PARALLEL-VECTOR EQUATION SOLVERS FOR FINITE ELEMENT ENGINEERING APPLICATIONS

Duc T. Nguyen

Parallel-Vector Equation Solvers for Finite Element Engineering Applications

Parallel-Vector Equation Solvers for Finite Element Engineering Applications

Duc Thai Nguyen
Old Dominion University
Norfolk, Virginia

Springer Science+Business Media, LLC

Library of Congress Cataloging-in-Publication Data

Nguyen, Duc T.
Parallel-vector equation solvers for finite element engineering applications/Duc Thai Nguyen.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4613-5504-5 ISBN 978-1-4615-1337-7 (eBook)
DOI 10.1007/978-1-4615-1337-7
1. Finite element method. 2. Parallel processing (Electronic computers) 3. Differential equations—Numerical solutions. I. Title.

TA347.F5 N48 2001
620'.001'51535—dc21

2001038333

ISBN 978-1-4613-5504-5

Originally published by Kluwer Academic / Plenum Publishers, New York in 2002
Softcover reprint of the hardcover 1st edition 2002

10 9 8 7 6 5 4 3 2 1

A C.I.P. record for this book is available from the Library of Congress.

To Dac K. Nguyen
Thinh T. Thai
Hang N. Nguyen
Eric N. D. Nguyen and Don N. Nguyen

Preface

In spite of the fact that parallel-vector computational (equation solution) algorithms have been receiving a lot of attentions for over 20 years, and a large number of research articles have been published during this period, a limited number of texts and research books have been written on the subject. Most (if not all) existing texts on this subject have been written by computer scientists, and/or applied mathematicians. Majority of existing texts have been over-emphasizing on theoretical developments of new, and/or promising parallel (equation solution) algorithms (for varieties of applications in Engineering and Science disciplines). Materials presented in most existing texts are either too condense (without enough important detailed explanations), or too advance for the typical senior undergraduate and/or graduate engineering students. It should be emphasized here that while many important theoretical developments, which have significant impacts on highly efficient existing parallel-vector (equation solution) algorithms, have been carefully discussed and well-documented in current texts, important detailed computer implementations of the developed algorithms, however, have been usually omitted. Furthermore, it should be kept in minds that while few existing texts in this subject (direct equation solution algorithms for parallel and/or vector computers) have been written by computer scientists, and/or applied mathematicians, truly large-scale models (which require parallel and vector capabilities offered by modern high-performance computers) are often generated, solved, and interpreted by the engineering communities.

This book is written to address the concerns mentioned above and is intended to serve as a textbook for senior undergraduate, and graduate "engineering" students. A number of state-of-the-art FORTRAN codes, however, have been developed and discussed with great details in this textbook. Special efforts have been made by the author to present the materials in such a way to minimize the mathematical background requirements for typical senior undergraduate, and graduate engineering students. Thus, compromises between rigorous mathematics and practical simplicities are sometimes necessary.

This book has several unique features that distinguish it from other books:

1. Simplicity: The book has been written and explained in simple fashion, so that senior undergraduate and first year graduate students (in Civil, Mechanical, Aerospace, Electrical, Computer Science and Mathematic departments) can understand the presented materials with minimum background requirements. A working (undergraduate) knowledge in FORTRAN codings is helpful to understand the "detailed codings" of the presented materials. Some (undergraduate) linear algebra background should be useful, although it is NOT a requirement for reading and understanding the materials in the book. Undergraduate background in Matrix Structural Analysis and/or Finite Element Analysis should be useful, only for the materials presented in Chapter 3. Graph theories have not been traditionally introduced in the undergraduate/graduate engineering curriculums and therefore graph theories are not required to understand the materials presented in this book.
2. Algorithms are discussed for different parallel and/or vector computer platforms: Parallel and/or vectorized algorithms for various types of direct equation solvers are

presented and discussed for both "shared memory" (such as the Cray-2, Cray-YMP, Cray-C90, Convex) and "distributed memory" (such as the Intel i860, Intel Paragon, IBM-SP2, Meiko) computer platforms. The vectorized algorithms can also be "efficiently" executed on IBM-R6000/590 workstations. The vectorized algorithms and their associated FORTRAN codes can also be executed (with less efficiency) on other workstations and/or personal computers (P.C.) without having vectorized capabilities.

3. More emphasis on important detailed FORTRAN computer implementations: Efforts have been made to explain to the readers on important detailed FORTRAN computer implementations of various algorithms presented in the book. Thus, the readers should be able to incorporate the presented computer codes, subroutines into his/her application codes.
4. Several state-of-the-art FORTRAN equation solvers are discussed: While great amounts of effort have been spent to explain the detailed algorithms in a "simple fashion," many state-of-the-art equation solvers have been developed and presented in the book. Many of the presented solvers have been used by universities, large aerospace corporations and government research laboratories in the U.S., Europe and Asia.
5. Large-scale practical engineering finite element models are used: For derivations and explanations of various algorithms described in the book, small-scale examples are used to simplify and to facilitate the discussions. However, several medium to large-scale, practical engineering finite element models are used to demonstrate the efficiency and accuracy of the presented algorithms.
6. Algorithms are available for different types of linear equations: Different types of algorithms for the solutions of various types of system of simultaneous linear equations are presented in the book. Symmetrical/unsymmetrical, positive definite/negative definite/indefinite, incore/out-of-core, skyline/variable bandwidth/sparse/tridiagonal system of equations have all been treated in great detail by the author.

The book contains 11 chapters. Chapter 1 presents a brief review of some basic descriptions of shared and distributed parallel-vector computers. Measurements for algorithms' performance, and commonly "good practices" to achieve vector speed are also discussed in this chapter. Different storage schemes for the coefficient (stiffness) matrix (of system of linear equations) are discussed with great details in Chapter 2. Efficient parallel algorithms for generation and assembly of finite element coefficient (stiffness) matrices are explained in Chapter 3. Different parallel-vector "skyline" algorithms for shared memory computers (such as Cray-YMP, Cray-C90 etc...) are developed and evaluated in Chapter 4. These algorithms have been developed in conjunction with the skyline storage scheme, proposed earlier in Chapter 2. Parallel-vector "variable bandwidth" equation solution algorithms (for shared memory computers) are presented and explained in Chapter 5. These algorithms have been derived based upon the variable bandwidth storage scheme, proposed earlier in Chapter 2. Out-of-core equation solution algorithms on shared memory computers are considered in Chapter 6. These algorithms are useful for cases where very large-scale models need to be solved, and there are not enough core-memories to hold all arrays in the in-core

memories. Parallel-vector equation solution strategies for "distributed-memory" computers are discussed in Chapter 7. These equation solution strategies are based upon the parallel generation and assembly of finite element (stiffness) matrices, suggested earlier in Chapter 3. Unsymmetrical banded system of equations are treated in Chapter 8, where both parallel and vector strategies are described. Parallel algorithms for tri-diagonal system of equations on distributed computers are explained in Chapter 9. Sparse equation solution algorithms are presented in Chapter 10. Unrolling techniques to enhance the vector performance of sparse algorithms are also explained in this chapter. Finally, system of sparse equations where the coefficient (stiffness) matrix is symmetrical/ unsymmetrical and/or indefinite (where special pivoting strategies are required) are considered with great details in Chapter 11.

The book also contains a limited number of exercises to further supplement and reinforce the concepts and ideas presented. The references are listed at the end of each chapter.

The author would like to invite the readers to point out any errors that come to their attention. The author also welcomes any comments and suggestions from the readers.

Duc Thai Nguyen

Norfolk, Virginia

Acknowledgments

During preparation of this book, I have received (directly and indirectly) help from many people. First, I would like to express my sincere gratitude to my colleagues at NASA Langley Research Center: Dr. Olaf O. Storaasli, Dr. Jerrold M. Housner, Dr. James Starnes, Dr. Jaroslaw S. Sobieski, Dr. Keith Belvin, Dr. Peigman M. Maghami, Dr. Tom Zang, Dr. John Barthelemy, Dr. Carl Gray Jr., Dr. Steve Scotti, Dr. Kim Bey, Dr. Willie R. Watson, and Dr. Andrea O. Salas for their encouragement and support on the subject of this book during the past 13 years.

The close collaborative works with Dr. Olaf O. Storaasli and Dr. Jiangning Qin, in particular, have direct impacts on the writing of several chapters in this textbook.

I am very grateful to Dr. Lon Water (Maui, Hawaii), Professors Pu Chen (China), S.D. Rajan (Arizona), B.D. Belegundu (Pennsylvania), J.S. Arora (Iowa), Dr. Brad Maker (California), Dr. Gerald Goudreau (California), Dr. Esmond Ng (Lawrence Berkeley Laboratory, California) and Mr. Maurice Sancer (California) for their enthusiasm and supports of many topics discussed in this book.

My appreciation also goes to several of my former doctoral students, such as Dr. T.K. Agarwal, Dr. John Zhang, Dr. Majdi Baddourah, Dr. Al-Nasra, and Dr. H. Runesha who have worked with me for several years. Some of their research contributions have been included in this book.

In addition, I would like to thank my colleagues at Old Dominion University (ODU) and Hong Kong University of Science and Technology (HKUST) for their support, collaborative works, and friendship. Among them, Professor A. Osman Akan, Professor Isao Ishibashi, Professor Chuh Mei, and Professor Zia Razzaq at ODU, Professor T.Y. Paul Chang, and Professor Pin Tong at HKUST. Substantial portions of this textbook have been completed during my sabbatical leave period (January 1 - December 30, 1996) from O.D.U. to work at HKUST (during February 22 - August 22, 1996).

The successful publication and smooth production of this book are due to ODU skillful office supported staff: Mrs. Sue Smith, Mrs. Mary Carmone, Mrs. Deborah L. Miller, and efficient management and careful supervision of Mr. Tom Cohn, Ms. Ana Bozicevic, and Mr. Felix Portnoy, Editors of Kluwer/Plenum Publishing Corporation.

Special thanks go to Ms. Catherine John, from Academic Press (AP) Ltd., London, UK for allowing us to reproduce some materials from the AP textbook "Sparse Matrix Technology," (by Sergio Pissanetzky) for discussions in Chapter 10 (Tables 10.2 and 10.5) of our book.

Last but not least, I would like to thank my parents (Mr. Dac K. Nguyen, and Mrs. Thinh T. Thai), my family (Hang N. Nguyen, Eric N. D. Nguyen and Don N. Nguyen), whose encouragement has been ever present.

Duc T. Nguyen
Norfolk, Virginia

Disclaimer of Warranty

We make no warranties, express or implied, that the programs contained in this distribution are free of error, or that they will meet your requirements for any particular application. They should not be relied on for solving a problem whose incorrect solution could result in injury to a person or loss of property. The author and publisher disclaim all liability for direct, indirect, or consequential damages resulting from use of the programs or other materials presented in this book.

Contents

Parallel-Vector Equation Solvers for Finite Element Engineering Applications

1 Introduction

1.1 Parallel Computers

Modern high performance computers have multiple processing capabilities. The Convex, Sequent, Alliant, Cray-2, Cray-YMP [1.1] and Cray-C90 [1.2], parallel computers, for example, belong to the broad class of "shared memory" computers. The nCUBE, Intel i860, Intel Paragon, Meiko, and IBM-SP2 [1.3, 1.4] parallel computers, however, belong to the broad class of "distributed memory", or "message passing" computers. Shared memory computers, in general, consist of few (say 20, or less) processors. Each processor has its own local memory. Different processors, however, can be communicated to each other through shared memory area, as shown in Figure 1.1.

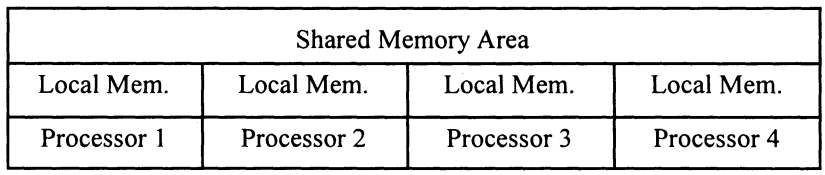


Figure 1.1 Shared memory parallel computers

Distributed memory (or message passing) computers, in general, consist of many (say hundreds, or thousands) processors (or nodes). Each processor has its own local memory, but the processor itself usually is less powerful (in terms of computational speed and memories) than its counterpart shared memory processor.

Communication amongst different nodes can only be done by message passing, as shown in Figure 1.2. Designing efficient algorithms, which can fully exploit the parallel and vector capabilities offered by shared memory computers have already been challenging tasks. It is generally safe to state that it is even more difficult to develop and to implement efficient parallel-vector algorithms on distributed memory computers!

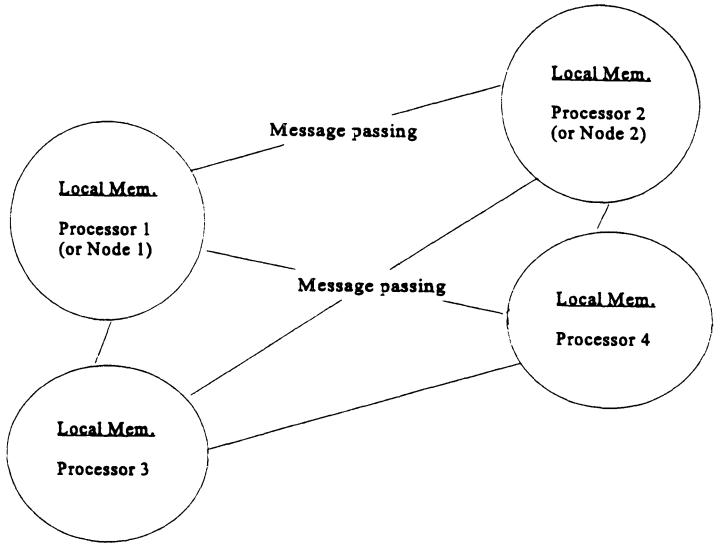


Figure 1.2 Distributed memory (or message passing) parallel computers

1.2 Measurements for Algorithms' Performance

The performance of parallel-vector algorithms can be measured by parameters, such as MFLOPS (Millions of FLoating point Operations Per Second), speed-up factor, and efficiency. Definitions of the above parameters are given in the following paragraphs.

$$MFLOPS = \frac{Total\ Number\ of\ Operations\ of\ a\ Given\ Algorithm}{10^6 * Time\ (in\ seconds,\ for executing\ a\ given\ algorithm)} \quad \textbf{(1.1)}$$

In Eq. (1.1), an operation is defined as a multiplication, division, addition, or subtraction.

$$SPEED\text{-}UP\ FACTOR = \frac{Time\ Obtained\ By\ Executing\ The\ Best\ Sequential\ Algorithm}{Time\ Obtained\ By\ Executing\ the\ Algorithm\ Using\ NP\ Processors} \quad \textbf{(1.2)}$$

In Eq. (1.2), NP represents the total number of processors used by a parallel algorithm

$$EFFICIENCY = \frac{SPEED\text{-}UP\ FACTOR}{NP} \quad \textbf{(1.3)}$$

Assuming $300*10^6$ operations, and 20 seconds (and 4 seconds) are required to execute the best sequential algorithm on a single processor (and 8 processors), respectively, then using Eqs (1.1 -1.3), one obtains

$$MFLOPS = \frac{300*10^6\ operations}{10^6*20\ seconds} = 15\ MFLOPS \quad \textbf{(1.4)}$$

$$SPEED\text{-}UP\,FACTOR = \frac{20\,Seconds}{4\,Seconds} = 5 \quad \textbf{(1.5)}$$

$$EFFICIENCY = \frac{5}{8} = 62.5\% \quad \textbf{(1.6)}$$

Typical curves for Time versus # Processors, and Speed-up-Factor versus # Processors are given in Figure 1.3. In practice, it is quite possible to see in Fig. 1.3 that time will increase (or speed-up factor will decrease) as the number of processors exceed, say 8 processors (as shown in Figure 1.3).

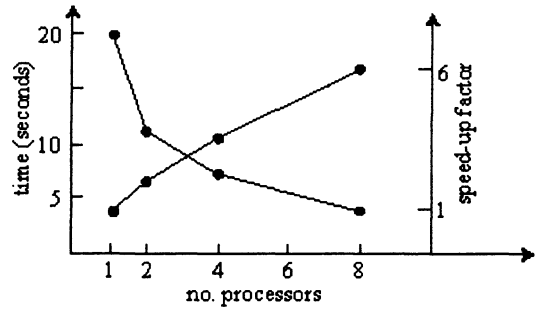


Figure 1.3 Typical curves for time and speed-up factor versus number of processors

1.3 Vector Computers

Vector computers utilize the concept of pipelining, which basically divides an arithmetic unit into different segments, each performs a subtask on a pair of operands (see Fig. 1.4).

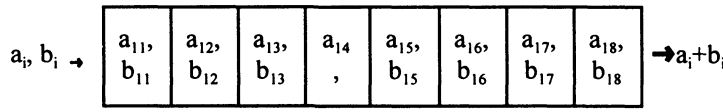


Fig. 1.4 Pipelining for vector addition

The main advantage of having several segmentations is that the results can be obtained at a rate 8 times faster (or more, depending on the number of segmentations), provided the data must reach the arithmetic unit quickly enough to keep the pipeline full at all time.

(a) Register-to-Register Processors

For register-to-register processors, the operands can be obtained directly from very fast memory, called vector registers, and store the results back into vector registers (see Fig. 1.5).

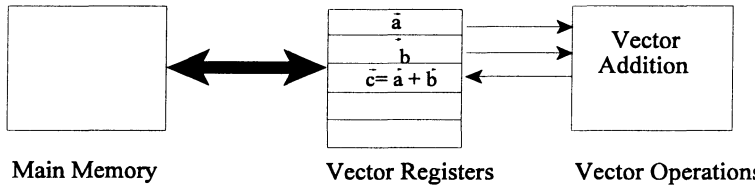


Figure 1.5 Register - to - register processors

(b) Vector Start-up Times

Vector operations do incur some overhead penalty, the amount of overhead cost depends on several factors, such as the vector lengths, the start-up time (time for the pipeline to become full), the time interval at which results are leaving the pipeline (related to the cycle time, or clock time of a particular computer).

(c) Sparse Matrix Computation

For sparse matrix computation, the following FORTRAN loop is often required (as shown in Figure 1.6)

```
D0  1 I = 1, N
K = INDEX (I)
Y(K)=a * X (I) + Y(K)
```

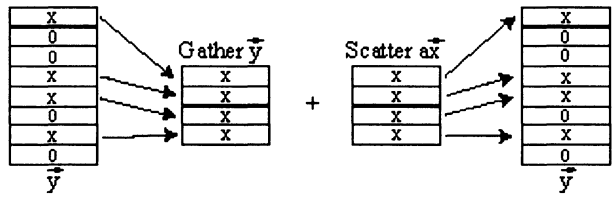


Figure 1.6 Sparse matrix loop

(d) Avoid Putting Subroutine and/or Function Calls Within D0 L00PS

It is not a good idea, in general, to put subroutine(s) and/or function(s) calls within the

D0 Loops (see Table 1.1), since it may prevent the compiler for vectorizing. Good vectorization will be realized, however, if the algorithm shown in Table 1.1 is modified, as shown in Table 1.2. It should also be noticed here that the index J (instead of I) is used for the inner loop in Table 1.2 to avoid any dependencies within the inner loop. In other words, for the fixed value of I, the inner-loop (shown in Table 1.2) does not have any dependencies on the index J.

Table 1.1 Vectorization can be prevented if subroutine call is placed inside D0 L00P

```
      D0   1   J = 2, N
      D0   1   I = 2, N
      CONST1 = VEL (I, J)
      CONST0 = VEL (I-1, J)
      CONST2 = VEL (I+1, J)
      CALL ABC (CONST1, CONST0, CONST2)
      VEL(I,J) = SQRT (CONST1)
1     CONTINUE
            ⋮
      SUBROUTINE ABC (CONST1, CONST0, CONST2)
      CONST1 = 3.8 * CONST1 + 2.7 * (6.2 + CONST0 + CONST2)
      RETURN
      END
```

Table 1.2 Vectorization can be realized if subroutine call is removed from D0 L00P

```
      DO 1 I = 2, N
      D0 1 J = 2, N
      VEL (I, J) = 3.8 * VEL (I, J) + 2.7 * (6.2 + VEL (I-1, J) + VEL(I + 1, J))
      VEL(I, J) = SQRT(VEL (I, J) )
1     CONTINUE
```

(e) Using Few Loops (with more work loads) Rather Than Using Many Loops (with less work loads)

The algorithm shown in Table 1.3 (a) basically involved with vector addition, to be followed by vector multiplication, and again by vector addition. While this algorithm will be vectorized, better vectorization can be realized by rewriting the algorithm into the form shown in Table 1.3 (b).

Table 1.3 Many loops (with less work loads) versus few loops (with more work loads)

(a) 3 Loops Are Used	(b) 1 Loop Is Used
CALL ADDV (N, A, B, RESULT1)	DO 1 I = 1, N
CALL MULV (N, RESULT1, A, RESULT2)	A(I) = (A(I) + B(I))*A(I)+B(I)
CALL ADDV (N, RESULT2, B, RESULT1)	1 CONTINUE

(f) Innermost D0-LOOP Should Have Long Vector Length

The algorithm shown in Table 1.4 has poor vector performance, since the innermost D0 LOOP only has the vector length of 8. Substantial improvements in the vector performance can be realized, however, simply by reversing the order of the I and J loops.

Table 1.4 Avoid using short vector length for inner-most D0 LOOP

```
      D0 1 I = 1, 4000
      D0 1 J = 1, 8
1     A(I, J) = (A(I,J) + B(I,J) ) * A(I, J) + B(I,J)
```

(g) Using Compiler Directive if Necessary

The appearance of the FORTRAN code segment, shown in Table 1.5, seems to indicate the dependency of the inner-most loop index L, and hence vectorization may not be recognized by the compiler. Careful examinations on the innermost loop computation, however, will reveal that for the fixed value of index K, the "even-values" for innermost loop index L (on the left hand side) depend only on the "odd-values" for index L. Thus, there is no dependency on the innermost index L, and therefore, usage of compiler directive (see CDIR$ IVDEP in Table 1.5) is appropriate. The compiler directive statement may be different for different computers. However, some modern (high-performance) vector computers do have compiler directive statements.

Table 1.5 Compiler directive should be used appropriately

```
      D0  3  K = 2, N, 2
      CDIR$ IVDEP
      D0  3  L = 2, N, 2
      A(L, K) = 7.8 * (A(L-1, K) + A(L+1, K) +A(L,K-1)+A(L, K+1)
3     CONTINUE
```

(h) Avoid to Use If Statement within Innermost DO-LOOP

Intrinsic functions may be used to replace If statement within innermost do-loop, as

illustrated in Table 1.6.

Table 1.6 Avoiding IF statement by using intrinsic functions

(a) Avoid IF Statement	(b) Use Intrinsic Statement
DO 22 K = 1, N	DO 22 K=1, N
IF (C(K). LT. 0.) C(K) = 0.	C(K) = AMAX1 (C(K), 0.)
22 A(K) = SQRT (C(K)) * . . .	22 A(k) = SQRT (C(K)) * . . .

(i) Avoid too Use Temporary Arrays

Even though both cases shown in Table 1.7 will be vectorized, case 2 will have better vector performance than case 1, because the former avoids using temporary array T(-)

Table 1.7 Avoid using temporary array

Case 1	Case 2
DO 1 J = 2, N	DO 1 J = 2, N
T(J) = D(J-1)	D(J) = E(J)
D(J) = E (J)	F(J) = D(J-1)
F(J) = T (J)	1 CONTINUE
1 CONTINUE	

(j) Avoid Scalar Variable Which are Computed before the Execution of the Containing Loop

Segments of the FORTRAN codes, shown in Table 1.8, can be used to illustrate the disadvantage of calculating scalar variables before the execution of the containing loop.

Table 1.8 Avoid computing scalar variables before execution of the containing loop

Case 1 (No Vectorization)	Case 2 (With Vectorization)
C = 0	C (1) = 0
DO 1 K = 2, N	DO 1 K=2, N
D = E (K) *F(K)	C(K) = E (K) * F(K)
G(K) = D + C	1 G(K) = C(K) + C(K-1)
1 C = D	

(k) "Vector Unrolling" and Dot Product Operations

Assuming a square matrix $[A]_{N,N}$ and a vector $\{x\}_{N,1}$ are given, and the objective is to compute matrix-vector multiplications ([A] * {x}), and to store the results into vector $\{y\}_{N,1}$. Algorithm based upon dot product operations (for matrix-vector multiplications) is given in Table 1.9, while better (load and store) vector performance can be obtained

by using dot product operations in conjunction with "vector unrolling," say level 2 (see the increment for the index I), is illustrated in Table 1.10.

Table 1.9 Dot-product operations

```
      DO 21 I = 1, N
      DO 21 J = 1, N
21    Y(I) = Y(I) + A(I,J) * X(J)
```

Table 1.10 Dot-product operations with "vector unrolling" level 2

```
      DO 21 I = 1, N, 2
      DO 21 J = 1, N
      Y(I) = Y(I) + A(I, J) *X(J)
21    Y(I+1) = Y (I+1) +A(I+1, J) * X(J)
```

For the fixed value of index I, the innermost loop operations (shown in Table 1.9) A(I,J) * X(J) is essentially the dot product of 2 vectors A(I,J) and X(J).

(l) "Loop Unrolling" and Saxpy Operations

Given the matrix $[A] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}$, and the vector $X = \begin{Bmatrix} 1 \\ 2 \\ 3 \end{Bmatrix}$, the product of [A}*{X}

≡ {y} can be computed by the following steps.

Step 1: "Partial" answer for vector {y} can be obtained by multiplying the first column of [A] with
the first component of {X}

$$\{y\} = \begin{Bmatrix} -1 \\ 2 \\ -1 \end{Bmatrix} *(2) + \begin{Bmatrix} 2 \\ -1 \\ 0 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 3 \\ -2 \end{Bmatrix} \quad \textbf{(1.7)}$$

Step 2: Multiplying the second column of [A] with the second component of {X}, then adding the results into the "partial" answer of {y}, in order to obtain the "updated" answer for {y}.

$$\{y\} = \begin{Bmatrix} 2 \\ -1 \\ 0 \end{Bmatrix} * (1) = \begin{Bmatrix} 2 \\ -1 \\ 0 \end{Bmatrix} \quad \textbf{(1.8)}$$

Step 3: Multiplying the third (or the last) column of [A] with the third component of

{X}, then adding the results into the "updated" answer for {y}, in order to obtain the "final" answer for {y}

$$\{y\} = \begin{Bmatrix} 0 \\ -1 \\ 1 \end{Bmatrix} * (3) + \begin{Bmatrix} 0 \\ 3 \\ -2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix} \qquad \textbf{(1.9)}$$

The operations involved in Eqs. (1.8) and (1.9) can be recognized in the form of Summation of a constant "a" times a vector {x} plus a vector {y}. Thus, the above operations are called Saxpy operations!

Algorithms based upon saxpy operation (for matrix-vector multiplications) is given in Table 1.11, while better (load and store) vector performance can be obtained by using saxpy operations in conjunctions with "loop unrolling," say level 2 (see the increment for the index J), is illustrated in Table 1.12.

Table 1.11 Saxpy operations

```
      DO 22 J = 1, N
      DO 22 I = 1, N
22    Y(I) = Y(I) + A(I, J) * X(J)
```

Table 1.12 Saxpy operations with "loop unrolling" level 2

```
      DO 22 J = 1, N, 2
      DO 22 I = 1, N
22    Y(I) = Y(I) + A(I, J) * X(J) + A(I, J +1) * X(J + 1)
```

For many computers, such as the Cray-2, Cray-YMP and Cray-C90 etc..., saxpy operations can be substantially faster than dot product operations. It is also important to emphasize here the key differences between dot product operations (and its associated vector unrolling operations), and saxpy operations (and its associated loop unrolling operations). The former will give the "final" results, while the latter will only give the "partial" results. Furthermore, "vector unrolling" operations involve with several FORTRAN statements within the inner-most DO LOOP, whereas "loop unrolling" operations involve with a single FORTRAN statement (but with more calculations attached) within the inner-most DO-LOOP.

(m) Stride

For a given 3x3 matrix [A], as discussed in the previous section, and assuming the matrix [A] is stored in a row-by-row fashion, then the basic dot-product operations (in Table 1.9) will have the stride to be equal to 3. The matrix [A] will be internally stored in a column-by-column fashion. Thus, for the given data, matrix A can be internally represented as a "long" vector;

$$[A] = \left\{ \begin{array}{c} 2 \\ -1 \\ 0 \\ \hline -1 \\ 2 \\ -1 \\ \hline 0 \\ -1 \\ 1 \end{array} \right\} = \left\{ \begin{array}{c} 1st\ column \\ \hline 2^{nd} column \\ \hline 3^{rd} column \end{array} \right\} \qquad \textbf{(1.10)}$$

Corresponding to a fixed I^{th}- row, say I = 1, the operations inside the innermost DO-loop will involve with the dot-product between the 2 vectors

$$A = \{2, -1, 0\} \ and \ X = \left\{ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} \right\} \qquad \textbf{(1.11)}$$

The first vector in Eq. (1.11) can be retrieved from the data shown in Eq. 1.10, at the 1^{st}, 4^{th} and 7^{th} locations, respectively. The "distance" between any 2 consecutive numbers is called the "stride." Thus, in this example (for algorithm shown in Table 1.9), one obtains:

stride = 4^{th} location - 1^{st} location = 7^{th} location - 4^{th} location
stride = 3

However, if saxpy operations are used for the same matrix-vector multiplications (as indicated in Eq. 1.7), then the stride will be equal to 1. The reason is because the 3 consecutive numbers used in Eq. (1.7), such as $\left\{ \begin{array}{c} 2 \\ -1 \\ 0 \end{array} \right\}$, can be retrieved from the 1^{st}, 2^{nd}, and 3^{rd} locations, respectively (see Eq. 1.10). Thus:

stride = 2^{nd} location - 1^{st} location = 3^{rd} location - 2^{nd} location
stride = 1.

For better vector performance, the smallest stride (such as stride = 1) is the most desirable!

1.4 Summary

Modern high-performance computers (Cray - C90, Intel Paragon, IBM - SP2 etc...) offer both parallel, cache, and vector processing capabilities. Thus, efficient algorithms need to exploit both parallel, cache, and vector capabilities. For nested do-loops (such as matrix factorization), effective vectorization need to be done at the innermost do-loop, while effective parallelization need to be done at the outermost do-loop.

1.5 Exercises

1.1 Given the coefficient matrix [A] and the vector {x} as following:

$$[A] = \begin{bmatrix} 4 & 1 & 2 & 0 \\ -1 & 5 & 1 & 1 \\ 2 & 1 & 3 & -2 \\ 0 & -1 & 2 & 8 \end{bmatrix} \quad and \quad \{x\} = \begin{Bmatrix} 1 \\ 2 \\ 3 \\ -1 \end{Bmatrix}$$

Using a hand calculator,
(a) find the product of [A]*{x} by employing "Dot Product" operations.
(b) find the product of [A]*{x} by employing "saxpy" operations.

