SIAM J. SCI. STAT. COMPUT.
Vol. 10, No. 6, pp. 1174-1185, November 1989

HIGH PERFORMANCE PRECONDITIONING*

HENK A. VAN DER VORST†

Abstract. The discretization of second-order elliptic partial differential equations over three-dimensional rectangular regions, in general, leads to very large sparse linear systems. Because of their huge order and their sparseness, these systems can only be solved by iterative methods using powerful computers, e.g., vector supercomputers. Most of those methods are only attractive when used in combination with a so-called preconditioning matrix. Unfortunately, the more effective preconditioners, such as successive over-relaxation and incomplete decompositions, do not perform very well on most vector computers if used in a straightforward manner. In this paper it is shown how a rather high performance can be achieved for these preconditioners.

Key words. preconditioning, ICCG, block bidiagonal systems, vector computers, parallel computers

AMS(MOS) subject classifications. 65F05, 65F10, 65F50, 65N20

1. Introduction. Among the methods that have been used successfully for solving the large linear systems that stem from discretized three-dimensional partial differential equations (PDEs) are:

—Preconditioned conjugate gradients for the symmetric positive definite case [18].

—Projection-type methods for the general nonsymmetric case, such as, e.g., Orthodir, Orthomin, and Orthores [33], GMRES [21], LSQR [20], Bi-CG [7], and CGS [24].

—Preconditioned adaptive Chebychev iteration [16], [26].

The success of these methods usually depends quite critically on the choice of preconditioner that has the effect of improving the spectrum of the linear operator involved.

Vector supercomputers, and parallel versions of them, play an important role in solving these large systems. Since the basic components of the methods, mentioned above, are operations like vector updates, innerproducts, and sparse matrix vector products, they are in principle quite easily implementable on vector computers with a high performance code as a result. This, however, is in general not the case for the most popular preconditioning techniques. Since they usually contain recurrence relations, they typically do not vectorize or parallelize very well in a standard way and therefore many authors have suggested better vectorizable alternatives [9], [11]-[15], [17].

In this contribution we will consider vector and parallel aspects of the so-called incomplete LU decompositions [18], [19] that are among the most frequently used preconditioners. Since the problems with respect to vector or parallel execution are similar for both the symmetric and the nonsymmetric case and since the preconditioners do not depend on the basic iteration method, we will study, for the sake of simplicity, the most simple case, namely, the preconditioned conjugate gradients algorithm for symmetric positive definite systems. Note, however, that most of the ideas outlined here can be carried over immediately to the other situations mentioned earlier (including successive over-relaxation (SSOR) and Gauss-Seidel preconditioning or smoothing).

We will show that, especially in the three-dimensional case, the preconditioned algorithm can be implemented by vectorized code that takes about as much CPU-time per iteration step as the unpreconditioned version. As preconditioning usually reduces

* Received by the editors October 5, 1988; accepted for publication (in revised form) February 10, 1989.

† Faculty of Technical Mathematics and Informatics, Delft University of Technology, P.O. Box 356, 2600 AJ Delft, the Netherlands.

significantly the number of iteration steps, this reduction then immediately translates to a similar reduction in CPU-time. With respect to vector computers we have in mind machines such as, e.g., CRAY X-MP, CYBER 205, ETA10, and for parallelism we aim, for the moment, for systems with only a moderate number of powerful processors, such as, e.g., CRAY X-MP/4, CRAY Y-MP/8, ETA10-G, or IBM 3090-600E (VF).

2. A sketch of the problem and a realistic goal. Let the linear system $Ax = b$ be the result of standard seven-point finite difference discretization of an elliptic second-order PDE, such as, e.g., $(au'_x)'_x + (bu'_y)'_y + (cu'_z)'_z + du = f$, over a rectangular grid imposed over a rectangular region.

To avoid questions with respect to mathematical aspects, including convergence properties and existence of the incomplete decompositions, we will further assume that A is a symmetric positive definite M-matrix.

Such a discretization matrix has a very regular structure when we number the unknowns, corresponding to the gridpoints, lexicographically such that the pointer in x-direction runs fastest, followed by the pointer in y-direction. Gridpoints with equal y and z coordinates define a line in the grid, those with equal z define a plane.

The matrix of the resulting linear system can be viewed as a block tridiagonal matrix of which the off diagonal elements are diagonal matrices, defining the relations between the unknowns in successive planes. The diagonal blocks by themselves can be seen as block tridiagonal matrices, corresponding to two-dimensional problems over each z-plane and, finally, the diagonal blocks of the latter are tridiagonal matrices defining the relations between unknowns along a line.

