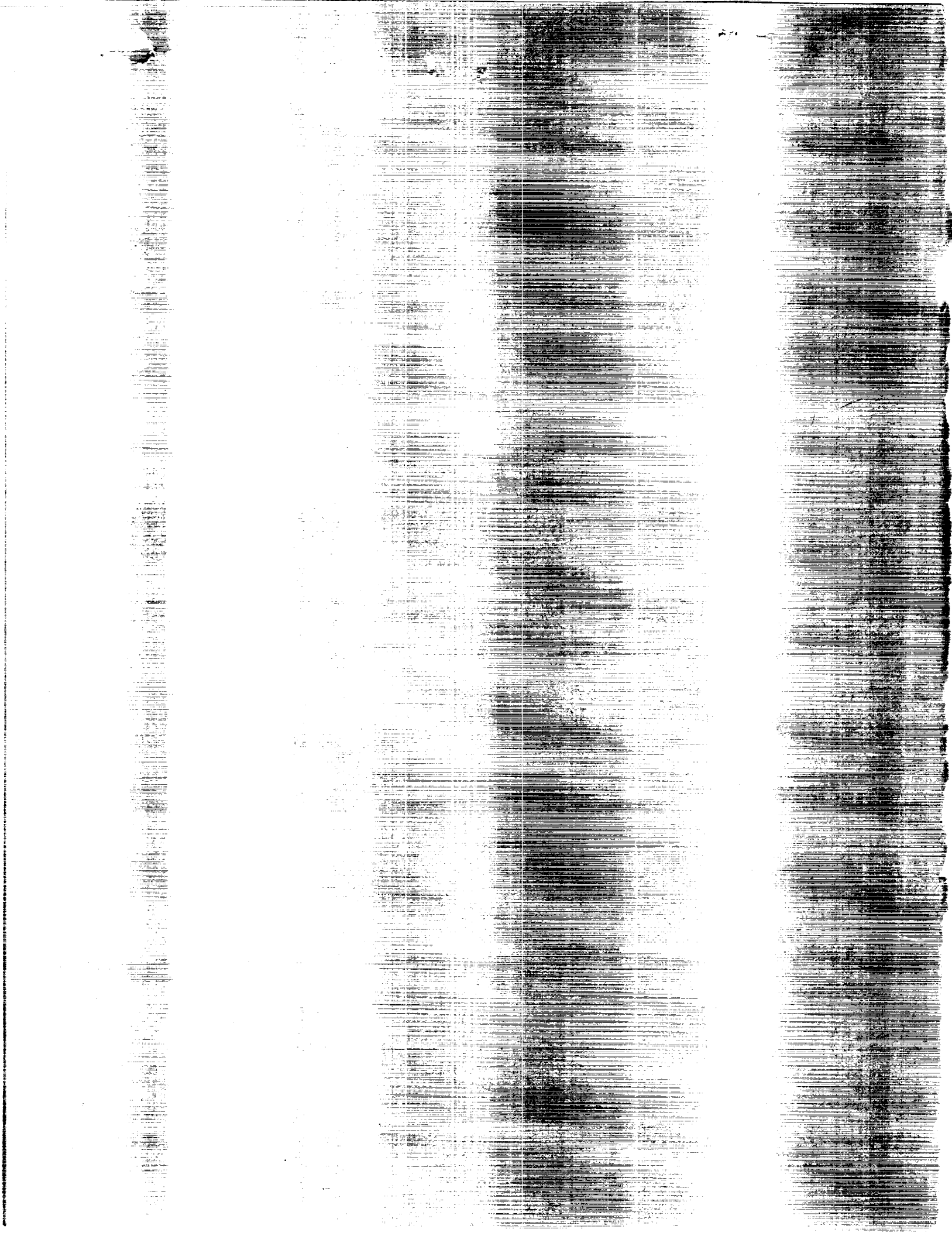


NASA Contractor Report 4159

The Solution of Linear Systems of Equations With a Structural Analysis Code on the NAS CRAY-2

Eugene L. Poole and Andrea L. Overman
Awesome Computing Inc.
Charlottesville, Virginia

Prepared by
Awesome Computing Inc.
for Analytical Services and Materials, Inc., for
NASA Langley Research Center
under Contract NAS1-18599

National Aeronautics
and Space Administration

**Scientific and Technical
Information Division**

1988

SUMMARY

Two methods for solving linear systems of equations on the NAS Cray-2 are described. One is a direct method; the other is an iterative method. Both methods exploit the architecture of the Cray-2, particularily the vectorization, and are aimed at structural analysis applications. To demonstrate and evaluate the methods, they were installed in a finite element structural analysis code denoted the Computational Structural Mechanics (CSM) Testbed. A description of the techniques used to integrate the two solvers into the Testbed is given. Storage schemes, memory requirements, operation counts, and reformatting procedures are discussed. Finally, results from the new methods are compared with results from the initial Testbed sparse Choleski equation solver for three structural analysis problems. The new direct solvers described in this report achieve the highest computation rates of the methods compared. The new iterative methods are not able to achieve as high computation rates as the vectorized direct solvers but are best for well-conditioned problems which require fewer iterations to converge to the solution.

THE SOLUTION OF LINEAR SYSTEMS OF EQUATIONS WITH A STRUCTURAL ANALYSIS CODE ON THE NAS CRAY-2

Eugene L. Poole and Andrea Overman
Awesome Computing Inc.
NASA Langley Research Center
Hampton, Virginia 23665-5225

INTRODUCTION

The solution of linear systems of equations on advanced parallel and vector computers is an important area of ongoing research. Much progress has been made in the development of algorithms which exploit advanced computer architectures. (See Ortega and Voigt[1] for a comprehensive review of many of these algorithms.) The major benefit of these algorithms is realized when they can be used to solve large-scale scientific applications problems. This report describes two equation solvers, one a direct method and the other an iterative method, which exploit the architecture of the NAS (National Aerodynamic Simulator) Cray-2 supercomputers. A description of the incorporation of both equation solvers into a very large finite element structural analysis code is given, and results are presented for three example structural analysis problems. The reduction of overall analysis time for each problem resulting from the new equation solvers is demonstrated.

The methods described in this report are used to solve the linear system of equations that occur in structural analysis applications. The form of the equations is

$$Ku = f \tag{1}$$

where K is assumed to be the symmetric, positive definite stiffness matrix, f is the load vector and u is the vector of unknowns, typically the displacements. Such linear systems can be large (from several thousands of unknowns to several hundred thousand unknowns) and often require significant computing resources, for both memory and execution time. The structure of the stiffness matrices in these applications is often sparse, although in many applications an ordering of the nodes which minimizes the bandwidth makes banded or profile (variable bandwidth) type storage of the matrices practical. The choice of a particular method to solve $Ku = f$ will depend on the non-zero structure of K and, in the case of the iterative methods described here, the condition number of K. In addition, the architecture of the computer, particularly for modern vector and parallel computers, influences both the choice and implementation of methods used to solve equation (1).

The organization of this report is as follows. First, the direct and iterative methods used in this research are described, including key implementation details for the NAS Cray-2. Second, the incorporation of these methods within a large-scale finite element structural analysis code is outlined. Finally, results are presented to demonstrate the performance of these equation solvers for three structural analysis applications problems.

DESCRIPTION OF METHODS

Direct methods usually consist of a factorization of the matrix K into triangular factors followed by the forward and backward solution of the resulting triangular systems. Iterative methods generally proceed from an initial guess, u^0, for the solution of (1) and, through an iterative process, refine the guess to a close approximation, u^k, of the exact solution. The choice of direct or iterative methods depends upon several factors which affect the relative performance of the methods. The following sections describe a direct Choleski method and a preconditioned conjugate gradient iterative method, and compare both methods by contrasting their memory requirements and several performance factors.

Direct Methods: Choleski

This section describes several implementations of the Choleski factorization, $K = LL^T$, for symmetric, positive definite linear systems. Here, L is a lower triangular matrix and its transpose is L^T. Only the lower triangular part of K is stored, and L, which is computed by modifying K, is stored in the same space as the original matrix K. The factorization can be carried out in many ways but, in general, elements of L are computed from K a row (or column) at a time beginning with row (or column) 1 of K and proceeding to the last row (or column). Additionally, the matrices are stored either by rows or by columns, and the various combinations of storage assignment and order of computation provide many possible implementations of Choleski factorization. (See Ortega[2] for a thorough description of these various implementations for vector and parallel computers.)

Implementation Considerations. The implementation chosen for Choleski factorization is guided by the architecture of the computer used for the computations. For vector computers such as the NAS Cray-2, algorithms which access vectors stored contiguously are preferred to algorithms which access vectors using constant stride or integer array references (indirect addressing). Also, on vector computers vector add-multiply updates (*vector* + *vector* × *scalar*), also referred to as *saxpy* operations, are preferred over inner products when a choice is possible. The basic implementation chosen for the Cray-2, sometimes referred to as the *kji form*, is illustrated in figure 1 for an $n \times n$ matrix with semi-bandwidth m. This implementation is also referred to as an *immediate update* algorithm meaning that as each column of L is computed, it is used to update all remaining columns of K within the band. For each stage, k, a column of L is computed by a vector divide (Loop2). Then columns $k+1$ through $k+m$ of K are updated using column k of L (Loop3). The key computation in figure 1 is the saxpy operation (Loop4) where column k of L is multiplied by the coefficient $L_{j,k}$ and subtracted from column j of K. If the coefficients of K are stored by columns, then the array accesses required in Loop4 are all stride one, minimizing memory bank conflicts.

Two features of the Cray-2 architecture limit the speed of the saxpy operation. First, since only one path exists between main memory and the vector registers, significant delays occur when loading and storing operands to and from the registers. Second, the Cray-2 does not provide "chaining" of vector instructions. Chaining would allow the memory accesses, multiplications and subtractions in the saxpy operation to be done almost concurrently, thereby more than doubling the effective computation speed.