1.2 Write a general purpose Fortran computer program to compute the product of [B] * {y}, using "saxpy" operations with loop-unrolling level 3, where:

$$[B] = \begin{bmatrix} A & & & \\ & A & & \\ & & A & \\ & & & A \end{bmatrix}_{20\times 20} \quad and \quad \{y\} = \begin{Bmatrix} x \\ x \\ x \\ x \\ x \end{Bmatrix}_{20\times 1}$$

The matrix $[A]_{4\times4}$ and vector $\{x\}_{4\times1}$ has been defined in Problem 1.1 (Hint: Algorithm shown in Table 1.12 needs to be modified slightly)

1.3 Same as described in Problem 1.2, but using loop-unrolling level 8

1.6 References

1.1 Cray-YMP manuals
1.2 Cray-C90 manuals
1.3 IBM-SP2 manuals
1.4 IBM-R6000/590 manuals

2 Storage Schemes for the Coefficient Stiffness Matrix

2.1 Introduction

For many important engineering and science applications [2.1, 2.2], the coefficient matrix [A] involved in the system of linear simultaneous equations

$$[A]\ \{X\} = \{b\} \tag{2.1}$$

is usually symmetrical, positive-definite and sparse. In structural engineering applications [A], {X} and {b} represent stiffness matrix, nodal displacement vector and nodal load vector, respectively.

Consider, for example, a 9 x 9 (stiffness) matrix [A] of the type:

$$[A]= \begin{bmatrix} A_{11} & A_{12} & 0 & A_{14} & 0 & 0 & 0 & 0 & 0 \\ A_{12} & A_{22} & A_{23} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & A_{23} & A_{33} & A_{34} & 0 & A_{36} & 0 & 0 & 0 \\ A_{14} & 0 & A_{34} & A_{44} & A_{45} & A_{46} & 0 & 0 & 0 \\ 0 & 0 & 0 & A_{45} & A_{55} & A_{56} & 0 & A_{58} & 0 \\ 0 & 0 & A_{36} & A_{46} & A_{56} & A_{66} & A_{67} & 0 & A_{69} \\ 0 & 0 & 0 & 0 & 0 & A_{67} & A_{77} & A_{78} & 0 \\ 0 & 0 & 0 & 0 & A_{58} & 0 & A_{78} & A_{88} & A_{89} \\ 0 & 0 & 0 & 0 & 0 & A_{69} & 0 & A_{89} & A_{99} \end{bmatrix} \tag{2. 2}$$

There are several different storage schemes that can be used for the matrix expressed in Eq. (2.2). Different storage schemes will lead to different storage requirements. Furthermore, as will be explained in subsequent chapters, the choice for the appropriate storage scheme will be influenced (or dictated) by the equation solution strategies to be employed for solving the unknown vector {X} from Eq. (2.1).

Since effective equation solution algorithms are heavily influenced by the storage scheme used to store the coefficient matrix [A] (shown in eqs. 2.1-2.2), the objective of this chapter is to describe some common storage schemes used in many practical engineering and science applications.

2.2 Full Matrix

This is the simplest way to store matrix [A] of Eq. (2.2). The entire matrix is stored, thus 81 (=9x9) words of memory is required to store matrix [A].

2.3 Symmetrical Matrix

In this storage scheme, only the upper triangular portion (including diagonal terms) of matrix [A] needs to be stored (due to the symmetrical property of [A]). Thus, this storage scheme only requires 45 (=9x10/2) words of memories.

2.4 Banded Matrix

By observation, the maximum bandwidth (the maximum "distance" between the diagonal term and the last non-zero term in each row) of the matrix [A] is 4 (including the diagonal term). Furthermore, taking the advantage of symmetry, matrix [A] can be stored as

$$[A] = \begin{matrix} A_{11} & A_{12} & 0 & A_{14} \\ A_{22} & A_{23} & 0 & 0 \\ A_{33} & A_{34} & 0 & A_{36} \\ A_{44} & A_{45} & A_{46} & 0 \\ A_{55} & A_{56} & 0 & A_{58} \\ A_{66} & A_{67} & 0 & A_{69} \\ A_{77} & A_{78} & 0 & 0 \\ A_{88} & A_{89} & 0 & 0 \\ A_{99} & 0 & 0 & 0 \end{matrix} \qquad \textbf{(2.3)}$$

It should be noted here that the diagonal terms of matrix [A] in Eq. (2.2) are shifted to become the first column of the rectangular matrix [A] in Eq. (2.3). This banded storage scheme requires 36 (9x4) words of memory.

2.5 Variable Banded Matrix [2.3]

In this storage scheme, the matrix [A] shown in Eq. (2.2) is stored as a *1-dimensional array* according to the following *row-by-row* fashion

A= | A_{11} | A_{12} | 0 | A_{14} | A_{22} | A_{23} | 0 | A_{33} | A_{34} | 0 | A_{36} | A_{44} | A_{45} | A_{46} | A_{55} | A_{56} |

| 0 | A_{58} | A_{66} | A_{67} | 0 | A_{69} | A_{77} | A_{78} | 0 | A_{88} | A_{89} | A_{99} |

The above 1-dimensional (row-by-row) array corresponds to the following 2-dimensional array (or matrix) representation:

$$
[A] = \begin{array}{ccccccccc}
A_{11} & A_{12} & 0 & \mathbf{A_{14}} & & & & & \\
 & A_{22} & A_{23} & \mathbf{0} & & & & & \\
 & & A_{33} & A_{34} & 0 & \mathbf{A_{36}} & & & \\
 & & & A_{44} & A_{45} & \mathbf{A_{46}} & & & \\
 & & & & A_{55} & A_{56} & 0 & \mathbf{A_{58}} & \\
 & & & & & A_{66} & A_{67} & 0 & \mathbf{A_{69}} \\
 & S & Y & M. & & & A_{77} & A_{78} & \mathbf{0} \\
 & & & & & & & A_{88} & \mathbf{A_{89}} \\
 & & & & & & & & A_{99}
\end{array} \qquad \textbf{(2.4)}
$$

It should be observed that in Eq. (2.4), the (imaginary) vertical (see bold face numbers) enveloped lines always keep shifting toward the right direction. The lower triangular portion of [A] in Eq. (2.4) is not shown due to the symmetry of [A]. As it will be explained with more details, this variable banded (row-by-row) storage scheme requires 28(=4+3+4+3+4+4+3+2+1) words of memory.

2.6 Skyline Matrix [2.1, 2.4]

In this storage scheme, the matrix [A] shown in Eq. (2.2) is stored as a *1-dimensional array* according to the following column-by-column fashion.

$$
[A] = \begin{array}{ccccccccc}
A(1) & A(3) & & A(9) & & & & & \\
 & A(2) & A(5) & A(8) & & & & & \\
 & & A(4) & A(7) & & A(15) & & & \\
 & & & A(6) & A(11) & A(14) & & & \\
 & & & & A(10) & A(13) & & A(21) & \\
 & & & & & A(12) & A(17) & A(20) & A(25) \\
 & S & Y & M. & & & A(16) & A(19) & A(24) \\
 & & & & & & & A(18) & A(23) \\
 & & & & & & & & A(22)
\end{array} \qquad \textbf{(2.5)}
$$

The "height" of each column (including the diagonal term) of [A] in Eq. (2.5) is usually referred to as the "skyline" of matrix [A]. It can be observed from Eq. (2.5) that the original values of A(8), A(20) and A(24) are all zeroes, since these numbers correspond to A_{24}, A_{68} and A_{79}. As it will be explained with more details in subsequent chapters, these initial zero values may become non-zero values later on (during the factorization phase). In the literature, the non-zero values for A(8), A(20) and A(24) are commonly referred to as "fills-in." This skyline (column-by-column) storage scheme requires only 25(=1+2+2+4+2+4+2+4+4) words of memory

2.7 Sparse Matrix [2.5]

In this storage scheme, only non-zero terms of matrix [A] are stored according to the following 1-dimensional, (row-by-row), integer and real arrays:
Thus, the integer array IA (N+1), where N is the size of the matrix, describes the

$$IA \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 3 \\ 4 \\ 6 \\ 8 \\ 10 \\ 12 \\ 13 \\ 14 \\ 14 \end{Bmatrix} \qquad \textbf{(2.6)}$$

starting locations of the first non-zero, off-diagonal term in each row. The number of non-zero, off-diagonal terms in each row of matrix [A] can be easily computed from the IA(N+1) array, for example:

the number of non-zero, off-diagonal terms in the 1st row is IA(2)-IA(1) =2
the number of non-zero, off-diagonal terms in the 2nd row is IA(3)-IA(2)=1
the number of non-zero, off-diagonal terms in the 6th row is IA(7)-IA(6)=2
the number of non-zero, off-diagonal terms in the 7th row is IA(8)-IA(7)=1
the number of non-zero, off-diagonal terms in the 8th row is IA (9)-IA(8)=1
the number of non-zero, off-diagonal terms in the 9th row is IA(10)-IA(9)=0

The column numbers of the non-zero, off-diagonal terms in each row can be described by an array JA(NCOF), where NCOF is the total number of non-zero, off-diagonal terms of matrix [A] (before factorization). For the matrix data given by Eq. (2.2), one has:

NCOF = IA (N+1) - IA(1) = 14-1 = 13

$$JA \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 4 \\ 3 \\ 4 \\ 6 \\ 5 \\ 6 \\ 6 \\ 8 \\ 7 \\ 9 \\ 8 \\ 9 \end{Bmatrix} \qquad \textbf{(2.7)}$$

The numerical values of diagonal terms of [A] can be described by the array D(N), where N=9, and:

$$D \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{pmatrix} = \begin{Bmatrix} A_{11} \\ A_{22} \\ A_{33} \\ A_{44} \\ A_{55} \\ A_{66} \\ A_{77} \\ A_{88} \\ A_{99} \end{Bmatrix} \tag{2.8}$$

The numerical values of off-diagonal terms of [A] can be described by the array AN (NCOF), where NCOF = 13, and:

$$AN \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ . \\ . \\ . \\ . \\ . \\ . \\ 13 \end{pmatrix} = \begin{Bmatrix} A_{12} \\ A_{14} \\ A_{23} \\ A_{34} \\ A_{36} \\ A_{45} \\ . \\ . \\ . \\ . \\ . \\ . \\ A_{89} \end{Bmatrix} \tag{2.9}$$

In this truly sparse storage scheme, the number of storage requirements for matrix [A], see Eq. (2.2), is only 22 (=9 for storing diagonal terms +13 for storing off-diagonal terms). For large-scale engineering and science applications, sparse storage scheme is the most efficient one.

2.8 Detailed Procedures for Determining the Mapping Between 2-D Array and 1-D Array in Skyline Storage Scheme

The stiffness matrix [A] can be expressed either in a 2-dimensional array, or in a 1-dimensional array, as indicated by Eq. (2.2), or Eq. (2.5), respectively. The key issue that needs to be discussed in this section is evolved around the following question:

How can we find the mapping between Eq. (2.2) and Eq. (2.5) ?? In other words, how do we know that A_{55} and A_{69} in Eq. (2.2) will be mapped into A_{10} and A_{25} in Eq. (2.5), respectively ??

Let's first define the "column heights" of a given symmetrical matrix [A], such as shown in Eq. (2.2), as following:

Column height of the i^{th} column of a matrix A is defined as the "distance" between the diagonal term and the furthest non-zero term of the same i^{th} column. The diagonal term itself, however is NOT included in the calculation for the distance.

With the above definition, the column heights of the symmetrical matrix [A], shown in Eq. 2.2, can be defined by the integer array ICOLH(N) as following:

$$ICOLH \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 1 \\ 1 \\ 3 \\ 1 \\ 3 \\ 1 \\ 3 \\ 3 \end{Bmatrix} \tag{2.10}$$

Thus, column height of column 1 of [A] is 0
column height of column 2 of [A] is 1
column height of column 8 of [A] is 3
column height of column 9 of [A] is 3

Once the column heights array ICOLH(N) is known, we can easily determine the mapping of the diagonal terms' locations between Eq. (2.2) and Eq. (2.5), through the integer array MAXA (N+1), where:

$$MAXA\,(1) = 1 \tag{2.11}$$

$$MAXA\,(I+1) = MAXA(I) + ICOLH(I) + 1 \tag{2.12}$$

Using the known column height information (shown in Eq. 2.10), the mapping of the diagonal locations, array MAXA(N+1), can be determined (by referring to Eqs. 2.11 and 2.12) as:

$$MAXA \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \\ 4 \\ 6 \\ 10 \\ 12 \\ 16 \\ 18 \\ 22 \\ 26 \end{Bmatrix} \tag{2.13}$$

From Eq. (2.13), we can easily see that the diagonal terms A_{11}, A_{22}, A_{88} and

A_{99} (of the 2-dimensional array, shown in Eq. 2.2) will be mapped into the locations 1, 2, 18 and 22, respectively (of the 1-dimensional array, shown in Eq. 2.5).

The total number of storages, NTERMS, required to store the entire stiffness matrix [A], in a skyline format, is given as

$$\begin{aligned} NTERMS &= MAXA(N+1) - MAXA(1) \\ &= MAXA(10) - 1 = 26 - 1 = 25 \end{aligned} \tag{2.14}$$

Finally, the mapping between the 2-dimensional stiffness array [A], shown in Eq. (2.2), and the 1-dimensional stiffness array [A], shown in Eq. (2.5), can be established as:

$$A_{i,j} = A[Maxa(j) + j - i] \tag{2.15}$$

Using Eq. (2.15), the term $A_{5,5}$ will be stored in the 1-dimensional array as

$A_{5,5} = A[Maxa(5) + 5 - 5]$

$A_{5,5} = A\ [Maxa(5)] = A(10)$

Similarly, the term $A_{6,9}$ can be mapped into the 1-dimensional array as

$A_{6,9} = A[Maxa(9) + 9 - 6]$

$A_{6,9} = A[Maxa(9) + 3] = A(22+3)$

$A_{6,9} = A(25)$

2.9 Determination of the Column Heights (ICOLH) of a Finite Element Model

Figure 2.1 represents a simple structure which is modeled by 4 rectangular elements (with 2 translational degree-of-freedom at each node) and 9 nodes. Nodes 1 through 3 are constrained by the pin support boundary conditions. Thus, there are no translational motions in these three nodes. To specify the boundary conditions at each node, the following conventions are adopted:

If a particular degree-of-freedom (dof) of a node is fixed (due to support boundary condition), then this dof is assigned a value 1. If a particular degree-of-freedom (dof) of a node is free to move, then this dof is assigned a value 0.

Since at each node, there are usually at most 6 dof (3 translational dof, T_x, T_y & T_z and 3 rotational dof R_x, R_y & R_z about the three coordinate axis), one can construct the integer array ID(6, NUMNP) from Figure 2.1 as:

$$[ID]_{6x9} = \begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ T_x & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_y & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_z & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ R_x & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ R_y & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ R_z & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} \quad \textbf{(2. 16)}$$

Since there are 9 nodes in the finite element model, shown in Figure 2.1, the integer array ID in Eq. (2.16) has 9 columns.

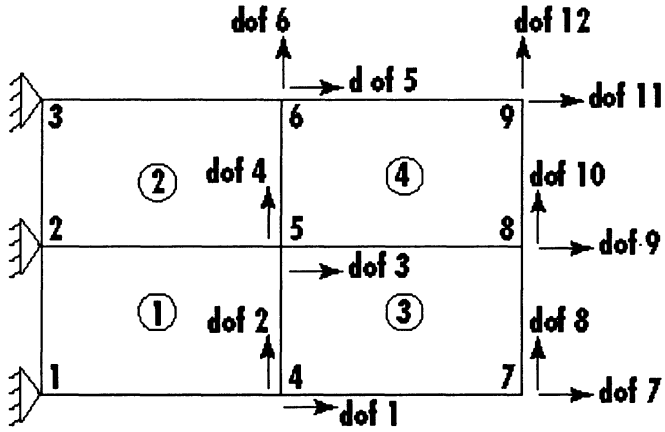


Figure 2.1 Finite element model with 4 rectangular elements

It is obvious from Eq. (2.16) and from Figure 2.1 that there are two (free to move) translational dof at nodes 4 to 9, while the rests of the dof at these nodes are fixed. All dof at nodes 1 to 3 are also fixed.

Thus, the ID array in Eq. (2.16) can be modified to become:

$$[ID]_{6x9} = \begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ T_x & 0 & 0 & 0 & 1 & 3 & 5 & 7 & 9 & 11 \\ T_y & 0 & 0 & 0 & 2 & 4 & 6 & 8 & 10 & 12 \\ T_z & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ R_x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ R_y & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ R_z & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \quad \textbf{(2. 17)}$$

Equation (2.17), therefore, will give us the global dof (or global equation) associated with each node. Since each of the rectangular element (shown in Figure 2.1) is associated with 4 nodes, there are 8 dof associated with each rectangular element.

Since the nodal numbers associated with each rectangular elements are known (refer to Figure 2.1), the global dof, (array LM), associated with each element can be defined (with the help of Eq. 2.17) as:

Rectangular element 1 is connected by nodes 5, 2, 1 & 4, hence

$$LM^{(e=1)} = \begin{pmatrix} 3 \\ 4 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 2 \end{pmatrix} \quad \textbf{(2. 18)}$$

Rectangular element 2 is connected by nodes 6, 3, 2 & 5, hence

$$LM^{(e=2)} = \begin{pmatrix} 5 \\ 6 \\ 0 \\ 0 \\ 0 \\ 0 \\ 3 \\ 4 \end{pmatrix} \quad \textbf{(2. 19)}$$

Rectangular element 3 is connected by nodes 8, 5, 4 & 7, hence

$$LM^{(e=3)} = \begin{pmatrix} 9 \\ 10 \\ 3 \\ 4 \\ 1 \\ 2 \\ 7 \\ 8 \end{pmatrix} \quad \textbf{(2.20)}$$

Rectangular element 4 is connected by nodes 9, 6, 5 & 8, hence

$$LM^{(e=4)} = \begin{pmatrix} 11 \\ 12 \\ 5 \\ 6 \\ 3 \\ 4 \\ 9 \\ 10 \end{pmatrix} \quad \textbf{(2. 21)}$$

The total structural stiffness matrix [A] of Figure 2.1 has 12 active dof (refer to Eq. 2.17), which can be obtained from the contributions of 4 rectangular element stiffness, as shown in the following Figure 2.2.

[A] =

	1	2	3	4	5	6	7	8	9	10	11	12
1	① ③	① ③	① ③	① ③			③	③	③	③		
2		① ③	① ③	① ③			③	③	③	③		
3			① ②	① ②	② ④	② ④	③	③	③ ④	③ ④	④	④
4				① ②	② ④	② ④	③	③	③ ④	③ ④	④	④
5					② ④	② ④			④	④	④	④
6						② ④			④	④	④	④
7							③	③	③	③		
8								③	③	③		
9									③ ④	③ ④	④	④
10										③ ④	④	④
11											④	④
12												④

Figure 2.2 Total stiffness matrix of 4 rectangular finite element model

The column heights, ICOLH, of matrix [A] shown in Figure 2.2 can be identified as

$$ICOLH \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \end{pmatrix} = \begin{Bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 2 \\ 3 \\ 6 \\ 7 \\ 8 \\ 9 \\ 8 \\ 9 \end{Bmatrix} \tag{2.22}$$

2.10 Computer Implementation for Determining Column Heights

In the previous section, a simple finite element model has been used to illustrate detailed steps to obtain the assembled, structural stiffness matrix. Once the structural stiffness matrix has been assembled, the column heights (refer to Eq. 2.22) and diagonal locations (refer to Eqs. 2.11 and 2.12) can be easily determined.

The purpose of this section is to present and explain a simple algorithm (in the form of "pseudo" FORTRAN coding, shown in Table 2.1) to obtain the column heights information directly from element connectivity data and without the need to assemble the structural stiffness matrix.

It should be emphasized here that for parallel-vector skyline (column-by-column) equation solution strategies (to be discussed in Chapter 4), column heights information is crucially important, since it contains enough information to describe the non-zero terms required during the factorization phase.

Table 2.1 Algorithm to find column heights

1	C.	NEQ = number of equations
2	C.	NEL = number of elements (say = 4)
3	C.	NDOFPE = number of degree-of-freedom (dof) per element
4	C+ + + +	Initialized column height array
5		DO 1 I = 1, NEQ
6		ICOLH (I) = 0
7	1	CONTINUE
8	C+ + + +	Looping through all finite elements
9		DO 2 J = 1, NEL
10	C+ + + +	Looping through all dof per element to find min. dof
11		MINDOF = 10000000
12		DO 3 K=1, NDOFPE (say = 8)

```
13      C. . . . . LM(K) = global dof (or equation) associated with each element
14                 IDOF = LM(K)
15                 If (IDOF.EQ.O) GO TO 3
16                 If (IDOF.LT.MINDOF) MINDOF = IDOF
17      3          CONTINUE
18      C+ + + +   Begin to find and update column heights
19                 DO 4 K = 1, NDOFPE
20                 IDOF = LM(K)
21                 IF (IDOF.EQ.O) Go to 4
22                 ICH = IDOF - MINDOF
23                 IF (ICOLH(IDOF).LT.ICH) ICOLH(IDOF) = ICH
24      4          CONTINUE
25      2          CONTINUE
```

In Table 2.1, lines 10 through 17 will determine the minimum degree-of-freedom number (=MINDOF, see line 16) associated with each finite element. It is assumed that all the degree-of-freedoms (dof) associated with each element (=LM(K), see lines 14, 20) is known before using the algorithm shown in Table 2.1.

The IF statements on lines 15 and 21 will skip those dof with zero prescribed displacement boundary conditions. These boundary conditions are also referred to as Dirichlet boundary conditions.

Lines 18 through 24 (in Table 2.1) will find and update the column heights of all dof associated with each finite element. The IF statement on line 23 will assure that the "old" column height be updated only if its old value is less than its current (column height) value.

For the data shown in Figure 2.1 and Eqs. (2.18-2.21), the column heights after processing element 1 (the smallest dof = MINDOF = 1) can be computed as (refer to Table 2.1):

$$ICOLH\begin{pmatrix}3\\4\\1\\2\end{pmatrix} = \begin{Bmatrix}3-1\\4-1\\1-1\\2-1\end{Bmatrix} = \begin{Bmatrix}2\\3\\0\\1\end{Bmatrix}$$

After processing element 2 (the smallest dof = MINDOF = 3), we have

$$ICOLH\begin{pmatrix}5\\6\\3\\4\end{pmatrix} = \begin{Bmatrix}5-3\\6-3\\3-3\\4-3\end{Bmatrix} = \begin{Bmatrix}2\\3\\0\\1\end{Bmatrix} \quad \begin{matrix}\rightarrow 2\\ \rightarrow 3\end{matrix} \left\{\begin{matrix}\textit{previous values were kept, see IF statement on line 23 of algorithm given in Table 2.1}\end{matrix}\right\}$$

After processing element 3 (the smallest dof = MINDOF = 1), we have

$$ICOLH \begin{pmatrix} 9 \\ 10 \\ 3 \\ 4 \\ 1 \\ 2 \\ 7 \\ 8 \end{pmatrix} = \begin{Bmatrix} 9-1 \\ 10-1 \\ 3-1 \\ 4-1 \\ 1-1 \\ 2-1 \\ 7-1 \\ 8-1 \end{Bmatrix} = \begin{Bmatrix} 8 \\ 9 \\ 2 \\ 3 \\ 0 \\ 1 \\ 6 \\ 7 \end{Bmatrix}$$

After processing element 4 (the smallest dof = MINDOF = 3), we have:

$$ICOLH \begin{pmatrix} 11 \\ 12 \\ 5 \\ 6 \\ 3 \\ 4 \\ 9 \\ 10 \end{pmatrix} = \begin{Bmatrix} 11-3 \\ 12-3 \\ 5-3 \\ 6-3 \\ 3-3 \\ 4-3 \\ 9-3 \\ 10-3 \end{Bmatrix} = \begin{Bmatrix} 8 \\ 9 \\ 2 \\ 3 \\ 0 \\ 1 \\ 6 \\ 7 \end{Bmatrix} \begin{matrix} \\ \\ \\ \\ \rightarrow 2 \\ \rightarrow 3 \\ \rightarrow 8 \\ \rightarrow 9 \end{matrix}$$

It should be noticed here that after processing all 4 elements (using the algorithm presented in Table 2.1), the final updated column height array ICOLH (-) will have the corrected values as indicated earlier in Figure 2.2 and Eq. (2.22).

2.11 Summary

In this chapter, various storage schemes for storing the coefficient stiffness matrix have been discussed. The detailed algorithm and computer implementation of the column-by-column (skyline) storage scheme has also been presented, since this skyline storage scheme will lead to the development of efficient parallel-vector skyline equation solver in Chapter 4. The variable bandwidth storage scheme will be further discussed in Chapter 5, since it has direct impact on the development of efficient parallel-vector variable bandwidth equation solver. The use of sparse storage scheme will be discussed with more details in Chapter 10, where efficient sparse equation solver will be presented.

2.12 Exercises

2.1 Given the following 9x9 symmetrical stiffness matrix [A]:

A =

1.	10.							
	2.	11.						
		3.		12.	13.			
			4.	14.				
				5.		15.		
					6.	16.		
S	Y	M.				7.		17.
							8.	18.
								9

(a) Using the 1-D array B(-) to store the above matrix A in a "variable banded" fashion (Hint: read section 2.4) ??

(b) How many "real" words of computer memory required by the above 1-D array B(-) to store the given matrix [A] ??

2.2 Using the data, shown in Problem 2.1, for matrix [A]:

(a) Using the 1-D array C(-) to store the matrix [A] in a "skyline" fashion??

(b) How many "real" words of computer memory required by the above 1-D array C(-) ??

(c) Construct the integer (column heights) array ICOLH (-) for this example??

(d) Construct the integer (diagonal locations) array MAXA(-) for this example ??

2.3 Using the data, shown in Problem 2.1, for matrix [A]

(a) Using the 1-D arrays D(-) to store the matrix [A] in a "sparse" fashion??

(b) Construct the 1-D arrays IA(-), JA(-), D(-) and AN(-) for this example (Hint: see Eqs. 2.6-2.9) ??

(c) How many "real" words of computer memory required by the above 1-D array AN(-) ??

2.13 References

2.1 K.J. Bathe, Finite Element Procedures, Prentice-Hall (1996)

2.2 T.R. Chandrupatla and A.D. Belegundu, Introduction to Finite Elements In Engineering, Prentice-Hall (2nd Edition, 1997)

2.3 O.O. Storaasli, D.T. Nguyen and T.K. Agarwal, "A Parallel-Vector Algorithm for Rapid Structural Analysis on High Performance Computers," NASA-TM-102614 (March 1990)

2.4 O.O. Storaasli, D.T. Nguyen and T.K. Agarwal, "The Parallel Solution of Large-Scale Structural Analysis Problems on Supercomputers,' AIAA Journal, Vol. 28, No. 7, pp. 1211-1216 (July 1990).

2.5 A. George and J.W. Liu, Computer Solution of Large Sparse Positive Definite System, Prentice-Hall (1981).

3 Parallel Algorithms for Generation and Assembly of Finite Element Matrices

3.1 Introduction

The solution of simultaneous linear equations or eigenvalue equations can be considered as a major component of many existing finite element codes, since it represents a major fraction of CPU time for the solution process in statics, free vibration, transient response, structural optimization, and control structure interaction (CSI) of large-scale, flexible space structures. Researchers are endeavoring to develop efficient parallel algorithms for solving large systems of linear equations, eigenvalue equations, and much progress has been recently reported in the literature [3.1-3.7].

Since the time for solving large system of linear equations has been reduced significantly by using the recently developed parallel-vector equation solvers [3.1-3.7], generating (complicated) element (stiffness and mass) matrices and assembling the total (structural) matrices may now represent a significant amount of the total CPU time for many practical engineering applications [3.4, 3.8-3.9]. This is especially true for nonlinear structural analysis [3.8, 3.10], structural optimization and CSI [3.11, 3.12], since in these application, new element matrices need to be generated and assembled in an iterative procedure.

The objective of this chapter is to develop algorithms for parallel generation and assembly of element matrices which exploit advanced computer with multiple processors (such as Cray-C90, Cray-J90, Cray-T90, Cray SV1, Intel Paragon, Mieko, and IBM-SP2). The derivation of the new methods are presented, and practical examples are given to demonstrate the effectiveness of the new methods.

3.2 Conventional Algorithm to Generate and Assemble Element Matrices

In this section, a conventional assembly procedure for generating and assembling element matrices is described. To facilitate the discussions, a simple three-bar truss structure, shown in Figure 3.1, is used in this section.

In Figure 3.1, each (truss) element stiffness matrix has the dimension of a 4-by-4 matrix, since each truss element is connected by 2 nodes (node i and node j), and each node has 2 degree-of-freedom (dof). For example, element ② is connected by node i =1 and node j=3. Also, the 4 dof associated with element ② are u_1, u_2, u_5, and u_6.

The total (or structural) stiffness matrix has the dimension of a 6-by-6 matrix, since the structural stiffness matrix has a total of 6 dof (u_1, u_2, ..., u_6).

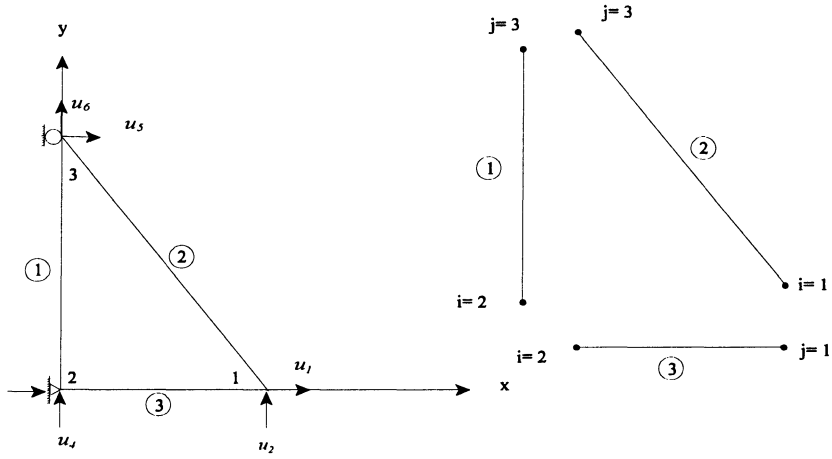


Figure 3.1 Three bar truss example

In the conventional procedure, the element stiffness (or mass) matrices are generated and their contributions to the total (structural) stiffness (or mass) matrix can be obtained in a form of a pseudo-Fortran code, as shown in Table 3.1

Table 3.1 Conventional generation and assembly structural stiffness matrix

DO 1 e = 1, 3 elements
• generate element stiffness matrix $[k^{(e)}]$
• add contribution of $[k^{(e)}]$ to structural matrix $[K] = \Sigma\, k^{(e)}$
1 Continue

Upon executing the algorithm shown in Table 3.1, each of the (4 by 4) element stiffness matrix $[k^{(e)}]$ will have its contribution to the (6 by 6) total (or structural) stiffness matrix [K], as shown in Eq. (3.1)

$[K]_{6x6}$ =

	1	2	3	4	5	6
1	②③	②③	③	③	②	②
2		②③	③	③	②	②
3			①③	①③	①	①
4				①③	①	①
5	S	Y	M.		②①	②①
6						②①

(3. 1)

In Eq. (3.1), only the upper triangular portion of the structural stiffness matrix [K] is presented, since matrix [K] is symmetrical and positive definite. At this stage, it is only important to see the contributions of each element stiffness matrix into the total

stiffness matrix [K]. Thus, only the element numbers (shown in circles) are shown in Eq. (3.1), and their actual numerical values are NOT shown in Eq. (3.1).

