number of the coefficient matrix, even for the best case (i.e. CG with symmetric positive definite coefficient matrix). For large ill conditioned systems (matrices with high condition number), the computational cost can be very high. This can be circumvented by transforming the coefficient matrix to an equivalent one with a lower condition number such that the transformed system yields the same solution as that of the original one. The process of solving a system ($Ax = b$), using preconditioning can be explained by transforming this into an equivalent one as

$$M_1^{-1} A M_2^{-1} (M_2 x) = M_1^{-1} b \tag{1}$$

where $M = M_1 M_2$ is the precondition matrix and M approximates A as closely as possible ($M \approx A$). Here the transformed coefficient matrix $M_1^{-1} A M_2^{-1}$ should ideally have as small a condition number as possible to reduce the number of iterations. In the above process if $M_1 = I$, it is called right preconditioning and if $M_2 = I$, it is called left preconditioning. For preconditioning to be effective, the additional cost of finding the suitable M ($=M_1 M_2$) should be offset by the saving from faster convergence, so that the overall cost of computation is reduced. In addition the computational costs of using a preconditioner also depends on the machine architecture on which the solution procedure is carried out, e.g. the preconditioning process involving significant amount of sequential operations may not be readily suitable on a parallel machine.

The simplest and least expensive type of preconditioning is what is often referred to as diagonal preconditioning. In this preconditioning the matrix A is approximated by its diagonal elements and the preconditioner $M = D$, where D is the diagonal of A. There is no cost incurred in finding this and minimal extra storage beyond that of the matrix itself is required. Even though this preconditioning has the above virtues, for a SPD matrix using diagonal preconditioning does not improve the convergence rate [17, p. 302]. However for nonsymmetric systems this type of preconditioning has been found to be effective [18].

Other preconditioners used are Jacobi, Gauss-Siedel type iterative solvers and Successive Overrelaxation methods (SOR) [1] which uses a relaxation parameter with the Gauss-Siedel technique for optimum convergence. However, in most cases it is not possible to find the optimum relaxation parameter and its use can be counterproductive [18]. An extension of SOR to symmetric systems called symmetric successive overrelaxation (SSOR) [1] follows one sweep in one direction with another in the other direction to maintain the symmetry of the preconditioner. The Gauss-Siedel method, SOR and SSOR although relatively simple to implement have dependencies in them which do not allow effective parallelization [19]. On the other hand the Jacobi method can be implemented in parallel quite easily.

Another class of preconditioners belong to the category where the precondition matrix is an approximate or incomplete factorization of the coefficient matrix. Here $\boldsymbol{M} = \boldsymbol{LU}$ where $\boldsymbol{L}$ and $\boldsymbol{U}$ are the lower and upper triangular matrices such that $\boldsymbol{A} = \boldsymbol{LU} + \boldsymbol{E}$ with $\boldsymbol{E}$ an error matrix (Meijerink and Van Der Vorst [20]). Unlike the earlier methods finding the precondition matrix in this case is not trivial. The most important aspect of the incomplete factorization is the computational cost encountered during the factorization process. Being mostly a sequential process these methods can lead to high costs when attempted on parallel machines. For some aspects of parallel implementation see Van Der Vorst [21] and Bramley et al. [22]. In polynomial preconditioners, the fact that for diagonally dominant matrices the inverse of $\boldsymbol{A}$ can be represented by a power series $\boldsymbol{B}$, where $\boldsymbol{B} = \boldsymbol{I} - \boldsymbol{A}$, is exploited. In such instances

$$\boldsymbol{A}^{-1} = \boldsymbol{I} + \boldsymbol{B} + \boldsymbol{B}^2 + \boldsymbol{B}^3 \ldots\ldots \qquad (2)$$

and by truncating the infinite series an approximation to A^{-1} can be obtained. Some examples of its use can be found in Saad [23] and Johnson et al. [24].

A number of researchers have used these methods to solve the compressible/incompressible Navier-Stokes equations both with finite difference/finite volume and finite element discretizations on a number of computer architectures. Venkatakrishnan [25] used the GMRES*(m)* method to solve the thin layer compressible Navier-Stokes equation. He found that GMRES*(m)* with incomplete LU preconditioning [20] was the most efficient on a Cray Y-MP. Vuik [26] used GMRES(m) with different preconditioners to solve the momentum and pressure equations within the framework of the SIMPLE [27] algorithm. Shakib et al. [28] have also used GMRES with various preconditioners to solve non-symmetric systems arising in finite element analysis. Johan et al. [29] extended this work to the CM-2 for complex unstructured mesh geometries. Xu et al. [30] used the GMRES solver with preconditioning for a fully implicit formulation on 8 transputers of a Meiko Computing Surface computer. They used a modified diagonal preconditioner and obtained good speedups in their parallel implementation. Chin et al. [31] used CG and CGSTAB [32] to solve the fully coupled incompressible Navier-Stokes on a staggered grid using ILU preconditioning. Baker et al. [33] used the CG method to solve large symmetric systems produced by an oil reservoir simulation on a distributed array of transputers. They found that Incomplete Cholesky Conjugate Gradient algorithm (ICCG) [34] was not very effective on distributed transputers and quickly lost its effectiveness. On the other hand the use of Jacobi iterations as a preconditioner to the conjugate gradient (JPCG) was found to parallelize well. Kadioglu and Mudrick [18] used GMRES*(m)* to solve elliptic problems arising in meteorology applications. They used different preconditioners based on incomplete LU decomposition [34], Jacobi or diagonal preconditioning, iterative Gauss-Siedel preconditioning and successive over-relaxation (SOR). They tested these preconditioners for two-dimensional and three-dimensional fields with N = 5832 to 100,440. They found that overall the diagonal preconditioning gave the best efficiency.

One of the most attractive features of the Krylov based methods is their relative simplicity and generality of application. They have the versatility to be used for general sparse matrices which would arise from various finite difference approximations in Cartesian or generalized body fitted coordinates. In the current study several of these methods are tested on the CM-5 in the data parallel mode of operation. The CM-5 at NCSA contains 512 processing nodes with each node having 32 MBytes of memory giving a collective memory of 16 GBytes. The processing nodes in each partition (CM-5 at NCSA can have partitions of 32, 64, 128, 256 and 512 nodes) are controlled by a control processor. The control processor broadcasts instructions to the nodes, initiates and synchronizes execution. The control processors and processing nodes are connected by scalable data and control networks. The data network is used for the bulk of inter processor communications and data movement.

Table 1 outlines the different Krylov based methods tested. The methods annotated by CMSSL_ are programmed by using the Connection Machine Scientific Libraries (CMSSL V3.1) [35]. The library provides the basic algorithmic structure for each method, however preconditioners and matrix vector multiplications have to be supplied by the user through a reverse communication interface. This allows the programmer to have considerable latitude in the layout of arrays to suit his/her particular needs in the framework of the larger problem being solved. Another advantage of using the CMSSL library calls is once the structure of the subroutine which calls this library has been established, all the Krylov based methods in the library can be used within this framework. It was also the intent of this work to compare hand coded implementations of some of these methods with those in CMSSL. In Table 1, CMSSL_QMRLAL uses a three term look ahead Lanczos procedure while CMSSL_QMR2 is based on two-term recursions without look ahead. Both these algorithms require the transpose of the system matrix $\boldsymbol{A}$ and the preconditioner $\boldsymbol{M}$.

The overall objective of this study was to use the methods outlined in Table 1 and to establish their usefulness and efficiency on the CM-5. Two preconditioners, both of which could be parallelized with relative ease were also tested with these methods. Further, symmetric and nonsymmetric systems generated from a Navier-Stokes solver were tested. The order of the systems ranged from $N = 262144$ to 3145728.

The report is organized as follows: Section 2 gives some background information on the pressure-Poisson equation and its characteristics. Section 3 describes the preconditioners used in the present study. Section 4 describes the implementation of these methods on the CM-5. Section 5 describes the data sets used to test the methods. Section 6 discusses the results and section 7 summarizes and concludes the report.

2. Governing Equations and Pressure-Poisson Equation

Since the linear systems used in the present work are generated from the discretization of the pressure Poisson equation, it would be useful to establish the framework within which the pressure equation is derived. The time-dependent, non-dimensionalized, incompressible continuity and momentum equations are of the form:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{3}$$

$$\frac{\partial u_i}{\partial T} + \frac{\partial}{\partial x_j}(u_i\, u_j) = -\frac{\partial p}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i}{\partial x_j^2} \tag{4}$$

where i = 1, 2, 3 corresponds to x (u), y (v), and z (w) coordinates (velocities) in three-dimensions and Re is the Reynolds number.

In the present work, equations (3) and (4) are solved on a staggered grid. The two-dimensional topology of this grid is shown in figure 1, in which the u-velocity and v-velocity are staggered with respect to pressure in the x and y direction, respectively. Time integration of the discretized equations is performed by using the fractional-step method of Kim and Moin [40]. First an intermediate velocity field $(\tilde{u}_i)$ is calculated. In this step the contribution of the pressure gradient to momentum balance is completely neglected. In the advancement to $(\tilde{u}_i)$, an explicit or a semi-implicit formulation can be used. The intermediate velocity field is then made divergence free at $(n+1)\Delta T$ (u_i^{n+1}) by solving a Poisson-pressure or pressure equation for p^{n+1}. The correction to $(\tilde{u}_i)$ to make the velocity field divergence free is given by:

$$u_i^{n+1} = \tilde{u}_i - \Delta T \frac{\partial p^{n+1}}{\partial x_i} \tag{5}$$

The pressure field p^{n+1} is obtained by applying the continuity equation (equation (3)) at time $(n+1)$ and solving the resulting pressure-Poisson equation

$$\frac{\partial}{\partial x_i}\left\{\frac{\partial p^{n+1}}{\partial x_i}\right\} = \frac{1}{\Delta T}\ \frac{\partial \tilde{u}_i}{\partial x_i} \tag{6}$$

The correct and consistent solution of this equation is extremely important in obtaining a divergence free velocity field. These issues have been dealt with in some detail in Tafti [41] and Najjar and Vanka [42]. The LHS of equation (6) is the Laplacian for pressure and the RHS is the mass source term of the intermediate velocity field. On a Cartesian grid, the system matrix formed by the finite-volume discretization of equation (6) is symmetric on a uniform as well as a nonuniform or stretched grid. However, for the sake of the present study we make the system nonsymmetric on a nonuniform grid by dividing equation (6) by the cell volume. Then the degree of nonsymmetry is a function of the stretching factor. Second-order accurate central difference approximations to the divergence and gradient operators are used to construct the Laplacian. These approximations result in a banded system which exhibit diagonal dominance. The number of off-diagonal bands exhibited by the system depend on the boundary conditions. They can vary from six for Neumann boundary conditions in all three directions to twelve bands for periodic boundary conditions in all three directions. The final form of the discretized equation can be written as:

$$\begin{aligned}(A_p)\, p^{n+1}_{(i,j,k)} = {} & A_n\, p^{n+1}_{(i,j+1,k)} \quad + A_s\, p^{n+1}_{(i,j-1,k)} \\ & + A_e\, p^{n+1}_{(i+1,j,k)} \quad + A_w\, p^{n+1}_{(i-1,j,k)} \\ & + A_h\, p^{n+1}_{(i,j,k+1)} \quad + A_l\, p^{n+1}_{(i,j,k-1)} \\ & - \frac{1}{\Delta T}\, \tilde{m}_{(i,j,k)}\end{aligned} \tag{7}$$

where the subscripts e, w, n, s, h, and l correspond to the neighboring nodes in the x, y and z directions and $\tilde{m}$ is the mass source. For periodic boundary conditions in a coordinate direction the following relationships are applied $p(n) = p(2)$ and

$p(1) = p(n\text{-}1)$. The property of symmetry is exhibited by the following relationships in each coordinate direction

$$
\begin{aligned}
A_{e(i,j,k)} &= A_{w(i+1,j,k)} \\
A_{n(i,j,k)} &= A_{s(i,j+1,k)} \\
A_{h(i,j,k)} &= A_{l(i,j,k+1)}
\end{aligned}
\tag{8}
$$

which in matrix notation corresponds to $\boldsymbol{A} = \boldsymbol{A}^T$ or on an elemental basis $a_{mn} = a_{nm}$, where $\boldsymbol{A}$ corresponds to the system matrix of order N. In the current formulation, these conditions are only satisfied when a uniform grid distribution is used in all coordinate directions. To get an estimate of the degree of nonsymmetry the following formula is used:

$$
U = \sqrt{\frac{1}{N} \sum_{N} (A_{e(i,j,k)} - A_{w(i+1,j,k)})^2 + (A_{n(i,j,k)} - A_{s(i,j+1,k)})^2} \tag{9}
$$

In this study the z direction is always periodic and hence $A_{h(i,j,k)} = A_{l(i,j,k+1)}$. Further for a symmetric system, $U = 0$.