Loop1 $k = 1$ to $n - 1$

$L_{k,k} = K_{k,k}^{1/2}$

Loop2 $s = k + 1$ to $\min(k + m, n)$

$L_{s,k} = K_{s,k} / L_{k,k}$

EndLoop2

Loop3 $j = k + 1$ to $\min(k + m, n)$

Loop4 $i = j$ to $\min(k + m, n)$

$K_{i,j} = K_{i,j} - L_{i,k} * L_{j,k}$

EndLoop4

EndLoop3

EndLoop1

$L_{n,n} = K_{n,n}^{1/2}$

Figure 1. *kji* **Choleski Factorization for** $n \times n$ **Matrix, Semi-Bandwidth=** m

Loop Unrolling. The algorithm shown in figure 1 was significantly improved by utilizing the technique known as *loop unrolling*. Loop unrolling can minimize the number of time-consuming memory references by holding vectors longer in the fast registers. In addition, loop unrolling adds additional vector computations within a loop. As a result, many of the multiplication and subtraction operations and memory references will overlap, leading to greater speed. A modified kji method using loop unrolling is illustrated in the algorithm shown in figure 2.

Loop1 $k = 1$ to $n - r - 1$ in steps of r

Compute r columns of L updating
appropriate columns of K

Loop2 $j = k + r$ to $\min(k + m, n)$

Loop3 $i = j$ to $\min(k + m, n)$

$K_{i,j} = K_{i,j} - L_{i,k} * L_{j,k} - L_{i,k+1} * L_{j,k+1} -$
$\ldots - L_{i,k+r-1} * L_{j,k+r-1}$

EndLoop3

EndLoop2

EndLoop1

Finish any remaining columns of L

Figure 2. Modified *kji* **Choleski Factorization for** $n \times n$ **Matrix, Semi-Bandwidth =** m**, Using Loop Unrolling Level =** r

In the modified algorithm, column j of K is updated using r columns of L in Loop3. The r columns are computed using a jki, or delayed update, algorithm. In a delayed update factorization algorithm, the columns (rows) of L are computed as before but the corresponding column (row) of K is updated just prior to computing that column (row) of L by using *previously* computed columns (rows) of L. The modified algorithm in figure 2 is a combination of kji and jki Choleski factorization. This change means that column j of K is loaded into the vector registers once for every r columns of L computed. In loop3, the loading of each of the r columns of L into the vector registers can be overlapped with the multiplication of the previous column of L by the appropriate scalar. Column j is stored in main memory only after all r columns of L have been used to update it. The modified algorithm, using $r = 4$, approximately doubled the computation rate of the factorization compared to the original kji algorithm.

Local Memory. A further increase in the computation rate for the Choleski factorization was realized by utilizing the fast local memory on the Cray-2. The 16,000 word local memory, accessible only through registers, can improve vector performance significantly for two reasons. First, the number of clock cycles required until the first word of a vector fetch instruction arrives in the vector register is 4 cycles compared with a minimum of 57 cycles from main memory. Secondly, memory bank conflicts are eliminated so that contiguous elements of vectors stored in local memory can be accessed each clock cycle. For the algorithm in figure 2, the r columns of L are copied into local memory before the updating of the columns of K. Because of the immediate updating characteristic of the algorithm, the r columns of L are accessed many times as the columns of K in Loop3 are updated. The faster memory accesses of these columns of L resulting from the use of local memory improved the computation rate by approximately 50 percent compared to the modified algorithm in figure 2. The combination of the jki and kji Choleski algorithms is based on modifications to the LINPACK[3] routine SGEFA for solving full general matrices using LU factorization on the Cray-2. The SGEFA routine was obtained from NAS personnel and also contains a machine language routine written by Cray Research personnel, Chao Wu Yang and Kuo Long Wu.

Variable Bandwidth. Finally, an additional decrease in factorization time is achieved by using variable bandwidth, or profile, storage of the columns of K and L rather than banded storage. The algorithm in figure 2 is modified to accomodate variable length columns of K. The time and memory savings resulting from using profile storage depends on the matrix structure. The savings for some test problems was nearly a factor of two for memory and more than a factor of two for execution time compared to the corresponding banded storage algorithms. The combined effect of loop unrolling, use of local memory, and profile storage of L and K reduced the computation time for the factorization by a factor of nearly 6 for some test problems, compared with the basic algorithm shown in Figure 1.

Triangular Solves. The solution of the triangular systems,

$$Lz = f \tag{2a}$$

$$L^T u = z \tag{2b}$$

is a forward solution, (2a), followed by a backward solution, (2b). As in the factorization of K, there are several possible implementations of these processes. Since the factorization is much more costly than the triangular solves, the data structure used for the decomposition often determines the forward and backward solution process. The lower triangular matrix is stored by columns for both algorithms in figures 1 and 2, so the forward solution is carried out using saxpy operations with stride one vector accesses. Figure 3 shows the *column sweep* algorithm used for the forward solution.

Loop1 $k = 1$ to n
  $z_k = f_k / L_{k,k}$
  Loop2 $i = 1$ to $\min(m, n-k)$
    $f_{k+i} = f_{k+i} - l_{k+i,k} * z_k$
  EndLoop2
EndLoop1

Figure 3. Column Sweep Forward Solution, $Lz = f$, Semi-Bandwidth=m

For each column, k, the unknown z_k is computed and then the linear combination of scalar z_k multiplied by column k of L is subtracted from f. The key computation is the vector saxpy in Loop2 which uses stride 1 memory accesses. For the backward solution, $L^T u = z$, the columns of L are now the rows of L^T and so a different approach which uses inner products is used to insure stride one memory accesses. Figure 4 illustrates this inner product algorithm. For each row, the inner product in Loop2 is computed and used to update the vector z. For very large bandwidth problems, this loop can be replaced with a call to the Cray function SDOT for a small increase in the computation rate.

Loop1 $k = n$ to 1
  Loop2 $i = 1$ to $\min(n-k, m)$
    $z_k = z_k - L_{k+i,k} * u_{k+i}$
  Endloop2
  $u_k = z_k / L_{k,k}$
Endloop1

Figure 4. Backward Solution, $L^T u = z$, Semi-Bandwidth=m

The loop unrolling techniques used for the factorization can also be applied to the forward and backward solutions but the increase in computation rate results in only a small decrease in total solution time since the factorization time dominates. Local memory can also be utilized in the solution process for smaller problems where the right hand side, z, can be stored in the available local memory. Since the amount of local memory available is usually less than 16,000 words, for very large problems local memory is not used for the triangular

solutions. The forward and backward solution times required only 1 to 3 percent of the time required for factorization of K.

Iterative Methods: Preconditioned Conjugate Gradient

One class of iterative methods can be described by the general form

$$u^{k+1} = Hu^k + d, \quad k = 0, 1, \ldots \tag{3}$$

where the matrix H is related to the original matrix K either by a splitting of K or perhaps by an approximate factorization of K into the product of upper and lower triangular matrices. The vector d is calculated from the original right hand side, f. Some of the well known iterative methods of this type include the Jacobi method, SOR (Successive Overrelaxation) and SSOR (symmetric SOR). The convergence of such a method to the desired solution depends upon the properties of H which in turn are influenced by the original matrix K.

A second class of iterative methods includes the so-called minimization methods. For the linear system (1), consider the associated quadratic function

$$Q(u) = 1/2u^T Ku - u^T f \tag{4}$$

For positive definite matrices, K, the minimizer of Q is the solution of (1). Many iterative methods which minimize Q are of the general form

$$u^{k+1} = u^k - \alpha_k p^k \tag{5}$$

where the vectors, p^k, are *direction* vectors and the scalar quantities, α_k, determine the distance along p^k in which to move in updating u^k. Various methods for choosing the p^k define different iterative methods. See Ortega[2] for a discussion of both classes of iterative methods on vector and parallel computers.

Implementation Considerations. The iterative method considered in this report, the preconditioned conjugate gradient method, is of the second type and is shown in figure 5. The notation (r, q) denotes the usual inner product of two vectors r and q. Most of the computations performed in figure 5 are easily vectorized on the Cray-2. Each iteration requires 2 inner products, (5a) and (5e) of figure 5, three vector updates (saxpy), (5b),(5c) and (5f) of figure 5, a matrix-vector multiplication (5a) of figure 5, and the preconditioning step (5d) of figure 5. The inner products and vector updates vectorize automatically using the Cray CFT77 compiler, and for large problems the Cray functions SDOT and SAXPY can be used for a small improvement in the computation rate. For large problems, the matrix-vector multiplication and the preconditioning step dominate the computations performed by each iteration.

Matrix-Vector Multiplication. The stiffness matrices for very large structural analysis applications are generally very sparse, and so the matrix-vector multiplication algorithm should be designed to perform well for sparse storage schemes. Two sparse storage schemes

Choose u^0; Set $r^0 = f - Ku^0$
Solve $Mq^0 = r^0$; Set $p^0 = q^0$
Loop $k = 0, 1, ...$

5a) $\alpha_k = -(r^k, q^k)/(p^k, Kp^k)$

5b) $x^{k+1} = x^k - \alpha_k p^k$

5c) $r^{k+1} = r^k + \alpha_k K p^k$

Test for convergence

5d) Solve $Mq^{k+1} = r^{k+1}$

5e) $\beta_k = (r^{k+1}, q^{k+1})/(r^k, q^k)$

5f) $p^{k+1} = q^{k+1} + \beta_k p^k$

Figure 5. Preconditioned Conjugate Gradient Algorithm

are considered for the conjugate gradient methods. The first scheme stores the coefficients of K by diagonals in order to increase the vector lengths in the matrix-vector multiplication. The length of the diagonals storing non-zero coefficients of K depends upon the ordering of the equations in K and on the amount, if any, of extra zeros allowed between succesive non-zeros in each diagonal. The second scheme uses sparse storage of the lower triangular part of K by columns, the same storage used for the preconditioning matrix. In either case, since K is symmetric, only the lower triangular part is stored.