For the solution of this linear system of order N we consider the preconditioned conjugate gradient method. It is well known that the unpreconditioned cg method vectorizes very well, but the preconditioned version is much more troublesome. In fact, it would be very attractive to find a preconditioner that satisfies the following requirements:

(1) It should lead to a substantial reduction in iteration steps;

(2) The amount of work per iteration step should be roughly the same as for unpreconditioned cg; and

(3) The computational speed for each iteration step should have the same order of magnitude as the unpreconditioned cg process.

We will show in this paper that these requirements can all be met, for systems with the above described structure, for some well-known vector supercomputers.

For comparison purposes we start with a discussion of the unpreconditioned cg method. This method has the following form [8]:

```
$x_0$ is an initial guess for the solution $x$ of $Ax = b$
$r_0 = b - Ax_0$,    $p_0 = r_0$
for $i = 0, 1, 2, \cdots$ until the method is stopped:
```

$$\alpha_i = \frac{(r_i, r_i)}{(p_i, Ap_i)},$$

$$x_{i+1} = x_i + \alpha_i p_i,$$

$$r_{i+1} = r_i - \alpha_i A p_i,$$

$$\beta_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)},$$

$$p_{i+1} = r_{i+1} + \beta_i p_i.$$

Per iteration step the cg-method comprises the following computational work: two innerproducts; three vector-updates; and one matrix-vector multiplication. All these basic operations perform very well on vector machines such as, e.g., the CRAY X-MP and CYBER 205. In three-dimensional problems almost any discretized linear system is rather large, i.e., it leads to vector lengths close to those for which limit performances for the basic operations are practically observed.

For situations in which the matrix was represented by seven (long) diagonals, rather than a blockwise structure, we observed computational speeds of about 180 Mflops on a single processor CRAY X-MP and about 105 Mflops on a two-pipe CYBER 205.

3. The (M)ICCG algorithm.

A very successful preconditioning matrix is obtained by constructing a so-called incomplete triangular decomposition of the matrix A. When applied to positive definite systems, the correspondingly preconditioned cg method is usually referred to as ICCG [18], [19] or MICCG [10]. In this section we will discuss computational aspects of these methods as well as their mutual relation.

For ICCG we construct an incomplete Choleski decomposition of A. If A is written as $A = L + \operatorname{diag}(A) + L^T$, where L is the strict lower triangular part of A, then the standard incomplete Choleski decomposition can be represented by

$$K = (L + D)D^{-1}(D + L^T), \tag{3.1}$$

in which D is a diagonal matrix. Its elements follow from the relation

$$\operatorname{diag}(K) = \operatorname{diag}(A). \tag{3.2}$$

When the diagonal elements of A are denoted by $a_{i,1}$ (where i refers to the row) and the other three nonzero diagonals in L^T by $a_{i,2}$, $a_{i,3}$, and $a_{i,4}$, respectively, then the typical form for the relation of the elements d_i of D is

$$d_i = a_{i,1} - \frac{a_{i-1,2}^2}{d_{i-1}} - \frac{a_{i-nx,3}^2}{d_{i-nx}} - \frac{a_{i-nxny,4}^2}{d_{i-nxny}} \tag{3.3}$$

(nx and $nxny$ denote the distances of the respective diagonals to the main diagonal; nx = number of gridpoints in the x-direction, $nxny = nx * ny$ and ny = number of gridpoints in the y-direction).

From the discussion in § 4 it will be clear that the construction of D (for ICCG as well as MICCG), by relations such as (3.3), can be vectorized.

MICCG differs from ICCG by a different construction of D. It follows from the requirement

$$\operatorname{Rowsum}(K) = \operatorname{Rowsum}(A). \tag{3.4}$$

Since the preconditioning matrix in MICCG has the same form as the one in ICCG, the vectorization problems in both methods are exactly the same. These problems will be discussed in § 4.

We will now comment briefly on the convergence behavior of ICCG and MICCG. It has been observed that for (academic) model problems MICCG often converges much faster than ICCG. However, in many realistic (industrial) problems ICCG turns out to be the winning method. To study the relation between ICCG and MICCG, we include a parameter α in the relation (3.4) so that the relations for the d_i typically take the form

$$
\begin{aligned}
d_i &= a_{i,1} - a_{i-1,2} * (a_{i-1,2} + \alpha a_{i-1,3} + \alpha a_{i-1,4})/d_{i-1} \\
&\quad - a_{i-nxny,3} * (\alpha a_{i-nx,2} + a_{i-nx,3} + \alpha a_{i-nx,4})/d_{i-nx} \\
&\quad - a_{i-nxny,4} * (\alpha a_{i-nxny,2} + \alpha a_{i-nxny,3} + a_{i-nxny,4})/d_{i-nxny}.
\end{aligned} \tag{3.5}
$$

For $\alpha = 1$ the standard MICCG-method results and note that for $\alpha = 0$ we obtain the ICCG-method. By varying α we can smoothly shift from ICCG to MICCG. There is another way of looking at this. MICCG can be seen as a modification to ICCG in which the approximation errors in the incomplete decomposition are lumped to the diagonal. With α we can control the degree of lumping [2], [3].

