Conjugate-gradient subroutines for the IBM 3090 Vector Facility

by Giuseppe Radicati di Brozolo
Marcello Vitaletti

This paper describes a set of optimized subroutines for use in solving sparse, symmetric, positive definite linear systems of equations using iterative algorithms. The set has been included in the Engineering and Scientific Subroutine Library (ESSL) for the IBM 3090 Vector Facility (VF). The subroutines are based on the conjugate-gradient method, preconditioned by the diagonal or by an incomplete factorization. They make use of storage representations of sparse matrices that are optimal for vector implementation. The ESSL vector subroutines are up to six times faster than a scalar implementation of the same algorithm.

Introduction

In the solution of large simulation problems, such as those encountered in fluid dynamics or structural analysis, the need often arises for the numerical solution of partial differential equations by finite-difference or finite-element methods. This, in turn, requires the solution of many very large linear systems of equations. In a typical problem there may be tens of thousands of unknowns, but only five to thirty nonzero coefficients per row of the associated "sparse" matrix, depending on the discretization technique used.

Systems of linear equations can be solved either by direct or by iterative algorithms. Direct methods that take advantage of the sparsity structure of the matrix are frequently used for problems of small to moderate size [1, 2]. However, direct methods are difficult to implement efficiently on vector processors because of the short length of the vectors involved in the computation. Iterative methods have proven very successful in the solution of large problems, because they generally require less storage and fewer arithmetic operations than direct sparse methods. Furthermore, iterative algorithms can be efficiently implemented on modern vector processors.

Release 2 of the Engineering and Scientific Subroutine Library (ESSL) includes two subroutines for the solution of large sparse, symmetric, positive definite linear systems of equations using the preconditioned conjugate gradient [3–5]; the subroutines are optimized for use with the IBM 3090 VF [6, 7]. The sparse-matrix vector product is one of the basic computational kernels in conjugate-gradient and other iterative methods. For matrices with a general sparsity pattern, this operation can be vectorized using gather–scatter vector operations if the matrix is stored in a suitable format [8]. Preconditioning by an incomplete Cholesky factorization [9–11] increases the computational cost of each conjugate-gradient iteration, but it generally accelerates the rate of

convergence, so that the global number of floating-point operations of the conjugate gradient preconditioned by an incomplete Cholesky factorization (ICCG) is less than that of the conjugate gradient preconditioned by the diagonal (DCG). Both preconditioning strategies are implemented in the ESSL subroutines.

In this paper we compare the efficiency of the ESSL iterative sparse-matrix subroutines to that of JCG, the FORTRAN implementation of the conjugate gradient preconditioned by the diagonal in ITPACK. ITPACK [12, 13] is a software library developed at the University of Texas at Austin, distributed as FORTRAN source code, which contains several adaptive accelerated iterative algorithms. In the comparisons both the scalar and vector versions of JCG were used.

The first test problem used to compare the performance was a linear system of 64000 equations arising from the discretization of an elliptic partial differential equation with mixed-type boundary conditions, on a three-dimensional $40 \times 40 \times 40$ regular grid. The second test problem was a linear system of 109375 equations arising from the discretization of the diffusion equation with mixed-type boundary conditions, on a three-dimensional irregular grid. On the IBM 3090 VF, by vectorizing ICCG we obtained speedups of a factor of two over its scalar version, whereas by vectorizing DCG we obtained speedups by factors of four to six. As a consequence, on some problems the vector version of DCG is more efficient than that of ICCG, even though it requires a larger number of floating-point operations.

Many applications require the solution of linear systems with the same coefficient matrix and different right-hand sides. When using the ESSL subroutines for ICCG, it is possible to take advantage of this because the preconditioner needs to be computed only once and can be used again for all the right-hand sides. The cost of computing the incomplete factorization is in general not very large, and is equivalent to that of a few iterations.

When an implicit time-differencing scheme is used to solve an initial-value partial differential equation, it is necessary to solve a sparse linear system of equations at each discrete time interval. If an iterative method is used and the solution vectors at successive time intervals are close to one another, the rate of convergence may be accelerated by using the solution at one interval as the initial approximation for the solution at the next interval.

Conjugate-gradient algorithm

The conjugate-gradient algorithm is an iterative algorithm for use in solving symmetric, positive definite systems of linear algebraic equations. The algorithm was originally proposed in 1952 by Hestenes and Stiefel [3]; it is discussed in [4, 5, 14]. It is often used in production-type codes for the solution of large sparse systems. In contrast to other iterative algorithms, such as successive over-relaxation (SOR), it requires no information on the value of extreme eigenvalues in order to estimate the optimal acceleration parameters. The algorithm can be accelerated effectively by various preconditioning strategies. (Preconditioning is essentially the transformation to an equivalent linear system with a lower condition number [10, 11, 15–18].)

Given A, a symmetric and positive definite matrix of order n, and a vector $\mathbf{y}$, the preconditioned conjugate-gradient algorithm is an iterative method for solving the system of linear equations

$$A\mathbf{x} = \mathbf{y}. \tag{1}$$

Given a suitable nonsingular preconditioning matrix $M \sim A$ and an initial approximation to the solution $\mathbf{x}_0$, the method is given by the following: set $\mathbf{r}_0 = \mathbf{y} - A\mathbf{x}_0$, and $\mathbf{p}_0 = M^{-1}\mathbf{r}_0$;

do for $i = 0, 1, 2, \cdots$ until convergence

$$\alpha_i = (\mathbf{r}_i, M^{-1}\mathbf{r}_i)/(\mathbf{p}_i, A\mathbf{p}_i), \tag{2a}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{p}_i, \tag{2b}$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i A\mathbf{p}_i, \tag{2c}$$

$$\beta_{i+1} = (\mathbf{r}_{i+1}, M^{-1}\mathbf{r}_{i+1})/(\mathbf{r}_i, M^{-1}\mathbf{r}_i), \tag{2d}$$

$$\mathbf{p}_{i+1} = M^{-1}\mathbf{r}_{i+1} + \beta_{i+1}\mathbf{p}_i, \tag{2e}$$

end do.

