Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.J.F.M. Schlichting, H.A. van der Vorst

Solving 3D block bidiagonal linear systems
on vector computers

Department of Numerical Mathematics Report NM-R8819 December

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Solving 3D Block Bidiagonal Linear Systems on Vector Computers

J.J.F.M. Schlichting
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Control Data BV, P.O. Box 111, 2285 VL Rijswijk, The Netherlands

H.A. van der Vorst
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Technical University Delft, Faculty of Mathematics and Informatics,
Julianalaan 132, 2628 BL Delft, The Netherlands

Standard 7-point finite difference discretization of 2nd order pde's over a rectangular grid over a 3-dimensional block leads, with the usual lexicographical ordering of the gridpoints, to block tridiagonal linear systems. In many popular iterative methods for the solution of these systems triangular systems have to be solved which have a block bidiagonal structure. This is often recognized to be the major bottleneck on vector computers, with respect to the computational speed, when carried out in a straight-forward manner.

In this paper we will discuss different techniques for the vectorization of the solution of 3D-block bidiagonal systems. We will report on actually observed performances for the ICCG algorithm, for which these bidiagonal systems have the reputation to spoil the overall performance. The potentially most powerful of the vectorization techniques leads to long vector operations, at the cost, however, of strides and indirect addressing. Since the CYBER 205 is generally believed to stay behind in performance under such circumstances, we have chosen this machine to show in detail how these vectorization techniques can be implemented with almost equal performance as in the contiguous vector case. Our methods are directly applicable to the ETA-10 family of supercomputers, and may be adapted to other vector computers as well.

1980 mathematics Subject Classification (1985 Revision): 65F10, 65V05, 65W05.
1982 CR Categories: 5.14, 4.6.
Key Words & Phrases: block bidiagonal linear system, Cyber 205, ICCG algorithm, vectorization.
Note: This paper has been submitted for publication elsewhere.

1. A SKETCH OF THE PROBLEM.

If we discretize second order partial differential equations over a rectangular grid imposed over a block in R^3 by standard 7-point finite difference discretization then a block tridiagonal system $Ax = b$ results:

$$A = \begin{bmatrix} A_1 & G_1 & & & \\ D_2 & A_2 & G_2 & & \\ & & & & \\ & & & G_{n_z-1} & \\ & & D_{n_z} & A_{n_z} & \end{bmatrix} \tag{1.1}$$

D_k and G_k are diagonal matrices, n_z denotes the number of gridpoints (unknowns) in the z-direction. The dimension of the matrix blocks is $n_x n_y$, in which n_x, n_y denote the number of gridpoints in x,y-

direction, respectively. The elements of these blocks correspond to (x,y)-planes of the grid The matrices A_k themselves have a block tridiagonal structure:

$$A_k = \begin{bmatrix} A_{1k} & F_{1k} & & & \\ C_{2k} & A_{2k} & F_{2k} & & \\ & & & & \\ & & & F_{n_y-1k} & \\ & & C_{n_yk} & A_{n_yk} & \end{bmatrix} \tag{1.2}$$

C_{jk} and F_{jk} are diagonal matrices. All the blocks in A_k are of dimension n_y and they correspond to x-lines in a gridplane.
Finally, A_{jk} itself is a (point-) tridiagonal matrix:

$$A_{jk} = \begin{bmatrix} a_{1jk} & e_{1jk} & & & \\ b_{2jk} & a_{2jk} & e_{2jk} & & \\ & & & & \\ & & & e_{n_x-1jk} & \\ & & b_{n_xjk} & a_{n_xjk} & \end{bmatrix} \tag{1.3}$$

In many practical situations the linear system $Ax = b$ is solved by an iterative method. In most of these methods we have to solve, as part of the algorithm, triangular linear systems in which the triangular matrices have the same non-zero structure as the lower or upper diagonal part of A. As examples of these methods we mention SOR, SSOR, Stone's SIP-method, ICCG, preconditioned methods like GMRES, ORTHODIR, ORTHORES, ORTHOMIN and Chebychev-iteration and Multigrid with Gauss-Seidel or ILU Smoothing. In many cases a lower bidiagonal as well as an upper bidiagonal block system has to be solved in each iteration. Since the vectorization problems for each case are identical we will concentrate on the lower triangular case. If we denote the lower triangular block matrix by L, then typically we have to solve $Lz = r$ for z.

