A Parallel-Vector Algorithm for Rapid Structural Analysis on High-Performance Computers

Olaf O. Storaasli
Structural Mechanics Division
NASA Langley Research Center, Hampton, VA 23665-5225

Duc T. Nguyen and Tarun K. Agarwal
Department of Civil Engineering
Old Dominion University, Norfolk, VA 23529-0369

Abstract

A fast, accurate Choleski method for the solution of symmetric systems of linear equations is presented. This direct method is based on a variable-band storage scheme and takes advantage of column heights to reduce the number of operations in the Choleski factorization. The method employs parallel computation in the outermost DO-loop and vector computation via the "loop unrolling" technique in the innermost DO-loop. The method avoids computations with zeros outside the column heights, and as an option, zeros inside the band. The close relationship between Choleski and Gauss elimination methods is examined. The minor changes required to convert the Choleski code to a Gauss code to solve non-positive-definite symmetric systems of equations are identified. The results for two large-scale structural analyses performed on supercomputers, demonstrate the accuracy and speed of the method.

Nomenclature

e_a	error norm for solution residuals
e_s	strain energy error norm
{f}	load vector
hpm	hardware performance monitor (Cray)
i,j,k	DO loop indices
ja	job accounting utility (Cray)
[K]	stiffness matrix
MFLOPS	Million FLoating point OPerations/Second
m_{ij}	multipliers for forward substitution
n	number of equations
NP	number of processors
{R}	error residual for solution: [K] {x} - {f}
RAM	Random Access Memory
SAXPY	$\sum$ ax + y, or scalar $*$ vector + vector
second	CPU time function (Cray)
SRB	space shuttle Solid Rocket Booster
timef	elapsed time function (Cray)
[U]	upper triangular, factored stiffness matrix
u_{ij}	terms of upper-triangular matrix
{x}	static structural displacements

1. Introduction

Since the invention of the first electronic computer by Atanasoff to solve matrix equations of order 29 in 1939[1], researchers in many scientific and engineering disciplines have found their problems invariably reduced to solving systems of simultaneous equations that simulate and predict physical behavior. Currently, the solution of linear systems of equations on advanced parallel-vector computers is a key area of research with applications in many disciplines[2-6]. Structural analysis codes in wide use today were developed on single processor computers and often do not fully exploit the vector or parallel processing capability of modern high-performance computers. To achieve a high level of efficiency on parallel-vector supercomputers, a restructuring of the equation solution procedure and the memory and data management of these structural analysis codes is required. For example, the skyline storage technique used in many sequential structural analysis codes lacks the efficiency of other storage techniques used in the solution of linear systems of equations on vector computers[7-8]. Of equal importance, several parallel equation solvers have been demonstrated to work well for static and dynamic structural analyses, eigenvalue and buckling analyses, sensitivity analysis and structural optimization[9-15]. Since high-performance computers currently have both parallel and vector capability, the algorithms that exploit both will achieve optimal performance for these computers.
Based on favorable experience on sequential computers, a parallel-vector Choleski algorithm using a skyline storage scheme was developed and shown to have excellent parallel performance on a Cray 2 supercomputer as the number of processors increased[16]. However, the skyline scheme was found to prohibit the traditional loop unrolling technique used to optimize vector performance, so a less powerful "vector unrolling" strategy was used.

The present paper describes a new algorithm that overcomes the deficiency of skyline storage by using a variable-band storage scheme. The objective of this paper is to describe this new algorithm for solving matrix equations and to demonstrate its accuracy and speed by solving large-scale structural analysis applications on Cray supercomputers.

Since equation solution algorithms depend on the storage scheme selected, two of the storage schemes used most

often are discussed in Section 2 of the present paper. A description of how the basic Choleski method was implemented to achieve both vector and parallel speed is discussed in Section 3. The parallel FORTRAN language, **Force**[17], used to implement this method, is also discussed in Section 3. The results obtained for two large-scale structural analysis problems to evaluate the performance of the algorithm are discussed in Section 4. The minor changes required to convert this newly-developed code from a Choleski algorithm to a Gauss algorithm for solving non-positive-definite symmetric systems of equations are identified with examples in Appendix A. A description of the input data with a simple example is in Appendix B. A listing of the code and its use, both in a stand-alone mode and in the CSM Testbed[18], is described in Appendix C.

2. Data Storage Schemes

The Choleski method for the solution of simultaneous equations requires the decomposition of the matrix of stiffness coefficients, [K], into an upper-triangular, factored stiffness matrix, [U]. Details of this matrix decomposition are given in Section 3 and Appendix A. Two methods most often used in structural analysis codes to store [U] are the variable-band, and skyline techniques.

For large finite-element applications, the user defines the geometry, finite elements and loads of the finite-element model. The user may use automated algorithms to reorder the resulting stiffness matrix, [K], in the form that is most efficient for the solver. The reverse Cuthill-McKee algorithm[19] reorders the [K] matrix into a near minimum bandwidth, and thus is used for the examples in this paper.

In a row-oriented, variable-bandwidth Choleski approach, the bandwidth of each row of the upper-triangular matrix, [U], is defined as the number of coefficients from a diagonal term to the last non-zero coefficient of the row, excluding the diagonal term. The coefficients of the stiffness matrix for a stiffened panel with a circular cutout (bottom of Fig. 1), are plotted in a variable-band format as shown in Fig. 1.

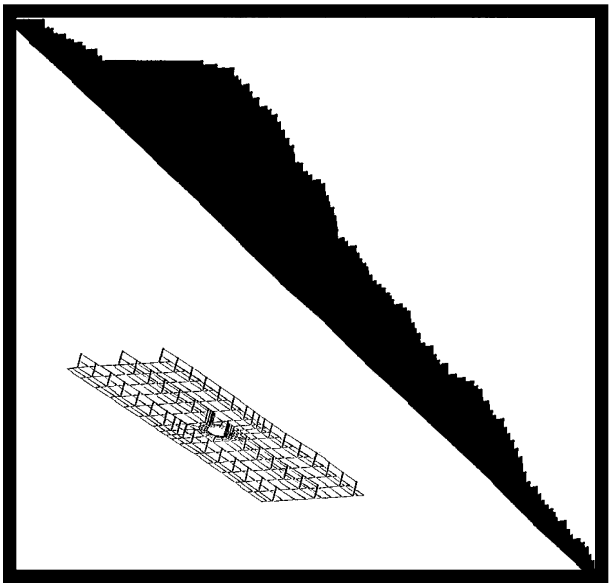

Fig. 1 Variable-band row storage of panel matrix.

The coefficients of the matrix are stored by rows where each row represents a degree of freedom in the finite-element model. The variable-band storage includes all zero coefficients within the so called "profile" which is defined by the ragged right edge of the matrix represented in Fig. 1. Variable-band storage requires less memory than earlier schemes which stored all coefficients within the maximum bandwidth, since earlier schemes stored and operated on many zeros outside the variable-band profile.

The same panel stiffness matrix is stored by columns in the skyline format, like skyscrapers, in Fig. 2 from each diagonal coefficient up to the last nonzero directly above it.

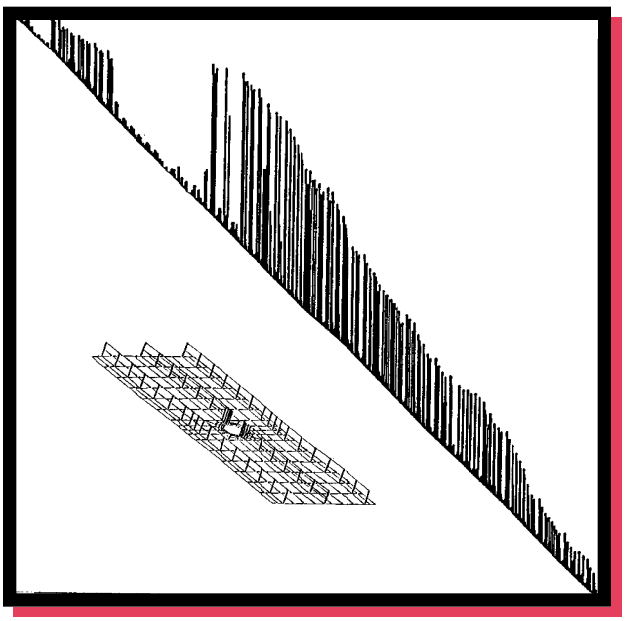

Fig. 2 Skyline column storage of panel matrix.

In this column-oriented storage scheme, the column height is defined as the number of coefficients from a diagonal coefficient to the last nonzero coefficient in the same column, excluding the diagonal coefficient, as shown in Fig. 2. This skyline format requires fewer coefficients to store and operate on during equation solution as indicated by the many zeros (white spaces) in Fig. 2. The panel example is used for illustrative purposes only, as in many applications, the reduction in storage offered by the skyline approach is not so pronounced.

Factorization of a matrix using skyline storage has the advantage that calculations with zeros outside the skyline need not be performed since zeros remain in these locations after factorization. Although the skyline method has the advantage of minimizing the storage and number of operations required on sequential computers, it cannot achieve optimal vector speed on high-performance computers since it cannot use efficient SAXPY operations (i.e., $\sum$ ax + y, or scalar $*$ vector + vector). SAXPY operations achieve optimal performance on vector computers since they continually stream operations to separate add and multiply units which can operate simultaneously.

To compare the storage schemes in detail, the location of the coefficients in the upper half of a 9x9 symmetric stiffness matrix are shown in Fig. 3 as a simple illustrative example.

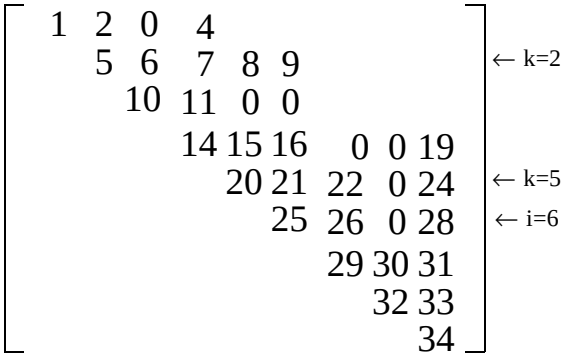

Fig. 3 Variable-band storage of stiffness matrix.

The non-zero integers in Fig. 3 are the index (location) of each stiffness coefficient stored contiguously in a one-dimensional array. The 34 matrix coefficients are numbered row-wise according to a variable-band storage scheme, where for illustrative purposes, the seven zeros are stored within five of the rows. The skyline storage scheme requires only 29 locations to store the same matrix, since the five zeros in columns 3, 7 and 8 in Fig. 3 fall outside the skyline and need not be stored. The two zeros in row 3 must be stored in both the variable-band and skyline storage schemes since they may become non-zero during factorization. The bandwidth of row 2 in Fig. 3 is 4, excluding the diagonal coefficient, and the height of column 6 is 4, excluding the diagonal coefficient.

