Centrum voor Wiskunde en Informatica

REPORT *RAPPORT*

Vectorizing Matrix Operations arising from PDE Discretization on 9-Point Stencils

J.G. Blom, J.G. Verwer

Department of Numerical Mathematics

NM-R9221 1992

Vectorizing Matrix Operations arising from PDE Discretization on 9-Point Stencils

J.G. Blom, J.G. Verwer
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract. When solving a system of PDEs, discretized on 9-point stencils over a nonrectangular domain, the linear systems that arise will have matrices with an irregular block structure. In this paper we discuss the vectorization, for these type of matrices, of the matrix-vector multiply and of the Incomplete LU factorization and backsolve.

1991 Mathematics Subject Classification: Primary: 65Y20. Secondary: 65F10, 65N22.

1991 CR Categories: G1.3, G1.8.

Keywords & Phrases: vectorization, nonsymmetric sparse linear systems, nonrectangular domain, matrix-vector multiplication, ILU preconditioning, hyperplane method.

Note: This work was supported by Cray Research, Inc. under grant CRG 92.05, via the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF).

1. Introduction

In [11, 9, 10, 13, 12] an adaptive-grid finite-difference method is studied to solve time-dependent two-dimensional systems of partial differential equations (PDEs). Among others, a research code is developed which uses an implicit time-stepping method. This poses the task of solving at each time step large systems of nonlinear equations. In [12] modified Newton in combination with a direct sparse matrix solver was used to solve these systems. In [2], it was shown that Krylov-type iterative solvers combined with standard Incomplete LU preconditioning were much faster. We used in this paper the ILU preconditioner and the iterative solvers GMRES[7] and CGS[8] from the Sparse Linear Algebra Package (SLAP), and a CGS-variant BiCGStab[14]. SLAP is a public domain code written by Greenbaum and Seager (with contributions of several other authors) that is available from Netlib[4].

The greater part of the iterative solvers is highly vectorizable (triads, dot products, etc.). In the SLAP code, which is intended for general sparse matrices, the required matrix operations, however, are much less amenable to vectorize. Since no matrix structure is assumed, the ILU factorization is more a searching process than a computational process. The ILU backsolve and the matrix-vector multiply do vectorize, but the resulting vector lengths are relatively small. On the other hand the ILU preconditioning has proven itself very robust and efficient for our problems, with respect to the convergence rate, in combination with Krylov-type methods. In addition, later experiments with the iterative solver GMRESR[15] showed that, for this solver, in various cases the expensive ILU preconditioning can be replaced by a simple diagonal scaling, although at the cost of much more iterations. In these cases the computational time will be dominated by the matrix-vector multiplications. We therefore decided to replace the matrix-vector multiply and the ILU preconditioning routines of the SLAP library with routines written especially for our type of problem: systems of PDEs discretized on a 9-point stencil and on a nonrectangular grid, by which we mean a grid that is bounded by an arbitrary right-angled polygon (cf. Fig. 3).

There are well-known techniques to vectorize the matrix-vector multiply and the ILU preconditioning process for a *scalar* PDE defined on a *rectangular domain* and discretized using a 5- or 7-point stencil. The objective of this paper is to report on the vectorization of the matrix operations that

Report NM-R9221
ISSN 0169-0388
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

occur when solving a *system* of PDEs discretized on a *9-point stencil* and on a *nonrectangular domain.* The computations were performed on a Cray Y-MP but our findings will hold also on comparative vector computers (comparative with respect to $n_{1/2}$-values and gather/scatter operations).

2. Preliminaries

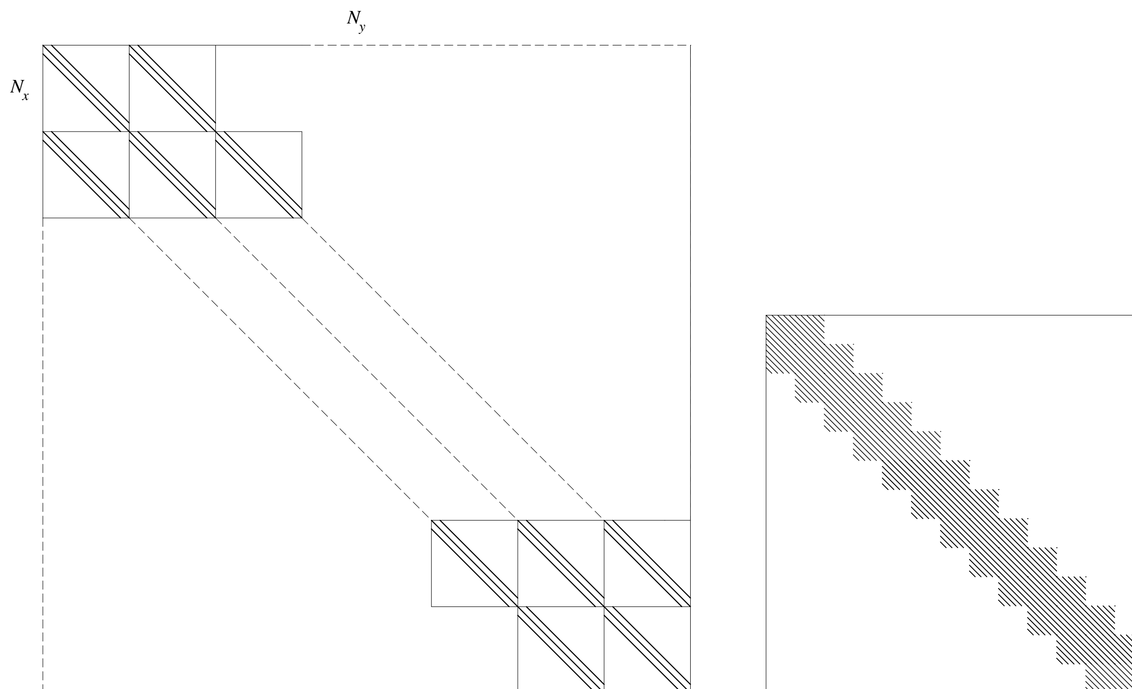

FIGURE 1.
Left: rectangular grid, $N_x \times N_y$ points, 9-point stencil
Right: $N_x \times N_x$ block-tridiagonal block for a system of 5 PDEs

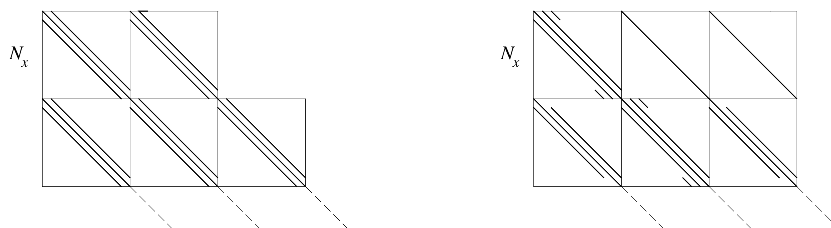

FIGURE 2.
Left: first-order discretization at boundaries
Right: second-order discretization at boundaries (first-order derivatives)

Spatial discretization of two-space dimensional PDE systems containing at most second order derivatives, very often takes place on the 9-point stencil. If the domain is rectangular and the nodes are in natural order, i.e., stored along grid lines in, e.g., the x-direction, the matrices in those systems are sparse and very regular. For a scalar PDE the matrix is then block tridiagonal and each block is