If we plot (as in Fig. 1) the number of iterations for the preconditioned cg process as a function of α, then typically this number decreases for α increasing from zero (ICCG) toward one (MICCG), but it almost always suddenly jumps upward for α very close to one. For the earlier-mentioned model-problems the jump may be only small, but in most cases it is there.

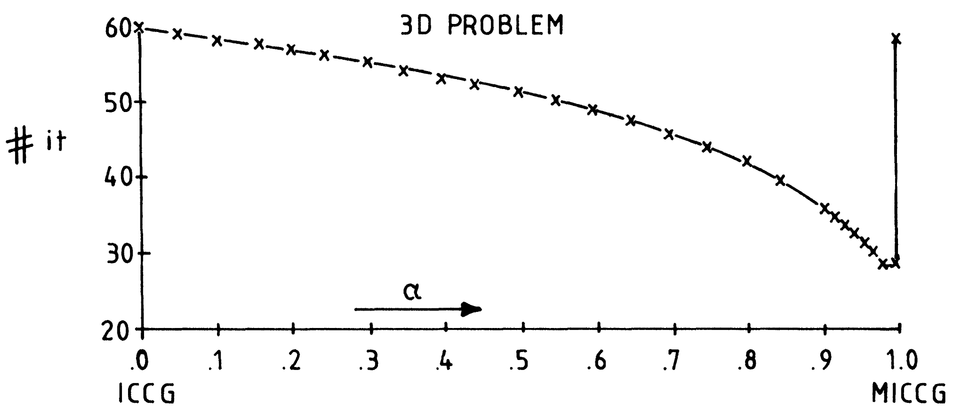

FIG. 1. *Three-dimensional problem.*

Similar experiences have been observed in, e.g., [1], [25], and [32]. The explanation given in [32] is that values of α very close to one lead to an early loss of orthogonality in the cg-iterations that spoils the convergence behavior. Many experiments carried out so far indicate that $\alpha = .95$ is a safe cure for this problem, and it is observed that MICCG ($\alpha = .95$) leads to a very substantial reduction in the number of iteration steps with respect to the unpreconditioned cg-process.

We will now consider some general implementation aspects of the (M)ICCG-process.

Once the diagonal D of the incomplete decomposition (3.1) has been computed, it is of advantage to scale the matrix A:

$$\tilde{A} = D^{-1/2} A D^{-1/2}. \tag{3.6}$$

The incomplete decomposition of $\tilde{A} = \tilde{L} + \operatorname{diag}(\tilde{A}) + \tilde{L}^T$ can now be represented as

$$\tilde{K} = (\tilde{L} + I)(I + \tilde{L}^T), \tag{3.7}$$

so that scaling has led to a computationally cheaper preconditioner. The earlier described cg algorithm is now applied to the preconditioned system

$$(\tilde{L}+I)^{-1}\tilde{A}(I+\tilde{L}^T)^{-1}\tilde{x}=(\tilde{L}+I)^{-1}\tilde{b}, \tag{3.8}$$

where $\tilde{b}=D^{-1/2}b$.

From the solution $\tilde{x}$ of (3.8) the solution x of $Ax=b$ is obtained via

$$x=D^{-1/2}(I+\tilde{L}^T)^{-1}\tilde{x}. \tag{3.9}$$

Note that the matrices $(\tilde{L}+I)^{-1}$ and $(I+\tilde{L}^T)^{-1}$ need not be formed explicitly (nor would it be advisable to do so), since these matrices occur throughout the cg process only in matrix-vector products. For example, the vector $y=(\tilde{L}+I)^{-1}z$ is, for given vector z, computed by solving y from the sparse lower triangular system

$$(\tilde{L}+I)y=z. \tag{3.10}$$

The form of the linear system (3.8) suggests that the amount of computational work per cg-iteration is considerably increased by preconditioning, since we have to solve two sparse triangular systems in each step. The amount of work for this is roughly the same as for the computation of the matrix vector product Ay.