3. Preconditioners

It was not the intent of this work to do an exhaustive study of the different preconditioners available in the literature. The choice of preconditioners was guided by simplicity and relative ease of implementation on the CM-5 parallel architecture. Hence, only two preconditioners which clearly fitted these requirements were used and tested. Both preconditioners utilize Jacobi iterations.

If matrix $\boldsymbol{A}$ is decomposed into $\boldsymbol{D} - \boldsymbol{L} - \boldsymbol{U}$, where $\boldsymbol{D}$ is a diagonal matrix consisting of a_{mm}, and $\boldsymbol{L}$ and $\boldsymbol{U}$ are lower and upper triangular matrices then the general expression for a Jacobi iterative preconditioner can be written as:

$$\boldsymbol{x}^{(k)} = \boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})\,\boldsymbol{x}^{(k-1)} + \boldsymbol{D}^{-1}\boldsymbol{b}, \qquad k = 1, 2, 3 \ldots\ldots K \quad (10)$$

where (k) is the iteration number.

3.1 <u>Jacobi or Diagonal Preconditioning *(D)*</u>

When K=1, and $\boldsymbol{x}^{(0)}$= 0, then equation (10) reduces to what is often referred to as the Jacobi or diagonal preconditioner. Based on the notation in equation (1), the Jacobi or diagonal preconditioner is expressed as:

$$\boldsymbol{M} = \boldsymbol{M_1} = \boldsymbol{D} \approx \boldsymbol{A} \qquad (11)$$

This was used in its simplest form and was constructed by taking the reciprocal of A_p in equation (7), which corresponds to the diagonal of $\boldsymbol{A}$. The diagonal preconditioning as described in equation (11) is only applied as a conditioner within the main iterative loop of the Krylov methods and not as a preconditioner. In the current study, a good initial guess of the pressure field from the previous time step already existed, and it was found that using equation (11) as a preconditioner corrupted the initial pressure field. As a result much better convergence was obtained without its explicit use as a preconditioner.

3.2 <u>Jacobi Iterative Preconditioner *(J)*</u>

For K>1 in equation (10), by using the recursion in k it can be shown that

$$\boldsymbol{x}^{(k)} = \boldsymbol{T}^k \boldsymbol{x}^{(0)} + \left\{\sum_{j=0}^{k-1} \boldsymbol{T}^j\right\} \boldsymbol{D}^{-1}\boldsymbol{b} \qquad (12)$$

where (k) is the kth iterate to the solution $\boldsymbol{x}$ with initial guess $\boldsymbol{x}^{(0)}$ and $\boldsymbol{T} = \boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})$. As the iteration number tends to infinity it can be shown that [43] if the spectral radius of $\boldsymbol{T} < 1$ then

$$\lim_{k \to \infty} \boldsymbol{T}^k \to 0 \quad ,$$

$$\lim_{k \to \infty} \left\{ \sum_{j=0}^{k-1} \boldsymbol{T}^j \right\} \to (\boldsymbol{I} - \boldsymbol{T})^{-1} \tag{13}$$

Applying equation (13) to equation (12) as $k \to \infty$ we get

$$\boldsymbol{x}^{(k)} = \{\boldsymbol{I} - \boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})\}^{-1} \boldsymbol{D}^{-1}\boldsymbol{b} \tag{14}$$

Based on equation (1) as $k \to \infty$ the Jacobi iterations can be expressed as

$$\boldsymbol{M_1} = \boldsymbol{D}, \quad \boldsymbol{M_2} = \{\boldsymbol{I} - \boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})\} \tag{15}$$

This is a useful formalism in that it shows the Jacobi iterative preconditioner in terms of right and left preconditioners. $\boldsymbol{M_1}^{-1}$ is similar to diagonal preconditioning and $\boldsymbol{M_2}^{-1}$ can be approximated by equation (2), which is the same as polynomial preconditioning. In this respect, the Jacobi iterations can be thought of as a combination of diagonal and polynomial preconditioning. However, in reality when it is used as a preconditioner, K does not tend to infinity and equation (15) is not really valid. By numerical experiments, in methods which explicitly required $\boldsymbol{M_1}^{-1}$ and $\boldsymbol{M_2}^{-1}$, the best efficiency was obtained when $\boldsymbol{M_1} = \boldsymbol{I}$ and $\boldsymbol{M_2} = \boldsymbol{M}$ was used. The preconditioner $\boldsymbol{M}$ was applied as:

$$\begin{aligned} p^{k}_{(i,j,k)} = \{ & A_n\, p^{k-1}_{(i,j+1,k)} + A_s\, p^{k-1}_{(i,j-1,k)} \\ & + A_e\, p^{k-1}_{(i+1,j,k)} + A_w\, p^{k-1}_{(i-1,j,k)} \\ & + A_h\, p^{k-1}_{(i,j,k+1)} + A_l\, p^{k-1}_{(i,j,k-1)} \\ & - \frac{1}{\Delta T}\, \tilde{m}_{(i,j,k)} \} / (A_p) \qquad k = 1,2,3 \ldots\ldots K \end{aligned} \tag{16}$$

where k is the iteration number and K is the total number of sweeps.

The Jacobi iterative method is known for its efficient error reduction properties in the high wavenumbers and this is done fairly efficiently in the first few iterations after which its effectiveness starts deteriorating. This property can be used to great advantage in its use as a preconditioner. Both, for the diagonal preconditioning and the Jacobi methods, variants using under-relaxation factors can be used and in some cases may prove more effective. However, this was not tried in this study. In the text that follows, the notation "D" will be used for diagonal preconditioning and "J" for Jacobi iterative preconditioner implemented as described above at the end of the previous paragraph.

4. Implementation Details on the CM-5[1]

The methods are implemented on the CM-5 not as separate independent entities but to fit with the overall solution procedure of the Navier-Stokes solution algorithm. The features and implementation issues of the whole computer program are discussed in detail in Tafti [44]. Some of the important implementation details pertaining to the present methods will be outlined here.

[A] All three dimensional arrays defined as *(nz,nx,ny)* have a *(:serial, :news,:news)* layout to be compatible with finite-difference coefficients and other grid metric arrays which are two-dimensional *(nx,ny)* and have layouts *(:news,:news)*. All code blocks doing computations use do loops in the *:serial* z axis, i.e., computations are performed for each x and y plane.

In constructing the computer program [44], it is assumed that the z direction is always homogeneous and the geometry does not vary in that direction. This allows all the finite-difference coefficients to be two-dimensional which results in considerable savings in memory utilization for high-order approximations, which may require up to 200 to 300 coefficient arrays to be stored. However, since all the variable arrays are three-dimensional, for arithmetic operations to be compatible with the finite difference coefficients it is

[1] In this section k is used to denote the grid index in the z direction.

necessary to define the z direction as *serial*. This allows the use of the following programming strategy for arithmetic operations:

```
do k = k_b, k_e
rhs(k,:,:) = a(:,:) * u(k,:,:)  + b(:,:) * u(k+1,:,:) + ..........          (17)
.....
.....
enddo
```

where arrays a and b are two-dimensional finite difference coefficient arrays and u is a three-dimensional array. This code block is used whenever arithmetic operations are performed between two- and three-dimensional arrays. In the present context the finite-difference coefficients for the Laplacian in equation (6) are two-dimensional while the variable arrays are three-dimensional. Hence in the matrix-vector multiply and the preconditioning step, the computations are performed for each x-y plane similar to that shown in equation (17). This treatment has some extra overheads associated with it since the instruction set has to be initiated for each value of k. However, the overall savings in memory for the full Navier-Stokes algorithm far outweigh these considerations.

One side effect of this programming paradigm is that periodicity has to be explicitly applied in the spanwise or z direction because of its *:serial* nature. On the *:news* grid, the use of a circular shift implicitly assumes periodicity, however in a *:serial* dimension the application of periodicity requires buffer or overlap elements on either side. This is accomplished by defining the z dimension as *(k_b-3:k_e+3)*, where *k_b* to *k_e* are the active computational nodes and define the physical period. The three nodes on either side are a consequence of a seven point stencil used in high-order differencing in each direction for the full Navier-Stokes algorithm and periodicity is explicitly applied as follows:

```
p(k_b-1,:,:) = p(k_e,:,:)
```

$$
\begin{array}{l}
p(k\_b\text{-}2,:,:) = p(k\_e\text{-}1,:,:) \\
p(k\_b\text{-}3,:,:) = p(k\_e\text{-}2,:,:) \\
p(k\_e{+}1,:,:) = p(k\_b,:,:) \\
p(k\_e{+}2,:,:) = p(k\_b{+}1,:,:) \\
p(k\_e{+}3,:,:) = p(k\_b{+}2,:,:)
\end{array} \qquad (18)
$$

For the Krylov methods, periodicity in the *z* direction as given in equation (18) has to be explicitly applied to each of the vectors in the solution procedure, since the vectors are only calculated or updated from *k_b* to *k_e*. This is necessary because in the matrix-vector multiplications and the J preconditioning steps, values of the vector for $k<k\_b$ and $k>k\_e$ are needed.

It should be pointed out that all other scalar-vector and vector-vector operations encountered are performed on the three-dimensional array. An example of this would be:

$$p1(:,:,:) = a * v1(:,:,:) + b * z1(:,:,:) \qquad (19)$$

where *a*, and *b* are scalars and *p1* and *z1* are vectors.

