Krylov Subspace Methods on Supercomputers

Youcef Saad

December, 1988

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.40

NASA Cooperative Agreement Number NCC 2-387

RIACS

Research Institute for Advanced Computer Science

Krylov Subspace Methods on Supercomputers

Youcef Saad

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.40
December, 1988

This paper presents a short survey of recent research on Krylov subspace methods with emphasis on implementation on vector and parallel computers. Conjugate gradient methods have proven very useful on traditional scalar computers, and their popularity is likely to increase as three dimensional models gain importance. A conservative approach to derive effective iterative techniques for supercomputers has been to find efficient parallel/ vector implementations of the standard algorithms. The main source of difficulty in the incomplete factorization preconditionings is in the solution of the triangular systems at each step. We describe in detail a few approaches consisting of implementing efficient forward and backward triangular solutions. Then we discuss polynomial preconditioning as an alternative to standard incomplete factorization techniques. Another efficient approach is to reorder the equations so as improve the structure of the matrix to achieve better parallelism or vectorization. We give an overview of these ideas and others and attempt to comment on their effectiveness or potential for different types of architectures.

Key words: Large linear systems; Krylov subspace methods; Iterative methods; Preconditioned Conjugate Gradient; Polynomial preconditioning; Incomplete LU preconditioning.

This paper is to appear in the *SIAM Journal on Statistical and Scientific Computing*.

Work reported herein was supported by the National Science Foundation under Grants No. US NSF-MIP-8410110 and US NSF DCR85-09970, the US Department of Energy under Grant No. DOE DE-FG02-85ER25001, and by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

1 Introduction

Scientific problems that are tackled today are perhaps a few orders of magnitude more complex than those dealt with just one decade ago. This trend is likely to accelerate because of the rapid improvement in computational power that will result from parallel processing. However, a characteristic of the current trend in supercomputing is that the easy gains in speed are over: big improvements are only possible by radical changes in architecture as well as software and algorithms.

In this paper we consider the impact of modern supercomputers on the design of iterative methods for solving large linear systems of equations. Because of the increased importance of three-dimensional models, iterative methods are again playing a major role. There is a general consensus that for problems arising from partial differential equations in three-dimensional domains, direct methods alone are too costly, both in terms of storage and computation. For example a $50 \times 50 \times 50$ grid with five degrees of freedom per grid point, such as Euler's equation in fluid dynamics, will lead to matrices of size N=625,000 and of bandwidth $m \approx 25000$. Even though reordering techniques can be used to exploit the sparsity of the matrix, the complexity of the problem is still very high. For a survey of the impact of supercomputing on aerodynamics research, where these types of problems arise, see Peterson [93]. The main attraction of iterative methods in an example like this one is their low storage requirement. In fact in many cases the matrix need not be stored entirely because it consists of a small number of blocks that are repeated many times.

Krylov subspace techniques, of which the Conjugate Gradient (CG) method is an example, have increasingly been viewed as general purpose iterative methods, especially since the discovery and popularization of preconditioning techniques [76]. Although these techniques may fail for matrices that are not M-matrices, they are effective for the large class of problems arising from partial differential equations of the elliptic type. For extensions and modifications of point incomplete factorizations see [73,121,83]. An important gap in the literature concerns the development of truly general purpose iterative solvers that could replace direct methods with minimum risk of failure.

It is interesting to observe that Krylov subspace methods became popular at almost the same time the first vector computers appeared in the marketplace in the mid-seventies. As a result, considerations for vector implementations were given early on [49,33,125,61,60]. The survey paper of Ortega and Voigt [91] gives an exhaustive bibliography for research done before 1985 in the general area of solution of partial differential equations on supercomputers. Much still remains to be done as many new techniques for vectorizing and parallelizing standard preconditioners were considered only recently. Along the way several alternative methods have emerged that turned out to be effective even on sequential computers. We will not describe these in this paper but we should mention two such techniques which show tremendous potential. The first is the class of domain decomposition methods. The reader is referred to the recent survey by T.F. Chan [26] and to the excellent paper on parallel implementations by W. Gropp and D. Keyes [64]. The attraction of domain decomposition methods is that they have been implemented naturally in many engineering applications in the past and as a result one may readily benefit from this experience.

The second class of methods that show promise is that of hierarchical multigrid techniques, in the context of finite element discretizations. Briefly, these techniques amount to using hierarchical basis functions, i.e., a basis that consists not only of the nodal functions at the finest grid, but also of the coarser basis functions from which those fine grid functions have been obtained. Thus the function space is identical but its basis has changed. The remarkable property shown by Yserentant [134] is that for two-dimensional problems, the coefficient matrix arising from the discretization of elliptic partial differential equations with such bases has a condition number of $O((-Log\ h)^2)$ instead of the usual $O(h^{-2})$. This has been exploited by A. Greenbaum et al. [51] and Adams and Ong [4] in the context of preconditioned conjugate gradient methods. The idea here is that there is no need to precondition such matrices, except possibly to use a simple diagonal scaling, since their condition numbers are so favorable.

In the brief overview presented in this paper we will mostly consider the problem of implementing Preconditioned Conjugate Gradient type methods for solving linear systems. The framework is that of large sparse linear systems that are not necessarily well structured. We will attempt to address the implementation issues for parallel machines with shared as well as distributed memory, and with small number of processors as well as large number of processors.

Before concluding this introduction we would like to comment that the volume of publications on iterative methods is so large that survey papers can no longer be exhaustive. For additional reading we recommend other survey papers by Ortega and Voigt [91], by Axelsson [13,14] and the recent book by Ortega [90].

The organization of the paper is as follows. In Section 2 we will give an overview of Krylov subspace methods without consideration to parallel implementations. Then we will address the questions of parallel / vector implementations of two specific examples of such methods: the conjugate gradient method for symmetric problems and GMRES for non-symmetric problems. The effective implementation of the forward and backward solves in preconditioned Krylov subspace methods is important enough that we will devote a separate section to it, namely Section 4. In Section 5 we will describe an alternative to incomplete factorization techniques, based on polynomial preconditionings, and give some comparisons with the standard approach. Then we will look at other preconditioners in Section 6, and reordering techniques in Section 7. We will make some concluding remarks in Section 8.

2 An overview of Krylov subspace methods

Given an initial guess x_0 to the linear system

$$Ax = b, \tag{1}$$

a general *projection method* seeks an approximate solution x_m from an affine subspace $x_0 + K_m$ of dimension m by imposing the Petrov-Galerkin condition

$$b - Ax_m \perp L_m \tag{2}$$

where L_m is another subspace of dimension m. A Krylov subspace method is a method for which the subspace K_m is the Krylov subspace

$$K_m(A, r_0) = span\{r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0\}, \tag{3}$$

in which $r_0 = b - Ax_0$. When there is no ambiguity we will denote $K_m(A, r_0)$ by K_m. The different versions of Krylov subspace methods arise from different choices of the subspaces K_m and L_m and from the ways in which the system is preconditioned. The most popular choices of K_m and L_m are the following.

1. $L_m = K_m = K_m(A, r_0)$. This is the orthogonal projection or Galerkin case. The conjugate gradient method is a particular instance of this method when the matrix is symmetric positive definite. Another method in this class is the Full Orthogonalization Method (FOM) [102] which is closely related to Arnoldi's method for solving eigenvalue problems [7]. Also in this class is ORTHORES [57], a method that is mathematically equivalent to FOM. Axelsson [10] also derived an algorithm of this class for general nonsymmetric matrices.

2. $L_m = AK_m; K_m = K_m(A, r_0)$. With this choice of L_m, it can be shown, see e.g., [108] that the approximate solution x_m minimizes the residual norm $\|b - Ax\|_2$ over all candidate vectors in $x_0 + K_m$. In contrast, there is no similar optimality property known for methods of the first class when A in nonsymmetric. Because of this, many methods of this type have been derived for the nonsymmetric case [11,57,38,109]. The Conjugate Residual method [28] is the analogue of conjugate gradient method that is in this class. The GMRES algorithm [109], which we will describe in Section 3.5, is an extension of the Conjugate Residual method to nonsymmetric problems.

3. $L_m = K_m(A^T, r_0); K_m = K_m(A, r_0)$. Clearly, in the symmetric case this class of methods reduces to the first one. In the nonsymmetric case, the biconjugate gradient method (BCG) due to Lanczos [68] and Fletcher [44] is a good representative of this class. There are various mathematically equivalent formulations of the biconjugate gradient method [103], some of which are more numerically viable than others. An efficient variation on this method, called CGS (Conjugate gradient squared) was proposed by Sonneveld [118,95].

4. $L_m = K_m = K_m(A^TA, A^Tr_0)$. This is nothing but the conjugate gradient method applied to the normal equations $A^TAx = A^Tb$, often referred to as CGNR[38]. The condition number of the normal equations is likely to be too large for most problems to make this approach competitive with the approaches 1 to 3, except possibly for indefinite problems, i.e., problems for which the symmetric part is not positive definite. LSQR [92] is an implementation that is somewhat less sensitive to large condition numbers. Moreover, for least squares problems with non-square matrices, one must either explicitly or implicitly use an approach based on the normal equations. We put in this category also the method of conjugate gradients applied to $AA^Ty = b$, whose solution y is trivially related to x by $x = A^Ty$. This is often referred to as CGNE, or Craig's method. If we express the Galerkin conditions in terms of the y variable, then, clearly, $K_m = K_m(AA^T, r_0)$ and $L_m = K_m$. Using the relationship $x = A^Ty$ between the x and y variables, we can translate the Galerkin condition that y satisfies in terms of the x variable to find that for the variable x Craig's method corresponds to taking $K_m = K_m(A^TA, A^Tr_0)$ and $L_m = A^{-T}K_m$. Moreover, the main difference between CGNR and CGNE is that the first minimizes the residual norm

over K_m while the second minimizes the error norm over K_m.

Note that the methods recently presented in [112], although related to Krylov subspace methods, do not belong to one of the above classes.

An important factor in the success of conjugate gradient-like methods is the preconditioning technique. This typically consists of replacing the original linear system (1) by, for example, the equivalent system

$$M^{-1}Ax = M^{-1}b \tag{4}$$

In the classical case of the incomplete LU preconditionings, the matrix M is of the form $M = LU$ where L is a lower triangular matrix and U is an upper triangular matrix such that L and U have the same structure as the lower and upper triangular parts of A respectively. In the general sparse case, the incomplete factorization is obtained by performing the standard LU factorization of A and dropping all fill-in elements that are generated during the process. This is referred to as ILU(0), or IC(0) in the symmetric case. There are more elaborate factorizations that allow a limited amount of fill-in to take place. In particular in the structured case one can let some fill-in appear along specific diagonals to get incomplete factorizations, denoted by ICCG(k), which more closely approximate A [76]. In the general sparse case this technique has been generalized by introducing the notion of level of fill-in [128]: the level of a fill-in element is defined as one plus the sum of the levels of the L and U elements from which it is spawned in the elimination process. Initially, all elements have level of fill-in equal to 0. ILU(k) is then defined as the incomplete factorization that is obtained by dropping all fill-in elements whose level exceeds k. An interesting question that comes to mind immediately is to see whether the more accurate factorizations will perform better in the context of parallel processing. The rationale is that we will deal with denser matrices and therefore the iterative technique will benefit from better vectorization and data locality. Unfortunately, the process of updating the levels is extremely expensive and sequential in nature. Experiments with such techniques reveal that the preprocessing phase of computing the incomplete LU factorization with any level of fill exceeding one dominates the computing time on (multi)-vector processors or and was rarely competitive with the simpler ILU(0) or ILU(1) preconditioners [5,116].

Another class of successful preconditioning techniques based on block factorizations was popularized by Concus, Golub and Meurant [31]. For reasons of space we will only briefly give an overview of these in Section 6, but we should mention that their performance has been well documented in the literature; see Meurant [82,78], Eisenstat et al. [36], and Axelsson[13]. There exist many other ways of defining incomplete factorizations of a given matrix, most of which are based on some form of diagonal dominance. For instance many of the standard point or line relaxation techniques such as Gauss-Seidel, SOR, SSOR, or ADI, see [126], can be used as preconditioners.

The Generalized Conjugate Gradient (GCG) method, introduced by Concus and Golub [30] and Widlund[129] can be viewed as a particular case of a preconditioned conjugate gradient method where the preconditioning matrix is $M = A + A^T$. With this special choice of the preconditioning matrix M, there is a three term recurrence similar to that of the conjugate gradient method. On the other hand, each step of GCG requires the solution of a linear system with the matrix M, which may be uneconomical.

For the standard ILU/IC preconditioners, solving a linear system with the matrix M, requires performing a forward and a backward triangular system solution at every step of the Krylov subspace method. This may constitute the main bottleneck in iterative methods if not carefully implemented on supercomputers and will be discussed in detail later.