Eisenstat [6] has shown that much of the extra work can be avoided by noting that the vector $(\tilde{L}+I)^{-1}\tilde{A}(I+\tilde{L}^T)^{-1}\tilde{p}_i$ (see the scheme in § 2) can be written as

$$(\tilde{L}+I)^{-1}\tilde{A}(I+\tilde{L}^T)^{-1}\tilde{p}_i=(I+\tilde{L})^{-1}[\tilde{p}_i+(\operatorname{diag}(\tilde{A})-2I)\tilde{t}]+\tilde{t}, \tag{3.11}$$

in which $\tilde{t}=(I+\tilde{L}^T)^{-1}\tilde{p}_i$.

Hence the expression at the right-hand side of (3.11) can be computed at about the cost of solving two triangular systems. For our class of three-dimensional model problems this comes down to $\approx$15 N flops (N is the order of the system), whereas Ap_i in the unpreconditioned process takes $\approx$13 N flops. From this we conclude that (M)ICCG can be carried out with about the same number of flops per iteration as unpreconditioned cg. Note that we now have fulfilled two of the requirements put on the preconditioner in § 2. In the next section we will consider the third requirement, on the performance of the preconditioned method that is a problem since, unfortunately, we have in fact replaced the fastest part of the unpreconditioned algorithm (i.e., Ap_i) by the more or less problematic solution of triangular systems.

4. The vectorized solution of the triangular systems. In this section we will discuss several approaches for achieving high computational speeds for the solution process of y from $(\tilde{L}+I)y=z$ (the same approaches hold, of course, for the upper triangular system).

Of course the computational work can be vectorized, or be carried out in parallel, when the order of the unknowns is changed, e.g., by red-black ordering. In general this leads to a different preconditioner, so it essentially changes the method, and experiments conducted by Duff and Meurant [34] indicate that for most reordering schemes the number of iterations doubles (or worse). However, for the given preconditioner there exist ordering schemes that do not change the preconditioner and that lead to exactly the same iterates as does the lexicographical ordering. The most obvious one, or at least the easiest programmable one, is the *plane-diagonal-wise ordering* that we will explain now.

If we indicate the unknowns, as well as the matrix coefficients by three indices, i, j, k, so that i refers to the index of the corresponding gridpoint in the x-direction, j to the index in the y-direction, and k in the z-direction, then the typical statement for the computation of the elements of the vector y takes the form

$$y_{i,j,k}=z_{i,j,k}-a_{i-1,j,k,2}\,y_{i-1,j,k}-a_{i,j-1,k,3}\,y_{i,j-1,k}-a_{i,j,k-1,4}\,y_{i,j,k-1} \tag{4.1}$$

($a_{i,j,k,2}$ represents the matrix coefficient referring to the next unknown in the x-direction and so forth).

First we accumulate all contributions to y from the previous, already computed, plane in the grid:

$$\bar{z}_{i,j,k}=z_{i,j,k}-a_{i,j,k-1,4}\,y_{i,j,k-1}\quad \text{for all } i, j, k \text{ fixed.} \tag{4.2}$$

This is a vector operation. Now we define grid-diagonal m as the collection of gridpoints (i, j, k), for which $i+j=m$, and k is fixed.

Then we note that the unknowns corresponding to grid-diagonal m in a plane (k fixed) can be computed independently (or in vector mode) from the unknowns corresponding to the (previous) grid-diagonal $m-1$.

This approach leads to vector operations of length $O(n_x)$, n_x being the number of gridpoints in the x-direction. Therefore the vector lengths are rather short. This is not too bad for computers with a low $n_{1/2}$-value for this type of vector operation such as, e.g., the CRAY X-MP.

If the unknowns are not explicitly reordered in memory, then the diagonal-wise computation is typically carried out with stride n_x-1 and this may lead to memory bank conflicts. For computers such as the CYBER 205 a stride $\neq 1$ must be avoided anyway.

As an illustration we present some results obtained for the CRAY-2 for two-dimensional problems. For three-dimensional problems for which the number of unknowns in x and y direction are similar to this two-dimensional case, we expect a little bit better Mflops-rates. We report here on the two-dimensional situation only since we have not actually tested this algorithm in the three-dimensional case. For a 100 * 100 grid the CPU-times and computational speeds for a certain PDE are:

	CPU-time	Mflops-rate
cg (with diagonal scaling)	0.76 sec	82 Mflops
MICCG ($\alpha=.95$, Eisenstat-trick, unknowns explicitly reordered to avoid strides)	0.23 sec	53 Mflops

For a 200 * 200 grid (and a different PDE) we obtain

	CPU-time	Mflops-rate
cg (with diagonal scaling)	4.13 sec	76.5 Mflops
MICCG (as above)	0.84 sec	58 Mflops

The interesting thing about these figures is not so much the CPU-times, since they depend on the PDE and the required solution, but the Mflops-rates, that are independent of those factors. We recall that the performance of unpreconditioned cg (that is equal to the performance for cg on the diagonally scaled system) is mainly determined by highly vectorizable components. The matrix vector product is the most CPU-time consuming one of these. Precisely this operation in our implementation of MICCG has been replaced by the solution of the more cumbersome triangular systems. Nevertheless, the performance (per iteration) drops only from ~80 Mflops to ~60 Mflops. For the CYBER 205 equally impressive results have been obtained. For computers for which strides are not so much a problem (e.g., CRAY X-MP) the performance of MICCG with either explicit reordering or the original ordering does not differ so much, and in both cases is quite high (~100–120 Mflops).