[B] <u>All regular grid communications along *:news* axes (*x* and *y*) are done using *pshifts* and the shifted arrays are stored in three-dimensional arrays.</u>

Regular grid communications are the most prevalent form of inter-processor communications. They come up when the calculation pertaining to node *(i,j,k)* need values from neighboring nodes *(i+1,j,k)*, *(i,j+1,k)*, etc. All regular communications are performed using the intrinsic circular shift operator *cshift*. *Cshift* implicitly assumes periodicity and wraps around the shift operation. For example, the *i+1* value obtained through a *cshift* operation on node *i_end* will fetch the value from node *i_beg*. As a consequence of this, periodic boundary conditions are built into this operation and no special treatment is needed for applying periodicity. In the present implementation, the *cshift* operation is also used for boundary conditions which are not periodic. This is accomplished by using the finite-difference coefficients as arithmetic masks which are set to zero in the pre-processing stage when dictated by the boundary conditions.

The *cshift* operations in the *:news* directions *x* and *y* are used with the *pshift* construct, which performs multiple *cshifts*. An example of this usage in calculating a 4 point stencil in *x* and *y* for the array *p* is as follows:

```
   call pshift(4,id1,ier,cmssl_cshift,pip1,p,dim=2,shift=1,
  1                      cmssl_cshift,pim1,p,2,-1,
  1                      cmssl_cshift,pjp1,p,3,1,
  1                      cmssl_cshift,pjm1,p,3,-1)
```
(20)

where *dim=2* and *3* refer to the *x* and *y* directions, respectively.

In the Krylov methods, grid communications are encountered in the matrix-vector multiplications, the Jacobi preconditioner and the calculation of the residual vector. Another form of interprocessor communications is sum reductions encountered in the inner product of two vectors. The inner products are calculated using the CMSSL [35] library call

call gbl_gen_inner_product_noadd(prod,rho1,rho2,ier) (21)

where *rho1* and *rho2* are the vectors whose inner product is calculated and stored in *prod*.

Further, to comply with the framework of the whole Navier-Stokes solver additional detailed layout directives are used for the *:news* axes in the form of *:block* and *:procs* directives. The *:block* directive corresponds to a block of data local to a vector unit on a given axis and *:procs* corresponds to the number of vector units the axis is spread over. Its usage is as follows:

```
array3d(:serial, :block=nxb:proc=nxp,:block = nyb:proc=nyp)
array2d(:block=nxb:proc=nxp,:block = nyb:proc=nyp)
```
(22)

For more details on the most efficient utilization of these directives see [44].

The programs for CG, BiCG, CGS, BiCGSTAB and GMRES were implemented by essentially following the algorithms given in the references cited in Table 1. The number of matrix-vector multiplies, vector inner products, floating point operations on vectors and storage required are given in Table 2. The floating point operations given in Table 2 are not exact but are very good representations of the actual count. The residue *(e)* is calculated every 10 iterations for all methods except GMRES where it is calculated before each restart. Additionally in GMRES, operations involved in solving the Hessenberg matrix are not included in Table 2. These are found to be negligible. There are also additional arrays used outside of the main algorithm which are local to subroutines. At most about five to six other three-dimensional arrays are required for passing variables to subroutines and for storing shifted arrays. Additionally GMRES also requires two 2-dimensional matrices of size $(m+1,m+1)$ and four one-dimensional arrays of size $(m+1)$. The storage required for these are negligible compared to $m+6$ three dimensional arrays of order N.

We will make a few comments on the implementation of some of the methods. For GMRES, the restart version from reference [11] is used as the basic algorithm. The upper Hessenberg matrix is formed from orthonormal vector basis which are computed using a modified Gram-Schmidth process [45]. The solution of the least square minimization problem involving the Hessenberg matrix is carried out using the standard fast Given's rotation [46]. This step is carried out as soon as a new column is added to the Hessenberg matrix (i.e. at every iterative step in the restart loop when a new vector of the orthonormal basis is formed). Once the rotation is applied to the Hessenberg matrix, the last element of the rotated matrix gives the residual norm of the approximate solution. However, in the present implementation, the new approximate solution is computed from the solution of the above least square problem at the end of the restart loop and compared with the convergence criterion. So convergence was always achieved at the end of computing all the vectors up to the restart value. In the current implementation, the Given's rotation was done sequentially on the partition manager.

The preconditioned version of BiCGSTAB is implemented based on Bi-CGSTAB-P given in [8]. Also the modifications to the original algorithm suggested by [8, p. 637] are incorporated. In the implementation of BiCG, CMSSL_QMRLAL, and CMSSL_QMR2, the matrix transpose operations (matrix^T-vector multiply and preconditioning with $\boldsymbol{M}^T$) are applied by using the relationships in equation (8) and applying them to the respective operator.

5. Details of Calculations Performed

These methods were tested for data obtained from (a) a turbulent channel flow simulation [44], (b) a time-dependent calculation of a three-dimensional lid driven cavity. The Navier-Stokes solver was used to generate the necessary information at a given time step, which was then used by the Krylov methods to calculate the updated pressure field by solving equation (6). The turbulent channel flow simulations were performed at a wall Reynolds number of Re_t = 180. More details can be found in [44]. The initial conditions were used from Moin and Kim [47]. They consist of perturbations applied to the mean turbulent flow profile. The pressure field and the mass source term are obtained after approximately 0.1 time units of integration. For the driven cavity problem, the lid is set in motion with $u = 1$ and $w = 1$ at T=0 making the flow field three-dimensional with a Reynolds number Re=1000. The pressure field and the mass source term are obtained after approximately 0.1 time units of integration. Table 3 tabulates the different cases tested. The maximum and minimum values of the grid spacing give an indication of the grid stretching and also the maximum cell aspect ratios. Cases A, C and E are symmetric while cases B, D, and F are nonsymmetric with the degree of nonsymmetry being the largest for case D. The convergence criterion for all the calculations was set to ε = 1.E-8, where ε is defined in Table 3. The 64^3 cases were run on a 32 node partition of the CM5 (128 Vector Units). The detailed layout for these cases was as follows (see equation (22)) *nxb = 4, nxp = 16, nyb = 8, nyp = 8*. Cases E and F were run on a 64 node partition of the CM5 with *nxb = 16, nxp = 16, nyb = 16, nyp = 6*.

6. Results

The different runs made for Cases A through D are tabulated in Tables 4 through 7, repectively. The *cm elapsed time* was calculated by using cm timers[2]. For the symmetric cases it was found that CG performed the best in terms of CPU time. None of the other methods were competitive with it, hence for Cases A, C and E, only the runs with CG are shown. In general, all the other methods were 1.5 to 2 times more expensive for symmetric systems.

A number of calculations were done to study the effect of preconditioning on the data sets A to D. For the preconditioner J, the number of sweeps K was varied to see its effect on the convergence and the CPU time to convergence. The

[2]None of the runs were in the dedicated mode of operation and the dedicated timings could be as much as 10 to 15 percent less than those reported.

number of sweeps K was varied from two to five. Figure (2) shows the convergence characteristics for representative runs. Figure 2(a) is for the symmetric case runs A1_0 through A1_D in Table 4 and figure 2(b) corresponds to the nonsymmetric runs B4_0 through B4_D in Table 5. From figure (2) and the other runs which were made, the following conclusions were reached:

(A) For symmetric positive definite systems, the CG method by itself is the most efficient. Referring to figure 2(a) and runs C1_ in Table 6, although preconditioner J decreases the iterations substantially, the amount of work done is more and the CPU time increases. For each sweep of the preconditioner J, communication operations of the type in equation (20) have to be initiated and the overheads associated with them are quite high. On the other hand the overheads associated with the D preconditioner are quite small. In spite of this for the symmetric cases tested, the D preconditioner does not decrease the number of iterations for convergence by a great degree. This result is similar to that in Ref. 17.

(B) For the nonsymmetric cases, preconditioner J is quite effective. For most of the methods tested its use reduces the CPU time by a factor 1.5 to 2. It is particularly effective in conjunction with GMRES(m), where in some cases the CPU time is reduced by as much as a factor of 3. The D preconditioner is more effective than it is for symmetric systems and in spite of its simplicity performs remarkably well and compares quite favorably with the J preconditioner in terms of CPU time. However, it fails to converge and becomes unstable with the CGS method for case B (run B1_D).

(C) A curious phenomenon which was found with the J preconditioner was its bias towards even number of sweeps. This trend can be clearly seen in Figure 2 and the other runs made for symmetric and nonsymmetric systems. For odd number of sweeps, its effectiveness decreases (increase in the number of iterations for convergence) and for some cases even becomes unstable (runs B1_J1, B1_J3) . This can be explained by studying the eigen value characteristics of an equivalent one-dimensional preconditioned system on a uniform grid.

On a uniform grid the one-dimensional equivalent of equation (7) can be represented by A_p = 1.0, A_e = -0.5 and A_w = -0.5. The system matrix $\boldsymbol{A}$ can be written as $\boldsymbol{A} = \boldsymbol{I} - \boldsymbol{G}$, where $\boldsymbol{G}$ contains the off-diagonal ($\boldsymbol{L} + \boldsymbol{U}$) elements. Based on equation(15) and equation (2), the Jacobi preconditioner $\boldsymbol{M}_K^{-1}$ can be represented by $(\boldsymbol{I} + \boldsymbol{G} + \boldsymbol{G}^2 + ... + \boldsymbol{G}^{K-1})$, where K is the total number of sweeps or iterations. Consequently, the preconditioned system $\boldsymbol{M}_K^{-1}\boldsymbol{A}$ can be written as $\boldsymbol{M}_K^{-1}\boldsymbol{A} = \boldsymbol{I} - \boldsymbol{G}^K$ Since the eigen values of $\boldsymbol{G}^K$lie between (-1,1) for odd k and (0,1) for even K, the preconditioned system $\boldsymbol{M}_K^{-1}\boldsymbol{A}$ will have a smaller spread of eigen values for even K. In other words, the condition number of the preconditioned system does not show a monotonic decrease with K but rather an oscillating trend, in which it increases for odd K over the previous even K. This leads to much better spectral properties of the preconditioned system for even number of sweeps, which result in fewer iterations for convergence than for odd number of sweeps.

(D) For the J preconditioner, K = 2 is the most cost effective for all the runs made. Although the increase in the number of even sweeps (K = 2,4,6...) reduces the number of iterations for convergence, the increase in work done in applying the preconditioner has a larger effect.

(E) Overall J2 is better than the D preconditioner in spite of the higher overheads associated with the former.

Next we compare the convergence characteristics and CPU times of the CGS, BiCG and BiCGSTAB and GMRES*(m)* methods. Figure 3(a) and (b) show the convergence characteristics for Case B without and with the J2 preconditioner, respectively. Figure 4(a) and (b) show similar results for Case D. The following observations can be made:

(A) Convergence of CGS is extremely erratic with large fluctuations in the residue. These fluctuations are considerably damped in BiCG and BiCGSTAB.

(B) Preconditioner J2 is quite effective and reduces the computational time by a factor of about 1.6 to 2.0 for CGS, BiCG and BiCGSTAB. For GMRES*(m)*, the reduction in computational time is by a factor of about 3.0.

(C) BiCGSTAB and GMRES*(m)* with J2 preconditioning show the best balance between stability, convergence and CPU requirements.

Runs B4_J2_m and D4_ J2_m show the effect of the the number of vectors stored in GMRES*(m)* with the J2 preconditioner on the CPU time and number of iterations required for convergence. Figure (5) shows the results for runs B4_J2_m. As expected the number of restarts necessary to obtain convergence decreases as the number of vectors stored increase. Also a relatively small orthonormal basis (m=20 to 30) seems to be sufficient to reduce the number of restarts to less than 10. This is consistent with the observation that for a matrix which is positive definite, the value of m need not be very large [11].