The algorithm used for the matrix-vector multiplication for a symmetric n × n matrix stored by diagonals is shown in figure 6. Matrix multiplication by diagonals is described in Madsen[4] and Poole[5]. Each diagonal below the main diagonal, stored in A, beginning at position *ist*, is used twice in Loop3 with the row and column indices reversed for p and d. The FORTRAN code for Loop3 will not automatically vectorize since a potential vector dependency exists. A vector dependency occurs in Loop3 whenever the sub-vector of d beginning at *colm* overlaps with the sub-vector of d beginning at *row*. If the two statements in Loop3 are separated into two loops to calculate d_{row+i} and d_{colm+i}, the dependency is removed and both loops are vectorized automatically. The disadvantage of this approach is that the column of A used in both loops must then be loaded into the vector registers twice, reducing the computation speed of the algorithm.

By exploiting the single path to memory on the Cray-2, the dependency in Loop3 can be removed by ensuring that the updated sub-vector beginning at d_{row} is stored before the sub-vector beginning at d_{colm} is loaded from main memory. Unfortunately, the FORTRAN programmer does not have this level of control over the manner in which the compiler generates assembly code instructions. However, the CFT77 compiler does generate assembly code which will produce correct results for the FORTRAN code for Loop3 even when vector dependencies exist. The order of vector instructions generated by the CFT and CFT77 compilers for Loop3 with the compiler directive *ivdep* added to force vectorization is shown in figure 6. Only the CFT77 compiler produces correct results when vector dependencies exist in Loop3. The key difference in the two compilers is that the CFT generated code

Loop1 $i = 1$ to n
$d_i = a_i * p_i$
Endloop1

Loop2 $k = 1$ to $ndiags$
$ist = istart(k)$
$row = irows(k)$
$colm = icolms(k)$
$len = ilens(k)$

Loop3 $i = 0$ to $clen(k) - 1$
$d_{row+i} = d_{row+i} + A_{ist+i} * p_{colm+i}$
$d_{colm+i} = d_{colm+i} + A_{ist+i} * p_{row+i}$
Endloop3

Endloop2

	CFT Compiler		CFT77 Compiler	
1)	LOAD	p_{colm+i}	LOAD	p_{colm+i}
2)	LOAD	a_{ist+i}	LOAD	a_{ist+i}
3)	LOAD	d_{row+i}	LOAD	d_{row+i}
4)	LOAD	p_{row+i}	LOAD	p_{row+i}
5)	MULT	$A_{ist+i} * p_{colm+i}$	MULT	$A_{ist+i} * p_{colm+i}$
6)	LOAD	d_{colm+i}	ADD	$d_{row+i} + (A_{ist+i} * p_{colm+i})$
7)	ADD	$d_{row+i} + (A_{ist+i} * p_{colm+i})$	STORE	d_{row+i}
8)	MULT	$A_{ist+i} * p_{row+i}$	LOAD	d_{colm+i}
9)	STORE	d_{row+i}	MULT	$A_{ist+i} * p_{row+i}$
10)	ADD	$d_{colm+i} + (A_{ist+i} * p_{row+i})$	ADD	$d_{colm+i} + (A_{ist+i} * p_{row+i})$
11)	STORE	d_{colm+i}	STORE	d_{colm+i}

Figure 6. Matrix - Vector Multiplication $Kp = d$ for Diagonal Storage of K

loads d_{colm+i} (step 6) before it stores d_{row+i} (step 9) while the CFT77 compiler stores d_{row+i} in step 7 and loads d_{colm+i} in step 8. A substantial improvement in computation speed can be realized in this case by using the compiler directive *ivdep* and the CFT77 compiler.

The second algorithm used for the matrix-vector multiplication for an n × n matrix storing only the lower triangular non-zero coefficients of K by columns, is shown in figure 7. Array A stores the non-zero coefficients of K, and integer arrays *indx*, *cptr*, and *clen* store the row indices for each coefficient, the starting position in A for each column of K, and the number of non-zero coefficients in each column, respectively. In both the saxpy operation, updating

Loop1 $k = 1$ to n
$ist = cptr(k)$

Loop2 $i = 0$ to $clen(k) - 1$
$row = indx(ist + i)$
$d_{row} = d_{row} + A_{ist+i} * p_k$
$d_k = d_k + A_{ist+i} * p_{row}$
Endloop2

Endloop1

Figure 7. Matrix - Vector Multiplication $Kp = d$

d_{row}, and the inner product, updating d_k, non-contiguous elements of d are accessed using the index array $indx$. This operation requires vector gather and scatter operations on the Cray-2 which can add significantly higher overhead due to memory bank conflicts. The saxpy and inner product operations can be carried out using calls to Cray functions $SPAXPY$ and $SPDOT$. However, the coefficients in each column of K, stored in array A beginning at A_{ist}, must then be loaded once for each call. To eliminate the unnecessary extra load of each column of K, the functions are not used and the FORTRAN code for Loop2 in figure 7 is vectorized automatically by the CFT77 compiler. The loop unrolling techniques described above for the banded factorization cannot be used for this general sparse matrix - vector multiplication scheme. The vector performance of this algorithm is limited by both the indirect addressing and by the short vector lengths determined by the average number of non-zero coefficients in each row.

Preconditioning. The preconditioning step, (5d), in figure 5, enhances the convergence rate of the basic conjugate gradient method at the expense of additional work performed at each iteration. M is a symmetric positive definite matrix chosen to approximate K in some sense. If the preconditioning matrix M is chosen to be the identity matrix then the method shown in figure 5 is just the basic conjugate gradient method. If M is chosen to be K then the method converges in one step exactly. The trade-off in selecting an appropriate M is to choose a good approximation to K that improves convergence while minimizing the overhead of solving the system (5d). A simple choice for M is to choose M to be the main diagonal of K. This choice is sometimes referred to as Jacobi preconditioning and is denoted here as the JCG method. This preconditioning is most often implemented by symmetrically scaling K so that the main diagonal entries are all 1.0. M is then the identity matrix and the basic conjugate gradient method is used for the new system

$$\hat{K}\hat{u} = \hat{f} \tag{6}$$

where $\hat{K} = D^{-1/2}KD^{-1/2}$, $\hat{u} = D^{1/2}u$, $\hat{f} = D^{-1/2}f$ and D is a diagonal matrix with diagonal entries equal to the diagonal entries of K. This form of preconditioning adds very little overhead to the basic conjugate gradient algorithm without preconditioning since step (5d) is not required at each iteration. The matrix-vector multiplication by diagonals algorithm in figure 6 is used to obtain longer vector lengths and to eliminate the

need for indirect addressing required by the sparse column storage scheme used in figure 7, improving the computation speed on the Cray-2. The major drawback of the JCG method is that the convergence rate compared to the basic conjugate gradient method is not improved as much as for some other preconditioning schemes. For some example structures problems this simple preconditioning strategy is not sufficient for convergence even when the number of iterations is equal to the number of equations.

Incomplete Choleski Factorization. A much studied choice for M has been to use incomplete Choleski factorization of K where $K = LDL^T + R$. (See for example, Meijerink and van der Vorst[6,7] Poole and Ortega[8].) The matrix R is never actually calculated but represents the error made by performing incomplete factorization of K. The incomplete Choleski conjugate gradient method (ICCG) described in this report chooses L to be a unit lower triangular matrix with the same non-zero structure as the corresponding part of K. D is a diagonal matrix used for this form of Choleski decomposition, avoiding the calculation of square roots in the decomposition process. This type of preconditioning requires the additional work of both a forward and backward solution of sparse triangular systems for each conjugate gradient iteration.

Loop1 $i = 1$ to n
  $D_i = (1 + \gamma) * K_{i,i}$
Endloop1

Loop2 $k = 1$ to $n - 1$
  Loop 3 $s = 1$ to $clen(k)$
    $L_{s,k} = K_{s,k}/D_k$
  Endloop3

  Expand column k of L

  Loop4 $j = k + 1$ to last row in column k
    $D_j = D_j - L_{j,k} * D_k * L_{j,k}$
    is $D_j <= 0$?; if so increase γ and start over
    Loop5 $i = 1$ to $clen(j)$
      $row = indx(i)$
      $K_{i,j} = K_{i,j} - L_{row,k} * D_k * L_{j,k}$
    Endloop5
  Endloop4

Endloop2

Figure 8. kji **Sparse Incomplete Choleski Factorization**

Figure 8 shows the sparse kji algorithm used for the incomplete factorization of K. The procedure used is similar to the Choleski decomposition described in figure 1. Sparse storage is used for both K and L with an index array to store the row index for each coefficient. The columns of L are computed starting with column 1 and proceeding through

column n in n stages. At the k^{th} stage, column k of L is formed by a vector divide and expanded so that the entire column out to the last non-zero coefficient is stored temporarily while it is used to update columns of K beginning with column $k+1$ in Loop4. The key computational step in the incomplete factorization process is the vector updating of columns of K by subtracting the product of the scalars D_k and $L_{j,k}$ and the elements of vector $L_{row,k}$ from column j of K. The computations in Loop5 of figure 8 are performed corresponding only to non-zero coefficients in column j of K while in Loop4 of figure 1 all of the elements produced by the linear combination of column k of L and the scalar $L_{j,k}$ are subtracted from column j of K. This incomplete updating of the columns of K ensures the same non-zero structure for L but requires that elements in column k of L are accessed indirectly using the index array $indx$. Since K must be saved for the matrix-vector multiplication in the conjugate gradient algorithm, the columns of K are not actually modified but rather are initially copied into L and modified appropriately at each stage.

The parameter γ is used to enhance convergence and insure stability of the decomposition. Instability occurs in the incomplete decomposition when a negative element of D is computed causing loss of positive definiteness of the preconditioning matrix. If a negative diagonal element is computed during the factorization, γ is increased, D is computed again, and the factorization is repeated. The values chosen for γ are determined experimentally but in practice small positive values for γ ($0 \leq \gamma < .15$) work best. This modification to the incomplete decomposition is based on the "shifting method" of Manteuffel[9]. The vector speed of the incomplete decomposition is limited by the short vector lengths (number of non-zeros in each column of K) and by the indirect addressing used for the decomposition. The sparse storage scheme and indirect addressing requirements make the use of the loop unrolling techniques and local memory impossible for the incomplete decomposition.