The parallel-vector Choleski method, described in Section 3, uses a variable-band storage scheme to achieve optimal vector performance combined with the skyline column heights to avoid calculations with zeros outside the skyline.

3. Parallel-Vector Choleski Method Development

Basic Sequential Choleski Method

In the sequential Choleski method, a symmetric, positive-definite stiffness matrix, [K], can be decomposed as

$$[K] = [U]^T [U] \tag{1}$$

with the coefficients of the upper-triangular matrix, [U]:

$$u_{ij} = 0 \quad \text{for} \quad i > j \tag{2}$$

$$u_{11} = \sqrt{K_{11}} \; ; \; u_{1j} = \frac{K_{1j}}{u_{11}} \quad \text{for } j \geq 1 \tag{3}$$

$$u_{ii} = \sqrt{K_{ii} - \sum_{k=1}^{i-1} u_{ki}^2} \quad \text{for } i > 1 \tag{4}$$

$$u_{ij} = \frac{K_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj}}{u_{ii}} \quad \text{for } i, j > 1 \tag{5}$$

When j=i, the numerator of Eq. 5 is identical to Eq. 4 without the square root operation, which simplifies coding.

Regardless of whether the Choleski or Gauss method is used (see Appendix A), the basic skeleton FORTRAN sequential code for matrix factorization is given in Fig. 4 with comments inserted to explain the connection to Eqs. 3-5.

```
      DO 1  i = row#1, row#n
      DO 2  k = top row# of i^th column, i-1
c     compute multiplication factor, xmult
      xmult = U(k,i)
cgauss xmult = U(k,k) * U(k,i) replaces above statement
      DO 3  j = i, k + row length of row k
c     calculate the numerator of Eq. 5
                U(i,j) = K(i,j) - xmult * U(k,j)
   3  Continue
   2  Continue
c     calculate final value of U(i,i) as in Eq. 4
      U(i,i) = SQRT(U(i,i))
cgauss remove above statement
c     DO loop 4 divides the numerator of Eq. 5 by u_ii
      xinv = 1/U(i,i)
      DO 4 j = i+1, i + row length of row i
      U(i,j) = U(i,j) * xinv
   4  Continue
   1  Continue
```

Fig. 4 Sequential Choleski variable-band skeleton code for matrix factorization.

To use the Gauss solution method (i.e., for non-positive-definite systems of equations, see Appendix A), only two FORTRAN statements, labeled cgauss in Fig. 4, change.

The multiplier constants, xmult, and the column height information[16,20] are utilized in the DO 2 loop in Fig. 4 to avoid operations with zeros outside the column height (or skyline). The parameter, k, of the DO 2 loop is illustrated in Fig. 3. For i=6 (in DO 1 of Fig. 4), the index k (in DO 2) has the values from 2 to 5 as shown in Fig. 3.

Although [K] and [U] are two-dimensional arrays in Fig. 4, in the actual Choleski factorization code, both are stored in a one-dimensional array (as in Table 3 of **16**). The modifications required for the basic, sequential Choleski code to achieve optimal vector and parallel performance (i.e., minimal solution time) are given next.

Vectorize Choleski Code with Loop Unrolling

For a single processor with vector capability, the loop-unrolling technique (suitable for SAXPY operations) can be exploited to significantly improve performance. The SAXPY operation is one of the most efficient computations on vector computers since vector operations are performed in parallel on separate add and multiply functional units.

In Fig. 3, for example, once the first four rows of the factored matrix, [U], have been completely updated, row 5 can be updated according to the numerator of Eq. 5:

$$u_{5j} = k_{5j} - u_{15} * u_{1j} - u_{25} * u_{2j} - u_{35} * u_{3j} - u_{45} * u_{4j} \qquad (6)$$

In Eq. 6, u_{15}, u_{25}, u_{35} and u_{45} are multiplier constants. Thus, u_{15} (or u_{25}, u_{35}, u_{45}), u_{1j} (or u_{2j}, u_{3j}, u_{4j}) and k_{5j} play the role of the terms a, x and y, respectively, in SAXPY operations. The SAXPY operations in Eq. 6 are also loop unrolled to level 4 since operations on four rows are stacked together into one FORTRAN arithmetic statement. This loop unrolling is possible since "partial" updated values of row 5 can be computed when any of the first four rows are complete.

In a previous paper using the column-oriented Choleski method[16], once the first four columns of the factored matrix, [U], were completely updated, all terms of column 5 were updated. For example, u_{25} was computed by Eq. 5 as:

$$u_{25} = \frac{k_{25} - (u_{12} * u_{15})}{u_{22}} \qquad (7)$$

The term u_{25} in Eq. 7 was computed directly as the "final" updated value, and could not be expressed in terms of "partial" updates as is the case in Eq. 6. Therefore, the loop unrolling technique could not be used in this case. Instead, a vector unrolling strategy[16] was used to improve the vector performance in Eq. 5.

However, in the present paper, the sequential Choleski code in Fig. 4 can be modified to include loop-unrolling, say to level 4 as is shown in Fig. 5.

```
      DO 1  i = row#1, row#n
      DO 2  k = top row# of i^th column, i-1, 4
      DO 3  j = i, k + row length of row k
c     Eq. 6 (numerator of Eq. 5) code follows
          U(i,j) = K(i,j) - U(k,i) * U(k,j)
                          - U(k+1,i) * U(k+1,j)
                          - U(k+2,i) * U(k+2,j)
                          - U(k+3,i) * U(k+3,j)
   3  Continue
   2  Continue
c     repeat loop 2 to update i^th row by extra k values
c     for DO 2 k = 1, 10, 4, extra k values are 9,10
      U(i,i) = SQRT(U(i,i))
      xinv = 1/U(i,i)
      DO 4 j = i+1, i + row length of row i
      U(i,j) = U(i,j) * xinv
   4  Continue
   1  Continue
```

Fig. 5 Vectorized Choleski factorization code (with level 4 loop unrolling).

Using the loop-unrolling technique, the total number of load and store instructions and operations between the main memory and the vector registers is reduced significantly for nested DO-loops. The modified outer loop (DO 2 in Fig. 5), has an increment equal to the level of unrolling, while the innermost loop (DO 3 in Fig. 5) contains more arithmetic computations in a single FORTRAN statement

than the basic code. For vector supercomputers, such as Cray, SAXPY operations are known to be faster than dot-product operations used in the skyline method. The use of a variable-band is preferred to the skyline storage scheme since it permits the SAXPY operations of Eq. 6.

In addition to vector capability, modern high-performance computers also have multiple processors which can operate in parallel. Considerably more work is required by engineers to achieve parallel performance gains than to achieve vector performance gains, since code must be restructured for processor synchronization and load balancing. The parallel-vector Choleski method was coded (in the **Force** parallel FORTRAN language) as the computer program **pvsolve**. **Pvsolve** will be described after a brief synopsis of **Force**.

Parallel FORTRAN Language, Force

Force is a preprocessor which produces executable parallel code from a combination of FORTRAN and a set of simple, yet portable, parallel extensions tailored to run efficiently on parallel computers[17]. The parallel extensions used in **pvsolve** are **Prescheduled** DO, **Shared** and **Private** variables, **Produce** and **Copy**. **Prescheduled** DO causes all processors to execute the same DO-loop statements in parallel simultaneously with each processor using a different DO-loop index. Variables can be either **Shared** between all processors or **Private** (each processor has its own value for the same variable name). Care should be taken to avoid large **Private** arrays, as they are stored in different memory locations for each processor. Therefore, **Shared** arrays are preferred to **Private** arrays. **Copy** and **Produce** are used to synchronize tasks. **Copy** X into Y stores X in Y only if X is "full" (i.e., a signal to all processors to resume their computations), otherwise the processor waits. **Produce** X = K assigns K to X and marks X as "full". If X is "full", **Produce** waits until X is "empty" (i.e., a signal for processors to wait) before assigning K to X. **Force** permits algorithms to be independent of both the computer and the number of processors, as the number of processors is not specified until run time.

Parallel-Vector Choleski Factorization

In Choleski-based methods, a symmetric, positive definite stiffness matrix, [K], can be decomposed as shown in Eq. 1.

For example, u_{57} can be computed from Eq. 5 as:

$$u_{57} = \frac{k_{57} - u_{15}u_{17} - u_{25}u_{27} - u_{35}u_{37} - u_{45}u_{47}}{u_{55}} \qquad (8)$$

The calculations in Eq. 8 for the term u_{57} (of row 5) only involve columns 5 and 7. Furthermore, the "final value" of u_{57} cannot be computed until the final, updated values of the first four rows have been completed. Assuming that only the first two rows of the factored matrix, [U], have been completed, one still can compute the second partially-updated value of u_{57} as designated by superscript (2):

$$u_{57}^{(2)} = k_{57} - u_{15}\,u_{17} - u_{25}\,u_{27} \qquad (9)$$

If row 3 has also been completely updated, then the third partially-updated value of u_{57} can be calculated as:

$$u_{57}^{(3)} = u_{57}^{(2)} - u_{35}\,u_{37} \qquad (10)$$

This observation suggests an efficient way to perform Choleski factorization in parallel on NP processors. For example, each row of the coefficient stiffness matrix, [K], is assigned to a separate processor.

From Equation 8, assuming NP = 4, it is seen that row 5 cannot be completely updated until row 4 has been completely updated. In general, in order to update the i^{th} row, the previous (i-1) rows must already have been updated. For the above reasons, any NP consecutive rows of the coefficient stiffness matrix, [K], will be processed by NP separate processors. As a consequence, while row 5 is being processed by a particular processor, say processor 1, then the first (5-NP) rows have already been completely updated. Thus, if the i^{th} row is being processed by the p^{th} processor, there is no need to check every row (from row 1 to row i-1) to make sure they have been completed. It is safe to assume that the first (i-NP) rows have already been completed as shown in the triangular cross-hatched region of Fig. 6.

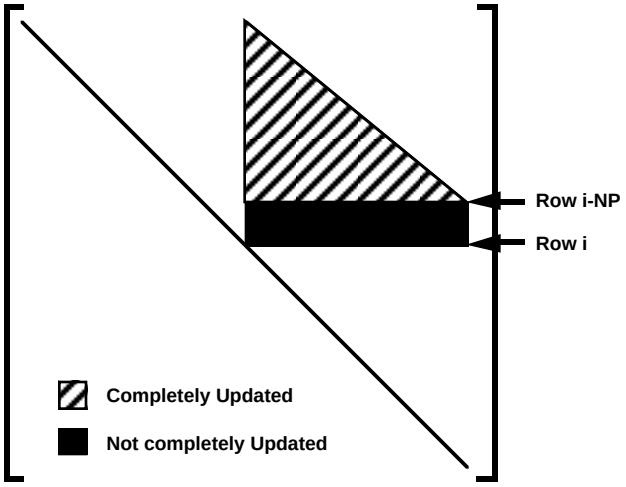

Fig. 6 Information required to update row i.

