

Sparse Matrix Methods in MSC/NASTRAN

Shawn Shamsian and Louis Komzsik

The MacNeal-Schwendler Corporation
815 Colorado Blvd.
Los Angeles, Ca

Abstract:

In the past years we have been investigating the feasibility of new sparse matrix methods in MSC/NASTRAN. After careful studies we launched a two year, two phase development plan. The first phase, which recently completed for version 67, entails incorporating sparse matrix algorithms into the MPYAD, FBS, and DECOMP modules. This paper describes these new algorithms and presents the results obtained on experimental version 67 systems.

1.0 Introduction

Numerically intensive operations such as matrix decomposition, forward and backward substitutions, and matrix multiplication are the main sources of CPU usage in the solution of large linear equation systems. The large order matrices involved in such operations, which represent the physical properties of the system, are often of very low density, where the density is defined by:

$$\rho = \frac{w}{t} * 100\%$$

ρ is the percent density, w is the number of non-zero terms, and t is the total number of terms in the matrix. Matrix densities of well below 1% are common to occur in the solution of larger size FEM problems.

It is apparent that if algorithms used in matrix operations on such matrices make no distinction between zero and non-zero terms of the matrix, then most of the CPU time will be wastefully spent on zero operations. On the other hand, such straightforward algorithms can always achieve a high number of floating point operations per second (FLOPS) because of their simplistic approach. So, the challenge presented to the algorithm designer is to simultaneously:

1. Reduce the number of floating point operations, and
2. Keep the number of FLOPS at a high level.

In order to achieve these goals, many software changes had to be made, among which include:

1. New data structures to represent the matrices,
2. New data access schemes,
3. New numerical kernels which work on these data structures, and
4. New matrix algorithms.

In the following sections, a brief description of the above topics is given. Then the MSC/NASTRAN Version 67 preliminary results are presented.

2.0 Data Structures

The data structure historically used for matrix representation in MSC/NASTRAN is based on strings, where strings are referred to the contiguous non-zero terms in a column. In this representation the data structure of each column consists of the following elements:

1. A column header which marks the start of a new column and identifies the column number
2. Strings, where each string in turn consists of:
 - (a) A string header which marks the start of a new string and contains the length of the string and also the row number corresponding to the first element of the string
 - (b) A segment containing the numeric terms of the string
 - (c) An optional string trailer which marks the end of the string and contains the length of the string and also the row number corresponding to the last element of the string

In this representation the numeric terms of the matrix and the topological information concerning the location of these terms are interspersed. In addition, in many cases the average number of terms in a string for a particular matrix is far below the desired vector length in a vector operation.

In order to make the data structure more suitable for efficient vector operations, we are considering the following basic structure for possible implementation in phase 2:

1. A column header which marks the start of a new column and identifies column number and the number of non-zero terms in the column
2. A contiguous record containing the row numbers at which the non-zero terms appear in that column (in ascending order)
3. A contiguous record containing the non-zero terms in the column (corresponding to the indices in the previous record)

This representation allows for vectorization without the need to store redundant zero terms. However, the storage requirement for this representation is almost always higher than that in the previous method. Storage requirement for the old method can be predicted by

$$S1 = n * 2 + s * 2 + w * p$$

where $S1$ is the total number of words required for the matrix storage, n is the number of columns, s is the number of strings, w is the total number of non-zero terms in the matrix, and p is the number of words per each matrix term. The new scheme requires:

$$S2 = n * 2 + w + w * p$$

And if we approximate s by (w/l) , where l is the average string length of the matrix, then in order to have a lower storage requirement in the sparse scheme than the previous storage method we must have:

$$w < \frac{w}{l} * 2$$

or

$$l < 2$$

This means that the storage requirement of the sparse method will always be higher if the average string length of the matrix exceeds 2 words.

The results of the phase 2 studies will indicate whether separate representations are required for matrices of high and low densities or whether the sparse representation is practical in all cases.