The elements of z and r will be denoted by z_{ijk} and r_{ijk} according to their corresponding gridpoints. The solution of $L_z = r$ leads to a recurrency of the following type:

$$z_{ijk} = r_{ijk} - b_{ijk}z_{i-1jk} - c_{ijk}z_{ij-1k} - d_{ijk}z_{ijk-1} \tag{1.4}$$

Since recurrence relations in general lead to reduced performances on vector computers it might seem that we are in trouble because of recurrencies in all three grid directions. In the next section we will discuss techniques to vectorize relations like (1.4).

2. Vectorization techniques.

2.1. Partial vectorization

A popular approach for improving the performance of (1.4) on vector computers is known as partial vectorization. Note that the recurrency in the k-index refers to z-values corresponding to the previous (i,j)-plane. Before starting the actual computation of the z-values for a given fixed index k, we can first compute the contribution $d_{ijk}z_{ijk-1}$ for all i and j. This is a vector operation of length n_xn_y. Similarly, the contribution $c_{ijk}z_{ij-1k}$ can be computed, for fixed j, for all i as a single vector operation.
Schematically, we obtain the following algorithm

$$
\begin{aligned}
&\text{for } k = 1,2,\ldots,n_z \text{ do}\\
&\text{(a)}\quad \{z_{ijk} = r_{ijk}z_{ijk-1} - 1, \text{ for } i = 1,\ldots,n_x, j = 1,\ldots,n_y\}\\
&\quad\ \text{for } j = 1,2,\ldots,n_y \text{ do}\\
&\text{(b)}\quad\ \{z_{ijk} = z_{ijk} - c_{ijk}z_{ij-1k}, \text{ for } i = 1,\ldots,n_x\}\\
&\text{(c)}\quad\ \text{for } i = 2,3,\ldots,n_x \text{ do } z_{ijk} = z_{ijk} - b_{ijk}z_{i-1jk}\\
&\quad\ \text{end } i\\
&\quad \text{end } j\\
&\text{end } k
\end{aligned}
\tag{2.1}
$$

(we assume that variables with an index 0 are equal to 0). For vector register machines it is often advantageous to split (2.1(a)) over the lines and combine it with (2.1(b)), because this saves the store operation for z in (2.1(a)) and the load in (2.1((b)).
Many compilers automatically replace the recursion (2.1(c)) by optimized code. Nevertheless, the computational speed for (2.1(c)) is often rather low as compared with the other operations and, in agreement with Amdahl's law, the overall performance will be low. We will take this partially vectorized algorithm (2.1) as the yardstick with which we compare the other approaches.

2.2 Plane-wise diagonal ordering

As in the 2D situation, the recursion over the i and j directions can be vectorized, for fixed values of k, by computing the unknowns successively over "diagonals": $i+j=$ constant. For more details on this see e.g., [1,4,5]. The recurrency in the k-direction can be vectorized as is shown in section 2.1. Though in some situations this approach leads to fairly high performances, the performance is often limited because of the relatively small vector lengths involved.

2.3 Hyperplane ordering

From the expression (4.1) we conclude that, as soon as all elements z_{ijk}, with $i+j+k = l-1$ for given l, are known then all z_{ijk} for which $i+j+k = l$ can be computed independently (i.e. in vector mode or in parallel).

The set of indices $\{(i,j,k) | i+j+k = l \text{ for fixed } l\}$ will be denoted by H_l and will be called a "diagonal hyperplane". All elements z_{ijk} corresponding to H_l can be computed in vector mode with vector lengths of $O((n_x n_y n_z)^{2/3})$, but the problem is that the vector elements are not located with equal strides in memory. Vectorization can be achieved using indirect addressing and then the success of the hyperplane approach depends on the ability of a given computer to handle indirect addressing efficiently.