Next we compare the methods which were programmed by the authors to the CMSSL routines. The objective for this exercise was to compare the hand coded versions of CG, BiCG, CGS, BiCGSTAB, and GMRES with their CMSSL counterparts and also to investigate the QMR methods in CMSSL (see Table 1). Two runs are made for each method, one with no preconditioning and the other with J2 preconditioning. These are also summarized in Tables 4-7.

Comparing the performance of CG, BiCG, CGS with their CMSSL counterparts, no major differences were observed in the convergence characteristics nor the CPU requirements. However, noticeable differences are observed in the behavior of BiCGSTAB and GMRES. CMSSL_BiCGSTAB failed to converge for Cases B and D and became unstable[3]. On the other hand CMSSL_BiCGSTAB2 was found to be quite similar in its behavior to BiCGSTAB. Major differences in convergence characteristics were observed in CMSSL_GMRES. When no preconditioning was used, the CMSSL version took many more iterations to converge and about twice as much CPU time. However, with the preconditioner J2, this difference was considerably attenuated as can be seen in Table 5 and 7 between runs B4_ and B9_ and D4_ and D9_, respectively. Figure 6(a) and (b) show the convergence for runs B4_ and B9_ . In figure 6(a), the rates of convergence are the same for the two cases, except that in the CMSSL version the residual norm is elevated at the end of the first iterate or restart. This behavior, although present in figure 6(b), has a lesser effect on the CPU time for convergence. Similar trends are found in runs D4_ and D9_.

[3]Reported to CMSSL Staff at Thinking Machines Corporation in July 1994.

Figure 7(a) to (d) show the convergence behavior for the CMSSL routines CGS, QMRS, QCGS and QMRCGS for case D with and without preconditioning. It is seen that CGS [7], QMRS [14] and QCGS [39] show very similar convergence patterns. In fact, the residues obtained from CGS, QCGS and QMRS are exactly the same. The reason for this behavior is not known[4]. Between the three, CGS takes the least amount of CPU time. Preconditioning with J2 reduces the CPU time by an approximate factor of two. On the other hand, QMRCGS [15] exhibits much smoother convergence than CGS but takes about the same CPU time. In all cases, the J2 preconditioner reduces the CPU time by an approximate factor of two.

Figures 8(a) to (d) compare the convergence characteristics of CMSSL routines BiCG, BiCGSTAB2, QMRBiCGSTAB and QMRBiCGSTAB2 for Case D with and without preconditioning. Convergence of BiCGSTAB2 is considerably smoother than BiCG. Although BiCGSTAB2 also reduces the iterations to convergence, its takes about the same CPU time as BiCG. Both, QMRBiCGSTAB and QMRBiCGSTAB2 [16] smooth out the oscillations seen in BiCGSTAB2. However, the number of iterations are approximately the same between the three methods. J2 preconditioning reduces the CPU time by a factor of approximately 1.5. QMRBiCGSTAB and QMRBiCGSTAB2 with J2 preconditioner take the least amount of CPU time for convergence. Between the two, QMRBiCGSTAB2 seems to be slightly more efficient. Although not shown, QMR2 [38] exhibits the same convergence behavior as QMRBiCGSTAB but takes more iterations and CPU time to converge. There were some problems encountered with the use of

---

[4]Reported to CMSSL Staff at Thinking Machines Corporation in July 1994.

QMRLAL and the results are not shown[5]. The same trends as discussed in figures 7 and 8 are found for Case B.

Tables 8 and 9 tabulate the runs made for cases E and F with N = 3145728. For these cases only some of the CMSSL routines which showed good results for the previous cases were tested with and without the J2 preconditioner. Overall, the convergence trends are quite similar to the other cases. For the symmetric case E, CG without any preconditioning seems to perform the best. Both the programmed version of CG and the CMSSL routine show very similar results and convergence characteristics. For the nonsymmetric case F, BiCGSTAB, CMSSL_BiCGSTAB2, CMSSL_QMRBiCGSTAB and CMSSL_QMRBiCGSTAB2 show the best results in terms of CPU time. The use of the J2 preconditioner, reduces the CPU time by a factor of about two for CGS, and BiCG, while this factor is nearer to three for BiCGSTAB, BiCGSTAB2 and the QMR versions of BiCGSTAB. However, it is considerably higher for GMRES. GMRES(30) with J2 preconditioning is about eight times faster than without preconditioning, and for CMSSL_GMRES*(30)* the reduction in CPU time with the preconditioner J2 is by a factor of twelve. This is partly due to the fact that CMSSL_GMRES without any preconditioning took twice as much CPU time as GMRES. Similar differences, as highlighted in figure 5, between GMRES and CMSSL_GMRES are also encountered for case F.

Based on the operation counts given in Table 2, performance estimates for the methods programmed by the authors are given. As stated earlier, none of the runs are made in a dedicated mode, and the performance may vary. At the same time, the estimated MFLOPS give a good "feel" for how these methods perform.

---

[5]Reported to CMSSL Staff at Thinking Machines Corporation in July 1994.

It is observed that the J2 preconditioner, actually reduces the MFLOP rate. This is consistent with the fact that for each sweep of the J2 preconditioner, grid communication operations have to be initiated which have a large overhead associated with them. Most of the methods perform between 800-1000 MFLOPS on 64 processing nodes. It should also be mentioned that the coding strategy as suggested by equation (7) tends to be less efficient than operating on the three-dimensional array, but was used to comply with the data structure of the full Navier-Stokes algorithm.

7. Conclusions

Several Krylov based methods are tested for solving large systems on the CM-5. Case studies with both symmetric and nonsymmetric diagonally dominant system matrices are conducted. The order of the system varied from large (N = 262144) to very large (N = 3145728). The coefficient matrices and the source terms in the pressure equation are generated by the solution of the incompressible Navier-Stokes equations for a turbulent channel flow simulation and a three-dimensional flow in a driven cavity. The methods used to solve the pressure equation ranged from CG for symmetric systems, BiCG, CGS and BiCGSTAB, their Quasi Minimal Residual (QMR) versions, and GMRES for non-symmetric systems. These are outlined in Table 1. Two preconditioners, one based on matrix diagonalization and the other based on Jacobi iterations are tested. It is shown that Jacobi iterations are equivalent to a combination of diagonal and polynomial preconditioning.

It is found that for the diagonally dominant symmetric systems tested, CG without the two preconditioners tested performed the best. For the nonsymmetric cases tested, the QMR versions of BiCGSTAB [16] seemed to be the best performers in terms of CPU time. BiCGSTAB and BiCGSTAB2 were a close second. However, there were no major gains observed by the use of the QMR versions over the base methods in all cases. In general, the Jacobi iterative preconditioner was quite effective with these methods and reduced the CPU time by about a factor of 1.5-2.0 for systems with N = 262144. This factor was about 2.5 to 3.0 for the large system with N = 3145728. Although GMRES was not the most cost effective for the cases tested, it showed the smoothest convergence characteristics of all the methods and maintained monotonicity for all cases. The Jacobi iterative preconditioner was found to be particularly effective on GMRES*(30)* for N = 3145728, in which case it reduced the CPU time by a factor of 8. Performance studies on 64 nodes of the CM-5 gave an estimated performance between 800 and 1000 MFLOPS.

In conclusion, Krylov methods are quite attractive to solve for general sparse systems arising in computational fluid dynamics. They are readily parallelizable on massively parallel architectures and are quite efficient. Better preconditioners with high parallel efficiency would help in making these methods more cost effective.

Acknowledgments

The authors would like to thank Prof. Faisal Saied of the Computer Science Department at University of Illinois Urbana-Champaign for his help and useful suggestions during the course of this work.

References

1. L. Hageman and D. Young, *Applied Iterative Methods*, Academic Press, New York, 1981.

2. R. Freund, G. Golub, and N. Nachtigal, *Iterative Solution of Linear Systems*, Tech. Report NA-91-05, Stanford University, Stanford, CA, 1991.

3. J. Demmel, M. Heath, and H. Van Der Vorst, *Parallel Linear Algebra*, in Acta Numerica, Vol. 2, Cambridge Press, New York, 1993.

4. D. Keyes, and W. Gropp, *A Comparison of Domain Decomposition Techniques for Elliptic Partial Differential Equations and their Parallel Implementation*, SIAM J. Sci. Stat. Comput., Vol. 8, No. 2, pp. 166-202, March 1987.

5. M. Hestenes, and E. Stiefel, *Methods of Conjugate Gradients for Solving Linear Systems*, J. Res. Bur. Standards, Vol. 49, No. 6, pp. 409-436, Dec. 1952.

6. R. Fletcher, *Conjugate Gradient methods for Indefinite Systems*, Lecture Notes in Math., 506, 1976.

7. P. Sonneveld, CGS, *A Fast Lanczos-Type Solver for Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comput., Vol. 10, No. 1, pp. 36-52, January 1989.

8. H. Van Der Vorst, *BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comput., Vol. 13, No. 2, pp. 631-644, March 1992.

9. M. Gutknecht, *Variants of BiCGSTAB for Matrices with Complex Spectrum*, IPS Research Report No. 91-14, Interdisciplinary Project Center for Supercomputing. ETH-Zetrum, CH-8092, Zurich, 1991.

10. Hotler, W.H., Navon, I.M., and Oppe., T.C., *Parallelizable Preconditioned Conjugate Gradient methods for the Cray Y-MP and TMC CM-2*, Technical report, Supercomputer Computations Research Institute, Florida State University, December 1991.

11. Y. Saad, and M. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Non-Symmetric Linear Systems*, SIAM J. Sci. Comput., Vol. 7, No. 3, pp. 856-869, July 1986.

12. R. Freund, and N. Nachtigal, *QMR: A Quasi-Minimal Residual Method for Non-Hermitian Linear Systems*, Numer. Math. 60, pp. 315-339, 1991.

13. R. Freund, M. Gutknecht, and N. Nachtigal, *An Implementation of the Look-Ahead Lanczos Algorithm for Non-Hermitian Matrices, Part I and II*, Tech. Rep. 90.45-46, RIACS, NASA Ames Research Center, (1990).

14. R. Freund, and T. Szeto, *A Quasi-Minimal Residual Squared Algorithm for Non-Hermitian Linear Systems,* in Proceedings of the 1992 Copper Mountain Conference on Iterative Methods, 1992.

15. R. Freund, *A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems*, SIAM J. Sci. Comput., Vol. 14, pp. 470-482, 1993.

16. T. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. Tong, *QMRCGSTAB: A Quasi-Minimal Residual Variant of the BI-CGSTAB Algorithm for Non-Symmetric Systems*, CSRD Tech. Report 1231, University of Illinois at Urbana-Champaign, 1992.

17. G. Golub and J. Ortega, Scientific Computing and Differential Equations, An Introduction to Numerical Methods, Academic Press, Inc. 1992.

18. M. Kadioglu, and S. Mudrick, *On the Implementation of the GMRES(m) Method to Elliptic Equations in Meteorology*, J. Comp. Phys. 102, pp. 348-359, 1992.

19. V. Eijkhout, *Analysis of Parallel Incomplete Point Factorizations*, Linear Algebra Appl., Vol. 154-156, pp. 723-740, 1991.