Referring to Figure 3.1, one can clearly see that elements ① and ③ both have contributions to node 2 (or degree-of-freedoms u_3 and u_4), while elements ② and ③, for example, both have contributions to node 1 (or degree-of-freedoms u_1 and u_2). These observations are also reflected in Eq. (3.1).

In a parallel computer environment, however, synchronization is a problem. This can be demonstrated by the simple 3-bar truss of Figure 3.1. In Table 3.1, if each e^{th} finite element is assigned to a separate processor, then at node 1 (see Figure 3.1), for example, both elements ②and ③ will often need to write their element stiffness contribution to the structural stiffness matrix at the same locations, simultaneously! In other words, generating all three element stiffness matrices (for elements ①, ② & ③) simultaneously and independently will be a trivial task. Assembling these element stiffness matrices simultaneously, however, do have complications and problems!

This synchronization problem can be partially overcome by either setting the local locks in the common shared memory pool, or by special numbering of the elements throughout the entire structure [3.10]. The first method, setting the local locks, reduces the speed of parallel assembly considerably. The second method (special numbering of the elements), while improving the speed of parallel assembly, is not general since the method will not work if a large number of processors are used and the number of substructures is small.

To alleviate the above synchronization problem, new alternative methods are proposed and discussed in the following sections.

3.3 Node-By-Node Parallel Generation and Assembly Algorithms [3.8-3.9]

In this new algorithm, element matrices will be generated and assembled in a node-by-node fashion. For a two-node (node i and node j) truss element (refer to Figure 3.1), for example, a two dimensional, 4x4 element stiffness matrix $[k^{(e)}]$ can be symbolically represented as:

$$[k^{(e)}] = \begin{bmatrix} k_{ii}^{(e)} & k_{ij}^{(e)} \\ k_{ji}^{(e)} & k_{jj}^{(e)} \end{bmatrix} \qquad \textbf{(3.2)}$$

In Eq. (3.2), $k_{ii}^{(e)}$ and $k_{jj}^{(e)}$ refer to the 2x2 sub-matrices which represent a portion of an element stiffness matrix attached to node i and node j, respectively. The coupling effect between nodes i and j of an element stiffness matrix $[k^{(e)}]$ is represented by a 2x2 sub-matrix $k_{ij}^{(e)}$ (or its transpose 2x2 sub-matrix $k_{ji}^{(e)}$). Using the three-bar truss structure (shown in Figure 3.1) as a simple illustration, the new node-by-node algorithm can be described in the following step-by-step procedure:

Step 1: Element Connectivity Data
The standard element connectivity data of the three-bar truss structure (see Figure 3.1) can be readily obtained as shown in Table 3.2.

Table 3.2 Element connectivity

Element Number	Node-i	Node-j
1	2	3
2	1	3
3	2	1

In general, a truss element is connected by nodes i and j. The selection of nodes i and j are arbitrary.

Step 2: Node Connectivity Information
In this step, elements which are attached to nodes i and nodes j of the entire structure need to be identified. The global degree-of-freedom (dof) associated with each node can be readily identified. For the three-bar truss structure shown in Figure 3.1, the node connectivity information can be generated as shown in Table 3.3

Table 3.3 Node connectivity for 2-D truss elements

Node Number	Global dof	Elements with "Node Number" Node i	Node j
1	1 2	2	3
2	3 4	1,3	None
3	5 6	None	1,2

In Table 3.3, element 2 for example, appears in the last 2 columns because element 2 is connected by node i = 1 and node j = 3. Similarly, element 1 also appears in the last 2 columns because element 1 is connected by node i = 2 and node j = 3. It should be emphasized here, however, there are no elements attached to node j = 2. Similarly, there are no elements attached to node i = 3. This step represents an additional overhead cost of the proposed new algorithm, since the information generated in this step is usually not required in conventional finite element codes. In actual computer implementation, the additional cost (in terms of computer CPU time), however, has been found to be negligible.

Step 3: Parallel Generation and Assembly (G&A) of Element Stiffness Sub-matrix $k_{ii}^{(e)}$ for Each Node of a Structure

Considering a typical truss member, such as truss member 2 as shown in Figure 3.1, this

member is connected by 2 nodes (node i = 1, and node j = 3). The four dof associated with this member are u_1, u_2 (associated with node i) and u_5 & u_6 (associated with node j). The element stiffness matrix $[k^{(e)}]$ for this 2-dimensional truss is a 4 x 4 symmetrical matrix, which can be partitioned as (refer to Eq. 3.2).

$$[k^{(e=2)}]_{4x4} = \begin{array}{c} \\ u_1 \\ u_2 \\ u_5 \\ u_6 \end{array} \begin{array}{c} \begin{array}{cc} \textit{node i} & \textit{node j} \\ u_1 \; u_2 & u_5 \; u_6 \end{array} \\ \left[\begin{array}{cc} k_{ii}^{(e)} & k_{ij}^{(e)} \\ \text{-------} & \\ k_{ji}^{(e)} & k_{jj}^{(e)} \end{array}\right] \end{array} \begin{array}{l} \\ \text{node i} \\ \\ \text{node j} \end{array} \qquad \textbf{(3.3)}$$

In this step, the portion $K_{ii} = \sum_e k_{ii}^{(e)}$ of the structural (or global) stiffness (or mass) matrix K is generated and assembled in a parallel computer environment. From the nodal connectivity information generated in the previous step, each node can be assigned to a separate processor. Thus, in the three-bar truss structure (see Figure 3.1), node 1 will be assigned to processor 1. Processor 1, therefore will generate $k_{ii}^{(e)}$ portions of element e=2 (see the 3rd column of Table 3.3), and add the contribution to appropriate locations (dof 1 and 2). Simultaneously, node 2 is assigned to processor 2 which will generate $k_{ii}^{(e)}$ portions of elements e=1 and 3 in a sequential fashion, and add its contribution to appropriate locations (dof 3 and 4) of the structural stiffness matrix K. At the same time, processor 3 is assigned to node 3 (associated with dof 5 and 6). In this particular three-bar truss example, processor 3 is idle, as there are no elements with node i=3 (refer to the 3rd column of Table 3.3). The parallel generation and assembly (G & A) of $K_{ii} = \sum_e k_{ii}^{(e)}$ for each structural node can be represented as shown in Table 3.4.

Table 3.4 Parallel generation and assembly of $k_{ii}^{(e)}$

node i = 1,	i = 2,	i = 3				
	1	2	3	4	5	6
1	②	②				node i = 1,
2		②				
3			①③	①③		node i = 2,
4				①③		
5						node i = 3,
6						

It is important to realize that in Table 3.4, processors 1 and 2 simultaneously generate their own contributions to different locations of K_{ii}. There is no overlapping between processor 1 (which is assigned to dof 1 and 2), processor 2 (which is assigned to dof 3 and 4, and processor 3 (which is assigned to dof 5 and 6). Thus, in this step, parallel G & A computation can be done without any communication among

processors. The actual numerical values of K_{ii} are NOT given, since they are NOT important in this discussion. Thus, only the element numbers which have contribution to K_{ii} are given (see numbers in circle of Table 3.4). In this particular example (refer to Figure 3.1), work-balancing is not good, since processors 1, 2 and 3 have to process 1, 2, and 0 element stiffness matrices, respectively.

As will be seen later in this chapter, however, for real, practical, large-scale structural problems, the work-balancing among processors are quite good, and therefore excellent parallel speed up factors can be achieved.

Step 4: Parallel Generation and Assembly (G & A) of Element Stiffness Sub-matrix $k_{jj}^{(e)}$ for Each Node of a Structure.

In this step, the portion $K_{jj} = \sum k_{jj}^{(e)}$ of the structural stiffness matrix K is generated and assembled in a parallel computer environment. Each node is again assigned to a separate processor, and the information in the last column of Table 3.3 is used here. Processor 1, which is assigned to node 1, will generate $k_{jj}^{(e)}$ of element e=3, and add its contribution to appropriate locations (dof 1 and 2) of the structural stiffness matrix [K]. Simultaneously, processor 3 which is assigned to node 3, will generate $k_{jj}^{(e)}$ of elements e = 1 and 2, and add its contribution to appropriate locations (dof 5 and 6) of the structural stiffness matrix [K]. At the same time, processor 2 is assigned to node 2 (associated with dof 3 and 4). In this particular example (refer to Figure 3.1), processor 2 is idle, since there are no elements with node j =2. The parallel G & A of $k_{jj}^{(e)}$ for each structural node is shown in Table 3.5

Table 3.5 Parallel generation & assembly of $k_{jj}^{(e)}$

	1	2	3	4	5	6	
1	③	③					node j = 1
2		③					
3							node j = 2
4							
5					①②	①②	node j = 3
6						①②	

Step 5: Parallel Generation & Assembly (G & A) of Element Stiffness Sub-matrix $k_{ij}^{(e)}$ for Each Node of a Structure

In this step, the portion $K_{ij} = \sum k_{ij}^{(e)}$ of the structural stiffness matrix K is generated and assembled in a parallel computer environment. To find out what elements are attached to a given node of a structure, information on either node i (see the 3rd column of Table 3.3) or node j (see the 4th column of Table 3.3) can be used to generate the portion K_{ij} of the structural stiffness matrix K.

In this section, the information for nodes j is used in this step. Thus, processor 1 is assigned to node 1 to process element e = 3. Element 3 is connected to dof 1, 2, 3 and 4, and its contribution to K_{ii} and K_{jj} have already been done in step 3 and step 4, respectively.

In this step, processor 1 will generate $k_{ij}^{(e=3)}$ and add its contribution to the appropriate locations of K_{ij}. Simultaneously, processor 3 is assigned to node 3 to

process elements 1 and 2. Processor 3 will therefore, generate $k_{ij}^{(e)}$ for elements e = 1 and 2, and add its contribution to the appropriate locations of K_{ij}. In this particular example, processor 2 is idle since there are no elements with node j = 2. The parallel G & A of $k_{ij}^{(e)}$ for each structural node is conveniently represented in Table 3.6.

Table 3.6 Parallel generation & assembly of $k_{ij}^{(e)}$

	1	2	3	4	5	6
1			③	③	②	②
2			③	③	②	②
3					①	①
4					①	①
5						
6						

Since the structural stiffness matrix K is symmetric, only the upper-half of K_{ii}, K_{jj} and K_{ij} are considered in step 3, step 4 and step 5, respectively.

The above five-step procedure to generate and assemble element stiffness matrices in parallel is quite general, since there is no assumption on the type of element used in the finite element model. For a more convenient and efficient computer implementation, the execution in step 5, for the coupling terms $k_{ij}^{(e)}$ and $k_{ji}^{(e)}$, can be and should be included in steps 3 and 4. Thus, the overhead cost due to the re-calculation of some parameters for generating the element stiffness matrix can be reduced.

The actual computer implementation of Baddourah-Nguyen's algorithm for generation and assembly of two-dimensional truss elements (see Figure 3.1, and Table 3.3) can be shown with a "pseudo-Fortran" code in Table 3.7. The variable NEL(n), shown in Table 3.7, represents the number of elements to be processed by the n^{th} processor.

Table 3.7 Actual computer implementation for G & A of two-dimensional truss elements

For each n^{th} processor:

DO 1 e = 1, NEL (n)

c. . . . . Step 3 and Step 5 combined

c. . . . . Generate & Assembly $k_{ii}^{(e)}$ and $k_{ij}^{(e)}$ (where j > i)

	1	2	3	4	5	6
1	②	②			②	②
2		②			②	②
3						
4						
5						
6						

Processor 1

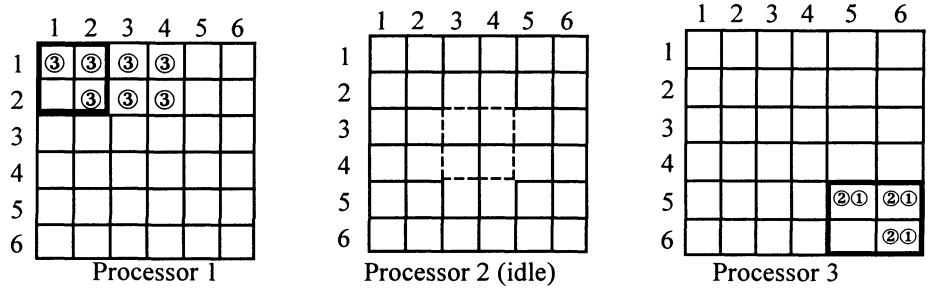
	1	2	3	4	5	6
1						
2						
3			①③	①③	①	①
4				①③	①	①
5						
6						

Processor 2

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

Processor 3 (idle)

1 Continue
DO 2 e=1, NEL (n)
c. . . . Step 4 and Step 5 combined
c. . . . Generate & Assembly $k_{jj}^{(e)}$ and $k_{ji}^{(e)}$ (where i> j)



2 Continue
For a three-node triangular element (with 2 translational dof at each node, refer to Figure 3.2), a two dimensional, 6x6 element stiffness matrix $k^{(e)}$ can be symbolically represented as:

$$k^{(e)} = \begin{bmatrix} k_{ii}^{(e)} & k_{ij}^{(e)} & k_{im}^{(e)} \\ & k_{jj}^{(e)} & k_{jm}^{(e)} \\ & & k_{mm}^{(e)} \end{bmatrix} \tag{3.4}$$

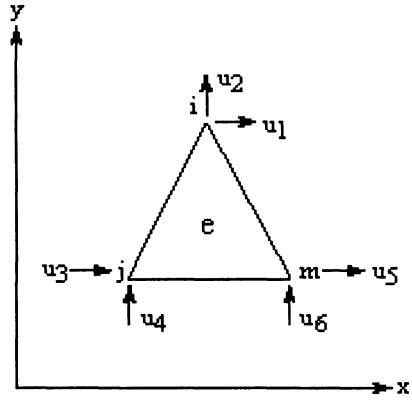


Figure 3.2 Three-node triangular element

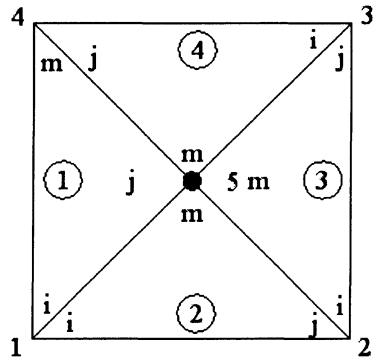


Figure 3.3 Triangular elements

In Eq. (3.4), $k_{ii}^{(e)}$, $k_{jj}^{(e)}$ and $k_{mm}^{(e)}$ refer to the 2x2 sub-matrices which represent a portion of an element stiffness matrix attached to node i, node j and node m, respectively. The coupling effect between nodes i, j and m of an element stiffness matrix $k^{(e)}$ is represented by the sub-matrices $k_{ij}^{(e)}$, $k_{im}^{(e)}$ and $k_{jm}^{(e)}$. Thus, for a three-node triangular element, an additional step needs to be inserted before the last step (step 5) for parallel generation and assembly of $k_{mm}^{(e)}$ for each node m of the structure. As an illustrative example, a four triangular element structure is shown in Figure 3.3. The corresponding node connectivity for this structure is shown in Table 3.8 (Similar to Table 3.3)

Table 3.8 Node connectivity for 2-D triangular elements

Processor (or Node) Number	Global dof	Elements with "Node Number"		
		Node i	Node j	Node m
1	1, 2	①,②	None	None
2	3, 4	③	②	None
3	5, 6	④	③	None
4	7, 8	None	④	①
5	9, 10	None	①	②,③,④

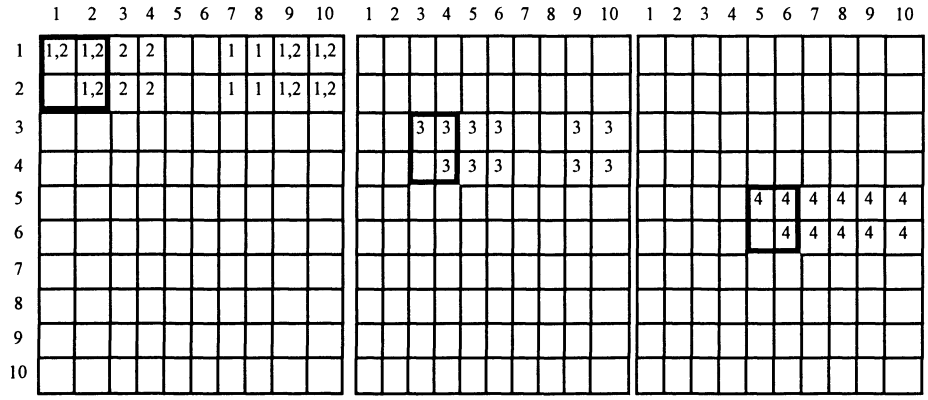
The actual computer implementation of Baddourah-Nguyen's algorithm for generation and assembly of two-dimensional triangular elements (see Figure 3.3 and Table 3.8) can be shown with a "pseudo-Fortran" code in Table 3.9.

Table 3.9 Actual computer implementation for G&A of two-dimensional triangular elements

For each n^{th} processor:
DO 1 e=1, NEL (n)

c.... Generate & Assembly $k_{ii}^{(e)}$, $k_{ij}^{(e)}$ and $k_{im}^{(e)}$ (where j, m > i)

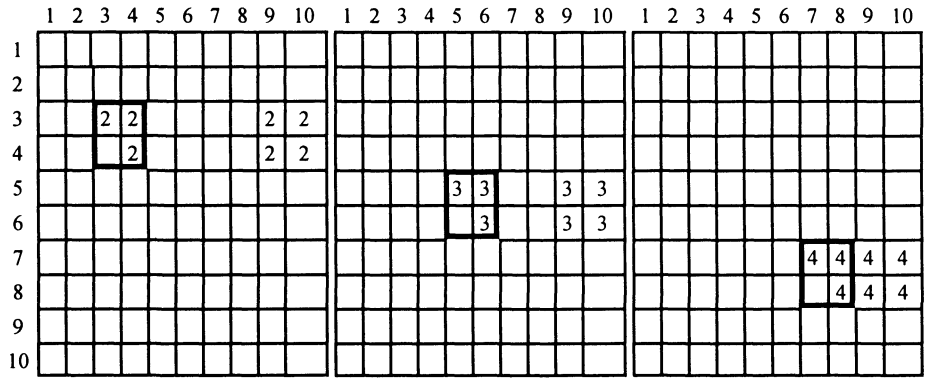
c... Processors 4 and 5 are idle

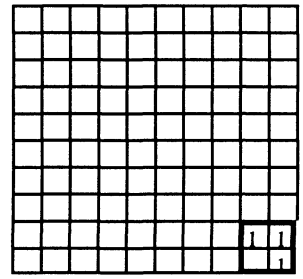


1 continue
DO 2 e=1, NEL (n)

c.... Generate & Assembly $k_{jj}^{(e)}$, $k_{jm}^{(e)}$ and $k_{ji}^{(e)}$ (where m, i > j)

c... Processor 1 is idle



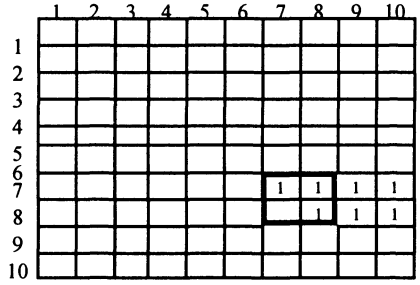


Processor 5

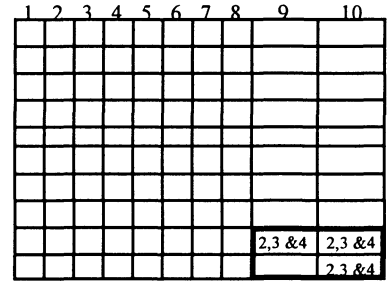
2 continue
DO 3 e = 1, NEL (n)

c.... Generate & Assemble $k_{mm}^{(e)}$, $k_{mi}^{(e)}$ and $k_{mj}^{(e)}$ (where i, j > m)

c..... Processors 1, 2 and 3 are idle



Processor 4



Processor 5

3 continue

3.4 Additional Comments on Baddourah-Nguyen's (Node-by-Node) Parallel Generation and Assembly (G&A) Algorithm

In general, the overhead cost is increased with elements which have more nodes attached to them. For example, examining Table 3.3 carefully (2-D truss elements), one can see that the proposed G&A algorithm requires generating each element stiffness matrix TWICE (refer to the 3rd and 4th columns of Table 3.3). However, due to perfect parallel computation (without any communication required) of three processors used, the "net gain" in parallel speed-up is still $\frac{3}{2}$ = 1.5. This parallel speed-up will be significantly better for large finite element model with larger number of processors.

Similar observation in Table 3.8 (2-D triangular elements) will reveal that each element stiffness matrix is generated THREE TIMES (refer to 3rd, 4 th and 5 th columns of Table 3.8). However, since 5 processors have been used (without any communication between processors), the "net gain" in parallel speed-up is still $\frac{5}{3}$. This parallel speed-up will be scalable for large finite element model with larger number of processors.

Practical finite element models have a large number of degree-of-freedom. For these large models, many finite elements need to be processed. Furthermore, there are fewer processors available as compared to the number of elements in a large finite element model. Thus, a balance of the work load among the processors is well preserved. The idle time of the processors, therefore, is significantly reduced through this generation and assembly process.

3.5 Applications of Baddourah-Nguyen's Parallel G & A Algorithm

The proposed parallel algorithm [3.9] for G & A the element stiffness matrices has been coded using a parallel Fortran language FORCE [3.13] (FORtran with Concurrent Extension). It should be emphasized here that FORCE is used merely for convenient purpose, other parallel software language such as PVM, or MPI (Message Passing Interface) can also be used for parallel implementation of the proposed G & A algorithm. Four structural examples were used to evaluate the numerical performance of the new algorithm. In all examples considered in this section, elapsed (or wall clock) time in a multiuser (non-dedicated) computer environment are reported. The times given include the overhead cost in step 2 (of Section 3.2), and the G & A of the structural stiffness matrix. The computation speed-up is defined in most parallel algorithms as

$$speed\text{-}up = \frac{time\ for\ 1\ processor}{time\ for\ N\ processsor} \tag{3.5}$$

While parallel implementation of the "conventional" method for G & A of the structural stiffness matrix has not completely resolved the synchanization problem [3.10], this new parallel G & A algorithm is quite general, and a significant reduction in elapsed time has been observed (see Tables 3.10 → 3.13) when multiple processors are used.

Even better speed-up factors can be expected in all these examples in a dedicated computer environment, where the required processors are NOT shared by different users.

Example 1: Three-Dimensional Truss Structure
A pattern of a three-dimensional truss structure is shown in Figure 3.4. The 5 story x 150 bay structure has 3416 three-dimensional truss elements, 612 nodes with 3 dof per node. The elapsed time for this example using the new algorithm is presented in Table 3.10.

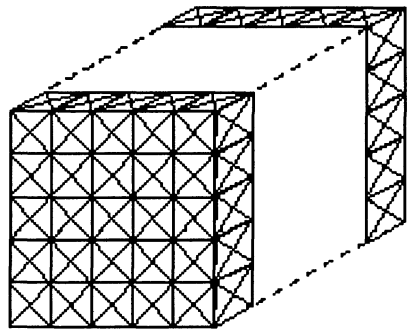


Figure 3.4 Three-dimensional pattern of truss structure

Table 3.10 3-D Truss: speed-up on Cray Y-MP

Number of Processors	Generate/Assemble Stiffness Matrix (sec)	Speed-up
1	0.1057	1.0000
2	0.0544	1.9418
4	0.0285	3.7052
6	0.0205	5.1619

For this example, the elapsed time using the conventional algorithm (on a single processor) is 0.0855 seconds. As can be expected on a single processor, the new method is a little slower than the conventional method. This is due to the overhead cost (refer to Step 2) of the new algorithm. The power of the new method is fully realized on massively parallel MIMD computers where the overhead becomes negligible.

Example 2: Two-Dimensional Frame Structure
A pattern of a two-dimensional frame structure is shown in Figure 3.5.

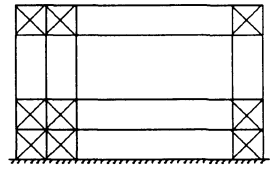


Figure 3.5 Two-dimensional frame structure

The 5 story x 150 bay structure has 1005 frame elements, 306 nodes with three DoF per node. The elapsed time for this example using the new algorithm is presented in Table 3.11.

Table 3.11 2-D Frame: speed-up on Cray Y-MP

Number of Processors	Generate/Assemble Stiffness Matrix (secs)	Speed-up
1	0.0295	1.0000
2	0.0152	1.9390
4	0.0083	3.5502
6	0.0071	4.1541

For this example, the elapsed time using the conventional algorithm (on a single processor) is 0.0242 seconds.

Example 3: Two Dimensional Plate Structure

A pattern of a plate which is modeled by three-node triangular elements is shown in Figure 3.6.

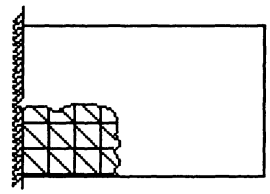


Figure 3.6 Two-dimensional plate structure

The 5 story x 150 bay structure has 1500 triangular elements, 906 nodes with two DoF per node. The elapsed time for this example is presented in Table 3.12.

Table 3.12 2-D Plate: speed-up on Cray Y-MP

Number of Processors	Generate/Assemble Stiffness Matrix (secs)	Speed-up
1	0.0607	1.0000
2	0.0305	1.9909
4	0.0164	3.6990
6	0.0121	5.0362

For this example, the elapsed time using the conventional algorithm (on a single processor) is 0.0400 seconds. The overhead cost (0.0607 seconds -0.0400 seconds) for this example is larger than in the preceding examples. This is expected since the 3-node plate element used requires more computations than the simpler 2-node truss and 2-node

frame elements. Furthermore, the amount of redundant (or overhead) works in evaluating element stiffness matrices, in general, is increased with increasing number of nodes (or dof) per element.

Example 4: Three-Dimensional Beam Finite Element Model
A three-dimensional beam finite element antenna model to study Control-Structure Interaction (CSI) [3.11] is shown in Figure 3.7.

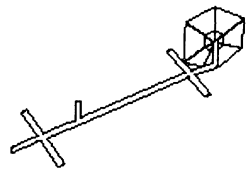


Figure 3.7 Three-dimensional CSI antenna model

The structure has 1647 beam elements, 537 nodes with six DoF per node. The elapsed time for this example is presented in Table 3.13.

Table 3.13 CSI speed-up on Cray Y-MP

Number of Processors	Generate/Assemble Stiffness Matrix (secs)	Speed-up
1	1.3396	1.0000
2	0.6927	1.9339
4	0.3711	3.6098

For this example, the elapsed time using the conventional algorithm (on a single processor) is 1.0989 seconds.

In all the above examples (see Tables 3.10-3.13), the elapsed time is decreased with increasing number of processors. For linear structural analysis, the generation and assembly of the structural stiffness matrix need to be done only once. Thus, more significant time-saving can be expected in nonlinear structural analysis, structural optimization and control-structure interaction where the element matrices are updated and assembled repeatedly.

3.6 Qin-Nguyen's G & A Algorithm

In this section, a massively parallel G & A of element stiffness matrices is developed for large-scale structural analysis on massively parallel computers with distributed-memory (such as the Intel Gamma, Delta, Paragon, Mieko, IBM-SP2). The same algorithm can also be applied for shared memory computers (such as Cray YMP, Cray C-90). A block-skyline column storage scheme is used to enhance the vector performance of the equation solver (to be discussed in Chapter 7) and to reduce the

memory demand for each processor. This column storage scheme can also be used to improve the parallel performance of generation and assembly of element stiffness matrices. The parallel computers used in this section are the Intel iPSC/860 (such as the Gamma computer with 128 processors and the Delta computer with 512 processors). Both computers have the same floating-point operation speed, while the Delta machine has a higher communication rate as compared to that of the Gamma (say 3 to 1 ratio, for matrices with an average half-bandwidth around 1000).

It is important to emphasize here that any proposed G & A algorithms should be compatible with the equation solvers to be developed in subsequent chapters.

For illustrative purpose, let's consider a two-dimensional frame structure as shown in Figure 3.8. This 2-D frame structure has 16 nodes, each node has 3 degree-of-freedom (2 translational dof in the x & y directions, and 1 rotational dof about the z-direction). Nodes 13-16 have no degree-of-freedoms since these nodes are constrained by the support boundary conditions. There are 21 frame finite elements, each element is connected by 2 nodes, hence each element has 6 dof, and the entire structure has 36 dof.

To simplify the discussions, assuming there are 3 processors available, thus each processor can be assigned to store block columns of the total stiffness matrix [K], as shown in Figure 3.9 (where the block size k = 4 is shown). In real computer implementation, however, block size k = 8 is selected, since this block size will give near optimum performance during G & A and equation solution phases (to be discussed with more details in Chapter 7). Thus column numbers (or dof numbers) 1-4, 13-16 and 25-28 belong to processor 1 (or P_1), column numbers 5-8, 17-20 and 29-32 belong to processor 2 (or P_2). Similarly, column numbers 9-12, 21-24 and 33-36 belong to processor 3 (or P_3).

The dof associated with each finite element are known, once the finite element model (see Figure 3.8) has been defined. This information is shown in the first 2 columns of Table 3.14. Since each dof (or each column) can be mapped into a particular processor (refer to Figure 3.9), element number to processor number(s) mapping can be easily established as shown in the 3rd column of Table 3.14.

Once the information shown in Table 3.14 is known, the mapping from processor number to element number(s) can be readily identified as shown in Table 3.15.

In fact, once the dof associated with each element are known (refer to the 2nd column of Table 3.14), Table 3.15 can be easily obtained in a parallel fashion, as indicated in the following "pseudo-FORTRAN" coding (refer to Table 3.16).