In steps (2a), (2d), and (2e) it is necessary to solve a linear system to compute $M^{-1}\mathbf{r}$. In an actual computer implementation, the solution is computed once per iteration and saved in a temporary array.

In exact arithmetic the conjugate-gradient algorithm converges to the solution in at most n iteration steps. This is not true in approximated arithmetic. The process may actually diverge for very ill-conditioned problems because of roundoff errors. It is essential to find a good preconditioning matrix $M \sim A$ to accelerate the convergence rate of the basic conjugate-gradient algorithm, particularly if the condition number of the matrix is large.

The problem is to find M so that the process converges in few iterations, and so that the cost of each iteration is not too large in order to reduce the global computational cost of the algorithm. This means that M must be easily invertible and must not require too much computer storage.

• *Preconditioning by the diagonal*

Here the preconditioning matrix is the main diagonal D_A of the matrix A. The resulting algorithm is designated as the Jacobi conjugate gradient, because it is equivalent to a polynomial acceleration of the basic Jacobi method [4]. The computation of $D^{-1}\mathbf{r}$ need not be carried out within the iteration loop. The algorithm may be simplified as follows:

1. Scale the matrix problem by computing $\tilde{A} = D_A^{-1/2} A D_A^{-1/2}$; $\tilde{\mathbf{y}} = D_A^{-1/2}\mathbf{y}$; $\tilde{\mathbf{x}} = D_A^{1/2}\mathbf{x}$. The scaled matrix $\tilde{A}$ has unit diagonal.
2. Apply the iterative process (2) to the system: $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$, where $M = I$, the unit matrix.
3. Scale back the solution: $\mathbf{x} = D_A^{-1/2}\tilde{\mathbf{x}}$.

• *Preconditioning by an incomplete factorization*

In this case, the preconditioning matrix $M = LDL^T$ is an incomplete factorization of A. Here L is the lower incomplete factor, with unit diagonal, and D is a diagonal matrix. Many alternative factorizations are possible. Usually L is computed with a modified Cholesky algorithm, forcing L to have the same sparsity pattern as A:

do for $i = 1, 2, \cdots, n$

do for $j = 1, 2, \cdots, i-1$

if $A_{ij} = 0$, then $L_{ij} = 0$,

else if $A_{ij} \neq 0$, then $L_{ij} = \dfrac{A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} D_k}{D_j}$

end do.

$L_{i,i} = 1$,

$D_i = A_{i,i} - \sum_{k=1}^{i-1} L_{i,k}^2 D_k$

end do.

Two sparse triangular systems must be solved at each iteration to compute $M^{-1}\mathbf{r}_i = LDL^T\mathbf{r}_i$. If L has the same sparsity pattern as A, this requires approximately the same number of arithmetic operations as the sparse-matrix vector multiplication. Therefore, the cost of one iteration is considerably higher than that of one iteration of DCG. However, the global number of arithmetic operations for the algorithm is generally lower, because the higher cost per iteration is more than offset by the faster convergence rate of the algorithm.

• *Stopping criteria*

In exact arithmetic the iterative procedure converges to the exact solution in a finite number of steps. In practice, the iteration procedure is terminated when the approximate solution $\bar{\mathbf{x}}$ is *close* to the exact solution:

$$\frac{\| \mathbf{x} - \bar{\mathbf{x}} \|_2}{\| \bar{\mathbf{x}} \|_2} \leq \varepsilon, \tag{3}$$

where ε is the desired relative accuracy. Because A is a symmetric, positive definite matrix,

$$\frac{\| \mathbf{x} - \bar{\mathbf{x}} \|_2}{\| \bar{\mathbf{x}} \|_2} = \frac{\| A^{-1}\bar{\mathbf{r}} \|_2}{\| \bar{\mathbf{x}} \|_2} \leq \lambda_M(A^{-1}) \frac{\| \bar{\mathbf{r}} \|_2}{\| \bar{\mathbf{x}} \|_2}, \tag{4}$$

where $\bar{\mathbf{r}} = \mathbf{y} - A\bar{\mathbf{x}}$ is the residual vector and $\lambda_M(A^{-1})$ is the maximum eigenvalue of A^{-1}. $\lambda_M(A^{-1}) = 1/\lambda_m(A)$, the minimum eigenvalue of the matrix A. Combining (3) and (4), we use

$$\frac{\| \bar{\mathbf{r}} \|_2}{\lambda_m(A) \| \bar{\mathbf{x}} \|_2} \leq \varepsilon \tag{5}$$

as a criterion to estimate the error in the solution to terminate the iteration procedure. If no information on the minimum eigenvalue is available, $\lambda_m(A)$ can be estimated adaptively in the iteration procedure at each iteration step as a function of the coefficients α_i and β_i [4, 5]. One obtains a monotonically decreasing sequence λ_m^i that converges to the minimum eigenvalue $\lambda_m(A)$. This adaptive computation is not costly.

Computer implementation

For each iteration step, a computer implementation of the conjugate-gradient algorithm requires three vector updates, two inner products, and one sparse-matrix vector product. In addition, preconditioning by an incomplete factorization requires the solution of two sparse triangular systems, which requires approximately the same number of floating-point operations as the sparse-matrix vector product. The first three kernels are efficiently vectorizable. The sparse-matrix vector product is vectorizable using indirect addressing vector instructions if the sparse matrix is stored in a suitable form. The solution of the sparse triangular systems is not vectorizable unless the sparse matrix has a special regular structure [19–23].

There are several ways to represent a sparse matrix in a computer; here we describe three of them. The row-wise representation of a sparse matrix is used in several software packages for scalar processors (ITPACK [12], SLMATH [24]), and is discussed, for example, by Gustavson [25]; this method is not used in Release 2 of ESSL. A second method, the compressed-matrix representation, has been adopted in ITPACKV [13], in ELLPACK [26], and in the iterative sparse-matrix routines in Release 2 of ESSL [27]. This storage mode is efficient on a vector processor if each row of the matrix has approximately the same number of nonzero elements. This is true in many sparse matrices that arise from the finite-element or finite-difference discretization of partial differential equations.