20. J. Meijerink and H. Van Der Vorst, *Guidelines for the Usage of Incomplete Decompositions in Solving sets of Linear Equations As They Occur in Practical Problems,* J. Comput. Phys., 44, pp. 134-155, 1981.

21. H. Van Der Vorst, *High Performance Preconditioning*, SIAM J. Sci. Stat. Comput., Vol. 10, No. 6, pp.1174-1185, Nov. 1989.

22. R. Bramley, H-C. Chen, U. Meier, and A. Sameh, On Some Parallel Preconditioned CG Schemes, Eds. O. Axelsson and L. Y. Kolotilina, Vol. 1457 of Lecture Notes in Mathematics, Spinger-Verlag, pp. 17-27, 1990.

23. Y. Saad, *Practical Use of Polynomial Preconditionings for the Conjugate Gradient Method*, SIAM J. Sci. Statist. Comput., Vol. 6, pp. 865-881, 1985.

24. O. Johnson, C. Micchelli, and G. Paul, *Polynomial Preconditioners for Conjugate Gradient Calculations*, SIAM J. Numer. Anal. Vol. 20, No. 2, pp. 362-375, April 1983.

25. V. Venkatakrishnan, *Preconditioned Conjugate Gradient Methods for the Compressible Navier-Stokes Equations*, AIAA J., Vol. 29, No. 7, pp.1092-1100, July, 1991.

26. C. Vuik, *Solution of the Discretized Incompressible Navier-Stokes Equations with the GMRES Method*, Int. J. for Num. Methods in Fluids, Vol. 16, pp. 507-523, 1993.

27. S. Patankar, *Numerical Heat Transfer and Fluid Flow*, McGraw-Hill, New York, 1980.

28. F. Shakib, T. Hughes and Z. Johan, *A Multi-Element Group Preconditioned GMRES Algorithm for Nonsymmetric Systems Arising in Finite Element Analysis*, Computer Methods in Applied Mechanics and Engineering, Vol. 75, pp. 415-456, 1989.

29. Z. Johan, T. Hughes, K. Mathur, and S. Johnsson, *Data Parallel Finite Element Tecchniques for Computational Fluid Dynamics on the Connection Machine Systems*, Parallel Computational Fluid Dynamics' 92, Ed. R. Pelz, A. Ecer, and J. Hauser, Elsevier Science Publishers B. V., pp. 215-229, 1993.

30. X. Xu, N. Qin, and B. Richards, *Parallelising Explicit and Fully Implicit Navier-Stokes Solutions for Compressible Flows*, Parallel Computational Fluid Dynamics' 92, Ed. R. Pelz, A. Ecer, and J. Hauser, Elsevier Science Publishers B. V., pp. 407-418, 1993.

31. P. Chin, E. D'Azevedo, P. Forsyth, and W-P. Tang, *Preconditioned Conjugate Gradient Methods for the Incompressible Navier-Stokes Equations*, Int. J. Num. Methods in Fluids, Vol. 15, pp. 273-295, 1992.

32. H. Van Der Vorst , and P. Sonneveld, *CGSTAB, A More Smoothly Converging Variant of CGS,* Tech. Rep. 90-50, Delft University of Technology, 1990.

33. M. Baker, K. Bowler, and R. Kenway, *MIMD Implementations of Linear Solvers for Oil Reservoir Simulation*, Parallel Computing 16, pp. 313-334, 1990.

34. J. Meijerink, and H. van Der Vorst, *An iterative Solution Method for Linear Systems of Which The Coefficient Matrix is a Symmetric M-Matrix*, Math. Comp. 31, pp. 148-162, 1977.

35. *CMSSL for CM Fortran, CM-5 Edition, Version 3.1*, Thinking Machines Corporation, Cambridge, Massachusetts, June 1993.

36. G. Golub, and C. Van Loan, *Matrix Computations*. 2nd. ed., Baltimore: John Hopkins University Press, 1989.

37. R. Freund, M. Gutknecht, and N. Nachtigal, *An Implementation of the Look-ahead Lanczos Algorithm for Non-Hermitian Matrices*, SIAM J. Sci. Comput., Vol. 14, No. 1, pp. 137-158, January 1993.

38. R. Freund, and N. Nachtigal, *An Implementation of the QMR Method Based on Coupled Two-Term Recurrences*, RIACS Tech. Report 92.15, NASA Ames Research Center, Moffet Field, 1992.

39. C. Tong, *A Comparative Study of Preconditioned Lanczos Methods for Non-Symmetric Linear Systems*, Sandia Tech. Report SAND91-8240, 1992.

40. J. Kim and P. Moin, *Application of a Fractional-Step Method to Incompressible Navier-Stokes Equations*, J. Comput. Phys., vol. 59, pp. 308-323, 1985.

41. D. K. Tafti, *A Study of High-Order Spatial Finite Difference Formulations for the Incompressible Navier-Stokes Equations*, NCSA preprint 031, Sep. 1993.

42. F. Najjar and S. P. Vanka, *High-Order Formulation of the Pressure-Poisson Equation on Collocated Finite-Difference Grids,* Presented at ASME Fluids Engineering Summer Annual Meeting, Lake Tahoe, June 1994.

43. R. Burden, J. Faires, and A. Reynolds, *Numerical Analysis*, Prindle, Weber & Schmidt, August 1980.

44. D. Tafti, *Features and Implementation Issues for a High-Order Finite Difference Algorithm for the Direct and Large Eddy Simulation of Incompressible Turbulence on the CM-5*, NCSA Preprint 047, June 1994.

45. R. Barrett et al. *Templates For the Solution of Linear Systems: Building Blocks For Iterative Methods*.

46. D. Watkins, *Fundamentals of Matrix Computations*, John Wiley and Sons, 1991.

47. P. Moin, and J. Kim, *Numerical Investigation of Turbulent Channel Flow*, J. Fluid Mech., Vol. 118, pp. 341-377, 1982.

Table 1. Summary of Krylov based methods which were tested in the present study.

Method	Reference
CG	Hestenes and Stiefel [5]
BiCG	Fletcher [6]
CGS	Sonneveld [7]
BiCGSTAB	Van der Vorst [8]
GMRES	Saad and Shultz [11]
CMSSL_CG	Golub and Van Loan [36]
CMSSL_BiCG	Fletcher [6]
CMSSL_CGS	Sonneveld [7]
CMSSL_BiCGSTAB	Van der Vorst [8]
CMSSL_BiCGSTAB2	Gutknecht [9]
CMSSL_GMRES	Saad and Shultz [11]
CMSSL_QMRCGS	Freund [15]
CMSSL_QMRLAL	Freund et. al. [37] Freund and Nachtigal [12]
CMSSL_QMR2	Freund and Nachtigal [38]
CMSSL_QMRS	Freund and Szeto [14]
CMSSL_ QMRBiCGSTAB	Chan et al. [16]
CMSSL_ QMRBiCGSTAB2	Chan et al. [16]
CMSSL_QCGS	Tong [39]

Table 2. Floating point operations and storage associated with each method.

Method	Av $(13N)I_N$	A^Tv $(13N)I_N$	$v_1 . v_2$ $(2N-1)I_N$	$v_1=a^*v_2$ $(N)I_N$	precon-D $(N)I_N$	precon-J $(13KN)I_N$	ε $14(N)I_N$	Vectors stored I
CG	1	0	2	9	2	2	0.1	5
BiCG	1	1	2	15	2	2	0.1	9
CGS	2	0	2	19	2	2	0.1	9
BiCG STAB	2	0	4	16	2	2	0.1	9
GMRES(m)	m+1	0	S+m+1	3+5m+3S	(m+1)	(m+1)	1	m+6

m = number of vectors stored in GMRES

$$S = \sum_{i=1}^{m} i \;\; = \frac{1}{2}m(m+1)$$

k = number of sweeps in preconditioner J
N = order of system
I_N = iterations to convergence (For GMRES I_N is number of restarts)

Table 3. Different cases used for testing the Krylov Methods.

Case	Flow	B.Cs			Δx_{min}	Δy_{min}	Δz_{min}	Δx_{max}	Δy_{max}	Δz_{max}	ε_0	U eqn(9)
		x	y	z								
A	Channel 64x64x64	P	N	P	0.1963	0.0323	0.0982	0.1963	0.0323	0.0982	5.5E-3	0
B	Channel 64x64x64	P	N	P	0.1963	0.0120	0.0982	0.1963	0.0844	0.0982	4.4E-3	0.9259
C	Cavity 64x64x64	N	N	P	0.0161	0.0161	0.0982	0.0161	0.0161	0.0982	1.3E-4	0
D	Cavity 64x64x64	N	N	P	0.0041	0.0041	0.0982	0.0478	0.0478	0.0982	4.0E-4	7.626
E	Channel 256x96x128	P	N	P	0.0491	0.0213	0.0491	0.0491	0.0213	0.0491	4.5E-4	0
F	Channel 256x96x128	P	N	P	0.0491	0.0033	0.0491	0.0491	0.0760	0.0491	3.3E-4	4.032

P - periodic boundary conditions or homogeneous direction
N = Neumann boundary conditions or inhomogeneous direction.

$\varepsilon = \sqrt{\frac{1}{N} \sum_{N} (\mathbf{Ax}-\mathbf{b})^2}$ *, e_0 is initial residue.*

U - degree of nonsymmetry, defined in equation (9)

Table 4. Runs made with Case A

Run	Method	Precon	K	I_N	cm elapsed time(secs)
A1_0	CG	-	-	270	7.2
A1_J2	"	J	2	130	8.6
A1_J3	"	J	3	150	12.8
A1_J4	"	J	4	100	10.4
A1_J5	"	J	5	120	14.9
A1_D	"	D	-	250	9.3
A2_0	CMSSL_CG	-	-	262	6.9
A2_J2	"	J	2	129	8.9

Table 5. Runs made with Case B

Run	Method	Precon	K	I_N	cm elapsed time(secs)
B1_0	CGS	-	-	740	35.1
B1_J2	"	J	2	170	20.9
B1_J3	"	J	3	NC	-
B1_J4	"	J	4	130	25.8
B1_J5	"	J	5	NC	-
B1_D	"	D	-	NC	-
B2_0	BiCG	-	-	760	37.2
B2_J2	"	J	2	230	29.0
B2_J3	"	J	3	250	41.2
B2_J4	"	J	4	150	30.0
B2_J5	"	J	5	190	45.2
B2_D	"	D	-	430	30.7
B3_0	BiCGSTAB	-	-	680	34.4
B3_J2	"	J	2	140	17.6
B3_J3	"	J	3	150	28.2
B3_J4	"	J	4	90	18.7
B3_J5	"	J	5	130	30.8
B3_D	"	D	-	300	18.8
B4_0_30	GMRES(30)	-	-	38	68.0
B4_J2_30	"	J	2	8	23.2
B4_J3_30	"	J	3	10	35.3
B4_J4_30	"	J	4	6	24.5
B4_J5_30	"	J	5	8	37.6
B4_D_30	"	D	-	17	33.5
B4_J2_10	GMRES(10)	J	2	30	24.5
B4_J2_20	GMRES(20)	J	2	13	23.0
B4_J2_40	GMRES(40)	J	2	6	25.8
B4_J2_50	GMRES(50)	J	2	5	30.2
B4_J2_60	GMRES(60)	J	2	4	33.8
B5_0	CMSSL_CGS	-	-	563	28.6
B5_J2	"	J	2	147	19.1
B6_0	CMSSL_BiCG	-	-	755	38.5
B6_J2	"	J	2	211	28.3
B7_0	CMSSL_BiCGSTAB	-	-	NC	-
B7_J2	"	J	2	NC	-
B8_0	CMSSL_BiCGSTAB2	-	-	500	38.8
B8_J2	"	J	2	118	23.6
B9_0_30	CMSSL_GMRES(30)	-	-	53	112
B9_J2_30	"	J	2	10	31.0
B10_0	CMSSL_QMRCGS	-	-	563	33.3
B10_J2	"	J	2	150	21.4
B11_0	CMSSL_QMRLAL	-	-	NC	-
B11_J2	"	J	2	NC	-
B12_0	CMSSL_QMR2	-	-	736	43.9
B12_J2	"	J	2	208	38.5
B13_0	CMSSL_QMRS	-	-	740	57.4

B13_J2	"	J	2	170	33.2

Table 5. Runs made with Case B (continued).