Sparse Triangular Solves. The majority of the computational work for preconditioning consists of forward and backward solutions at each iteration using the sparse matrix L. The algorithm used for both solution steps is the same as previously described for the direct Choleski method. However, the elements of vectors u and z in figures 3 and 4 must be accessed using index arrays due to the sparse storage scheme used for L. As before, indirect addressing and shorter vector lengths limit the maximum attainable vector speed. Both the incomplete factorization and the triangular solution steps required for preconditioning use column storage of the sparse matrix, L. The matrix-vector multiplication algorithm shown in figure 7 is used with incomplete Choleski preconditioning since sparse column storage of K and L is used.

Comparison of Direct and Iterative Methods

Several factors affect both the choice of direct or iterative methods and the relative performance of each method. Memory requirements vary greatly for direct and iterative methods for many problems. A computationally slower method that can solve a given problem in main memory may solve the linear system faster than a computationally faster method that requires more memory than is available in main memory. In addition, a method which is computationally slower on a vector computer such as the Cray-2 but requires significantly less work may still have the fastest execution time for a given problem. This

section contrasts memory requirements and performance factors for the Choleski direct solvers and the preconditioned conjugate gradient solvers on the NAS Cray-2.

Memory Requirements. The memory requirements for direct solvers are determined by the storage of the factored matrix L. Though the original matrix K is usually sparse in finite element applications, the factored matrix contains many new "fill" elements within a band (or profile). The number of coefficients stored for a symmetric matrix with n equations and semi-bandwidth m using banded storage is nm. For profile storage, the number of coefficients stored depends upon the average semi-bandwidth, $\overline{m}$, since each row or column is stored only out to the last non-zero coefficient. The bandwidth of the matrix is determined by node connectivity and by the ordering of equations in the linear systems. Several algorithms exist which generate orderings which minimize bandwidth and/or profile for a wide class of problems. (See, for example, George and Liu[10] and Everstine[11].) For many problems solved using the finite element method the bandwidth grows as the problem size increases. For very large problems the memory requirements for banded solvers may exceed the main memory of the computer and methods which use sparse matrix storage schemes may be necessary.

The memory requirements for iterative methods are generally much less than for direct methods since there is no factorization which produces new non-zero terms. Both the input matrix, K, and the preconditioning matrix, L, can be stored using sparse storage schemes. For the conjugate gradient method preconditioned by diagonal scaling of the input matrix, the key computational step is the matrix-vector multiplication required at each iteration. In this case diagonal storage of K together with matrix multiplication by diagonals described in the previous section is used to increase vector lengths and computation speed. For diagonal storage the memory requirements depend not only upon the number of non-zero coefficients but also upon the number of zeros allowed between successive non-zero coefficients in each diagonal. The addition of zeros between successive non-zero coefficients in a diagonal is controlled by a parameter, $max0s$. Increasing $max0s$ increases the vector length of the diagonals but requires more storage and introduces unnecessary computations into the matrix-vector multiplication. The tradeoff between increased computation rate from using longer vectors and the additional time for unnecessary computations determines the optimum value for $max0s$. Typical memory requirements for the total number of coefficients stored using diagonal storage for some test problems using a value of $max0s$ that minimizes runtime are between $1.5C$ and $2C$ where C is the number of non-zero coefficients in the upper triangular part of K.

Incomplete Choleski preconditioning for the conjugate gradient method requires the additional solution of triangular systems, equations (2a) and (2b), at each iteration. L has the same non-zero structure as the lower triangular part of K as described previously. The non-zero coefficients of L are stored by columns with integer row indices stored for each coefficient as well. The storage requirement for L is $2C$. If K is stored using this same sparse storage scheme then the memory requirements for the matrices stored for ICCG is $3C$ since the same index array can be used for the coefficients of both matrices.

Total memory requirements for both direct and iterative methods include matrix storage as well as pointer arrays and other miscellaneous vectors used by the algorithms for the

computations. Memory requirements, measured by the number of single precision words, for the banded and profile direct Choleski solvers as well as three preconditioned conjugate gradient methods are given in table 1. The three preconditioned conjugate gradient methods are: 1) JCG, diagonally scaling using diagonal storage for K, 2) ICCG1, incomplete Choleski with sparse column storage of K and L, and 3) ICCG2, incomplete Choleski preconditioning with diagonal storage of K and sparse column storage of L.

Performance Factors. The effective use of computers like the Cray-2, which have special features that have significant effects on the speed at which computations are performed, necessitates consideration of often conflicting factors in choosing which method to use for a given task. Often an algorithm that requires more computations may actually be significantly faster than another requiring fewer computations if the first algorithm better exploits the vector capability of the Cray-2. In this section two main factors affecting the relative performance of direct and iterative methods are discussed. These two factors are the amount of computations required by each method and the rate at which the computations can be done. Formulas are given for the number of floating point operations for each method, allowing one to predict the relative performance of each method based on a few key parameters such as problem size, convergence rate and computation rate.

Table 2 gives formulas for the number of floating point arithmetic operations required by each method in terms of the problem size parameters given and the number of iterations, *iter*, for the iterative methods. For the direct band and profile Choleski methods, the number of computations is proportional to nm^2 and $n\overline{m}^2$, respectively. For the iterative conjugate gradient methods, the number of computations is proportional to the number of iterations required for convergence and C, the number of non-zero coefficients in K and M. As n, m, and $\overline{m}$ increase for very large structural analysis problems the number of non-zero coefficients in each row of K generally remains constant. The resulting effect on the amount of computation is that for very large problems banded equations solvers may require significantly more computation than iterative methods. Another factor which affects bandwidth for structural analysis problems is the type of elements used in the finite element model. Higher order elements generally lead to more coefficients and larger bandwidth matrices for a given size problem.

The work per iteration for diagonally scaled conjugate gradient (JCG) is much less than for incomplete Choleski preconditioning (ICCG), but is offset by an increase in the number of iterations required for convergence compared to ICCG. The diagonal storage scheme used for JCG increases the amount of computations required for the matrix-vector multiplication when extra zeros are stored between successive non-zero coefficients along diagonals of K in order to obtain longer vector lengths. There is a trade off between increased computation rate and additional operations performed because of the added zeros.

The rate of computation for each method is the second major factor affecting the relative performance of these methods. The direct Choleski methods achieve the highest computation rates of the methods studied in this report. The combined effect of longer vector lengths (a function of m), loop unrolling, and use of fast local memory yield computation rates as high as 200 Mflops (million floating point operations per second) for medium bandwidth problems. The diagonally scaled preconditioned conjugate gradient method

Table 1. Memory Requirements for Direct and Iterative Methods

Method	Matrix Storage	Total Memory
Choleski Banded	nm	$nm + 5n$
Choleski Profile	$n\overline{m}$	$n\overline{m} + 7n$
JCG	$amax^* + 4d$	$amax^* + 4d + 8n$
ICCG1	$3nc + 2n$	$3nc + 11n$
ICCG2	$3nc + 2n$	$3nc + 11n$

* $C \leq amax \leq nm$

Table 2. Performance Factors for Direct and Iterative Methods

Method	Work (+/×) Operations	Estimated Rate Rate (Mflops)
Choleski Banded	$nm^2 + 6nm - \frac{2}{3}m^3$ $\frac{7}{2}m^2 - \frac{17}{6}m$	$105 - 200$
Choleski Profile	$O(n\overline{m}^2)$	$70 - 187$
JCG	$4D + 2n + iter \times (4D + 10n)$	$21 - 82$
ICCG1	$4C - n + iter \times (8C + 11n)$	$9 - 15$
ICCG2	$4C - n + iter \times (4C + 4D + 11n)$	$11 - 25$

JCG - Jacobi Preconditioned Conjugate Gradient (diagonal scaling) using diagonal storage of K

ICCG1 - Incomplete Choleski Preconditioned Conjugate Gradient using sparse column storage of K and L

ICCG2 - Incomplete Choleski Preconditioned Conjugate Gradient using diagonal storage for K, sparse column storage for L

C - number of non-zero coefficients in upper triangular part of K

D - number coefficients stored for upper triangular part of K using diagonal storage

d - number of diagonals used to store K

n - number of equations, dimension of K

m - semi bandwidth of K including main diagonal

$\overline{m}$ - average semi-bandwidth of K including main diagonal

using diagonal storage of K is not as fast as the Choleski method since the loop unrolling techniques do not apply to the matrix-vector multiplications required at each iteration. The slowest method was incomplete Choleski conjugate gradient which suffered from the use of indirect addressing and shorter vector lengths. This imbalance in the computation rates for the direct and iterative methods grants a substantial advantage to the direct Choleski banded solvers for many problems.

INTEGRATION OF METHODS INTO CSM TESTBED

An important concern in the use of supercomputers for structural analysis applications is the development of software which can be used for a great variety of applications while exploiting the architecture of the computer. While some of the optimization of code for the Cray-2 is automated through the use of the compilers, the greatest gains are still achieved only when extensive changes are made both to the methods used and the underlying data structures. It is often not feasible to rewrite a very large existing code to exploit a particular architecture and therefore methods which can be adapted to existing codes are very important. This section describes the integration of the direct and iterative solvers, described previously, into the CSM Testbed, a large, research oriented, finite element code developed at NASAs Langley Research Center. (See Gillian[12]).

Data Structures. The key detail in implementing the equation solvers designed for the Cray-2 architecture into the Testbed software was the data structures. The Testbed software system functions through a global database management system with individual *processors*, actually FORTRAN subroutines, called by a high level command language, CLAMP, (e.g., Felippa[13]) performing various tasks in a particular job and creating or reading output through the global database. The generation of stiffness matrices is accomplished by several different processors producing element stiffness matrices, defining boundary conditions, applying loads, ordering nodes, and assembling the stiffness matrices. The stiffness matrices are stored in a block sparse form. For each node, blocks dimensioned as a number of degrees-of-freedom $\times$ degrees-of-freedom contain non-zero coefficients associated with each node connected in the finite element discretization. The sparse out-of-core Choleski solver used by the Testbed code (processors INV and SSOL) factors and solves the stiffness matrices using this data structure. A major obstacle for this solver on the Cray-2 is that the operations carried out in factoring the stiffness matrix and solving the resulting triangular systems are carried out using the small $dof \times dof$ blocks. The vector length of these operations is typically 6 or less, and the code is faster when run without vector optimization. Another drawback in the current Testbed code is that the use of out-of-core solvers such as INV drastically increases the I/O overhead incurred in the solution process thereby increasing the wall clock time for the solution process substantially.