For structured sparse matrices, it is possible to use the compressed-diagonal storage mode. This scheme is the most rigid, since it can represent efficiently only matrices with a regular diagonal structure. Using this scheme, the sparse-matrix vector product operates on contiguous memory locations, and it is vectorizable without using indirect-addressing-type operations. The scheme is similar to that described by Madsen et al. [28].

In [8] the performance on the 3090 VF of a vector implementation of the sparse-matrix vector product using

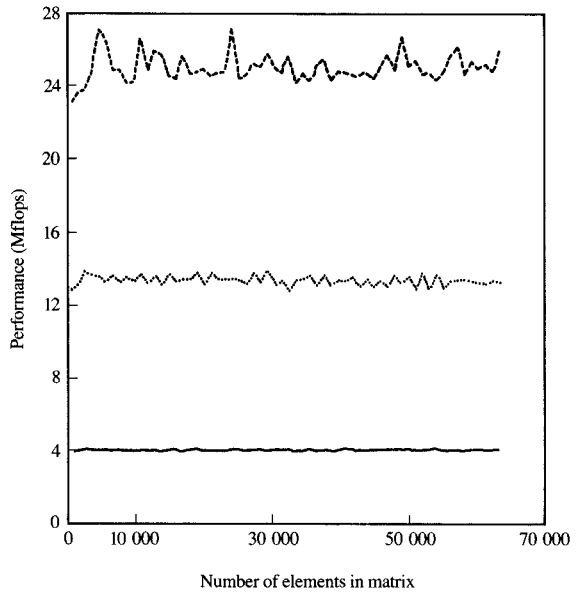

Figure 1

Performance of the sparse-matrix vector product vs. matrix size. The performance of a scalar implementation based on the representation by rows (solid curve) is compared to the performance of two vector implementations based on the compressed-matrix (dotted curve) and the compressed-diagonal (dashed curve) storage modes. The test matrix contained between 15 and 20 nonzero elements per row, arranged along its diagonals.

these storage representations was compared to that of a scalar code based on the row-wise representation of a sparse matrix, which is the most efficient on a scalar processor. The following was found:

- A speedup by a factor of two to four could be achieved for matrices stored in the compressed-matrix storage mode, where gather-scatter vector operations are needed to access the computer storage.
- A speedup by a factor of four to six could be achieved for matrices with diagonal structure stored in the compressed-diagonal mode.

In **Figure 1** we compare the performance of the matrix vector product using these three different storage techniques for a sparse matrix having between 15 and 20 nonzero elements per row.

Mehlem [20] described another data structure suitable for processing sparse matrices on vector computers. This scheme is a generalization of the diagonal representation, but is more rigid than the compressed-matrix representation. For general matrices it requires about the same amount of storage as the compressed-matrix storage mode, and on the 3090 VF it yields approximately the same performance for most problems. For symmetric matrices, if the scheme is applicable, it is possible to use only half the storage. When the matrices are very large and do not fit the core memory, the smaller memory requirements for symmetric problems permit a higher performance than use of the compressed-matrix storage because paging of data to secondary devices is reduced.

- *Storage by rows*

Given a general sparse $m \times n$ matrix A with ne nonzero elements, the matrix is represented by FORTRAN arrays: a real array AR and two integer arrays JA and IA set up as follows:

- The real array AR of length ne contains the nonzero elements of A, stored by row in contiguous memory locations.
- The integer array JA of length ne contains the column numbers of each nonzero element in A stored in the corresponding elements of AR.
- The integer array IA of length $m + 1$ contains pointers to the starting position of each row of A in arrays AR and JA. That is, each element IA(i) points to the beginning of row i in AR and JA. Row i contains IA($i + 1$) − IA(i) elements. In particular, IA(1) = 1 and IA($m + 1$) = $ne + 1$.

Within a row, the elements need not be ordered. The example in **Figure 2** illustrates this storage scheme on a 6×6 sparse matrix which contains 18 nonzero entries.

The following FORTRAN program performs the matrix vector product using this storage scheme:

```
      DO 20 I=1,M
         Y(I) = 0.
         DO 10 J = IA(I), IA(I+1)-1
            Y(I) = Y(I) + X(JA(J))*AR(J)
10       CONTINUE
20    CONTINUE
```

Arrays AR and JA are accessed in an orderly fashion. Array X, on the contrary, is accessed randomly with addresses specified by JA.

If the matrix is symmetric, only half of the matrix coefficients need to be stored. Usually the upper triangle is stored. In this case, the sparse-matrix vector product is performed with random accesses to both the X and Y arrays. Thus, although only half of the memory is required to represent the matrix, the performance of the sparse symmetric-matrix vector product is comparable to that of the general case. If, however, the matrix does not fit in the core memory, the saving in memory can be crucial in minimizing paging.