3.0 Data access schemes

The traditional schemes of accessing matrices in MSC/NASTRAN are:

1. Column based, where the column can be identified by:
 - (a) A pointer to the beginning of the numeric terms in the column

- (b) The length of the column
 - (c) The record containing the numeric terms of the column including the zero terms
2. String based; the strings are identified as follows:
- (a) A pointer to the start of the numeric terms in the strings
 - (b) The number of terms in the string
 - (c) The row number corresponding to the first term in the string
 - (d) The record containing the numeric terms of the string
3. Band based, where the band is the portion of the column between the first and the last non-zero terms, including the zero terms that fall in between. The band is identified by:
- (a) A pointer to the beginning of the band
 - (b) The length of the band
 - (c) The row number corresponding to the first term in the band
 - (d) The record containing the numeric terms of the band including the zero terms

The first scheme is ideal for dense matrices and lends itself to easy vectorization. But when dealing with sparse matrices, it can create a large overhead due to the redundant operations involved. The string-based scheme which is used to eliminate redundant operations, is not, however, optimal for vectorization when the strings are short. The third solution, the band scheme, can be successfully used on matrices with low average bandwidths relative to their size. It is suitable for vectorization but shares the same problems as the first scheme in dealing with general sparse matrices, where the average bandwidth is relatively large and the number of non-zero terms within the band is relatively small.

The scheme suggested for accessing a sparse matrix column is as follows:

1. A pointer to the record containing the ordered set of row numbers at which the non-zero terms appear

2. A pointer to the record containing the numeric terms corresponding to the row numbers above
3. The number of non-zero terms in the column
4. The record containing the ordered set of row numbers
5. The record containing the non-zero terms of the column

This scheme can both help eliminate the redundant operations and facilitate the vectorization. The only disadvantage here is the use of indirect addressing which is later discussed in more detail.

4.0 Numerical Kernels

To assure good performance on a wide variety of computers, the internal loops of the numerical modules are executed by a few numerical kernels. The most frequently used kernels in MSC/NASTRAN are the DOT and the simple AXPY kernels which work with the traditional data structures discussed in the previous sections.

The function of the DOT kernel could be mathematically described as follows:

$$SUM = \sum_{i=1}^n X(i) * Y(i)$$

where X and Y are arrays of size n . When dealing with sparse matrices, however, a relatively large number of zero terms may exist in arrays X or Y resulting in a large number of redundant zero operations. The DOTI kernel which is the sparse equivalent to the DOT kernel is designed to reduce such redundant operations. DOTI can be described by:

$$SUM = \sum_{j=1}^m X(INDX(j)) * YC(j)$$

In this case X is an array of size n , YC is a compressed array of size m , $m \leq n$ containing only the non-zero terms in Y , and $INDX$ is an integer array containing the row numbers corresponding to the non-zero terms in Y .

However, the nested indexing of the array X , which is called indirect addressing, is less efficient than the direct addressing used in the implementation of the DOT kernel. This implies that the DOT kernel is always more efficient when operating on very dense arrays, and that the DOTI kernel is more efficient when operating on very sparse arrays, and that somewhere in the middle the DOTI kernel is equal in efficiency to the DOT kernel.

Tests that we have conducted on several machine architectures indicate the density break even point to be around 35 to 50 percent. This is very encouraging since as previously mentioned the density of the matrices commonly encountered in FEM is far below this point.

The mathematical description for the AXPY kernel is as follows:

$$Y(i) = s * X(i) + Y(i) \quad i = 1, 2, \dots, n$$

where X and Y are arrays of length n , and s is the scalar multiplier. The sparse equivalent to this kernel, AXPI, can be described by:

$$Y(INDX(j)) = s * XC(j) + Y(INDX(j)) \quad j = 1, 2, \dots, m$$

where XC is a compressed array of length m , $m \leq n$ holding the non-zero terms in X , and $Y, INDX, s$, and n are as described above.

The density break even point for these two kernels was found to be the same as that for the DOT and DOTI kernels. However, one obvious advantage in using the AXPI kernel over the DOTI kernel is that, if it is assured that the scalar multiplier is always non-zero, then no zero multiply operations will ever occur in the execution of the AXPI kernel. The use of the DOTI kernel, although may greatly reduce the number of zero multiply operations, will never eliminate the possibility of such redundant operations.

5.0 Sparse Algorithms

In this section, the three sparse algorithms incorporated so far in MSC/NASTRAN version 67 are briefly described.

5.1 Sparse MPYAD

The sparse multiply-add method which performs the operation

$$AB \pm C$$

or

$$A^T B \pm C$$

is algorithmically similar to the present MPYAD method. In the transpose case, the matrix A is first transposed and written to a scratch file. This step should not be very expensive since A is relatively sparse when this method is selected.

The difference is that A is stored in a new sparse form. In particular, all the non-zero terms of a column are stored in a contiguous real memory region, and the row indices are in a separate integer array. With this configuration the sparse kernel AXPI can be used with vector lengths of size NZC (number of non-zero terms in a column) rather than being limited to the string length.