When the index-triples over H_l are ordered appropriately then there are local regularities in the storage pattern. This may be exploited for some architectures. The CYBER 205 has the reputation of being only optimal for contiguous vectors. Therefore we believe it is a suitable target machine to study the effects of a careful implementation of the Hyperplane approach. We will show that this approach may lead to surprisingly high performances.

For experimental results for the approaches in 2.1, 2.2, and 2.3 on CRAY X-MP and the Japanese supercomputers, see [6,7].

3. Implementations of the hyperplane ordering for the Cyber 205

Since the solution algorithm of the block lower bidiagonal system can be vectorized similarly as the solution algorithm for the block upper bidiagonal system we will focus only on the lower bidiagonal case. With respect to the hyperplane ordering the algorithm (2.1) is replaced by

$$\begin{aligned}
&\text{for } l = 4, 5, \ldots, n_x + n_y + n_z \text{ do} \\
&\text{(a)} \quad \{z_{ijk} = r_{ijk} - d_{ijk} z_{ijk-1}, \text{ for all } (i,j,k) \in H_l\} \\
&\text{(b)} \quad \{z_{ijk} = z_{ijk} - c_{ijk} z_{ij-1k}, \text{ for all } (i,j,k) \in H_l\} \\
&\text{(c)} \quad \{z_{ijk} = z_{ijk} - b_{ijk} z_{i-1jk}, \text{ for all } (i,j,k) \in H_l\} \\
&\text{end } l
\end{aligned} \tag{3.1}$$

It is obvious that the parts (a), (b) and (c) in (3.1) are very similar and therefore we will restrict ourselves to only one of them, namely part (c). This part may be rewritten as

$$\begin{aligned}
&\text{for } i = \max(2, l - n_y - n_z), \ldots, \min(n_x, l-2) \text{ do} \\
&\quad \text{for } j = \max(1, l - i - n_z), \ldots, \min(n_y, l - i - 1) \text{ do} \\
&\quad\quad k = l - i - j \\
&\quad\quad z_{ijk} = z_{ijk} - b_{ijk} z_{i-1jk} \\
&\quad \text{end } j \\
&\text{end } i
\end{aligned} \tag{3.2}$$

Assuming that the elements z_{ijk} and b_{ijk} are stored in the standard FORTRAN lexicographical ordering, we need a GATHER operation for each of the input element sets $\{z_{ijk}\}$, $\{b_{ijk}\}$ and $\{z_{i-1jk}\}$ and a SCATTER operation for the results $\{z_{ijk}\}$ in (3.2). For all hyperplanes H_l together there are $(n_x - 1)n_y n_z$ elements involved in the update step (c). Ignoring for the moment all start-up overhead in the various vector operations, the execution times for the operations involved in (c) are approximately

1.4 clock cycles per result of a SCATTER or GATHER
0.5 clock cycles per result of a multiply or subtract

(for the Cyber 205 the length of a clockcycle is 20 nanoseconds).
Henceforth the total execution time for step (c) for all l together is given in first order approximation by

$$(4*1.4 + 2*0.5)(n_x - 1)*n_y*n_z = 6.6(n_x - 1)n_y n_z \text{ clock cycles.} \tag{3.3}$$

Note that the GATHER operation for the elements b_{ijk} needs to be carried out only once. For the next iterationstep, of which the solution of the block lower bidiagonal system is only a part, we can reuse the result of the GATHER operation on b_{ijk}. This reduces the execution time for step (c), for all l together, to

$$5.2(n_x - 1)n_y n_z \text{ clock cycles} \tag{3.4}$$

In order to further reduce the CPU time for step (c) (as well as for the other steps of course), the elements z_{ijk} are rearranged explicitly in memory according to the hyperplanes (the order *per* hyperplane is defined by (3.2)) and we denote the set of z_{ijk}, corresponding to one specific hyperplane H_l, by Z_l, likewise the set of b_{ijk} corresponding to H_l is denoted by B_l.