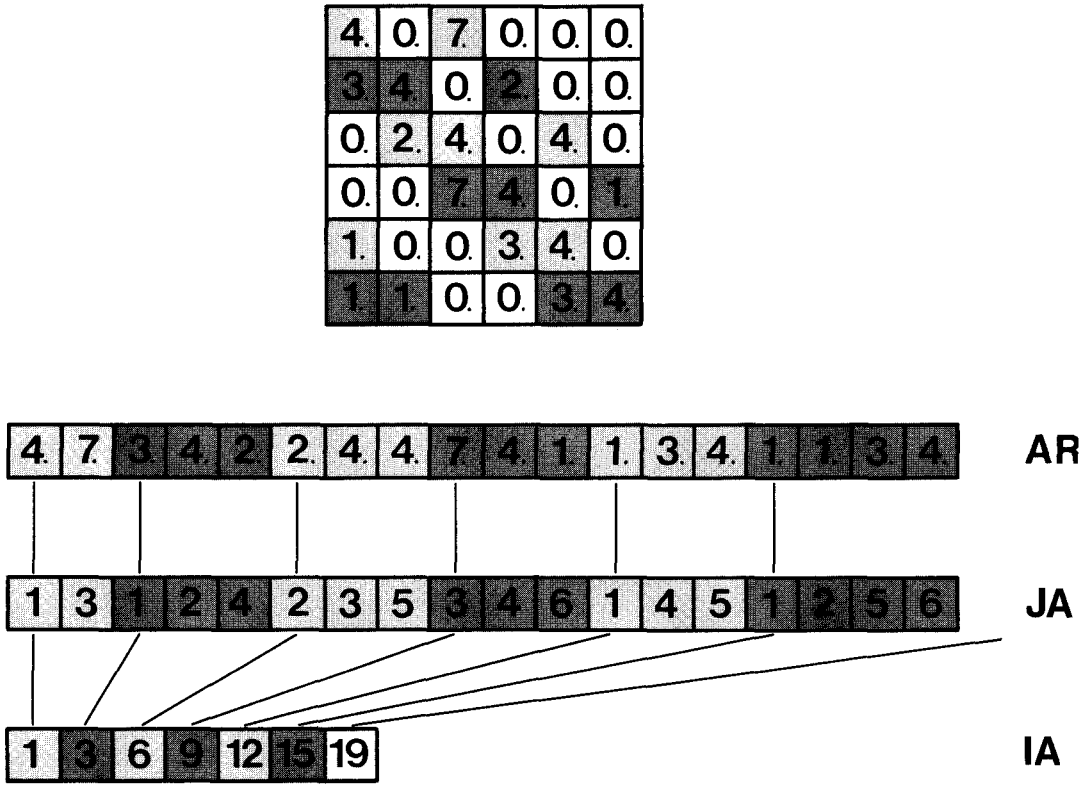

Figure 2

Storage by rows of a sparse matrix.

In this algorithm $Y(i)$ is computed by performing an inner product of the nonzero elements in the ith row by the corresponding elements of the X array. On a scalar processor, the matrix vector product using the row-wise representation is efficient because only the nonzero elements of A are processed. For this reason, the row-wise representation has been used in several sparse-matrix software packages for scalar processors. On a vector processor, by using this matrix representation, the sparse-matrix vector product can be vectorized with the use of indirect-addressing-type operations to collect the elements of the X array involved in each inner product. The vector code will, however, be inefficient because the vectors involved in the operations are very short. On the 3090 VF, when there are 15 or fewer elements per row, the scalar version of this algorithm is faster than the vector version because of the start-up time for vector instructions.

- *Compressed-matrix storage mode*

Given a general sparse $m \times n$ matrix A having a maximum of nz nonzero elements in each row, the matrix is represented using a real array AC and an integer array KA as follows:

- The real array AC contains m rows and nz columns: Each row of AC contains the nonzero elements of the corresponding row of the matrix A. If a row of A has fewer than nz nonzero elements, the corresponding row in AC is padded with zeros. The elements within a row can be stored in any order.
- The integer array KA contains m rows and nz columns; it contains the column numbers of the nonzero elements of matrix A that are stored in the corresponding positions in array AC. If the corresponding element in AC is zero, any index in the range $1 \cdots n$ may be used.

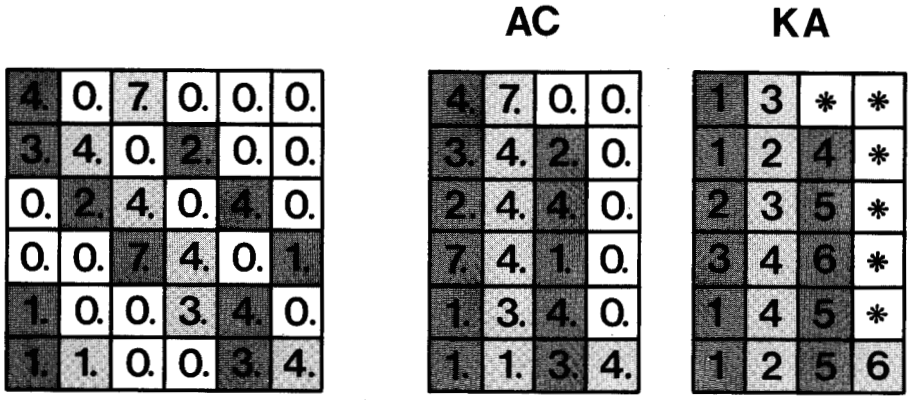

Figure 3

Sparse matrix in compressed-matrix storage mode.

Unless all the rows of the sparse matrix have approximately the same number of nonzero elements, this storage scheme will require a large amount of storage. **Figure 3** illustrates the scheme on the same matrix used in the previous example. In this case $m = n = 6$ and $nz = 4$. An asterisk (*) in the array KA indicates that any valid index may be used because the value of the corresponding element in the array AC is zero.

The following is an example of a FORTRAN program which might be used to perform the matrix vector product using this storage scheme:

```
      DO 10 I=1,M
         Y(I) = 0.
10    CONTINUE
      DO 30 J=1,NZ
         DO 20 I=1,M
            Y(I) = Y(I) + X(KA(I,J))*AC(I,J)
20       CONTINUE
30    CONTINUE
```

Arrays AC and KA are accessed in an orderly fashion. Array X, on the contrary, is accessed randomly with addresses specified by KA. The computation of Y is achieved by summing nz vectors of length m that are the result of an element-by-element product of two vectors. This algorithm is the equivalent for sparse matrices of the matrix vector multiplication algorithm for full matrices based on SAXPY [29].

On a scalar processor, use of this algorithm does not result in optimal performance because, if some rows have less than nz elements, it will result in the execution of a certain number of multiplications by zero. On a vector processor, when this representation is used, the matrix vector multiplication vectorizes by using indirect-addressing-type operations to collect the elements of array X involved in each element-by-element product. In this case the vector operations have length m. Moreover, the result of one element-by-element vector multiplication may be kept in a vector register and added directly to the result of the successive multiplication, thus saving loads and stores to memory.