Synchronization checks are required only for the rows between (i-NP+1) and (i-1) as shown in the rectangular solid region of Fig. 6. Since the first (i-NP) rows have already been completely factored, the i^{th} row can be "partially" processed by the p^{th} processor as shown in Equations 9-10.

The vectorized Choleski code in Fig. 5 has been modified for parallel processing. The resulting skeleton factorization part of the full **pvsolve** code is shown in Fig. 7 with parallel (**Force**) statements in boldface type.

```
      Shared K(21090396)
      Private i,j,k,temp,xinv
c     {X} vector used to indicate when row is finished
      [U] overwrites [K] in actual code to reduce storage
c     calculate U(1,1) in Eq. 3 on one processor
      U(1,1) = SQRT(K(1,1))
c     divide row#1 by U(1,1) as in Eq. 3
c     declare row#1 finished
      Produce X(1) = U(1,1)
c     start all available processors
      Presched DO 1 i = row#2, row#n
c     lock processor if row# (i-NP) is not finished
c     release lock when row is finished
      IF(i-NP.GT. 0) then
      Copy X(i-NP) into temp
      End if
      DO 2 k = top row# of the ith column, i-NP, 4
c     skip DO 3 if all multipliers are zero: zero checking
            DO 3 j = i, k + rowlength of row k
            U(i,j) = K(i,j) - U(k,i) * U(k,j)
                            - U(k+1,i) * U(k+1,j)
                            - U(k+2,i) * U(k+2,j)
                            - U(k+3,i) * U(k+3,j)
    3 continue
    2 continue
c     lock the processor if row# (i-1) not finished
c     release the lock when row#(i-1) is finished
      Copy X(i-1) into temp
      DO 4 k=max(top row# of ith column, i-NP+1), i-1
      DO 5 j = i, k + rowlength of row k
      U(i,j) = U(i,j) - U(k,i) * U(k,j)
    5 continue
    4 continue
      U(i,i) = SQRT(U(i,i))
      xinv = 1/U(i,i)
      DO 6 j = i+1, i + rowlength of row i
      U(i,j) = U(i,j) * xinv
    6 continue
c     broadcast to all processors that row i is finished
      Produce X(i) = U(i,i)
    1 End Presched DO
```

Fig. 7 Parallel-vector Choleski skeleton code (with level 4 loop unrolling).

Solution of Triangular Systems

The forward/backward solution can be made parallel in the outermost loop by using synchronization statements, and can result in excellent computation speed-up for an increasing number of processors on computers where synchronization time is fast compared to computation time. However, on Cray computers, the computations for the forward/backward solution time are so fast that for better performance in **pvsolve**, they are done on one processor with long vectors rather than introducing synchronization overhead on multiple processors. A further time reduction for one processor is obtained by using loop unrolling in the forward elimination and vector unrolling[16] (another form of loop unrolling) in the backward substitution.

4. Evaluation of Method for Structural Analyses

To test the effectiveness of **pvsolve**, described in Section 3, two large-scale structural analyses have been performed on the Cray Y-MP supercomputer at NASA Ames Research Center. These analyses involved calculating the static displacements resulting from initial loadings for finite element models of a high speed research aircraft and the space shuttle solid rocket booster (SRB). The aircraft and SRB models were selected as they were large, available finite-element models of interest to NASA. The Cray Y-MP was selected as it is a high-performance supercomputer with parallel-vector capability. To verify the accuracy of the displacements as calculated from the equilibrium equation (i.e. [K]{x} = {f}), the residual vector,

$$\{R\} = [K]\{x\} - \{f\} \qquad (11)$$

is calculated, and the absolute error norm,

$$e_a = \sqrt{\{R\}^T \{R\}} \qquad (12)$$

and strain energy error norm,

$$e_s = \{x\}^T [K] \{x\} - \{x\}^T \{f\} \qquad (13)$$

are evaluated. If no computer roundoff error occurs, all components in the residual vector, {R} are zero. However, performing billions of operations during equation solution introduces roundoff which, for accurate solutions, results in small values for {R}, e_a and e_s in Eqs. 11-13.

The solution times using **pvsolve** for the SRB application were also obtained on Cray 2 supercomputers at NASA Ames and NASA Langley and compared with solution times for the skyline algorithm in a previous paper[16].

In the following applications, code is inserted in **pvsolve** to calculate the elapsed time and number of operations taken by each processor for equation solution. The Cray timing and performance utilities (**timef**, **hpm**, **ja** and **second**) are used to measure the time, operations and speed of the equation solution on each processor. For each problem, the number of Million FLoating point OPerations is divided by the solution time, in Seconds, to determine the overall performance rate of the solver in MFLOPS. The timings obtained are conservative, since they were made with other users on the systems. In every case, times would be less and MFLOP rates more if **pvsolve** were run in a dedicated computer environment.

High Speed Research Aircraft

To evaluate the performance of the parallel-vector Choleski solver, a structural static analysis has been performed on a 16,146 degree-of-freedom finite-element model of a high-speed aircraft concept[21], shown in the upper right of Fig. 8:

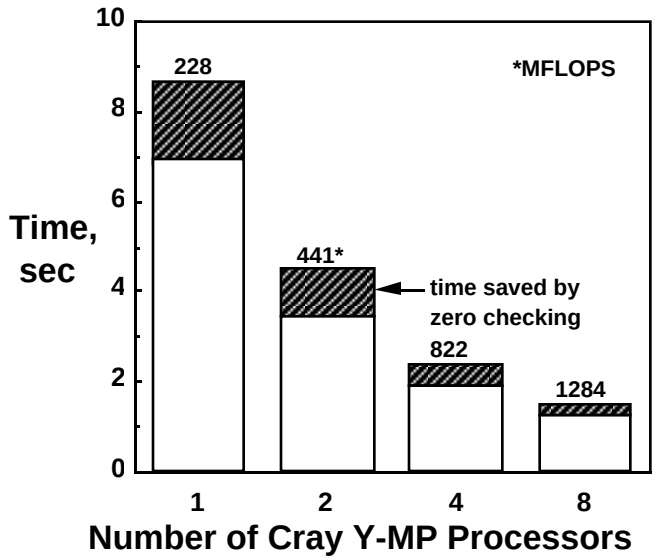

Fig. 8 Effect of more processors on analysis time (High-Speed Research Aircraft).

Since the structure is symmetric, a wing-fuselage half model is used to investigate the overall deflection distribution of the aircraft. The finite element model of the aircraft is generated using the CSM Testbed[18] where the stiffness matrix and load vector are in the form of processor ITER (with reset sipr=-2), described further in Appendix B. The half model contains 2851 nodes, 4329 4-node quadrilateral shell elements, 5189 2-node beam elements and 114 3-node triangular elements. The stiffness matrix for this model has a maximum semi-bandwidth of 600 and an average bandwidth of 321. The half-model is constrained along the plane of the fuselage centerline and subjected to upward loads at the wingtip and the resulting wing and fuselage deflections are calculated.

The numerical accuracy of the static displacements calculated is indicated by the small absolute and strain energy error norms of 0.000009 and 0.000005, respectively, computed from Eqs. 12-13. These residuals are identical no matter how many processors are used. The small values of the residuals indicates that the solution satisfies the original force-displacement equation. The residuals are independant of the number of processors indicating no error is introduced by synchronizing the calculations on multiple processors.

The time taken for a typical finite element code to generate the mesh, form and factor the stiffness matrix is 134 seconds on a Cray Y-MP (802 seconds on a Convex 220) of which the matrix factorization is 51 seconds. Using **pvsolve**, the factorization for this aircraft application requires 2 billion operations which reduces to 1.4 billion when operations with zeros are eliminated. Although CPU time is less for one processor, elapsed time is reported as it is the only meaningful measure of parallel performance. Factoring [K] with no zero checking takes 8.68 and 1.54 elapsed seconds (at a rate of 228 and 1284 MFLOPS) on one and eight Cray Y-MP processors, respectively, as shown in Table 1.

Table 1 Matrix decomposition time (MFLOPS) for aircraft on Cray Y-MP:
16,146 equations, bandwidth=600 max, 321 average
5,579,839 matrix size, 499,505 nonzeros

Processors	Sec (MFLOPS)	Sec (MFLOPS) with zero-checking
1	8.68 (228)	6.81 (203)
2	4.50 (441)	3.46 (399)
4	2.41 (822)	1.89 (730)
8	1.54 (1284)	1.29 (1071)

Eliminating operations with zeros within the variable bandwidth (zero checking, see Fig. 7) further reduces the solution time to 6.81 and 1.29 seconds, respectively, on one and eight processors. However, the reduced time with zero checking is accompanied by a reduction in computation rate (MFLOPS), since the added IF statements also reduce the number of operations. The reduction in computation time (nearly proportional to the number of processors) and the portion of time saved by zero-checking are shown in Fig. 8. The number above the bars (in MFLOPS) in Fig. 8 show the increased computation rate as the number of processors increases.

Space Shuttle Solid Rocket Booster (SRB)

In addition to the high-speed aircraft, the static displacements of a two-dimensional shell model of the space shuttle SRB, shown in the upper right of Fig. 9, have been calculated.

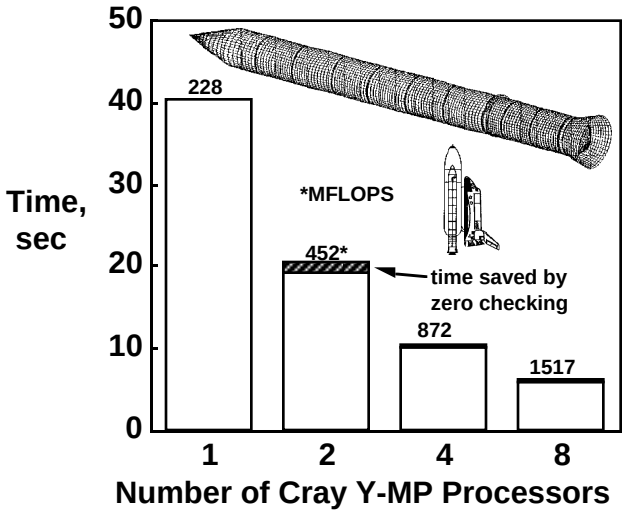

Fig. 9 Effect of more processors on analysis time (Space Shuttle SRB).