3 Preconditioned Conjugate Gradient.

3.1 The algorithm

By far the most popular Krylov subspace method for solving symmetric positive definite linear systems is the preconditioned conjugate gradient method, a version of which is described below. Here M represents the preconditioning matrix.

Algorithm: preconditioned CG

1. *Preprocess:* Compute preconditioner M.
2. *Start:* $r_0 := b - Ax_0$, $p_0 := z_0 := M^{-1}r_0$.
3. *Iterate:* Until convergence do,
   (a) $w := Ap_i$
   (b) $\alpha_i := (r_i, z_i)/(w, p_i)$
   (c) $x_{i+1} := x_i + \alpha_i p_i$
   (d) $r_{i+1} := r_i - \alpha_i w$
   (e) $z_{i+1} := M^{-1}r_{i+1}$
   (f) $\beta_i := (r_{i+1}, z_{i+1})/(r_i, z_i)$
   (g) $p_{i+1} := z_{i+1} + \beta_i p_i$

The above algorithm is nothing but the conjugate gradient method applied to the linear system $M^{-1}Ax = M^{-1}b$ in which the standard Euclidean inner product is replaced by the inner product $(x, y)_M = (Mx, y)$. It has the property of computing an approximate solution whose preconditioned residual vector $M^{-1}(b - Ax_i)$ is M-orthogonal to all the previous preconditioned residual vectors. Therefore it is a Krylov subspace method of the first type (Galerkin) with the matrix A replaced by the preconditioned matrix $M^{-1}A$ and the standard dot product replaced by the M dot product.

Concerning supercomputer implementation, one observes that the main operations in the above algorithm are the following.

1. Setting up of the preconditioner;

2. Matrix vector multiplications (3-a);

3. Vector updates (3-c, 3-d and 3-g);

4. Dot products (3-b and 3-f).

5. Preconditioning operations (3-e)

In the above list the potential bottleneck is in setting-up the preconditioner (1) and in the solution of linear systems with M, i.e., operation (5). Because of their importance, we will address the problem of efficient implementation of the operations in a separate section. Also potentially time consuming are matrix by vector products which deserve a particular attention. The rest of the algorithm consists essentially of dot products and vector updates. Since the dot products may be the source of bottlenecks on some machines, particularly those with a large number of processors, they should be carefully examined. This issue will also be discussed separately.

3.2 Matrix by vector products.

Matrix by vector multiplications are relatively easy to implement efficiently on most supercomputers. The first observation that has been made in this context is that this operation can be performed by diagonals when the matrix is regularly structured, i.e., when it consists of a few diagonals [62]. The matrix can be stored in a rectangular array $DIAG(1:n, 1:ndiag)$ and the offsets of these diagonals from the main diagonal may be stored in a small integer array $IOFF(1:ndiag)$.

After initializing the vector y to zero, the main loop for computing $y = Ax$ is as follows:

```
      DO 10 J=1, NDIAG
         JOFF = IOFF(J)
         DO 20 I=1, N
            Y(I) = Y(I) + DIAG(I,J)*X(JOFF+I)
20       CONTINUE
10    CONTINUE
```

This can be implemented efficiently and thus excellent megaflops rates can be reached on vector machines when the matrix is large enough.

For general sparse matrices there has been several attempts to obtain similar performances by either generalizing the diagonal storage scheme [89,133] or by reordering the matrix so as to obtain a diagonal structure [3,97]. We will only discuss the first approach here. This approach is of interest only for matrices whose maximum number of nonzeros per row, $jmax$, is small. One then stores the entries of the matrix in a real array $COEFF(1:n, 1:jmax)$ together with an integer array $JCOEFF(1:n, 1:jmax)$ that stores the column numbers of each entry of $COEFF$. We refer to this as the ITPACK/ELLPACK format. The above FORTRAN loop then becomes,

```
      DO 10 J=1, JMAX
         DO 20 I=1, N
            Y(I) = Y(I) + COEFF(I,J)*X(JCOEFF(I,J))
20       CONTINUE
10    CONTINUE
```

The main difference between this loop and the previous one is the presence of indirect addressing in the innermost computation. Note that if the number of nonzeros per row varies substantially, then many zero elements must be stored unnecessarily, and this scheme may become inefficient.

When considering matrices from real applications it is interesting to observe that there is always some structure to them. In particular most matrices are such that a large percentage of their elements belong to a few diagonals. One can therefore extract a small number of diagonals, store them as was described above for structured matrices and put the rest on the elements in a general sparse matrix storage. The complexity of a code that does this conversion is on the order of the nonzero elements and is therefore affordable. The payoff could be quite high especially on vector machines that do well on long vectors. Clearly, here again many zero elements may have to be added to fill the diagonals and so one must be careful when assessing performances.

All the above storage schemes are specialized to some degree to certain types of matrices. These can perform well in many instances but their lack of generality is a serious limitation. Unfortunately, as is often the case, there is a conflict between generality and efficiency. One of the most general schemes for storing sparse matrices is the compressed sparse matrix storage scheme described next. The data structure consists of three arrays. First, a real array $A(1 : nnz)$ stores the nonzero elements of the matrix row-wise, i.e., the elements of a given row are stored contiguously. Then an integer array $JA(1 : nnz)$ stores the column positions of the elements in the real array A. Finally, an integer array $IA(1 : n + 1)$ is a pointer array, in that its i-th entry points to the beginning of the i-th row in the arrays A and JA. This data structure is often referred to as the general sparse format, or the A, JA, IA format. With this storage scheme each component of the resulting vector y can be easily computed independently as the dot product of the i-th row of the matrix with the vector x. In FORTRAN 8-X, we can write this as

```
      DO 10 I=1, N
            K1 = IA(I)
            K2 = IA(I+1)-1
            Y(I) = DOTPRODUCT( A(K1:K2) , X(JA(K1:K2)) )
10    CONTINUE
```

From the implementation point of view, an important observation is that the outer loop can be performed in parallel. On a machine like the Alliant FX-8, the synchronization of this outer loop is inexpensive and the performance of the above program can be excellent.

On distributed memory machines the above loop can be split and a number of its steps

will be executed in each processor. The splitting may be done in such a way that roughly the same amount of work is performed in each processor, taking other parts of the CG algorithm into consideration. The part of the matrix that is needed is loaded in each processor initially. However, interprocessor communication will be needed to get necessary parts of the vector x that do not reside in a given processor. For general sparse matrices, it may not be easy to find the mapping that achieves the best overall time.

The indirect addressing involved in the second vector in the dot product loop is handled by a special hardware instruction called a *Gather* operation. The vector $X(JA(k1 : k2))$ is first gathered from memory into a vector of contiguous elements. The dot product is then carried out as a standard dot product operation between two dense vectors.

In case the matrix is stored by columns instead of rows, we can use the following program to compute $y = Ax$,

```
      DO 10 J=1, N
            K1 = IA(J)
            K2 = IA(J+1)-1
            Y(JA(K1:K2)) = Y(JA(K1:K2)) + X(J) * A(K1:K2)
10    CONTINUE
```

Clearly, the above code also computes the product of the transpose of a matrix by a vector, when the matrix is stored row-wise in the A, JA, IA format. Normally, the vector $Y(JA(k1 : k2))$ is gathered and the SAXPY operation is performed in vector mode. Then the resulting vector is 'scattered' back into the positions $JA(*)$, by what is called a *Scatter* operation. However, a major difficulty with the above FORTRAN program is that it is intrinsically sequential. First, the outer loop is not parallelizable as it is, but this may be remedied as will be described shortly. Second, the inner loop involves writing back results of the right hand side, into memory positions that are determined by the indirect address function JA. To be correct Y(JA(1)) must be copied first and then Y(JA(2)), etc.. However, if it is known that the mapping JA(i) is one-to-one then the order of the assignments no longer matters. Since compilers are not capable of deciding whether this is the case, a compiler directive from the user is necessary for the Scatter to be invoked. Going back to the outer loop, one can split it in p distinct parts and compute p sub-sums into p temporary (full) vectors, that are added after completion to get the result vector y. This last part constitutes additional work but it is highly vectorizable and parallelizable.

The first vector machines that appeared did not perform too well on sparse computations because they were not equipped with special instructions for Gather and Scatter. The beneficial impact of hardware "Scatter" and "Gather" on vector machines has been discussed in [69].

For vector machines the previous two techniques are likely to perform poorly because they involve vectors that are usually very short. For example for a typical two dimensional problem with one unknown per grid point, the number of nonzeros per row is at most 5, when finite differences are used. One option is to use one of the schemes based on diagonal or generalized banded format described above. However the following scheme related to the

stripe structure of Melhem [77], is a more general alternative. We start from the A, JA, IA data structure and reorder the rows of the matrix according to their number of nonzeros, decreasingly. Then, a new data structure is built by constructing what we call "jagged diagonals" (j-diagonals). We store as a dense vector the leftmost element from each row, together with an integer vector containing the column positions of each element. This is followed by the second jagged diagonal consisting of the elements in second position from the left. As we build more and more of these diagonals, their length decreases. The number of j-diagonals is equal to the number of nonzero elements of the first row, i.e., to the largest number of nonzero elements per row. To multiply a matrix by a vector using this scheme one can proceed as follows, where we denote by IDIAG(j) the pointer to the beginning of the j-th jagged diagonal, and by JDIAG(k) the column position of the element stored in A(k):

```
      DO 10 J=1, NDIAG
            K1 = IDIAG(J)
            K2 = IDIAG(J+1)-1
            LEN = K2-K1+1
            Y(1:LEN) = Y(1:LEN) + A(K1:K2)*X(JDIAG(K1:K2))
10    CONTINUE
```

On one processor of the Cray-2, the asymptotic speed of the above code is around 39 Mflops [94]. Note that since we assume that the rows of the matrix A have been permuted the above code will compute a permutation of the vector Ax, for the unpermuted matrix A. It is possible to permute the result back to the original ordering after the execution of the above program. One can also postpone this operation until the final solution has been computed, so that only two permutations on the solution vector are needed, one at the beginning and one at the end. For preconditionings that require a different ordering of the unknowns to achieve a good efficiency, it will be necessary to perform a permutation before or within each call to the preconditioning subroutines.

As an illustration we show in the next table the performance of the following five different ways of multiplying a matrix by a vector:

1. Row-wise storage , (sparse dot product form);
2. Column wise storage, (sparse saxpy form);
3. Diagonal storage, (triad form);
4. Itpack format
5. Jagged diagonal format;

The test was done on an Alliant FX-80, in double precision arithmetic, using 5-point and 7-point matrices for 2-D and 3-D rectangular grids.

Notice the wide differences in performance obtained between these five kernels that perform the same operation. On the Alliant FX-80, method 2, using the column-wise storage

GRID SIZE	METHOD				
	1	2	3	4	5
20 x 20	6.56	0.19	8.04	9.43	7.11
20 x 20 x 10	7.06	0.21	7.013	10.13	4.05
30 x 30	6.7	0.19	8.98	10.83	8.19
30 x 30 x 10	5.42	0.21	6.48	9.69	3.32

Table 1: Megaflop rates for five matrix by vector multiplication kernels on an Alliant FX-80.

is the worst performer. It is important to stress that relative performance is machine dependent: the dot product scheme which performs well here would not too well on vector machines. Also of interest, and to some extent disturbing, is the variation in performance obtained for different matrices *with the same kernel.* These discrepancies are especially noticeable in Kernel 5, using the jagged diagonal format. Here the large degradation in performance when passing from two-dimensional to three-dimensional grids is due to the fact that the amount of data that is used is so large that it does not fit in the 128 KB cache. For the 30 x 30 x 10 matrix, the number of double precision words needed to hold the matrix plus the input and output vectors is roughly 7 x N + 2 x N = 9 x N, i.e., 81K words, while the cache size is only 64K words. At this point, execution time is dominated by memory traffic. Every time a j-diagonal is used, i.e., for each outer loop, the above code sweeps through the j-diagonal array $A(k1 : k2)$ itself plus the integer arrays for the indirect addressing $JDIAG(k1 : k2)$ and finally through the vectors x and y, with a very high cache-miss ratio. This situation might be remedied by unrolling the j loop to avoid unnecessary reloading of the vectors x and y.

One problem that seems not to have been studied in the literature is that of performing simple operations with general sparse matrices on SIMD machines like the MPP or the Connection machine. On such machines much of what has been accomplished is to test the usual symmetric conjugate gradient method for easy model problems [27,25]. The difficulty with the more realistic general sparse problems is the apparent necessity of resorting to indirect addressing, a difficult operation on these architectures. Hammond and Law [55] propose a hardware solution based on systolic arrays. This challenging problem must be solved before SIMD machines can be considered real contenders to MIMD machines in the race for practical supercomputers.