On the 3090 VF, the matrix vector product using this storage scheme may be implemented using the following vector instructions to perform the gather and the nz element-by-element vector products:

- *VL* (*load integer vector*): Load pointer integers from array KA.
- *VLID* (*load direct*): Load elements of array X using the pointer integers (gather).
- *VMAD* (*multiply and add*): Perform element-by-element vector product of elements of X and a column of AC, which is referenced directly from memory, and sum to previous result. This instruction executes a floating-point multiplication and addition in each machine cycle.

After the nz element-by-element vector products have been computed, the result is contained in a vector register and must be stored with a vector instruction. In this algorithm, the vector hardware is used efficiently because all of the vector instructions are executed on long vectors, and the use of data in the vector register is optimized. Using long vectors minimizes the impact of the start-up time.

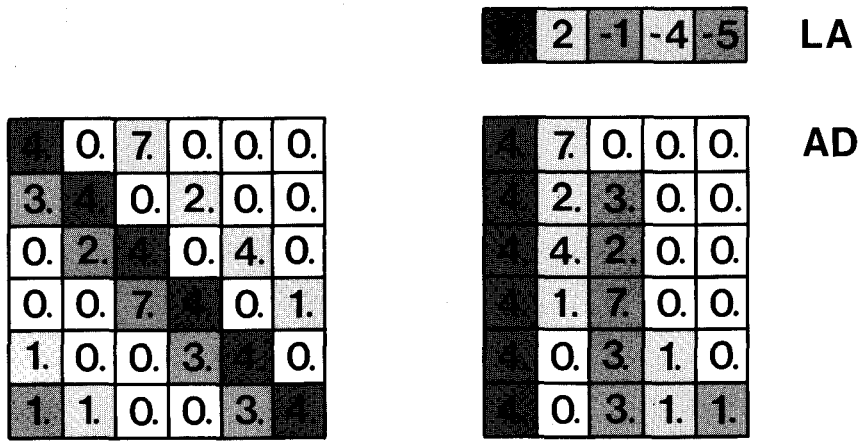

Figure 4

Sparse matrix in compressed-diagonal storage mode.

• *Compressed-diagonal storage mode*

Let A be a sparse $n \times n$ matrix with nd diagonals with nonzero entries. The diagonal number $k = j - i$ of element a_{ij} is a constant along each diagonal of the matrix; it is positive for the superdiagonals and negative for the subdiagonals. The matrix is stored using a rectangular array AD, and an array LA of pointers:

- AD is a real array, with n rows and nd columns. AD contains the diagonals of A that have at least one nonzero entry. These diagonals, including all the zeros, are stored in a column of AD in n contiguous memory locations. The superdiagonals are padded to a length n with k trailing zeros, and the subdiagonals are padded with $|k|$ leading zeros, where k is the diagonal number. Each nonzero element a_{ij} of the original matrix A is stored in row i of array AD.
- LA is an integer array of length nd. LA(i) is the diagonal number of the diagonal stored in column i of AD.

This matrix representation requires that entire diagonals be stored, and requires a large amount of memory if the nonzero entries of A are not concentrated along a few diagonals. If the matrix is symmetric, only the main diagonal and one of each couple of identical diagonals k and $-k$ need to be stored in array AD.

Figure 4 illustrates this storage scheme on the same matrix used in Figure 2. In this case $n = 6$ and $nd = 5$.

The following is a FORTRAN program to perform the matrix vector product using this storage scheme:

```
      DO 10 I = 1,N
         Y(I) = 0.
10    CONTINUE
      DO 30 L = 1,ND
         K = LA(L)
         N1 = MAX(1,1-K)
         N2 = MIN(N,N-K)
         DO 20 I = N1,N2
            Y(I) = Y(I) + X(K+I)*AD(I,L)
20       CONTINUE
30    CONTINUE
```

In this program, Y is computed by summing nd vectors of length n that are the result of an element-by-element product of two vectors. All of the arrays involved in the operations are stored in contiguous memory locations, while in the algorithm for the compressed-matrix representation it is necessary to gather elements of array X from noncontiguous memory locations.

This storage scheme is efficient on both scalar and vector processors, because the matrix vector product does not require random accesses to the elements of array X and may therefore be implemented with fewer instructions; furthermore, the local memory is used more efficiently. The scheme is efficient provided the matrix has a diagonal structure; otherwise the product would require a large number of multiplications by zero.

Table 1 Performance of DSMCG and DSDCG using diagonal preconditioning (64000 equations).

Subroutine	*Iterations*	*CPU seconds*	*Mflops*	*Speedup*
JCG (ITPACK)	231	74	5	1
JCG (ITPACKV)	231	28	12	2.6
DSMCG	231	19	20	4
DSDCG	231	13	30	6

Table 2 Performance of DSMCG and DSDCG preconditioned by an incomplete factorization (64000 equations).

Subroutine	*Iterations*	*CPU seconds*	*Mflops*	*Speedup*
ICCG (scalar)	74	46	4	1
DSMCG	74	20	9	2
DSDCG	74	18	9	2

The matrix vector product using this storage scheme may be implemented on the 3090 VF using the following vector instruction to compute the *nd* element-by-element vector products:

- *VLD (load)*: Load elements of array X. Different contiguous segments of X are involved in each element-by-element vector product.
- *VMAD (multiply and add)*: Perform element-by-element vector product of the segment of X and a column of AD, which is referenced directly from memory, and sum to result of previous element-by-element vector product.

In this case the vector hardware is used efficiently because all the vector operations are executed on long vectors that lie in contiguous memory locations.