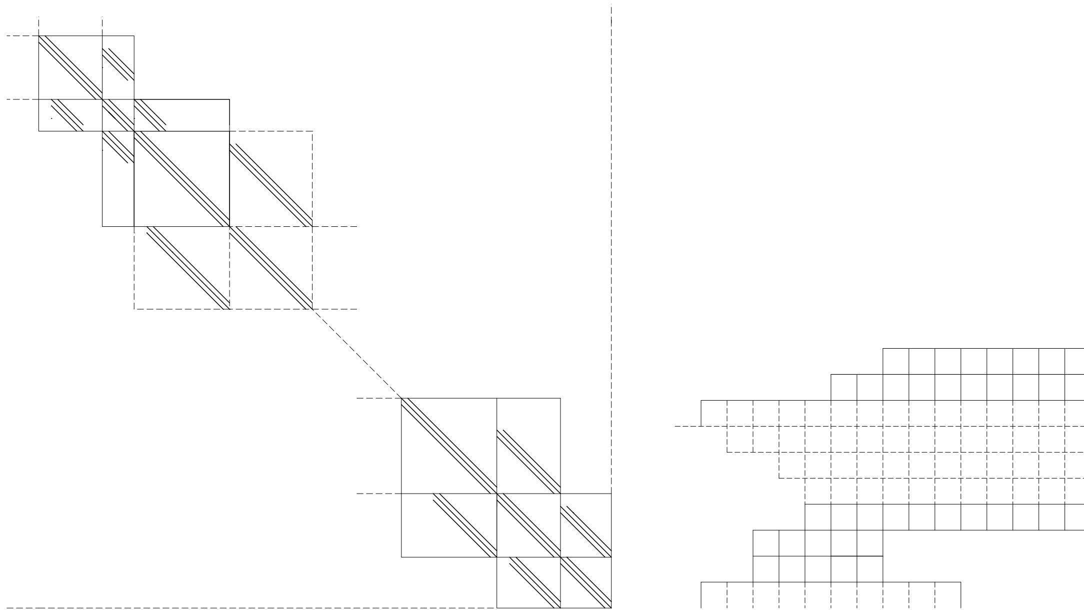

FIGURE 3.
Nonrectangular grid, 9-point stencil, first-order discretization at boundaries

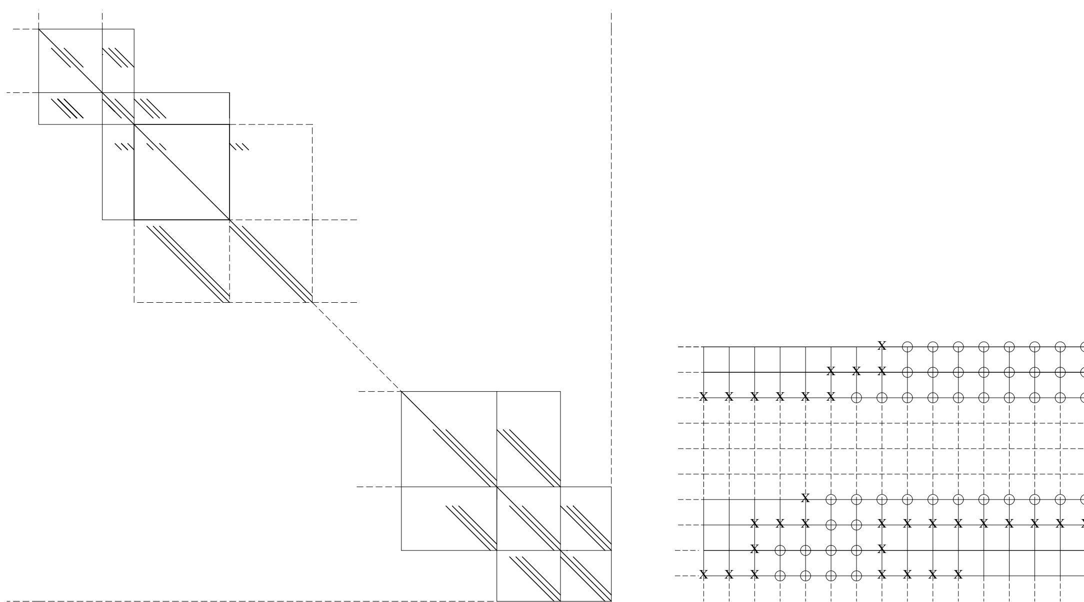

FIGURE 4. Locally refined grid, internal boundary points marked by X, internal grid points and non-Dirichlet boundary points by ◯

a tridiagonal matrix itself. For systems, and if the components are alternatingly stored, each block itself is a block-tridiagonal matrix (see Fig. 1). For such systems much research has been done on the vectorization of the matrix operations. E.g., the matrix-vector multiply will be speeded up when the matrix is stored diagonal-wise instead of column-wise like in SLAP. For the incomplete factorization several vectorization possibilities are available, one of which, first published in [1] for a scalar PDE and a 5- or 7-point stencil, reorganizes the order of the computations, although the resulting ILU factorization is the same as when proceeding the nodes in natural order. This appears to have a favorable influence on the stability of the operation and on the degree to which the eigenspectrum of the matrix is approximated, this in contrast to the Multicolor Methods (see, e.g., [5]) where one has to balance between a good preconditioner and an optimal vector performance.

In this paper we will report on the vectorization of the matrix-vector multiplication and the ILU preconditioning of less regular matrices. First of all, if the boundary equations of the PDE contain at most first-order derivatives and are discretized using second order one-sided differences some irregularity arises in the matrix when compared with first order one-sided differences as is depicted in Fig. 2. These irregularities in the first and last row of each diagonal block and the extra block in the first and last block-row makes a diagonal-wise matrix vector multiply very inefficient due to the large number of unnecessary operations. Second, if the domain of the PDE is not rectangular, then the diagonal $N_x \times N_x$ blocks will be replaced by blocks with as dimension the row length of the relevant row of the grid, while the off-diagonal blocks will be rectangular instead of square (cf. Fig. 3). Note that the structure of the matrix is still symmetric around the diagonal. However, in this case direct diagonal storage is not feasible any longer, and it is necessary to use some kind of indirect addressing. With Dirichlet boundaries or, as in the case of locally uniform grid refinement, with internal boundaries consisting of interpolated coarse grid values the structure of the matrix is no longer symmetric (see Fig. 4).

3. Matrix-vector multiplication

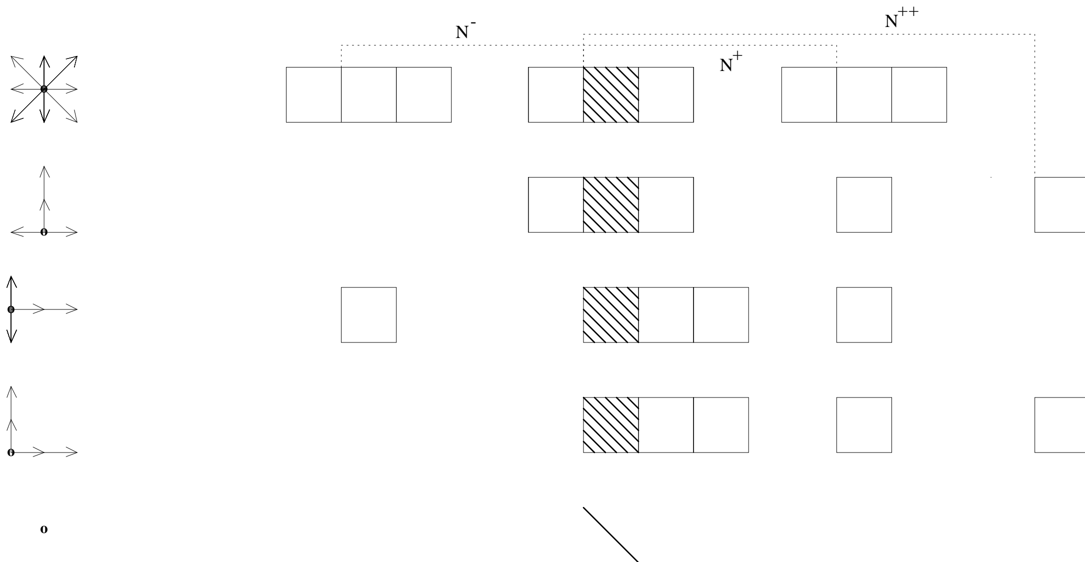

FIGURE 5. Structure of block-rows: internal point,
lower boundary point, left boundary point, lower left corner, Dirichlet boundary point

The matrices that we need to deal with (cf. Fig. 4) have in case of a system of PDEs with `NPDE` components different `NPDE`-block row structures for the internal points and for the specific boundary

points (see Fig. 5, the not displayed boundary points have analogous structures). In all cases the maximum number of blocks in a row is 9, but the placement and the distance from the block to the main diagonal varies. In this section we will discuss a number of storage modes, viz., the SLAP column format and various ways of diagonal storage modes. The inner loop of a MATVEC operation will consist in all cases of 1 store, 1 multiplication and 1 addition; but different storage modes can have their influence on the number of loads, the length of the vector and the chaining possibilities.