3.3 The problem of the dot products

It has been observed that the dot products in the conjugate gradient algorithm constitute a bottleneck on many parallel or vector machines. This is because when all the vectors in the algorithm are split equally among the processors dot products require global communication. However, this need not be a problem unless the number of processors becomes large.

Another minor difficulty caused by the dot products is the fact that they constitute

two synchronization points in the algorithm. The dot products must be completed before anything else can be done and no other computation can be done while they are being computed. As was observed independently in [59] and [105] this can be partially overcome by exploiting the orthogonality of the vectors r_{i+1} and r_i from which we can derive the equality,

$$\|r_{i+1}\|_2^2 + \|r_i\|_2^2 = \alpha_i^2 \|Ap_i\|_2^2. \tag{5}$$

As a result β_{i+1} can be computed from α_i and Ap_i, leading to the following formulation, when $M = I$ (No preconditioning). Note that the algorithm can easily be extended to the preconditioned case; see [81,80].

Algorithm: CG, version 2

Start: $p_0 := r_0 := b - Ax_0$ and $\rho_0 := \|r_0\|_2^2$

Iterate: For $i = 0, 1, \ldots$ until convergence do:

1. Compute $w := Ap_i$, (Ap_i, r_i) and $\|Ap_i\|_2^2$;
2. Compute the scalars

$$\begin{aligned} \alpha_i &:= \rho_i/(Ap_i, r_i) \quad , \quad cos^2\theta_i := \frac{(Ap_i, r_i)^2}{\rho_i \|Ap_i\|_2^2} \\ \beta_i &:= tan^2\theta_i = 1/cos^2\theta_i - 1 \quad , \quad \rho_{i+1} := \beta_i \rho_i \end{aligned}$$

3. Compute
   - $x_{i+1} := x_i + \alpha_i p_i$
   - $r_{i+1} := r_i - \alpha_i w$
   - $p_{i+1} = r_{i+1} + \beta_i p_i$

Unfortunately, the above version of CG is unstable and there is a simple analysis to understand the difficulty and to remedy it. To simplify the notation we introduce the quantities,

$$c_i \equiv cos^2\theta_i \quad , \quad s_i \equiv sin^2\theta_i \quad , \quad t_i \equiv tan^2\theta_i \quad , \quad \tau_i \equiv c_i\rho_i = \frac{(Ap_i, r_i)^2}{\|Ap_i\|_2^2} \tag{6}$$

Then we have the following result on the relative error on ρ_{i+1},

$$\frac{\delta\rho_{i+1}}{\rho_{i+1}} \approx \frac{\delta\rho_i}{\rho_i} + \frac{\delta t_i}{t_i} \tag{7}$$

Similarly for t_i,

$$\frac{\delta t_i}{t_i} \approx -\frac{\delta c_i}{c_i^2}\frac{1}{t_i} = -\frac{\delta c_i}{c_i^2(1/c_i - 1)} = \frac{-\delta c_i}{c_i s_i} \tag{8}$$

and finally for c_i,

$$\frac{\delta c_i}{c_i} \approx \frac{\delta \tau_i}{\tau_i} - \frac{\delta \rho_i}{\rho_i} \tag{9}$$

From (7), (8) and (9) we get

$$\frac{\delta \rho_{i+1}}{\rho_{i+1}} \approx \frac{\delta \rho_i}{\rho_i} - \frac{1}{s_i}\left[\frac{\delta \tau_i}{\tau_i} - \frac{\delta \rho_i}{\rho_i}\right] = (1 + \frac{1}{s_i})\frac{\delta \rho_i}{\rho_i} - \frac{1}{s_i}\frac{\delta \tau_i}{\tau_i} \tag{10}$$

The above formula suggests that there is a danger of catastrophic error when s_i gets close to zero. In fact even if s_i is not small, the above expression tells us that the accumulated error may grow exponentially. To simplify our model a little further, let us assume that $\tau_i = (Ap_i, r_i)^2/\|Ap_i\|_2^2$, a quantity that is computed locally and independently of ρ_i has no error in it, i.e., that $\delta\tau_i = 0$. Then the above formula yields,

$$\frac{\delta \rho_{i+1}}{\rho_{i+1}} \approx (1 + \frac{1}{s_i})\frac{\delta \rho_i}{\rho_i} \tag{11}$$

The meaning of the above relation is that even under the strong condition that there are no *local* errors introduced, an initial error on ρ_0 will affect the predicted value of ρ_{i+1} by an error which grows like the product of the factors $1 + 1/s_i$. The resulting algorithm will be unstable in general.

A simple remedy would be to keep track of the growth of the estimated error and recompute occasionally the residual norm by the usual formula. In fact Meurant [81,80] recomputes it at every step and shows that the additional dot product is worth the cost on a vector machine like the Cray X-MP. Note that the scalar s_i on which the error estimate is based, is available at every step for free. The above analysis can be extended to the preconditioned conjugate gradient method, by using the M-inner product instead of the Euclidean inner product.

One can take the above idea of post-poning inner products one step farther and ask whether it is possible to derive a version of the conjugate gradient method that has as few synchronization points as possible. One idea is to try to execute m steps of the conjugate gradient algorithm at once [29]. However, one can expect to encounter the same stability difficulties as before. Another simple and reliable way to reduce bottlenecks due to inner products is to use polynomial preconditioning as will be described in Section 5.

There are some difficult issues when implementing algorithms on parallel/vector machines pertaining to the best use of the available hardware and software. We have mentioned the delicate problem of optimizing the use of vector registers, the hardware gather and scatter operations, and the disastrous effect that cache memories may have on sparse computations. Work by Meurant on a Cray X-MP-48 [80], Seager[115], and Koniges[66] addressed the question of how to implement conjugate gradient methods and best exploit multitasking and microtasking. Efficient implementations of different preconditioning techniques is more difficult on multi vector processor machines such as the Alliant FX-8 because of the complicated side effects of the memory hierarchy [56,75]. Thus, it seems that there are no all-purpose preconditioning techniques.

3.4 Preconditioned GMRES

GMRES [109] is an effective conjugate gradient-like algorithm for solving general large sparse linear systems of equations of the form

$$Ax = b. \tag{12}$$

Assuming a preconditioner M is used on the left, see [38], we will be solving instead of (12), the preconditioned linear system

$$M^{-1}Ax = M^{-1}b. \tag{13}$$

A brief description of the preconditioned GMRES method follows. Details can be found in [109].

Algorithm : Preconditioned GMRES

1. *Start:* Choose x_0 and a dimension m of the Krylov subspaces.

2. *Arnoldi process:*

   - Compute $r_0 = M^{-1}(b - Ax_0)$, $\beta = \|r_0\|$ and $v_1 = r_0/\beta$.
   - For $j = 1, 2, .., m$ do:

$$\begin{aligned} h_{i,j} &= (M^{-1}Av_j, v_i), \quad i = 1, 2, ..., j, \\ \hat{v}_{j+1} &= M^{-1}Av_j - \sum_{i=1}^{j} h_{i,j}v_i \\ h_{j+1,j} &= \|\hat{v}_{j+1}\|, \\ v_{j+1} &= \hat{v}_{j+1}/h_{j+1,j}. \end{aligned} \tag{14}$$

   Define H_m as the $(m+1)$ x m upper Hessenberg matrix whose nonzero entries are the coefficients h_{ij}.

3. *Form the approximate solution:*

   - Find the vector y_m which minimizes the function $J(y) = \|\beta e_1 - H_m y\|$ where $e_1 = [1, 0, \ldots 0]^T$, among all vectors of R^m.
   - Compute $x_m = x_0 + V_m y_m$

4. *Restart:* If satisfied stop, else set $x_0 \leftarrow x_m$ and goto 2.

Each outer loop of the above algorithm, i.e., the loop consisting of steps 2, 3, and 4, is divided in two main stages. The first stage is an Arnoldi step and consists of building a basis of the Krylov subspace K_m. The second consists of finding in the affine space $x_0 + K_m$ the approximate solution x_m which minimizes the residual norm. This is found by solving the least squares problem of size $m+1$ of step 3, whose coefficient matrix is the upper Hessenberg matrix H_m.

Note that in practice, the least squares solution of the $(m+1) \times m$ problem in 3 is solved by a QR factorization of the matrix H_m which is updated at each step of the Arnoldi process in 2. With this implementation we can obtain at no additional cost the residual norm of the corresponding approximate solution x_k without having to actually compute it; for details see [109]. This allows us to stop at the appropriate step.

When the preconditioned matrix is positive real, then GMRES is theoretically equivalent to GCR [38] and to ORTHODIR [53] and it is known to converge. Moreover, it is less costly both in terms of storage and arithmetic [109]. It can be shown that, in exact arithmetic, the method cannot break down although it may be very slow or even stagnate in cases when the matrix is not positive real. For details on stagnation and breakdown behaviors of Arnoldi and GMRES methods, see the recent analysis by Brown [24].

Consider the implementation of the above algorithm on a vector or parallel machine. As before we start by enumerating the main kernels of the algorithm.

1. Setting-up the preconditioner;
2. Matrix by vector multiplication;
3. Orthogonalizing a vector against a set of orthogonal vectors;
4. Vector updates;
5. Preconditioning operation.

The work involved in solving the small least squares problem in step 3 of the algorithm is negligible for large linear systems.

The new operation here with respect to the conjugate gradient method is the orthogonalization of the vector Av_i against the previous $v's$. The usual way of accomplishing this is via the modified Gram Schmidt process, which is basically a sequence of subprocesses of the form

- Compute $\alpha = (y, v)$
- Compute $\hat{y} := y - \alpha v$

consisting of orthogonalizing a vector y against another vector v of norm one. Thus the outer loop of the modified Gram-Schmidt is not parallelizable, but the inner loop, i.e. each subprocess, can be parallelized by dividing the inner product and saxpy operations among processors. Although this constitutes a perfectly acceptable approach for a small number of processors, the elementary subtasks may be too small for this approach to be efficient on a

large number of processors. In that case one solution is to use a standard Gram-Schmidt process with reorthogonalization. This would replace the previous sequential orthogonalization process by a matrix operation of the form $\hat{y} = y - VV^T y$, i.e., BLAS-1 kernels are replaced by BLAS-2 kernels. Further, we should point out that BLAS-3 type kernels that usually allow extremely high performances on machines with caches or local memories [47] cannot be used here because at every step we have only one vector to orthogonalize against all previous ones. To some extent this may be remedied by using block methods considered next. An alternative technique for the orthogonalization process in GMRES was recently proposed by H. Walker [127]. This technique, based on the Householder process, has superior numerical properties, and can easily be parallelized.

Radicati and Robert[98] compare the performance of several Krylov subspace methods on an IBM 3090 VF. In particular they discuss good strategies for implementing the main kernels on this machine and provide an exhaustive set of experiments. The paper focuses on iterative solvers rather than preconditioners and concludes that although it is difficult to make any definite statements as to an overall best method, on the average GMRES and CGS did perform better than the other iterative methods tested.

3.5 Block Krylov subspace methods

The idea of using a block of vectors instead of a single vector in methods such as the conjugate gradient algorithm and the Lanczos algorithm has been suggested by several authors as a simple means for increasing parallelism [107,106,85,87]. The main idea can be explained by assuming that we have to solve a linear system of the form $AX = B$ where the right hand side is no longer a single vector but an $N \times p$ matrix. Then a natural modification of the CG algorithm consists of replacing all the operations with single vector by operations with blocks of p vectors. After convergence, the block conjugate gradient algorithm would have solved the p systems simultaneously. If only one linear system must be solved then additional artificial right hand sides must be created. In absolute terms this approach is not efficient, i.e., the total number of arithmetic operations is likely to be much higher than with the standard single vector method. Note that there are many problems involving linear systems with several right hand sides in which case a block method becomes very attractive even on scalar machines.

A well-known attraction of the block methods is that for out-of-core problems they tend to reduce the number of accesses to secondary storage. Thus a block Lanczos algorithm was found preferable to the single vector Lanczos algorithm in the context of eigenvalue calculations [52,70]. Many of the modern supercomputers emphasize a hierarchical organization of the memory using principles similar to those used for secondary storage in traditional computers, except that the number of levels in the hierarchy is higher. One example of this architecture is the CEDAR multiprocessor of the University of Illinois [67]. Access to data from the global shared memory in CEDAR is more expensive than access to data within each local memory (cluster memory). For this reason, block methods may be important since they will help reduce inter-processor communication, in the same way they have been used in the past to reduce the number of accesses to secondary storage [52]. Block methods

can also be combined with the other alternatives described in this paper, such as reordering, and offer an excellent potential for shared memory machines with a hierarchical organization of memory.