Table 3.16 Parallel algorithm to find which elements belong to each processor

1	For each processor i (where i = processors 1, 2 & 3)
2	DO 1 K = 1, ALLELS (say K=7)
3 c....	Get the dof, ELDOF(-) array, associated with the K^{th} element
4 c...	Get ELDOF(1, 2, 3, 4, 5, 6) = 25, 26, 27, 28, 29, 30
5	DO 2 M = 1, 6
6	If [ELDOF(M) belongs to processor i] Then
7	. Record element K belongs to processor i
8	. Exit loop 2, and go to loop 1
9	Endif
10	2 Continue
11	1 Continue

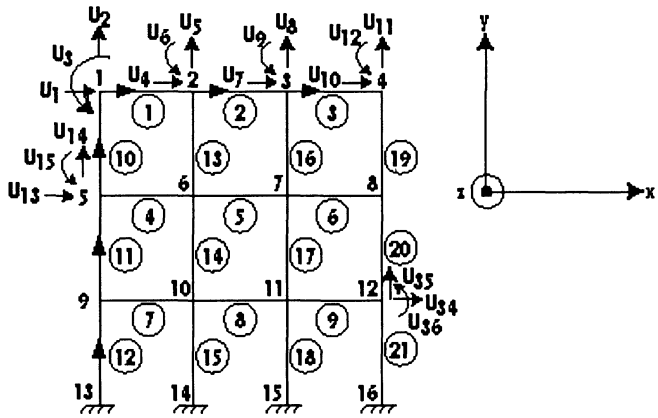


Figure 3.8 A 2-D frame structure

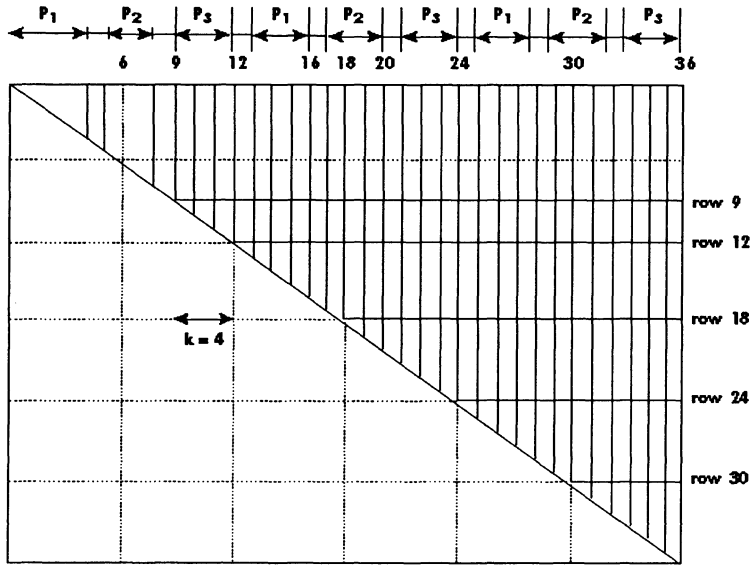


Figure 3.9 Block columns (skyline) storage scheme for Qin-Nguyen's G & A algorithm

Table 3.14 Degree-of-freedom and processor number associated with each finite element

Frame Element Number	Frame Element DOF Numbers	Processor Number
1	1, 2, 3, 4, 5, 6	P_1, P_2
2	4, 5, 6, 7, 8, 9	P_1, P_2 P_3,
3	7, 8, 9, 10, 11, 12	P_2, P_3
4	13,14,15,16,17,18	P_1, P_2
5	16,17,18,19,20,21	P_2, P_3
6	19,20,21,22,23,24	P_2, P_3
7	25,26,27,28,29,30	P_1, P_2
8	28,29,30,31,32,33	P_1, P_2 P_3
9	31,32,33,34,35,36	P_2, P_3
10	13,14,15,1,2,3	P_1
11	25,26,27,13,14,15	P_1
12*	0,0,0,25,26,27	P_1
13	16,17,18,4,5,6	P_1, P_2
14	28,29,30,16,17,18	P_1, P_2
15*	0,0,0,28,29,30	P_1, P_2
16	19,20,21,7,8,9	P_2, P_3
17	31,32,33,19,20,21	P_2, P_3
18*	0,0,0,31,32,33	P_2, P_3
19	22,23,24,10,11,12	P_3
20	34,35,36,22,23,24	P_3
21	0,0,0,34,35,36	P_3

Table 3.15 Frame elements associated with each processor

Processor Number	Frame Element Numbers	Number of Elements
1	1,2,4,7,8, 10-15,	11
2	1-9, 13-18	15
3	2,3,4,5,8,9, 16-21	12

In Table 3.16, variable ALLELS (on line 2) refers to the total number of finite elements. For the example shown in Figure 3.8, ALLELS =21 (also refer to Table 3.14). The if statement (on line 6 of Table 3.16) will assure that an element will not be recorded more than once by the same processor.

Each processor will scan through all finite elements (refer to the "do loop" on line 2). The dof associated with the K^{th} element are known, and are given by the integer array ELDOF (M), where M = 1 through 6 (since each 2-D frame element has 6 dof). Once the dof are known, the corresponding processor number can be identified (for example, by referring to the information shown in Figure 3.9).

Assuming the value of K (on line 2) is 7. All three processors (P_1, P_2 and P_3) will examine the 6 dof (25, 26, 27, 28, 29, 30) associated with frame element number 7 (refer to lines 4 and 6). Upon exiting from loop 2 (refer to line 10), element 7 will be recorded by processor 1 (since element 7 contains dof 25, 26, 27 and 28), and also will be recorded by processor 2 (since element 7 also contains dof 29 and 30). Element 7, however, will NOT be recorded by processor 3 (since all dof 25-30 are NOT belonging to processor 3, refer to Figure 3.9).

The parallel G & A algorithm proposed by Qin & Nguyen [3.4] can be summarized in Table 3.17.

Table 3.17 Qin-Nguyen's parallel G & A algorithm

```
1       For each processor i (where i=processors 1, 2 & 3)
2       DO 1 K = 1, NEL ( i)
3   c....  Get the 6 dof associated with the K'h element, ELDOF (M)
4.  c... Where M = 1, 2, 3, 4, 5 and 6
5       • Generate element (stiffness) matrix of the Kth element
6       • Assemble the entire (or just a portion of) stiffness
7         matrix of the Kth element into a global matrix
8     1 Continue
```

In Table 3.17, NEL (i), on line 2, represents the number of elements which belong to the i^{th} processor. Based upon the data presented in Table 3.15, then NEL (1) = 11, NEL (2) = 15 and NEL (3) = 12. The first element of processors 1, 2 and 3 is element 1, element 1 and element 2, respectively. Similarly, from Table 3.15, the last element of processors 1, 2 and 3 is element 15, element 18 and element 21, respectively.

Assuming processors 1, 2 and 3 are trying to simultaneously generate element

stiffness matrices (of elements 1, 1 and 2, respectively) and add (or assemble) its contribution to the appropriate locations of the total (or global) stiffness matrix K. At this stage, processor 1 (or P_1) will generate a 6 by 6 element stiffness matrix (for element 1). However, it assembles only a portion (associated with dof 1-4) of its 6x6 element matrix into the appropriate locations of the total matrix [K] as shown in Figure 3.10. At the same moment, processor 2 (or P_2) will also generate a 6 by 6 element stiffness matrix (for element 1). However, it assembles only a portion (associated with dof 5-6) of its 6x6 element matrix into proper locations of the total matrix [K]. Simultaneously, processor 3 will generate a 6 by 6 element stiffness matrix (for element 2). However, it assembles only a portion (associated with dof 9) of its 6x6 element matrix into proper locations of the total matrix [K]. From Figure 3.10, it is important to observe that there is no overlapping among processors when each processor simultaneously adds its contribution to global matrix [K].

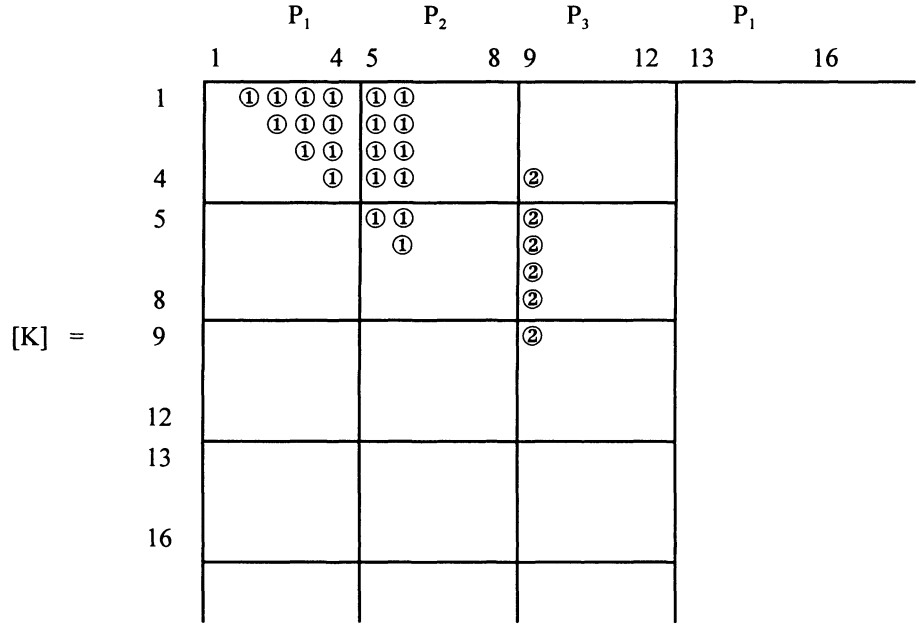


Figure 3.10 Parallel G & A of element stiffness matrices for elements 1 and 2 by processors 1, 2 and 3, respectively

The numerical values of the total (or global) stiffness matrix [K] are unimportant at this point, hence, only element numbers are shown in Figure 3.10.

3.7 Applications of Qin-Nguyen's Parallel G & A Algorithm

2-D truss structures with 'nb' bays and 'ns' stories (shown in Figure 3.11) are denoted as nb x ns. Table 3.18 gives the CPU times for the generation and assembly of the global stiffness matrix (on the Gamma computer with up to 128 processors). For the 200 x 6 model, nel = 4806, neq = 2412. For the 1 x 16500 model, nel = 82,500 and neq = 66,000.

Table 3.18 CPU times for the generation and assembly of the global matrix

nb x ns	k	1	2	4	8	16	32	64	128
200x6	4	.337	.206	.1036	.0593	.0290	.01398	.00718	.0034
200x6	8	.339	.1885	.1145	.0569	.0273	.01412	.00664	.0033
1x16500	4	4.64	3.537	1.793	.9083	.4515	.2267	.114	.057
1x16500	8	-	2.985	1.507	.7564	.376	.188	.0945	.0481

*small overhead timings for obtaining Table 3.15 is not included here.

As it can be seen from Table 3.18 that in general, larger block size k will lead to higher speedup, because there will be fewer elements that contribute to more than one processors. The price one has to pay is, however, more memory will be required for larger block size (to be discussed and explained in Chapter 7). The block size will also have the effects on the performance of the equation solver, which will be discussed in Chapter 7.

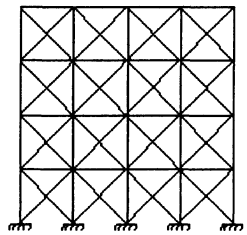


Figure 3.11 Two dimensional pattern of truss structure with "nb" bays and "ns" stories

Larger-Scale numerical examples of 2-D truss structures with 750 bays and 6 stories (with 18,006 truss elements) and 1096 bays and 41 stories (with 179,785 truss elements) are presented in Table 3.19.

Table 3.19 Performance of Qin-Nguyen G&A algorithm for large-scale truss structures

Number Processors	8	16	32	64	128	256	512
750 bays, 6 stories (Intel Gamma Computer)		0.1585	0.0806	0.0403	0.0188		
750 bays, 6 stories (Intel Delta Computer)	0.3203	0.1506	0.0785	0.0377	0.0176	0.0095	0.00463
1096 bays, 41 stories (Intel Delta Computer)						0.12251 (38.46 Mflops)	

It is interesting to note that there are a few places in Table 3.19 which indicate more than ideal speed-up factors. In this work, a processor will generate a complete element stiffness matrix even though it may need only a portion of it. This kind of a "redundant computation" may be reduced when more processors are used.

In Table 3.20 [3.5], the finite element used is an eight-node solid element with 3 d.o.f. per node, thus there are 24 d.o.f. per element. Since every 8 d.o.f. are stored in one processor, it is generally true to state that the same element should be shared by AT LEAST three (24/8=3) processors, or AT MOST (the worst case) by eight processors.

The above observations imply that for the 24 d.o.f. solid element, the speed-up for generation and assembly is AT BEST NP/3, or AT WORST NP/8. When the number of processors (NP) is small and the problem size is fixed, the "actual" speed- up may be more or less than the "predicted" range. However, when NP is large enough, the change of the speed-up will be proportional to the change of NP (i.e., when NP is doubled, the speed-up will also be doubled.)

In Table 3.20, the "actual" change in speed-up shown in row 5 can be easily obtained from the "actual" change in time shown in row 4. Furthermore, the change in NP shown in row 6 is in close agreements with row 5 when NP is large. The "actual" number of elements processed by each processor is shown in row 2, the "predicted" worst case is shown in row 3 and is in good agreement with row 2 when NP is large.

Table 3.20 "Predicted" and "actual" speed-up for Qin-Nguyen's G/A of 24 d.o.f. solid elements on multiple processors MEIKO computer

Nel=15**3	NP=8	NP=10	NP=16	NP=20	NP=30	NP=64	NP=96
#Els./Processor (actual)	1830	2120	1620	1300	870	415	270
#Els./Processor (predicted)	3375	2700	1687	1350	900	421.9	281.3
"actual" time for G/A (seconds)	30.01	34.85	26.68	21.22	14.20	6.73	4.49
"actual" change in speed-up	N/A	1.1613 (=34.85/ 30.01)	1.30622	1.2573	1.4944	2.11 (=14.20/ 6.73)	1.499
"predicted" change in speed-up	N/A	1.25 (=10/8)	1.60	1.25	1.50	2.13 (=64/30)	1.50

3.8 Summary

Simple and efficient parallel algorithms for generating and assembling the structural stiffness (or mass) matrix have been developed and tested in a non-dedicated computer environment on different supercomputers. The new algorithms circumvent the processor synchronization problem associated with implementing the conventional approach on a

parallel machine.

Good speed-up factors were obtained even for small to medium scale structural examples. Better speed-up factors can be expected from the new algorithms in a truly dedicated computer environment. The new parallel algorithms are general since there is no assumption made on the type of finite elements used.

While both parallel G & A algorithms (presented in Sections 3.2 and 3.5) have offered excellent speed up as the number of processors increased, the Qin-Nguyen's G & A algorithm is a preferred choice due to its simplicity, generality (the algorithm can be easily used to construct the stiffness matrix in either column-by-column, or row-by-row fashion) and portability (the algorithm can be implemented on either distributed, or shared memory computers).

3.9 Exercises

3.1 In Figure 3.9 (which is related to Figure 3.8), the following assumptions have been made:

(a) 3 processors were used, and

(b) each processor stored block (4) columns of the coefficient stiffness matrix.

Re-construct Tables 3.14, 3.15 and Figure 3.10, due to the following changes in the assumptions:

(c) 4 processors and block (4) columns are used ?

(d) 3 processors and block (6) columns are used ?

(e) 4 processors and block (6) columns are used ?

3.2 A two dimensional finite element problem is modeled by 4 triangular and 1 rectangular elements as shown in Figure P3.2. For this problem, construct a Table similar to the one explained in Table 3.8 ??

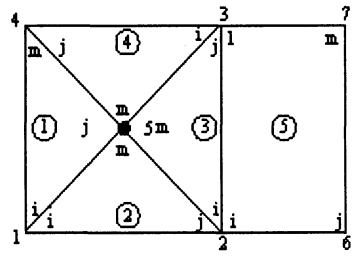


Figure P3.2

3.10 References

3.1 Storaasli, O.O., Nguyen, D.T., and Agarwal, T.K., "Parallel-Vector Solution of Large-Scale Structural Analysis Problems on Supercomputers," *AIAA Journal*, Volume 28, Number 7, July 1990, pp. 1211-1216 (also in *Proceeding of the 30th AIAA/ASME/ASCE/AHS Structures, Structural Dynamics and Materials Conference,* Mobil, AL, April 3-5, 1989, pp. 859-867).

3.2 Qin, J., Gray, C.E. Jr., Mei, C., and Nguyen, D.T., "Parallel-Vector Equation Solver for Unsymmetric Matrices on Supercomputers," Computing System in Engineering, Vol. 2, No. 2/3, pp. 197-201, 1991.

3.3 Qin, J., Gray, C.E. Jr., Mei, C., and Nguyen, D.T., "Parallel-Vector Equation Solver for Unsymmetric Matrices on Supercomputers," *Proceedings of the NASA-USAF Symposium on Parallel Methods on Large-Scale Structural Analysis and Physics Applications*, Hampton, VA, February 5-6, 1991.

3.4 Qin, J., and Nguyen, D.T., "A New Parallel-Vector Finite Element Analysis Software on Distributed Memory Computers," Proceedings of the AIAA/ASME/ASCE/AHS 34th SDM Conference, La Jolla, CA (April 19-22, 1993).

3.5 Maker, B.N., Qin, J. and Nguyen, D.T., "Performance of NIKE3D with PVSOLVE on Vector and Parallel Computers," Computing Systems In Engineering Journal (1995).

3.6 Qin, J., Nguyen, D.T., and Zhang, Y., "Parallel-Vector Lanczos Eigensolver for Structural Vibration Problems," in Proceeding of Fourth International Conference on Recent Advances in Structural Dynamics, July 15-18, London, UK, 1991.

3.7 Qin, J., and Nguyen, D.T., "A Parallel-Vector Equation Solver for Distributed-Memory Computers," presented at The 2nd Symposium on Parallel Computational Methods for Large Scale Structural Analysis and Design, February 24-25, 1993, Norfolk, VA.

3.8 Baddourah, M.A., "Parallel-Vector Computation for Geometrically Nonlinear Frame Structural Analysis and Design Sensitivity Analysis," Ph.D. Thesis, Old Dominion University, Department of Civil and Environmental Engineering, (1991).

3.9 Baddourah, M.A., Storaasli, O.O., Carmona, E.A. and Nguyen, D.T., "A Fast Parallel Algorithm for Generation and Assembly of Finite Element Stiffness and Mass Matrices." Proceedings of the AIAA/ASME/ASCE/AHS 32nd SDM Conference, Baltimore, MD (April 8-10, 1991).

3.10 Chien, L.S., and Sun, C.T., "Parallel Processing Techniques for Finite Element Analysis of Nonlinear Large Truss Structures," *Computer & Structures*, Volume 31, No. 6, pp. 1023-1029, 1989.

3.11 Belvin, W.K., Elliott, K.E., Bruner, A., Sulla, J., and Bailey, J., "The LaRC CSI Phase-O Evolutionary Model Tested: Design and Experimental Results, *Proceedings of the Fourth Annual NASA/DOD Conference on Control/Structure Interaction Technology*, Orlando, Florida, November 5-7, 1990.

3.12 Belvin, W.K., Maghami, P.G. and Nguyen, D.T., "Efficient Use of High-Performance Computers for Integrated Controls and Structures Design," Computing Systems in Engineering Journal, Vol. 3, No. 1-4, pp. 181-188 (1992).

3.13 Jordan, 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.

4 Parallel-Vector Skyline Equation Solver on Shared Memory Computers

4.1 Introduction

The solution of linear systems of equations on advanced parallel and/or vector computers is an important area of ongoing research.[4.1 - 4.2]. The development of efficient equation solvers is particularly important for static and dynamic structural analyses, eigenvalue and buckling analyses, sensitivity analysis and structural optimization procedures.[4.3 - 4.5] Research has been directed towards developing either effective vector methods or parallel methods to solve linear systems of equations. However, modern supercomputers now have both parallel and vector capability; algorithms that exploit both capabilities are the most desirable.

This chapter presents a direct Choleski-based equation solver which exploits both parallel and vector features of supercomputers. The objective of this chapter is to describe this new equation solver and to evaluate its performance by solving structural analysis problems on high-performance computers.

4.2 Choleski-based Solution Strategies

The key to reducing the computation time for structural analysis is to reduce the time to solve the resulting linear system of equations. Using matrix notations, the linear system of equations can be conveniently expressed as:

$$[K]\{Z\} = \{F\} \tag{4.1}$$

For many engineering applications, the coefficient matrix (or stiffness matrix, for structural engineering applications) often has nice properties, such as symmetry and positive definiteness. In Eq. 4.1, {Z} and {F} are unknown (or nodal displacements, for structural engineering applications) and known (or nodal forces, for structural engineering applications) vectors, respectively.

On sequential computers, direct methods based on Choleski factorization are both accurate and fast in solving a wide range of structural analysis problems. These methods are used in most commercial finite element codes. Choleski-based methods have also been found to be accurate and fast in solving structural analysis problems on parallel computers.

In the Choleski methods, the unknown vector {Z} can be found in three distinct steps.

First Step: Factorization (or Decomposition)

In this step, the coefficient matrix [K] can be decomposed as

$$[K] = [U]^T [U] \tag{4.2}$$

where [U] is an upper triangular matrix, and thus $[U]^T$ is a lower triangular matrix.

Second Step: Forward Solution

Assuming the matrix [U] in Eq. 4.2 has already been obtained, one can substitute Eq. 4.2 into Eq. 4.1 to obtain

$$[U]^T [U] \{Z\} = \{F\} \tag{4.3}$$

The product [U] * {Z} in Eq. 4.3 can be renamed as vector {y}. Hence, Eq. 4.3 becomes:

$$[U]^T * \{y\} = \{F\} \tag{4.4}$$

The forward solution step is completed upon solving the unknown vector {y} from Eq. 4.4.

Third Step: Backward Solution

From the second step, one has

$$[U] \{Z\} = \{y\} \tag{4.5}$$

where the factorized matrix [U] and vector {y} have already been found from the first and second steps, respectively.

The final (and original) unknown vector {Z} can be found by solving Eq. 4.5. For a single right-hand-side vector {F}, the first step is the most time consuming step. Approximately 90% or more of the total equation solution time is spent in the factorization step alone. The remaining 10% or less of the total equation solution time is spent on Forward/Backward solution steps.

4.3 Factorization

In Choleski-based methods, a symmetric, positive definite (stiffness) matrix [K] can be factorized as indicated in Eq. 4.2. The U_{ij} terms of matrix [U] in Eq. 4.2 can be computed according to the following formulas

$$U_{ij} = 0 \qquad for \qquad i > j \tag{4.6}$$

$$U_{11} = \sqrt{K_{11}} \quad ; \quad U_{1j} = \frac{K_{1j}}{U_{11}} \tag{4.7}$$

$$U_{ii} = \left(K_{ii} - \sum_{k=1}^{i-1} U_{ki}^2 \right)^{1/2} \qquad for \quad i > 1 \tag{4.8}$$

$$U_{ij} = \frac{\left(K_{ij} - \sum_{k=1}^{i-1} U_{ki} U_{kj} \right)}{U_{ii}} \qquad for \quad i,j > 1 \tag{4.9}$$

Having obtained the upper triangular matrix [U] from the Choleski decomposition phase, the solution vector {Z} for the system of simultaneous equations (see Eq. 4.1) is found by the forward (see Eq. 4.4) and backward (see Eq. 4.5) substitution phases. Equation 4.6 is obvious, since [U] is an upper triangular matrix, and therefore, its lower triangular portion must be zero. The determination of the Eqs. 4.7 - 4.9 can be easily understood with the help of the following example of a 3 x 3 [K] matrix. From Eq. 4.2, one has

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} = \begin{bmatrix} u_{11} & 0 & 0 \\ u_{12} & u_{22} & 0 \\ u_{13} & u_{23} & u_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{4.10}$$

Equating the upper triangular portion of the left hand side of Eq. 4.10 with the corresponding right hand side, one obtains:

$$K_{11} = u_{11}^2 \qquad or \qquad u_{11} = \sqrt{K_{11}} \tag{4.11}$$

$$K_{12} = u_{11}\, u_{12} \qquad or \qquad u_{12} = \frac{K_{12}}{u_{11}} \tag{4.12}$$

$$K_{13} = u_{11}\, u_{13} \qquad or \qquad u_{13} = \frac{K_{13}}{u_{11}} \tag{4.13}$$

$$K_{22} = u_{12}^2 + u_{22}^2 \qquad or \qquad u_{22} = \sqrt{K_{22} - u_{12}^2} \tag{4.14}$$

$$K_{23} = u_{12}\, u_{13} + u_{22}\, u_{23} \quad or \quad u_{23} = \frac{K_{23} - u_{12}\, u_{13}}{u_{22}} \tag{4.15}$$

$$K_{33} = u_{13}^2 + u_{23}^2 + u_{33}^2 \quad or \quad u_{33} = \sqrt{K_{33} - u_{13}^2 - u_{23}^2} \tag{4.16}$$

Thus, it can be seen that Eqs. 4.7 - 4.9 are general versions of Eqs. 4.11 - 4.16.

The order of computation for the matrix [U] can be dictated by the storage schemes used to store the "stiffness" matrix [K] and its corresponding factorized matrix U. For example, a closer look into Eqs. 4.11 - 4.16 will reveal the facts that matrix U

has been computed according to a row-by-row storage scheme. Eqs. 4.11 - 4.13 compute the first row of matrix [U], and Eqs. 4.14 and 4.15 compute the second row of matrix U. Similarly, Eq. 4.16 computes the third row of matrix U. In this case, since matrix U can be computed in a row-by-row fashion, the original stiffness matrix [K] should also be stored in a row-by-row fashion (refer to Section 2.4 of Chapter 2). A more detailed row-by-row storage scheme and its associated equation solution strategy, however, will be discussed in Chapter 5.

In this chapter, however, the original stiffness matrix [K], shown in Eq. 4.10, will be stored, and factorized in a column-by-column (or skyline) fashion[4.6]. Details of the skyline storage scheme has already been discussed in Section 2.5. To simplify the discussions, assuming the 3 x 3 matrix [K], shown in Eq. 4.10, is completely full. In practice, only the upper-half of the matrix [K] needs to be stored in a 1-dimensional array A(-), column-by-column fashion. Furthermore, for each column of the matrix [K], its corresponding numerical values will be stored from the diagonal term and in the upward direction. As an example, the 1st, 2nd and 3rd columns of [K] will be stored in a 1-D array A(-), according to the following patterns:

$$A\begin{pmatrix}1\\2\\3\\4\\5\\6\end{pmatrix} = \begin{Bmatrix}K_{11}\\K_{22}\\K_{12}\\K_{33}\\K_{23}\\K_{13}\end{Bmatrix} \qquad \textbf{(4.17)}$$

The factorized matrix [U], therefore, should also be computed in the following column-by-column fashion, refer to Eqs. 4.11 - 4.16:

$$u_{11} = \sqrt{K_{11}} \qquad \text{Beginning of 1st column of [U]} \qquad \textbf{(4.11R)}$$

$$u_{12} = \frac{K_{12}}{u_{11}} \qquad \text{Beginning of 2nd column of [U]} \qquad \textbf{(4.12R)}$$

$$u_{22} = \sqrt{K_{22} - u_{12}^2} \qquad \textbf{(4.14R)}$$

$$u_{13} = \frac{K_{13}}{u_{11}} \qquad \text{Beginning of 3rd column of [U]} \qquad \textbf{(4.13R)}$$

$$u_{23} = \frac{K_{23} - u_{12}\,u_{13}}{u_{22}} \qquad \textbf{(4.15R)}$$

$$u_{33} = \sqrt{K_{33} - u_{13}^2 - u_{23}^2} \qquad \textbf{(4.16R)}$$

It is rather obvious to see that Eqs. 4.11 through 4.16 are simply the repeated equations of Eqs 4.11R through 4.16R. The major difference between these two sets of equations is only in the order of computations for the factorized terms u_{ij}. The former was based on a row-by-row factorization scheme, while the latter was based on a column-by-

column scheme. Furthermore, for each column, the factorized elements u_{ij} (for $j \geq i$) have been computed in the direction from the top element of the column to the diagonal element of the same column, refer to Eqs. 4.11R through 4.16R, and Figure 4.1.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ & \downarrow & \downarrow \\ & u_{22} & u_{23} \\ & & \downarrow \\ & & u_{33} \end{bmatrix}$$

Figure 4.1 Column-by-column (skyline) factorization

4.3.1 Basic, sequential skyline Choleski factorization: computer code (version 1)

To facilitate the discussion in this section, a full and symmetrical stiffness matrix [K], with nine rows and nine columns, is shown in Figure 4.2.

$$\begin{array}{cccccccccc} & & & \textit{Column} & & & \textit{Column} & & & \\ & & & I = 4 & & & J = 7 & & & \\ \begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} & K_{15} & K_{16} & K_{17} & K_{18} & K_{19} \\ & K_{22} & K_{23} & K_{24} & K_{25} & K_{26} & K_{27} & K_{28} & K_{29} \\ & & K_{33} & K_{34} & K_{35} & K_{36} & K_{37} & K_{38} & K_{39} \\ & & & K_{44} & K_{45} & K_{46} & K_{47} & K_{48} & K_{49} \\ SYM. & . & . & . & K_{55} & K_{56} & K_{57} & K_{58} & K_{59} \\ & & & & & K_{66} & K_{67} & K_{68} & K_{69} \\ & & & & & & K_{77} & K_{78} & K_{79} \\ & & & & & & & K_{88} & K_{89} \\ & & & & & & & & K \end{bmatrix} & & & & & & & & & \begin{array}{l} \\ \\ Row\ I\text{-}1 = 3 \\ Row\ I = 4 \\ \\ \\ Row\ J = 7 \\ \\ \\ \end{array} \end{array}$$

Figure 4.2 A full, symmetrical stiffness matrix [K]

In practice, the calculated factorized matrix [U], with its terms u_{ij}, will be stored in the same locations as the original terms K_{ij}. Furthermore, the original matrix [K] will *not* be full and it will be stored in the one-dimensional array, as it has been discussed in Sections 2.5, and 2.7 through 2.9. Factorizing the matrix [U] with implementations of these detailed features will be postponed in later sections of this chapter. Let us try to carefully examine Eqs. 4.8 and 4.9 by computing the following typical off-diagonal term of the matrix [U]. According to Eq. 4.9, one has

$$u_{ij}(with\ i = 4, j = 7) = u_{47} = \frac{K_{47} - \sum_{k=1}^{i-1} u_{k,i}\, u_{k,j}}{u_{44}} \qquad \textbf{(4.18)}$$

Eq. 4.18 can be written in the expanded form as

$$u_{47} = \frac{K_{47} - \left(u_{14}\,u_{17} + u_{24}\,u_{27} + u_{34}\,u_{37}\right)}{u_{44}} \tag{4.19}$$

Similarly, the typical diagonal term u_{77} can be computed according to Eq. 4.8 as

$$u_{77} = \sqrt{K_{77} - \left(u_{17}^2 + u_{27}^2 + u_{37}^2 + u_{47}^2 + u_{57}^2 + u_{67}^2\right)} \tag{4.20}$$

At this point, it should be pointed out that if i = j, then Eq. 4.9 will be automatically reduced to Eq. 4.8 (with the exceptions of the square root and dividing of u_{ii} operations). Thus, the most important equation in this section is Eq. 4.9, which is used to compute the off-diagonal terms, as well as the diagonal terms, for the reasons which have already been cited.

Equation 4.19 reveals an important fact that, in order to compute u_{47}, one needs to know the factorized matrix [U] associated with *only* columns, 4 and 7. The summation done inside the parenthesis of Eq. 4.19 can be considered as the dot product of two vectors

$$\begin{Bmatrix} u_{14} \\ u_{24} \\ u_{34} \end{Bmatrix} \text{ and } \begin{Bmatrix} u_{17} \\ u_{27} \\ u_{37} \end{Bmatrix}.$$

This dot product operation can be conveniently visualized by referring to columns 4 and 7 of Figure 4.2. Similarly, Eq. 4.20 indicates that, in order to compute u_{77}, one needs to know the factorized matrix [U] associated with *only* columns 7 and 7. In other words, only the factorized column 7 is needed in order to compute u_{77}. The summation done inside the parenthesis of Eq. 4.20 can be considered as the dot product of the vector

$\begin{Bmatrix} u_{17} \\ u_{27} \\ u_{37} \\ u_{47} \\ u_{57} \\ u_{67} \end{Bmatrix}$ on itself. This dot product operation can be conveniently visualized by referring to column 7 of Figure 4.2.