3.1. SLAP

The standard storage mode used in the SLAP library is the SLAP column format. In this format the non-zeros are stored counting down columns, except for the diagonal entry, which appears first in each 'column'. One integer array holds the row index for each nonzero and a second one holds the offsets of each column. The matrix vector multiply is then coded as

```
      DO ICOL = 1, N
         DO INDEX = JCOLA(ICOL), JCOLA(ICOL+1)-1
            Y(IROWA(INDEX)) = Y(IROWA(INDEX)) + A(INDEX)*X(ICOL)
         ENDDO
      ENDDO
```

The inner loop will vectorize using gather/scatter operations, resulting in 3 loads (`A`, `IROWA`, `Y`). The advantage of this storage mode is its flexibility. However, for a scalar PDE the vector length of the inner loop will be only 9.

3.2. Diagonal storage

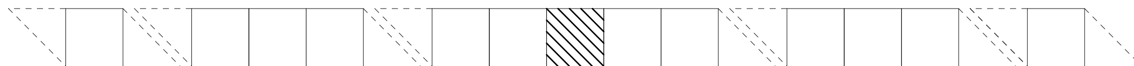

FIGURE 6. Natural storage along diagonals

To get a longer vector length the usual way is to store the matrix diagonal-wise. In case of a rectangular domain and a second-order discretization of the boundary equations this means that one gets 14.`NPDE`$-$ 3 vectors of length `N` $= (N_x \times N_y)$.`NPDE` (the block diagonal is 5.`NPDE` because of the left and right boundaries) and 2.`NPDE` $- 2$ vectors of length `NX` $= N_x$.`NPDE` for the lower and upper boundaries (cf. Fig. 6). A matrix-vector multiply looks like

```
C Inner 5 blocks
      DO IDIAG = -3*NPDE+1, 3*NPDE-1
         DO IROW = MAX(-IDIAG,0)+1, MIN(N,N-IDIAG)
            Y(IROW) = Y(IROW) + AD(IROW,IDIAG)*X(IROW+IDIAG)
         ENDDO
      ENDDO
C Outer 6 blocks
      DO IDIAG = -2*NPDE+1, 2*NPDE-1
         DO IROW = -IDIAG+NX+1,N
            Y(IROW) = Y(IROW) + AL(IROW,IDIAG)*X(IROW+IDIAG-NX)
         ENDDO
         DO IROW = 1, N-IDIAG-NX
            Y(IROW) = Y(IROW) + AU(IROW,IDIAG)*X(IROW+IDIAG+NX)
         ENDDO
      ENDDO
C Outer boundary blocks
```

```
      DO IDIAG = -NPDE+1, NPDE-1
         DO IROW = 1, NX
            Y(IROW) = Y(IROW) + BU(IROW,IDIAG)*X(IROW+IDIAG+2*NX)
         ENDDO
         DO IROW = N-NX+1, N
            Y(IROW) = Y(IROW) + BL(IROW,IDIAG)*X(IROW+IDIAG-2*NX)
         ENDDO
      ENDDO
```

The inner loops will vectorize, without gather/scatter operations, using 3 loads and with a much larger vector length than in SLAP mode, viz., ca. N for the full diagonals and NX for the diagonals resulting from the second-order discretization of the upper and lower boundary equations.

If the domain is not rectangular it is impossible to avoid indirect addressing. The most 'direct' translation of the diagonal storage mode for the matrix with block-rows as in Fig. 5 is pictured in Fig. 6. But in contrast to the implementation on a rectangular domain, now only the inner 6.NPDE − 1 diagonals will translate in a loop with direct addressing. The other ones need indirect addressing of the multiplicand vector, resulting in 1 extra load. However, this should have, on the Cray Y-MP, not much influence on the CPU-time or Megaflop rate.

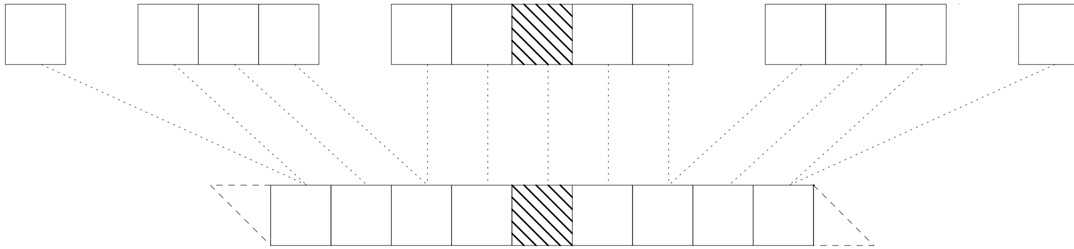

FIGURE 7. Block diagonal storage

In the above storage mode a total of 14.NPDE − 3 diagonals of length N and 4.NPDE − 2 of length NX are used and the maximum number of elements on a row is only 9.NPDE. One can also store the matrix such that a minimum number of diagonals (10.NPDE − 1) results (see Fig. 7). Assuming that INDM contains pointers to the left block in Fig. 7 and INDP to the blocks right of the diagonal, a matrix-vector multiply will look like

```
      DO IDIAG = -NPDE, NPDE
         DO IROW = MAX(-IDIAG,0)+1, MIN(N,N-IDIAG)
            Y(IROW) = Y(IROW) + A(IROW,IDIAG)*X(IROW+IDIAG)
         ENDDO
      ENDDO
      DO IDIAG = NPDE+1, 5*NPDE-1
         DO IROW = 1, N
            Y(IROW) = Y(IROW) + A(IROW,-IDIAG)*X(INDM(IROW,-IDIAG)) +
                                A(IROW,+IDIAG)*X(INDP(IROW,+IDIAG))
         ENDDO
      ENDDO
```

The inner 2.NPDE + 1 diagonals are translated into an inner loop with direct addressing. The number of indirect addressed loops will be slightly less than in the previous case. In these loops 7 loads are needed for 4 floating-point operations.

Note that in both cases it is also possible to implement a second-order discretization of boundary equations with mixed derivatives, or any other as long as the total number of entries on a row is not

larger than 9.NPDE. The occurrence of non existing entries in the last inner loop can be prevented by letting at the appropriate places the indices INDM and INDP point to IROW and storing zeros in A.

4. ILU

In this section we will subscribe two different ways to perform the standard no-fillin Incomplete LU decomposition and the backsolve needed for the solution of a system. With standard no-fillin Incomplete LU decomposition we mean a factorization

$$A = LU + R \quad \text{or} \quad A = LDU + R$$

obtained by Gaussian elimination omitting all resulting entries at places where the original matrix had zeros.

4.1. SLAP

The SLAP package is intended for general sparse matrices and therefore especially the ILU factorization contains more sorting and searching processes than arithmetical computations. The factorization of A in LDU format in the SLAP library is as follows. First A, being in SLAP column format, is copied into L (the lower triangle of A) in SLAP row format, into D (the diagonal) and into U (the upper triangle of A) in SLAP column format. Then the standard incomplete LU decomposition will be performed and finally the diagonal D will be replaced by its inverse. The whole process, apart from the inverse of D, will not be vectorized.

Assuming that the matrices L, D, and U are stored as described above solving a system $Ax = b$ is implemented as follows

```
C L*Y = B
      DO I = 1, N
         X(I) = B(I)
      ENDDO
      DO IROW = 2, N
         DO J = IL(IROW), IL(IROW+1)-1
            X(IROW) = X(IROW) - L(J)*X(JL(J))
         ENDDO
      ENDDO
C D*Z = Y
      DO I = 1, N
         X(I) = X(I) * DINV(I)
      ENDDO
C U*X = Z
      DO ICOL = N, 2, -1
         DO J = JU(ICOL), JU(ICOL+1)-1
            X(IU(J)) = X(IU(J)) - U(J)*X(ICOL)
         ENDDO
      ENDDO
```

All loops will vectorize quite well, but the second and fourth are hampered by the fact that the loop length will be approximately 4.NPDE and, possibly, by indirect addressing.