4 Solution of sparse triangular systems

Each step of a preconditioned iterative method involves computing

$$z = M^{-1}y.$$

In typical preconditioners M is the product of a lower and an upper triangular matrix, often having the same sparsity pattern as the lower and the upper triangular parts of the original matrix. We consider in this section different ways of performing this operation which is critical to the performance of the preconditioned conjugate gradient method. We only consider lower triangular systems of the form

$$Lx = b. \tag{15}$$

Without loss of generality we will assume that L is unit lower triangular. If not the matrix can be scaled before the CG iteration is started so as to save N multiplications per CG step.

If we assume that the matrix L is stored row-wise in a general sparse format, using the standard sparse storage scheme,

- AL : nonzeros of L, stored by rows,
- JAL: column numbers for each element of AL,
- IAL: IAL(i) points to the start of row i in AL, JAL,

the sequential elimination sweep is as follows:

Algorithm: Forward elimination for a sparse triangular system

```
x(1)  =  b(1)
do i  =  2, N
         x(i)  =  b(i)
         do j= ial(i), ial(i+1)-1
               x(i)  =  x(i) - al(j) * x(jal(j))
         enddo
enddo
```

The outer loop corresponding to the variable i is sequential. The j loop is essentially a sparse dotproduct of the i^{th} row of L and the dense vector x. This dot product can be split among the processors and the partial results will then be added at the end. This is what the Alliant FX-8 compiler would do if the proper optimization options are invoked. However, if the length of the dot product is very short, the synchronization overhead and additional

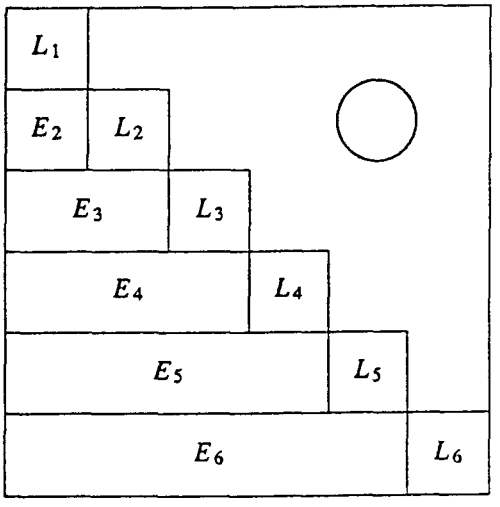


Figure 1: Block partitioning for L

operations involved will make this approach inefficient. Consequently, on a machine such as the Alliant FX-8, it is often better not to use the optimized version of the compiler for the above code. As an alternative we may store the matrix by columns and use a column oriented algorithm. However, the difficulties are identical and we omit the details.

We briefly describe two distinct approaches for breaking the sequential nature of the above implementation; for details see [6,5].

4.1 Blocking

In the blocking approach, the right hand side b and the solution x of (15) are partitioned into subvectors $b_1, b_2, ..., b_m$ and $x_1, x_2, ..., x_m$ respectively. According to this partitioning the matrix L will have the structure shown in Figure 1.

If we denote by $\hat{x}_j$ the vector consisting of the subvectors $x_1, x_2, ..., x_j$, then the algorithm for computing the solution of the linear system (15) can be written as follows:

Algorithm: Block forward elimination
$x_1 = L_1^{-1} b_1$
Do $i = 2, ..., m$:

$$x_i = L_i^{-1} (b_i - E_i \hat{x}_{i-1}) \tag{16}$$

Each step, except the first, consists of two basic operations: first a multiplication of the vector $\hat{x}_{i-1}$ by E_i and then a solution of a triangular system with the matrix L_i. The multiplication $E_i \hat{x}_{i-1}$, a sparse matrix times vector operation, causes no difficulty because all the inner products of rows of E_i with $\hat{x}_{i-1}$ can proceed in parallel. The other significant operation, computing $x_i = L_i^{-1} y$ by solving a lower triangular system involving L_i, is the same problem with which we started, although on a smaller scale.

The first option is simply to solve this smaller system by the usual sequential algorithm using forward elimination. Clearly, this suffers from the same sequential nature as the initial algorithm, but it does have the advantage of preserving the sparsity of the original problem and there is the potential benefit that the sequential bottleneck is now shorter at every step since at least the matrix by vector products are done in parallel. The experiments in [6] reveal that the performance of this alternative is not too different from that of the sequential algorithm; see also the experiments reported at the end of this section. The second option consists of computing the inverse of each matrix L_i at the beginning and storing it as a dense matrix. Then the triangular system solution in (16) can be replaced by the multiplication of a triangular matrix by a vector. The attraction is that multiplying a vector by a matrix is highly parallelizable and vectorizable and despite the additional arithmetic, could be cost-effective when solving a large number of systems with the same matrix. Note that an intermediate option for preconditioners is to compute a sparse approximate inverse for the triangular matrix L_i instead of a full inverse. We can afford to make such a substitution because we are only interested in using the inverse of L to precondition the original system. This approach is essentially to the one used in [125] for the diagonally structured problems.

4.2 Level scheduling

In the block forward elimination, the main bottleneck is in solving each of the subsystems (16). If the matrices L_i were diagonal matrices the main difficulty would be removed and (16) would be a fully parallelizable operation of the order of the block size. For many sparse matrices it is possible to obtain such a block triangular system simply by reordering the rows and columns of the coefficient matrix. We will describe a simple ordering called *level scheduling* [6,111,130], whose objective is to obtain a block lower triangular system like the one in Figure 1, where the E_i's are sparse rectangular matrices and the L_i's are diagonal blocks.

The idea of *forward scheduling* is as follows. Consider the following formula to compute

the i^{th} unknown,

$$x_i = \frac{1}{l_{ii}} \left(b_i - \sum_{j<i \ , \ l_{ij} \neq 0} l_{ij} x_j \right) \tag{17}$$

This can be executed after all the components x_j needed in the sum have been computed. The idea is to look at the adjacency graph of the matrix at the outset, and determine groups of equations that can be solved at the same time [6,111,130]. Recall that a node of the graph corresponds to a row of L or, equivalently, a component of x, and there is a directed edge from node j to node i if and only if $l_{ij} \neq 0$, which indicates that x_j must be known to solve for x_i. Since L is lower triangular, the adjacency graph is a directed acyclic graph. Figure 2 shows the digraph for a small sample matrix.

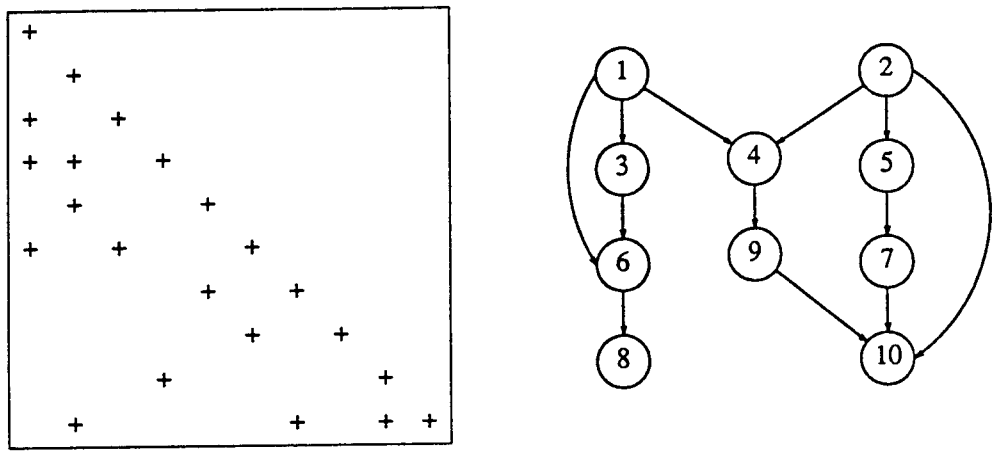
Thus, the first step of the solution algorithm consists of determining x_1 and any other unknowns for which there are no predecessors in the graph, i.e., all those unknowns x_i for which the off-diagonal elements of row i of L are zero. These unknowns will constitute the elements of the first level. The next step will determine in parallel all those unknowns that will have the nodes of the first level as their (only) predecessors in the graph.

More generally, we can define a root node as an imaginary node with links to the nodes having no predecessors, and the *depth* of a node as the maximum distance of that node from the root [130]. The introduction of the root node ensures that the depth of each node is defined from the same point. The depth of each node can be computed with one pass through the structure of the coefficient matrix L by

$$depth(i) = \begin{cases} 1 & \text{if } l_{ij} = 0 \text{ for all } \ j < i \\ 1 + max_{j<i}\{depth(j) \ , \ l_{ij} \neq 0\} & \text{otherwise} \end{cases} \tag{18}$$

A *level* of the graph is, by definition, the set of nodes with the same depth. Thus the depth of a node is the same as the block number of its row in the block algorithm, and the nodes of a level define the set of rows in a block. Let us assume that we now define a data structure for the levels: a permutation $q(1:n)$ defines the new ordering and $level(i), i = 1, \ldots, nlev+1$ points to the beginning of the i-th level in that array. Then the algorithm for solving the triangular systems can be written as,

Before reordering



After reordering by levels

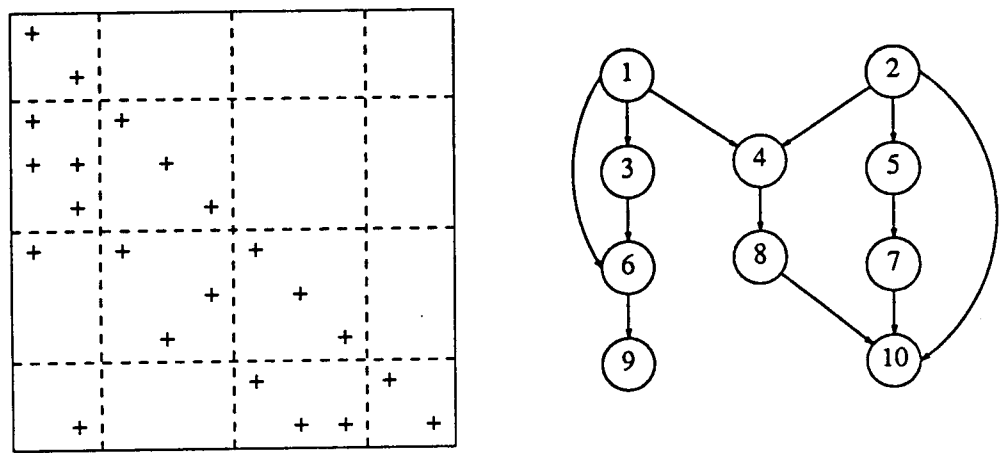


Figure 2: A sparse lower triangular matrix and its level structure.

Algorithm: Forward elimination with level scheduling

```
do lev=1, nlev
j1 = level(lev)
j2 = level(lev+1)-1
     do k = j1, j2
       i = q(k)
       do j= ial(i), ial(i+1)-1
           x(i) = x(i) - al(j) * x(jal(j))
       enddo
      enddo
 enddo
```

The k loop can be executed in parallel. The idea of forward level scheduling is a natural one for finite difference matrices on rectangles and several authors suggested it independently, [124,123,50,110,9]. It is also interesting to note that for the computer scientist the idea in the general context of irregularly structured matrices is a textbook example of scheduling for parallel processing. In fact the level scheduling approach described here is a "greedy" algorithm and is unlikely to be optimal. There is no reason why one should solve an equation as soon as it is possible and it may be preferable to use a *backward scheduling* [5] which consists of defining the levels from bottom up in the graph. Thus the last level consists of the leaves of the graph, the previous level consists of their predecessors, etc..

Another possibility is to use dynamic scheduling as opposed to static scheduling. The main difference is that the level structure is not preset; rather the order of the computation is determined at run-time. The clear advantage over pre-scheduled triangular solutions is that it allows processors to always execute a task as soon as its predecessors have been completed thus reaching a better load balancing. On loosely coupled distributed memory machines this approach may be the most viable since it will dynamically adjust to irregularities in the execution and communication times that can cause a lock-step technique to become inefficient. On the other hand, for those shared memory machines in which hardware synchronization is available and inexpensive, such as the Alliant FX-8, dynamic scheduling would have some disadvantages since it requires managing queues and generating explicitly busy waits. Both approaches have been tested and compared in [21] where it was concluded that on the Encore Multimax dynamic scheduling is usually preferable except for problems with few synchronization points and large amount of parallelism. In [54] a combination of prescheduling and dynamic scheduling was found to be the best approach on a Sequent balance 21000. There seems to have been no comparison of these two approaches on distributed memory machines or on shared memory machines with microtasking, or hardware synchronization features.