For each value of l the number of elements involved in step (c) is only marginally smaller than the number of index points in H_l. Therefore there is an advantage in modifying (3.2) to:

i. GATHER the elements of z required from H_{l-1} into an auxiliary array V_l (in the same order as the elements of B_l and Z_l. This amounts to $\approx 1.4\ n_x n_y n_z$ clock cycles for all l.
ii. Vector multiply V_l and B_l: $\approx 0.5\ n_x n_y n_z$ clock cycles for all l.
iii. Vector subtraction of Z_l and the result of V_l times B_l. This requires another $\approx 0.5\ n_x n_y n_z$ clock cycles and the results are now in the right order over H_l; hence the scatter operation is avoided.

} (3.5)

The total execution time for step (c), up to scheme (3.5), for all l is roughly given by

$$2.4\ n_x n_y n_z \text{ clock cycles.} \tag{3.6}$$

This result can still be further improved by using COMPRESS-EXPAND operations rather than GATHER-SCATTER operations. The COMPRESS-EXPAND vector operations on the CYBER 205 require approximately 0.5 clock cycle per element of the longest of the two vectors involved in the operation. The total length of the longest arrays is $n_x n_y n_z$, whereas the total length of the shortest arrays is $(n_x - 1) n_y n_z$, which is only little less in a relative sense.
This approach leads to the following implementation of step (c):

(i) EXPAND the Z_{l-1} to the vector W_l corresponding to the required order of Z_l (and B_l)
(ii) Vector multiply W_l and B_l.
(iii) Vector subtract the result in (ii) from Z_l.

} (3.7)

The total execution time for step (c) for all l is now reduced to

$$\approx 1.5\ n_x n_y n_z \text{ clock cycles} \tag{3.8}$$

Step (i) and (ii) in (3.7) can be combined on the CYBER 205 in one single "sparse vector" instruction which comprises the following operations.

(i^1) Expand the input data vectors under control of their associated bit vectors; the "holes" are filled up with a specified broadcast value: 0 for addition/subtraction and 1 for multiplication.
(ii^1) perform the required floating point operation on the expanded input vectors.
(iii^1) carry out a specified logical operation on the input bit vectors which gives an output bit vector.
(iv^1) COMPRESS the output data vector of step (ii^1) under control of the output bit vector of step (iii^1).

E.g., for the multiplication of B_l and the expanded Z_{l-1} we associate a bit vector that has value 1 in positions where the i-index of the Z-element is less that n_x and we insert a value 0 when an i-index of Z corresponding to H_{l-1} is equal to n_x (because they do not contribute to Z-values over H_l). The output bit vector should contain exclusively values 1, which is accomplished by a logical "or" operation (hence there is no compress in this situation).

With such a sparse vector instruction the CPU time for step (c), for all l together, is now reduced to

$$\approx n_x n_y n_z \text{ clock cycles.} \tag{3.9}$$

This is very close to the minimum number of clock cycles that is necessary for carrying out the same vector operations in step (c) on contiguous vectors. The only price we have paid (except for some additional overhead possibly) is that we process in all the operations $n_x n_y n_z$ elements instead of $(n_x - 1) n_y n_z$, as in (2.1), so that we are relatively very close to the minimum indeed. Combining this all together we note that the total $\approx 6 n_x n_y n_z$ floating operations required for the solution of the block

lower bidiagonal system $Lz = r$, as given by (1.4), can be carried out in $\approx 3n_x n_y n_z$ clockcycles of 20 *ns*. Hence the computational speed should be about 100 MFLOPS. For $n_x = n_y = n_z = 30$ we actually observed a speed of about 90 MFLOPS.

Note that this is *essentially the same speed* that we may expect for the computation of Ax, where A is as in section 1, if the computation is carried out in a diagonal-wise fashion (see, e.g., [4]).