- *Sparse-matrix subroutines*

Release 2 of ESSL includes two routines to compute the sparse-matrix vector product, and two routines for solving a system of linear equations using the preconditioned conjugate-gradient algorithm:

- DSMMX: To compute the matrix vector product for a sparse matrix in compressed-matrix storage mode.
- DSDMX: To compute the matrix vector product for a sparse matrix in compressed-diagonal storage mode.
- DSMCG: To solve a symmetric, positive definite system of linear equations, using the conjugate-gradient method, for a sparse matrix stored in compressed-matrix storage mode. Depending on the value of a parameter, either the conjugate gradient preconditioned by the diagonal or the conjugate gradient preconditioned by an incomplete factorization is used.
- DSDCG: To solve a symmetric, positive definite system of linear equations, using the conjugate-gradient method, for a sparse matrix stored in compressed-diagonal storage mode. Depending on the value of a parameter, either the conjugate gradient preconditioned by the diagonal or the conjugate gradient preconditioned by an incomplete factorization is used.
- DSMTM: To transpose a sparse matrix stored in compressed-matrix storage mode.
- DSRSD: To convert a sparse matrix from row-wise storage mode to compressed-matrix storage mode.

Performance of the conjugate-gradient subroutines

In this section we discuss the performance of the conjugate-gradient subroutines DSMCG and DSDCG. Our aim is to discuss how the representation used to store a sparse matrix affects the performance of the conjugate-gradient algorithm on a vector processor. The CPU time needed to compute the solution with subroutines DSMCG and DSDCG is compared to that of the scalar and vectorized versions of JCG, the FORTRAN implementation of the conjugate gradient preconditioned by the diagonal from the ITPACK-2C and ITPACKV libraries, and ICCG, a scalar FORTRAN implementation of the conjugate gradient preconditioned by an incomplete factorization.

The JCG subroutine in ITPACK-2C is based on the row-wise storage mode, which is the most efficient storage mode for computing the sparse-matrix vector product in *scalar* mode, while the JCG subroutine in ITPACKV is based on the compressed-matrix representation, and the sparse-matrix vector product is vectorized by the FORTRAN compiler. ITPACK does not contain an implementation of the conjugate gradient preconditioned by an incomplete factorization. The FORTRAN subroutines were compiled with the IBM VS FORTRAN Version 2 compiler at the highest level of optimization.

- *Test problem 1: An elliptic, three-dimensional, partial differential equation*

Consider first the system of linear equations generated by the numerical solution of the following self-adjoint elliptic partial differential equation on the unit cube

$$u_{xx} + 2u_{yy} + 3u_{zz} = 0 \qquad (6)$$

with mixed-type boundary conditions. This is one of the test problems provided with ITPACK to test the iterative algorithms; it is described in [12].

Equation (6) is discretized using a three-dimensional regular mesh. Using the standard seven-point finite-difference operator, this problem gives rise to a system of order $N = 1/h^3$ (h is the grid spacing). Each point has six *nearest neighbors*, and hence each row of the coefficient matrix has at most seven nonzero elements, which are

arranged along seven diagonals. This allows the matrix to be stored efficiently in both the compressed-matrix and the compressed-diagonal storage modes. In both storage representations almost no zeros need to be stored; using either representation, the matrix vector product can be performed efficiently.

We considered the linear system of 64000 equations arising from the discretization of Equation (6) on a mesh of $40 \times 40 \times 40$ gridpoints. Sparse-matrix problems of this size require between 10 and 30 megabytes of storage depending on the algorithm, and therefore fit in the core memory of the 3090. No paging to secondary storage devices is necessary, and as a consequence the CPU time is equivalent to the elapsed time in a dedicated environment.

If the preconditioned path of the DSMCG and DSDCG subroutines is chosen, an incomplete Cholesky factorization must be computed. The factorization can be used in the solution of other linear systems with the same coefficient matrix and a different right-hand side. The cost of the factorization in terms of iterations is largely independent of the size of the problem. **Table 1** compares the performance of the different scalar and vector implementations of DCG. **Table 2** compares the performance of the different scalar and vector implementations of ICCG. The time includes two seconds spent in computing the incomplete factorization. The measures were obtained on an IBM 3090 with a cycle time of 18.5 nanoseconds.

• *Test problem 2: Three-dimensional diffusion equation*
Consider the system of linear equations generated by the numerical solution of the diffusion equation over the unit cube with a corner missing (**Figure 5**),

$$\frac{\partial F}{\partial t} = \nabla(D \nabla F), \tag{7}$$

where D and F are scalar fields. Mixed-type boundary conditions were chosen. This test is a three-dimensional generalization of the model test problem proposed by Kershaw in [10]. We solve the linear system generated by the initial value problem at a given time step using implicit time differencing.

Each row of the sparse coefficients matrix has at most seven nonzero elements. Because the mesh is not regular, the nonzero coefficients are no longer arranged along seven diagonals. The nonzero coefficients result in the sparsity pattern shown in **Figure 6**, which can be stored efficiently in compressed-matrix storage mode but not in compressed-diagonal storage mode. We considered the linear system of 109375 equations arising from the discretization of Equation (7) on a mesh of $50 \times 50 \times 50$ gridpoints. The nonzero coefficients are arranged along 59 diagonals, which would make the use of the compressed-diagonal storage mode inefficient. This problem fits in the core memory of the 3090. **Table 3** compares the performances of the different scalar and vector implementations of DCG, while **Table 4** compares the performances of the different scalar and vector implementations of ICCG. The time includes three seconds to compute the incomplete factorization.

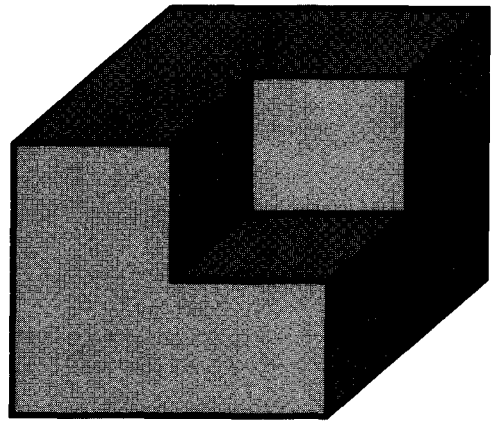

Figure 5
Irregular computational domain over which the diffusion equation was solved.