This SRB model is used to investigate the overall deflection distribution for the SRB when subjected to mechanical loads corresponding to selected times during the launch sequence[22]. The model contains 9205 nodes, 9156 4-node quadrilateral shell elements, 1273 2-node beam elements and 90 3-node triangular elements, with a total of 54,870 degrees of freedom. The stiffness matrix for this application has a maximum semi-bandwidth of 900 and an average bandwidth

of 383. A detailed description and analysis of this problem is given in references **22** and **23.**

The calculated absolute and strain energy residuals for the static displacements are 0.00014 and 0.0017, respectively, from Eqs. 12-13. This accuracy indicates that roundoff error in the displacement calculations is insignificant despite the 9.2 billion floating point operations performed.

The time for a typical finite element code to generate the mesh, form and factor the stiffness matrix is 391 seconds on the Cray Y-MP (15 hours on a VAX 11/785) of which the matrix factorization is 233 seconds (51,185 seconds on VAX). Using **pvsolve**, the factorization for this SRB problem, requires 40.26 and 6.04 seconds on one and eight Cray Y-MP processors, respectively, as shown in Table 2. Eliminating more than one billion operations on zeros further reduces the solution time to 5.79 seconds on eight processors but reduces the computation rate to 1444 MFLOPS. The CPU times are approximately 10 percent less than the elapsed times quoted on one processor.

Table 2 Matrix decomposition time (MFLOPS) (shuttle SRB on Cray Y-MP)

54,870 equations,bandwidth=900 max, 383 average
21,090,396 matrix size, 1,310,973 nonzeros

Processors	Sec. (MFLOPS)	Sec. (MFLOPS) with zero-checking
1	40.26 (228)	40.97 (224)
2	20.27 (452)	19.32 (425)
4	10.50 (872)	10.00 (821)
8	6.04 (1517)	5.79 (1444)

A reduction in matrix decomposition time by a factor of 7.08 on eight processors compared to one processor (for zero checking) is shown in Fig. 9. The corresponding computation rate for this matrix factorization, using eight processors on the Cray Y-MP is 1,517 MFLOPS. The previous fastest time to solve this problem on the Cray Y-MP using a sparse solver was 23 seconds on one processor and 9 seconds on eight processors for a speedup factor of 2.5[7,24].

For structural analysis problems with a larger average column height, and bandwidth than the aircraft or SRB discussed, one can expect **pvsolve** to perform computations at even higher MFLOPS rates since the majority of the vector operations are performed on long vectors. For example, a rate of 1784 MFLOPS has been achieved by **pvsolve** for a structural matrix with an average bandwidth of 699 on the eight-processor Cray Y-MP[25-26].

The decomposition time for the Shuttle SRB matrix using **pvsolve**, is compared to the skyline algorithm[16] in Fig. 10 for 1, 2 and 4 Cray 2 processors.

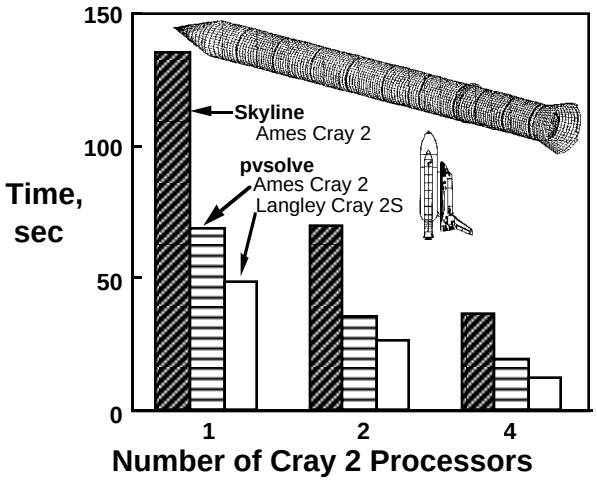

Fig. 10 SRB decomposition time comparison (pvsolve vs. skyline method[16]).

A reduction in decomposition time by a factor of 2 is shown for **pvsolve** in the figure for the Cray 2 at NASA Ames. An additional reduction in decomposition time of approximately 50 percent is shown for **pvsolve** on the newer Cray 2S at NASA Langley with faster memory access using static RAM compared to dynamic RAM on the Cray 2 at NASA Ames. The decomposition time for **pvsolve** using eight processors on the Cray Y-MP (six seconds in Fig. 9) is a reduction by factors of 23 and 6 when compared to the skyline solution on 1 and 4 Cray 2 processors, respectively, shown in Fig. 10.

The above results have been obtained using loop unrolling to level 9. On the Cray Y-MP supercomputer, the performance continues to increase until loop unrolling level 9, after which further performance gains are not significant compared to the complex coding required. The **pvsolve** code performed best with an odd number for loop unrolling, because both data paths to memory are used simultaneously at all times. The vector being modified plus the 9 unrolling vectors make ten total vectors, an even number, which keeps both data paths busy.

5. Concluding Remarks

A parallel-vector Choleski method for the solution of large-scale structural analysis problems has been developed and tested on Cray supercomputers. The method exploits both the parallel and vector capabilities of modern high-performance computers. To minimize computation time, the method performs parallel computation at the outermost DO-loop of the matrix factorization, the most time-consuming part of the equation solution. In addition, the most intensive computations of the factorization, the innermost DO-loop has been vectorized using a SAXPY-based scheme. This scheme allows the use of the loop-unrolling technique which minimizes computation time. The forward and backward solution phases have been found to be more effective to perform sequentially with loop-unrolling and vector-unrolling, respectively.

The parallel-vector Choleski method has been used to calculate the static displacements for two large-scale structural analysis problems; a high-speed aircraft and the space shuttle solid rocket booster. For both structural analyses, the static displacements are calculated with a high degree of accuracy as indicated by the small values of the absolute and strain energy error norms. The total equation solution time is small for one processor and is further reduced in proportion to the number of processors. The option to avoid operations with internal zeros in the matrix further reduces both the number of operations and the computation time for both applications.

Factoring the stiffness matrix for the space shuttle solid rocket booster, which formerly required hours on most computers and minutes on supercomputers by other methods, has been reduced to seconds using the parallel-vector variable-band Choleski method. The speed of **pvsolve** should give engineers and designers the opportunity to include more design variables and constraints during structural optimization and to use more refined finite-element meshes to obtain an improved understanding of the complex behavior of aerospace structures leading to better, safer designs. Since the algorithm is independent of the number of processors, it is not only attractive for current supercomputers, but also for the next generation of shared-memory supercomputers, where the number of processors is expected to increase significantly.

6. Appendix A

The row-oriented, sequential versions of both the Choleski and Gauss methods are presented together to illustrate how their basic operations are closely related and readily identified. To simplify the discussion, the following system of equations is used throughout this section:

$$[K]\{x\} = \{f\} \tag{14}$$

where
$$[K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \tag{15}$$

and
$$\{f\} = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix} \tag{16}$$

The solution of equations 14-16 is:

$$\{x\} = \begin{Bmatrix} 1 \\ 1 \\ 1 \end{Bmatrix} \tag{17}$$

The basic idea in both the Choleski and Gauss elimination methods is to reduce the given coefficient matrix, [K], to an upper triangular matrix, [U]. This process can be accomplished with appropriate row operations. The unknown vector, {x}, can be solved by the familiar forward and backward substitution.

Choleski Method

The stiffness matrix [K] of equation 15 can be converted into a Choleski upper-triangular matrix, [U], by appropriate row operations:

$$[K1] = [K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

$$\Rightarrow [K2] = \begin{bmatrix} \sqrt{2} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & -1 & 1 \end{bmatrix} \Rightarrow [K3] = \begin{bmatrix} \sqrt{2} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} \\ 0 & -1 & 1 \end{bmatrix}$$

$$\Rightarrow [K4] = \begin{bmatrix} \sqrt{2} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \Rightarrow [K5] = \begin{bmatrix} \sqrt{2} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} \\ 0 & 0 & \frac{1}{\sqrt{3}} \end{bmatrix}$$

where

Row 1 of [K2] = Row 1 of [K] /$\sqrt{\text{K1(1,1)}}$
Row 2 of [K2] = Row 1 of [K2] /$\sqrt{2}$ + Row 2 of [K1]
Row 2 of [K3] = Row 2 of [K2] /$\sqrt{\text{K2(2,2)}}$
Row 3 of [K4] = Row 2 of [K3] * $\sqrt{\frac{2}{3}}$ + Row 3 of [K3]
Row 3 of [K5] = Row 3 of [K4] /$\sqrt{\text{K4(3,3)}}$

The multiplier constants, m_{ij}, used in the forward substitution (or updating the right-hand side vector of Eq. 14) are the same as terms in the factorized upper-triangular matrix such that:

$$m_{12} = u_{12} = -\frac{1}{\sqrt{2}}\ ,\ m_{13} = u_{13} = 0\ ,\ m_{23} = u_{23} = -\frac{\sqrt{2}}{\sqrt{3}}$$

Gauss Elimination Method

As in the Choleski Method just described, the stiffness matrix, [K], of Eq. 15 can be converted into a Gauss upper-triangular matrix by appropriate row operations.

$$[K1] = [K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

$$\Rightarrow [K2] = \begin{bmatrix} 2 & -1 & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & -1 & 1 \end{bmatrix} \Rightarrow [K3] = \begin{bmatrix} 2 & -1 & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & 0 & \frac{1}{3} \end{bmatrix}$$

In this version of Gauss elimination, the multipliers m_{ij} can be obtained from the factored matrix, [U], as:

$$m_{12} = \frac{u_{12}}{u_{11}} = -\frac{1}{2}$$

$$m_{13} = \frac{u_{13}}{u_{11}} = \frac{0}{2} = 0$$

$$m_{23} = \frac{u_{23}}{u_{22}} = \frac{-1}{\frac{3}{2}} = -\frac{2}{3}$$

An alternative version of Gauss elimination where the final diagonal elements become 1 follows:

$$[K1] = [K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

$$\Rightarrow [K2] = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & -1 & 1 \end{bmatrix} \Rightarrow [K3] = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & -1 & 1 \end{bmatrix}$$

$$\Rightarrow [K4] = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \Rightarrow [K5] = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & 0 & 1 \end{bmatrix}$$

Since the final diagonal terms become one, in the computer code, the main diagonal of the factored matrix is used to store the diagonal terms before scaling.