It has been stated in Section 4.3, in particular to the references to Eqs. 4.11R through 4.16R, that column-by-column (or skyline) factorization of the given matrix [K] will proceed in the direction from left to right. In other words, column 1 is factorized first, then columns 2, 3, ... n are factorized. Furthermore, within each column, factorization will proceed in the direction from the top of the column down to the diagonal term of the same column. Thus, if one wishes to compute the factorized term u_{47} (the 4th term from the top of column 7), then it has been implied that the first 6 columns of [U], and the first 3 terms of column 7, such as u_{17}, u_{27} and u_{37}, of [U] had already been completely factorized.

Similarly, the computation of the factorized term u_{77} has the implications that the first 6 columns of [U], and the first 6 terms of column 7, such as u_{17}, u_{27}, ..., u_{67}, of [U] had already been completely factorized.

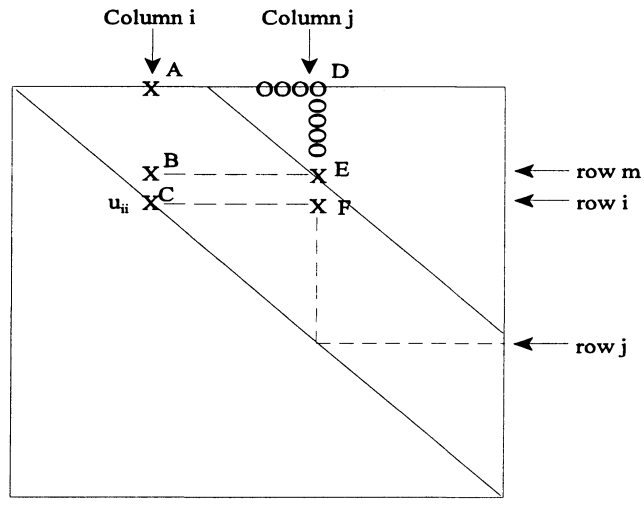


Figure 4.3 Factorization of general diagonal term u_{ii} (= point C) and general off-diagonal term U_{ij} (= point F)

As indicated in Figure 4.3, factorizing a typical diagonal term u_{ii} will require the dot product operations of column i on itself. Thus, according to Eq. 4.8 and referring to Figure 4.3, one has, (assuming column i has full height).

$$u_{ii} = \left[K_{ii} - \left(u_{1i}^2 + u_{2i}^2 + \ldots\ldots + u_{i-1,\,i-1}^2 \right) \right]^{1/2} \qquad \textbf{(4.21)}$$

Similarly, factorizing a typical off-diagonal term u_{ij} will require the dot product operations of column i and column j (above the i^{th} row). In general, a column may not have its full height. As an example, the j^{th} column in Figure 4.3 only has the height EF, whereas the i^{th} column has the full height AC. Thus, factorizing u_{ij} will require the dot product operations of segment BC for the i^{th} column, and segment EF (=BC) for the j^{th} column. In other words, according to Eq. 4.9, one has

$$u_{ij} = \frac{K_{ij} - \left(u_{m,i} * u_{m,j} + u_{m+1,i} * u_{m+1,j} + \ldots + u_{i-1,i}\, u_{i-1,j} \right)}{u_{ii}} \qquad \textbf{(4.22)}$$

A skeleton, pseudo FORTRAN code (version 1) for column-by-column Choleski factorization, is now given in Table 4.1.

The pseudo FORTRAN code, shown in Table 4.1, can be understood with the aid of Figure 4.2, and Eqs. 4.8 and 4.9. Referring to Table 4.1, different FORTRAN statements will perform different operations during the Choleski factorization.

Table 4.1 Basic (column oriented) Choleski factorization (version 1)

1		$U_{11} = SQRT(K_{11})$	*(referring to Fig. 4.2)*
2		DO 1 J = 2, N	*(Say J= 7th column)*
3		DO 2 I = Top Row # of Column J, Row J	*(Say I = 4th Row)*
4		SUM1 = 0	
5		DO 3 K = Top Row # of Column I, Row I-1	
6	3	SUM1 = SUM1 + U_{KI} * U_{KJ}	
7		$U_{IJ} = K_{IJ}$ - SUM1	
8		IF (I. EQ. J) THEN	
9		$U_{II} = SQRT(U_{IJ})$	
10		ELSE	
11		$U_{IJ} = U_{IJ}/U_{II}$	
12		ENDIF	
13	2	CONTINUE	
14	1	CONTINUE	

Referring to Table 4.1 and Figure 4.2 simultaneously, one observes:

Line 1: First column of the [K] matrix is factorized. The factorized diagonal term U_{11} is computed by using Eq. 4.8.

Line 2: This first do loop will sweep from column 2 to column N, with the increment 1. Let us just concentrate on a typical J^{th} column, say J = 7^{th} column.

Line 3: This second nested do loop is needed, since within the 7^{th} column, there are several terms (= 7 terms, since the [K] matrix is assumed to be full) that need to be factorized (U_{17}, U_{27}, U_{37}, U_{47}, ... U_{77}). Since the 7^{th} column is assumed to have its full height, this loop should sweep from row 1 to row J (=7). In general, if the column height of the 7^{th} column is shorter, then this loop may sweep from TOP ROW # OF COLUMN J (either row 2, or 3, etc....) Let us just concentrate on a typical term, say U_{47} (or I = 4).

Line 4: The summation is initialized.

Line 5: This 3^{rd} nested loop is needed, since the computation of U_{47} will require the dot product of 2 vectors $\begin{Bmatrix} u_{14} \\ u_{24} \\ u_{34} \end{Bmatrix}$ and $\begin{Bmatrix} u_{17} \\ u_{27} \\ u_{37} \end{Bmatrix}$. In this case, the index K of this loop will sweep from 1 to I-1 (= 4-1=3). Again, since the 4^{th} column may not have its full height, the index K of this loop will sweep from TOP ROW # OF COLUMN I to I-1.

Line 6: The dot product of the 2 vectors $\begin{Bmatrix} u_{14} \\ u_{24} \\ u_{34} \end{Bmatrix}$ and $\begin{Bmatrix} u_{17} \\ u_{27} \\ u_{37} \end{Bmatrix}$ are carried, and the result is stored in variable SUM1.

Line 7: The nominator of Eq. 4.8, or Eq. 4.9 is computed.

Lines 8, 9: If the index I and J are the same, then the factorized term is recognized as a diagonal term. Hence, Eq. 4.8 is applied.

Lines 10-12: If the index I and J are NOT the same, then the factorized term is an off-diagonal term. Hence, Eq. 4.9 is applied.

Line 13: End of the 2nd loop.

Line 14: End of the 1st loop.

4.3.2 Improved basic, sequential skyline Choleski factorization: computer code (version 2)

In FORTRAN computer coding, logical "IF" statement is not cheap to execute. The purpose of this section is to try to avoid the use of logical "IF" statement, as shown in statement number 8 of Table 4.1.

Referring to Figure 4.2, one can see that factorizing column 7 will require the computation of u_{17}, u_{27}, u_{37}, u_{47}, u_{57}, u_{67}, and u_{77}. Loop I of Table 4.1 (see statement #3) will sweep through from TOP ROW # OF COLUMN J (= row 1) to ROW J (= row 7). However, it is obvious that the first 6 rows of column 7 (= u_{17}, u_{27}, ... u_{67}) contain only off-diagonal forms, while the last row of column 7 contains the diagonal term u_{77}.

Realizing this fact, Table 4.2 offers a simple coding strategy to avoid the use of a logical IF statement during Choleski factorization. Comparing Tables 4.1 and 4.2, one can see that Loop 2 in Table 4.1 has been separated into 2 loops, loop 2 and loop 33, in Table 4.2. In Table 4.2, loop 2 now sweeps from TOP ROW # OF COLUMN J (= row 1) to ROW #J-1 (= row 6). Thus, even without an IF check on the index I and J, one knows for sure that loop 2 will compute only off-diagonal terms. Since only the first 6 rows (instead of 7 rows) of column 7 have been processed by the index I of loop 2 in Table 4.2, it implies that loop 3 (see Table 4.2) needs to be executed one more time to take care of the diagonal term u_{77}. Statements 11 through 14 of Table 4.2, therefore, are used to factorize diagonal term u_{77}.

Table 4.2 Basic Choleski factorization without IF statements (version 2)

1		U_{11} = SQRT (K_{11})	
2		DO 1 J = 2, N	*(Referring to Figure 4.2 Say J = 7th Column)*
3	C	TREAT ALL OFF-DIAGONAL TERMS FIRST	
4		DO 2 I = Top Row of Column J, Row J - 1	*(Say I = 4)*
5		SUM1 = 0	
6		DO 3 K = Top Row of Col. I, Row I - 1	
7	3	SUM1 = SUM1 + U_{KI} * U_{KJ}	
8		U_{IJ} = (K_{IJ} - SUM1) / U_{II}	
9	2	CONTINUE	
10	C C	NOW, TREAT THE CASE I = J (DIAGONAL TERM) SEPARATELY	
11		SUM1 = 0	
12		DO 33 K = Top Row of Col I, Row I - 1	
13	3	SUM1 = SUM1 + U_{KI} * U_{KI}	
14		U_{II} = SQRT (K_{II} - SUM1)	
15	1	CONTINUE	

4.3.3 Parallel-vector Choleski factorization (version 3)

From Eqs. 4.8 and 4.9, and by referring to Figure 4.2, it may seem the column-by-column Choleski factorization algorithm involves highly sequential operations. For example, factorizing all 7 terms (=u_{17}, u_{27}, ... u_{77}) in the 7th column of Figure 4.2 cannot completely be done, unless "all" previous columns 1 through 6 have been completely factorized.

However, a more careful look into Eqs. 4.8 and 4.9 and Figure 4.2 will reveal the following important facts, which can be further exploited later on to develop efficient parallel Choleski factorization algorithms for shared memory computers, such as Cray-YMP, Cray C-90 computers.

Using Eq. 4.9, one has

$$u_{17} = \frac{K_{17}}{u_{11}} \tag{4.23}$$

$$u_{27} = \frac{K_{27} - u_{12}\,u_{17}}{u_{22}} \tag{4.24}$$

$$u_{37} = \frac{K_{37} - \left(u_{13}\,u_{17} + u_{23}\,u_{27}\right)}{u_{33}} \tag{4.25}$$

$$u_{47} = \frac{K_{47} - \left(u_{14}\,u_{17} + u_{24}\,u_{27} + u_{34}\,u_{37}\right)}{u_{44}} \tag{4.26}$$

$$u_{57} = \frac{K_{57} - \left(u_{15}\,u_{17} + u_{25}\,u_{27} + u_{35}\,u_{37} + u_{45}\,u_{47}\right)}{u_{55}} \tag{4.27}$$

$$u_{67} = \frac{K_{67} - \left(u_{16}\,u_{17} + u_{26}\,u_{27} + u_{36}\,u_{37} + u_{46}\,u_{47} + u_{56}\,u_{57}\right)}{u_{66}} \tag{4.28}$$

$$u_{77} = \left[K_{77} - \left(u_{17}^2 + u_{27}^2 + u_{37}^2 + u_{47}^2 + u_{57}^2 + u_{67}^2\right)\right]^{1/2} \tag{4.29}$$

Equation 4.23 clearly indicates that as soon as factorization of column 1, or u_{11}, is done, the first term of the 7th column (= u_{17}) can be computed, even though columns 2 through 6 may NOT be done yet!

Similarly, Eq. 4.26 indicates that the term u_{47} of the 7th column can be readily computed as soon as column 4 has been completely factorized, even though columns 5 and 6 may not be done (or factorized) yet. Since the column-by-column factorization scheme proceeds in the direction from the top down to the diagonal term (of a given column) therefore, the computation of u_{47} has already implied that u_{17}, u_{27}, and u_{37} had already been completely factorized earlier.

The above observations will immediately lead to different options for parallel factorization strategies. In Figure 4.2, assuming that each column will be factorized by different processors. For example, Columns 1 through 9 will be handled by processors 1 through 4 (see Figure 4.4).

Option A: Make Only One (1) Synchronization Check

Again, we assume that the 7th column is currently being factorized by a particular processor, say the 2nd processor. In this option, the 2nd processor will make only 1 synchronization check:

Has column #6 been completely factorized, by the 1st processor, yet?

If the answer to the above questions is NO, then processor #2 will wait, until column 6 is done. If the answer is YES, then processor #2 will proceed to compute, in the direction from top down, u_{17}, u_{27},..., u_{77}. Finally, processor #2 will broadcast to all other processors that column #7 has been completely factorized.

Option B: Make "A Lot" of Synchronization Checks

In this option, the 2nd processor will make "a lot" of synchronization checks:

Has column #1 been completely factorized yet?

If the answer is NO, then it (= processor 2) will wait, until column 1 is done. If the answer is YES, then it will compute u_{17}, and proceed to ask the next question:

Has column #2 been completely factorized by processor 1 yet?

If the answer is NO, then it will wait, until column 2 is done. If the answer is YES, then it will compute u_{27}, and proceed to ask the next questions. etc.

Has column #6 been completely factorized by processor 1 yet?

If the answer is NO, then it will wait. If the answer is YES, then it will compute u_{67} and proceed to compute u_{77}, without asking any further questions, since processor #2 is the "owner" of column 7, and all required information to compute u_{77} are already available and belongs to this processor. Finally, processor #2 will broadcast to all other processors that column #7 has been completely factorized.

Option C: Make "A Few" Synchronization Checks

In this option, the 2nd processor will make "a few" synchronization checks:

Has column #3 been completely factorized, by the 2nd processor?

If the answer is NO, then it will wait, until column 3 is done, or else it will compute u_{17}, u_{27} and u_{37}. Then, another synchronization check will be made.

Has column #6 been completely factorized?

If the answer is NO, then it will wait, or else it will compute u_{47}, u_{57}, u_{67}, and u_{77}. Finally, all processors will be informed, by processor #2, that column 7 has been completely factorized.

Analysis of Options A, B, and C:

It is rather obvious to see that Option A is an extreme, where factorization has been conducted essentially in a sequential fashion. For example, unless all first 6 columns of the stiffness matrix [K] have been completely factorized, *none* of the terms in column 7 will be factorized, by processor #2.

Option B is another extreme, it will offer good parallel computation, but at the price of paying higher communication cost. In this option, assuming only the first 2 columns of stiffness matrix [K] have been completely factorized, the other columns 3 through 6 have not been factorized yet, then the first two terms, from the top, of column 7 (= u_{17} and u_{27}) can still be computed by the 2nd processor. In this option, if column J belongs to processor I, then the Ith processor will have to make (J-1) synchronization checks.

Option C is a compromise between the 2 options A and B, since only a "few," say 2 or 3 (instead of only "1" as in Option A, or "J - 1" as in Option B) synchronization checks need to be done. The key idea presented in Section C can also be partially explained in Figure 4.5.

The skeleton of a pseudo-FORTRAN Parallel-Vector Choleski factorization computer code (Version 3) is outlined in Table 4.3.

In Table 4.3, different statements will perform different operations during the Choleski factorization, as will be explained in the following paragraphs.

Line 1: All columns are initially declared as *NOT* done (or *NOT* factorized) yet.

Line 2: The first column is factorized by a particular processor.

Line 3: All processors are informed that column 1 has been done (or factorized).

Line 4: In the first do loop, different values of index J, which represent column numbers, are assigned to different processors for parallel factorization. For the matrix example shown in Figure 4.4, columns (2, 6), (3, 7), (4, 8), and (5, 9) are assigned to processors 1, 2, 3 and 4, respectively.

Line 5: The second do loop with index I, has already been explained in statement 4 of Table 4.2

Line 6: Synchronization checks are performed according to Option B.

Lines 7 - 16: In this third do loop, different off-diagonal terms are factorized by different processors. In Figure 4.4, for example, the off-diagonal terms u_{12}, u_{13}, u_{14} and u_{15} are being factorized simultaneously by processors 1, 2, 3 and 4, respectively. Upon completions these tasks, processor 1 will get out of the 2nd do loop and it proceeds to compute the factorized term u_{22} (for the case index I = J). At the same moment, processors 2, 3, and 4 all have to wait, "idle," since they cannot compute u_{23}, u_{24}, and u_{25} (refer to Eq. 4.9) unless the computation of the diagonal term u_{22} has been completed by processor 1.

As soon as u_{23}, u_{24} and u_{25} have been computed, by processors 2, 3, and 4, respectively, processor 2 will also get out of the 2nd loop and it proceeds to compute the diagonal term u_{33} (for the case index I = J).

Line 17: As soon as the diagonal terms of any columns have been computed (by any processor) the associated column will be declared to all other processors as completely done.

Line 18: Another J^{th} column will be processed by a processor

Col 1	*Processor* 1	P_2	P_3	P_4	P_1	P_2	P_3	P_4
Done	(*or* P_1)							
X	*X*	*X*	*X*	*X*	*X*	*	*X*	*X*
	X	*X*	*X*	*X*	*X*	*	*X*	*X*
		X	*X*	*X*	*X*	*	*X*	*X*
			X	*X*	*X*	*	*X*	*X*
SYM				*X*	*X*	*	*X*	*X*
					X	*	*X*	*X*
						*	*X*	*X*
							X	*X*
								X

Figure 4.4 Parallel and basic vector Choleski factorization

Table 4.3 Parallel vector Choleski factorization (version 3)

```
1    ALL COLUMNS ARE DECLARED AS NOT DONE (OR NOT
     FACTORIZED) YET
2    U11 = SQRT ( K11 )
3    BROADCAST TO ALL PROCESSORS THAT COL. #1 WAS DONE
     ALREADY
4    PARALLEL DO 1 J = 2, N (Referring to Figure 4.4)
5                                   DO 2 I = TOP ROW OF COL J, ROW J - 1
6    Is Col. #I Done? (If NOT, then wait here!)
7                 SUM1 = 0
8                 DO 3 K = TOP ROW OF COL I, ROW I - 1
9              3  SUM1 = SUM1 + UKI * UKJ
10                UIJ = (KIJ - SUM1) / UII
11             2  CONTINUE      (PROCESSOR #2, 3, 4 GO BACK LOOP 2
                                 PROCESSOR #1 EXIT LOOP 2)
12   C... NOW, TREAT THE CASE I = J SEPARATELY
13                •
14                •
15                •
16             UII = SQRT (KII - SUM1)
17   BROADCAST TO ALL PROCESSORS THAT COL. I (= COL J) WAS DONE
18   1    CONTINUE
```

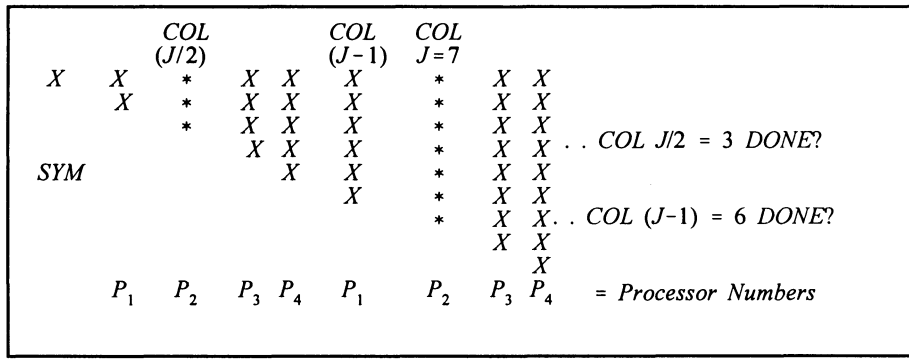


Figure 4.5 Parallel (with a few sychronization checks) and vector Choleski factorization

4.3.4 Parallel-vector (with "few" synchronization checks) Choleski factorization (version 4)

In the previous section, parallel Choleski factorization based on Option B strategy has been discussed. While Option B strategy has the advantage of keeping most of the processors busy (or less idle time) most of the time, it also has potential problems of increasing the communication costs. For most high-performance parallel computers, the communication rate is very slow as compared to the computation rate. This is

especially true for distributed memory parallel computers, such as the Intel Paragon, IBM-SP2, etc. ... For these reasons, parallel Choleski factorization based on Option C strategy (or Version 4) is presented in Table 4.4. Explanations for various statements in Table 4.4 are given in the following paragraphs.

Lines 1-3: The first three statements have already been explained in Table 4.3.

Line 4: Assuming the current column #7 (or J = 7) is being factorized by a particular processor, say processor #2 (or P_2), processor P_2 will make the first synchronization check:

Has column #J/2 (or 7/2 = 3) been completely factorized yet? If the answer is NO, then processor P_2 will wait until column #3 has been completely factorized, and declared as READY by another processor. If the answer is YES, then processor P_2 will proceed to statements 5-8.

Lines 5-8: Assuming column #3 has already been completely done (or completely factorized), a typical processor, say P_2, will compute the factorized terms u_{17}, u_{27}, and u_{37} (see Figure 4.5) according to Eq. 4.9.

Line 9: Again, assuming the current column #7 (or J = 7) is being factorized by a particular processor, say P_2, processor P_2 will make the second synchronization check:

Has column #J-1 (or 7-1 = 6) been completely factorized yet? If the answer is NO, then processor P_2 will wait, until column #6 has been completely factorized, and declared as READY by another processor. If the answer is YES, then processor P_2 will proceed to statements 10-13.

Lines 10-13: Assuming column #6 has already been completely done (or completely factorized), a typical processor, say P_2, will continue to compute u_{47}, u_{57} and u_{67} according to Eq. 4.9.

Lines 14, 15: A typical processor, say P_2, will continue to compute the last term of the J^{th} (= 7^{th}) column, which always happens to be the diagonal term (the case where index I = J) say u_{77}, according to Eq. 4.8.

Line 16: Once the diagonal term of the J^{th} (= 7^{th}) column has been factorized, the J^{th} column will be broadcasted by the processor P_2, to all other processors that this J^{th} column has been done.

Line 17: Processor P_2 will now move to its next column (say column #11, assuming the matrix shown in Figure 4.5 has much more then just 9 columns).

Table 4.4 Parallel (with a few synchronization checks) vector Choleski factorization (version 4)

1		U_{11} = SQRT (K_{11})
2		BROADCAST COL. #1 WAS DONE
3		PARALLEL DO 1 J = 2, N (Referring to Figure 4.5)
4		IS COL. # (J / 2) DONE? IF YES, THEN PROCEED. IF NO, THEN WAIT.
5		DO 2 I = TOP ROW OF COL J, ROW (J - 1) / 2
6		DO 3 K = TOP ROW OF COL I, ROW (I - 1)
7	3	SUM1 = SUM1 + U_{KI} * U_{KJ}
8	2	U_{IJ} = (K_{IJ} - SUM1) / U_{II}

```
9          IS COL. # (J - 1 ) DONE? IF YES, THEN PROCEED. IF NO, THEN
           WAIT.
10         DO 22 I = ROW (J - 1)/ 2 + 1, ROW ( J - 1 )
11         DO 33 K = TOP ROW OF COL I, ROW ( I - 1 )
12    33   SUM1 = SUM1 + U_KI * U_KJ
13    22   U_IJ = (K_IJ - SUM1 ) / U_II
14    C . . .NOW, TREAT THE CASE I = J SEPARATELY
15         U_II = SQRT ( K_II - SUM1 )
16         BROADCAST COL # I ( = COL # J ) WAS DONE
17    1     CONTINUE
```

4.3.5 Parallel-vector enhancement (vector unrolling) Choleski factorization (version 5)

In earlier discussions, it has been seen several times that the Choleski factorization will require three nested do-loops. Furthermore, it has also been explained in Table 4.4 that parallel computation can be exploited in the outermost (or the first) nested do-loop, whereas the most numerical intensive computation can be most effectively vectorized in the innermost (or the third) nested do-loop. In order to further improve the vector speed in the third nested do-loop (refer to statements 11 and 12 in Table 4.4) the general idea is to add more works inside the innermost (or the third) nested do-loop. This can be achieved by using "unrolling" techniques, which have been introduced and explained in Chapter One (through simple applications, such as Matrix-Vector multiplications).

In this section, the unrolling techniques will be extended and incorporated into the Choleski factorization algorithm.

To simplify the discussions, unrolling level 2 is used in Figure 4.6, where the 9 x 9 matrix is also assumed to be full. One also assumes that there are NP = 3 processors available.

Unrolling strategy means each processor will be the owner of a "block" of columns. Thus, unrolling level 2 means each processor will be the owner of block of 2 columns. According to this definition, block columns (2, 3) and block columns (8, 9) etc.... will be assigned to processor P_1. Similarly (assuming that the matrix size is much larger than 9, as shown in Figure 4.6) block columns 4 and 5 and block columns 10 and 11, etc. ... will be assigned to processor P_2. Finally, block columns 6 and 7, and block columns 12 and 13, etc. ... will be assigned to processor P_3.

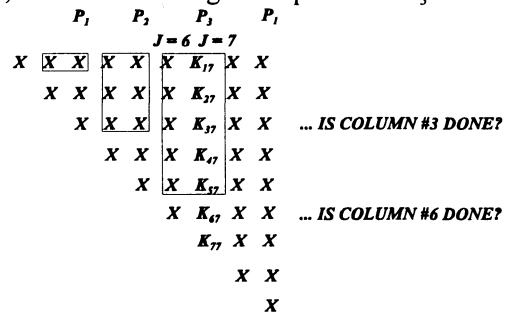


Figure 4.6 Parallel-vector (unrolling) enhancement Choleski factorization

The skeleton of a pseudo-FORTRAN parallel-vector "unrolling" Choleski factorization computer code (Version 5) is outlined in Table 4.5. It should be emphasized here, that version 5 simply reflects some simple modifications on version 4. The readers are strongly suggested to fully understand the previous versions (versions 1 through 4) before reading this section!

In Figure 4.6, one assumes the matrix size is 9 (or much larger), there are 3 processors available, and unrolling level 2 is used. In Table 4.5, different statements will perform different operations during the Choleski factorization, as will be explained in the following paragraphs. (Refer to Figure 4.6).

1: Depending on the total number of equations is an odd, or an even number, the first one (or first two) column(s) need to be factorized initially, by any processor(s).

2: Having been factorized completely, the first one (or two) column(s) are declared as done.

3: Assuming the total number of equations (= N) is an odd number. (i.e. N = 9).

In the first do loop (the outermost do loop), different values of index J, which represents "block" column numbers, are assigned to different processors for parallel factorization. For the matrix example shown in Figure 4.6, "block" columns 2 and 3, 8 and 9, etc. ... are assigned to processor P_1. Similarly, "block" columns 4 and 5, etc. ... are assigned to processor P_2. Finally, "block" columns 6 and 7, etc. ... are assigned to processor P_3. To simplify the following discussions, assuming currently columns J = 6 and J = 7 are being factorized by a particular processor, say P_3. This statement is completely equivalent to statement 3 of Table 4.4. The only difference is the increment of 2 is used in Table 4.5, due to the use of unrolling level 2.

4: Processor, (i.e. P_3) which possesses columns J (= 6) and J+1 (=7) will ask the following question:

Is column # (J/2), or column 6/2 = 3, done??

If the answer is YES, then processor (i.e. P_3) will proceed to the next statements, or else, it will wait, until column #J/2 is declared as done. This statement is equivalent to statement 4 of Table 4.4.

5: This statement is completely equivalent to statement 5 of Table 4.4. The only difference is the increment of 2 is used in Table 4.5, due to the use of unrolling of level 2. Thus, within each column (i.e. columns J and J + 1), a block of 2 rows are considered at a time.

6-11: Since there is a block of 2 columns, which are assigned to each processor, and within these 2 columns, a block of 2 rows are considered at a time. Thus, there will be 4 dot product operations to be executed in the third (inner-most) nested do loop, in order to compute the following 4 factorized terms:

$u_{I,J}$; $u_{I,J+1}$; $u_{I+1,J}$, and $u_{I+1,J+1}$

The reader should recall Table 4.4, where the third (inner-most) nested do-loop only computes 1 factorized term (= $u_{I,J}$),

12: Having factorized about half the column height of columns J and J + 1, a processor (i.e. P_3) will ask the 2nd question:

Is column #J - 1 done?

If YES, then the processor, P_3, will proceed to the next statement(s). If NO, then the processor, P_3, will wait.

13-16: The processor, P_3 will continue to factorize the remaining off-diagonal terms in columns J and J + 1. Again, 4 dot product operations are computed in DO LOOP 22. In other words, DO LOOP 22 is quite similar to loop 2.

17 and 18: The last diagonal term of column J (or u_{JJ}) is now factorized.

19 and 20: The last off-diagonal, $u_{J,J+1}$, and the diagonal terms $u_{J+1,J+1}$ of column J+1 are now factorized.

21: Columns J and J+1 are now declared as completely done (or completely factorized).

22: Processor, P_3, will move to the next "block" of columns, (i.e. columns 12, 13, etc. ...), assuming the matrix shown in Figure 4.6 has much more than just 9 columns.

In summary, the algorithm presented in Table 4.5 is quite similar to the one presented in Table 4.4. The major differences occur in the first (outer-most) nested do-loop, and the second nested do-loop.

In Table 4.4, since only 1 column at a time is assigned to a processor, and within each column, only 1 row is processed at a time, therefore, only 1 dot product operation is required in the third (inner-most) nested do-loop. However, in Table 4.5, since a "block" of 2 or more columns at a time are assigned to a processor, and within these "block" columns, a "block" of 2 or more rows are processed at a time, therefore, at least 4 or more dot product operations are required in the third (inner-most) nested do-loop.

Vector computers, such as the Cray, provide maximum vector speed to access data that are stored continuously. "Stride" is the distance between two adjacent elements of a vector involved in the computations (see Chapter 1). The elements of [U] are stored in column form, in order that they will be in contiguous storage locations. The column-oriented skyline storage scheme offers stride 1 storage and hence the optimum memory retrieval speed.

On parallel-vector supercomputers, such as the Cray-C90 that have high computational speeds, the synchronization time relative to computation time is significant. This synchronization overhead time is reduced by segmenting the loop into two nearly equal sections (see Figure 4.6, and also refer to statements 4 and 12 of Table 4.5) and performing the synchronization only twice for each column.