4.2. 'Hyperplane' method

The structure of the matrix A, and as a consequence the no-fillin factorization, is of course dependent on the ordering of the nodes. So a node-reordering strategy with the aim to get good vectorizing properties can have influence on the convergence of the iterative solver. E.g., the multicolor method

of Fujino and Doi[5] where nodes with the same color are handled in the same sweep (compare the red-black ordering for a scalar 5-point stencil) results in good vectorizing properties (for a 'small' number of colors) but needs to use a 'large' number of colors to have comparable convergence results. The no-fillin factorization corresponding with node ordering along grid lines and an alternating storage of the PDE-components proved to be a robust preconditioner for our class of PDEs. Since in many applications a good preconditioner is necessary to get convergence anyhow, we have decided to restrict ourselves to this preconditioner.

In [1] Ashcraft and Grimes show, that the ILU factorization and backsolve can be vectorized by reordering the computations (not the nodes). It should be emphasized that the thus obtained factorization and backsolve is, using exact arithmetic, equivalent to the recursive one. In their paper Ashcraft and Grimes perform a data dependency analysis for the factorization and backsolve of a matrix resulting from discretization on a rectangular domain of a scalar PDE using 5-, respectively, 7-point stencils. The resulting sets of nodes where the computation can proceed simultaneously are in case of a 5-point stencil grid diagonals and in case of a 7-point stencil with entries upper left and lower right skew grid diagonals (knight's move in chess). It should be noted, however, that a 7-point stencil with entries upper right and lower left results, like a 5-point stencil, in ordinary grid diagonals.

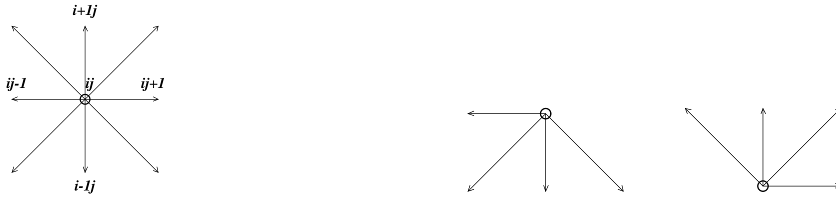

FIGURE 8. 9-point stencil,
dependency for ILU factorization and forward solve, respectively for backward solve

The data dependency analysis for the factorization goes as follows: One considers the equation for one grid node, which corresponds to entries in one row of the matrix, and analyzes from what entries these computations are dependent. Let the numbering of the grid be as in Fig. 8. Let the partitioning of a matrix A in columns, respectively rows, be given by

$$A = (a_{11}, \ldots, a_{N_xN_y}) = \begin{pmatrix} A^{11} \\ \vdots \\ A^{N_xN_y} \end{pmatrix},$$

and an element in row ij and column kl by α^{ij}_{kl}. Then a row of $A = LU + R$ is given by

$$A^{ij} = L^{ij}.U + R^{ij},$$

where

$$\begin{aligned} L^{ij}.U \;=\; (\;& 0, \ldots, 0, L^{ij}u_{i-1j-1}, L^{ij}u_{i-1j}, L^{ij}u_{i-1j+1}, L^{ij}u_{i-1j+2}, \\ & 0, \ldots, 0, L^{ij}u_{ij-2}, L^{ij}u_{ij-1}, L^{ij}u_{ij}, L^{ij}u_{ij+1}, L^{ij}u_{ij+2}, \\ & 0, \ldots, 0, L^{ij}u_{i+1j-2}, L^{ij}u_{i+1j-1}, L^{ij}u_{i+1j}, L^{ij}u_{i+1j+1}, 0, \ldots, 0). \end{aligned}$$

Of these the entries $L^{ij}u_{i-1j+2}$, $L^{ij}u_{ij-2}$, $L^{ij}u_{ij+2}$, and $L^{ij}u_{i+1j-2}$ do not correspond with entries in A and thus are ignored. The remaining lead to the set of equations

$$\begin{aligned} \lambda_{i-1j-1} &= \alpha_{i-1j-1}/v^{i-1j-1}_{i-1j-1} \\ \lambda_{i-1j} &= (\alpha_{i-1j} - \lambda_{i-1j-1}v^{i-1j-1}_{i-1j})/v^{i-1j}_{i-1j} \\ \lambda_{i-1j+1} &= (\alpha_{i-1j+1} - \lambda_{i-1j}v^{i-1j}_{i-1j+1})/v^{i-1j+1}_{i-1j+1} \end{aligned}$$

$$
\begin{aligned}
\lambda_{ij-1} &= (\alpha_{ij-1} - \lambda_{i-1j-1} v_{ij-1}^{i-1j-1} - \lambda_{i-1j} v_{ij-1}^{i-1j}) / v_{ij-1}^{ij-1} \\
v_{ij} &= \alpha_{ij} - \lambda_{i-1j-1} v_{ij}^{i-1j-1} - \lambda_{i-1j} v_{ij}^{i-1j} - \lambda_{i-1j+1} v_{ij}^{i-1j+1} - \lambda_{ij-1} v_{ij}^{ij-1} \\
v_{ij+1} &= \alpha_{ij+1} - \lambda_{i-1j} v_{ij+1}^{i-1j} - \lambda_{i-1j+1} v_{ij+1}^{i-1j+1} \\
v_{i+1j-1} &= \alpha_{i+1j-1} - \lambda_{ij-1} v_{i+1j-1}^{ij-1} \\
v_{i+1j} &= \alpha_{i+1j} - \lambda_{ij-1} v_{i+1j}^{ij-1} \\
v_{i+1j+1} &= \alpha_{i+1j+1},
\end{aligned}
$$

where the row indices ij are omitted. One can see that, like for the 7-point stencil with entries at the upper left and lower right, the dependency of the computations is as pictured in Fig. 8. The analysis of the forward and backward solve is analogous and results in dependencies as shown also in Fig. 8. Note that the modification of ILU from Gustafsson[6], which implies that the entries in LU that do not correspond with entries in A, $L^{ij} u_{i-1j+2}$, etc., are added to the main diagonal multiplied by a factor $0 \leq \alpha < 1$

$$
\tilde{v}_{ij}^{ij} = v_{ij}^{ij} + \alpha(\lambda_{i-1j+1} v_{i-1j+2}^{i-1j+1} + \lambda_{i-1j-1} v_{ij-2}^{i-1j-1} + \lambda_{i-1j+1} v_{ij+2}^{i-1j+1} + \lambda_{ij-1} v_{i+1j-2}^{ij-1}),
$$

results in the same dependency relation.

For systems of PDEs the same can be done with submatrices of `NPDE` $\times$ `NPDE` instead of matrix elements. The element v_{ij}^{ij} has then to be replaced by a lower and an upper submatrix.

For a nonrectangular domain it would be very inefficient to order the computations along skew grid diagonals as is done by Ashcraft and Grimes[1]. For irregular domains the sets of nodes for which the computations can be performed concurrently have to be build up. The first set for the factorization and the forward solve, $Ly = b$, consists of the lower-left corner and all the Dirichlet boundary points. The other nodes are proceeded along grid lines and are placed in the set with a number 1 higher than the maximum set number of the dependency nodes. The sets of nodes for the backward solves, $Ux = y$, are obtained analogously but starting from upper-right and proceeding backwards. Note that the sets of nodes for the forward and backward solve are in general different. This in contrast with Ashcraft and Grimes[1] where the sets are the same but proceeded from first to last, respectively, from the last to the first. The determination of these sets is an essentially sequential but simple process, which has to be done only once per grid.

The implementation of this 'hyperplane' method for nonrectangular domains is as simple as for rectangular domains. We give the code for the scalar case.