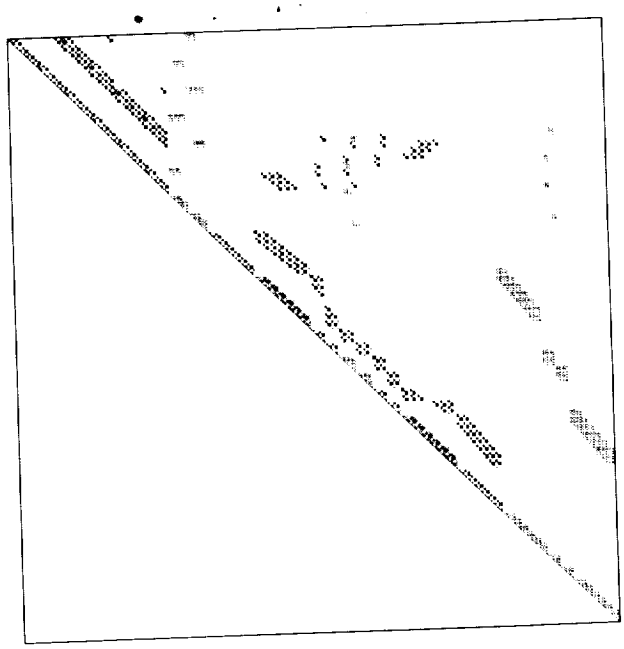
Matrix Reformating Procedure. The strategy used to reformat the Testbed stiffness matrices for the vectorized equation solvers is described next. The vectorized equation solvers require K to be stored in core using several different sparse and banded storage schemes. First, the coefficients of the unconstrained stiffness matrix are read from the global database into a temporary array. Second, the joint constraint information and

joint ordering sequence information is retrieved from the global database. Third, the appropriate pointer arrays for the new storage scheme are formed. Finally the coefficients of K are placed in a single dimensioned array and modifications are made to the right hand side, f, corresponding to any applied fixed displacements. For the direct Choleski methods an additional storage scheme was included to reformat Testbed stiffness matrices into the standard LINPACK banded storage. The reformating procedure is essentially sequential, but the time to reformat the matrices was small compared to the time to solve the equations for large problems. The success of this strategy means that efficient use of advanced computer architectures may be possible for large-scale applications codes originally written prior to the design of modern parallel and vector computers computers without completely rewriting the code.

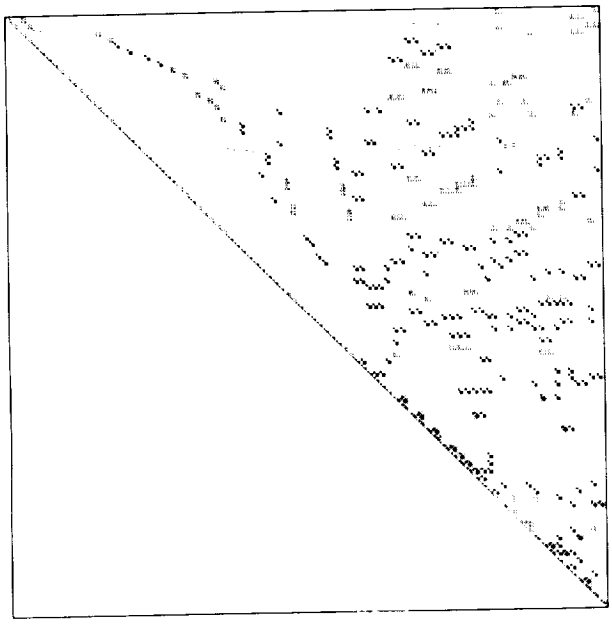
Node Reordering. An important part of equation solvers for general purpose finite element codes is the node reordering capability. The structure of the assembled stiffness matrices is determined by the node connectivities and node ordering scheme used in the finite element model. While the node connectivity is fixed by the problem definition and discretization, many possible node orderings are possible. The Testbed software contains a processor, RSEQ, which uses four different algorithms to automatically reorder nodes. These algorithms are nested dissection, minimum degree, reverse Cuthill-Mckee [10], and Gibbs-Poole-Stockmeyer[11].

The first two methods are used by sparse solvers and minimize fill in the factorization process. The last two are profile and bandwidth minimizing routines, respectively. The direct banded solvers are most efficient with the node orderings which minimize bandwidth while the sparse Choleski Testbed equation solver is most efficient with orderings which minimize fill. For the preconditioned conjugate gradient methods, the preconditioner used determines which ordering is best. Although the precise relationship between grid ordering and the convergence rate of ICCG is not known, experimental results show that the ordering of nodes can have a great effect on the convergence rate. In the test problems used with the ICCG method, the sparse, minimum fill orderings were better for the convergence rate of ICCG than the bandwidth minimizing orderings. However, in some cases, the ordering used to define the problem gave the best convergence rate. For the JCG method, the matrix structure has no effect on the convergence rate but the matrix structure is important for the storage requirements if diagonal storage is used. Orderings which minimize bandwidth also concentrate the coefficients near the main diagonal thereby minimizing the number of diagonals required for matrix storage by diagonals. As a result the vector lengths of the diagonals are longer, the number of extra zeros added between non-zero coefficients is less, reducing the memory requirements, and the computation speed is increased.

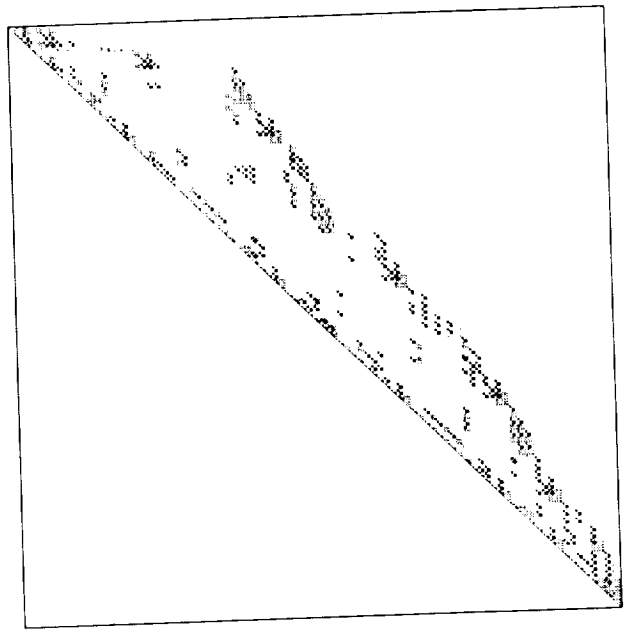
An example of the effect of node ordering on the non-zero structure of K is shown in figure 9 for a blade-stiffened panel with 648 degrees of freedom. The non-zero structure of the upper triangular matrix shown in figure 9a results from the node ordering used to define the finite element model. Figures 9b and 9c show the change in the matrix structure from using two different node reordering algorithms.



(a) No Node Reordering



(b) Minimum Degree Node Reordering

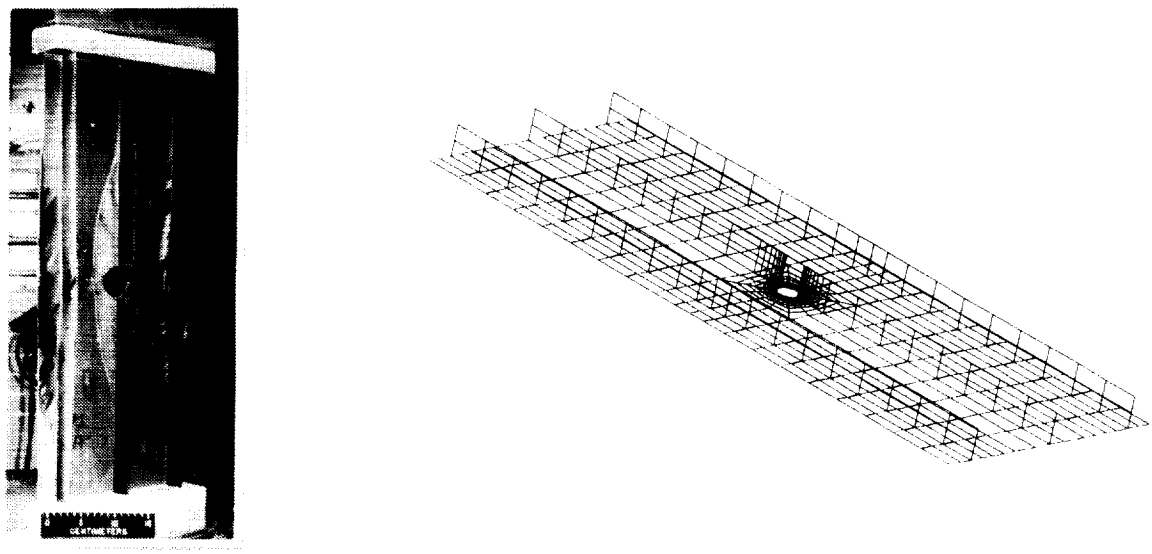


(c) Reverse Cuthill McKee Node Reordering

Figure 9. Pattern of Non-Zero Coefficients in the Upper Triangular Part of Stiffness Matrices for 648 D.O.F Stiffened Panel Problem Using Three Different Node Orderings

RESULTS

In this section, results are presented to demonstrate the performance of the direct and iterative equation solvers on three example problems. These problems represent a wide range of structural analysis applications. The three problems are a blade-stiffened panel, a cube-shaped solid, and the Space Shuttle Solid Rocket Booster. Several implementations of the new direct Choleski equation solvers described in this report are compared in detail for the blade-stiffened panel problem. The equation solution time is given for each implementation along with the computation rate, the amount of work, and the overhead time required to reformat the stiffness matrices. A similar detailed comparison of three preconditioned conjugate gradient iterative methods is given for the same blade-stiffened panel problem. Finally, the fastest equation solution times for both the new direct Choleski methods and the new iterative preconditioned conjugate gradient methods are compared with the initial Testbed sparse, node-oriented Choleski equation solver for all three structural analysis problems. The results show that the relative performance of the equation solvers varies for the different problems. Several important factors are given which influence the choice of equation solvers for these types of problems.