Note also, however, that once we have rearranged the elements of z hyperplane wise in memory we cannot immediately compute, e.g., Az in the nice diagonal-wise way, but that we have to GATHER the z-elements first. We may, however, also apply the previously described techniques for the computation of Az in a hyperplane wise fashion. In the next section we will point out that in many relevant situations it is not necessary to compute the matrix vector operations like Az and solutions to lower/upper block bidiagonal systems both within the given iterative method.

4. Preconditioned linear systems and Eisenstat's trick

It is at present generally accepted that the speed of convergence of most iterative methods for the approximate solution of a linear system $Ax = b$ can be greatly improved by preconditioning. This means that the original system is replaced by some other system, like e.g., $K^{-1}Ax = K^{-1}b$, where K^{-1} is an approximation to the inverse of A. The inverse of K is in most situations not explicitly computed, but K is given in such a form that vectors like $w = K^{-1}v$ can be efficiently computed by solving w from $Kw = v$, for given v.

Many popular preconditioners, like e.g., Incomplete Choleski, ILU, MILU and SSOR can be written in the general form

$$K = L\,U, \tag{4.1}$$

in which L and U have the same sparsity structure as the lower triangular part and upper triangular part of A, respectively.

For matrices with the sparsity structure as described in section 1 this implies that we have to solve block bidiagonal systems as discussed in that same section. It has been mentioned already that this often leads to rather low computational speeds on vector supercomputers, in fact often so low that preconditioning did seem to be no longer attractive. As we have shown in section 3 these block bidiagonal systems can be solved with high computational speeds if one is willing to take the programming effort. But also then there still is a complication since we have to compute vectors Ap and $K^{-1}q$, and choosing an optimal data structure for either of them complicates the efficient computation of the other.

For the nonzero structure as in section 1, however, the preconditioning matrix K for Incomplete Choleski, ILU, MILU and SSOR can be written alternatively in the form

$$K = (L_s + D)D^{-1}(D + U_s), \tag{4.2}$$

where L_s and U_s are equal to the strictly lower triangular and strictly upper triangular part of A, respectively. Also for other non-zero structures of A it makes sense to select preconditioning matrices of the form (4.2), for details see e.g., [3]. Eisenstat [2] has shown how preconditioning in this case can be carried out very efficiently. The trick comes down to applying the iterative method to the explicitly preconditioned system:

$$\tilde{A}y \equiv D^{\frac{1}{2}}(L_s + D)^{-1} A (D + U_s)^{-1} D^{\frac{1}{2}} y = D^{\frac{1}{2}}(L_s + D)^{-1} b \tag{4.3}$$

with $x = (D + U_s)^{-1} D^{\frac{1}{2}} y$

For each matrix vector multiplication with $\tilde{A}$ and some given vector p it follows from the observation $A = L_s + diag(A) + U_s$ that

$$
\begin{aligned}
& D^{\frac{1}{2}}(l_s+D)^{-1}A(D+U_s)^{-1}D^{\frac{1}{2}}p = \\
& D^{\frac{1}{2}}(L_s+D)^{-1}(L_s+D+diag(A)-2D+D+U_s)(D+U_s)^{-1}D^{\frac{1}{2}}p \qquad (4.4) \\
& = D^{\frac{1}{2}}\left[(D+U_s)^{-1}+(L_s+D)^{-1}(diag(A)-2D)+(L_s+D)^{-1}\right]D^{\frac{1}{2}}p \\
& = D^{\frac{1}{2}}\left[t+(L_s+D)^{-1}(diag(A)-2D)t+q\right],
\end{aligned}
$$

with

$$q \equiv D^{\frac{1}{2}}p \text{ and } t \equiv (D+U_s)^{-1}q.$$

Hence, except for three diagonal matrix vector products (two with $D^{\frac{1}{2}}$, one with $diag(A)-2D$) the matrix vector product $\tilde{A}p$ can be completely reconstructed from the solution of the two block bidiagonal systems that had to be solved in the preconditioning process anyhow. We can do even better by scaling the given matrix A appropriately so that the preconditioning matrix for the scaled matrix $\bar{A}$ can be written as $\bar{K}=(\bar{L}_s+I)(I+\bar{U}_s)$. This makes the computational process much mor efficient since we avoid then all the operations with D and $D^{\frac{1}{2}}$ in (4.4). For $\bar{A}$ the matrices in (4.4) are to be replaced: D by I, L_s by $\bar{L}_s$ and U_s by $\bar{U}_s$.