In the three-dimensional situation there are more possibilities for vectorization. Instead of considering grid-diagonals we might as well consider hyperplanes, defined by all triples (i, j, k) having an equal sum. If we denote the hyperplane, formed by all gridpoints for which $i+j+k=m$ by H_m, then it is obvious from (4.1) that all unknowns corresponding to H_m can be computed independently from those corresponding to H_{m-1}.

This approach leads to rather long vector lengths $(O(N^{2/3}))$, but the problem here is that the unknowns are not located equally spaced in memory, so that normally indirect addressing is required to identify those unknowns. On some vector supercomputers indirect addressing sort of spoils the performances (CYBER 205, NEC SX-2), on others the performances drop to about half the performance for the constant stride situation (CRAY X-MP, FUJITSU VP200). This explains that the performance per iteration of MICCG ($\alpha=.95$, Eisenstat-trick, hyperplane ordering) is about half the performance of the iteration of the unpreconditioned cg method on a CRAY X-MP. For the FUJITSU VP200 and the HITACHI S810/20 this is about 30 percent [28]. Hence on these machines we loose a factor of about three in the performance of MICCG, compared with cg, due to indirect addressing. However, for some of these computers the performance of the hyperplane ordering can be significantly improved, as we will show in the next section.

5. Implementation aspects of the hyperplane ordering.

For some vector computers the performance reducing effects of indirect addressing, involved in the hyperplane ordering approach, can be largely reduced or even be undone by a careful implementation. To illustrate this we consider in more detail the implementation for the CYBER 205 (and ETA-10).

We will focus again on solving the lower triangular system (3.10), for which (4.1), with respect to the hyperplane ordering, is replaced by

```
(5.1)    for $l=4, 5, \cdots, n_x+n_y+n_z$ do
             (a)  $\{y_{ijk}=z_{ijk}-a_{ijk-14}y_{ijk-1}$, for all $(i, j, k)\in H_l\}$
             (b)  $\{y_{ijk}=y_{ijk}-a_{ij-1k3}y_{ij-1k}$, for all $(i, j, k)\in H_l\}$
             (c)  $\{y_{ijk}=y_{ijk}-a_{i-1jk2}y_{i-1jk}$, for all $(i, j, k)\in H_l\}$
         end l
```

The implementation for the three parts (a), (b), and (c) is quite similar and we will discuss only part (c). This part (c) may be written as

```
(5.2)    for $i=\max(2, l-n_y-n_z), \cdots, \min(n_x, l-2)$ do
             for $j=\max(1, l-i-n_z), \cdots, \min(n_y, l-i-1)$ do
                 $k=l-i-j$,
                 $y_{ijk}=y_{ijk}-a_{i-1jk2}y_{i-1jk}$
             end j
         end i
```

This double loop defines our ordering of the (i, j, k) over H_l for each value of l. The obvious way to implement (5.2) is to store y_{ijk} and a_{i-1jk2} in the order in which they are used over H_l. This has to be done only once at the beginning of the cg process and, since we apply Eisenstat's trick, there is no need to store the matrix elements both in lexicographical and hyperplane ordering.

Having the elements of y and $a_{,2}$ reordered, there is still indirect addressing required for the elements y_{i-1jk} that are stored correspondingly to the ordering for H_{l-1}. For the CYBER 205 this leads to the following scheme for (5.2):

(5.3)

(i) GATHER the required elements y_{i-1jk} into an array V_l, in the order in which they are required to update the y_{ijk} over H_l. The "gaps" in V_l, when H_l is larger than H_{l-1}, are left zero.
This amounts for the CYBER 205 to $\approx 1.4n_xn_yn_z$ clock cycles for all values of l together.

(ii) The elements of V_l are multiplied by the a_{i-1jk2}: $\approx \frac{1}{2}n_xn_yn_z$ clock cycles for all l together.

(iii) The result from step (ii) is subtracted element-wise from the y_{ijk}: $\approx \frac{1}{2}n_xn_yn_z$ clock cycles for all l together.