B14_0	CMSSL_QMR BiCGSTAB	-	-	566	35.1
B14_J2	"	J	2	139	19.97
B15_0	CMSSL_QMR BiCGSTAB2	-	-	551	34.3
B15_J2	"	J	2	120	17.1
B16_0	CMSSL_QCGS	-	-	740	44.8
B16_J2	"	J	2	170	24.7

NC - not converged in 1000 iterations

Table 6. Runs made with Case C

Run	Method	Precon	K	I_N	cm elapsed time(secs)
C1_0	CG	-	-	170	4.4
C1_J2	"	J	2	120	7.9
C1_J3	"	J	3	100	8.5
C1_J4	"	J	4	90	9.4
C1_J5	"	J	5	80	9.8
C1_D	"	D	-	170	8.6
C2_0	CMSSL_CG	-	-	170	4.6
C2_J2	"	J	2	137	9.4

Table 7. Runs made with Case D

Run	Method	Precon	K	I_N	cm elapsed time(secs)
D1_0	CGS	-	-	650	31.1
D1_J2	"	J	2	150	19.0
D1_J3	"	J	3	170	27.1
D1_J4	"	J	4	100	20.1
D1_J5	"	J	5	130	30.8
D1_D	"	D	-	320	19.4
D2_0	BiCG	-	-	600	32.2
D2_J2	"	J	2	160	19.9
D2_J3	"	J	3	190	30.8
D2_J4	"	J	4	110	22.0
D2_J5	"	J	5	150	35.6
D2_D	"	D	-	340	20.8
D3_0	BiCGSTAB	-	-	420	20.4
D3_J2	"	J	2	90	11.4
D3_J3	"	J	3	120	19.8
D3_J4	"	J	4	80	16.1
D3_J5	"	J	5	100	23.9
D3_D	"	D	-	210	13.2
D4_0_30	GMRES(30)	-	-	36	62.2
D4_J2_30	"	J	2	7	20.5
D4_J3_30	"	J	3	8	28.2
D4_J4_30	"	J	4	4	17.0
D4_J5_30	"	J	5	6	28.7
D4_D_30	"	D	-	19	36.3
D4_J2_10	GMRES(10)	J	2	26	21.7
D4_J2_20	GMRES(20)	J	2	12	21.6
D4_J2_40	GMRES(40)	J	2	4	17.5
D4_J2_50	GMRES(50)	J	2	4	27.0
D4_J2_60	GMRES(60)	J	2	3	25.4
D5_0	CMSSL_CGS	-	-	605	31.4
D5_J2	"	J	2	114	15.2
D6_0	CMSSL_BiCG	-	-	585	30.1
D6_J2	"	J	2	159	21.2
D7_0	CMSSL_BiCGSTAB	-	-	NC	-
D7_J2	"	J	2	NC	-
D8_0	CMSSL_BiCGSTAB2	-	-	428	34.2
D8_J2	"	J	2	100	20.0
D9_0_30	CMSSL_GMRES(30)	-	-	67	145.8
D9_J2_30	"	J	2	7	20.9
D10_0	CMSSL_QMRCGS	-	-	624	36.9
D10_J2	"	J	2	114	17.1
D11_0	CMSSL_QMRLAL	-	-	NC	-
D11_J2	"	J	2	NC	-
D12_0	CMSSL_QMR2	-	-	585	34.3
D12_J2	"	J	2	145	26.9
D13_0	CMSSL_QMRS	-	-	610	47.4

D13_J2	"	J	2	132	26.2

Table 7. Runs made with Case D (continued).

D14_0	CMSSL_QMR BiCGSTAB	-	-	419	24.8
D14_J2	"	J	2	100	14.5
D15_0	CMSSL_QMR BiCGSTAB2	-	-	426	31.8
D15_J2	"	J	2	100	14.5
D16_0	CMSSL_QCGS	-	-	610	56.1
D16_J2	"	J	2	132	19.7

NC - not converged in 1000 iterations

Table 8. Runs made with Case E

Run	Method	Precon	K	I_N	cm elapsed time(secs)	Approx. MFLOPS
E1_0	CG	-	-	360	35.2	900
E1_J2	"	J	2	180	47.4	950
E2_0	CMSSL_CG	-	-	355	36.9	-
E2_J2	"	J	2	174	43.4	-

Table 9. Runs made with Case F

Run	Method	Precon	K	I_N	cm elapsed time(secs)	Approx. MFLOPS
F1_0	CGS	-	-	2310	422.8	900
F1_J2	"	J	2	NC[1] (430)	189.3[2]	750
F2_0	BiCG	-	-	2050	378.0	800
F2_J2	"	J	2	390	178.1	700
F3_0	BiCGSTAB	-	-	1890	344.0	900
F3_J2	"	J	2	280	130.0	700
F4_0_30	GMRES(30)	-	-	222	1676.2	1200
F4_J2_30	"	J	2	18	213.4	1000
F5_0	CMSSL_CGS	-	-	2098	389.4	-
F5_J2	"	J	2	414	200.9	-
F6_0	CMSSL_BiCGSTAB2	-	-	1629	465.8	-
F6_J2	"	J	2	241	176.7	-
F7_0_30	CMSSL_GMRES(30)	-	-	331	3283.3	-
F7_J2_30	"	J	2	19	267.9	-
F8_0	CMSSL_QMRCGS	-	-	NC[3]	474.2[4]	-
F8_J2	"	J	2	400	216.3	-
F9_0	CMSSL_QMR BiCGSTAB	-	-	2032	477.8	-
F9_J2	"	J	2	233	125.4	-
F10_0	CMSSL_QMR BiCGSTAB2	-	-	1625	360.3	-
F10_J2	"	J	2	252	131.9	-

NC - not converged in 3000 iterations.

1 - *converges to 1.5E-8 after 430 iterations and then starts diverging.*

2 - *Time for 430 iterations.*

3 - *residue becomes stationary at about1.8E-8 after 2123 iterations.*

4 - *Time for 2123 iterations.*

Figure 1: Staggered grid topology. The pressure equation is solved on the scalar node (p).

Figure 2: Effect of number of sweeps with the Jacobi iterative preconditioner (J) and diagonal preconditioner (D); (a) CG for symmetric system case A; (b) GMRES(30) for nonsymmetric system B.

Figure 3: Convergence characteristics for nonsymmetric system case B; (a) no preconditioning; (b) with Jacobi iterative preconditioning (J2).

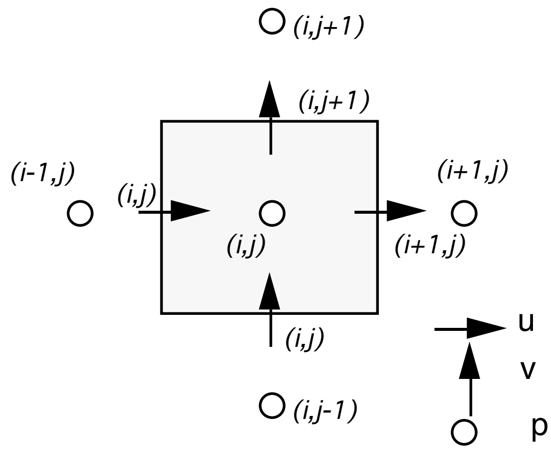
Figure 4: Convergence characteristics for nonsymmetric system case D; (a) no preconditioning; (b) with Jacobi iterative preconditioning (J2).

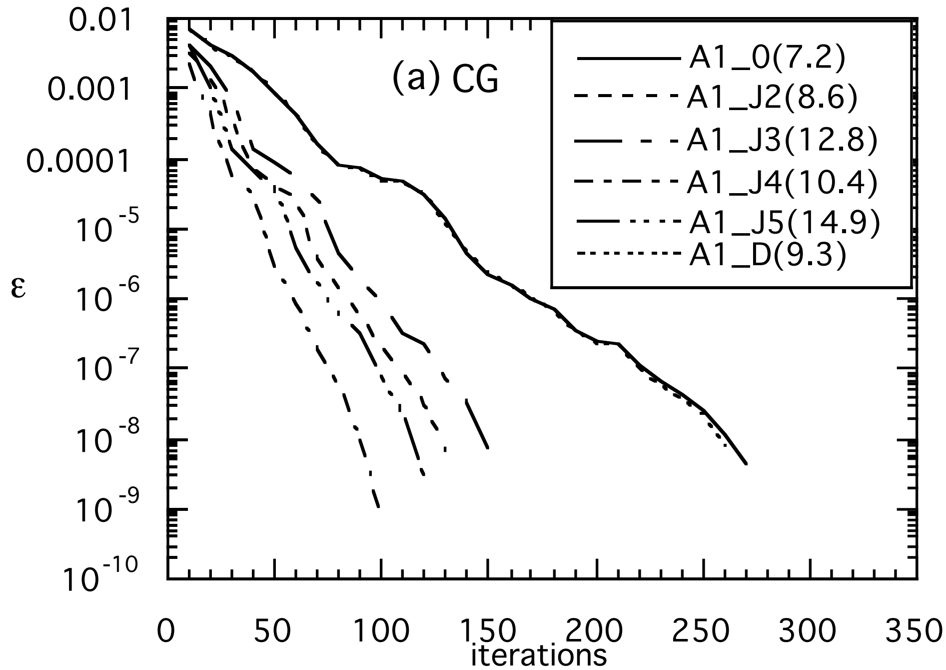
Figure 5: Effect of number of vectors stored on the convergence of GMRES(m) for nonsymmetric case B with Jacobi iterative preconditioner (J2).