Let the matrix A be stored in the array `A(N,-4:4)`, the lower diagonals in `A(.,-4:-1)`, the main diagonal in `A(.,0)`, and the upper diagonals in `A(.,1:4)`. Let `NIM(.,-4:-2)` contain the column index of the lower 3 diagonals and `NIP(.,2:4)` the column index of the upper 3 diagonals. Finally, let `S(IS(m-1)+1:IS(m))` contain the indices of the nodes in the m-th set (S_m) for which the computations can be done concurrently assuming that the computations for the nodes in $S_{1,\ldots,m-1}$ are done. First we have a loop for the nodes in S_1

```
      DO L = 1, IS(1)
         K = S(L)
         A(K,0) = 1.0 / (A(K,0))
      ENDDO
```

Then we loop over the rest of the 'hyperplane' sets.

```
      DO M = 2, IS(0)
         DO L = IS(M-1)+1, IS(M)
C   Compute lower diagonals
            K = S(L)
            A(K,-4) = A(K,-4)*A(NIM(K,-4),0)
```

```
           A(K,-3) = (A(K,-3) - A(K,-4)*A(NIM(K,-4),1))
     +               * A(NIM(K,-3),0)
           A(K,-2) = (A(K,-2) - A(K,-3)*A(NIM(K,-3),1))
     +               * A(NIM(K,-2),0)
           A(K,-1) = (A(K,-1) - A(K,-4)*A(NIM(K,-4),3)
     +                          - A(K,-3)*A(NIM(K,-3),2)) * A(K-1,0)
C    Compute inverse of main diagonal
           A(K,0) = 1.0 / (A(K,0) - A(K,-4)*A(NIM(K,-4),4)
     +                            - A(K,-3)*A(NIM(K,-3),3)
     +                            - A(K,-2)*A(NIM(K,-2),2)
     +                            - A(K,-1)*A(K-1      ,1))
C    Compute upper diagonals
           A(K,1) = A(K,1) - A(K,-3)*A(NIM(K,-3),4)
     +                     - A(K,-2)*A(NIM(K,-2),3)
           A(K,2) = A(K,2) - A(K,-1)*A(K-1,3)
           A(K,3) = A(K,3) - A(K,-1)*A(K-1,4)
        ENDDO
      ENDDO
```

All inner loops can be vectorized. For rectangular boundaries the loop length varies from 1, in the first and last loop, to $\min(N_x/2, N_y)$ in the middle loops.

For the backsolve we use 2 arrays holding the sets; `SL, ISL` the equivalent of `S, SL` in the factorization and `SU, ISU` the analogue for the backward solve. The forward solve is then implemented as follows

```
C Ly = b
      DO M = 2, ISL(0)
         DO L = ISL(M-1)+1, ISL(M)
            K = SL(L)
            B(K) = B(K) - A(K,-1)*B(K- 1)       - A(K,-2)*B(NIM(K,-2))
     +                  - A(K,-3)*B(NIM(K,-3)) - A(K,-4)*B(NIM(K,-4))
         ENDDO
      ENDDO
```

and the backward solve

```
C Ux = y
      DO L = 1, ISU(1)
         K = SU(L)
         B(K) = B(K) * A(K,0)
      ENDDO
C
      DO M = 2, ISU(0)
         DO L = ISU(M-1)+1, ISU(M)
            K = SU(L)
            B(K) = (B(K) - A(K,1)*B(K+ 1)      - A(K,2)*B(NIP(K,2))
     +                   - A(K,3)*B(NIP(K,3)) - A(K,4)*B(NIP(K,4)))
     +             * A(K,0)
         ENDDO
      ENDDO
```

For systems of PDEs the matrix A is stored in the array `A(N,NPDE,NPDE,-4:4)` such that `A(`ij`,1:NPDE,1:NPDE,.)` contains a block of `NPDE` $\times$ `NPDE` elements corresponding with node ij. After

factorization `A(.,1:NPDE,1:NPDE,-4:-1)` contains the lower block diagonals of L and `A(.,1:NPDE,1:NPDE,1:4)` the upper block diagonals of U. `A(.,ic,jc,0)` contains for `jc < ic` the main block diagonal of L (the main diagonal of L is equal to I) and for `jc` $\geq$ `ic` the main block diagonal of U, with the main diagonal inverted. The previously cited backward-solve loop will for systems of PDEs then look like

```
      DO M = 2, ISU(0)
         DO IC = NPDE, 1, -1
         DO JC = NPDE, 1, -1
         DO L = ISU(M-1)+1, ISU(M)
            K = SU(L)
               B(K,IC) = B(K,IC) - A(K,IC,JC,1)*B(K+1      ,JC)
     +                           - A(K,IC,JC,2)*B(NIP(K,2),JC)
     +                           - A(K,IC,JC,3)*B(NIP(K,3),JC)
     +                           - A(K,IC,JC,4)*B(NIP(K,4),JC)
         ENDDO
         ENDDO
         DO JC = NPDE, IC+1, -1
         DO L = ISU(M-1)+1, ISU(M)
            K = SU(L)
               B(K,IC) = B(K,IC) - A(K,IC,JC,0)*B(K,JC)
         ENDDO
         ENDDO
         DO L = ISU(M-1)+1, ISU(M)
            K = SU(L)
               B(K,IC) = B(K,IC) * A(K,IC,IC,0)
         ENDDO
         ENDDO
      ENDDO
```

Again the inner loops will vectorize with a loop length equivalent to the scalar case. One can increase the performance for a *fixed* number of components `NPDE` by interchanging where possible the `IC, JC` and `L` loops and unroll the inner loops:

```
         DO L = ISU(M-1)+1, ISU(M)
            K = SU(L)
CFPP$ UNROLL
         DO IC = NPDE, 1, -1
CFPP$ UNROLL
         DO JC = NPDE, 1, -1
               B(K,IC) = B(K,IC) - A(K,IC,JC,1)*B(K+1      ,JC)
     +                           - A(K,IC,JC,2)*B(NIP(K,2),JC)
     +                           - A(K,IC,JC,3)*B(NIP(K,3),JC)
     +                           - A(K,IC,JC,4)*B(NIP(K,4),JC)
         ENDDO
         ENDDO
         ENDDO
```

The loop length will, of course, be the same. Note that it is important to have the indirect addressing as the first index since otherwise extra (floating-point) vector multiplications are needed just for the indexing of an array.

5. Performance measurements

In this section we will compare the different MATVEC and ILU implementations. The timings were done on 1 processor of a Cray Y-MP, which has a clock cycle time of 6ns. This gives a theoretical peak performance on 1 processor of 167 Mflops and 333 when chaining an add and a multiply. Since during one cycle time 1 store and 2 loads can be performed, indirect addressing of (one of) the vector operands of a triad will reduce the performance at least with a factor of 2, bank conflicts left out of consideration. Each processor of the Cray Y-MP addresses 32 memory banks, or to be more precise subsections, so if two consecutive memory accesses have a stride of 32, 16 or 8, memory access conflicts will occur. With 'random' addressing the 32 memory subsections will in general be enough not to create a too large performance degradation as a result of memory conflicts.

To measure the Megaflop rate and the CPU time of a routine we used the Cray utility Perftrace[3], that gives the hardware performance by program unit. We first discuss the Megaflop rate, for increasing order of the matrix, of the vectorized implementations of the MATVEC operation and the ILU factorization and backsolve, using both direct and indirect addressing. Since the performance of the SLAP implementations is independent of the order of the matrix, we did not include the SLAP routines in this subsection. Next we compare the general purpose SLAP library codes with our routines on matrices that typically occur in adaptive grid environments.

5.1. Megaflop rates

To get an idea of the actual performance of our routines we called each routine 200 times with matrices arising from square grids with N_x ranging from 20 to 200, both for scalar PDEs and for a system with 2 components. For these matrices both the direct addressing variant and the indirect addressing variant can be used.

5.1.1. Performance of matrix-vector multiplication

N_x	scalar				2×2 blocks			
	DIAG		BDIAG		DIAG		BDIAG	
	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
20	6.4E-05	129	5.9E-05	116	2.7E-04	143	2.3E-04	129
40	2.4E-04	137	2.0E-04	137	1.1E-03	148	8.6E-04	137
60	5.4E-04	140	4.6E-04	132	2.3E-03	156	1.8E-03	145
80	8.6E-04	155	7.8E-04	140	4.0E-03	157	3.4E-03	139
100	1.4E-03	155	1.2E-03	140	6.2E-03	159	5.3E-03	140
120	1.9E-03	159	1.7E-03	142	9.0E-03	157	7.5E-03	143
140	2.6E-03	156	2.4E-03	141	1.2E-02	158	1.0E-02	141
160	3.4E-03	158	3.0E-03	144	1.6E-02	156	1.3E-02	142
180	4.3E-03	157	4.0E-03	138	2.0E-02	159	1.7E-02	142
200	5.4E-03	156	4.7E-03	144	2.5E-02	158	2.1E-02	141

Table 1. Perftrace report for MATVEC: Megaflop rate

For the matrix-vector multiply we tested the variant using fixed strides, called DIAG (see page 5), and the indirect-addressing variant BDIAG of page 6. DIAG needs 3 (independent) loads for 1 multiply and an add. So unless the stride `NX` in the last two loops causes bank conflicts one can expect to reach the peak performance of 167 Mflops on the Cray Y-MP, although at the cost of some superfluous computations. BDIAG needs for the indirectly addressed loops 7 loads of which 2 dependent for 2 multiplications and 2 adds, which gives again, apart from bank conflicts, an optimal peak performance of 167 Mflops. Note that only the latter can be used for irregular domains.