Hence the total execution time for step (5.1)(c) is, apart from startup overhead, equal to $\approx 2.4n_xn_yn_z$ clock cycles. This leads to a total time for solving both triangular subsystems of roughly $14.4n_xn_yn_z$ clock cycles. If we compare this with the time for computing Ay, where A has the structure as in § 1 (with $\operatorname{diag}(A) = I$) that is about $6n_xn_yn_z$ clock cycles, then we see that the preconditioning part has been vectorized, but it is still about 2.5 times as expensive as, e.g., the matrix vector multiplication.

However, the two steps (i) and (ii) in scheme (5.3) can be implemented much more efficiently on the CYBER 205 (and the ETA-10), when using one single sparse vector instruction that comprises the following actions:

(5.4)

(1) EXPAND the input data vectors under control of their associated bit vectors; the "holes" are filled up with a specified broadcast value: zero for addition/subtraction and one for multiplication.

(2) Perform the required floating point operation on the expanded input vectors.

(3) Carry out a specified logical operation on the input bit vectors that gives an output bit vector.

(4) COMPRESS the output data vector of step (2) under control of the output bit vector of step (3).

In our situation we associate with the expanded y_{i-1jk}-vector V_l a bit vector that has value one in positions where the i-index of the V_l element is less than n_x, and we insert a zero when this index equals n_x (since such an element does not contribute to the y_{ijk} over H_l). The output bit vector in step (3) of (5.4) should contain exclusively $1-s$, so that the COMPRESS in step (4) is not effective.

With such a sparse vector instruction the CPU time for step (c) in (5.1), for all l together, is now reduced to $\approx n_xn_yn_z$ clock cycles. Hence the total CPU-time for solving both triangular factors is about $6n_xn_yn_z$ clock cycles.

This means that solving *both* triangular systems takes about the same CPU-time as the matrix vector multiplication Ay itself, a quite remarkable result.

Note that the construction of the (modified) incomplete decomposition of A can be vectorized along the same lines as sketched above, which then also leads to a process that runs at near optimal speed on the CYBER 205. For more details on implementations for the CYBER 205, see [22]. Our analysis shows that the preconditioned cg process can be carried out, at least on the CYBER 205, with almost the same computational speed as the scaled cg-process. This implies that we also have succeeded in realizing our third requirement made in § 2. We believe that similar high performances for the hyperplane approach can be achieved for other vector computers too, possibly with the help of assembler coding.

As an illustration we present for a certain problem the CPU-times for different methods on a CYBER 205 (the 34 * 34 * 34 problem has been taken from [28, problem 2]:

Method	CPU-time
straightforward ICCG(0)	11.2 sec
cg with diagonal scaled A	1.7 sec
MICCG ($\alpha = .95$, Eisenstat-trick, hyperplane-approach)	0.36 sec

For results, obtained for some of these approaches on Japanese supercomputers, see [27] and [28].

6. Parallelism in the solution of the triangular systems. In [29] it is shown how the idea of twisted factorization of a triangular matrix [4], [30] can be extended to the construction of a twisted incomplete factorization of block tridiagonal linear systems of the type described in § 2. This approach can be used for the construction of a high performance preconditioner on, e.g., a CRAY X-MP/2. This has actually been carried out by constructing an incomplete factorization that is twisted only in the z-direction and optimizing the recursions over each of the two-dimensional-subblocks by the grid diagonal-wise approach discussed in § 4.

This approach has as an interesting side effect that it seems to reduce the number of (preconditioned) cg-steps slightly. Since it can be implemented using also the earlier explained Eisenstat-trick [6], a speed-up of slightly more than two could be obtained on a two-processor CRAY X-MP in comparison with the code on one processor described in § 4.

A further increase in the degree of parallelism, suitable for more parallel/vector processors, is obtained by following the twisted approach also for the two-dimensional subblocks or by neglecting certain off-diagonal blocks in the incomplete twisted factorization, similar to the approach for the regular incomplete decomposition as suggested by Seager [23]. If the twisted factorization for the two-dimensional subblocks is done in a straightforward manner as in [29], then the remaining recursions are no longer vectorizable. However it can be done in such a way that nothing of the vectorizability (obtained by the diagonal-wise approach in two dimensions) is lost. This can be achieved by numbering the gridpoints for each two-dimensional subgrid as follows:

$$\begin{bmatrix} & & & 22 & 14 & 8 & 4 & 2 \\ 19 & & & & & 16 & 10 & 6 \\ 11 & 17 & & & & & 18 & 12 \\ 5 & 9 & 15 & & & & & 20 \\ 1 & 3 & 7 & 13 & 21 & & & \end{bmatrix}.$$

It is clear that the computation for the unknowns on the diagonals with odd numbered gridpoints can be carried out in parallel with the unknowns on the diagonals with the even numbered gridpoints (except for the diagonal right in the middle that must wait for completion until all the other diagonals have finished). Indeed, the computation over each of the diagonals themselves in this ordering scheme is completely vectorizable, as we have already seen in § 4.