For example, $u_{11} = 2$; $u_{22} = \frac{3}{2}$; and $u_{33} = \frac{1}{3}$. The multiplier m_{ij} is obtained from the factored matrix, [U], as:

$$m_{12} = u_{12} * u_{11} = -\frac{1}{2} \times 2 = -1$$

$$m_{13} = u_{13} * u_{11} = 0 \times 2 = 0$$

$$m_{23} = u_{23} * u_{22} = -\frac{2}{3} \times \frac{3}{2} = -1$$

Similarities of Choleski and Gauss Method

1) The Choleski and Gauss solution procedures are quite similar since both methods can be expressed in terms of row operations which differ only by the scale-factors as explained above.

2) For both methods, the multipliers, m_{ij}, used in the forward substitution (to update the right-hand-side vector of Eq. 14) can always be recovered conveniently from the factored, upper triangular matrix, [U].

3) Both methods can be adapted to solve unsymmetric systems of linear equations. The basic procedure is essentially the same as that outlined above except that the computer storage increases since the lower triangle matrix of the factored matrix is used to store the multipliers, m_{ij}. In some applications, partial pivoting may be useful.

4) Since the multipliers of the Choleski method are identical to its factored, upper triangular matrix, [U], the Choleski method is slightly more efficient than the Gauss method. However, the Gauss method can also be used to solve non-positive-definite systems of equations.

7. Appendix B

The input data and arguments required to call the equation solver, **pvsolve**, together with a simple 21-equation example are given in this Appendix. The user should have a limited knowledge of parallel computing and the parallel FORTRAN language **Force**[17]. **Pvsolve** contains a **Force** subroutine, PVS, which may be called by general purpose codes. The information required by PVS to solve systems of simultaneous equations (i.e., [K]{u} = {f}) is transferred via arguments in the call statement:

Forcecall PVS(a,b,maxa,irowl,icolh,neq,nterms,iif,opf)

where:

a = a real vector, dimensioned nterms, containing the coefficients of the stiffness matrix, [K].

b = a real vector, dimensioned neq, containing the load vector, {f}. Upon return from subroutine PVS, b contains the displacement solution, {u}.

maxa = an integer vector, dimensioned neq, containing the location of the diagonal terms of [K] in vector {a}, equal to the sum of the number coefficients.

irowl = an integer vector, dimensioned neq, containing the row lengths (i.e., half-bandwidth of each row excluding the diagonal term) of [K].

icolh = an integer vector, dimensioned neq, containing the column heights (excluding the diagonal term) of each column of the stiffness matrix, [K].

neq = number of equations to solve (= degrees of freedom).

nterms = the dimension of the vector, {a}, [= maxa(neq)].

iif = 1 factor system of equations without internal zero check
= 2 factor system of equations with internal zero check
= 4 perform forward/backward substitution
= 5 perform forward/backward substitution and error check

opf, ops = an integer vector, dimensioned to the number of processors (8 for Cray Y-MP), containing the number of operations performed by each processor during factor and solve, respectively.

For example, the values of these input variables to solve a system of 21 equations, whose right hand side is the vector of real numbers from 1. to 21., and [K] is the symmetric, positive-definite matrix in Fig. B1 are given in Table B1.

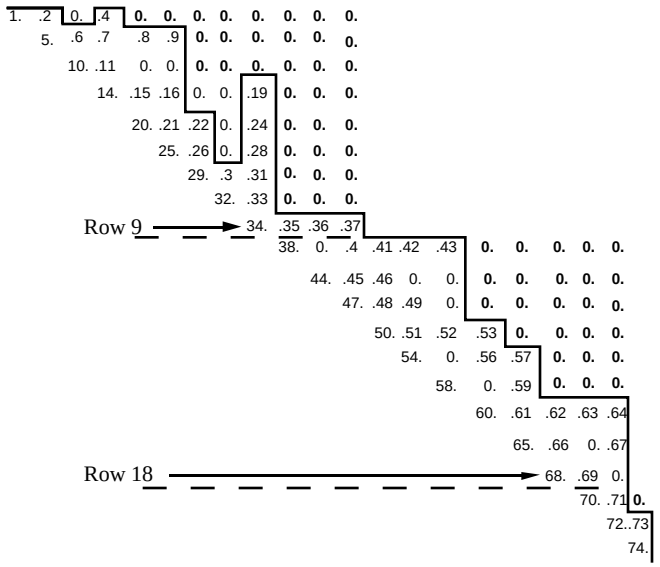

Fig. B1 Example [K] matrix with 21 equations.

The line in Fig. B1 represents the skyline defined by the column heights which extend up to the last nonzero in each column. The "extra zeros" outside the skyline (in boldface in Fig. B1) are required to achieve level 9 loop unrolling. The DO 2 loop in Fig. 5 illustrates this for level 4 loop unrolling. The vectors {a}, {b}, {maxa}, {icolh}, and {irowl} which are read by **pvsolve** are given in Table B1:

Table B1 Pvsolve input to solve [K]{x}={b} (example with 21 equations)

i	a(i)	b(i)	maxa(i)	icolh(i)	irowl(i)
1	1.	1.	1	0	11
2	.2	2.	13	1	10
3	0	3.	24	1	9
4	.4	4.	34	3	8
5	0	5.	43	3	7
6	0	6.	51	4	6
7	0	7.	58	2	5
8	0	8.	64	1	4
9	0	9.	69	5	3
10	0	10.	73	1	10
11	0	11.	84	2	9
12	0	12.	94	3	8
13	5.	13.	103	3	7
14	.6	14.	111	4	6
15	.7	15.	118	5	5
16	.8	16.	124	3	4
17	.9	17.	129	3	3
18	0	18.	133	2	2
19	0	19.	136	3	2
20	0	20.	139	4	1
21	0	21.	141	1	0
22	0				
23	0				
24	10.				
25	.11				
26-33	0				
34	14.				
35	.15				
36	.16				
37-38	0				
39	.19				
.	.				
.	.				
.	.				
135	0				
136	70.				
137	.71				
138	0				
139	72.				
140	.73				
141	74.				

where neq = 21 and nterms = 141. This input data is read at the beginning of the **pvsolve** program from the file 'COEFS.COLM' by subroutine CSMIN (see listing in Appendix C). The Force subroutine, PVS is then called twice; first to factor the matrix (iif = 2), and second to perform the forward/backward solution for displacements with error checking (iif = 5). A record is kept of number of floating point operations performed by each processor to factor and solve the matrix (totf, tots) as well as the elapsed (et0-et5) and task CPU time (t0-t5) on each processor at six key stages in the solution. Subroutine NORM reads the original matrix and load vector from the file 'COEFS.COLM' and evaluates the residual (Eq. 11) and the error norms (Eqs. 12-13).