So for both versions the maximum performance is 1 flop per cycle. The Perftrace report in Table 1 shows that this peak performance is almost achieved, even for the indirect addressing mode. The latter is even faster, since in the block diagonal storage mode no superfluous computations are done.

5.1.2. Performance of ILU factorization and backsolve

N_x	scalar				2 × 2 blocks				2 × 2 unrolled			
	HP		HP_set		HP		HP_set		HP		HP_set	
	Av.Tm.	Mf	Av.Tm	Mf	Av.Tm	Mf	Av.Tm	Mf	Av.Tm	Mf	Av.Tm	Mf
20	3.9E-04	34	4.4E-04	28	9.4E-03	15	8.6E-03	12	5.8E-03	19	5.9E-03	17
40	9.2E-04	57	1.1E-03	45	2.1E-02	23	2.0E-02	20	1.3E-02	33	1.4E-02	29
60	1.6E-03	74	1.9E-03	58	3.3E-02	31	3.3E-02	28	2.1E-02	45	2.3E-02	38
80	2.3E-03	88	3.1E-03	64	4.7E-02	38	4.9E-02	33	3.0E-02	54	3.5E-02	45
100	3.2E-03	100	4.6E-03	67	6.2E-02	44	6.8E-02	37	4.1E-02	63	5.0E-02	50
120	4.2E-03	107	5.8E-03	77	7.8E-02	49	9.0E-02	40	5.2E-02	71	6.5E-02	55
140	5.8E-03	106	8.7E-03	70	1.1E-01	48	1.2E-01	40	7.2E-02	69	9.3E-02	53
160	7.2E-03	111	1.1E-02	72	1.3E-01	52	1.5E-01	42	8.8E-02	74	1.2E-01	55
180	8.6E-03	118	1.4E-02	71	1.5E-01	56	1.8E-01	44	1.1E-01	78	1.4E-01	56
200	1.0E-02	122	1.6E-02	77	1.8E-01	59	2.2E-01	46	1.2E-01	83	1.7E-01	59

TABLE 2. Perftrace report for ILU factorization: Megaflop rate

N_x	scalar				2 × 2 blocks				2 × 2 unrolled			
	HP		HP_set		HP		HP_set		HP		HP_set	
	Av.Tm.	Mf	Av.Tm	Mf	Av.Tm	Mf	Av.Tm	Mf	Av.Tm	Mf	Av.Tm	Mf
20	3.0E-04	30	2.7E-04	25	1.7E-03	23	1.8E-03	16	1.1E-03	32	1.2E-03	23
40	6.5E-04	49	7.0E-04	39	3.8E-03	35	4.2E-03	27	2.4E-03	52	3.0E-03	37
60	1.1E-03	65	1.2E-03	50	6.1E-03	47	7.2E-03	35	4.0E-03	69	5.3E-03	48
80	1.5E-03	79	1.9E-03	57	8.7E-03	57	1.1E-02	41	5.7E-03	83	8.5E-03	53
100	2.0E-03	90	2.8E-03	61	1.1E-02	66	1.6E-02	45	7.8E-03	94	1.2E-02	58
120	2.6E-03	100	3.8E-03	65	1.5E-02	74	2.0E-02	50	1.0E-02	103	1.6E-02	63
140	3.5E-03	99	5.4E-03	62	2.0E-02	74	2.8E-02	49	1.4E-02	103	2.3E-02	61
160	4.4E-03	104	6.8E-03	64	2.4E-02	80	3.5E-02	51	1.7E-02	110	2.9E-02	62
180	5.1E-03	112	8.2E-03	67	2.8E-02	85	4.3E-02	53	2.1E-02	113	3.6E-02	63
200	6.0E-03	119	9.9E-03	69	3.3E-02	90	5.1E-02	55	2.4E-02	121	4.2E-02	67

TABLE 3. Perftrace report for backsolve: Megaflop rate

For the ILU factorization and backsolve we tested the (indirectly addressed) version as given on page 9, respectively, page 10 (called HP_set in the tables) and the directly addressed variant where the loops are along skew grid diagonals (HP). For the 2-component system we timed also the unrolled version (cf. page 11). Note that the vector length for the loops has a maximum of $N_x/2$.

In contrast with the matrix-vector multiply we find here a significant difference between the versions using direct addressing and using indirect addressing. This difference was not foreseen, since, e.g., for the forward solve coded with indirect addressing, as implemented on page 10, a typical loop needs 13 loads for 9 flops and inspection of the assembly code shows that the loads are arranged such that bank conflicts will not occur and a total number of 9 cycles is needed, provided that chaining indeed occurs and is not prevented by a too large number of instructions in between. However, the Cray Private Systems Programmers Reference Manual gives as one of the Hold Issue Conditions for the gather/scatter instructions a gather/scatter instruction in progress, which means that one can have

only 1 gathered load or scattered store at a time. Since 12 of the 13 loads and the store are indirect addressed, this means that the forward solve loop on page 10 takes at least 13 clock cycles for the 9 flops, which makes the performance measurements in Table 3 understandable. A possibility to improve the performance of the ILU factorization and the forward solve loop is to reorder the rows of the matrix and of the right hand-side in advance, resulting in the following loop for the forward solve

```
      DO M = 2, ISL(0)
         DO K = ISL(M-1)+1, ISL(M)
            B(K) = B(K) - A(K,-1)*B(NIM(K,-1)) - A(K,-2)*B(NIM(K,-2))
     +                  - A(K,-3)*B(NIM(K,-3)) - A(K,-4)*B(NIM(K,-4))
         ENDDO
      ENDDO
```

Indeed, measurements show that this loop is ca. 1.75 times faster than the original loop. A problem, however, is that the backward solve needs a different ordering of the matrix rows, thereby introducing in the backsolve procedure either a reordering after the forward solve or extra indirect addressing in the backward solve. In the experiments hereafter we will not consider this adaptation of the ILU preconditioning.

5.2. Comparison with SLAP

In this subsection we compare the SLAP routines with our implementations of the matrix-vector multiplication and the ILU preconditioning process on matrices resulting from systems of PDEs with 1, 2, and 3 components, discretized on 3 levels generated by the adaptive grid code. At internal nodes entries are generated assuming 9-point stencils, even if the PDE does not contain mixed derivatives. The discretization used is a second-order central difference scheme on the interior of the domain and a second-order one-sided difference formula for the boundary equations. Since the latter does not fit into the matrix structure needed for the vectorized implementation of the ILU factorization and backsolve, we replaced it, for the ILU timings, by a first-order discretization. We assume that this will have little influence on the convergence rate of the iterative process that solves the linear system. Timings are done on a uniform grid of 81×81 nodes and on the grids generated by the adaptive grid code starting on a uniform 21×21 grid and using 3 levels.

5.2.1. Test matrices

The first set of matrices comes from a scalar PDE example used in [11]. The corresponding grids generated for 3 levels are shown in Fig. 9. The number of nodes in these grids are 441, 1013 and 2421. The second test set originates from a system of two PDEs ([2]). The corresponding grids at the 3 levels are shown in Fig. 10 and contain 441, 797, respectively, 719 nodes. From the PDE system with 3 components as used in [12] we generated 3 level grids at two different times, T_1 and T_2 (cf. Fig. 11), to time for different numbers of nodes and different grid connectivity. For the first the number of nodes is 441, 462 and 966, and for the second 441, 929 and 2323. If a grid at a level contains disjunct subgrids, one matrix is generated, but of course the set of Dirichlet points at the grid interfaces allows concurrency in the computations performed with the 'hyperplane' method.