Storage requirements for this method consist of:

- The area to store one column of the result matrix
- The area to store numeric terms in A (at least for one column)
- The area to store row indices of A (at least for one column)
- The area to store pointers to the beginnings of numeric terms and row indices of each column (at least for one column)

5.2 Sparse DECOMP

The input matrix A is first read in the preface stage to create the following sparsity information:

- One integer vector of length NZA (where NZA is the number of non-zero terms of the upper half of the matrix A) containing the column indices
- One integer vector of length NZA containing the row indices
- One real vector of length NZA containing the actual non-zero terms

Also at this stage, the empty (null) rows and columns of the matrix are eliminated.

The actual elimination process could be executed in different sequences. The performance of each sequence is obviously different. To find an effective elimination sequence, a symbolic elimination is also performed during the preface. An important criterion, at this stage, is to minimize the fill in created in each step of the elimination process, thereby reducing the numerical work and the I/O requirement.

The minimum degree pivoting algorithm chooses pivots from the diagonal of the current reduced matrix A in the row with the least number of non-zeros. The non-zeros in that row represent all the distinct variables that connect the corresponding variable. A graph approach is used, a diagonal term is regarded as a node, while an off-diagonal term is represented as an edge between two nodes.

The elimination of a chosen variable (node) is performed after severing all the edges connected to it. This leaves a reduced graph where the same process can continue until the reduced graph is a single node only. Then, using this term as the root, we can create a so-called "minimum degree assembly tree" of the matrix. The final pivoting sequence, then, is obtained by searching the assembly tree. Note that this sequence may be changed due to numerical reasons. So the final numerical decomposition is achieved by partitioning and permutation of the matrix A as

$$PAP = \begin{bmatrix} E & C^T \\ C & B \end{bmatrix}$$

to obey the pre-calculated elimination order and to assure that the inverse

of the s by s E submatrix exists. Then the decomposition algorithm is in the form of:

$$PAP = \begin{bmatrix} I_s & 0 \\ CE^{-1} & I_{n-s} \end{bmatrix} \begin{bmatrix} E & 0 \\ 0 & B - CE^{-1}C^T \end{bmatrix} \begin{bmatrix} I_s & E^{-1}C^T \\ 0 & I_{n-s} \end{bmatrix}$$

The numerical work is mainly executed by the vector kernels. Besides the conventional DOT and AXPY kernels, the sparse DOTI and AXPI kernels are also used.

5.3 Sparse FBS

The sparse option of FBS executes the forward-backward substitution from the factor of the sparse decomposition. Thereby, we must consider the permutations executed; i.e., we have to solve the

$$LDL^T PX = PB$$

matrix equation. The usual forward pass solves

$$LY = PB$$

for Y . The main kernel that is used for the numerical work at this stage is the sparse AXPI kernel. The backward pass gives X from

$$DL^T(PX) = Y$$

The sparse dot product kernel DOTI is the main kernel at this point.

The storage requirement consists of a real area for the right-hand side B , the result X , and an integer vector of length N holding the permutation information. An area is also needed for storing the factor. If the entire factor cannot be held in memory at one time, then a buffer of length MP is reserved, where MP is the maximum number of non-zeros in any pivot row.

6.0 Practical Results

In the following subsections, preliminary results for the sparse methods obtained on experimental version 67 systems are presented.

6.1 MPYAD Results

A series of parametric tests were created to show the performance characteristics of the new algorithm compared to the existing MPYAD methods. The variables in these tests were the string length, which ranged from 1 to 32, and the density of the test matrices, which ranged from 1% to maximum density allowed by their string length. In all the cases presented, A and B are of size 500 X 500, and C is purged.

The tests were run originally on our VAX 6000, which is our development machine, and later on a CONVEX C1 which is a vector machine and has vector implementation of the DOTI, and AXPI kernels. The results obtained are summarized in the following tables:

VAX 6000-420 - MSC/NASTRAN Ver67exp								
average string length								
	best old method				new sparse method			
Density	1	2	8	32	1	2	8	32
1%	5.0	4.9	4.4	3.8	0.9	0.7	0.5	0.5
3%	6.1	5.7	4.7	4.2	1.8	1.3	1.1	0.9
6%	10.1	7.6	5.6	5.0	4.1	2.9	2.2	1.7
12%	22.5	16.0	8.8	6.4	11.7	8.3	6.7	5.5
25%	73.2	49.1	25.2	13.9	36.7	30.9	25.5	15.4
50%	238.0	163.5	86.8	56.4	177.9	144.3	118.2	86.4
67%	N/A	252.6	143.4	107.6	N/A	254.4	202.6	176.2
89%	N/A	N/A	242.0	175.4	N/A	N/A	350.6	297.6
97%	N/A	N/A	N/A	232.0	N/A	N/A	N/A	382.7

Table 1.