8. Appendix C

A listing of the parallel-vector solution algorithm, **pvsolve**, coded in the parallel FORTRAN language, **Force**[17], follows in this Appendix. The code extends the skeleton code in Fig. 7 considerably by using loops unrolled to level 9 (instead of 4), one-dimensional vectors with pointers (instead of arrays) and by including the code for input/output, data handling, initialization, timing and counting operations. Following the **pvsolve** code is the command file used to obtain the static displacements for the aircraft and SRB structures using the Solid State Disk and 1,2,4 and 8 Cray Y-MP processors. The **pvsolve** code is all FORTRAN except for the **cdir$ ivdep** vector directive, and the **Force** parallel directives in **boldface** type. The dimension of the variables given on line 2 is for the static analysis of the 16,146 equation research aircraft and should be replaced by the dimensions given in line 3 to obtain the space shuttle SRB displacement solution All variables are **Private** unless they are declared as **Shared**.

```
      Force PVSOLVE of np ident me
      Shared real a(5208900),b(16150),at(499600),opf(8)
csrb  Shared real a(21090500),b(54890),at(1350761)
      Shared real t0(8),t1(8),t2(8),t3(8),t4(8),t5(8),ops(8)
      Shared real et0(8),et1(8),et2(8),et3(8),et4(8),et5(8)
      Shared integer maxa(16150),irow(16150),irowl(16150)
      Shared integer icoln(499600),icolh(16150),nc,neq
      End declarations
            et0(me)=timef()/1000.
            t0(me)=second()/np
       if (me.eq.1) then call CSMIN(a,b,maxa,irowl,icolh,neq,
     +     nterms,irow,icoln,nc,maxbw,8,locrow,iavebw)
        write(*,*)'* PVSOLVE - pvsolve - PVSOLVE Mar. 1990'
        write(*,*)'* Parallel-Vector equation SOLVEr by Olaf'
        write(*,*)'* Storaasli, Tarun Agarwal and Duc Nguyen'
        write(*,*)'* PVSOLVE is not authorized for use or'
        write(*,*)'* release to a third party without prior NASA'
        write(*,*)'* approval by Olaf Storaasli: 804-864-2927'
        write(*,*)'* FAX: 804-864-8318 or olaf@icase.edu'
        write(*,*)'* ',np,' proc. solve ',neq,' equations, nc= ',nc
        write(*,*)'* bandwidth: max= ',maxbw,', avg.= ',iavebw
        write(*,*)'* [k] matrix size, nterms= ',nterms,' words'
      endif
            et1(me)=timef()/1000.
            t1(me)=second()/np
      Barrier
      End barrier
            et2(me)=timef()/1000.
            t2(me)=second()/np
call PVS to factor [k] with internal zero check (iif = 2).....
            iif = 2
      Forcecall PVS(a,b,maxa,irowl,icolh,neq,nterms,iif,opf(me))
            et3(me)=timef()/1000.
            t3(me)=second()/np
call PVS to backsolve for {u} (iif = 4, 5 error check eqs. 11-13)
            iif = 5
      Forcecall PVS(a,b,maxa,irowl,icolh,neq,nterms,iif,ops(me))
            et4(me)=timef()/1000.
            t4(me)=second()/np
      Barrier
            nat=499600
            umax = abs(b(1))
            do 1 i=1,neq
1           umax = amax1(umax,abs(b(i)))
            write(*,*)'* Maximum displacement = ',umax
          if(iif.eq.5 ) call NORM(irowl,icoln,b,neq,nc)
c.....reorder displacements and write to CSM Testbed.....
           call TOCSM(b,irowl,icoln,at,at,icoln,8,nat)
                tmax1=0
                tmax2=0
                tmax3=0
                totf=0
                tots=0
         write(*,*)'** elapsed & cpu task time (sec) *****'
        write(*,*)'proc. force input Barrier factor f/b'
        do 2 i=1,np
          write(*,3)'wall ',i,et0(i),et1(i),et2(i),et3(i),et4(i)
          write(*,3)'tcpu ',i,t0(i),t1(i),t2(i),t3(i),t4(i)
                tmax1=max(tmax1,et3(i)-et2(i))
                tmax2=max(tmax2,et4(i)-et3(i))
                tmax3=max(tmax3,et4(i)-et2(i))
                totf=totf+opf(i)/1000000.
2               tots=tots+ops(i)/1000000.
3        format(a,i2,5f9.5)
         write(*,*) tmax1,' secs decomp, ',totf,
       + ' million ops. at ',totf/tmax1,' mflops '
         write(*,*) tmax2,' secs solve , ',tots,
       + ' million ops. at ',tots/tmax2,' mflops'
         write(*,*) tmax3,' secs TOTAL , ',totf+tots,
       + ' million ops. at ',(tots+totf)/tmax3,' mflops'
        End barrier
                et5(me)=timef()/1000.
                t5(me)=second()/np
          write(*,*)'proc. ',me,' tot wall=',et5(me),'tcpu=',t5(me)
                call exit(0)
      Join
      end
      Forcesub PVS(a,b,maxa,irowl,icolh,neq,nterms,iif,ops)
     +        of np ident me
       dimension a(*),b(*),icolh(*),maxa(*),irowl(*)
      Async real x(16150)
      End declarations
       if(iif.le.2) then
      Presched do 9 i = 1, neq
      Void x(i)
9      End presched do
                ops = 0
      Barrier
                a(1) = sqrt(a(1))
                xinv= 1.0/a(1)
cdir$ ivdep
           do 20 k = 1, irowl(1)
20           a(k+1) = xinv*a(k+1)
                ops = ops + irowl(1)+2
      Produce x(1)=a(1)
      End barrier
c.....factor stiffness matrix in parallel from row 2 to neq
      Presched do 100 i = 2, neq
                im1 = maxa(i)
                ic1 = icolh(i)
c......get indices to segment column i in 3 parts.............
            ibot = i - 9*( (i-1)/9 )
            icol = ic1 - ibot + 1
            icolp= icol/9
            itop = icol - 9*icolp
           jrow = i - ic1
            jm1  = maxa(jrow) + ic1
            jjrow=irowl(jrow)
            if (itop. ge. 1) then
           icopy = jrow + itop - 1
            if (isfull(x(icopy))) go to 331
```

```
      Copy  x(icopy)  into  temp
       endif
c......................................................................
331       go  to  (101,102,103,104,105,106,107,108),  itop
        go to 150
cdir$   ivdep
101       do 111 k = 1, jjrow-ic1+1
            km1 = k-1
111         a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
        go to 150
102                    jm2 = jm1 + jjrow
cdir$   ivdep
         do 112 k = 1, jjrow-ic1+1
            km1 = k-1
112          a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
   +                                 -a(jm2)*a(jm2+km1)
        go to 150
103                    jm2 = jm1 + jjrow
                       jm3 = jm2 + jjrow -1
cdir$   ivdep
         do 113 k = 1, jjrow -ic1+1
            km1 = k -1
113           a(im1+km1) = a(im1+km1) - a(jm1)*a(jm1+km1)
   +                          -a(jm2)*a(jm2+km1)  -a(jm3)*a(jm3+km1)
        go to 150
104                    jm2 = jm1 + jjrow
                       jm3 = jm2 + jjrow -1
                       jm4 = jm3 + jjrow -2
cdir$   ivdep
         do 114 k = 1, jjrow -ic1+1
            km1 = k -1
114            a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
   +                          -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
   +                          -a(jm4)*a(jm4+km1)
        go to 150
105                    jm2 = jm1 + jjrow
                       jm3 = jm2 + jjrow -1
                       jm4 = jm3 + jjrow -2
                       jm5 = jm4 + jjrow -3
cdir$   ivdep
         do 115 k = 1, jjrow -ic1+1
            km1 = k -1
115            a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
   +                          -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
   +                          -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
        go to 150
106                    jm2 = jm1 + jjrow
                       jm3 = jm2 + jjrow -1
                       jm4 = jm3 + jjrow -2
                       jm5 = jm4 + jjrow -3
                       jm6 = jm5 + jjrow -4
cdir$   ivdep
         do 116 k = 1, jjrow -ic1+1
             km1= k -1
116            a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
   +                          -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
   +                          -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
   +                          -a(jm6)*a(jm6+km1)
        go to 150
107                    jm2 = jm1 + jjrow
                       jm3 = jm2 + jjrow -1
                       jm4 = jm3 + jjrow -2
                       jm5 = jm4 + jjrow -3
                       jm6 = jm5 + jjrow -4
                       jm7 = jm6 + jjrow -5
cdir$   ivdep
         do 117 k = 1, jjrow -ic1+1
            km1 = k -1
117            a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
   +           -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
   +           -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
   +           -a(jm6)*a(jm6+km1)-a(jm7)*a(jm7+km1)
        go to 150
108                    jm2 = jm1 + jjrow
                       jm3 = jm2 + jjrow -1
                       jm4 = jm3 + jjrow -2
                       jm5 = jm4 + jjrow -3
                       jm6 = jm5 + jjrow -4
                       jm7 = jm6 + jjrow -5
                       jm8 = jm7 + jjrow -6
cdir$   ivdep
         do 118 k = 1, jjrow -ic1+1
            km1 = k -1
118            a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
   +                      -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
   +                      -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
   +                      -a(jm6)*a(jm6+km1)-a(jm7)*a(jm7+km1)
   +                      -a(jm8)*a(jm8+km1)
150     ops = ops + itop*(jjrow -ic1+2)*2
      ll = 1
          idiv = 1
        if (icolp.le.ll) then
          ll  =icolp
          idiv1=1
       else
          idiv1=icolp-ll+1
       endif
          jtop = ic1
          jbot = ic1-itop+1
        do 10 l = 1,  ll
          jtop = jtop - itop
          jbot = jbot - 9*idiv1
          itop = 9*idiv1
          idiv1 = idiv
         if (l.eq.ll) then
          icopy = i - 1
         else
          icopy = i -jbot +ibot-1
         endif
        if(isfull(x(icopy))) go to 332
      Copy  x(icopy)  into  temp
c.....unroll to level 9: fast vector saxpy operations......
332         do 200 j = jtop, jbot, -9
            jj1 = i-j
            jjrow = irowl(jj1)
            jm1 = maxa(jj1) + j
            jm2 = jm1 + jjrow
            jm3 = jm2 + jjrow -1
            jm4 = jm3 + jjrow -2
            jm5 = jm4 + jjrow -3
            jm6 = jm5 + jjrow -4
            jm7 = jm6 + jjrow -5
            jm8 = jm7 + jjrow -6
            jm9 = jm8 + jjrow -7
        if(iif.eq.2) then
         if (a(jm9).ne.0.0) then
cdir$   ivdep
        do 300 k = 1, irowl(jj1) -j+1
          km1 = k -1
300          a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
   +         -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
   +               -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
   +               -a(jm6)*a(jm6+km1)-a(jm7)*a(jm7+km1)
   +               -a(jm8)*a(jm8+km1)-a(jm9)*a(jm9+km1)
```

```
        ops = ops + 18*(irowl(jj1)-j+1)
       else
             if(a(jm4).ne.0.0) then
         go to 301
        else
           if((a(jm1).eq.0.0).and.(a(jm2).eq.0.0).and.
     +        (a(jm3).eq.0.0)) go to 302
       endif
cdir$  ivdep
301     do 310 k = 1, irowl(jj1) -j +1
          km1 = k -1
310           a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
     +              -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
     +              -a(jm4)*a(jm4+km1)
        ops = ops + 8*(irowl(jj1)-j+1)
302     if((a(jm5).eq.0.0).and.(a(jm6).eq.0.0).and.
     .     (a(jm7).eq.0.0).and.(a(jm8).eq.0.0)) go to 200
cdir$  ivdep
       do 320 k = 1, irowl(jj1) -j +1
          km1 = k -1
320           a(im1+km1) = a(im1+km1)-a(jm5)*a(jm5+km1)
     +              -a(jm6)*a(jm6+km1)-a(jm7)*a(jm7+km1)
     +              -a(jm8)*a(jm8+km1)
        ops = ops + 8*(irowl(jj1)-j+1)
       endif
      else
cdir$  ivdep
       do 330 k = 1, irowl(jj1) - j +1
          km1 = k -1
330           a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
     +              -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
     +              -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
     +              -a(jm6)*a(jm6+km1)-a(jm7)*a(jm7+km1)
     +              -a(jm8)*a(jm8+km1)-a(jm9)*a(jm9+km1)
        ops = ops + 18*(irowl(jj1)-j+1)
      endif
200    continue
10     continue
       ll=i-1
       if (isfull(x(ll))) go to 333
    Copy x(ll) into temp
c.....................................................................
333         go to (201,202,203,204,205,206,207,208) ibot-1
       go to 250
201                jjrow = irowl(i-1)
                   jm1 = maxa(i-1) +1
cdir$  ivdep
         do 211 k= 1, jjrow
           km1 = k-1
211                a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1