Table 4.5 Parallel-vector enhancement (vector unrolling)
Choleski factorization (version 5)

1.		FACTORIZE COL. 1, IF NUMBER OF EQUATIONS (N) IS ODD
	or,	FACTORIZE COLS. 1 & 2, IF NUMBER OF EQUATIONS (N) IS EVEN
2.		BROADCAST COL. 1 WAS DONE, IF N IS ODD
	or,	BROADCAST COL. 2 WAS DONE, IF N IS EVEN
3.		PARALLEL DO 1 J = 2, N, 2 (IF N IS ODD), SAY J = 6
	or,	PARALLEL DO 1 J = 3, N, 2 (IF N IS EVEN)

	IS COL. # (J / 2) DONE? IF YES, THEN PROCEED. IF NO, THEN WAIT
	DO 2 I = TOP ROW # OF COLUMN J, ROW (J - 1) / 2, [2]
	For the innermost loop, DO 3 : Compute a set 4 dot products for
	$u_{I,J} = \dfrac{\left(K_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj} \right)}{u_{ii}}$ **(4.9)**
	$u_{I,\,J+1}$ = . . .
	$u_{I+1,\,J}$ = . . .
3	$u_{I+1,\,J+1}$ = . . .
2	CONTINUE
	IS COL. # (J - 1) DONE? IF YES, THEN PROCEED. IF NO, THEN WAIT.
	DO 22 I = ROW (J-1)/2 + 1, ROW (J - 1), [2]
	For the innermost loop, DO 33: Compute a set of 4 dot products for
	$u_{I,J}$; $u_{I,J+1}$; $u_{I+1,\,J}$ and $u_{I+1,\,J+1}$
33	CONTINUE
22	CONTINUE
	C... NOW, FACTORIZE THE REMAINED TERMS OF COLUMNS J AND J + 1
	$u_{J,J}$ = . . .
	$u_{J,\,J+1}$ = . . .
	$u_{J+1,\,J+1}$ = . . .
C....	DECLARED COLUMNS J & (J+1) HAVE BEEN COMPLETELY DONE (*or completely factorized*)
1	CONTINUE

4.3.6 Parallel-vector (unrolling) skyline Choleski factorization (version 6)

The column-oriented skyline Choleski method was implemented in a computer code to exploit both parallel and vector capability of supercomputers. In actual computer code, the elements of [U] overwrite the elements of [K]. The columns of [K] are stored one after the other in a single vector array, and elements of each column are arranged from the diagonal up. This storage arrangement is referred to as skyline storage and is illustrated in Figure 4.7. Column 5 in Figure 4.7, for example, represents the terms K_{55}, K_{54}, K_{53}, and K_{52} of the stiffness matrix [K]. In a vector skyline storage scheme, the corresponding terms for K_{55}, K_{54}, K_{53} and K_{52} are stored in a one dimensional array A at the locations A(10), A(11), A(12), and A(13), respectively (please refer to section 2.5 of Chapter 2). A typical skyline storage scheme for structural engineering applications, such as aircraft panel finite element model, is shown in Figure 4.8.

A variable bandwidth storage scheme for the same aircraft panel finite element model, is shown in Figure 4.9. The factorization, forward and backward solution

strategies associated with the variable storage scheme, however, is postponed until the next chapter.

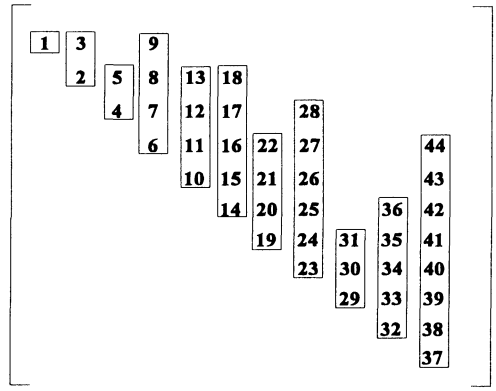


Figure 4.7 Skyline storage scheme for stiffness matrix before and after decomposition

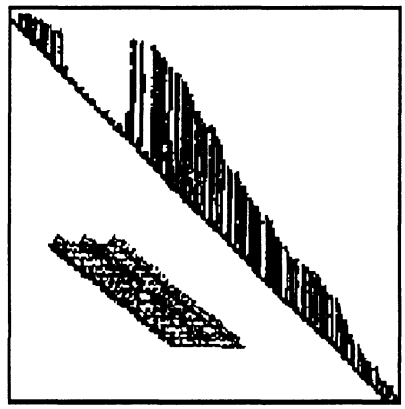


Figure 4.8 Skyline column storage of panel stiffness matrix

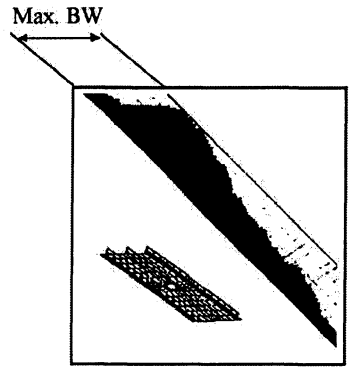


Figure 4.9 Variable bandwidth row storage of panel stiffness matrix

Version 6 of the skyline Choleski factorization is outlined in Table 4.6.

Table 4.6 Parallel vector (unrolling) "skyline" Choleski factorization (version 6)

This version is "exactly" the same as shown in Version 5. However, the two-dimensional stiffness matrix [K], or its factorized matrix [U], will be converted into a one-dimensional array A, using the mapping explained in Section 2.5 of Chapter 2.

U_{KI}	=	A	[MAXA (I) + I - K]
U_{KJ}	=	A	[MAXA (J) + J - K]
$U_{I+1, J+1}$	=	A	[MAXA (J + 1) + (J + 1) - (I + 1)]
U_{IJ}	=	A	[MAXA (J) + J - I]
$U_{I+1, J}$	=	A	[MAXA (J) + J - (I + 1)]
K_{IJ}	=	A	[MAXA (J) + J - I]
$U_{I, J+1}$	=	A	[MAXA (J + 1) + (J + 1) - I]
$U_{J,J}$	=	A	[MAXA (J)]
$U_{J, J+1}$	=	A	[MAXA (J + 1) + (J + 1) - J]
$U_{J+1, J+1}$	=	A	[MAXA (J + 1)]
$K_{J,J}$	=	A	[MAXA (J)]

Note: In actual coding, the decomposed matrix [U] will overwrite the original stiffness matrix [K]. For clarity, however, these 2 matrices have been shown under different names.

4.4 Solution of Triangular Systems

It will be explained in subsequent sections that the forward and backward solutions (see Eqs. 4.4 and 4.5) both require only two nested do-loops, instead of three nested do-loops, as required during the factorization phase.

The forward elimination and backward substitution phases can be made parallel in the first loop (as it will be explained in Sections 4.4.1 and 4.4.2) which requires synchronization statements. This parallel implementation was tested and resulted in excellent computation speed-up for an increasing number of processors, but it suffered from the added time for synchronization on Cray computers. Because of this synchronization overhead time, the forward-backward solution phases were found to be faster on one Cray processor (without parallel constructs) than on multiple processors. Further time reduction for one processor was also obtained by using second-level vector unrolling in the forward elimination and second-level loop-unrolling in the backward substitution.

4.4.1 Forward solution

In the forward solution phase, since the factorized matrix [U] has already been found (in Section 4.3), the forward solution vector {y} can be solved from Eq. 4.4.

In order to derive the general formula for the unknown vector {y}, a simple system with only 3 unknowns will be considered in details, as shown in Eq. 4.30

$$\begin{bmatrix} u_{11} & 0 & 0 \\ u_{12} & u_{22} & 0 \\ u_{13} & u_{23} & u_{33} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \end{Bmatrix} \tag{4.30}$$

From the first part of Eq. 4.30, one can write

$$u_{11} y_1 = F_1 \tag{4.31}$$

Thus

$$y_1 = \frac{F_1}{u_{11}} \tag{4.32}$$

From the second part of Eq. 4.30, one can write

$$u_{12} y_1 + u_{22} y_2 = F_2 \tag{4.33}$$

Thus

$$y_2 = \frac{\left(F_2 - u_{12} y_1\right)}{u_{22}} \tag{4.34}$$

Similarly, from the last part of Eq. 4.30, one can write

$$u_{13} y_1 + u_{23} y_2 + u_{33} y_3 = F_3 \tag{4.35}$$

or

$$y_3 = \frac{F_3 - u_{13} y_1 - u_{23} y_2}{u_{33}} \tag{4.36}$$

In general, for any matrix size, the solution Eqs. 4.32, 4.34 and 4.36, can be written as

$$y_j = \frac{F_j - \sum_{i=1}^{j-1} u_{ij} y_i}{u_{jj}} \tag{4.37}$$

In order to see how the operations shown in Eq. 4.37 can actually be executed in parallel, Eq. 4.30 will be expanded into a system of 9 unknowns, as indicated in Eq. 4.38.

$$\begin{bmatrix} u_{11} & & & & & & & & \\ . & X & & & & & & & \\ . & & X & & & & & & \\ u_{41} & & & X & & & & & \\ u_{51} & & & & X & & & & \\ u_{61} & & & & & X & & & \\ u_{71} & & & & & & X & & \\ u_{81} & & & & & & & X & \\ u_{91} & & & & & & & & X \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \\ F_7 \\ F_8 \\ F_9 \end{Bmatrix} \begin{matrix} P_1 \\ P_2 \\ P_3 \\ P_1 \\ P_2 \\ P_3 \\ P_1 \\ P_2 \\ P_3 \end{matrix} \tag{4.38}$$

Assuming the matrix shown in Eq. 4.38 is full, and 3 processors (P_1, P_2 and P_3) are available, thus the unknowns y_1, y_4 and y_7 (in rows 1, 4 and 7) will be processed by processor P_1. Similarly, processors P_2 and P_3 will handle rows 2, 5, 8 and rows 3, 6, 9, respectively.

The unknowns, say y_4, y_5 and y_6 can be computed from Eq. 4.37 as

$$y_4 = \frac{F_4 - \left(u_{14} y_1 + u_{24} y_2 + u_{34} y_3\right)}{u_{44}} \tag{4.39}$$

$$y_5 = \frac{F_5 - \left(u_{15} y_1 + u_{25} y_2 + u_{35} y_3 + u_{45} y_4\right)}{u_{55}} \tag{4.40}$$

$$y_6 = \frac{F_6 - \left(u_{16} y_1 + u_{26} y_2 + u_{36} y_3 + u_{46} y_4 + u_{56} y_5\right)}{u_{66}} \tag{4.41}$$

From Eqs. 4.39 through 4.41, it is obvious to recognize that as soon as the first unknown, y_1, has been computed, the unknown y_2 can be "completely" determined, by using Eq. 4.34 and all other unknowns (say y_3, y_4, . . . , y_n) can be "partially" computed AT THE SAME TIME.

For example, the unknowns y_4, y_5 and y_6 can be "partially" and simultaneously (in parallel) computed as:

Incomplete $y_4 = F_4 - u_{14} y_1 \quad \ldots$ (computed by P_1) **(4.42)**

Incomplete $y_5 = F_5 - u_{15} y_1 \quad \ldots$ (computed by P_2) **(4.43)**

Incomplete $y_6 = F_6 - u_{16} y_1 \quad \ldots$ (computed by P_3) **(4.44)**

While the above parallel forward solution strategy may offer good parallel speeds (as the number of equations increased) synchronization checks are obviously required. For example, synchronization checks are needed to make sure that the 2nd unknown (= y_2) has been "completely" computed (say, by processor P_2) before the next term of Eqs. 4.42 through 4.44 can be simultaneously processed by processors P_1, P_2 and P_3 (as shown in Eqs. 4.45 through 4.47).

Incomplete $y_4 = F_4 - u_{14} y_1 - u_{24} y_2 \quad \ldots$ **(4.45)**

Incomplete $y_5 = F_5 - u_{15} y_1 - u_{25} y_2 \quad \ldots$ **(4.46)**

Incomplete $y_6 = F_6 - u_{16} y_1 - u_{26} y_2 \quad \ldots$ **(4.47)**

As it has been mentioned in Section 4.4, on many supercomputers, the synchronization overhead time will make the use of multiple processors to be slower than simply using a single processor with a good vectorized code. During the Choleski factorization phase, however, the synchronization overhead time is much less significant, since factorization phase requires a higher number of operations as compared to the forward solution phase. Because of these reasons, the following paragraphs will be devoted to the development of efficient vectorized codes for forward solution phase using only a single processor.

A straight forward implementation of Eq. 4.37 will lead to the computer code presented in Table 4.7. Explanations of different FORTRAN statements in Table 4.7 are given as follows:

1: The first loop with the index J will sweep from the first unknown (or the first row) to the last unknown (or the last row).

2: The summation in Eq. 4.37 is initialized.

3, 4: The summation of the product (see Eq. 4.37) is computed inside the second nested do-loop (with the index I).

5: The unknown y_J is computed according to Eq. 4.37.

6: The index J of the first do-loop is increased, for the computation of the next unknown.

It should be mentioned here that the forward solution vector {y} will overwrite the original right-hand-side vector {F}.

In Table 4.7, the stiffness matrix has been stored as a 2-D array and is assumed to be full. In actual computer coding, the stiffness matrix will be stored in a 1-D array, and according to skyline fashion (refer to Section 2.5). Assuming the one dimensional integer array, diagonal pointer MAXA(-) has already been computed from Eq. 2.12, and the column heights array ICOLH(-) has also been calculated from Table 2.1. The forward solution algorithm presented in Table 4.7 can be modified to the skyline storage

scheme as shown in Table 4.8. Explanations of different FORTRAN statements in Table 4.8 are given in the following:

1, 2: These first two statements have already been explained in statements 1 and 2 of Table 4.7.

3: Referring to Figure 4.10, and assuming that the first three unknowns have already been found. The fourth unknown (or J = 4th row) y(J = 4) can be found by Eq. 4.37 as

$$y_4 = \frac{F_4 - \left(u_{41} y_1 + u_{42} y_2 + u_{43} y_3\right)}{u_{44}} \tag{4.48}$$

In actual computer coding, one will store the factorized matrix in the upper triangular matrix, hence, Eq. 4.48 should be re-written as

$$y_4 = \frac{F_4 - \left(u_{14} y_1 + u_{24} y_2 + u_{34} y_3\right)}{u_{44}} \tag{4.49}$$

or

$$y_J = \frac{\left[F_j - \left(u_{1J} y_1 + u_{2J} y_2 + u_{3J} y_3\right)\right]}{u_{JJ}} \tag{4.50}$$

or

$$y_J = \frac{\left[F_j - \sum_{I=1}^{Icolh(J)} u_{IJ} y_I\right]}{u_{JJ}} \tag{4.51}$$

The number of terms inside the parenthesis of Eq. 4.49 is equal to the column height (excluding the diagonal term) of the Jth column. For this reason, the second do-loop (with the index I) in Table 4.8 will sweep from I = 1 to ICOLH(J), or from I = ICOLH(J) to 1 (with an increment-1).

4: The summation in Eq. 4.51 is computed. Notice that the first product inside the summation sign is

$$u_{14} * y_1 \tag{4.52}$$

Since the factorized stiffness matrix is expressed in a 1-D array U(-), expression in Eq. 4.52 can also be written as

$$U\left[MAXA\,(4) + 4 - 1\right] * y_1 \tag{4.53}$$

where the value 3 (= 4 - 1) in expression 4.53 represents the column height of the 4th column in Figure 4.10 or, using the index I and J notations, expression 4.53 becomes

$$U\left[Maxa\,(J) + I\right] * y\,(J - I)$$

which is precisely the expression used in statement 4 of Table 4.8.

5: The final, complete unknown solution y(J) can be found from Eq. 4.51. Recalling U_{JJ} = U[Maxa(J)].

6: The index J of the first do-loop is increased (for the computation of the next unknown).

The forward solution algorithm presented in Table 4.8 can be further modified to incorporate unrolling techniques (similar ideas have been presented in Section 1.3 of Chapter 1), as shown in Table 4.9 and the associated Figure 4.11.

Explanations of different FORTRAN statements in Table 4.9 are given in the following (assuming that the unrolling of level 2 is used).

1, 2: The first unknown y_1, will be solved separately. Since the original stiffness and its factorized matrix is symmetrical, the row or its correspondent column is merely the image of each other.

3: The first do-loop with the index J will sweep from column (or its imaged row) 2 through the last column NEQ, with the increment 2 (since unrolling level 2 is used, hence a "block" of 2 columns are grouped together). It is important here to refer to Figure 4.11 where columns J(= 4) and J + 1 (= 5) are grouped together. For reasons which will soon be explained, any column within a group must have a column height of 1 unit more than its previous column height.

In the example presented in Figure 4.11, since both columns 4 and 5 have the column height of 3 (excluding diagonal terms), one needs to add 1 extra zero term on to the top of the 4^{th} column to make its column height becomes 4. Thus, the 4^{th} column height is 1 unit more than the 3^{rd} column height within the same group.

4, 5: Two summations (SUM1 and SUM2) are initialized.

6: The second do-loop with the index I will sweep from the column height of the J^{th} column down to value 1, with the increment of -1. This statement has essentially the same role as statement 3 of Table 4.8.

7 - 9: Since the unrolling of level 2 is used in the first outermost do-loop, the index J has the increment of 2, which implies that 2 consecutive columns (or its imaged 2 consecutive rows) are considered at a time. Because of this reason, there are 2 summations (not just one summation, as shown in statement 4 of Table 4.8) to be calculated inside the second do-loop, with the index I.

At this point, it should be obvious to understand there is a need to add extra zero(s) to some column height(s), as mentioned earlier in statement 3. By adding the appropriate extra zero(s) to some column(s), it will make the operations within the 2^{nd} loop, with index I, to be done properly. For example, considering the case where the index J of the first do-loop has a fixed value, say J = 4, it is *NOT* possible for the index I to have the values 3 (= column height of the 4^{th} column) through 1 with increment -1 (or rows 3 through 1 of column J = 4) and the same index I to have the values 3 through 2 (or rows 3 through 2 of column J+1 = 5)!

By adding 1 extra zero (in the example of Figure 4.11) to column 5, it is now possible for the index I to have the same values 3 through 1 (or rows 3 through 1) with increment -1, in both columns J (= 4) and J+1 (= 5).

Obviously, the extra term $u_{J,\,J+1} = u_{4,\,5} = U[Maxa(J+1)+1]$ will have to be included separately when we compute the unknown y_{J+1} (refer to statement 11).

10: The forward solution unknown y_j can now be calculated according to Eq. 4.37.

11: The forward solution unknown y_{J+1} can also be calculated now, by referring to Eq. 4.37. Also, referring to Figure 4.11, one can write the following expression to calculate y_{J+1} or y_5:

$$y_5 = \frac{F_5 - \left(u_{51} y_1 + u_{52} y_2 + u_{53} y_3 + u_{54} y_4\right)}{u_{55}} \tag{4.54}$$

or, in general

$$y_{J+1} = \frac{F_{J+1} - \sum_{i=1}^{J} u_{J+1,i} * y_i}{u_{J+1,J+1}} \tag{4.55}$$

The product $u_{5,4} * y_4$, or $u_{4,5} * y_4$, or $u_{J,J+1} * y_J$, or u[Maxa(J+1)+1] * y_J has not yet been included in the calculation of SUM2 in statement 8. That is the reason we have to modify the value of SUM2 to include the product term u[Maxa(J+1)+1] * y_J in here.

12: The index J of the first do-loop is increased (by the increment 2) for the computation of the next 2 unknowns.

Table 4.7 Basic scheme for forward solution

```
1.          DO 1 J = 1, NEQ
2.          SUM1 = 0.
3.          DO 2 I = 1, J - 1
4.      2   SUM1 = SUM1 + U(I, J) * Y(I)
5.          Y(J) = (Y(J) - SUM1) / U(J,J)
6.      1   CONTINUE
```

Table 4.8 Skyline scheme for forward solution

```
1.          DO 1 J = 1, NEQ
2.          SUM1 = 0.
3.          DO 2 I = COLH(J), 1, - 1
4.      2   SUM1 = SUM1 + U [MAXA(J) + I] * Y(J - I)
5.          Y(J) = (Y(J) - SUM1) / U (MAXA(J))
6.      1   CONTINUE
```

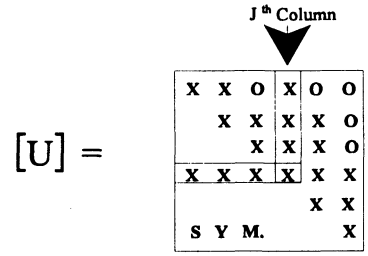


Figure 4.10 Skyline scheme for forward solution

Table 4.9 Skyline and vector-unrolling scheme for forward solution

```
Assuming NEQ = number of equations, say an odd number
1.      C . . .     TREAT COLUMN 1 SEPARATELY
2.          Y(1) = Y(1) / U (MAXA (1))
3.          DO 1      J = 2, NEQ,  2
4.          SUM1 = 0.
5.          SUM2 = 0.
6.          DO 2      I = COLH(J), 1, -1
7.          SUM1 = SUM1 + U (MAXA(J) + I) * Y (J - I)
8.          SUM2 = SUM2 + U (MAXA(J + 1) + 1 + I) * Y (J - I)
9.      2   CONTINUE
10.         Y(J) = (Y(J) - SUM1) / U (MAXA (J))
11.         Y(J+1) = [Y(J+1) - SUM2 - U (MAXA (J+1) + 1) * Y(J)] /
            U [MAXA (J+1)]
12.     1   CONTINUE
```

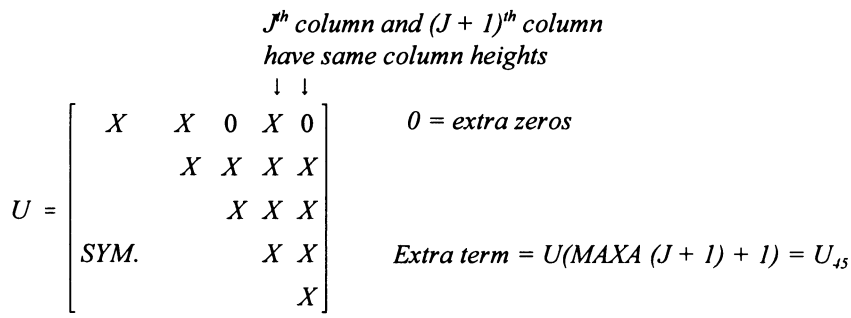


Figure 4.11 Skyline and vector unrolling scheme for forward solution

4.4.2 Backward solution

In the backward solution phase, since the factorized matrix [U] and the forward solution vector {y} have already been found, the backward solution vector {Z} can be obtained from Eq. 4.5.

In order to derive the general formula for the unknown vector {Z}, a simple system with only 4 unknowns will be considered in detail, as shown in Eq. 4.56

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \begin{Bmatrix} Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{Bmatrix} \tag{4.56}$$

The last part of Eq. 4.56 can be used to solve for the last unknown Z_4, as following

$$Z_4 = \frac{y_4}{u_{44}} \tag{4.57}$$

Similarly, the unknowns Z_3, Z_2, and Z_1 can also be obtained from the 3rd, the 2nd and the 1st parts of Eq. 4.56, respectively.

$$Z_3 = \frac{y_3 - u_{34} Z_4}{u_{33}} \tag{4.58}$$

$$Z_2 = \frac{y_2 - \left(u_{23} Z_3 + u_{24} Z_4\right)}{u_{22}} \tag{4.59}$$

and

$$Z_1 = \frac{y_1 - \left(u_{12} Z_2 + u_{13} Z_3 + u_{14} Z_4\right)}{u_{11}} \tag{4.60}$$

In general, for any matrix size, one can write:

$$Z_j = \frac{y_j - \sum_{i=j+1}^{N} u_{ji} Z_i}{u_{jj}} \tag{4.61}$$

As soon as the last unknown Z_4 has been "completely" solved, the "partial" solutions for all remaining unknowns can be computed simultaneously, by different parallel processors. For example, assuming the unknown Z_3 has not been completely computed yet, one can simultaneously compute the "partial" solution for

Incomplete $Z_2 = y_2 - \left(u_{24} Z_4 \quad \ldots . \right)$ (4.62)

Incomplete $Z_1 = y_1 - \left(u_{14} Z_4 \quad \ldots . \right)$ (4.63)

While the above parallel backward solution may offer good parallel speeds (as the number of equations increased) synchronization checks are needed to make sure that, say the 3rd unknown (= Z_3) has been "completely" computed (say by a particular processor) before the next term of Eqs. 4.62 and 4.63 can be simultaneously processed by the other processors

Incomplete $Z_2 = y_2 - \left(u_{24} Z_4 + u_{23} Z_3 \quad \ldots . \right)$ (4.64)

Incomplete $Z_1 = y_1 - \left(u_{14} Z_4 + u_{13} Z_3 \quad \ldots . \right)$ (4.65)

The above parallel backward solution strategy, however, suffers significant overhead time for synchronization checks (refer to the parallel forward solution phase discussed in Section 4.4.1). Because of these reasons, the following paragraphs will be

devoted to the development of efficient vectorized code for backward solution, using only a single processor.

In actual computer coding, once the (last) unknown has been found, the right-hand-side vector {y} in Eq. 4.56 can be updated as follows:

$$\bar{y}_3 = y_3 - u_{34} Z_4 \tag{4.66}$$

$$\bar{y}_2 = y_2 - u_{24} Z_4 \tag{4.67}$$

$$\bar{y}_1 = y_1 - u_{14} Z_4 \tag{4.68}$$

In practice, the solution vector {Z} will overwrite the right-hand-side vector {y}. A straight forward implementation of Eq. 4.61 will lead to the computer code presented in Table 4.10.

Explanations of different FORTRAN statements in Table 4.10 are given in the following:

1: The first do-loop with index J will sweep from the last unknown (or the last column) to the first unknown (or the first column) with the increment -1. Since the original stiffness matrix and its factorized matrix is symmetrical, the J^{th} column and the J^{th} row are identical.

2: The (last) unknown y_J, (i.e. y_4) is finally computed (see Eq. 4.57).

3: The second do-loop with index I will sweep from row #J-1 (= row 3) to the Top Row # of column J (= row 1, assuming the stiffness matrix is full), with the increment -1.

4: The right-hand-side vector {y} is updated (or modified) according to the nominator of Eq. 4.61, or for the specific case of the matrix shown in Eq. 4.56, according to Eqs. 4.66 through 4.68.

5: The index J of the first do-loop is *decreased* by 1, in order to compute subsequent unknowns y_{J-1} (= y_3), y_{J-2} (= y_2), etc. . . .

To enhance the vector speed in the 2nd (inner-most) do-loop, "loop unrolling" technique can be incorporated (refer to Section 1.3 of Chapter 1) into the backward solution phase, as shown in Table 4.11, where unrolling level 2 (or a "block" of 2 columns are grouped together) is demonstrated.

Explanations of different FORTRAN statements in Table 4.11 are given in the following. For a better understanding of the algorithm presented in Table 4.11, the readers are also encouraged to specifically refer to Eq. 4.56. The column heights of columns 1, 2, 3, and 4 (see Eq. 4.56) are assumed to be 0, 1, 2, and 3, respectively. Thus:

$$ICOLH \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{Bmatrix} \tag{4.69}$$

1: This statement has the same role as statement 1 of Table 4.10. The only difference is that the increment of -2 is used here (due to the unrolling of level 2) instead of -1 as used earlier.

2, 3: The last 2 unknowns y_J and y_{J-1} (say y_4 and y_3) are finally and "completely" computed (see Eq. 4.57). Notice that vectors {y} and {Z} are overwritten on each other.

4: The index I of the second do-loop will sweep from J - ICOLH(J), or 4 - ICOLH(4) = 4 - 3 = 1, to J - 2, or 4 - 2 = 2.

5: The "partial" solution for the unknowns y_1 and y_2 are computed, as follows (notice that vectors {y} and {Z} are overwritten on each other).

$$y_1 = y_1 - u_{1,4} * y_4 - u_{1,3} * y_3 \quad \textbf{(4.70)}$$

$$y_2 = y_2 - u_{2,4} * y_4 - u_{2,3} * y_3 \quad \textbf{(4.71)}$$

It should be emphasized here, that the last products $u_{1,3} * y_3$ in Eq. 4.70, and $u_{23} * y_3$ in Eq. 4.71 are included here (but have not been included in statement 4 of Table 4.10) as a direct consequence of the use of unrolling level 2 (or, increment -2) for index J in statement 1..

6: The index J of the first do-loop is decreased by -2, in order to compute subsequent "block" unknowns y_{J-2} (= y_2) and y_{J-3} (= y_1), and etc. . . .

Table 4.10 Basic scheme for backward solution

```
1.          DO 1 J = N, 1, -1              (say, J = 4)
2.          Y(J) = Y(J) / U(J, J)
3.          DO 2 I = J - 1, TOP ROW OF COLUMN J, -1
4.      2   Y(I) = Y(I) - U(I, J) * Y(J)
5.      1   CONTINUE
```

Table 4.11 Loop-unrolling scheme for backward solution

```
1.          DO 1 J = N, 1, -2
2.          Y(J) = Y(J) / U(J, J)
3.          Y(J - 1) = [Y (J - 1) - U (J - 1, J) * Y (J)] / U (J - 1, J - 1)
4.          DO 2 I = J - ICOLH (J), J - 2, +1
5.      2   Y(I) = Y(I) - U(I, J) * Y(J) - U(I, J - 1) * Y(J - 1)
6.      1   CONTINUE
```

4.5 Force: A Portable, Parallel FORTRAN Language

Force is a machine-independent tool for parallel programming and, with certain exceptions noted in the "Force User's Manual"[4.7], it includes all FORTRAN constructs and compiler options. It is a preprocessor which produces executable parallel code from FORTRAN augmented with several simple parallel extensions. These parallel extensions include such constructs as Pre- and Self-scheduled DO-loops (for parallel computations), Barriers, Produce and Consume (for synchronization). Force permits

users to write efficient, yet portable, parallel code without referring to the many details of multitasking or parallel programming found in vendor manuals. Thus, engineers and numerical analysts can concentrate on developing effective parallel algorithms or solution strategies for different engineering and/or scientific applications. Programs written in Force are easily ported to and run on other parallel computers on which Force is installed. In this chapter, Force is used (for convenient purpose only) to develop and implement the parallel FORTRAN code on high-performance computers, such as the Convex, Cray-2, Cray-YMP, and Cray C-90. The developed parallel-vector algorithms can also be implemented in other parallel environments such as PVM[4.10], or MPI (Message Passing Interface) [4.11] etc...

4.6 Evaluation of Methods on Example Problems

To test the effectiveness of the parallel-vector skyline Choleski solver, several structural analysis problems were solved on various high performance computers. Furthermore, for user's convenience, a simple algorithm to automatically generate the coefficient stiffness matrix [A] and load vector {B} for solving the unknown vector {x} from [A] {x} = {B} is also shown in Table 4.12.

In the solution of the following example problems, code was inserted to measure the time spent by each processor during the equation solution. The Cray timing function, tsecnd, was used to measure the total computation time used by each Force processor. In addition, for each problem, the number of **m**illion **fl**oating point **op**erations, MFLOP, was calculated and then divided by the solution time, in seconds, to determine the overall performance rate of the solver in MFLOPS.

Table 4.12 Code for stiffness and load generation

```
1.              MAXA(1) = 1
2.              A(1) = 2.
3.              DO 51 I = 2, NEQ
4.              COLHT = MIN(I - 1, HALFBW)
5.              MAXA(I) = MAXA(I - 1) + COLHT
6.              A(MAXA(I)) = 2.
7.      51      B(I) = 2.
8.              IEND = MIN(I + HALFBW,NEQ)
9.              DO 52 I = 1,NEQ
10.             DO 52 J = I + 1, IEND
11.             LOCATE = MAXA(J) + J - I
12.             A(LOCATE) = 1.0 / (I+J)
13.             B(I) = B(I) + A(LOCATE)
14.             B(J) = B(J) + A(LOCATE)
15.     52      CONTINUE
```

Example 4.1: 10,000 degree-of-freedom, 800 Bandwidth Test Problem
The parallel-vector supercomputer available for testing the column-oriented skyline Choleski solver was the Cray-YMP.