To illustrate the performance of the solvers described in this section we show in Table 2 speed-ups of the various methods considered here over the sequential version Lsol on a

number of matrices. The experiments have been performed on an Alliant FX-8, with 8 processors, using double precision arithmetic. The meanings of the labels are as follows:

- blksla : Blocking method with sequential solve;
- blksla : Blocking method with dense inverse solve;
- levsl : level scheduling solve;
- jagsl : level scheduling using the jagged diagonal format for the matrix-vector products $E_i\hat{x}_{i-1}$ in (16).

The first two sets of matrices are from the sets RUA and RSA of the Harwell-Boeing collection of sparse matrices[35]. The last set consists of 5-point or 7-point matrices on rectangular domains, with the numbers referring to the grid sizes.

In the table n represents the size of the matrix while nnz is its total number of nonzero elements. The numbers shown in the last four columns are the speed-ups of the four techniques they refer to, as compared to the times obtained for Lsol, the sequential method whose program is shown at the beginning of this section. Recall that the number of arithmetic operations performed by blksla, levsl and jagsl is identical with that of the sequential solve, while blkslb requires more operations. The block size used in blksla and blkslb is always 16. Note that the speed-up of blksla may be less than one and is otherwise negligible. Blkslb on the other hand can reach a speed-up that exceeds three in some cases despite the additional work involved. The speed-ups achieved by levsl are more consistent than the other techniques and they are generally lower for the denser matrices. Jagsl performed extremely well on the three-dimensional grid problems, sometimes reaching speed-ups that exceed the number of processors due to a better vectorization. The performance on some of the matrices could also be very poor. These matrices are those that have a large number of very short jagged diagonals [6].

In [22,21] and [5,6] a number of additional experiments are presented to study the performance of level scheduling within the context of preconditioned conjugate gradient methods.

5 Polynomial preconditioning

Polynomial preconditioning consists of choosing a polynomial s and replacing the original linear system by

$$s(A)Ax = s(A)b \tag{19}$$

which is then solved by a conjugate gradient type technique. There have been several recent publications on the use of such preconditioners motivated mostly by their potential on vector computers [8,58,105,60,115,131]. However, the idea of using polynomial preconditioning is an old one and has been suggested by Stiefel [119] for eigenvalue calculations. Later, Rutishauser [100] adapted Stiefel's method for linear systems. The idea of polynomial preconditionings reappeared with the paper by Dubois et al. [33] which was spurred by the potential of these techniques for vector machines. Their motivation was that many matrices have a diagonal

Matrix			Speedup over Lsol			
Name	n	nnz	blksla	blkslb	levsl	jagsl
steam2	600	7180	1.15	2.68	5.15	2.43
orsirr-2	886	3428	0.97	2.55	4.96	2.79
jpwh-991	991	3529	1.33	2.43	5.06	3.05
sherman1	1000	2375	0.87	1.94	4.46	4.07
sherman4	1104	2445	0.85	1.75	4.87	4.37
orsreg-1	2205	8169	1.10	2.50	5.51	5.36
sherman5	3312	11571	0.88	1.93	4.70	2.72
sherman3	5005	12519	0.97	1.96	5.03	6.68
bcsstk06	420	4140	0.91	2.84	2.58	0.43
bcsstk19	817	3835	0.89	2.88	1.41	0.45
bcsstk09	1083	9760	1.13	2.79	3.96	0.98
bcsstk27	1224	28675	1.33	3.51	1.73	0.22
bcsstm12	1473	10566	1.13	2.74	3.92	0.95
bcsstm13	2003	11973	0.99	1.92	2.71	0.44
bcsstk23	3134	24156	1.08	2.82	4.74	1.38
bcsstk21	3600	15100	1.20	2.70	5.83	5.87
20x20x1	400	1160	0.96	2.15	3.24	2.27
20x20x10	4000	15200	1.12	2.46	5.67	7.25
20x20x20	8000	30800	1.12	2.54	5.57	8.29
20x20x30	12000	46400	1.13	2.54	5.51	8.87
30x30x1	900	2640	0.97	2.10	3.84	2.88
30x30x10	9000	34500	1.13	2.50	5.55	8.27
30x30x20	18000	69900	1.14	2.51	5.33	9.29
30x30x30	27000	105300	1.14	2.54	5.29	9.77

Table 2: Speedup over the sequential method of four triangular system solvers, on an Alliant FX-8.

structure, in which case matrix by vector multiplications can be performed at near peak speed of the machine. Several papers discussing better polynomials or performance issues followed [58,105,60,80,37,131]. However, there are often doubts surrounding the usefulness of the method on supercomputers. Before discussing the advantages and disadvantages of the method we begin with an overview.

To be efficient, polynomial preconditionings require the determination of an optimum polynomial s. The preconditioned matrix $s(A)A$ should be as close as possible to the identity matrix in some sense. One possible criterion is to make the spectrum of the preconditioned matrix as close as possible from that of the identity. For example denoting by $\sigma(A)$ the spectrum of A, and by Π_k the space of polynomials of degree not exceeding k, we may wish to solve,

$$\begin{aligned} &\text{Find } s \in \Pi_k \text{ that minimizes:} \\ &\max_{\lambda \in \sigma(A)} |1 - \lambda s(\lambda)| \end{aligned} \tag{20}$$

Unfortunately, this problem involves all the eigenvalues of A and is harder to solve than the original problem. What is usually done is to replace Problem (20) by the problem obtained from replacing the set $\sigma(A)$ by some continuous set E that encloses it:

$$\begin{aligned} &\text{Find } s \in \Pi_k \text{ that minimizes:} \\ &\max_{\lambda \in E} |1 - \lambda s(\lambda)|. \end{aligned} \tag{21}$$

Thus, polynomial preconditioning techniques start with the assumption that we have a rough idea of the spectrum of the matrix A. We will come back to this problem later.

Consider first the particular case where A is symmetric positive definite in which case E can be taken to be the interval $[\lambda_{min}, \lambda_{max}]$ containing the eigenvalues of A. The best residual polynomial $1 - \lambda s(\lambda)$ in this case is a shifted and scaled Chebyshev polynomial of the first kind, and its three-term recurrence results in a simple three-term recurrence for the approximate solution [53]. An alternative considered by Johnson et al. in [58] is to use a least squares polynomial on the interval instead of the infinity norm polynomial. In other words we need to solve

$$\begin{aligned} &\text{Find } s \in \Pi_k \text{ that minimizes:} \\ &\|1 - \lambda s(\lambda)\|_w, \end{aligned} \tag{22}$$

where w is some weight function on the interval $(\lambda_{min}, \lambda_{max})$, and $\|.\|_w$ is the L_2-norm associated with the corresponding inner product. Because of the fact that the distribution of eigenvalues matters more than condition numbers for the preconditioned conjugate gradient method, it has been observed in [58] that least squares polynomials tend to perform better than those based on the uniform norm, in that they lead to a better overall clustering of the spectrum. Moreover, as was already observed by Rutishauser [100], in the symmetric case there is no need for accurate eigenvalue estimates: it suffices to use the simple bounds that are provided by Gershgorin's theorem. In [105] it was also observed that in some cases

the least squares polynomial over the Gershgorin interval, may perform as well as the infinity norm polynomial over $[\lambda_{min}, \lambda_{max}]$. Note that this is only a minor advantage of least squares polynomials since effective adaptive procedures exist to compute $\lambda_{min}, \lambda_{max}$; see [53] for symmetric problems and [72,41] for nonsymmetric problems. We should add that the observations made in [58] and in [105], and the simplicity of a method that bypasses eigenvalue estimates, have made least squares polynomials more popular for polynomial preconditionings. For the simple case of the model 5-point Laplace matrix, the best polynomials up to the degree 10 are listed in [105] and are readily usable.

We now consider the more general nonsymmetric case. Given a set of approximate eigenvalues of A, we can construct a region E in the complex plane which ideally would contain the eigenvalues of the matrix A. There are several choices for E. The first idea is to use an ellipse E [74,72] that encloses an approximate convex hull of the spectrum. Then the shifted and scaled Chebyshev polynomials are optimal when the focii of the ellipse are on the real axis and nearly optimal in other situations and the use of these polynomials leads again to an attractive three-term recurrence. A second alternative is to use a polygon H that contains $\sigma(A)$ [117,104]. A notable advantage of using polygons is that they may better represent the shape of an arbitrary spectrum. The polynomial is not explicitly known but it may be computed by a Remez algorithm. As in the symmetric case an alternative is to use an L_2-norm instead of the infinity norm, i.e., to solve (22) where this time w is some weight function defined on the boundary of H. Smolarski and Saylor used a discrete norm on the polygon. Their original algorithm is unstable [117] but an improved alternative is proposed in [113]. In [104] we used an L_2-norm associated with Chebyshev weights on the edges of the polygon and expressed the best polynomial as a linear combination of Chebyshev polynomials associated with the ellipse of smallest area containing H. If the contour of H consists of μ edges each with center c_i and half-length d_i, then the weight on each edge is defined by

$$w_i(\lambda) = \frac{2}{\pi} \, |d_i - (\lambda - c_i)^2|^{-1/2}. \tag{23}$$

With these weights, or any other Jacobi weights on the edges, there is a finite procedure to compute the best polynomial *that does not require numerical integration*; for details see [104]. The particular case where A is symmetric but indefinite was examined in detail in [101].

Still in the context of using polygons instead of ellipses, yet another attractive possibility proposed by Fischer and Reichel [43] is to avoid the problem of best approximation altogether and interpolate the function $1/z$ with a polynomial at the Fejer points of E, i.e., the points $e^{2ji\pi/k}, j = 0, 1, ..., k$ that are conformally mapped from the unit circle to H. This is known to be an asymptotically optimal process. There are numerous publications related to this approach and the use of Faber polynomials; see the references in [43]. However, we should point out that these papers are more concerned with the approximation theory problem than with the actual linear algebra problem. Often the tests performed are contrived into problems for which the spectrum is known to lie in nice and known regions. Real problems do not have nice spectra enclosed in rectangles that happen to be well separated from the origin.

We also mention that although we have only discussed approaches based on the formulations (21), (22), there are other less known possibilities based on minimizing $\|1/\lambda - s(\lambda)\|_\infty$; see Freund [46] and the references therein.

The next question that arises is: how to get the polygon H? In the symmetric positive definite case, where H reduces to an interval, this is not too difficult to do because of the relationship between the Lanczos algorithm and the conjugate gradient method; see e.g., [32,53]. This seems to have been proposed first by Rutishauser [100] who suggested using the QD algorithm for computing the eigenvalues of the tridiagonal matrix obtained from the CG/Lanczos procedure. In the nonsymmetric case an adaptive procedure was proposed by Manteuffel [74,72]. An improved technique that exploits Arnoldi's method was later developed in [41]. For illustration, we describe next an implementation based on a combination with GMRES [109]. Eigenvalue estimates can be computed from the Hessenberg matrices generated from GMRES. Thus, the idea is to use a certain number, say m_1, of steps of GMRES to get eigenvalue estimates and to improve the current solution. The eigenvalue estimates are used to improve the current polygon H and to compute the next least squares polynomial. A number of steps (say m_2) of GMRES are then performed for the preconditioned system $s(A)Ax = s(A)b$. This procedure is outlined below.

Algorithm: Polynomial Preconditioned GMRES

1. *Restart:*

   Compute current residual vector $r := b - Ax$.

2. *Adaptive GMRES run:*

   Run m_1 steps of GMRES for solving $Ad = r$. Update x by $x := x + d$. Get eigenvalue estimates from the eigenvalues of the Hessenberg matrix.

3. *Compute new polynomial.* Refine H from previous hull H and new eigenvalue estimates. Get new best polynomial s_k.

4. *Polynomial Iteration:*

   - Compute the current residual vector $r = b - Ax$.
   - Run m_2 steps of GMRES applied to $s_k(A)Ad = s_k(A)r$. Update x by $x := x + d$.
   - Test for convergence. If solution converged then stop else goto 1.

We now discuss some of the advantages and disadvantages of polynomial preconditionings, starting with the advantages. The main attraction of polynomial preconditioning is that the only operations involving the matrix are products with vectors. As a result parallelism of order N can be achieved for each step of the preconditioned conjugate gradient method. As a consequence of this, portability is facilitated because we only need to concen-

trate on optimizing matrix by vector multiplications and a few other basic kernels. A second advantage is that we need fewer dot products than with the nonpreconditioned conjugate gradient method to solve a linear system. The dot products can be bottlenecks for large number of processors but may not cause any difficulty otherwise. Thus, the main attraction of polynomial preconditioning is when the number or processors is very large. In this situation the usual incomplete LU factorization described in Section 3 will not be very helpful except for matrices with special structures. In many cases one may reorder the matrix to obtain a highly parallel LU solve, e.g., by using a red-black ordering as is described in Section 7. Polynomial preconditioning can be used when this is not possible or in combination with reordering techniques. Another important point is that polynomial preconditioning can be combined with a subsidiary relaxation-type preconditioning such as SSOR [3,37]. Finally, polynomial preconditionings can be very useful in solving some special linear systems such as complex linear systems arising from the Helmhotz equation [45] and those arising from the biharmonic equations[132].