(a) Stiffened Panel

(b) Finite Element Model

Figure 10. Composite Blade-Stiffened Panel with Discontinuous Stiffener

Description of Example Problems

Three example problems are considered using the CSM Testbed for model generation and problem solution. The first problem is a blade-stiffened graphite-epoxy panel with

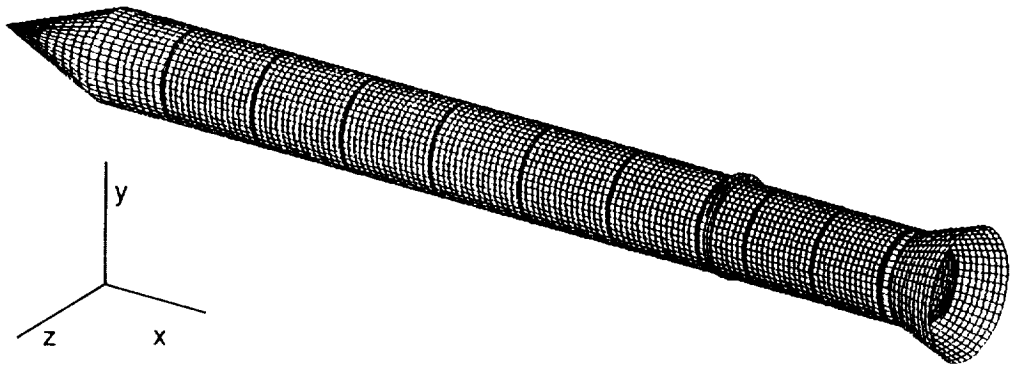
a discontinuous stiffener and a central cutout hole. The panel is shown in figure 10a. The panel stiffeners lie perpendicular to the panel, introducing three dimensions to the finite element model, but the problem is essentially a two dimensional problem. The panel problem was selected as a CSM focus problem because it has characteristics which often require a global/local analysis. These characteristics include a discontinuity (the hole), eccentric loading, large displacements, large stress gradients, and a brittle material system. This problem represents a generic class of laminated composite structures with discontinuities in which the interlaminar stress state becomes important. The geometry and laminate properties are given in reference 14. The loading is uniform axial compression with the loaded ends clamped and the sides free. The 3768 degree-of-freedom finite element model shown in figure 10b contains 576 4-node quadrilateral elements. Another 3768 degree-of-freedom model containing 144 9-node quadrilateral elements was also considered. The input is parameterized so that mesh sizes can be easily changed. The number of non-zero coefficients per row and the bandwidth are greater for the stiffness matrix arising from the 9-noded elements compared to the 4-node elements.

The second problem is a cube-shaped, isotropic solid constrained at the corner nodes on one face and loaded with uniform pressure along the face opposite the constrained nodes. This problem is representative of the detailed three dimensional model required in a local stress analysis around the hole in the blade stiffened panel problem. The finite element model used for the cube problem is composed of 729 8-noded solid elements with 3000 degrees of freedom and contains equal numbers of elements along each axis.

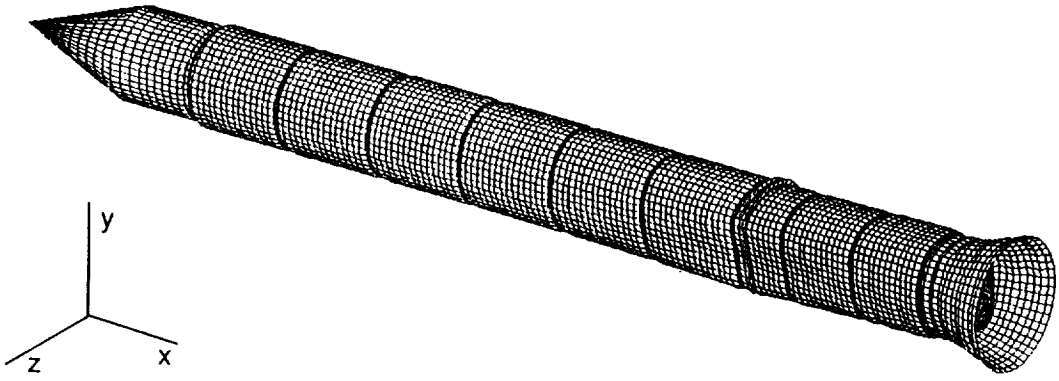
The third problem is a linear elastic static analysis of the Space Shuttle Solid Rocket Booster (SRB) loaded by uniform internal pressure. This problem is representative of large scale structural analysis problems and demonstrates the reduction in analysis time possible using the new equation solvers. The three-dimensional finite element model shown in figure 11a is composed of 9205 nodes and includes 9156 4-noded quadrilateral shell elements, 1273 2-node beam elements and 90 3-node triangular elements. The resulting stiffness matrix is 54,870 $\times$ 54,870 . An oblique view of the deformed geometry with exaggerated deflections resulting from a uniform internal pressure of 1000 psi is shown in figure 11b. A detailed description and analysis of this problem is given in reference 14.

Performance of Direct Choleski Solvers

The execution times, computation rates, amount of work and overhead time required to reformat matrices is given in table 3 for the vectorized Choleski solvers described in the previous section. The LINPACK times, included as a reference, were obtained using the LINPACK routines SPBFA and SPBSL available in the Cray-2 library, SCILIB. The two different finite element models of the 3768 D.O.F stiffened panel problem compared in table 3 demonstrate the effect of bandwidth on the performance of the direct solvers. For each problem, the nodes were ordered using the reverse Cuthill-McKee[10] algorithm to minimize bandwidth. The times given include the cpu time required by each method for both the factorization of K and the required forward and backward triangular solves (equations 2a and 2b). The computation rates given are computed using the total number of vector additions and multiplications required by each method and the times for each method. The overhead given for each method is the time in seconds required to reformat



(a) Finite Element Model of SRB



(b) Deformed Geometry Plot of SRB Shell Model Loaded by Internal Pressure

Figure 11. Space Shuttle Solid Rocket Booster

the stiffness matrix from the Testbed sparse matrix storage into the required banded storage. Higher computation rates are possible using dedicated user mode on the Cray-2, and a substantial increase in the computation rate may also be possible if the algorithms are written in assembly code for the Cray-2. However, the results show that substantial improvements are possible without the loss of portability of the code.

The basic *kji* method is slightly faster than the LINPACK routines for the smaller bandwidth panel problem but is essentially the same as LINPACK for larger bandwidth problems. The LINPACK routines use a form of Choleski factorization which computes inner products in the innermost loop. For larger bandwidth problems the inner products are nearly as fast as the saxpy operations used by the *kji* method. The positive effect of loop

Table 3. Comparison of Banded and Profile Choleski Methods for a 3768 D.O.F. Stiffened Panel Problem

Using 576 4-Noded Elements, 2910 Equations
Semi-bandwidth=425, Average Semi-bandwidth=230

Method	Time (sec)	Rate (mflops)	Work (-/x)	Overhead (reformat)
LINPACK	10.2	47	479,104,101	.43
kji Banded	9.0	53	479,104,101	.43
kji Banded*	5.2	93	479,104,101	.43
kji Banded**	3.7	130	479,104,101	.43
kji Profile	4.4	45	196,321,325	.44
kji Profile*	2.3	87	198,255,170	.45
kji Profile**	1.7	119	198,255,170	.45

Using 144 9-Noded Elements, 2910 Equations
Semi-bandwidth=860, Average Semi-bandwidth=437

Method	Time (sec)	Rate (mflops)	Work (-/x)	Overhead (reformat)
LINPACK	27.0	64	1,737,086,982	.69
kji Banded	27.4	63	1,737,086,982	.69
kji Banded*	17.7	98	1,737,086,982	.69
kji Banded**	12.7	137	1,737,086,982	.71
kji Profile	12.7	57	724,292,145	.74
kji Profile*	7.9	87	729,507,322	.74
kji Profile**	5.7	128	729,507,322	.74

* Loop unrolling to level 4.

**Loop unrolling to level 4 and use of local memory.

unrolling and use of local memory by the modified *kji* methods is demonstrated by the decrease in execution time and increase in computation rate for both panel problems. The effect of bandwidth on execution time is shown by comparison of execution time and work for both panel problems. Doubling the bandwidth (425 to 860) results in a nearly 4-fold increase in the number of operations performed and an increased execution time. The longer vector lengths which result from the increased bandwidth improves the computation rate

for the second panel problem, but only slightly. The reduced times for the profile Choleski methods are also due to reduced operation counts. The average bandwidth given for each problem is a measure of the reduction of the bandwidth when using the profile Choleski method compared to the full bandwidth using the kji methods with banded storage only. The improvements made to the basic kji method resulted in an approximately 5-fold decrease in the execution time for both panel problems with maximum computation rates of over 130 Mflops.