+km1)
       go to 250
202                jjrow = irowl(i-2)
                   jm1 = maxa(i-2) +2
                   jm2 = jm1 + jjrow
cdir$  ivdep
           do 212 k = 1, jjrow -1
                              km1 = k -1
212          a(im1+km1)=a(im1+km1)-a(jm1)*a(jm1+km1)
     +                          -a(jm2)*a(jm2+km1)
       go to 250
203                jjrow = irowl(i-3)
                   jm1 = maxa(i-3) + 3
                   jm2 = jm1 + jjrow
                   jm3 = jm2 + jjrow -1
cdir$  ivdep
           do 213 k = 1, jjrow -2
                              km1 = k - 1
213          a(im1+km1)=a(im1+km1)-a(jm1)*a(jm1+km1)
     +            -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
       go to 250
204                jjrow = irowl(i-4)
                   jm1 = maxa(i-4) + 4
                   jm2 = jm1 + jjrow
                   jm3 = jm2 + jjrow -1
                   jm4 = jm3 + jjrow -2
cdir$  ivdep
           do 214 k = 1,jjrow  -3
                   km1 = k -1
214          a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
     +            -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
     +            -a(jm4)*a(jm4+km1)
       go to 250
205                jjrow = irowl(i-5)
                   jm1 = maxa(i-5) + 5
                   jm2 = jm1 + jjrow
                   jm3 = jm2 + jjrow -1
                   jm4 = jm3 + jjrow -2
                   jm5 = jm4 + jjrow -3
cdir$  ivdep
           do 215 k = 1, jjrow -4
                   km1 = k -1
215          a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
     +            -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
     +            -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
       go to 250
206                jjrow = irowl(i-6)
                   jm1 = maxa(i-6) +6
                   jm2 = jm1 + jjrow
                   jm3 = jm2 + jjrow -1
                   jm4 = jm3 + jjrow -2
                   jm5 = jm4 + jjrow -3
                   jm6 = jm5 + jjrow -4
cdir$  ivdep
           do 216 k = 1, jjrow -5
                   km1 = k -1
216          a(im1+km1) = a(im1+km1)-a(jm1)*a(jm1+km1)
     +            -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
     +            -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
     +            -a(jm6)*a(jm6+km1)
       go to 250
207                jjrow = irowl(i-7)
                   jm1 = maxa(i-7)+7
                   jm2 = jm1 + jjrow
                   jm3 = jm2 + jjrow -1
                   jm4 = jm3 + jjrow -2
                   jm5 = jm4 + jjrow -3
                   jm6 = jm5 + jjrow -4
                   jm7 = jm6 + jjrow -5
cdir$  ivdep
           do 217 k = 1, jjrow -6
                   km1 = k -1
217          a(im1+km1)=a(im1+km1)-a(jm1)*a(jm1+km1)
     +            -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
     +            -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
     +            -a(jm6)*a(jm6+km1)-a(jm7)*a(jm7+km1)
       go to 250
208      jjrow =irowl(i-8)
                   jm1 = maxa(i-8) + 8
         jm2 = jm1 + jjrow
         jm3 = jm2 + jjrow -1
         jm4 = jm3 + jjrow -2
         jm5 = jm4 + jjrow -3
                   jm6 = jm5 + jjrow -4
```

```
          jm7 = jm6 + jjrow -5
          jm8 = jm7 + jjrow -6
cdir$   ivdep
           do 218 k = 1, jjrow -7
                    km1 = k -1
218             a(im1+km1)=a(im1+km1)- a(jm1)*a(jm1+km1)
   +               -a(jm2)*a(jm2+km1)-a(jm3)*a(jm3+km1)
   +               -a(jm4)*a(jm4+km1)-a(jm5)*a(jm5+km1)
   +               -a(jm6)*a(jm6+km1)-a(jm7)*a(jm7+km1)
   +              -a(jm8)*a(jm8+km1)
250       ops = ops + 2*(ibot-1)*(jjrow -ibot +2)
          a(im1) =sqrt(a(im1))
          xinv = 1.0/a(im1)
cdir$   ivdep
       do 260  k = 1, irowl(i)
260         a(im1+k) = xinv *a(im1+k)
        ops = ops + irowl(i) +2
       Produce x(i) = a(im1)
100  End  presched do
       els e
c.....forward reduction- unroll to level 3 for fast vector speed:
c.....each 3 rows of [k] must end in the same column number..
     Barrier
        ops = 0
        ibot = neq -3* (neq/3)
       do 510 i = 1,neq-ibot,3
        im1 = maxa(i)
        im2 = maxa(i+1)
        im3 = maxa(i+2)
        xmult1 = b(i)/a(im1)
         xmult2 = (b(i+1) - xmult1*a(im1+1))/a(im2)
         xmult3 = (b(i+2) - xmult1*a(im1+2)
   +                  - xmult2*a(im2+1))/a(im3)
        b(i) =   xmult1
        b(i+1) = xmult2
        b(i+2) = xmult3
cdir$   ivdep
         do 520 j = i+3, i+irowl(i)
520           b(j) = b(j) - xmult1*a(im1+j-i)
   +                  - xmult2*a(im2+j-i-1)
   +              - xmult3*a(im3+j-i-2)
510         ops = ops + 6*(irowl(i)-2)+ 9
        if ( ibot.eq.1) then
         b(neq) = b(neq)/a(maxa(neq))
         ops = ops + 1
       else
        if (ibot.eq.2) then
        im1 = neq -1
         b(im1) =  b(im1)/a(maxa(im1))
         b(neq) = (b(neq) -b(im1)*
   + a(maxa(im1)+1))/a(maxa(neq))
         ops = ops + 4
       endif
       endif
c........back substitution with vector unrolling follows...
           b(neq) = b(neq)/a(maxa(neq))
           ops = ops +1
           jm1 = neq -1
     if (ibot .eq. 2) then
           im1 = neq -1
           b(im1)=(b(im1)-
a(maxa(im1)+1)*b(neq))/a(maxa(im1))
           ops = ops + 3
           jm1 = neq -2
     endif
      if (ibot .eq. 0) then
           im1 = neq -1
           b(im1)=(b(im1)-a(maxa(im1)+1)*b(neq))/a(maxa(im1))
           im2 = neq -2
           b(im2)  =(b(im2)-a(maxa(im2)+1)*b(im1)
   +          -a(maxa(im2)+2)*b(neq))/a(maxa(im2))
           ops = ops + 8
           jm1 = neq -3
     endif
      do 1010 i = jm1,1,-3
        im1 = maxa(i)
        im2 = maxa(i-1)
        im3 = maxa(i-2)
        xmult1 = 0.0
        xmult2 = 0.0
        xmult3 = 0.0
cdir$   ivdep
        do 1020 j=i+1, irowl(i)+i
           xmult1 = xmult1 + a(im1+j-i)*b(j)
           xmult2 = xmult2 + a(im2+j-i+1)*b(j)
1020       xmult3 = xmult3 + a(im3+j-i+2)*b(j)
         b(i) = (b(i) - xmult1)/a(im1)
          b(i-1) = (b(i-1) - a(im2+1)*b(i) - xmult2)/a(im2)
          b(i-2) = (b(i-2)-a(im3+2)*b(i)-a(im3+1)*b(i-1)
   +             -xmult3)/a(im3)
1010         ops = ops + 6*(irowl(i)) +12
        End  barrier
           endif
           return
           end
      subroutine NORM(irow,icoln,x,neq,nc)
       dimension irow(*),icoln(*),x(*),b(neq),diag(neq),offdia(nc)
c.....get error error norm: [a]*{x}={b}: read file COEFS.COLM
c..... ([xqt iter with reset sipr=-2 in CSM Testbed) where:
c.....nc=number of nonzero, off-diagonal terms of [k]
c.....irow(neq)=no. of nonzeros in each row w/o diagonal
c.....icoln(nc)=column no. of nonzero terms of [k] by row
c.....diag(neq)=diagonal terms of [k], b(neq)=load vector
c.....offdia(nc)=nonzero, offdiagonal terms of [k]
           rewind(8)
           read(8)  neq,neq2,nc,nc2,jdof,jt,ndof
           read(8) (irow(i) ,i= 1 , neq)
           read(8) (icoln(i), i = 1 , nc)
           read(8)( diag(i), i = 1 , neq )
           read(8)( offdia(i), i = 1, nc )
           read(8)( b(i), i = 1 , neq )
           icount = 0
      do 1 i = 1 , neq
1          diag(i) = diag(i) * x(i)
      do 2 i = 1 , neq - 1
           nonz = irow(i)
      do 2 j = 1 , nonz
           icount = icount + 1
           locate=  icoln(icount)
           diag(i) = diag(i) + offdia(icount)*x(locate)
2          diag(locate)=diag(locate)+offdia(icount)*x(i)
           enorm = 0.0
           fnorm = 0.0
           snorm = 0.0
      do 3 i = 1 , neq
           diag(i) = diag(i) - b(i)
           enorm = enorm + diag(i) * diag(i)
           fnorm = fnorm + b(i)*b(i)
3          snorm = snorm + diag(i)*x(i)
           write(*,*)'* ABSOLUTE error norm = ',sqrt(enorm)
           relerr = sqrt(enorm/fnorm)
           write(*,*) '* RELATIVE to load = ',relerr
           write(*,*) '* STRAIN ENERGY error norm = ',snorm
           return
```

```
            end
       subroutine CSMIN(a,b,maxa,irowl,icolh,neq,nterms,
     +        irow,icoln,nc,maxbw,iin,locrow,iavebw)
        dimension a(*),b(*),maxa(*),irowl(*),icolh(*),irow(*),icoln(*)
c.....read binary file COEFS.COLM output by iter(sipr=-2)...
        open(unit=8,file='COEFS.COLM',form='unformatted',
     +       access='sequential',status='old')
       read(iin) neq,neq2,nc,nc2,jdof,jt,ndof
      read(iin) (irow(i), i = 1,neq)
       read(iin) (icoln(i), i=1,nc)
c.....initialize column heights.....................................
      loop =9
      do 100 i = 1, neq
100       icolh(i) = 0.0
      icount = 1
      do 110 i = 1, neq-1
         do 110 j =1, irow(i)
            jcol = icoln(icount)
            nowht = jcol - i
            if (nowht.gt.icolh(jcol)) icolh(jcol)=nowht
110         icount = icount+1
c.....find the row-lengths.................................
       iseg1 = loop*neq/loop
      jcount = 0
      icount = 1
       do 120 i = 1, iseg1, loop
         jcount = jcount + irow(i)
           if (icoln(jcount).gt.icount) icount=icoln(jcount)
        do 130 j = i+1, i+loop-1
          jcount =jcount + irow(j)
130         if (icoln(jcount).gt.icount) icount=icoln(jcount)
        do 140 j = i,i+loop-1
140         irowl(j) = icount - j
120      continue
      do 150 i = iseg1+1,neq
150      irowl(i) = neq - i
c.....locate diagonal elements in vector {a}.................
      maxa(1) = 1
      do 160 i =1, neq
160       maxa(i+1) = maxa(i) + irowl(i) +1
      icount = 1
      do 170 i = 1, neq-1
         do 170 j = 1, irow(i)
            jcol = icoln(icount)
            locate = maxa(i) +jcol - i
            icoln(icount) = locate
170         icount =icount +1
      nterms = maxa(neq+1) - 1
      do 180 i = 1, nterms
180       a(i) = 0.0
       read(iin)  (a(maxa(i)), i=1,neq)
        read  (iin)  (a(icoln(i)),i=1,nc)
       read( iin)  (b(i), i=1,neq)
c.....find maximum and average bandwidths.............
      maxbw = 0
      iavebw = 0
       do 190 i = 1, iseg1, loop
         if (irowl(i) .gt. maxbw) then
           maxbw = irowl(i)
           locrow = i
         endif
190        iavebw = iavebw + loop*irowl(i) - (loop)*(loop-1)/2
      do 200 i = iseg1+1,neq
200       iavebw = iavebw + irowl(i)
      iavebw = iavebw/(neq+1)
      maxbw =maxbw + 1
      return
      end
       subroutine TOCSM(x,irowl,icoln,b,u,irtoj,iin,nat)
       dimension  irowl(*),icoln(*),b(*),u(*),x(*),irtoj(*)
       character*40  libnam
        common /constr/jt,jdf,jddf,inex(6),mexin(6),ksym(3),q,qq
c            convert static displacements calculated by pvsolve
c            to csm testbed joint reference frame for [k]{u}={f}
c            assume each node has 6 degrees-of-freedom ( i.e.,
c            u(14) is the 2nd dof of node #3) and
c            jdof = number of joints * number of dof per joint
        read '(a)',libnam
        nu  =  lmopen('old',0,libnam,0,1000)
        call  dal(nu,11,jt,18,-1,lseq,ierr,nwds,ne,lb,ityp,
     + 4hJDF1,4hBTAB,1,8)
c......read COEFS.COLM as in subroutine NORM.............
       rewind iin
         read(iin)  n,n,nc,nc,jdof,jt,ndof
         if(nat.ge.2*jdof.and.nat.ge.ncoef)  then
        read (iin) (irowl(i),i=1,n)
        read(iin) (icoln(i),i=1,nc)
        read(iin) (b(i),i=1,n)
        read(iin) (b(i),i=1,nc)
        read(iin) (b(i),i=1,jdof)
c......COEFS.COLM stores joint-to-row before row-to-joint.
c......only row-to-joint info. needed, so storage reused...
        read(iin) (jtorj(i),i=1,2*jdof)
       else
        write(*,*) 'error in TOCSM: insufficient memory'
       endif
c......initialize joint displacement.....................
          do 4 i=1,jdof
4            u(i)=0.
          do 1 i=jdof+1,jdof+n
             locate = irtoj(i)
1            u(locate) = x(i-jdof)
c.....put prescribed displacements in vector {u}.....
          do 2 i = jdof+n+1,2*jdof
                if(irtoj(i).ne.0) then
                locate = irtoj(i)
                u(locate)= b(i-jdof)
               endif
2       continue
c.....write displacements for first 3 joint locations
        njoint = jdof/6
        do 3 i=1,3
             i1 = (i-1)*6 + 1
             i2 = i*6
3         write(6,5) i,(u(j),j=i1,i2)
5         format('jt',i5,' disp=',6e11.3)
c.....put displacements in csm testbed library file
c        'libnam' (load set 1, constraint set 1)
             iset = 1
             ncon = 1
             nrhs = 1
             nwds = jdof*nrhs
          call gmsign('PVSOLVE')
          call  dal(nu,0,0,0,1,lseq,ierr,nwds,jt,jdf,-1,
     + 4hSTAT,4hDISP,iset,ncon)
          call  rio(nu,1,2,lseq,1,nrhs,u(1),nwds,-1,jt)
          call  gmclos(nu,0,9999)
             return
             end
```