The main disadvantage of polynomial preconditionings is their poor performance on sequential machines or parallel machines with small number of processors. In the numerical experiments reported in [5] on matrices with sizes ranging from $N = 434$ to $N = 2205$, polynomial preconditioning compares favorably only in a few cases with the parallel implementation of ILU preconditioning, on an Alliant FX-8. On the optimistic side, these numbers may look better if one thinks of the potential speed-ups that might be achieved on a machine with a much larger number of processors, since typically there is a parallelism of order N (the size of the matrix) for polynomial preconditioning but a parallelism of order $\sqrt{N}$ for ILU preconditioning. Unfortunately, theoretical comparisons of the two approaches are difficult. One such comparison was proposed by Axelsson [13] for the simple case of the diagonally scaled Neumann series approach [33], applied to a symmetric positive definite matrix. It was concluded that in this case polynomial preconditioning was rarely competitive with the nonpreconditioned conjugate gradient method. More precisely, polynomial preconditioning can outperform the standard conjugate gradient method only when the cost of the matrix by vector multiplication is less than half the cost of the other operations in a CG step. This is clearly a very stringent condition, that is likely to be satisfied only for machines with a very large number of processors, i.e., when the dot products start dominating the cost of a CG step. The model used in [13] is restricted to the simplest polynomial preconditioning, and it is not known whether a similar conclusion might be proved for the more sophisticated preconditioners.

A second disadvantage of polynomial preconditioning is that there are often alternatives that may do better. For example, for the case of a small number of processors, the parallel implementations of the standard ILU preconditioning discussed earlier will be difficult to outperform. For machines with large numbers of processors, reordering techniques have the potential of being superior to polynomial preconditioning, although extensive comparisons on such machines are still lacking.

It is often argued that one weakness of these methods is their requirements for eigenvalue estimates. In fact we found this to be a rather minor drawback because of the possibility of combining the process with an algorithm like GMRES or Arnoldi. We would also like

Matrix	N	ILU(0)	Polyn.
JPWH_991	991	0.30	0.31
ORSIRR_1	1030	0.74	2.88
ORSREG_1	2205	1.74	5.77
Sherman5	3312	3.50	4.20

Table 3: Times for solving four linear systems on an Alliant FX/8.

to mention that there has been very little work on polynomial preconditioning or Krylov subspace methods for highly non-normal matrices; see however the recent analysis in [120].

To give an idea of the performance of polynomial preconditionings as compared with ILU preconditioners, we show in Table 3 the time that it takes to solve a few linear systems with matrices from the Harwell-Boeing collection. The stopping criterion used was to stop as soon as the residual norm drops by a factor of 10^{-7}. This was done on an Alliant FX-8. The polynomial preconditioning used here was combined with a diagonal or block diagonal scaling. The degree of the polynomials used was 10. It is worth pointing out that the time for the first linear system ($JPWH\_991$) using the sequential version of ILU, was about 1.3 sec or roughly 4.3 times slower than with level scheduling. Similarly, for Sherman5 level scheduling was roughly 3.14 faster than with sequential forward and backward solves.

6 Block and other preconditioners

When vector machines first appeared, the preconditioned conjugate gradient method was just becoming popular as an iterative technique. As a result one of the first issues that this technique was facing concerned the vectorization of the preconditioner, the most popular of which was Meijerink and van der Vorst's incomplete Choleski factorization. The first idea in this context was to use cyclic reduction to solve the bidiagonal systems that arise during the lower triangular solves [63,60]. Van der Vorst [125] suggested replacing the inverse of each of the bidiagonal matrices by a simple Neumann series. This introduced a parallelism of order $n = \sqrt{N}$ for a problem originating from an $n \times n$ grid. The performance of these alternatives were, as expected, better than the sequential algorithm. However, it appeared that the vectorizable variant was not as reliable as the original ICCG. Moreover, often the speed-ups obtained were not satisfactory because of the short vector lengths that these methods involve.

Block incomplete factorizations were popularized independently by Axelsson [15] and by Concus, Golub, and Meurant [31]. See, however, a complete bibliography regarding these methods in [13], which indicates early work by Underwood dating back to 1976. Given a block tridiagonal matrix $A = tridiag(B_i^T, A_i, B_{i+1})$, the basic idea is derived from the standard Block Gaussian elimination process, in which A is factored as,

$$A = (D - L)D^{-1}(D - L)^T \tag{24}$$

where L is the negative of the strict lower triangular part of A, and D is a block diagonal matrix $D = diag(D_i)$, defined through the recurrence,

$$D_i = A_i - B_i D_{i-1}^{-1} B_i^T, \quad i = 2, 3, .., n. \tag{25}$$

The first observation made here [15,31] is that even though the blocks D_i are dense, a sparse approximation to them can easily be found. For example, one technique is to use a banded approximation Γ_i to the inverse D_i^{-1}. If we denote the resulting approximations to D_i by Δ_i, then the recurrence (25) becomes,

$$\Delta_i = A_i - B_i \Gamma_{i-1} B_i^T, \quad i = 2, 3, .., n. \tag{26}$$

Then the preconditioning matrix would be

$$M = (\Delta - L)\Delta^{-1}(\Delta - L)^T \tag{27}$$

Solving a linear system with M is not fully satisfactory on vector machines since the forward and backward solutions involve sequential banded solutions.

As block preconditioners emerged as contenders to the scalar incomplete factorizations [31], several variants were quickly derived and tested as vectorizable options [82,78,12,13]. The simplest such option proposed in [78] is to exploit the fact that both Δ_i and Γ_i are available, and therefore any multiplication by Δ_i^{-1} can be replaced by a multiplication by Γ_i. The "inverse-free" factorization proposed by Axelsson is to write M as

$$M = (I - L\Delta^{-1})\Delta(I - L\Delta^{-1})^T \tag{28}$$

Then, if Γ is block diagonal matrix approximating the inverse of Δ, the idea is to approximate the inverse of the factor $(I - L\Delta^{-1})$ by the so-called Euler expansion

$$(I - L\Gamma)^{-1} = \prod_{k=0}^{p} \left[I + (L\Gamma)^{2^k})\right] \tag{29}$$

where the integer p is carefully chosen. A detailed analysis of block preconditioners including the one just outlined was proposed by Axelsson and Polman [18].

There has been a number of extensions and related techniques proposed. For example, Meurant [80] suggested an alternative consisting of performing the LU factorization not only from top to bottom, but also from bottom to top, at the same time. The resulting decomposition is called the twisted incomplete factorization and is related to the WZ factorization [42]. The motivation here was to take advantage of multitasking on Cray machines. The gain in performance that was reported on a Cray X-MP was limited. Using a slightly different viewpoint, Rodrigue and Wolitzer [99] have argued for a form of incomplete cyclic reduction to derive incomplete factorizations of block tridiagonal matrices.

The more recent effort in preconditioning methods has been mainly in three directions. The first is in deriving preconditioners from domain decomposition techniques [65,79,26, 17]. The second direction is in hierarchical basis methods [4,19,51,134], whereby the finite

element basis is chosen in a such a manner as to yield well conditioned matrices. Finally, there is now increasing interest in methods that will take advantage of the special nature of three-dimensional problems [16,122]. Some early comparisons of domain decomposition preconditioners and standard block preconditioners reported in [79] seem to indicate that for large number of processors the former might be better.

We should also mention work related to domain decomposition methods which consists of multisplitting iterative techniques [88], and element-by-element (EBE) preconditioning techniques [84]. A recent collection of papers on domain decomposition methods may be found in the proceedings [48] and [23].

7 Reordering for parallelism

A convenient and simple way to achieve a high level of parallelization is to reorder the equations and the unknowns. For example, equations that originate from 5-point matrices can be reordered according to the red-black ordering to yield systems of the form,

$$\begin{pmatrix} D_1 & E \\ E^T & D_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{30}$$

where D_1 and D_2 are diagonal matrices. The triangular system to be solved at each step of incomplete ICCG are highly parallelizable due to the structure of the above matrix. However, one might be suspicious that the rate of convergence of the algorithm is not as good as that for the original ordering. Experience with model problems suggest that the deterioration of the convergence rate rarely exceeds a factor of two [114,71]. However, for more realistic and difficult problems convergence behavior for reordered matrices may be unpredictable [116]. As was mentioned in Section 4, in order to avoid bottlenecks in the forward and backward sweeps, it is desirable to have a matrix with a block structure in which each diagonal block is diagonal. We refer to this structure as Diagonal Diagonal Block (DDB) structure. The L matrix will then have the form of Figure 1, where each L_i is a diagonal matrix. There are many ways of achieving such structures; see, e.g., [71]. For matrices whose adjacency graphs are planar, a simple strategy is to start by setting up the level structure of the graph. This consists of using a Cuthill-Mc Kee ordering starting from some initial point; see e.g., [34]. The levels are then colored alternately in red and black. The next step is to group a certain number of levels of the same color in the same set. The number of levels chosen may be varied and will depend on the desired size of the diagonal blocks. For example, maximum parallelism may be achieved by putting all the red levels together in one set and all the black levels in another set. For matrices with property A, this will lead to the usual red-black ordering. Since each level set may consist of connected vertices, coloring may again be needed within each set. This technique was suggested in [71]. Interestingly, experiments for model problems reveal that the number of iterations needed for convergence does not vary too much as the block size increases, deteriorating by a factor of two in the worst case.

More generally, multicoloring is a general term that refers to reordering the unknowns to obtain a matrix that has the DDB structure mentioned above. Several authors have

considered multicoloring techniques and the various effects on convergence of either the preconditioned conjugate gradient or the underlying relaxation technique [1,86,2,39,97,96].

For matrices having property A, once the matrix is reordered in the form (30), an alternative to using PCG on the corresponding system directly is to solve the reduced system which involves only the black unknowns, namely,

$$(D_1 - ED_2^{-1}E^T)x_2 = b_1 - ED_2^{-1}b_2. \tag{31}$$

The above system is again a sparse linear system involving only half of the unknowns. Moreover, setting up the preconditioner is an inexpensive and fully parallel process even for unstructured problems [20]. The resulting system can again be preconditioned by incomplete Choleski factorization with level scheduling as described before. However, as revealed by experiments on various model problems, an excellent alternative is to use the diagonal of the reduced matrix as a preconditioner. This was observed for both well sctuctured [75] and unstructured problems[20]. In [20] it was found that for the reduced system, ICCG with level scheduling outperformed diagonal preconditioning, only for very large problems because the overhead in setting up the level structure and the preconditioner is difficult to offset given that diagonal scaling performs so well. A detailed analysis by Elman and Golub [40] on model problems indicate that it is often more effective to solve the reduced system by iterative methods than the original system, a fact which was observed empirically [38,36]. The reduced system approach appeared to be the best option on vector or multivector processors in experiments reported in many papers. It also has excellent potential for machines with a large number of processors of SIMD type. Its only limitation is that it does not generalize to matrices that do not have property A.

8 Conclusion

We have presented an overview of numerical techniques to solve realistic large linear systems by Krylov subspace methods on supercomputers. For machines with a small number of processors, the standard preconditionings can be efficiently implemented and they often constitute the most efficient approach. The advantage of this over using less conventional techniques is that these preconditioners are reliable and their behavior is well understood from experience on standard scalar machines. For machines with a very large number of processors, these techniques will not be sufficient to achieve satisfactory speed-ups. We have discussed two possible alternatives for this case. The first consists of using polynomial preconditioning and the second consists of reordering the equations by multi-coloring techniques. Although these alternatives offer a good potential, they are not sufficiently well understood for general problems. For example, reordering affects the convergence rate of the conjugate gradient method in a way that is difficult to predict for general sparse problems, while polynomial preconditioning may not be competitive against rival techniques such as the unpreconditioning conjugate gradient method or the multi-colored incomplete Choleski/LU.

These observations suggest that it is crucial that researchers experiment with existing massively parallel machines in order to better understand the effects that are difficult to

predict from theory, such as impact of communication costs, rates of convergence of new algorithms, ways of mapping the data, etc..

Research in the direction of domain decomposition and hierarchical basis preconditioners, two natural ideas for parallel processing, is very active. It is interesting to observe that the successful ideas in parallel numerical methods have often been derived from existing techniques that are either adapted or slightly modified. This is the case of domain decomposition methods which have been extensively used in structural analysis. The search for parallelism has forced researchers to take a second look at many old techniques sometimes resulting in remarkable success. Ortega and Voigt concluded in [91] that there has been very few "truly parallel algorithms" invented, as opposed to modifications of existing algorithms. This has been even more pronounced for iterative methods, perhaps because there are many known algorithms that do offer a large degree of parallelism or that can be simply modified to improve their parallelism. However, a common difficulty is that those iterative methods that are intrinsically highly parallel, such as the Jacobi iteration, often converge much more slowly than those with a more sequential nature, such as the Gauss-Seidel iteration.