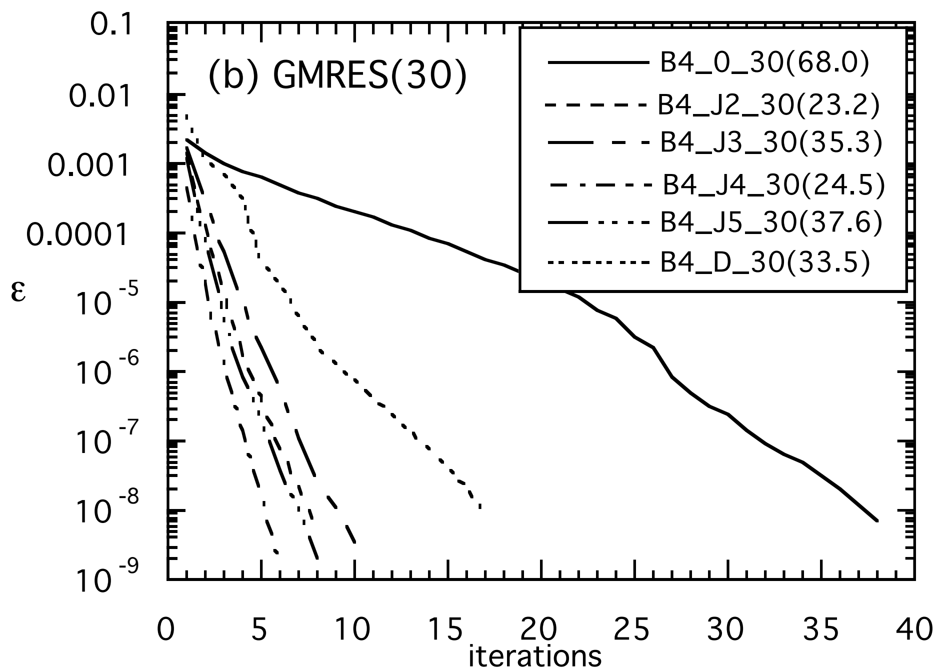
Figure 6: Comparison of GMRES(30) with CMSSL_GMRES(30) for nonsymmetric case B; (a) no preconditioning; (b) with Jacobi iterative preconditioner (J2).

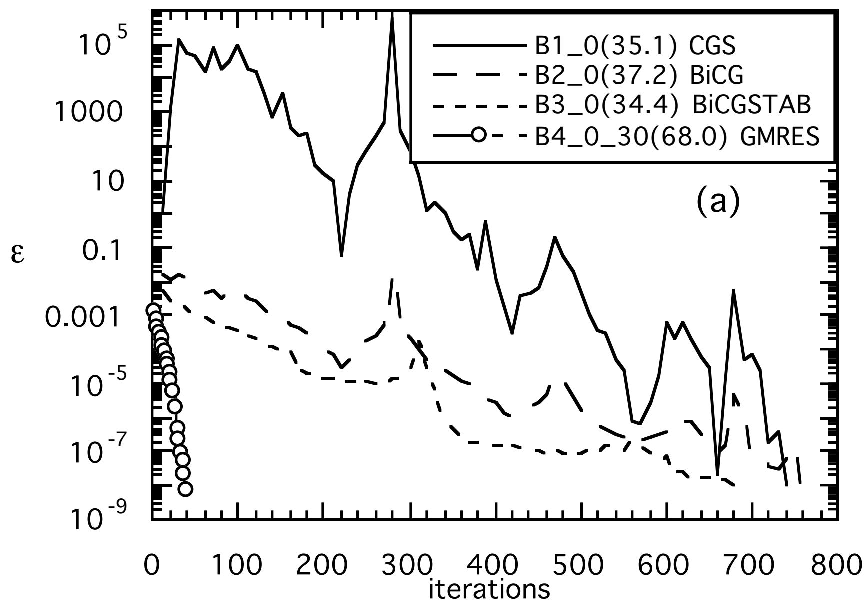
Figure 7: Convergence characteristics for nonsymmetric case D with and without preconditioning(J2); (a) CMSSL_CGS; (b) CMSSL_QMRS; (c) CMSSL_QCGS; (d) CMSSL_QMRCGS.

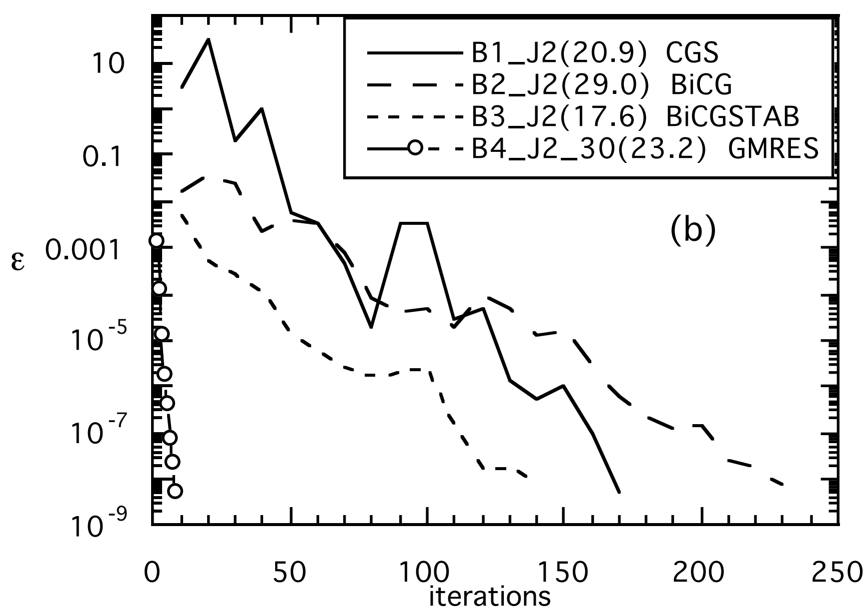
Figure 8: Convergence characteristics for nonsymmetric case D with and without preconditioning(J2); (a) CMSSL_BiCG; (b) CMSSL_BiCGSTAB2; (c) CMSSL_QMRBiCGSTAB; (d) CMSSL_QMRBiCGSTAB2.

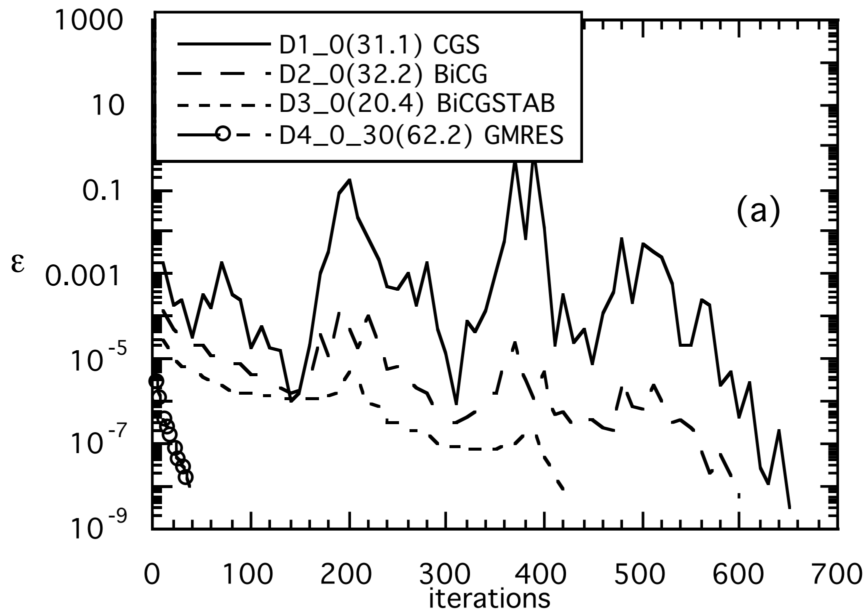

(i,j+1)
(i,j+1)
(i-1,j)
(i,j)
(i,j)
(i+1,j)
(i+1,j)
(i,j)
u
v
(i,j-1)
p


0.01
0.001
0.0001
10-5
10-6
10-7
10-8
10-9
10-10
ε
(a) CG
A1_0(7.2)
A1_J2(8.6)
A1_J3(12.8)
A1_J4(10.4)
A1_J5(14.9)
A1_D(9.3)
0
50
100
150
200
250
300
350
iterations

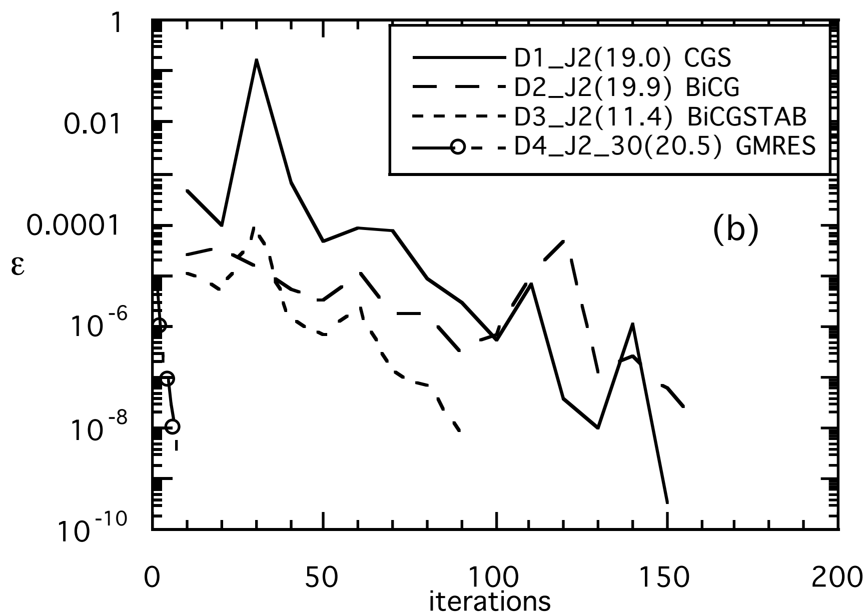

0.1
0.01
0.001
0.0001
10-5
10-6
10-7
10-8
10-9
ε
(b) GMRES(30)
B4_0_30(68.0)
B4_J2_30(23.2)
B4_J3_30(35.3)
B4_J4_30(24.5)
B4_J5_30(37.6)
B4_D_30(33.5)
0
5
10
15
20
25
30
35
40
iterations


B1_0(35.1) CGS
B2_0(37.2) BiCG
B3_0(34.4) BiCGSTAB
B4_0_30(68.0) GMRES
(a)
ε
iterations

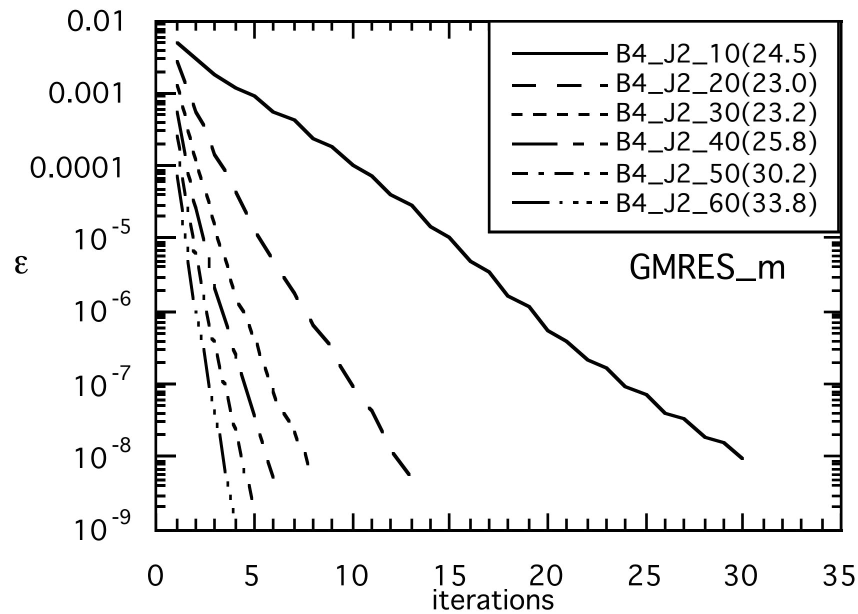

B1_J2(20.9) CGS
B2_J2(29.0) BiCG
B3_J2(17.6) BiCGSTAB
B4_J2_30(23.2) GMRES
(b)
ε
iterations