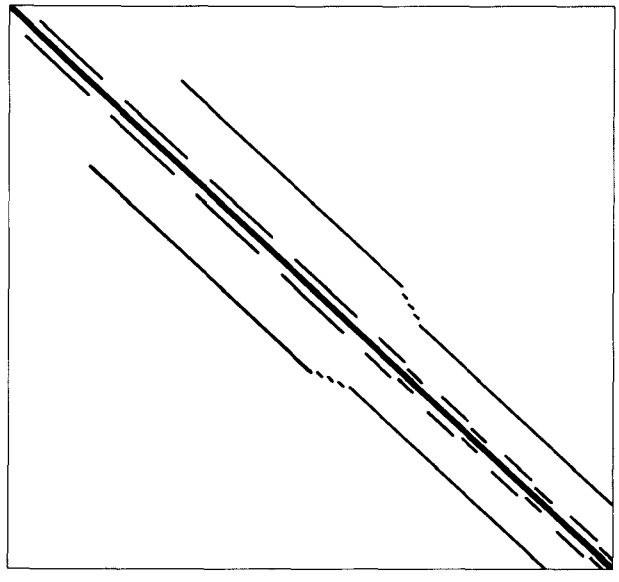

Figure 6
Typical sparsity pattern generated using a seven-point difference scheme over the computational domain in Figure 5.

Table 3 Performance of DSMCG using diagonal preconditioning (109 375 equations).

Subroutine	*Iterations*	*CPU seconds*	*Mflops*	*Speedup*
JCG (ITPACK)	85	46	5	1
JCG (ITPACKV)	85	33	12	2.6
DSMCG	85	12	19	4
DSDCG	Not applicable			

Table 4 Performance of DSMCG preconditioned by an incomplete factorization (109 375 equations).

Subroutine	*Iterations*	*CPU seconds*	*Mflops*	*Speedup*
ICCG (scalar)	30	30	4	1
DSMCG	30	16	9	2
DSDCG	Not applicable			

Table 5 Performance of DSMCG with different types of preconditioning, for solving system of 1473 equations involving use of the matrix BCSSTM12.

Subroutine	*Iterations*	*CPU seconds*	*Speedup*
JCG (ITPACK)	164	1.62	1
DSMCG (diagonal)	164	0.75	2.1
DSMCG (Cholesky)	11	0.25	6.5

• *Test problem 3: Harwell–Boeing sparse-matrix collection*
Duff et al. [30] have described a set of sparse test matrices from a wide variety of scientific and engineering disciplines, which is offered as a standard benchmark for comparative studies of algorithms. We have used several of the symmetric matrices to test the performance, accuracy, and robustness of the ESSL sparse-matrix subroutines. We summarize briefly the results we have obtained in solving a linear system of equations involving use of the matrix BCSSTM12, which are rather typical of the results we obtained with this set of test matrices. **Table 5** compares the performance of the ITPACK scalar subroutine JCG with diagonal preconditioning and DSMCG with two different types of preconditioning. The matrix BCSSTM12 has order 1473, 10 566 nonzero elements, and a maximum of 22 nonzero elements per row. When the compressed-matrix storage is used, 40% of the matrix entries in the sparse representation are zero. Both the subroutine JCG and the subroutine DSMCG with diagonal preconditioning require 164 iterations to reach convergence. The speedup of the vectorized version is 2.15, even though 40% of the arithmetic operations in the sparse-matrix vector product involve zeros.

The subroutine DSMCG preconditioned by an incomplete factorization requires only 11 iterations to converge; it is three times faster, in this experiment, than DSMCG preconditioned by the diagonal. These figures are typical of other problems in the Harwell–Boeing sparse-matrix collection.

Conclusions

With the use of the scalar FORTRAN codes, preconditioning by an incomplete factorization generally yields a better performance than preconditioning by the diagonal. The greater cost per iteration is more than offset by the large reduction in the number of iterations. Both scalar algorithms perform at roughly five Mflops, and both make good use of the IBM 3090, even though its memory is accessed randomly.

Using the conjugate-gradient vector subroutines in ESSL results in a significant speedup. The vectorized conjugate gradient preconditioned by the diagonal takes full advantage of the vector features of the IBM 3090 VF. The subroutines perform at between 20 and 30 Mflops depending on the sparse-matrix representation. On the contrary, only about half of the operations in the conjugate gradient preconditioned by an incomplete factorization are vectorizable. The solution of the two triangular systems is a recursive algorithm which accounts for almost half of the operations required. As a consequence, the subroutines make only a moderate use of the vector feature, as the 9-Mflops performance indicates. Whether or not preconditioning by an incomplete factorization will improve the overall performance of the conjugate-gradient algorithm in a vector environment depends on the problem under consideration. For the first two problems we described, which had a moderate condition number, the use of the conjugate gradient preconditioned by the diagonal yielded a higher performance than the use of the conjugate gradient preconditioned by an incomplete factorization—although the total number of floating-point operations of the preconditioned algorithm was smaller. On the contrary, other examples, such as our third test problem, indicated that for ill-conditioned problems, preconditioning significantly improves overall performance in a vector as well as in a scalar environment.

Acknowledgments

We would like to thank the members of the ESSL software development group of the IBM Data Systems Division at Kingston, New York, coordinated by S. Schmidt, for their valuable support, and for helping to develop our programs into IBM products. We also wish to thank the members of the Mathematical Sciences Department of the IBM T. J. Watson Research Center at Yorktown Heights, New York, and, in particular, F. Gustavson and J. B. Shearer, for their help and advice. Our thanks are also due to Y. Robert of the CNRS Laboratoire TIM3, Grenoble, France, and to I. Efrat of the IBM Haifa Scientific Center, Israel, for several stimulating discussions and suggestions.