CONVEX C1 - MSC/NASTRAN Ver67exp								
average string length								
best old method					new sparse method			
Density	1	2	8	32	1	2	8	32
1%	3.6	2.8	1.8	1.1	1.2	1.0	0.9	0.9
3%	5.7	4.7	2.2	1.6	2.6	1.7	1.2	1.0
6%	8.5	7.6	4.6	2.6	4.0	3.3	1.8	1.6
12%	13.9	11.0	8.3	6.1	7.3	5.4	3.7	3.4
25%	24.6	19.7	13.6	12.2	15.5	12.1	8.2	7.4
50%	44.9	37.0	29.4	24.3	38.9	33.7	26.8	25.7
67%	N/A	46.9	37.5	31.8	N/A	53.0	44.2	42.4
89%	N/A	N/A	47.8	42.6	N/A	N/A	74.7	70.3
97%	N/A	N/A	N/A	47.2	N/A	N/A	N/A	83.9

Table 2.

In the above table, the best CPU time in each category is printed in bold face.

6.2 DECOMP and FBS Results

Six problems belonging to the automobile and aerospace industries were selected. Three were run on a CRAY/YMP and the other three on an IBM 3090. The intent was to compare the performance of the new sparse DECOMP and FBS against the classical active column decomposition and regular FBS.

The characteristics of the problems run on CRAY are shown in the following table:

CRAY Test problems			
Char/Problem	Helicopter	Door	Body
# of grid points	1832	6618	9066
# of elements	4388	6399	8223
# of DOF	5487	34769	47071
Bandwidth	135	677	1083
Nonzero terms	87302	965735	1250813
Factor size	338114	5869137	7668507
Table 3.			

The CRAY/YMP 832 used has a clock cycle of 6.41 nsec. The result are tabulated below:

CRAY/YMP 832 - MSC/NASTRAN Ver67exp				
Solve Meth.	Traditional		Sparse	
TEST/TIME	CPU	I/O	CPU	I/O
Helicopter	53.7	27.6	47.0	19.3
Door	231.1	163.5	140.2	80.0
Body	506.5	156.4	277.3	57.9
Table 4.				

The characteristics of the problems run on the IBM ES/3090 are as follows:

IBM Test problems			
Char/Problem	Shocktower	Trackbar	Seatback
# of grids	4218	4714	5289
# of elements	3901	4328	5177
# of DOF	21926	26649	27276
Bandwidth	456	274	349
Nonzero terms	538786	664606	641584
Factor size	2665062	3468821	2477810

Table 5.

The above problems were run on an IBM ES/3090 with a clock cycle of xxx nsec. The results are presented in the table below:

IBM ES/3090 400S - MSC/NASTRAN Ver67exp				
Solve Meth.	Traditional		Sparse	
TEST/TIME	CPU	I/O	CPU	I/O
Shocktower	344.1	1090.3	149.6	584.1
Trackbar	232.8	897.8	182.1	629.5
Seatback	351.5	1213.5	213.0	701.6

Table 6.

7.0 Conclusion

The parametric study conducted on the sparse MPYAD method shows that, in general, this method performs best when all or some of the following matrix characteristics are present:

- A high number of strings per column
- A low number of terms per string
- A low density

When the above requirements are fulfilled, the new method always outperforms the classical ones. Also, by comparing the VAX and CONVEX results, it could be observed that vectorization allows the sparse method to be competitive even in the higher density situations.

The preliminary results of our tests on the sparse DECOMP and FBS are very encouraging and show significant improvements in the overall CPU times. It is important that the I/O times (due to the reduction of factor size and the amount of data transfer) are also reduced. On the other hand, the new sparse solution method (DECOMP + FBS) has a significantly higher memory requirement and has in fact a size limitation depending on the memory size. The speed-up ratios between the two solution methods are mainly dependent on problem characteristics, and partly related to the efficiency of the indirect addressing operations on the specific machine.

References

1. Komzsik L., MSC/NASTRAN Handbook of Numerical Methods, MSC Publication (to appear in 1990)
2. Dodson D. S. and Lewis J. G., "Proposed Sparse Extension to the BLAS," SIAM Conference 1989
3. Komzsik L., "On the Adaptation of MSC/NASTRAN to Supercomputers," Proceedings of International Workshop on Supercomputing Tools for Science and Engineering, Pisa Italy, Dec. 1989
4. MSC/NASTRAN Programmer's Manual, MSC Publication, May 1986