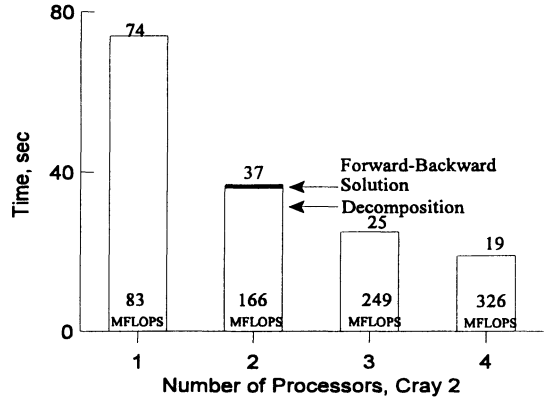


Figure 4.12 Computation time reduction for test problem with 10,000 equations and 800 bandwidths

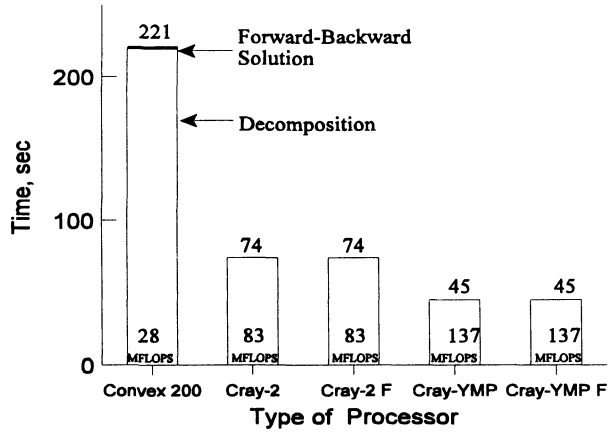


Figure 4.13 Time comparison for one processor with 10,000 equations and 800 bandwidths. (F denotes Force used)

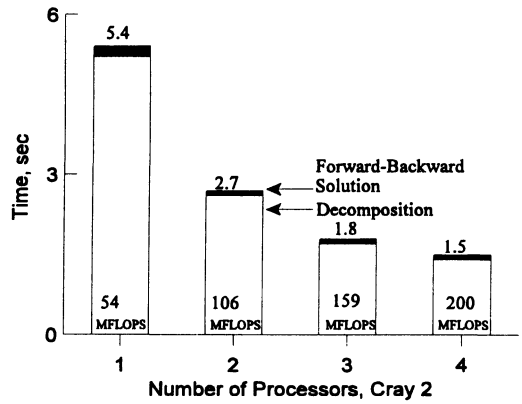


Figure 4.14 Computation time reduction for 3000 equation (cube)

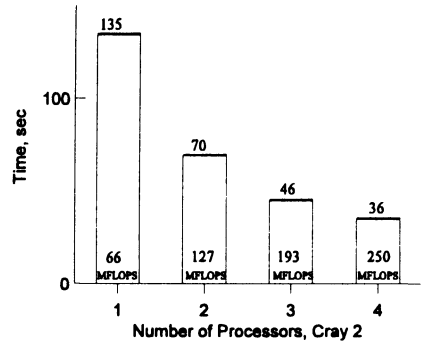


Figure 4.15 Computation time reduction for space shuttle solid rocket booster with 54,870 equations

Although it is fast and has eight processors, the Cray-YMP used has only eight million words of main memory available for a single job. This maximum eight million words restricted the size of the largest problem that could be solved since the equation solvers described are currently implemented only for main (in-core) memory storage of [K]. Thus, the largest problem possible to solve on all three high performance computers was designed to have a stiffness coefficient matrix with 10,000 degrees-of-freedom (DoF) and 800 bandwidth. In this test problem, the coefficient matrix A (stored in a vector skyline form) and the load vector B are generated according to Table 4.12. In this table, NEQ represents the number of equations, or the Degree-of-Freedom (DoF). The integer array MAXA is used to store the location of the diagonal terms of the stiffness matrix. This test problem is useful for debugging and can also serve to evaluate the performance of this and other methods on any shared memory parallel-vector computer.

The total processing time (of both the decomposition and the forward-backward solution phase) for the test problem on the Cray-2 using 1, 2, 3, and 4 processors is shown in Figure 4.12. The figure indicates a computation speedup of 3.9, (74 seconds divided by 19 seconds), at a rate of 328 MFLOPS on four processors. The major computation time was spent in the decomposition phase, while very little time was spent in the forward-backward solution phase as indicated in Figure 4.12.

The total computation time for this test problem on the Convex-220, Cray-2 and Cray-YMP computers is shown in Figure 4.13. From this figure, one can see the relative performance of the skyline Choleski solution method on a single processor for the three computers. Here again the forward-backward solution phase represents a very small fraction of the total solution time. Figure 4.13 also indicates that the Cray-YMP runs almost twice as fast as the Cray-2, which in turn runs about three times as fast as the Convex-220. By using Force (see Section 4.5) the same code was implemented on the three computers. The code is running on the Cray-YMP in multitasking environment (using 1 through 8 processors). The Cray-YMP multitasking timing subroutine is not fully developed, thus accurate timings for eight processors on the Cray-YMP are not known. However, preliminary indications are that this method achieves computation speedups on the Cray-YMP similar to those achieved on the Cray-2.

To measure the overhead of Force, the same code, with all Force statements removed, was implemented on the Cray-2 and Cray-YMP. The results obtained using Force on the Cray-2 and Cray-YMP (denoted Cray-2(F) and Cray-YMP(F) in Figure 4.13) do not introduce any significant additional overhead in computation time.

Example 4.2: Three-Dimensional Cube Problem

Solution algorithms often display different characteristics on different classes of problems. To investigate the behavior of the solver on the three-dimensional problem, a cube-shaped, isotropic solid undergoing compression was solved. This 3000 degree of freedom problem has a maximum bandwidth of 336 and average bandwidth of 313. The 1000 node cube (10 nodes along each axis) was constrained at the corner nodes and contained 729 eight-node solid elements (nine solid elements along each axis). The computation times for multiple processors are shown in Figure 4.14. The computation time reduction is in direct proportion to the number of processors used, with a speedup of 3.6 for four processors. The result from a vectorized code on one processor is also given in Figure 4.14.

Example 4.3: Space Shuttle Solid Rocket Booster (SRB) Problem

To evaluate the performance of the skyline Choleski solver on a large-scale static structural analysis problem, a two-dimensional shell model of the Space Shuttle solid rocket booster was solved. This SRB model was used to investigate the overall deflection distribution for the SRB when subjected to mechanical loads corresponding to selected times during the launch sequence [4.8]. The model contains 9205 nodes, 9156 four-node quadrilateral shell elements, 1273 two-node beam elements and 90 three-node triangular elements, with a total of 54,870 degrees of freedom. This problem has a maximum bandwidth of 894 and an average bandwidth of 381. A detailed description and analysis of this problem is given in reference [4.8 and 4.9].

For this problem, the parallel-vector skyline Choleski method developed in this chapter took 135 and 36 seconds on one and four processors, respectively.

By using four processors, the speed up obtained by the skyline Choleski method was found to be 3.75 as can be seen in Figure 4.15. This good speedup of the skyline Choleski method makes it attractive for supercomputers with more than four processors, such as the Cray-YMP, or the Cray C-90.

The computation rate (i.e., MFLOPS) shown in Figures 4.12-4.15, is best for problems with a large average bandwidth (i.e. Figure 4.12). The skyline Choleski solver operates on vectors ranging in length from one to the column height of each column of the upper triangular matrix [U]. For problems with a small average column height, the major portion of the computation is performed on short vectors which results in a low MFLOPS rate. In large structural analysis problems with large average column height, the majority of the vector operations are performed on long vectors, which results in higher MFLOPS rates.

4.7 Skyline Equation Solver Computer Program

For the complete listing of the FORTRAN source codes, instructions in how to incorporate this equation solver package into any existing application software (on any specific computer platform), and/or the complete consulting service in conjunction with this equation solver etc... the readers should contact:

Prof. Duc T. Nguyen
Director, Multidisciplinary Parallel-Vector Computation Center
Civil and Environmental Engineering Department
Old Dominion University
Room 135, Kaufman Building
Norfolk, VA 23529 (USA)
Tel = (757) 683-3761, Fax = (757) 683-5354
Email = dnguyen@odu.edu

4.8 Summary

A portable and efficient skyline Choleski method for the solution of large-scale structural analysis problems has been developed and tested on three high performance computers - Convex-220, Cray-2 and Cray-YMP. The newly-developed equation solver exploits both parallel and vector capabilities of modern high performance computers. A unique feature of this method is the strategy used to minimize computation time by performing parallel computation at the outermost DO-loop of the decomposition phase, the most time-consuming part of the total equation solution time. In addition, the most intensive computation of the decomposition phase was vectorized at the innermost DO-

loop. The dot-product-based factorization scheme prohibits the traditional use of the well-known loop-unrolling technique used for saxpy operations. To overcome this difficulty, a novel use of "vector unrolling" has been introduced in the column-oriented Choleski algorithm to reduce computation time. For the forward and backward solution phases, it was found to be more effective to perform vector-unrolling and loop unrolling, respectively, using vector rather than parallel code.

The new method was coded in a modular, and portable fashion using a generic parallel FORTRAN, called Force. The generality and portability of the method should make the use of it attractive for other engineering and scientific applications.

The newly-developed parallel-vectorized Choleski method has been applied to the solution of several small- to large-scale structural analysis problems. For all problems, the total equation solution time was reduced significantly, in direct proportion to the number of processors used.

4.9 Exercises

4.1 Given the following 21 x 21 symmetrical matrix [K]

$$[K] = \begin{array}{cccccccccccccccccccccc}
 & 11. & .2 & 0. & .4 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & & & & & & & & & \\
 & & 55. & .6 & .7 & .8 & .9 & 0. & 0. & 0. & 0. & 0. & 0. & & & & & & & & & \\
 & & & 110. & .11 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & & & & & & & & & \\
 & & & & 114. & .15 & .16 & 0. & 0. & .19 & 0. & 0. & 0. & & & & & & & & & \\
 & & & & & 620. & .21 & .22 & 0. & .24 & 0. & 0. & 0. & & & & & & & & & \\
 & & & & & & 725. & .26 & 0. & .28 & 0. & 0. & 0. & & & & & & & & & \\
 & & & & & & & 829. & .3 & .31 & 0. & 0. & 0. & & & & & & & & & \\
 & & & & & & & & 932. & .33 & 0. & 0. & 0. & & & & & & & & & \\
\text{Row 9} & & & & & & & & & 934. & .35 & .36 & .37 & & & & & & & & & \\
 & & & & & & & & & & 938. & 0. & .4 & .41 & .42 & .43 & 0. & 0. & 0. & 0. & 0. \\
 & & & & & & & & & & & 104. & .45 & .46 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
 & & & & & & & & & & & & 214. & .48 & .49 & 0. & 0. & 0. & 0. & 0. & 0. \\
 & & & & & & & & & & & & & 225. & .51 & .52 & .53 & 0. & 0. & 0. & 0. \\
 & & & & & & & & & & & & & & 315. & 0. & .56 & .57 & 0. & 0. & 0. \\
 & & & & & & & & & & & & & & & 415. & 0. & .59 & 0. & 0. & 0. \\
 & & & & & & & & & & & & & & & & 616. & .61 & .62 & .63 & .64 \\
 & & & & & & & & & & & & & & & & & 716. & .66 & 0. & .67 \\
\text{Row 18} & & & & & & & & & & & & & & & & & & 968. & .69 & 0. \\
 & & & & & & & & & & & & & & & & & & & 887. & 71 & 0. \\
 & & & & & & & & & & & & & & & & & & & & 777. & .73 \\
 & & & & & & & & & & & & & & & & & & & & & 897
\end{array}$$

a. Find the column height (integer) array ICOLH(-) of the above matrix.
b. Find the diagonal location (integer) array MAXA(-) of the above matrix.
c. How many (real) words of computer memories are required to store the above matrix?

4.2 For the matrix [K] given in Problem 4.1:
a. Find (using hand calculator) the first 4 columns of the factorized matrix [U] using the Choleski algorithm.
b. The original term $K_{35} = 0.$, without any actual computation, explain your reason(s) for saying the corresponding factorized term $U_{35} = 0.??$, or $U_{35} \neq 0.??$.

c. The original term $K_{11,14} = 0.$, without any actual computation, explain your reason(s) for saying the corresponding factorized term $U_{11,14} = 0.??$, or $U_{11,14} \neq 0.??$.

4.3 For the matrix [K] given in Problem 4.1, and assuming "vector unrolling" level 3 is used:

a. How many "additional" (real) words of computer memories will be required due to vector unrolling's (level 3) strategies?

b. Assuming 4 processor (P_1, P_2, P_3, and P_4) are used in this example, and according to the following given information

Processor	"Columns" of Matrix [K] Which Belong To a Processor
P_1	1, 2, 3, 13, 14, 15
P_2	4, 5, 6, 16, 17, 18
P_3	7, 8, 9, 19, 20, 21
P_4	10, 11, 12

Without any actual computation, and assuming the first 5 columns of the factorized matrix [U] have already been completely factorized, identify which terms (if any) U_{ij} of the matrix [U] can be factorized by processors P_1, P_2, P_3 and P_4, respectively?

4.10 References

4.1 Storaasli, O.O., D.T. Nguyen, and T.K. Agarwal, "Parallel-Vector Solution of Large-Scale Structural Analysis Problems on Supercomputers," *AIAA Journal*, Vol. 28, No. 7, July 1990, pp.1211-1216 (also in *Proceedings of the 30th AIAA/ASME/ASCE/AHS Structures, Structural Dynamics and Materials Conference*, Mobile, AL, April 3-5, 1989, pp.859-867).

4.2 Qin, J. and D.T. Nguyen, "A New Parallel-Vector Finite Element Analysis Software on Distributed Memory Computers," *Proceedings of the AIAA/ASME/ASCE/AHS 34th SDM Conference*, La Jolla, CA, April 19-22, 1993.

4.3 Nguyen, D.T. and K.T. Niu, "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.

4.4 Nguyen, D.T., J.S. Shim, and Y. Zhang, "The Component Mode Method In a Parallel Computer Environment," *Proceedings of the 29th AIAA/ASME/ASCE/AHS Structures, Structural Dynamics and Materials Conference*, Williamsburg, VA, April 18-20, 1988, AIAA Paper No. 88-2395.

4.5 Qin, J. and D.T. Nguyen, "A Parallel-Vector Simplex Algorithm on Distributed Memory Computers," accepted to appear in Structural Optimization Journal (1996).

4.6 Bathe, K.J., Finite Element Procedures, Prentice-Hall (1996).

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

4.8 Knight, N.F., R.E. Gilliam, and M.P. Nemeth, "Preliminary 2-D Shell Analysis of the Space Shuttle Solid Rocket Boosters," *NASA TM-100515*, 1988.

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

4.10 Beguelin, A., Dongarra, J., Geist, G.A., Manchek, R. and Sunderam, V., "A User's Guide to PVM: Parallel Virtual Machine," Technical Report TM-11826, ORNL, 1991.

4.11 William D. Gropp and Ewing Lusk. A test implementation of the MPI draft message-passing standard. Technical Report ANL-92/47, Argonne National Laboratory, December 1992.

5 Parallel - Vector Variable Bandwidth Equation Solver on Shared Memory Computers

5.1 Introduction

In the previous chapter, parallel and vectorized Choleski algorithms which were based on the skyline (column-by-column) storage scheme for shared memory computers (such as the Cray-2, Cray-YMP, Cray-C90, etc....) had been discussed. The factorized algorithms discussed in Chapter 4 have been based upon the "dot product" operations. For certain types of shared memory computers (such as Cray-YMP, Cray-C90, etc....), "Saxpy" operations (to be explained in more detail, later on in this chapter) are known to be faster than "dot product" operations[5.1]. The skyline storage scheme and its associated parallel and vectorized algorithms has been found to prohibit the traditional "loop unrolling" technique used to optimize vector performance, so a less powerful "vector unrolling" strategy was used. This chapter describes a different algorithm that overcomes the deficiency of skyline storage by using a variable bandwidth storage scheme. The objective of this chapter 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 shared memory (such as Cray) supercomputers.

5.2 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]. Two methods most often used in structural analysis codes to store [U] are the variable-band, and skyline storage schemes.

For large finite-element applications, the user defines the geometry, finite elements and loads of the finite-element model[5.2]. 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[5.3] reorders the [K] matrix into a near minimum bandwidth, and thus is used for the examples in this chapter.

In a row-oriented, variable-bandwidth Choleski approach, the bandwidth of each row of the upper-triangular matrix [U], is generally defined as the number of coefficients from a diagonal term to the last non-zero coefficient of the row, excluding the diagonal term. Exceptions to this definition, however, can be found in row 3 of

Figure 5.3 (and is also indicated in row 3 of Eq. 5.4). The coefficients of the stiffness matrix for a stiffened panel with a circular cutout (bottom of Figure 5.1), are plotted in a variable-band format as shown in Figure 5.1.

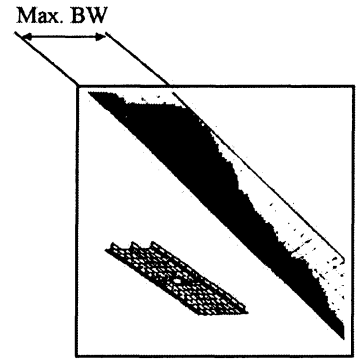


Figure 5.1 Variable-band row storage of panel matrix

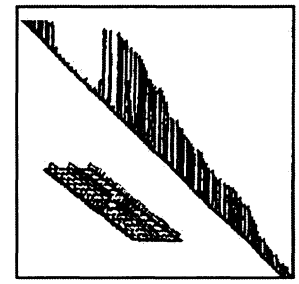


Figure 5.2 Skyline column storage of panel matrix

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 Figure 5.1. Variable-band storage requires less memory than earlier schemes which stored all coefficients within the maximum bandwidth, (refer to the distance Max. BW, shown in Figure 5.1), since earlier schemes stored and operated on many zeros outside the variable-band profile (see the double cross hatch region in Figure 5.1).

The same panel stiffness matrix is stored by columns in the skyline format, like skyscrapers, in Figure 5.2 from each diagonal coefficient up to the last nonzero directly above it.

In the 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 Figure 5.2. This skyline format requires fewer coefficients to store and operate on during equation solution as indicated by the many zeros (white spaces) in Figure 5.2. 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 (as compared to variable bandwidth storage scheme).

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, which stands for Summation of **AX** Plus **Y**, (i.e., $\sum$ ax + y, or scalar * vector + vector). Details on the skyline (column-by-column) storage scheme and its associated Choleski factorization, forward and backward solution algorithms have already been fully discussed in Chapter 4. SAXPY operations achieve optimal performance on vector computers since they continually stream operations to separate add and multiply units which can operate simultaneously.

$$\begin{bmatrix} 1 & 2 & 0 & 4 & & & & & & \\ & 5 & 6 & 7 & 8 & 9 & & & & \leftarrow\ k = 2 \\ & & 10 & 11 & 0 & 0 & & & & \\ & & & 14 & 15 & 16 & 0 & 0 & 19 & \\ & & & & 20 & 21 & 22 & 0 & 24 & \leftarrow\ k = 5 \\ & & & & & 25 & 26 & 0 & 28 & \leftarrow\ i = 6 \\ & & & & & & 29 & 30 & 31 & \\ & & & & & & & 32 & 33 & \\ & & & & & & & & 34 & \end{bmatrix}$$

Figure 5.3 Variable-band storage of stiffness matrix

To better understand the variable bandwidth storage schemes in greater detail, the location of the coefficients in the upper half of a 9x9 symmetric stiffness matrix are shown in Figure 5.3 as a simple illustrative example. The non-zero integers in Figure 5.3 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 (rows 1, 3, 4, 5, and 6). The skyline storage scheme requires only 29 locations to store the same matrix, since the five zeros in columns 3, 7 and 8 in Figure 5.3 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. Using numerical data shown in Figure 5.3, and referring to Equation 4.9 in Chapter 4, one has

$$u_{35} = \frac{K_{35} - u_{23} * u_{25}}{u_{33}} \qquad \textbf{(5.1)}$$

$$u_{36} = \frac{K_{36} - u_{23} * u_{26}}{u_{33}} \qquad \textbf{(5.2)}$$

Thus, even though the original values for K_{35} and K_{36} are zeros, their correspondent factorized matrix u_{35} and u_{36} are NOT zeros, as indicated in Eqs. 5.1 and 5.2. The bandwidth of row 2 in Figure 5.3 is 4, excluding the diagonal coefficient, and the height of column 6 is 4, excluding the diagonal coefficient.

Using the simple stiffness matrix example shown in Figure 5.3, one can compute the column height (integer) array, for variable bandwidth storage scheme, as follows:

$$ICOLH \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{pmatrix} = \begin{Bmatrix} 0 \\ 1 \\ 1 \\ 3 \\ 3 \\ 4 \\ 2 \\ 1 \\ 5 \end{Bmatrix} \tag{5.3}$$

The variable bandwidth, or row length, of each row associated with the example shown in Figure 5.3 can be computed as:

$$IROWL \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{pmatrix} = \begin{Bmatrix} 3 \\ 4 \\ 3 \\ 5 \\ 4 \\ 3 \\ 2 \\ 1 \\ 0 \end{Bmatrix} \tag{5.4}$$

In actual computer code implementation, the stiffness matrix [K] is usually stored in a one-dimensional array (instead of using a two-dimensional array as shown in Figure 5.3) in the following row-by-row fashion

$$\{K\}^T = \begin{Bmatrix} 1,2,0,4,5,6,7,8,9,10,11,0,0,14,15,16,0,0,19, \\ 20,21,22,0,24,25,26,0,28,29,30,31,32,33,34 \end{Bmatrix} \tag{5.5}$$

The numerical values of diagonal terms for the (two dimensional array) stiffness matrix can be identified as (refer to Figure 5.3):

$$\left.\begin{aligned} K_{11} &= 1 \\ K_{22} &= 5 \\ K_{33} &= 10 \\ &\vdots \\ K_{99} &= 34 \end{aligned}\right\} \tag{5.6}$$

In the corresponding one-dimensional array, the above nine diagonal terms will be located at the 1st, 5th, 10th, 14th, 20th, 25th, 29th, 32nd, and 34th positions of Eq. 5.5. The "diagonal pointer" (integer) array MAXA, therefore, can be defined as:

$$MAXA \left\{ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \quad = N \\ \hline 10 \quad = N+1 \end{array} \right\} = \left\{ \begin{array}{c} 1 \\ 5 \\ 10 \\ 14 \\ 20 \\ 25 \\ 29 \\ 32 \\ 34 \\ \hline 35 \end{array} \right\} \tag{5.7}$$

The integer MAXA array can, therefore, be used as the "mapping" between the two-dimensional and the one-dimensional arrays stiffness matrix.

The column heights integer array ICOLH can be computed from any given finite element models, according to the procedures described in Section 2.9 of Chapter 2. Once the column height information is known, the row length (IROWL) array shown in Eq. 5.4 can be easily calculated (the reader may refer to a segment of Subroutine CSMIN, given in Section 5.12 of this chapter for detailed computer codings).

The array MAXA, shown in Eq. 5.7, can be conveniently computed from IROWL according to the following formulas

$$MAXA(1) = 1 \qquad (always\ !) \tag{5.8}$$

$$MAXA(i+1) = MAXA(i) + IROWL(i) + 1 \; for \; i = 1, 2, \ldots N \tag{5.9}$$

$$MAXA(N+1) = MAXA(N) + 1 \tag{5.10}$$

In Eqs. 5.9 and 5.10, N represents the total number of equations. The total number of terms (or total number of memory requirements, in words) can be computed as

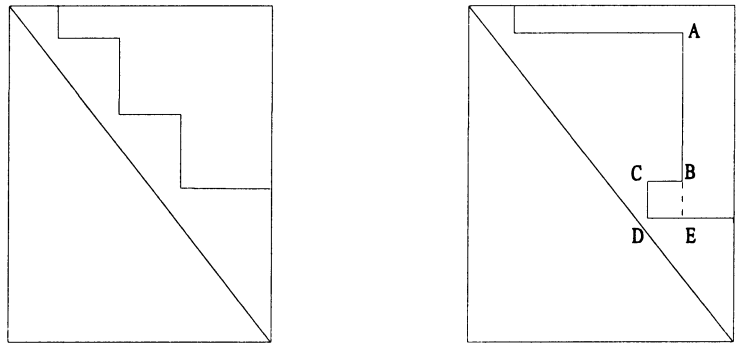
$$NTERMS = MAXA(N+1) - 1 \tag{5.11}$$

For the example shown in Figure 5.3, one has (referring to Eq. 5.7, or Eqs. 5.8 through 5.10) NTERMS = 35 - 1 = 34. It should be emphasized at this point, that in the variable band storage scheme, the row length of i^{th} row MUST also satisfy the following criteria:

$$IROWL\left(i^{th}\ row\right) \geq IROWL\left((i-1)^{th}\ row\right) - 1 \tag{5.12}$$

For example, since the row length of the 2nd row of Figure 5.3 is 4, therefore, the row length of the 3rd row of Figure 5.3 MUST BE "at least" 3. The requirement stated in Eq. 5.12 is necessary to guarantee the possibilities of a zero term may become non-zero during the factorization (see explanations given in Eqs. 5.1 and 5.2). In general, using

the variable bandwidth storage scheme, the envelope of the coefficient (stiffness) matrix may have the form shown in Figure 5.4



(a) Acceptable Envelope

(b) Unacceptable Envelope
(The vertical line CD must move to the right direction, at least to location BE!)

Figure 5.4 Acceptable and unacceptable variable bandwidth storage scheme

The parallel-vector Choleski method, described in later sections of this chapter, 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.

5.3 Basic Sequential Variable Bandwidth Choleski Method

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

$$[K] = [U]^T [U] \tag{5.13}$$

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

$$u_{ij} = 0 \quad for \quad i > j \tag{5.14}$$

$$u_{11} = \sqrt{K_{11}}\ ; \quad u_{1j} = \frac{K_{1j}}{u_{11}} \quad for \quad j \geq 1 \tag{5.15}$$

$$u_{ii} = \sqrt{K_{ii} - \sum_{k=1}^{i-1} u_{ki}^2} \quad for \quad i > 1 \tag{5.16}$$

$$u_{ij} = \frac{K_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj}}{u_{ii}} \qquad for \qquad i, j > 1 \tag{5.17}$$

When j=i, the numerator of Eq. 5.17 is identical to Eq. 5.16 without the square root operation, which simplifies coding.

For better understanding of the developments of the basic (sequential) variable bandwidth Choleski code for factorization, let us consider a full, symmetric, positive definite matrix, where the matrix size N = 9 as shown in Figure 5.5

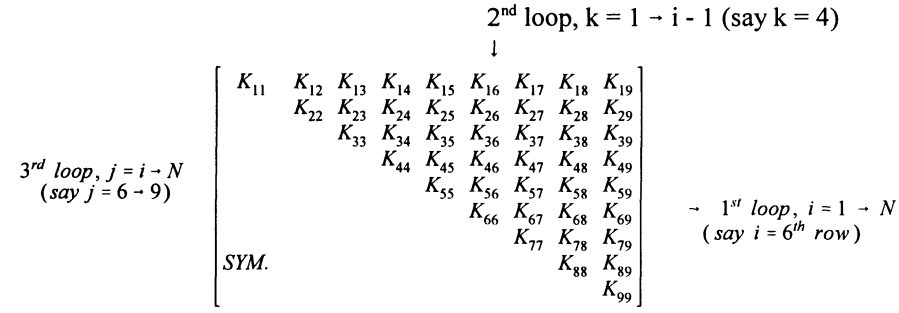


Figure 5.5 A full, symmetric stiffness matrix

In a variable bandwidth storage scheme, since the stiffness matrix has been stored in a row-wise fashion, factorization will also be done according to a row-by-row fashion (the reader should recall that factorization has been done according to a column-by-column fashion in Chapter 4!) as illustrated in the skeleton FORTRAN code shown in Figure 5.8.

According to Figure 5.5 and Eq. 5.16, the factorized diagonal term u_{66} can be computed as

$$u_{66} = \left[K_{66} - \left(u_{16}^2 + u_{26}^2 + u_{36}^2 + u_{46}^2 + u_{56}^2 \right) \right]^{1/2} \tag{5.18}$$

The above summation, within the inner parenthesis, can be considered as the inner product of

$$\begin{Bmatrix} u_{16} \\ u_{26} \\ u_{36} \\ u_{46} \\ u_{56} \end{Bmatrix} \cdot \begin{Bmatrix} u_{16} \\ u_{26} \\ u_{36} \\ u_{46} \\ u_{56} \end{Bmatrix}$$

or, with the aid of Figure 5.5, can be considered as the dot product of column #6 (excluding the diagonal term) onto itself. Similarly, the factorized off-diagonal term u_{69} can be computed from Eq. 5.17 as

$$u_{69} = \frac{K_{69} - \left(u_{16}\,u_{19} + u_{26}\,u_{29} + \ldots\ldots + u_{56}\,u_{59}\right)}{u_{66}} \qquad \textbf{(5.19)}$$

Again, the summation within the parenthesis in Eq. 5.19 can be considered as the dot product of column #6 and column #9, such as

$$\begin{Bmatrix} u_{16} \\ u_{26} \\ u_{36} \\ u_{46} \\ u_{56} \end{Bmatrix} \cdot \begin{Bmatrix} u_{19} \\ u_{29} \\ u_{39} \\ u_{49} \\ u_{59} \end{Bmatrix}$$

From Eqs. 5.18 and 5.19, one can easily see that if we wish to factorize the "entire" row #6 of Figure 5.5 (for example, to compute u_{66}, u_{67}, u_{68}, and u_{69}) then we only need to know the factorized terms in the rectangular region "right above" the 6th row. In general, the factorized information required in order to factorize any ith row of a full matrix, and variable bandwidth matrix can be shown in Figure 5.6 and Figure 5.7, respectively.

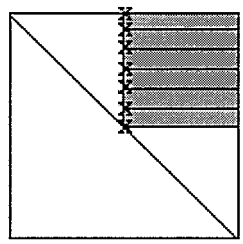


Figure 5.6 Choleski factorization of the ith row of a full matrix

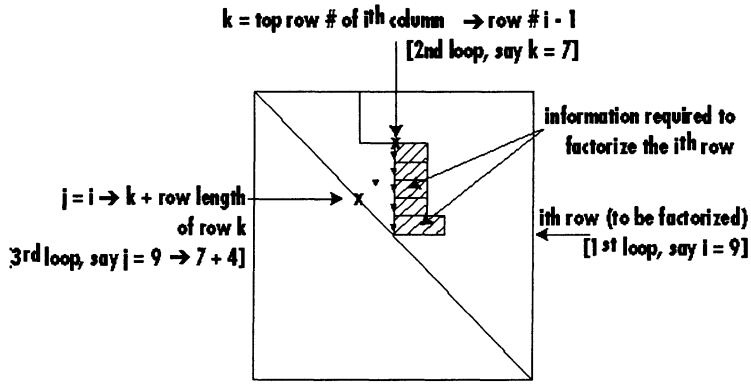


Figure 5.7 Choleski factorization for the i^{th} row of a variable bandwidth matrix.