This approach can be used to further optimize the performance on the CRAY X-MP/2, since the two-dimensional block in the middle of the twisted factorization, when carried out as in the beginning of the section, degraded the performance slightly. Using the twisted diagonal approach the computation can be done entirely in parallel. Obviously, this approach can also be used to carry out the computation almost entirely in four parallel (and each of them vectorizable) parts.

For computers that allow for an efficient implementation of the hyperplane approach, additional parallelism may also be obtained by constructing a twisted factorization over the hyperplane blocks. The parallel variants and their interesting effects on the convergence behavior are currently the subject of further research.

7. Some other ideas and remarks. Other preconditioners such as, e.g., incomplete decompositions with more fill-in [19], or polynomial preconditioners, can also be vectorized [1], but they cannot be implemented at similar low costs as the standard (modified) incomplete Choleski preconditioners. Moreover, they usually require more memory to store the triangular factors.

When the matrix of the linear system has a less regular structure, as in finite-element discretizations, it is often possible to construct an incomplete decomposition and to use some of the earlier mentioned projection type methods to solve the preconditioned linear system. Though this can be done using Eisenstat's trick, the main problem is now that the recursions cannot easily be vectorized as in the regular structured case. One approach to vectorize at least part of the computation is to use a truncated Neumann series approach for subblocks of the matrix, as suggested in [5] and [31].

We will illustrate the effects of this approach by applying the technique to the regular structured block tridiagonal systems described in § 2. If we partition the matrix in subblocks so that the subblocks along the diagonal correspond to the recursions over a gridline in the x-direction, then the recursions over the y- and z-directions can be vectorized trivially. The recursions for the remaining subblocks can be solved formally by writing

$$y_{jk} = (I + B_{jk})^{-1} z_{jk}. \tag{7.1}$$

B_{jk} represents the off-diagonal matrix coefficients, in the (j, k)th subblock, referring to previous gridpoints on the gridline (with fixed j and k). The vectors y_{jk} and z_{jk} represent the parts of y and z over the (j, k)th gridline. The matrix B_{jk} in this case consists of only one nonzero diagonal, but in general situations the B_{jk} can, of course, be more complicated.

In the truncated Neumann-series approach $(I + B_{jk})^{-1}$ is approximated and y_{jk} is replaced by $\hat{y}_{jk}$:

$$\hat{y}_{jk} = I - B_{jk}(I - B_{jk}) z_{jk}. \tag{7.2}$$

This is more expensive in terms of flops, but it offers more possibilities with respect to vectorization. The approach is more successful when $\|B_{jk}\|_\infty$ gets smaller and becomes much smaller than one. For an analysis, see [31].

Example. For a nicely structured block tridiagonal three-dimensional system, coming from a 35 * 35 * 35-grid, standard ICCG(0) took 60 iterations to get the desired result, while the above sketched truncated Neumann series approach required 61 iterations.

On a CRAY X-MP the standard ICCG(0)-approach took 3.1 CPU-seconds (~30 Mflops), and the Neumann series approach took 0.91 seconds (~115 Mflops). This would be quite an improvement if there was no other vectorization technique available (as there is for this type of matrix).

For the CYBER 205 the figures are:

standard ICCG(0): 11.2 sec (~8 Mflops)
the Neumann approach: 2.0 sec (~50 Mflops).

Although matrices with this nice structure lend themselves to better performances, as shown in § 4, the example indicates that the Neumann series approach might be useful in situations where the other approaches fail. This will also be the subject of further research.

REFERENCES

[1] C. Ashcraft and R. Grimes, *On vectorizing incomplete factorizations and* SSOR *preconditioners*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 122-151.

[2] O. Axelsson and G. Lindskog, *On the eigenvalue distribution of a class of preconditioning methods*, Numer. Math., 48 (1986), pp. 479-498.

[3] ———, *On the rate of convergence of the preconditioned conjugate gradient method*, Numer. Math., 48 (1986), pp. 499-523.

[4] I. Babuška, *Numerical stability in problems of linear algebra*, SIAM J. Numer. Anal., 9 (1972), pp. 53-77.

[5] P. F. Dubois, A. Greenbaum, and G. H. Rodrigue, *Approximating the inverse of a matrix for use in iterative algorithms on vector processors*, Computing, 22 (1979), pp. 257-268.