We conclude that the preconditioned algorithm can be carried out with virtually the *same amount of flops* per iteration step as the unpreconditioned algorithm. Hence a reduction in the number of iteration steps immediately translates to about the same reduction in flops. But, what is more, we have seen in section 3 that the preconditioning itself can be carried out, at least for the CYBER 205, at the *same computational speed* as the matrix vector product with A itself. The important implication is that the *preconditioned* algorithm can be executed with about the same computational speed (MFLOPS-rate) *per iteration* as the *unpreconditioned* algorithm. Therefore any reduction in the number of iteration steps, due to the preconditioner, immediately translates to about the same reduction in CPU-time (which is as much as we could wish).

5. Examples

A given linear system $Ax=b$, with a symmetric positive definite matrix A, as in section 1, scaled such that $diag(A)=I$, was solved by the unpreconditioned conjugate gradient algorithm and by the MICCG algorithm (for details about our special tuning of MICCG, see [6]). Note that forcing $diag(A)=I$ is as good as we can do in the unpreconditioned process, since it corresponds to diagonal scaling preconditioning for CG for $Ax=b$ which is known to be quite effective (see [3, 7]).
For two different, but still modest problem sizes, we list for both methods the numbers of iterations, the MFLOPs-rates and the CPU-times (the CPU-times include the time required for stopping criteria, etcetera) for the CYBER 205.

1. $n_x = 20\ n_y = 20\ n_z = 20$

	diag. scaled CG	MICCG
number of iterations	100	19
MFLOPs-rate	104	69
CPU-time	0.180 sec	0.057 sec

2. $n_x = 35\ n_y = 35\ n_z = 35$

	diag. scaled CG	MICCG
number of iterations	175	30
MFLOPs-rate	100	91
CPU-time	1.733 sec	0.358 sec

Similar results, scaled of course with respect to different clock cycle lengths, have been obtained on an ETA-10-P computer.

The linear systems themselves will not be described here in more detail, since the only aspect that we want to show is the computational speed and this is the same for any linear system of the same dimensions. Furthermore, the numbers of iterations are only listed in order to show that a reduction in this number corresponds to roughly the same reduction in CPU-time (except for too modestly dimensioned systems). Hence the above listed numbers nicely confirm the conclusions made at the end of section 4.

REFERENCES

1. C. ASHCRAFT and R. GRIMES, On vectorizing incomplete factorizations and SSOR preconditioners, SIAM J. Sci. Stat. Comput., 9 (1), pp. 122-151, 1988.
2. S.C. EISENSTAT, Efficient implementation of a class of preconditioned conjugate gradient methods, SIAM J. Sci. Stat. Comput. 2, pp. 1-4, 1981.
3. J.A. MEIJERINK and H.A. VAN DER VORST, Guidelines for the Usage of Incomplete Decompositions in Solving Sets of Linear Equations as They Occur in Practical Problems, J. of Comp. Physics, 44 (1), pp. 134-155, 1981.
4. H.A. VAN DER VORST, The performance of FORTRAN implementations for preconditioned conjugate gradients on vector computers, Parallel Computing 3, pp. 49-58, 1986.
5. H.A. VAN DER VORST, (M) ICCG for 2D problems on vector computers, Report no. A-17, Data Processing Center, Kyoto University, 1986.
6. H.A. VAN DER VORST, High performance preconditioning, to appear in SIAM J. Sci. Stat. Comput.
7. H.A. VAN DER VORST, ICCG and related methods for 3D problems on vector computers, to appear in Computer Physics Communications.