The command file to compute static displacements for the research aircraft and space shuttle SRB on the Cray Y-MP using 1 to 8 processors follows. The first statements specify the UNIX C-shell is used and the maximum number of processors

(NCPUS) that may be requested is 8. The stiffness matrix data (COEFS.COLM) and program (**pvsolve**) are then copied to the solid state disk ($WRKDIR). Using the hardware performance monitor, hpm, to count operations, times and MFLOPS, the displacements for the aircraft and SRB are then calculated by **pvsolve** on 8,4,2 and 1 processors. The results are appended to the file 'out' which, upon completion, is copied to the home directory:

```
#!/bin/sh
NCPUS=8
export NCPUS
cd $WRKDIR
date >out
cp /u/ra/storaasl/nasp/COEFS.COLM .
cp /u/ra/storaasl/srb/pvsolve .
date >>out
ja
hpm -g0 -d forcerun pvsolve 8 >>out 2>&1
hpm -g0 -d forcerun pvsolve 4 >>out 2>&1
hpm -g0 -d forcerun pvsolve 2 >>out 2>&1
hpm -g0 -d forcerun pvsolve 1 >>out 2>&1
ja -cstfl
date >>out
cp /scr5/storaasl/srb/COEFS.COLM .
date >>out
hpm -g0 -d forcerun pvsolvesrb 8 >>out 2>&1
hpm -g0 -d forcerun pvsolvesrb 4 >>out 2>&1
hpm -g0 -d forcerun pvsolvesrb 2 >>out 2>&1
hpm -g0 -d forcerun pvsolvesrb 1 >>out 2>&1
date >>out
cp out $HOME
```

Pvsolve is run in the CSM Testbed[18] structural analysis software to compute the static displacements for the SRB using the *spawn command in the following runstream using four Cray Y-MP processors:

```
testbed
*open 1 srb.l01
[xqt iter
reset sipr = -2
stop
*close 1
*spawn pvsolve srb.l01 4
*open 1 srb.l01 /old
[xqt vprt
PRINT STAT DISP
[xqt gsf
[xqt psf
[xqt exit
```

The 'iter' reset option bypasses the lengthy solution process and just formats the data for **pvsolve**. **Pvsolve** computes the static displacements and writes them to the data set STAT.DISP.1.1 in the CSM Testbed library srb.l01. The stresses are then calculated and printed based on the displacements calculated by **pvsolve**. The pvsolve code above is compiled using **force** producing the executable file, pvs. The pvsolve in the *spawn command is the following script that resides in the directory containing the CSM Testbed executable files:

```
forcerun pvs $2 <<EOF
$1
EOF
```

9. References

[1]Mackintosh, A. R., "The First Electronic Computer", *Physics Today*, March 1987, pp. 25-32.

[2]Ortega, J. M., *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Publishing Corporation, New Jersey, 1988.

[3]Utku, S., Salama, M., and Melosh, R., "Concurrent Factorization of Positive Definite Banded Hermitian Matrices", *International Journal of Numerical Methods in Engineering*, Vol. 23, 1986, pp. 2137-2152.

[4]Farhat, C., Wilson, E., and Powell, G., "Solution of Finite Element Systems on Concurrent Processing Computers", *Engineering Computing*, Vol. 2, 1987, pp. 157-165.

[5]Chen, S., Dongarra, J., and Hsiung, C., "Multiprocessing Linear Algebra Algorithms on the Cray XMP-2: Experiences With Small Granularity", *Journal of Parallel Distributed Computing*, Vol. 1, 1984, pp. 22-31.

[6]Dongarra, J. J., Gustafson, F. G., and Karp, A., "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine", *SIAM Review*, Vol. 26, No. 1, January, 1984.

[7]Ashcraft, C. C., Grimes, R. G., Lewis, J. G., Peyton, B. W., and Simon, H. D., "Progress in Sparse Matrix Methods for Large Linear Systems on Vector Supercomputers", *The International Journal of Supercomputer Applications*, Vol. 1, No. 4, Winter 1987, pp. 10-30.

[8]Poole, E. L., and Overman, A. L., "The Solution of Linear Systems of Equations with a Structural Analysis Code on the NAS Cray 2", NASA CR 4159, Dec. 1988.

[9]Storaasli, O. O., and Bergan, P. G., "Nonlinear Substructuring Method for Concurrent Processing Computers", *AIAA Journal*, Vol. 25, No. 6, June 1987, pp. 871-876.

[10]Law, K., "A Parallel Finite Element Solution Method", *Computers and Structures*, Vol. 23, No. 6, 1986, pp. 845-858.

[11]Farhat, C. , and Wilson, E. L., "A Parallel Active Column Equation Solver", *Computers and Structures*, Vol. 28, 1988, pp. 289-304.

[12]Storaasli, O. O., Poole, E. L., Ortega, J. M., Cleary, A., and Vaughan, C., "Solution of Structural Analysis Problems on a Parallel Computer", *Proceedings of the AIAA/ASME/ASCE/AHS 29th Structures, Structural*

Dynamics and Materials Conference, Williamsburg, VA, April 18-20, 1988, pp. 596-605, AIAA Paper No. 88-2287.

13Storaasli, O. O., Bostic, S. W., Patrick, M., Mahajan, U., and Ma, S., "Three Parallel Computation Methods for Structural Vibration Analysis", *Proceedings of the AIAA/ ASME/ASCE/AHS 29th Structures, Structural Dynamics and Materials Conference*, Williamsburg, VA, Apr. 18-20, 1988, pp. 1401-1411, AIAA Paper No. 88-2391.

14Nguyen, D. T., Shim, J. S., and Zhang, Y., "The Component-Mode Method in a Parallel Computer Environment", *Proceedings of the AIAA/ASME/ASCE/AHS 29th Structures, Structural Dynamics and Materials Conference*, Williamsburg, VA, April 18-20, 1988, pp. 1705-1710, AIAA Paper No. 88-2438.

15Nguyen, D. T., and Niu, K. T., "A Parallel Algorithm for Structural Sensitivity Analysis on the FLEX/32 Multicomputer", *Proceedings of the 6th ASCE Structures Congress*, Orlando, FL, August 17-20, 1987, pp. 98-112.

16Storaasli, O. O., Nguyen, D. T., and Agarwal, T. K., "The Parallel Solution of Large-Scale Structural Analysis Problems on Supercomputers", *Proceedings of the AIAA/ASME/ASCE/AHS 30th Structures, Structural Dynamics and Materials Conference*, Mobile AL, April 3-5, 1989, pp. 859-867. Paper No. 89-1259 (to appear in *AIAA Journal*, Sept. 1990)

17Jordan, H. F., Benten, M. S., Arenstorf, N. S., and Ramann, A. V., "Force User's Manual: A Portable Parallel FORTRAN", NASA CR 4265, January, 1990.

18Stewart, C. B.(compiler), "The Computational Structural Mechanics Testbed User's Manual", NASA TM-100644, October 1989.

19George, A. and W-H Liu, J., *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1981.

20Bathe, K. J., *Finite Element Procedures in Engineering Analysis*, Prentice Hall, Inc., New York, 1982.

21Robins, W. A. et al., "Concept Development of a Mach 3.0 High-Speed Civil Transport", NASA TM 4058, Sept. 1988.

22Knight, N. F., McCleary, S. L., Macy, S. C., and Aminpour, M. A., "Large Scale Structural Analysis: The Structural Analyst, The CSM Testbed, and The NAS System", NASA TM-100643, March 1989.

23Knight, N. F., Gillian, R. E., and Nemeth, M. P., "Preliminary 2-D Shell Analysis of the Space Shuttle Solid Rocket Boosters", NASA TM-100515, 1987.

24Simon, H., Vu, P. and Yang, C., "Performance of a Supernodal General Sparse Solver on the Cray Y-MP: 1.68 GFLOPS with Autotasking", Scientific and Computing Analysis Division Report SCA-TR-117, Boeing Computer Services, Seattle, WA, March, 1989.

25Storaasli, O., Nguyen, D., and Agarwal, T., "Force on the Cray Y-MP", */u/nas/news The Numerical Aerodynamic Simulation Program Newsletter*, NASA Ames Research Center, Vol. 4, No. 7, July 1989, pp. 1-4.

26Storaasli, O. O., "New Equation Solver for Supercomputers", */u/nas/news The Numerical Aerodynamic Simulation Program Newsletter*, NASA Ames Research Center, Vol. 5, No. 1, January 1990, pp. 1-3.