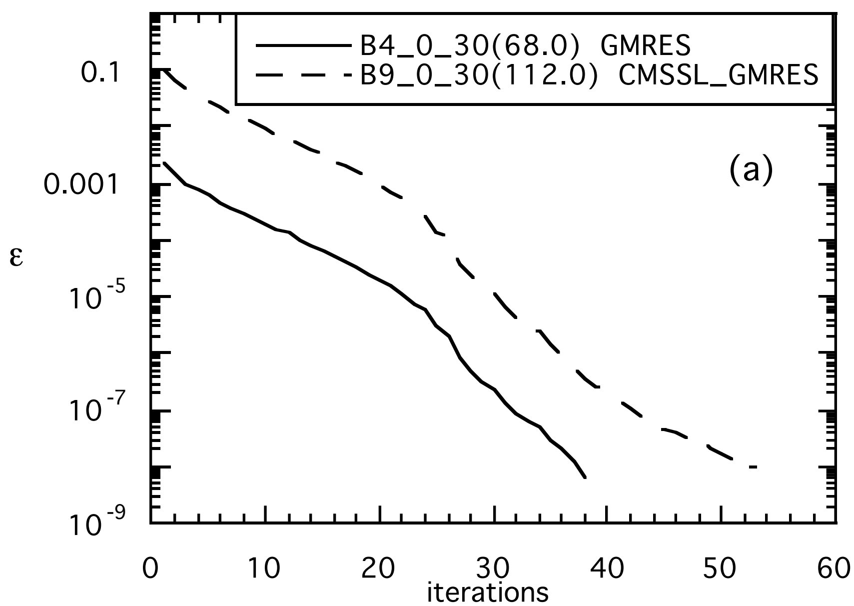
D1_0(31.1) CGS
D2_0(32.2) BiCG
D3_0(20.4) BiCGSTAB
D4_0_30(62.2) GMRES
(a)
ε
1000
10
0.1
0.001
10-5
10-7
10-9
0
100
200
300
400
500
600
700
iterations

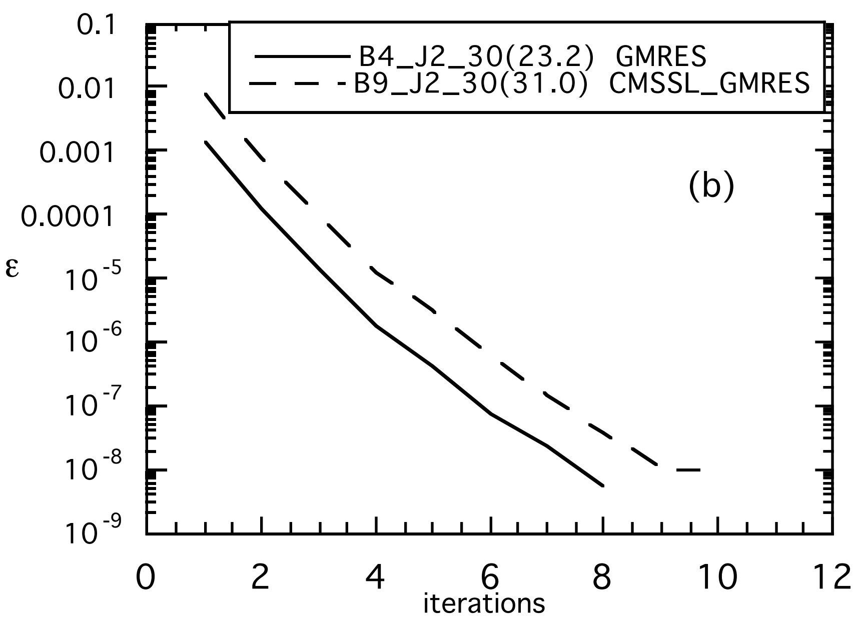


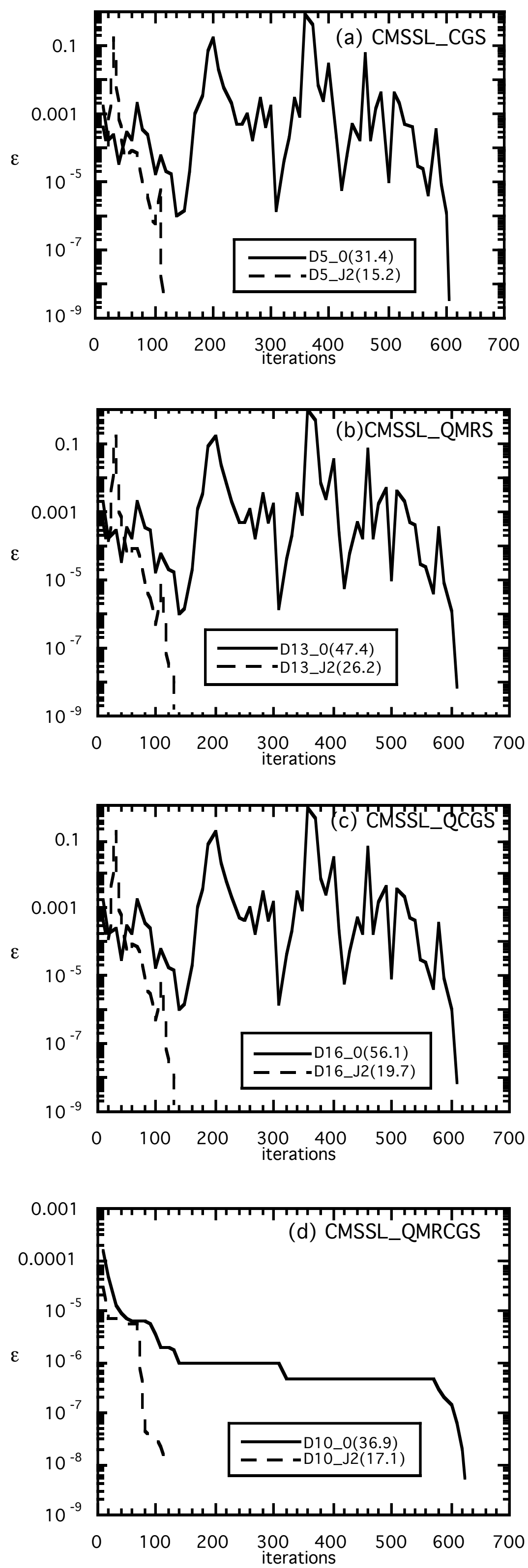
D1_J2(19.0) CGS
D2_J2(19.9) BiCG
D3_J2(11.4) BiCGSTAB
D4_J2_30(20.5) GMRES
(b)
ε
1
0.01
0.0001
10-6
10-8
10-10
0
50
100
150
200
iterations

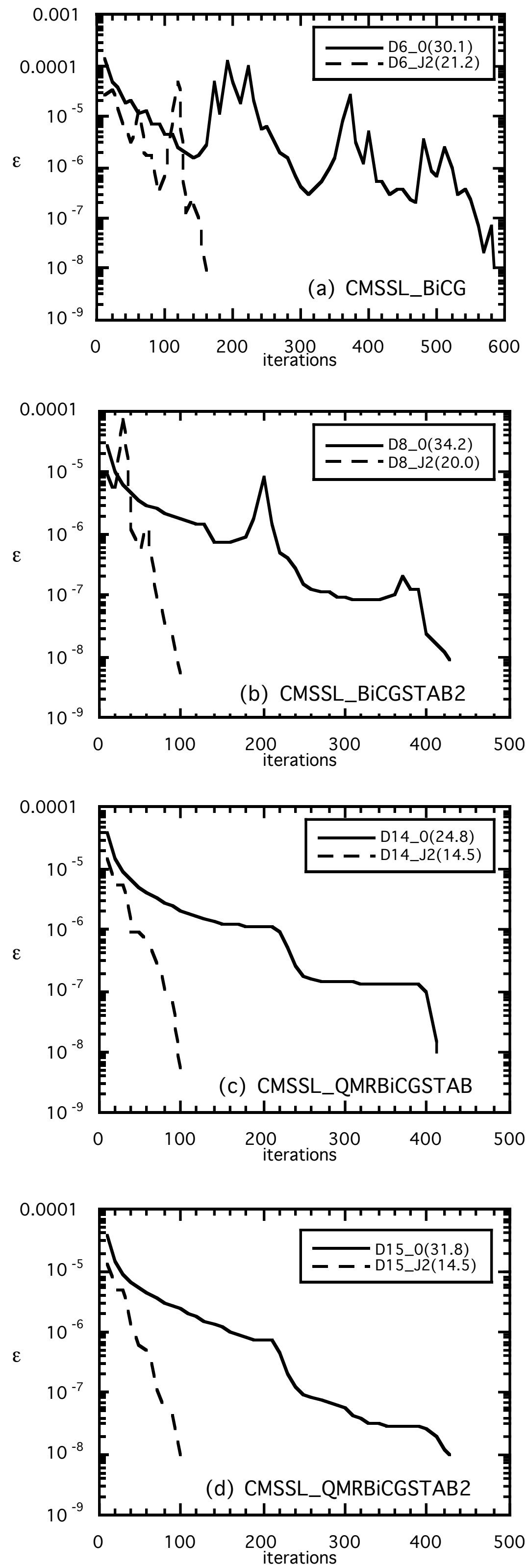


B4_J2_10(24.5)
B4_J2_20(23.0)
B4_J2_30(23.2)
B4_J2_40(25.8)
B4_J2_50(30.2)
B4_J2_60(33.8)
GMRES_m
ε
iterations
0.01
0.001
0.0001
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
0 5 10 15 20 25 30 35


B4_0_30(68.0) GMRES
B9_0_30(112.0) CMSSL_GMRES
(a)
0.1
0.001
ε
10^{-5}
10^{-7}
10^{-9}
0
10
20
30
40
50
60
iterations


B4_J2_30(23.2) GMRES
B9_J2_30(31.0) CMSSL_GMRES
(b)
0.1
0.01
0.001
0.0001
ε
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
0
2
4
6
8
10
12
iterations


(a) CMSSL_CGS
D5_0(31.4)
D5_J2(15.2)
(b)CMSSL_QMRS
D13_0(47.4)
D13_J2(26.2)
(c) CMSSL_QCGS
D16_0(56.1)
D16_J2(19.7)
(d) CMSSL_QMRCGS
D10_0(36.9)
D10_J2(17.1)
ε
iterations
0
100
200
300
400
500
600
700
0.1
0.001
10 -5
10 -7
10 -9
0.0001
10 -6
10 -8


0.001
0.0001
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
ε
D6_0(30.1)
D6_J2(21.2)
(a) CMSSL_BiCG
0 100 200 300 400 500 600
iterations
0.0001
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
ε
D8_0(34.2)
D8_J2(20.0)
(b) CMSSL_BiCGSTAB2
0 100 200 300 400 500
iterations
0.0001
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
ε
D14_0(24.8)
D14_J2(14.5)
(c) CMSSL_QMRBiCGSTAB
0 100 200 300 400 500
iterations
0.0001
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
ε
D15_0(31.8)
D15_J2(14.5)
(d) CMSSL_QMRBiCGSTAB2
0 100 200 300 400 500
iterations