We have not discussed issues related to software development and hardware. Although the trend towards parallelism is clear, the question as to which type (s) of architecture will prevail in the long term remains difficult to answer. A consequence is that in the near term at least, developers of numerical software will face challenging problems. They must not only select and develop different algorithms for different machines but also acquire a good understanding of the machines and the software that runs them in order to be able to produce reasonable performance. Moreover, porting a program from one machine to another usually requires substantial reprogramming effort which may involve changing languages, completely remapping the data, and extensive testing to tune the code. Given the variety of machines that are available these efforts are increasingly viewed as intolerably time consuming and non cost-effective. Thus, the need for better programming environments and languages for parallel machines is becoming critical and as iterative methods are gaining importance, they should be considered from the perspective of these other important issues.

References

[1] L. Adams. M-step preconditioned conjugate gradient methods. *SIAM J. Sci. Stat. Comp*, 6:452–463, 1985.

[2] L. Adams and H. Jordan. Is SOR color-blind? *SIAM J. Sci. Stat. Comp*, 6:490–506, 1985.

[3] L. M. Adams. *Iterative algorithms for large sparse linear systems on parallel computers.* PhD thesis, University of Virginia, Applied Mathematics, Charlottsville, VA 22904, 1982. Also available as NASA Contractor Report 166027.

[4] L. M. Adams and E. G. Ong. A comparison of preconditioners for GMRES on parallel computers. In A. Noor, editor, *Parallel Computations and their Impact on Mechanics*, pages 171–186, Amer. Soc. Mech. Engr., Dec. 1987.

[5] E. C. Anderson. *Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations.* Technical Report 805, CSRD, Univ. of Illinois, Urbana, Illinois, 1988. MS Thesis.

[6] E. C. Anderson and Y. Saad. *Solving sparse triangular systems on parallel computers.* Technical Report 794, University of Illinois, CSRD, Urbana, Illinois, 1988.

[7] W. E. Arnoldi. The principle of minimized iteration in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951.

[8] S. F. Ashby. *Polynomial Preconditioning for Conjugate Gradient Methods.* PhD thesis, Computer Science Dept., University of Illinois, Urbana, Illinois, 1987. Available as Technical Report 1355.

[9] C. C. Ashcraft and R. G. Grimes. On vectorizing incomplete factorization and SSOR preconditioners. *SIAM J. Sci. Stat. Comput.*, 9:122–151, 1988.

[10] O. Axelsson. Conjugate gradient type-methods for unsymmetric and inconsistent systems of linear equations. *Lin. Alg. Appl.*, 29:1–16, 1980.

[11] O. Axelsson. A generalized conjugate gradient, least squares method. *Num. Math.*, 51:209–227, 1987.

[12] O. Axelsson. Incomplete block matrix factorization preconditionings. The ultimate answer? *J. Comp. Appl. Math.*, 17:3–18, 1985.

[13] O. Axelsson. A survey of preconditionied iterative methods for linear systems of algebraic equations. *BIT*, 25:166–187, 1985.

[14] O. Axelsson. *A survey of vectorizable preconditioning iterative methods for large scale finite element matrix problems.* Technical Report CNA-190, Center for Numerical Analysis, University of Texas, Austin, Texas, 1984.

[15] O. Axelsson, S. Brinkkemper, and V. P. Ill'n. On some versions of incomplete block-matrix factorization iterative methods. *Lin. Alg. and its Appl.*, 58:3–15, 1984.

[16] O. Axelsson and V. Eijkhout. *Vectorizable Preconditioners for Elliptic Difference Equations in Three Space Dimensions.* Technical Report -, University of Nijmegen, Dept. of Math., Nijmegen, The Netherlands, 1988.

[17] O. Axelsson and B. Polman. *Block preconditioning and domain decomposition methods.* Technical Report 8807, Department of Mathematics, Catholic University, Toernooiveld, Nijmegen, The Netherlands, 1988.

[18] O. Axelsson and B. Polman. On approximate factorization methods for block matrices suitable for vector and parallel processors. *Lin. Alg. and its Appl.*, 77:3–26, 1986.

[19] O. Axelsson and P. S. Vassilevski. *Algebraic multilevel preconditioning methods, I.* Technical Report 8811, Department of Mathematics, Catholic University, Toernooiveld, Nijmegen, The Netherlands, 1988.

[20] C. L. Baucom. *Reduced systems and the preconditioned conjugate gradient method on a multiprocessor.* Technical Report, CSRD, University of Illinois, Urbana, Illinois, 1988.

[21] D. Baxter, J. Saltz, M. H. Schultz, and S. C. Eisenstat. *Preconditioned Krylov solvers and methods for runtime loop parallelization.* Technical Report 655, Computer Science, Yale University, New Haven, CT, 1988.

[22] D. Baxter, J. Saltz, M. H. Schultz, S. C. Eisenstat, and K. Crowley. *An experimental study of methods for parallel preconditioned Krylov methods.* Technical Report 629, Computer Science, Yale University, New Haven, CT, 1988.

[23] J. Bramble, T. F. Chan, R. Glowinski, and O. Widlund. *Second international symposium on domain decomposition methods.* SIAM, Philadelphia, 1989.

[24] P. N. Brown. *A theoretical comparison of the Arnoldi and GMRES algorithms.* Technical Report UCRL-98630, Lawrence Livermore Nat. Lab., Livermore, California, 1988.

[25] O. A. Mc Bryan. *The Connection Machine: PDE solution on 65,536 processors.* Technical Report LA-UR-86-4219, Los Alamos National Lab, Los Alamos, New Mexico, 1986.

[26] T. F. Chan. Analysis of preconditioners for domain decomposition. *SIAM J. Num. Anal.*, 24:382–390, 1987.

[27] T. F. Chan, C. C. Kuo, and C. Tong. *Parallel Elliptic Preconditioners: Fourier Analysis and Performance Evaluation.* Technical Report 88-22, Computational and Applied Mathematics, UCLA, 1988.

[28] R. Chandra. *Conjugate Gradient Methods for Partial Differential Equations.* PhD thesis, Yale University, Computer Science Dept., New Haven, CT. 06520, 1978.

[29] A. Chronopoulos. *A class of parallel iterative methods implemented on multiprocessors.* PhD thesis, University of Illinois, Computer Science Dept., Urbana, Illinois, 1987.

[30] P. Concus and G. H. Golub. A generalized conjugate gradient method for nonsymmetric systems of linear equations. In R. Glowinski and J. L. Lions, editors, *Computing Methods in Applied Sciences and Engineering*, pages 56–65, Springer Verlag, New York, 1976.

[31] P. Concus, G. H. Golub, and G. Meurant. Block preconditioning for the conjugate gradient method. *SIAM J. Sci. and Stat. Comp.*, 6:309–332, 1985.

[32] P. Concus, G. H. Golub, and D. P. O'Leary. A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. In James R. Bunch and Donald J. Rose, editors, *Sparse Matrix Computations*, pages 309–332, Academic Press, New York, 1976.

[33] P. F. Dubois, A. Greenbaum, and G. H. Rodrigue. Approximating the inverse of a matrix for use on iterative algorithms on vectors processors. *Computing*, 22:257–268, 1979.

[34] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[35] I. S. Duff, R. G. Grimes, J. G. Lewis, and W. G. Jr. Poole. Sparse matrix test problems. *SIGNUM newsletter, ACM*, 17:22–, 1982.

[36] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. Block preconditioned conjugate-gradient-like methods for numerical reservoir simulation. In *Proceedings of the SPE 1985 reservoir simulation symposium*, pages 397–405, Society of Petroleum Engineers of AIME, Richardson, TX, 1985. paper number 13534.

[37] S. C. Eisenstat, J. M. Ortega, and C. T. Vaughan. *Efficient polynomial preconditioning for the conjugate gradient method.* Technical Report RM-88-14, Dept. of Appl. Math., University of Virginia, Charlottesville, Virginia, 1988.

[38] H. C. Elman. *Iterative Methods for Large Sparse Nonsymmetric Systems of Linear Equations.* PhD thesis, Yale University, Computer Science Dpt., New Haven, CT., 1982.

[39] H. C. Elman and E. Agron. *Ordering techniques for the precondiotioning conjugate gradient method on parallel computers.* Technical Report UMIACS-TR-88-53, UMIACS, University of Maryland, College Park, Maryland, 1988.

[40] H. C. Elman and G. H. Golub. *Iterative Methods for cyclically reduced non-self-adjoint linear systems.* Technical Report CS-TR-2145, Dept. of Computer Science, University of Maryland, College Park, Maryland, 1988.

[41] H. C. Elman, Y. Saad, and P. Saylor. A hybrid Chebyshev Krylov subspace algorithm for solving nonsymmetric systems of linear equations. *SIAM J. Sci. Stat. Comp.*, 7:840–855, 1986.

[42] D. J. Evans. *Parallel Processing Systems, an advanced course.* Cambridge University Press, New York, 1982.

[43] B. Fischer and L. Reichel. A stable Richardson iteration method for complex linear systems. *Numer. Math.*, 54:225–241, 1988.

[44] R. Fletcher. Conjugate gradient methods for indefinite systems. In G.A. Watson, editor, *Proceedings of the Dundee Biennal Conference on Numerical Analysis 1974*, pages 73–89, University of Dundee,Scotland, Springer Verlag, New York, 1975.

[45] R. Freund. *On conjugate gradient type methods and polynomial preconditioners for a class of complex non-Hermitian matrices.* Technical Report 88-44, RIACS, NASA Ames Research Center, Moffett field, California, 1989.

[46] R. Freund. On polynomial approximation to $f_a(z) = (z-a)^{-1}$ with complex a and some applications to certain non-Hermitian matrices. *Approximation Theory and its Applications*, 1989.

[47] K. Gallivan, W. Jalby, and U. Meier. The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. *SIAM J. Sci. Stat. Comp.*, 8:1079–1084, 1987.

[48] R. Glowinski, G. H. Golub, G. A. Meurant, and J. Periaux. *First International Symposium on Domain Decomposition Methods for Partial Differential Equations.* SIAM, Philadelphia, 1988.

[49] A. Greenbaum and G. H. Rodrigue. *The incomplete Choleski conjugate gradient for the STAR (5-point) operator.* Technical Report UCID 17574, Lawrence Livermore National Lab., Livermore, California, 1977.

[50] A. Greenbaum. *Solving Triangular Linear Systems Using FORTRAN with Parallel Extensions on the NYU Ultracomputer Prototype.* Technical Report 99, Courant Institute, New York University, New York, NY, 1986.

[51] A. Greenbaum, C. Li, and H.Z. Chao. *Comparison of linear system solvers applied to diffusion type finite element equations.* Technical Report , Courant Institute, New York University, New York, NY, 1987.

[52] R. G. Grimes, J. G. Lewis, and H. D. Simon. *The implementation of a block Lanczos algorithm with reorthogonalization methods.* Technical Report ETA-TR-91, Boeing Computer Services, Seattle, WA, 1988.

[53] A. L. Hageman and D. M. Young. *Applied Iterative Methods.* Academic Press, New York, 1981.

[54] S. Hammond. *Efficient ICCG on a shared memory multiprocessor.* Technical Report, RIACS, NASA Ames research center, Moffett field CA., 1989. in preparation.

[55] S. W. Hammond and K. H. Law. Architecture and operation of a systolic engine for finite element computations. *Computers and structures*, 30:365–374, 1988.

[56] W. Jalby, U. Meier, and A. Sameh. *The behavior of conjugate gradient algorithms on a multivector processor with a hierarchical memory.* Technical Report 607, University of Illinois, CSRD, 1985.

[57] K. C. Jea and D. M. Young. Generalized conjugate gradient acceleration of nonsymmetrizable iterative methods. *Lin. Alg. Appl.*, 34:159–194, 1980.

[58] O. G. Johnson, C. A. Micchelli, and G. Paul. Polynomial preconditionings for conjugate gradient calculations. *SIAM J Numer. Anal.*, 20:362–376, 1983.

[59] S. L. Johnsson. Highly concurrent algorithms for solving linear systems of equations. In G. N. Birkhoff and A. Schoenstadt, editors, *Elliptic problem solvers II, Proceedings of the elliptic problem solvers conference, Monterey CA., Jan 10-12 1983*, pages 105–126, Academic Press, 1983.

[60] T. L. Jordan. Conjugate gradient preconditioners for vector and parallel processors. In G. N. Birkhoff and A. Schoenstadt, editors, *Elliptic problem solvers II, Proceedings of the elliptic problem solvers conference, Monterey CA., Jan 10-12 1983*, pages 127–139, Academic Press, 1983.