5.2.2. Performance of matrix-vector multiplication

In this subsection we compare the performance of the matrix-vector multiply routine SSMV (from SLAP) with our diagonally stored implementations DIAG (direct addressing, page 5) and BDIAG (indirectly addressed variant, page 6). For the performance of SSMV only the number of components, `NPDE`, is important; for the diagonally stored implementations DIAG and BDIAG the number of nodes and `NPDE`. Since the grid configuration is in neither case relevant, it is sufficient to discuss for each test problem only the configuration with the smallest (441) and largest (6561) number of nodes, corresponding with the uniform 21×21 grid and the uniform 81×81 grid. In Table 4 the results can

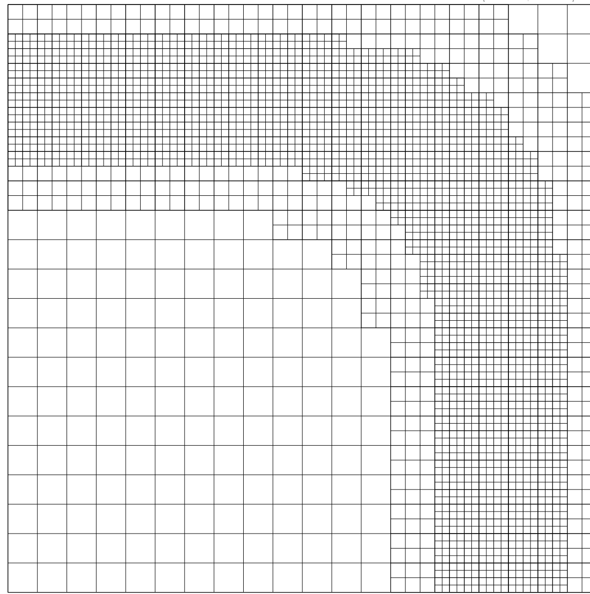

FIGURE 9. Grids for Problem I, NPDE = 1

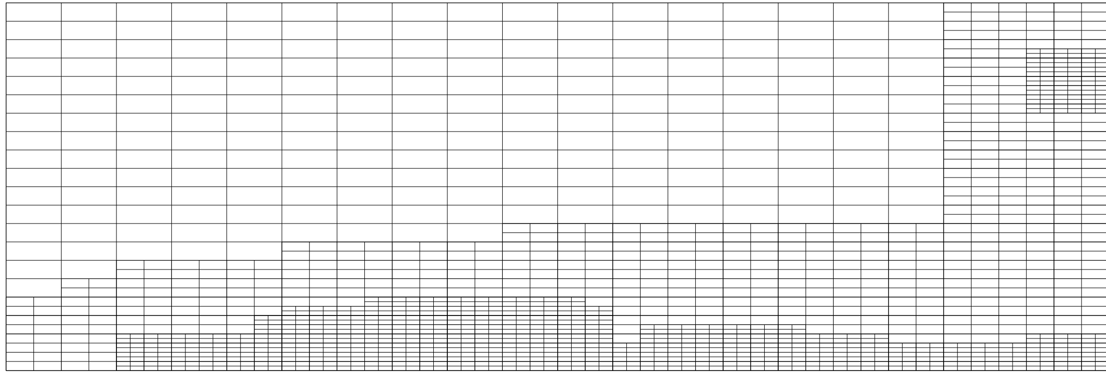

FIGURE 10. Grids for Problem II, NPDE = 2

Method	Problem I				Problem II			
	81 × 81		21 × 21		81 × 81		21 × 21	
	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSMV	6.8E-3	17	4.6E-4	16	1.6E-2	29	1.0E-3	29
DIAG	8.7E-4	158	6.7E-5	136	4.1E-3	158	2.9E-4	148
BDIAG	7.6E-4	146	6.1E-5	124	3.4E-3	142	2.6E-4	126

Method	Problem III			
	81 × 81		21 × 21	
	Avg.Time	Mf	Avg.Time	Mf
SSMV	2.5E-2	41	1.7E-3	39
DIAG	9.6E-3	157	6.6E-4	153
BDIAG	7.7E-3	146	5.6E-4	134

TABLE 4. Perftrace report for MATVEC

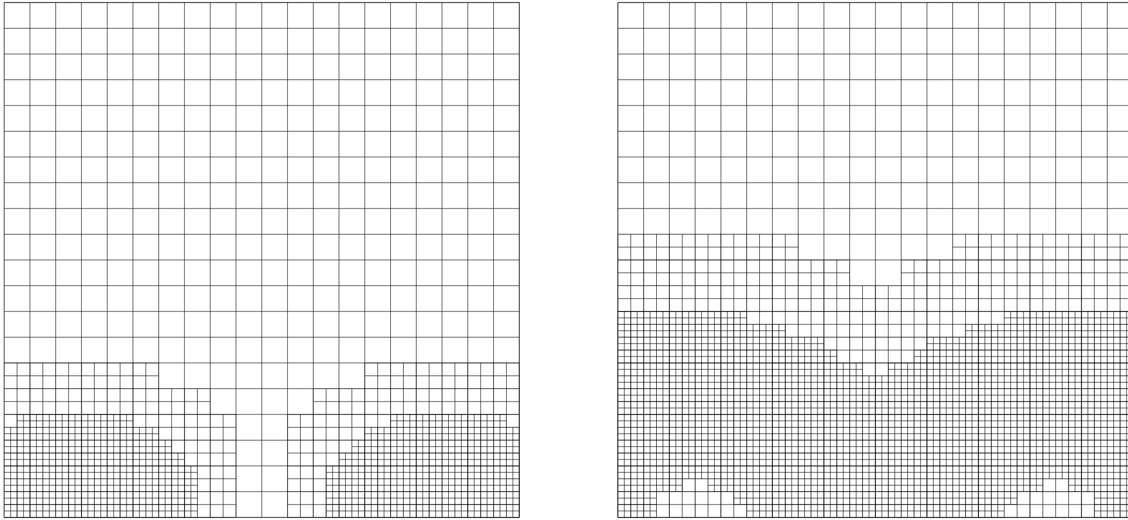

FIGURE 11. Grids for Problem III, NPDE $= 3$
Left at $t = T_1$, right at $t = T_2$

be found. As expected the SLAP routine does not attain a high Megaflop rate since the vector length is just 9.NPDE. The two diagonal storage modes result in an almost optimal Megaflop rate of 167 as was already mentioned in Section 5.1.1. Again, one is reminded that the DIAG implementation can not be used for nonrectangular domains. The conclusion from these figures is that we can use the same matrix-vector multiply routine BDIAG for both the rectangular domains as for the ones with irregular boundaries.

5.2.3. Performance of ILU factorization and backsolve

Method	81×81		21×21		level 2		level 3		3 levels	
	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSILUS	3.2E-1	1	2.0E-2	1	4.5E-2	1	1.1E-1	1	5.8E-2	1
HP_set	3.0E-3	67	4.6E-4	30	8.0E-4	37	1.5E-3	47	8.4E-4	44
HP	2.4E-3	87	4.2E-4	36						

TABLE 5. Perftrace report for ILU factorization for Problem I

Method	81×81		21×21		level 2		level 3		3 levels	
	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSLUI2	2.2E-2	9	1.5E-3	8	3.2E-3	8	7.4E-3	8	4.0E-3	8
HP_set	1.9E-3	59	2.9E-4	26	4.9E-4	33	8.9E-4	41	5.5E-4	37
HP	1.6E-3	78	3.1E-4	31						

TABLE 6. Perftrace report for backsolve for Problem I

For the ILU factorization and backsolve we compare the SLAP routine SSILUS, respectively, SSLUI2, with the implemented hyperplane ordering in the (indirectly addressed) version as given on page 9, respectively, page 10 (called HP_set in the tables) and, if possible, with the directly addressed variant,