Performance of Iterative Solvers

The performance of the three preconditioned conjugate gradient iterative methods described in this report is given in table 4 for the two panel problems. Results for three different node orderings are given, demonstrating the effect of node orderings on the convergence and execution time for the iterative methods. The three node orderings used are illustrated in figure 9 for a reduced-size panel problem. The first node ordering for each problem is the node ordering defined by the problem definition. For the panel problems the nodes around the central hole were numbered first, followed by the in-plane nodes, and then each group of stiffeners. The resulting non-zero matrix structure is shown in figure 9a for a 648 D.O.F blade-stiffened panel. The matrix in figure 9a does not have small bandwidth but has regular patterns within the regions numbered separately requiring fewer diagonals to store the matrix coefficients and resulting in longer vector lengths in the matrix-vector multiplications. The matrices shown in figure 9b and 9c are for the same 648 D.O.F panel using the minimum degree[10] and reverse Cuthill-McKee[10] orderings, respectively. The convergence of the JCG method is unaffected by the node ordering used as seen by comparing the number of iterations for JCG for each node ordering in table 4. However, a comparison of the work required for the three orderings for JCG in table 4 for both panel problems shows that the operation counts are much higher for both the minimum degree orderings and the reverse Cuthill-McKee orderings. The increased operation counts and slower computation rates result in slower execution times for JCG for both the minimum degree[10] and reverse Cuthill-McKee[10] orderings. For orderings which result in a more random distribution of non-zero coefficients, the number of diagonals used to store the coefficients increases as does the number of zeros added between successive non-zero coefficients along a given diagonal. In general, the bandwidth minimizing node orderings are good since they tend to concentrate the coefficients nearer the main diagonal, but the panel problems indicate that orderings which are regular within regions may also be good for minimizing diagonal storage requirements.

The incomplete Choleski preconditioned conjugate gradient methods, ICCG1 and ICCG2, are the same except that ICCG2 uses diagonal storage for the stiffness matrix to improve the computation rate. The larger overhead required by the JCG and ICCG2 methods in table 4 shows that the diagonal storage scheme requires a longer time for reformatting the matrices than the sparse storage scheme used for the preconditioning matrix L. A comparison of the convergence results and the amount of work for both panel problems shows that incomplete Choleski preconditioning does reduce the number of iterations required for convergence as well as the amount of work required but at the expense of much slower vector performance. The slower computation rates shown for the ICCG methods in table

Table 4. Comparison of Iterative Methods for 3768 D.O.F. Stiffened Panel Problem

Using 576 4-Noded Elements, 2910 Equations
50494 non-zero coefficients stored, $\gamma = .35$

Method	Convergence (iterations)	Time (sec)	Rate (mflops)	Work (+/x)	Overhead (reformat)
JCG	678	4.3	56	240,440,514	.56
ICCG1	231	9.4	10	97,521,702	.42
ICCG2	231	8.3	15	128,825,922	.62
JCG*	678	17.2	21	365,473,950	.55
ICCG1*	264	12.4	9	110,511,352	.41
ICCG2*	264	18.4	11	195,120,552	.61
JCG**	678	6.7	51	340,161,216	.55
ICCG1**	265	11.5	10	111,456,882	.42
ICCG2**	265	11.0	17	186,495,048	.61

Using 144 9-Noded Elements, 2910 Equations
88625 non-zero coefficients stored, $\gamma = .08$

Method	Convergence (iterations)	Time (sec)	Rate (mflops)	Work (+/x)	Overhead (reformat)
JCG	857	11.5	50	573,088,330	.95
ICCG1	224	12.0	14	161,115,062	.71
ICCG2	223	11.8	20	223,280,328	1.06
JCG*	857	29.3	26	765,933,912	.93
ICCG1*	158	10.3	11	118,240,182	.72
ICCG2*	158	15.3	13	201,472,198	1.02
JCG**	857	18.0	52	939,011,468	.95
ICCG1**	176	10.0	13	133,018,764	.71
ICCG2**	175	11.8	22	259,766,382	1.03

* Minimum Degree Node Ordering

** Reverse Cuthill-Mckee Node Ordering

JCG - Jacobi Preconditioned Conjugate Gradient (Diagonal scaling) using diagonal storage of K

ICCG1 - Incomplete Choleski Preconditioned Conjugate Gradient using sparse column storage of K and L

ICCG2 - Incomplete Choleski Preconditioned Conjugate Gradient using diagonal storage for K, sparse column storage for L

γ - Shifting parameter for incomplete factorization

4 are due to short vector lengths (less than 20 for the first panel problem) and indirect addressing required by the sparse triangular solves. The first node ordering was best for the first panel problem but the minimum degree ordering was best for the second problem indicating that the best node ordering varies for different problems. The convergence parameter used for both problems was experimentally obtained and was markedly different for each of the panel problems. For the runs given in table 4 the parameter γ was the same for all runs of each of the two problems, although small improvements in the number of iterations can also be obtained by varying the parameter for each ordering. The value of γ chosen for the panel problems is the largest γ required among the three orderings to insure positive diagonal elements in D. A general strategy for choosing γ for a given problem is to perform the incomplete factorization beginning with $\gamma = 0$ and repeating with an increased γ if negative diagonal terms are computed for the diagonal matrix D.

The execution times in table 4 show that JCG was the fastest method for the first panel problem but ICCG1 was fastest for the second problem. These mixed results for the iterative methods demonstrate the interaction between the computation rate and the amount of work for each method. In the first problem there are fewer non-zero coefficients in the stiffness matrix, reducing the amount of work per iteration. The ratio of work performed by JCG to the work performed by ICCG1 is nearly 2.5 to 1 and on a serial computer the ICCG1 method would be 2.5 times faster than the JCG method. However the JCG method runs 5.6 times faster than the ICCG1 method and so the combined effect of these two factors is that JCG is faster by a factor of 2 on the Cray-2. The ICCG2 method has a higher computation rate due to the longer vector lengths for the matrix-vector multiplications in each conjugate gradient iteration, but it also requires more work than the ICCG1 method, offsetting most of the gain in computation rate. For the second problem the ICCG1 method using the reverse Cuthill-McKee algorithm is slightly faster than the JCG method using the first node ordering. The minimum degree ordering converged fastest for this problem, but a lower computation rate resulted in a lower execution time than the reverse Cuthill-Mckee ordering. For both node orderings for ICCG1, the ratio of work for JCG using the fastest node ordering to work for ICCG1 is much higher (nearly 5 to 1) than for the first problem. This is largely because the second problem did not converge as fast for JCG as did the first (857 iterations compared to 678) while the ICCG methods converged faster for the second problem. In addition, computation rates are faster for the ICCG methods for the second problem due to longer vector lengths resulting from more coefficients in each row of the matrices. The greater reduction in the amount of work coupled with higher computation rates for the ICCG methods on the second panel problem improved the performance of the ICCG methods relative to JCG.

A general conclusion from the comparison of JCG and ICCG methods is that the incomplete Choleski preconditioning is not as effective for reducing execution time on vector computers like the Cray-2 as it has been on serial computers due to the severe penalty of reduced computation rate resulting from indirect addressing and short vector lengths. However, for problems where the preconditioning reduces the amount of work sufficiently, some improvement in execution time may be realized.

Comparison of Equation Solvers

Comparisons of four solvers - the fastest new direct Choleski solver, the JCG iterative solver, the fastest ICCG iterative solver, and the initial Testbed sparse Choleski equation solver (INV/SSOL) - are given in table 5 for all three example structures problems. The times for the sparse Choleski equation solver may not be representative of state-of-the-art sparse equation solvers but the times do indicate some key performance factors for sparse Choleski methods on the Cray-2. The memory requirements given in table 5 include the total number of 64-bit words required for the upper triangular matrices including the main diagonal and any required index arrays by each method.

Speed vs. Work. A comparison of the computation rates and the work for each method indicates that an increased amount of work sometimes offsets a fast computation rate for some methods. The direct vectorized Choleski solvers have much higher computation rates than the iterative or sparse Choleski solvers on all three problems. However, the direct vectorized Choleski solvers generally require much more work than both the iterative solvers and the Testbed sparse solvers. This extra work limits the speedup actually obtained as measured by the execution time. For example, on the SRB problem, the computation rate for the skyline Choleski solver is over 30 times greater than the computation rate of the Testbed sparse solver but at the expense of 3 times as much work compared to the sparse solver. As a result the execution time for the skyline solver is just over 10 times faster than the sparse solver. For the 3-D cube problem the fast convergence of the iterative methods results in a much lower amount of work compared to both other solvers. The JCG method is faster than both the sparse Choleski solver and the banded Choleski solvers for this problem even though the computation rate of JCG is only half that of the kji banded solver.

The ICCG iterative methods require less work than the JCG iterative method for all four problems in table 5, indicating the effect of improved convergence rates for the ICCG method compared to JCG. However, the JCG iterative method has a higher computation rate than the ICCG iterative methods for all four problems and requires less cpu time than ICCG for two of the four problems. On a scalar computer the ICCG methods would be the fastest for all four problems but the better vector performance of the JCG method relative to the ICCG methods gives different results on the Cray-2.