[6] S. C. Eisenstat, *Efficient implementation of a class of preconditioned conjugate gradient methods*, SIAM J. Sci. Statist. Comput., 2 (1981), pp. 1-4.

[7] R. Fletcher, *Conjugate Gradient Methods for Indefinite Systems*, Lecture Notes in Mathematics 506, Springer-Verlag, Berlin, Heidelberg, New York, 1976, pp. 73-89.

[8] G. H. Golub and C. F. van Loan, *Matrix Computations*, North Oxford Academic, Oxford, 1983.

[9] A. Greenbaum and G. Rodrigue, *The incomplete Choleski conjugate gradient method for the* STAR *(5-point operator)*, Report UCID-17574, Lawrence Livermore Laboratory, Livermore, CA, 1977.

[10] I. Gustafsson, *A class of 1st order factorization methods*, BIT, 18 (1978), pp. 142-156.

[11] O. G. Johnson and G. Paul, *Optimal parametrized incomplete inverse preconditioning for conjugate gradient calculations*, Res. Report RC 8644, IBM-Research Center, Yorktown Heights, NY, 1981.

[12] T. L. Jordan, *A guide to parallel computation and some* CRAY-1 *experiences*, in Parallel Computations, G. Rodrigue, ed., Academic Press, New York, 1982.

[13] D. S. Kershaw, *The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations*, J. Comput. Phys., 26 (1978), pp. 43-65.

[14] ———, *The solution of single linear tridiagonal systems and vectorisation of the* ICCG *algorithm on the* CRAY-1, Report UCID-19085, Lawrence Livermore Laboratory, Livermore, CA, 1981.

[15] J. R. Kightley and I. P. Jones, *A comparison of conjugate gradient preconditionings for three-dimensional problems on the* CRAY-1, Harwell Report CSS 162, AERE Harwell, UK, 1984.

[16] T. A. Manteuffel, *The Tchebychev iteration for nonsymmetric linear systems*, Numer. Math., 28 (1977), pp. 307-327.

[17] G. Meurant, *Numerical experiments for the preconditioned conjugate gradient method on the* CRAY X-MP/2, Report LBL-18023, University of California, Berkeley, CA, 1984.

[18] J. A. Meyerink 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 (1977), pp. 148-162.

[19] ———, *Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems*, J. Comput. Phys., 44 (1981), pp. 134-155.

[20] C. C. Paige and M. A. Saunders, LSQR: *An algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Software, 8 (1982), pp. 43-71.

[21] Y. Saad and M. H. Schultz, GMRES, *A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856-869.

[22] J. J. F. M. Schlichting and H. A. van der Vorst, *Solving* 3D *block bidiagonal linear systems on vector computers*, Report NM-R8819, CWI, Amsterdam, 1988.

[23] M. K. Seager, *Parallelizing conjugate gradient for the* CRAY X-MP, Parallel Computing, 3 (1986), pp. 35-47.

[24] P. Sonneveld, CGS, *a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 36-52.

[25] Y. USHIRO, *Vector computer version of* ICCG *method*, in: M. Mori, Parallel Numerical Computational Algorithms and Related Topics, RIMS Report 514, Kyoto University, 1984 (in Japanese).

[26] H. A. VAN DER VORST, *Iterative solution methods for certain sparse linear systems with non-symmetric matrix arising from* PDE-*problems*, J. Comput. Phys., 44 (1981), pp. 1-19.

[27] ———, (M)ICCG *for* 2D *problems on vector computers*, in Supercomputing, A. Lichnewsky and C. Saguez, eds., North-Holland, Amsterdam, 1988, pp. 321-334.

[28] ———, ICCG *and related methods for* 3D *problems on vector computers*, Report No. A-18, Kyoto University, Kyoto, Japan, 1987.

[29] ———, *Large tridiagonal and block tridiagonal linear systems on vector and parallel computers*, Parallel Comput., 5 (1987), pp. 45-54.

[30] ———, *Analysis of a parallel solution method for tridiagonal linear systems*, Parallel Comput., 5 (1987), pp. 303-311.

[31] ———, *A vectorizable variant of some* ICCG *methods*, SIAM J. Sci. Statist. Comput., 3 (1982), pp. 86-92.

[32] ———, *The effect of* MICCG *in the presence of rounding errors*, in progress.

[33] D. M. YOUNG AND K. C. JEA, *Generalized conjugate-gradient acceleration of nonsymmetrizable iterative methods*, Linear Algebra Appl., 34 (1980), pp. 159-194.

[34] I. S. DUFF AND G. A. MEURANT, *The effect of ordering on preconditioned conjugate gradient*, Report CSS 221, AERE Harwell, UK, 1988.