References

1. A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
2. A. George, J. W. Liu, and E. Ng, "User Guide for SPARSPAK: Waterloo Sparse Linear Equations Package," University of Waterloo, Ontario, Canada, 1980.
3. M. R. Hestenes and E. Stiefel, "Methods of Conjugate-Gradients for Solving Linear Systems," *J. Res. Nat. Bur. Stand.* **49,** 409–436 (1952).
4. L. A. Hagemann and D. M. Young, *Applied Iterative Methods*, Academic Press, Inc., New York, 1981.
5. G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 1983.
6. *IBM System/370 Vector Operations*, Order No. SA22-7125, IBM Corporation; available through IBM branch offices.
7. W. Buchholz, "The IBM System/370 Vector Architecture," *IBM Syst. J.* **25,** 51–62 (1986).
8. G. Radicati di Brozolo and M. Vitaletti, "Sparse Matrix Vector Product and Storage Representations on the IBM 3090 with Vector Facility," *Technical Report G513-4098*, IBM Italy, European Center for Scientific and Engineering Computing, Via Giorgione 159, 00147 Rome, Italy, 1986.
9. J. A. Meijerink and H. A. van der Vorst, "An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix Is a Symmetric M-Matrix," *Math. Comp.* **31,** 148–162 (1977).
10. D. S. Kershaw, "The Incomplete Cholesky-Conjugate-Gradient Method for the Iterative Solution of Systems of Linear Equations," *J. Comp. Phys.* **26,** 43–65 (1978).
11. O. Axelsson, "A Survey of Preconditioned Iterative Methods for Linear Systems of Algebraic Equations," *BIT* **25,** 166–187 (1985).
12. D. R. Kincaid, J. R. Respess, D. M. Young, and R. G. Grimes, "ALGORITHM 586—ITPACK 2C: A FORTRAN Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods," *ACM Trans. Numer. Soft.* **8,** 302–322 (1982).
13. D. R. Kincaid, T. C. Oppe, J. R. Respess, and D. M. Young, "ITPACKV 2C User's Guide," *CNA-191*, Center for Numerical Analysis, University of Texas at Austin, Texas, 1984.
14. V. Faber and T. Manteuffel, "Necessary and Sufficient Conditions for the Existence of a Conjugate-Gradient Method," *SIAM J. Numer. Anal.* **21,** 352–362 (1984).
15. P. Concus, G. H. Golub, and D. P. O'Leary, "A Generalized Conjugate-Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations," *Sparse Matrix Computations*, R. Bunch and D. J. Rose, Eds., Academic Press, Inc., New York, 1976, pp. 309–332.
16. O. G. Johnson, C. A. Micchelli, and G. Paul, "Polynomial Preconditioners for Conjugate-Gradient Calculations," *SIAM J. Numer. Anal.* **20,** 362–376 (1983).
17. Y. Saad, "Polynomial Preconditionings for the Conjugate-Gradient Method," *SIAM J. Sci. Stat. Comput.* **6,** 865–881 (1985).
18. T. A. Manteuffel, "An Incomplete Factorization Technique for Positive Definite Linear Systems," *Math. Comp.* **34,** 473–497 (1980).
19. D. S. Kershaw, "Solution of Single Tridiagonal Linear Systems and Vectorization of the ICCG Algorithm on the Cray-1," *Parallel Computations*, G. Rodrigue, Ed., Academic Press, Inc., New York, 1982, pp. 85–99.
20. R. Mehlem, "Toward Efficient Implementation of Preconditioned Conjugate Gradient on Vector Supercomputers," *The International Journal of Supercomputer Applications* **1,** 70–98 (1987).
21. G. Meurant, "The Block Preconditioned Conjugate-Gradient Method on Vector Computers," *BIT* **24,** 623–633 (1984).
22. H. A. van der Vorst, "The Performance of FORTRAN Implementations for Preconditioned Conjugate-Gradients on Vector Computers," *Parallel Comp.* **3,** 49–58 (1986).
23. H. A. van der Vorst, "A Vectorizable Variant of Some ICCG Methods," *SIAM J. Sci. Stat. Comput.* **3,** 350–356 (1982).
24. *Subroutine Library—Mathematics, User's Guide*, Order No. SH12-5300, IBM Corporation; available through IBM branch offices.
25. F. G. Gustavson, "Sparse Matrix Methods," *Research Report RC-8330*, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1980.
26. *Solving Elliptic Problems Using ELLPACK*, J. R. Rice and R. Boisvert, Eds., Springer-Verlag, New York, 1985.
27. *Engineering and Scientific Subroutine Library Guide and Reference*, Order No. SC23-0184, IBM Corporation; available through IBM branch offices.
28. N. K. Madsen, G. H. Rodrigue, and J. I. Karush, "Matrix Multiplication by Diagonals on a Vector/Parallel Processor," *Info. Process. Lett.* **5,** 41–45 (1979).
29. J. Dongarra, F. Gustavson, and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Rev.* **26,** 91–112 (1984).
30. I. S. Duff, R. G. Grimes, and J. G. Lewis, "Sparse Matrix Test Problems," *Report CSS 191*, Computer Science and System Division, Harwell Laboratory, Oxford, England, 1987.

Received August 25, 1987; accepted for publication June 28, 1988

Giuseppe Radicati di Brozolo *IBM Italy, European Center for Scientific and Engineering Computing, Via Giorgione 159, 00147 Rome, Italy.* Dr. Radicati received a doctorate in mathematics from the University of Pisa in 1981. He joined IBM in 1982 and initially worked on seismic data processing. In 1985 he participated in the design and development of VPSS/VF, the simulator of the IBM 3838 Array Processor for the IBM 3090 VF. In 1986 he helped design and develop the sparse-matrix codes for ESSL, the Engineering and Scientific Subroutine Library for the 3090 VF. Dr. Radicati's current research interests involve the study of algorithms for parallel and vector architectures.

Marcello Vitaletti *IBM Italy, European Center for Scientific and Engineering Computing, Via Giorgione 159, 00147 Rome, Italy.* Dr. Vitaletti received his doctorate in physics from the University of Rome in 1981. He joined IBM in 1984, and participated in the design and the development of VPSS/VF in 1985. Since 1986 he has been involved in the design and development of the ESSL for sparse systems of linear equations. Dr. Vitaletti's current interest is in the area of computational fluid dynamics, with emphasis on algorithms for vector and parallel architectures.