Method	81 × 81		21 × 21		level 2		level 3		3 levels	
	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSILUS	2.3E+0	1	1.4E-1	1	2.4E-1	1	2.0E-1	1	1.9E-1	1
HP_set	4.9E-2	33	8.9E-3	12	1.8E-2	10	1.2E-2	14	1.3E-2	12
HP_set_u	3.6E-2	46	6.2E-3	18	1.2E-2	15	8.0E-3	20	8.9E-3	17
HP	4.5E-2	40	9.8E-3	15						
HP_u	3.0E-2	56	6.2E-3	20						

TABLE 7. Perftrace report for ILU factorization for Problem II

Method	81 × 81		21 × 21		level 2		level 3		3 levels	
	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSLUI2	4.4E-2	15	3.0E-3	15	5.1E-3	14	4.5E-3	14	4.2E-3	14
HP_set	1.1E-2	42	1.9E-3	16	3.7E-3	14	2.4E-3	18	2.7E-3	16
HP_set_u	8.1E-3	57	1.3E-3	23	2.6E-3	20	1.7E-3	26	1.9E-3	22
HP	8.5E-3	59	1.9E-3	23						
HP_u	5.9E-3	83	1.2E-3	32						

TABLE 8. Perftrace report for backsolve for Problem II

Method	81 × 81		21 × 21		T_1 level 2		T_1 level 3		T_1 3 levels	
	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSILUS	7.4E-0	1	4.5E-1	1	4.1E-1	1	9.3E-1	1	6.0E-1	1
HP_set	1.5E-1	36	2.7E-2	14	2.0E-2	17	2.9E-2	25	2.5E-2	19
HP_set_u	1.2E-1	45	2.3E-2	16	1.6E-2	21	2.5E-2	30	2.1E-2	23
HP	1.4E-1	45	2.9E-2	17						
HP_u	9.7E-2	58	2.0E-2	20						
Method					T_2 level 2		T_2 level 3		T_2 3 levels	
					Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSILUS					9.6E-1	1	2.4E-0	1	1.3E-0	1
HP_set					3.9E-2	19	7.7E-2	24	4.7E-2	21
HP_set_u					3.3E-2	23	6.3E-2	29	4.0E-2	25

TABLE 9. Perftrace report for ILU factorization for Problem III

Method	81 × 81		21 × 21		T_1 level 2		T_1 level 3		T_1 3 levels	
	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSLUI2	6.9E-2	21	4.6E-3	20	4.4E-3	19	9.5E-3	20	6.2E-3	20
HP_set	2.4E-2	44	3.9E-3	18	3.0E-3	22	4.4E-3	32	3.7E-3	25
HP_set_u	1.5E-2	71	2.5E-3	28	1.8E-3	35	2.8E-3	49	2.4E-3	39
HP	1.8E-2	64	3.8E-3	25						
HP_u	1.3E-2	84	2.5E-3	33						
Method					T_2 level 2		T_2 level 3		T_2 3 levels	
					Avg.Time	Mf	Avg.Time	Mf	Avg.Time	Mf
SSLUI2					9.4E-3	20	2.4E-2	20	1.3E-2	21
HP_set					5.8E-3	25	1.2E-2	30	6.9E-3	27
HP_set_u					3.7E-3	38	7.2E-3	48	4.4E-3	42

TABLE 10. Perftrace report for backsolve for Problem III

where the loops are along skew grid diagonals (HP). For the multi-component PDEs we also timed the unrolled version HP_set_u, respectively, HP_u.

The performance of the SLAP ILU factorization routine SSILUS is, as expected, very poor, since it consists more of searching and sorting processes than of arithmetic computations. For the backsolve the vector length is ca. 4.`NPDE` resulting in better Megaflop rates. To judge the results for the hyperplane ordering in Tables 5-10 one should recall that the matrices generated are from relatively small test problems and so the ILU preconditioning will for these problems not attain a very good vector performance. As a reminder: the longest loop length for the 81×81 grids is 41 and for the 21×21 grids 11. For the grids on level 2 and 3 it ranges from 14 up to 26. As can be seen from the Perftrace reports the Megaflop rate is indeed not very impressive but the gain in CPU time in comparison with the SLAP code is large. For the factorization it ranges from a factor 100 for the scalar case to 30 for the 3-component system. Further, although the vector performance is low as a result of the small vector length, it is still a factor 5 faster than running in scalar mode.

6. Conclusions

In this paper we have discussed vectorizable implementations for the matrix-vector multiply, the standard Incomplete LU factorization and backsolve, for matrices that arise when solving systems of PDEs on a two-dimensional domain bounded by an arbitrary right-angled polygon and discretized in space using a 9-point stencil. The implemented matrix-vector multiply uses a block-diagonal storage with indirect addressing. The performance of this routine is already for a small number of nodes almost optimal, i.e., 1 result per clock cycle.

For the ILU factorization and backsolve concurrency is obtained by reordering the computations using a variant of the hyperplane method [1]. The performance of the standard hyperplane method reaches its optimum only for large systems, because the resulting vector length is half the number of nodes in the x-direction. The performance of the indirect addressing version has a Megaflop rate that is a factor 1.5 smaller. The reason for this, unexpected, performance drop is that the Cray Y-MP does not allow chaining of gather/scatter memory access. By explicitly reordering the rows of the matrix and the elements of the right hand-side vector the performance can be increased because the number of indirect memory accesses will be halved. A difficulty with respect to the explicit reordering is the fact that the order of computations for the backward solve is independent from the order of the factorization and the forward solve. Therefore explicit reordering will only speed up the factorization and the forward solve, but the backward solve will even need more indirect addressing. However, we can conclude that, even for the original, not explicitly reordered, version the gain in computational time compared with the general purpose routines of the SLAP library is significant, especially for the factorization.

References

[1] C. C. Ashcraft and R.G. Grimes. On vectorizing incomplete factorization and SSOR preconditioners. *SIAM J. Sci. Stat. Comput.*, 9(1):122–151, 1988.

[2] J.G. Blom, J.G. Verwer, and R.A. Trompert. A comparison between direct and iterative methods to solve the linear systems arising from a time-dependent 2D groundwater flow model. Report NM-R9205, CWI, Amsterdam, 1992. (to appear in Int. J. Comp. Fluid Dynamics).

[3] Cray Research, Inc. *UNICOS Performance Utilities Reference Manual*, SR-2040 6.0 edition.

[4] J.J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Commun. ACM*, 30:403–407, 1987. (netlib@research.att.com).

[5] S. Fujino and S. Doi. Optimizing multicolor ICCG methods on some vectorcomputers. In R. Beauwens, editor, *Proc. IMACS Int. Symp. Iterative Methods in Linear Algebra*. North-Holland, 1991.

[6] I. Gustafsson. A class of first order factorization methods. *BIT*, 18:142–156, 1978.

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

[8] P. Sonneveld. CGS: A fast Lanczos-type solver for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[9] R.A. Trompert and J.G. Verwer. Analysis of the implicit Euler local uniform grid refinement method. Report NM-R9011, CWI, Amsterdam, 1990. (to appear in SIAM J. Sci. Stat. Comput. Vol. 14, #2, 1993).

[10] R.A. Trompert and J.G. Verwer. Runge-Kutta methods and local uniform grid refinement. Report NM-R9022, CWI, Amsterdam, 1990. (to appear in Math. Comp.).

[11] R.A. Trompert and J.G. Verwer. A static-regridding method for two-dimensional parabolic partial differential equations. *Appl. Numer. Math.*, 8:65–90, 1991.

[12] R.A. Trompert, J.G. Verwer, and J.G. Blom. Computing brine transport in porous media with an adaptive-grid method. Report NM-R9201, CWI, Amsterdam, 1992. (to appear in Int. J. Numer. Meth. in Fluids, Vol. 15, 1992).

[13] J.G. Verwer and R.A. Trompert. An adaptive-grid finite-difference method for time-dependent partial differential equations. In D.F. Griffiths and G.A. Watson, editors, *Proc. 14-th Biennial Dundee Conference on Numerical Analysis*, pages 267–284. Pitman Research Notes in Mathematics Series 260, 1992.

[14] H.A. van der Vorst. BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2), 1992.

[15] H.A. van der Vorst and C. Vuik. GMRESR: A family of nested GMRES methods. Report 91-80, Faculty of Technical Mathematics and Informatics, TU Delft, the Netherlands, 1991.