Banded vs. Sparse Storage. The amount of work and memory requirements for the two panel problems illustrates a key difference between the vectorized Choleski solvers which use banded or profile storage and the iterative and sparse Choleski solvers which use various sparse storage schemes. For the banded and skyline storage schemes doubling the bandwidth of the panel stiffness matrices by using 9-node elements instead of 4-node elements doubled the storage requirements but nearly quadrupled the amount of work. This increase is expected since the amount of work for the factorization is proportional to the square of the bandwidth. For the iterative methods the storage requirements for the 9-node element panel problem are also doubled since the number of matrix coefficients is nearly doubled. However, the amount of work is only doubled for the iterative methods since the work for matrix-vector multiplications is proportional only to the number of non-zero coefficients in the matrix. This key difference between algorithms which use banded

Table 5. Comparison of Equation Solvers for Example Problems

Method*	Time (sec)	Rate (mflops)	Work (+/x)	Memory (64-bit Words)	Overhead (reformat)
Panel, 4-Noded Elements, 2910 Equations, 50,494 Coefficients Bandwidth=425, Average Bandwidth=229					
kji Profile	1.7	119	198,255,170	671,201	.45
JCG (678)	4.3	56	240,440,514	81,198	.56
ICCG2 (231)	8.3	15	128,825,922	249,414	.61
INV/SSOL	6.9	8	53,002,512	—	—
Panel, 9-Noded Elements, 2910 Equations, 88,625 Coefficients Bandwidth=860, Average Bandwidth=439					
kji Profile	5.7	128	729,507,322	1,277,961	.73
JCG (857)	11.5	50	573,088,330	168,580	.95
ICCG1 (176)	10.0	13	133,018,764	262,965	.70
INV/SSOL	6.5	8	48,771,072	—	—
3-D Cube, 8-Noded Solid Elements, 2988 Equations, 99,525 Coefficients Bandwidth=336, Average Bandwidth=315					
kji Banded	1.9	169	315,876,861	1,003,968	.52
JCG (125)	.9	83	73,619,432	132,418	.74
ICCG2 (53)	2.9	23	67,947,440	420,222	.84
INV/SSOL	51.8	3	168,123,600	—	—
SRB, 54,870 Equations, 1,311,308 Coefficients Bandwidth=894, Average Bandwidth=382					
kji Profile	75.2	127	9,573,921,190	20,978,317	7.54
JCG (3114)	697.3	49	34,039,466,610	2,632,762	10.95
ICCG2 (562)	455.5	20	9,205,794,048	5,326,524	12.31
INV/SSOL	821.3	4	2,974,589,780	—	—

* Number of iterations in ( ) for iterative methods.

kji Profile - Choleski, profile storage, loop unrolling to level 4, local memory
kji Banded- Choleski, banded storage, loop unrolling to level 4, local memory
JCG - Jacobi Preconditioned Conjugate Gradient (Diagonal scaling) using diagonal storage of K
ICCG1 - Incomplete Choleski Preconditioned Conjugate Gradient using sparse column storage of K and L
ICCG2 - Incomplete Choleski Preconditioned Conjugate Gradient using diagonal storage for K, sparse column storage for L
INV/SSOL - Testbed sparse, node-oriented Choleski factor and solve routines

storage schemes and algorithms which use sparse storage schemes means that for very large problems where the bandwidth grows as the problem size increases, the number of computations required by the banded storage algorithm may eventually make the sparse solvers faster even though they are not as efficient on the Cray-2 architecture.

Initial Solver vs. New Solvers. A comparison of the amount of work required by the banded solver to the work required by processors INV and SSOL for each problem indicates the varying effectiveness of finding node orderings which minimize fill for the sparse Choleski solver. The sparse Choleski solver was actually faster for the 9-node element panel problem even though the stiffness matrix contains more non-zero coefficients. This is most likely due to the minimum degree node ordering being more effective in minimizing fill for the 9-node element panel problem. The ratio of work required by the profile solver compared to the work required by processors INV and SSOL is over 15 to 1 while for the cube problem the same ratio for the banded Choleski solver to INV and SSOL is less than 2 to 1. A better sparse solver would probably make the sparse solver faster for the 9-node element panel problem, but it is doubtful that enough improvement could be made in the computation rate for the cube problem or the SRB problem to make the sparse solver competitive with the best times for the new solvers.

Overhead Time. The overhead time required to reformat the stiffness matrices from the sparse storage scheme used by the Testbed into the in-core storage schemes used by each solver is also given in table 5. The overhead is small compared to the equation solution time for each problem. This small overhead time is important because it demonstrates the possibility of obtaining significant reductions in solution time for computationally intensive portions of a large existing code without rewriting the entire code. This strategy can be applied to other computation modules in a large code, providing an interface between code which is designed to exploit a given advanced architecture and code which is written for a general purpose applications code independent of the specific computer architecture.

Summary. Substantial reductions in the time required to solve linear systems for structural analysis problems can be made by the correct choice of methods as demonstrated by a comparison of solution times in table 5. The choice of methods must be based on the relative efficiency of each method on the given computer architecture and on several additional factors which affect the amount of work required by each method. The direct methods which use banded or profile storage are much more efficient than the sparse Choleski solver as measured by computation rate but they require much more memory. For very large problems with only moderate average bandwidth, such as the SRB example problem, the profile solver is much faster than the sparse solver. Higher bandwidth problems require substantially more work for the banded or profile Choleski methods and fast sparse solvers may be better. The efficiency of sparse solvers depends greatly on finding node orderings which minimize the fill which occurs during the factorization stage but is also limited by the need for indirect addressing due to the sparse storage schemes. The relative performance of the preconditioned conjugate gradient iterative methods is influenced greatly by the condition number of the stiffness matrix. The diagonal storage scheme significantly improves the computation rate for many problems, and, for well conditioned problems, reduced memory requirements and faster execution times compared to the direct methods make the itera-

tive methods superior. The most effective preconditioning strategies are also very costly in terms of efficiency on the Cray-2, limiting their usefulness on many problems.

Concluding Remarks

Two approaches have been described for solving linear systems of equations and results have been presented for three structural analysis problems representative of a wide class of structural analysis problems. Direct methods have been described which exploit the architectural features of the Cray-2 and perform best for large problems with moderate size bandwidths. The iterative methods described are not able to exploit the Cray-2 architecture as effectively as the direct methods but are superior for well-conditioned problems and require less memory than the banded solvers for very large problems. The new methods were evaluated by installing the methods in the CSM Testbed and solving structural analysis problems. The new equation solvers significantly improve computerized structural analysis by reducing the equation solution time required by the analysis. Both the direct and iterative methods take advantage of the very large main memory of the Cray-2 by storing the matrices in main memory.

The strategy of incorporating computationally efficient modules or routines into large existing code is an effective way to significantly improve the performance of existing software of new advanced computer architectures. The small overhead required to reformat matrices for the problems considered in this report demonstrates the effectiveness of this strategy. More research is necessary both to develop iterative methods with better preconditioners which are not limited by short vector lengths and indirect addressing and to identify the classes of problems for which iterative methods are superior. Very large three dimensional problems with only displacement degrees of freedom may be well conditioned and thus attractive for iterative methods. Non-linear problems requiring many solutions of linear systems where the solution of one system from the previous step is a good approximation to the solution at the current step are also attractive for iterative methods. Parallel versions of the methods presented in this report also may result in significant reductions in execution time for very large problems by using the multi-processing capability of the Cray-2.

REFERENCES

1. Ortega, J.; and Voigt, R.: *Solution of Partial Differential Equations on Vector and Parallel Computers.* SIAM Review, Vol. 27, 1985, pp. 149-240.

2. Ortega, J.: *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York, 1988.

3. Dongarra, J.; Moler, C.; Bunch, J.; and Stewart, G.: *LINPACK Users' Guide* . SIAM, Philadelphia, 1979.

4. Madsen, N.; Rodrigue, G.; and Karush, J.: *Matrix Multiplication by Diagonals on a Vector/Parallel Processor.* Information Processing Letters, Vol. 5, 1976, pp. 41-45.

5. Poole, E.: *Multi-color Incomplete Cholesky Conjugate Gradient Methods on Vector Computers.* Ph.D. Dissertation, Applied Mathematics, University of Virginia, 1986.

6. Meijerink, J. A.; and van der Vorst, H. A.: *An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix.* Mathematics of Computation, Vol. 31, 1977, pp. 148-162.

7. Meijerink, J. A.; and van der Vorst, H. A.: *Guidelines for the Usage of Incomplete Decompositions in Solving Sets of Linear Equations as They Occur in Practical Problems.* Journal of Computational Physics, Vol. 44, 1981, pp. 134-155.

8. Poole, E.; and Ortega, J.: *Multicolor ICCG Methods for Vector Computers.* SIAM Journal of Numerical Analysis, Vol. 24, 1987, pp. 1394-1418.

9. Manteuffel, T. A.: *An Incomplete Factorization Technique for Positive Definite Linear Systems.* Mathematics of Computation, Vol. 34, 1980, pp. 473-497.

10. George, A.; and Liu, J.: *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Inc., New Jersey, 1981.

11. Everstine, G. C.: *The BANDIT Computer Program for the Reduction of Matrix Bandwidth for NASTRAN.* NSRDC Report 3872, March 1972.

12. Gillian, R. E.; and Lotts, C. G.: *The CSM Testbed Software System - A Development Environment for Structural Analysis Methods on the NAS CRAY-2.* NASA TM-100642, 1988.

13. Felippa, C. A.: *The Computational Structural Mechanics Testbed Architecture: Volume 1 - The Language.* NASA CR-178384, 1988.

14. Knight, N.F.; McCleary, S.L.; and Macy, S.C.: *Large Scale Structural Analysis: The Structural Analyst, the CSM Testbed, and the NAS System.* NASA TM-100643, 1988.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-4159		

4. Title and Subtitle	5. Report Date
The Solution of Linear Systems of Equations With a Structural Analysis Code on the NAS CRAY-2	December 1988
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Eugene L. Poole and Andrea L. Overman	
9. Performing Organization Name and Address	10. Work Unit No.
Analytical Services and Materials, Inc. Hampton, VA 23665-5225	505-63-01-10
	11. Contract or Grant No.
	NAS1-18599
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes
Subcontracted to Awesome Computing Inc., 12 Rivanwood Place, Charlottesville, Va. 22901
Langley Technical Monitor: W. Jefferson Stroud

16. Abstract

Two methods for solving linear systems of equations on the NAS Cray-2 are described. One is a direct method; the other is an iterative method. Both methods exploit the architecture of the Cray-2, particularily the vectorization, and are aimed at structural analysis applications. To demonstrate and evaluate the methods, they were installed in a finite element structural analysis code denoted the Computational Structural Mechanics (CSM) Testbed. A description of the techniques used to integrate the two solvers into the Testbed is given. Storage schemes, memory requirements, operation counts, and reformatting procedures are discussed. Finally, results from the new methods are compared with results from the initial Testbed sparse Choleski equation solver for three structural analysis problems. The new direct solvers described in this paper achieve the highest computation rates of the methods compared. The new iterative methods are not able to achieve as high computation rates as the vectorized direct solvers but are best for well-conditioned problems which require fewer iterations to converge to the solution.

17. Key Words (Suggested by Authors(s))

Computational Structural Mechanics
Equation Solvers
Cray-2/Vectorization
Structural Analysis/Numerical Analysis

18. Distribution Statement
Unclassified—Unlimited

Subject Category 64

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	36	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171