[61] T. L. Jordan. A guide to parallel computation and some CRAY-1 experiences. In Garry Rodrigue, editor, *Parallel Computations*, pages 1–50, Academic Press, 1982.

[62] T. I. Karush, N. K. Madsen, and G. H. Rodrigue. *Matrix Multiplication by Diagonals on Vector/Parallel Processors.* Technical Report UCUD, Lawrence Livermore National Lab., Livermore, CA, 1975.

[63] D. Kershaw. Solution of single tridiagonal systems and vectorization of the ICCG algorithm on the CRAY-1. In Garry Rodrigue, editor, *Parallel Computations*, pages 85–89, Academic Press, 1982.

[64] D. E. Keyes and W. D. Gropp. A comparison of domain decomposition techniques for elliptic partial differential equations. *SIAM J. Sci. Stat. Comp.*, 8:s166–s202, 1987.

[65] D. E. Keyes and W.D. Gropp. Domain decomposition techniques for nonsymmetric systems of elliptic boundary value problems: examples from cfd. In J. Bramble, T. F. Chan, R. Glowinski, and O. Widlund, editors, *Proceedings of the second international symposium on domain decomposition methods*, SIAM, Philadelphia, 1989.

[66] A. Koniges. *Parallel Processing of a preconditioned biconjugate gradient algorithm in CRAY supercomputers.* Technical Report -, Lawrence Livermore National Lab, Livermore, CA, 1987.

[67] D. J. Kuck, E. S. Davidson, D. L. Lawrie, and A. H. Sameh. Parallel supercomputing today and the CEDAR approach. *Science*, 231:967–974, 1986.

[68] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. of Res. NBS*, 49:33–53, 1952.

[69] J. G. Lewis and H. D. Simon. The impact of hardware scatter-gather on sparse Gaussian elimination. *SIAM J. Stat. Sci. Comp.*, 9:304–311, 1988.

[70] J. G. Lewis and H. D. Simon. *Numerical experience with the Spectral transformation Lanczos.* Technical Report MM-TR-16, Boeing Computer Services, Seattle, WA, 1984.

[71] A. Lichnewski. Some vector and parallel implementations for preconditioned gradient algorithms. In J. Kowalik, editor, *Proceedings of the NATO workshop on high speed computations*, pages 343–359, 1984.

[72] T. A. Manteuffel. Adaptive procedure for estimation of parameter for the nonsymmetric Tchebychev iteration. *Numer. Math.*, 28:187–208, 1978.

[73] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 34:473–497, 1980.

[74] T. A. Manteuffel. The Tchebychev iteration for nonsymmetric linear systems. *Numer. Math.*, 28:307–327, 1977.

[75] U. Meier and A. Sameh. The behavior of conjugate gradient algorithms on a multivector processor with a hierarchical memory. *Journal of Computational and Applied Mathematics*, 24:, 1988.

[76] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix. *Math. Comp.*, 31(137):148–162, 1977.

[77] R. Melhem. Solution of linear systems with striped sparse matrices. *Parallel Computing*, 6:165–184, 1988.

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

[79] G. Meurant. Domain decomposition versus block preconditioning. In R. Glowinski, G. H. Golub, G. Meurant, and J. Periaux, editors, *Proceedings of the first International Symposium on Domain Decomposition Methods for Partial Differential Equations*, SIAM, 1987.

[80] G. Meurant. Multitasking the conjugate gradient method on the CRAY X-MP/48. *Parallel Computing*, 5:267–280, 1987.

[81] G. Meurant. *Numerical experiments for the preconditioned conjugate gradient method on the CRAY X-MP/2.* Technical Report LBL-18023, Lawrence Berkeley Lab, Berkeley, California, 1984.

[82] G. Meurant. Vector preconditioning for the conjugate gradient on the CRAY-I and cdc CYBER 205. In J.L. Lions and R. Glowinski, editors, *Proceedings of the 6-th International Conference on Computing Methods in Engineering and Applied Sciences, Versailles, France, Dec. 12-16 1984*, page , INRIA, North-Holland, 1985.

[83] N. Munksgaard. Solving sparse symmetric sets of linear equations by preconditioned conjugate gradient method. *ACM. Trans. for Math Software*, 6:206–219, 1980.

[84] B. Nour-Omid and B. N. Parlett. Element preconditioning using splitting techniques. *SIAM J. on Sci. Stat. Comput.*, 6:761–770, 1985.

[85] D. O'Leary. The block conjugate gradient algorithm and related methods. *Lin. Alg. Appl.*, 29:243–322, 1980.

[86] D. O'Leary. Ordering schemes for parallel processing of certain mesh problems. *SIAM J. Sci. Stat. Comp.*, 5:620–632, 1984.

[87] D. O'Leary. Parallel implementations of the block conjugate gradient algorithm. *Parallel Computing*, 5:127–140, 1987.

[88] D. O'Leary and R. White. Multi-splittings of matrices and parallel solution of linear systems. *SIAM J. on Alg. Disc. Meth.*, 1:–, 1986.

[89] T. C. Oppe and D. R. Kincaid. The performance of ITPACK on vector computers for solving large sparse linear systems arising in sample oil reservoir simulation problems. *Communications in applied numerical methods*, 2:1–7, 1986.

[90] J. M. Ortega. *Introduction to parallel and vector solution of linear systems.* Plenum Press, New York, 1988.

[91] J. M. Ortega and R.G. Voigt. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27:149–240, 1985.

[92] C. C. Paige and M.A. Saunders. An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Software*, 8:43–71, 1982.

[93] V. L. Peterson. Impact of computers on aerodynamics research and development. *Proceedings of the IEEE*, 72:68–79, 1984.

[94] B. Philippe and Y. Saad. Solving large sparse eigenvalue problems on supercomputers. In *Proceedings of International Workshop on Parallel Algorithms and Architectures, Bonas, France Oct. 3-6 1988*, North-Holland, 1989.

[95] S. J. Polak, C. Den Heijer, W.H. A. Schilders, and P. Markowich. Semiconductor device modelling from the numerical point of view. *Int. J. Numer. Meth. Eng.*, 24:763–838, 1987.

[96] E. L. Poole. *Multi-color Incomplete Choleski Conjugate Gradient Methods for Vector Computers.* Technical Report 178117, NASA Langley research, Hampton, VA., 1986.

[97] E. L Poole and J. M. Ortega. Mullticolor ICCG methods for vector computers. *SIAM J. Numer. Anal.*, 24:1394–1418, 1987.

[98] G. Radicati and Y. Robert. *Vector and parallel CG-like algorithmns for sparse nonsymmetric linear systems.* Technical Report RR 681, IMAG, Univ. of Grenoble, France, Grenoble, France, Oct. 1987.

[99] G. Rodrigue and D. Wolitzer. Preconditioning by incomplete block cyclic reduction. *Math. Comp.*, 42:549–565, 1984.

[100] H. Rutishauser. Theory of gradient methods. In *Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-Adjoint Boundary Value Problems*, pages 24–49, Institute of Applied Mathematics, Zurich, Birkhauser Verlag, Basel-Stuttgart, 1959.

[101] Y. Saad. Iterative solution of indefinite symmetric systems by methods using orthogonal polynomials over two disjoint intervals. *SIAM J. on Numerical Analysis*, 20:784–811, 1983.

[102] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of Computation*, 37:105–126, 1981.

[103] Y. Saad. The Lanczos biorthogonalization algorithm and other oblique projection methods for solving large unsymmetric systems. *SIAM J. Numer. Anal.*, 19:470–484, 1982.

[104] Y. Saad. Least squares polynomials in the complex plane and their use for solving sparse nonsymmetric linear systems. *SIAM J. Num. Anal.*, 24:155–169, 1987.

[105] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Stat. Sci. Comput.*, 6:865–881, 1985.

[106] Y. Saad, A. Sameh, and P. Saylor. Solving elliptic difference equations on a linear array of processors. *SIAM J. on Sci. Stat. Comput.*, 6:1049–1063, 1985.

[107] Y. Saad and A. H. Sameh. A parallel block Stiefel method for solving positive definite systems. In M. H. Schultz, editor, *Proc. Elliptic Problem Solver Conf.*, pages 405–12, Academic Press, 1980.

[108] Y. Saad and M. H. Schultz. Conjugate gradient-like algorithms for solving nonsymmetric linear systems. *Mathematics of Computation*, 44(170):417–424, 1985.

[109] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[110] Y. Saad and M. H. Schultz. *Parallel Implementations of Preconditioned Conjugate Gradient Methods.* Research Report 425, Dept Computer Science, Yale University, 1985.

[111] J. H. Saltz. *Automated Problem Scheduling and Reduction of Synchronization Delay Effects.* Technical Report 87-22, ICASE, Hampton, VA, 1987.

[112] M. A. Saunders, H. D. Simon, and E. L. Yip. Two conjugate-gradient type methods for unsymmetric linear equations. *SIAM J. Num. Anal.*, 25(4):927–940, 1988.

[113] P. E. Saylor and D. C. Smolarski. Computing the roots of complex ortogonal kernel polynomials. *SIAM J. Sci. Stat. Comp.*, 9:1–13, 1988.

[114] R. Schreiber and W. P. Tang. Vectorizing the conjugate gradient method. In *Proc. Symposium CYBER 205 Applications*, Denver, Colorado, 1982.

[115] M. K. Seager. *Parallelizing conjugate gradient for the CRAY X-MP.* Technical Report, Lawrence Livermore National Lab, Livermore, CA, 1984.

[116] H.D. Simon. Incomplete LU preconditioners for conjugate gradient type iterative methods. In *Proceedings of the SPE 1985 reservoir simulation symposium*, pages 302–306, Society of Petroleum Engineers of AIME, Dallas, TX, 1988. Paper number 13533.

[117] D. C. Smolarski and P. E. Saylor. An optimum iterative method for solving any linear system with a square matrix. *BIT*, 28:163–178, 1988.

[118] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Scient. Stat. Comp.*, 10(1):36–52, 1989.

[119] E. L. Stiefel. Kernel polynomials in linear algebra and their applications. *U.S. NBS Applied Math. Series*, 49:1–24, 1958.

[120] L. N. Trefethen. *Approximation Theory and Numerical Linear Algebra.* Technical Report Numerical Analysis Report 88-7, Massachussetts Institute of Technology, MA, USA, 1988.

[121] H. A. van der Vorst. *High performance preconditioning.* Technical Report 88-54, Delft University of Technology, Faculty of Technical Mathematics and Informatics, 1988.

[122] H. A. van der Vorst. ICCG and related methods for 3-D problems on vector computers. In D. Truhlar, editor, *Workshop on Practical Iterative Methods for Large Scale Computations, Minneapolis MN., Oct. 23-25 1988*, Computer Physics Communication, vol. 53, 1989.

[123] H. A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on parallel and parallel computers. *Par. Comp.*, 5:303–311, 1987.

[124] H. A. van der Vorst. The performance of FORTRAN implementations for preconditioned conjugate gradient methods on vector computers. *Parallel Computing*, 3:49–58, 1986.

[125] H. A. van der Vorst. A vectorizable version of some ICCG methods. *SIAM J. Stat. and Sci. Comp.*, 3:350–356, 1982.

[126] R. S. Varga. *Matrix Iterative Analysis.* Prentice Hall, Englewood Cliffs, New Jersey, 1962.

[127] H. F. Walker. Implementation of the gmres method using householder transformations. *SIAM J. Sci. Stat. Comput.*, 9:152–163, 1988.

[128] J. W. Watts-III. A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. *Society of Petroleum Engineer Journal*, 21:345–353, 1981.

[129] O. Widlund. A Lanczos method for a class of non-symmetric systems of linear equations. *SIAM Journal on Numerical Analysis*, 15:801–812, 1978.

[130] O. Wing and J. W. Huang. A computation model of parallel solution of linear equations. *IEEE Transactions on Computers*, C-29:632–638, 1980.

[131] Y. S. Wong. Solving large elliptic difference equations on CYBER 205. *Parallel Computing*, 6:195–207, 1988.

[132] Y. S. Wong and H. Jiang. *Approximate polynomial preconditioning applied to biharmonic equations on vector computers.* Technical Report ICOMP-87-5, NASA Lewis Research Center, Inst. for Computational Mechanics in Propulsion, Cleveland, Ohio, 1987.

[133] D. M. Young, T.C. Oppe, D. R. Kincaid, and L. J. Hayes. *On the use of vector computers for solving large sparse linear systems.* Technical Report CNA-199, Center for Numerical Analysis, Univ. of Texas at Austin, Austin, Texas, 1985.

[134] H. Yserentant. On the multi-level splitting of finite element spaces. *Numer. Math.*, 49:379–412, 1986.