Regardless of whether the Choleski or Gauss method is used (see Section 5.8), the basic skeleton FORTRAN sequential code for matrix factorization, based on the variable bandwidth storage scheme, is given in Table 5.1 with comments inserted to explain its connection to Eqs. 5.15 through 5.17

Table 5.1 Sequential Choleski variable-band skeleton code for matrix factorization

1			DO 1 i = row#1, row#N
2			DO 2 k = top tow# of i^{th} column, i-1
3	c		compute multiplication factor, xmult
4			xmult = U(k,i)
5	cgauss		xmult = U(k,k) * U(k,i) replaces above statement
6			DO 3 j = i, k + row length of row k
7	c		calculate the numerator of Eq. 5.17
8			U(i,j) = K(i,j) - xmult * U(k,j)
9		3	Continue
10		2	Continue
11	c		calculate final value of U(i,i) as in Eq. 5.16
12			U(i,i) = SQRT (U(i,i))
13	cgauss		remove above statement
14	c		DO loop 4 divides the numerator of Eq. 5.17 by u_{ii}
15			xinv = 1/U(i,i)
16			DO 4 j = i+1, i + row length of row i
17			U(i,j) = U(i,j) * xinv
18		4	Continue
19		1	Continue

Detailed explanations of Table 5.1 are given in the following paragraphs (the reader should refer to Figure 5.7).

Line 1: Since the Choleski variable bandwidth factorization proceeds in a row-by-row fashion, the first loop, with the index i, will scan from row #1 to the last row #N, (say i = 9th row, as shown in Figure 5.7)

Line 2: Since the ith = 9th row is being factorized, it implies that all the previous rows (row 1 through row 8) have already been completely factorized. Thus, if the matrix is assumed to be full as shown in Figure 5.6, then the information required to factorize the ith = 9th row are rows k = 1 through 8. However, if the matrix has variable bandwidths as shown in Figure 5.7, then the information required to factorize the ith = 9th row are rows k = 4 through 8, or to be more general, the index k of the second loop should be k = top row number of the ith column, i - 1, (say k = 7, as shown in Figure 5.7).

Line 3-4: Copy the known value u_{ki} into the variable xmult, where xmult is referred to as "multiplier factor".

Line 5: This "comment" statement will be discussed in greater detail in Section 5.8

Line 6: Assuming k = 7 (as explained in Line 2) the "completely" factorized 7th row can be used to "partially" factorize row number i = 9. For example (referring to Figure 5.7 and Eqs. 5.16 and 5.17).

$$u_{9,9}\ (partially\ factorized) = K_{9,9} - \left(u_{7,9}^2 + u_{8,9}^2\right) \qquad \textbf{(5.20)}$$

$$u_{9,10}(partially\ factorized) = K_{9,10} - (u_{7,9} * u_{7,10} + u_{8,9} * u_{8,10}) \qquad \textbf{(5.21)}$$

$$u_{9,11}(partially\ factorized) = K_{9,11} - (u_{7,9} * u_{7,11} + u_{8,9} * u_{8,11}) \qquad \textbf{(5.22)}$$

Thus, the index j of the 3rd loop should be j = column 9 → column 11, or to be more general:

$$j = i \rightarrow k + row\ length\ of\ row\ k \qquad \textbf{(5.23)}$$

Expression given in Eq. 5.23 for the index j is valid, since in this example, i = 9, k = 7, and row length of row 7 = 4 (hence j = 9 → 7 + 4 , or j = 9 → 11)

Lines 7-9: Calculate the numerator of Eq. 5.17

Line 10: End of the 2nd nested do-loop

Lines 11-12: Calculate the final value of the diagonal term $u_{i,i}$

Line 13: Explanation of this "comment" statement will be postponed until Section 5.8

Lines 14-15: Since division is more time consuming than multiplication, the operation $xinv = \frac{1}{u_{ii}}$ is done outside of loop 4 (with the index j)

Lines 16-18: The numerator of Eq. 5.17 is divided by $u_{i,i}$ (or multiplied by xinv) to get the complete, final answer for the factorized off-diagonal term $u_{i,j}$

Line 19: End of the 1st nested do-loop

5.4 Vectorized 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 Figure 5.5, 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. 5.17

$$\begin{aligned} u_{5j} = k_{5j} &- u_{15} * u_{1j} \\ &- u_{25} * u_{2j} \\ &- u_{35} * u_{3j} \\ &- u_{45} * u_{4j} \end{aligned} \tag{5.24}$$

where j = 5 → N (say = 9).

In Eq. 5.24, 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. 5.24 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 completed.

In a previous chapter (using the column-oriented Choleski method) once the first four columns of the factored matrix, [U], were completely updated, all terms of column 5 were updated. For example, referring to Figure 5.5, u_{25} was computed by Eq. 5.17 as

$$u_{25} = \frac{k_{25} - \left(u_{12} * u_{15}\right)}{u_{22}} \tag{5.25}$$

The term u_{25} in Eq. 5.25 was computed directly as the "final" updated value, and could not be expressed in terms of "partial" updates as is the case in Eq. 5.24. Therefore, the loop unrolling technique could not be used in this case. Instead, a vector unrolling strategy was used (see Chapter 4) to improve the vector performance in Eq. 5.17.

However, in the present chapter, the sequential Choleski skeleton in FORTRAN code in Table 5.1 can be modified to include loop-unrolling, say to level 4 as is shown in Table 5.2

Table 5.2 Vectorized Choleski factorization code (with level 4 loop unrolling)

1		DO 1 i = row#1, row#N
2		DO 2 k = top row# of i^{th} column, i-1, 4
3		DO 3 j = i, k + row length of row k
4	c	Eq. 5.24 (numerator of Eq. 5.17) follows

5			$U(i,j) = K(i,j) - U(k,i) * U(k,j) - U(k+1,i) * U(k+1,j) - U(k+2,i) * U(k+2,j) - U(k+3,i) * U(k+3,j)$
6		3	Continue
7		2	Continue
8	c		repeat loop 2 to update i^{th} row by extra k values
9	c		for DO 2 k = 1, 10, 4, extra k values are 9, 10
10			U(i,i) = SQRT (U(i,i))
11			xinv = 1/U(i,i)
12			DO 4 j = i+1, i + row length of row i
13			U(i,j) = U(i,j) * xinv
14		4	Continue
15		1	Continue

Detailed explanations of Table 5.2 are given in the following paragraphs:

Line 1: In a row-by-row fashion, the first loop, with the index i, will scan all rows from 1 to N (say i = 5^{th} row in Figure 5.5)

Line 2: Since the ith = 5^{th} row is being factorized, it implies that all the previous rows (rows 1 through 4) have already been completely factorized. Thus, if the matrix is assumed to be full (only to simplify the discussions) as shown in Figure 5.5, then the information required to factorize $i^{th} = 5^{th}$ row are rows k = 1 through 4, or to be more general, the index k of the second loop should be k = top row number of the i^{th} column, i - 1.

However, we can improve the vector speed in the next (or third) nested do-loop by packing every 4 rows of the matrix together. Thus, the index k of the second loop should be modified as

k = top row number of the i^{th} column, i - 1, 4

Line 3: Explanations have been given earlier (see line 6 of Table 5.1)

Lines 4-5: Calculations for, say Eq. 5.24, or calculations for the numerator of Eq. 5.17. The index k of the second loop will "jump" from row k = 1 to row k = 5, because of the increment 4, as explained in Line 2, and therefore, rows k = 2, 3, and 4 will be skipped. The contributions of the "already completely factorized" rows 2, 3, and 4, however, are included in line 5, in the following forms

$$- U(k+1, i) * U(k+1, j)$$
$$- U(k+2, i) * U(k+2, j)$$
$$- U(k+3, i) * U(k+3, j)$$

Lines 6-7: The third and second loops are ended, respectively.

Lines 8-9: Suppose the index i, in the first loop, has the value i = 11, then the index k, in the second loop, is supposed to have the values k = 1 through 10. However, because of the increment 4 (or loop unrolling

level 4) the index k will only reach the value of k = 8. Thus, the "equivalent" of loop 2 needs to be executed for two more times, to take care of k = 9 and 10.

Lines 10-15: These lines have the same meaning as lines 12 and 15 through 19 in Table 5.1.

Using the loop-unrolling technique (see Lines 2 and 5 of Table 5.2), 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 Table 5.2), has an increment equal to the level of unrolling, while the innermost loop (DO 3 in Table 5.2) 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 in Eq. 5.24.

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 **pvs**. **PVS** will be described in Section 5.11 after a brief synopsis of **Force** in Section 5.5.

5.5 More on Force: A Portable, Parallel FORTRAN Language[5.4]

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. The parallel extensions used in **pvs** 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 if "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. Other parallel FORTRAN software, such as PVM [4.10] and MPI [4.11] can also be employed.

5.6 Parallel-Vector Choleski Factorization

In Choleski-based methods, a symmetric, positive definite stiffness matrix, [K], can be decomposed as shown in Eqs. 5.16 and 5.17. For example, u_{57} can be computed from Eq.

5.17 as:

$$u_{57} = \frac{k_{57} - u_{15}\,u_{17} - u_{25}\,u_{27} - u_{35}\,u_{37} - u_{45}\,u_{47}}{u_{55}} \tag{5.26}$$

The calculations in Eq. 5.26 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):

$$u_{57}^{(2)} = k_{57} - u_{15}\,u_{17} - u_{25}\,u_{27} \tag{5.27}$$

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

$$u_{57}^{(3)} = u_{57}^{(2)} - u_{35}\,u_{37} \tag{5.28}$$

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 Eq. 5.26, 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 "completely" factorize 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 Figure 5.8.

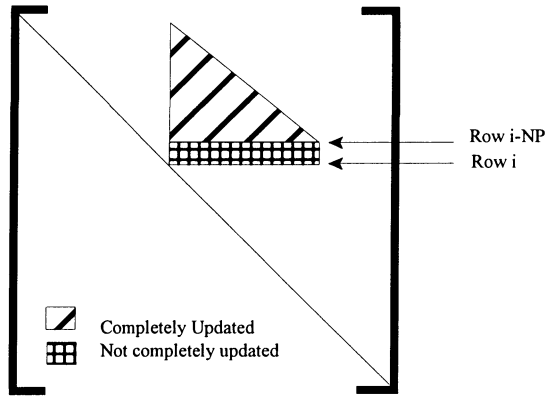


Figure 5.8 Information required to update row i

Synchronization checks are required only for the rows between (i-NP+1) and (i-1) as shown in the rectangular solid region of Figure 5.8. 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 Eqs. 5.27 and 5.28.

To simplify the discussions and to better understand the developments of Parallel-Vector Variable Bandwidth Choleski Factorization, a full and symmetrical (stiffness) matrix with 24 Degree-of-Freedom (DOF, or unknowns) and three processors (NP = 3)P_1, P_2 and P_3 are shown in Figure 5.9.

Since factorization will be done in a row-wise fashion, the three processors P_1, P_2 and P_3 will be systematically assigned to different rows, as shown in Figure 5.9.

To make the discussions more general, let's assume that a particular processor (say P_2) is currently trying to factorize a certain row (say row no. 17). Since processor P_2 is currently at row no. 17, it implies that **at least** the first fourteen rows (rows 1 through 14) have already been "completely" factorized. Since row no. 14 also belongs to processor P_2, therefore P_2 will jump to work on row no. 17 only if row no. 14 has been completely factorized, which also implies that all previous rows (rows no. 1 through 13) have also been completely factorized!

Furthermore, if P_2 is currently at row no. 17, then the other two processors (P_1 and P_3) MUST BE at either "right above" row no. 17 (if rows 15 and 16 have not been completely factorized yet!), or "right below" row no. 17 (if rows 15 and 16 have already been completely factorized)!

For example, if P_2 is at row no. 17 (i = 17), then P_1 can NOT be at row no. 1, or no. 4, or no. 7, or no. 10, or no. 13 because we have already proved that the previous (i - NP) rows (or 17 - 3 = 14 rows) must have been completely factorized. In practice, it is safer to assume that if P_2 is at the i^{th} row, then its neighboring processors (P_1, P_3, etc. . . .) will be right above it. Referring to Figure 5.9, one can see that processors P_1, P_2, and P_3 can do a significant amount of parallel computations in trying to factorize rows no. 16, no. 17, and no. 15, respectively. Processor P_1, for example, will have a lot of

work to do, since the "majority" of information required to factorize row no. 16 has already been available as shown by the rectangular region MFCD.

Thus, processor P_1 can use the available information (in the rectangular region MFCD) to "partially" factorize (or update) row no. 16. Processor P_1, however, will get the "completely" (or final) factorized row no. 16 only if its neighboring processor P_3 completely factorizes its own row (row no. 15).

While P_1 is working on row No. 16, at the same moment, processors P_2 and P_3 are simultaneously working on the "partially" factorized rows no. 17 and no. 15, respectively (using the available information in the rectangular regions, NICD and ABCD, respectively).

The black region shown in Figure 5.8 will correspond to rows 15 - 17 in Figure 5.9, where i = 17 and NP = 3.

The vectorized Choleski code in Table 5.2 has been modified for parallel processing. The resulting skeleton factorization part of the full **pvs** code is shown in Table 5.3 with parallel (**Force**) statements in boldface type.

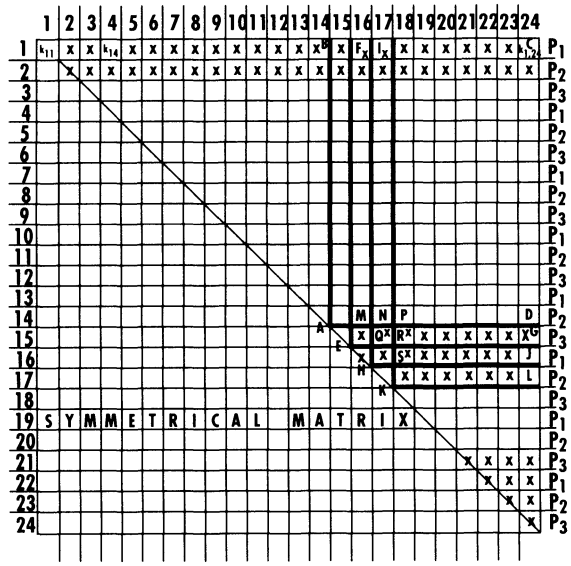


Figure 5.9 A full, symmetrical (stiffness) matrix with 24 DOF and NP = 3 processors (P_1, P_2, P_3)

Table 5.3 Parallel-vector Choleski skeleton code with level 4 loop unrolling

1		**Shared** K(21090396)
2		**Private** i,j,k,temp,xinv
3	c	{X} vector used to indicate when a row is completely factorized
4	c	[U] overwrites [K] in actual code to reduce storage
5	c	calculate U(1,1) in Eq. 5.16 on one processor

```
6                   U(1,1) = SQRT(K(1,1))
7       c
8       c           declare row#1 finished
9                   Produce X(1) = U(1,1)
10      c           start all available processors (for parallel computation)
11                  Presched DO 1 i = row#2, row#N
12      c           lock processor if row# (i-NP) is not completed
13      c           release lock when row is completely factorized
14                  IF (i-NP.GT.0) then
15                  Copy X(i-NP) into temp
16                  End if
17                  DO 2    k = top row# of the i^th column, i-NP, 4
18      c           skip DO 3 if all multipliers are zero: zero checking
19                          DO 3    j = i, k + rowlength of row k
20                  U(i,j) = K(i,j)   - U(k,i) * U(k,j)
21                                    - U(k+1,i) * U(k+1,j)
22                                    - U(k+2,i) * U(k+2,j)
                                      - U(k+3,i) * U(k+3,j)
23
24              3   continue
25              2   continue
26      c           lock the processor if row# (i-1) not finished
27      c           release the lock when row#(i-1) is finished
28                  Copy X(i-1) into temp
29                  DO 4    k=max(top row# of i^th column, i-NP+1), i-1
30                  DO 5    j=i, k + rowlength of row k
31                  U(i,j) = U(i,j) - U(k,i) * U(k,j)
32              5   continue
33              4   continue
34                  U(i,i) = SQRT(U(i,i))
35                  xinv = 1/U(i,i)
36                  DO 6    j = i+1, i + rowlength of row i
37                  U(i,j) = U(i,j) * xinv
38              6   continue
39      c           broadcast to all processors that row i is finished
40                  Produce X(i) = U(i,i)
41              1   End Presched DO
```

Explanations of Table 5.3 will be given in the following paragraphs:

Line 1: Stiffness matrix, stored in a one-dimensional real array, is declared as a "shared" variable. Thus, this array can be accessed by any processor.

Line 2: Some "private" variables are declared
Lines 3-10: These statements, with comment cards, are self-explained
Line 11: The outermost do-loop (with the index i) is executed in parallel (row-by-row fashion) by NP processors as described in Fig. 5.9.
Lines 12-13: Self-explained from comment cards
Lines 14-16: Check (synchronization) to make sure that the first (i - NP) rows have already been completely factorized? If the answer is YES, then proceed to the next statement. If the answer is NO, then processor(s) will wait in here!!
Lines 17-25: These statements play the same roles as lines 2 through 7 of Table 5.2. The major difference between Table 5.3 and Table 5.2 is that the second loop in Figure 5.9 is broken into two separated loops in Table 5.3 (see loop 2 in line 17 and loop 4 in line 29 of Table 5.3). In these statements, all processors are simultaneously trying to "partially" factorize its own row (using available information in rectangular regions ABCD, MFCD and NICD shown in Figure 5.9).
Lines 26-28: Check (synchronization) to see if any rows, within the sequential, black region shown in Figure 5.8, has been completely factorized?? If the answer if YES, then proceed to the next statement. If the answer if NO, then processor(s) will wait in here!!
Lines 29-33: These statements continue to play the same roles as the ones in lines 17 through 25. The key difference is "sequential" factorization will be done in here (see the black regions in Figure 5.8).
Lines 34-38: These statements play the same roles as the ones in lines 10 through 14 of Table 5.2.
Line 41: Self-explained from the comment card. The end of the first (parallel) nested do-loop, with the index i.

5.7 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 subroutine **pvs**, 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 (another form of loop unrolling) in the backward substitution.

It is interesting to notice that in Chapter 4, since the column-by-column storage scheme is used, vector unrolling has been used in the forward solution and loop unrolling has been used in the backward solution! The readers are asked to recall that vector unrolling is associated with "dot-product" operations and it will lead to "final" answer. On the other hand, loop unrolling is associated with "SAXPY" operations, and it will lead to "partial" (or "incomplete") answer.

5.7.1 Forward solution

To simplify the discussions, let us refer to Eq. 5.29 for the forward solution of the six simultaneous equations. In Eq. 5.29, the upper triangular matrix [U] which have already been factorized in earlier sections is also assumed to be a "full" upper triangular matrix.

In actual computer coding, the factorized matrix $[U]^T$ will be stored in a row-by-row fashion in a one-dimensional array. Thus, the numbers in each column of [U] (shown in Eq. 5.29) are stored next to each others, and therefore the "stride" (see Chapter One) between any two consecutive numbers within a column is equal to 1.

$$\begin{bmatrix} u_{11} & & & & & \\ u_{12} & u_{22} & & & & \\ u_{13} & u_{23} & u_{33} & & & \\ u_{14} & u_{24} & u_{34} & u_{44} & & \\ u_{15} & u_{25} & u_{35} & u_{45} & u_{55} & \\ u_{16} & u_{26} & u_{36} & u_{46} & u_{56} & u_{66} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \end{Bmatrix} \tag{5.29}$$

The explicit forward solution to obtain the unknown vector {y} in Eq. 5.29 can be given as

$$y_1 = \frac{F_1}{u_{11}} \tag{5.30}$$

$$y_2 = \frac{F_2 - (u_{12} y_1)}{u_{22}} \tag{5.31}$$

$$y_3 = \frac{F_3 - (u_{13} y_1 + u_{23} y_2)}{u_{33}} \tag{5.32}$$

$$y_4 = \frac{F_4 - (u_{14} y_1 + u_{24} y_2 + u_{34} y_3)}{u_{44}} \tag{5.33}$$

$$y_5 = \frac{F_5 - (u_{15} y_1 + u_{25} y_2 + u_{35} y_3 + u_{45} y_4)}{u_{55}} \tag{5.34}$$

$$y_6 = \frac{F_6 - (u_{16} y_1 + u_{26} y_2 + u_{36} y_3 + u_{46} y_4 + u_{56} y_5)}{u_{66}} \tag{5.35}$$

Observing Eqs. 5.30 through 5.35 carefully, one will recognize that it is definitely possible to get the "final" (or "complete") solution for any unknowns (say, y_5) by using the appropriate equation (say Eq. 5.34). It is NOT desirable, however, to get the "final" solution for y_5 directly (from Eq. 5.34), due to the following two reasons:

(a) The values of u_{15}, u_{25}, u_{35} and u_{45} are NOT stored consecutively (next to each others) in the one-dimensional array. Thus the "stride" between any of the above numbers is much larger than 1, hence, it will lead to poor vector performance (see Chapter One).

(b) Directly obtain the "final" answer for y_5 will require "dot-product" operations, which is known to offer much less vector speed on many shared memory computers, such as the Cray-YMP, Cray-C90, etc. . . . , as compared to the "SAXPY" operations.

For better vector performance, say, on Cray-type computers, and to fully utilize the SAXPY operations with "loop-unrolling" technique, the forward solution can be summarized by the following "key" strategies

Step 0: Let i = 1

Step 1: Solve "completely" for the "final" solution of y_i.

Thus,

$$y_1 = \frac{F_1}{u_{11}} \tag{5.36}$$

Step 2: Solve "partially" for the "incomplete" solutions of y_{i+1}, y_{i+2} . . . . y_N.

Thus,

$$y_2\,(incomplete) = F_2 - (u_{12}y_1) \tag{5.37}$$

$$y_3\,(incomplete) = F_3 - (u_{13}y_1) \tag{5.38}$$

$$y_4\,(incomplete) = F_4 - (u_{14}y_1) \tag{5.39}$$

$$y_5\,(incomplete) = F_5 - (u_{15}y_1) \tag{5.40}$$

$$y_6\,(incomplete) = F_6 - (u_{16}y_1) \tag{5.41}$$

Step 3: Let i = i + 1, and go back to step 1

Using the above strategies, the values of u_{12}, u_{13}, ..., u_{16} (required in Eqs. 5.37 through 5.41) are stored right next to each other (in the one-dimensional array U) and therefore their strides are all equal to 1. Furthermore, since the obtained forward solution vector {y} is "incomplete", SAXPY operations with loop-unrolling enhancements can be utilized!

The basic skeleton FORTRAN code to implement the strategies given by Eqs. 5.36 through 5.41 is now given in Table 5.4.

Table 5.4 Basic forward solution for row-by-row storage scheme

```
c       Initialize the unknown solution vector {y} to its corresponding
c       right-hand-side vector {F}
        DO 1  J = 1, N
    1   Y(J) = F(J)
c       Considering all unknowns
        DO 2  J = 1, N, 1
c       Obtaining the complete, "final"solution for the J^th unknown
c       see Eq. 5.36
                y(J) = y(J) / U(J,J)
c       Obtaining the partially, "incomplete" solutions for the other
c       (J + 1)^th, (J + 2)^th, . . . . , (N)^th  unknowns
c       See Eqs. 5.37 through 5.41
        DO 3  I = J + 1, N
        y(I) = y(I) - U(J,I) * y(J)
    3   continue
    2   continue
```

The operations involved inside loop 3 of Table 5.4 is called SAXPY operations, because it basically involves

$$\text{vector } y_i = \text{vector } y_i \pm \text{a constant } y_j * \text{vector } U_i$$

In the above operations, since the index j does NOT change within the loop 3, therefore y_j is considered as a constant and U_{ji} is considered as a vector.

Loop unrolling technique can be used to improve the vector performance of the algorithm shown in Table 5.4. To simplify the discussions, assuming loop-unrolling level 2 (or NUNROL = 2) is used, and the improved algorithm is self-explaining in Table 5.5 (only a skeleton of FORTRAN code is given).

Table 5.5 Loop-unrolling forward solution for row-by-row storage scheme

```
c       Initialize
        DO 1  J = 1, N
    1   Y(J) = F(J)
c       Considering all unknowns, but with increment NUNROL (say = 2)
        DO 2  J = 1, N, NUNROL
c       Obtaining the complete, "final" solution for a "few"
c       unknowns (depending on value of NUNROL)
```

$$Y(J) = \frac{y(J)}{U(J,J)}$$

$$y(J+1) = \frac{y(J+1) - U(J,J+1) * y(J)}{U(J+1,J+1)}$$

⋮

```
c       Obtaining the partially, "incomplete" solutions for other unknowns
        DO 3   I = J + NUNROL, N
        y(I) = y(I) - U(J,I) * y(J)
                    - U(J+1, I) * y(J+1)
                    -   ....
                    -   ....
    3   continue
    2   continue
```

In actual computer code implementation, loop-unrolling level 8, or level 9 (NUNROL = 8, or 9) has been used. Furthermore, the factorized matrix U has been actually stored in a one-dimensional array (using the diagonal pointer array MAXA, as explained in Eqs. 5.8 through 5.10). Finally, variable bandwidths (see array IROWL, as explained in Eq. 5.4) have also been employed to save both computer storage as well as to reduce the number of operations.

5.7.2 Backward solution

To simplify the discussions, let us refer to Eq. 5.42 for the backward solution of the same six simultaneous equations considered in Section 5.7.1.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} & u_{16} \\ & u_{22} & u_{23} & u_{24} & u_{25} & u_{26} \\ & & u_{33} & u_{34} & u_{35} & u_{36} \\ & & & u_{44} & u_{45} & u_{46} \\ & & & & u_{55} & u_{56} \\ & & & & & u_{66} \end{bmatrix} \begin{Bmatrix} Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \\ Z_6 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{Bmatrix} \qquad \textbf{(5.42)}$$

The explicit backward solution to obtain the unknown vector {Z} in Eq. 5.42 can be given as

$$Z_6 = \frac{y_6}{u_{66}} \qquad \textbf{(5.43)}$$

$$Z_5 = \frac{y_5 - (u_{56} Z_6)}{u_{55}} \qquad \textbf{(5.44)}$$

⋮

$$Z_1 = \frac{y_1 - \left(u_{12} Z_2 + u_{13} Z_3 + u_{14} Z_4 + u_{15} Z_5 + u_{16} Z_6\right)}{u_{11}} \quad \textbf{(5.45)}$$

Observing Eqs. 5.43 through 5.45 carefully, with special attention to Eq. 5.45, one will recognize that the "final", or "completed" solution for the unknowns, say Z_1, can be *(and should be)* obtained directly from Eq. 5.45. This strategy is preferred here, since the known quantities such as u_{12}, u_{13}, ..., u_{16} have already been stored in a consecutive (right next to each other) fashion (thus, the optimum stride 1 can be achieved). Since the final, complete solution for the unknowns can be obtained directly, vector-unrolling (*NOT loop-unrolling*) can also be used in conjunction with DOT-PRODUCT operations to enhance the vector speed.

The basic backward solution for row-by-row storage scheme is presented (in the form of a skeleton FORTRAN code) in Table 5.6 (assuming the factorized, upper triangular matrix is fully populated to simplify the discussions) and explanations are given in the following paragraphs.

Line 1: The first nested do-loop (with the index J) will cover from the last unknown to the first unknown (with the index increment of -1).

Line 2: The J^{th} unknown is first initialized to have the same value as the right-hand-side vector y(J)

Line 3: The second nested do-loop (with the index I) will scan from the value J + 1 to N to make sure that all terms inside the parenthesis (see the nominators of Eqs. 5.44 and 5.45) are included.

It should be noted here that most (if not all) FORTRAN compilers will automatically skip do-loop #2 if J + 1 is greater than N.

Line 4: The nominator of Eq. 5.44, or Eq. 5.45 etc. . . . . , is computed.

Line 5: Do-loop #2 is ended

Line 6: The final, complete solution for the J^{th} unknown is computed

Line 7: Do-loop #1 is ended.

The algorithm for backward solution presented in Table 5.6 can be modified to enhance its vector speed by utilizing the "vector unrolling" technique. Referring to Eq. 5.42, and assuming several rows (say every two rows, thus the level of unrolling is NUNROL = 2) are grouped together. The key strategies used by the algorithm presented in Table 5.7 can be summarized as follows (please also refer to Eq. 5.42, for a "specific" example):

Step 1: Solve completely for the last NUNROL (say, 2) unknowns Z_N, Z_{N-1} (for example Z_6 and Z_5) and let j = N - NUNROL

Step 2: Compute Z_j (incomplete) = Z_4 (incomplete) = $y_4 - (u_{45} Z_5 + u_{46} Z_6)$
Compute Z_{j-1} (incomplete) = Z_3 (incomplete) = $y_3 - (u_{35} Z_5 + u_{36} Z_6)$

Note: The effects of the triangular region of the coefficient matrix (see rows 3 and 4 of Eq. 5.42) have not yet been included in the calculations. It will, however, be incorporated in Step 3!

Step 3: Compute $Z_4\ (complete) = \dfrac{Z_4}{u_{44}}$

Compute $Z_3\ (complete) = \dfrac{Z_3 - u_{34} Z_4}{u_{33}}$

Step 4: Let j = j - NUNROL, and return back to Step 1, for the remaining unknowns

With the above four-step procedure in mind, explanations for Table 5.7 can now be given:

Line 1: Select the level of unrolling. Here, we select NUNROL = 2 (or grouping every two rows together)

Lines 2-3: Solve for the last NUNROL unknowns (see Step 1 of the above key strategies)

Line 4: This first nested do-loop (with index J) will consider all remaining unknowns (with the loop increment - NUNROL).

Lines 5-6: Initialize the next NUNROL unknowns to be equal to their corresponding right-hand-sides

Lines 7-10: This second nested do-loop (with index I) will "partially" compute the nominator (of Eqs. 5.43 through 5.45) of the next NUNROL unknowns (see Step 2 of the above key strategies)

Note: Because the increment of index J in the first loop (see line 4) has been set to "-NUNROL" (or -2), hence, there is a need to insert an EXTRA statement (see line 9) to compensate the larger increment of the index J.

Lines 11-12: Compute the final, complete unknowns (see Step 3 of the above key strategies, and also refer to the triangular regions in rows 3 and 4, and rows 1 and 2 in Eq. 5.42).

Line 13: Do-loop #1 is ended.

Table 5.6 Basic backward solution for row-by-row storage scheme

1		DO 1 J = N, 1, -1
2		Z(J) = Y(J)
3		DO 2 I = J + 1, N
4		Z(J) = Z(J) - U(J, I) * Z(I)
5	2	CONTINUE
6		$Z(J) = \frac{Z(J)}{U(J,J)}$
7	1	CONTINUE

Table 5.7 Vector unrolling backward solution for row-by-row storage scheme

1	NUNROL = 2
2	$Z(N) = \frac{Y(N)}{U(N,N)}$
3	$Z(N-1) = \frac{Y(N-1) - U(N-1,N) * Z(N)}{U(N-1,N-1)}$
4	DO 1 J = N - NUNROL, 1, -NUNROL

```
5           Z(J) = Y(J)
6           Z(J-1) = Y(J-1)
7           DO 2     I = J+1, N
8           Z(J) = Z(J) - U(J, I) * Z(I)
9           Z(J - 1) = Z(J-1) - U(J-1, I) * Z(I)
10      2   CONTINUE
11          Z(J) = Z(J) / U(J,J)
12          Z(J-1) = [Z(J-1) - U(J-1,J) * Z(J)] / U(J-1,J-1)
13      1   CONTINUE
```

5.8 Relations Amongst the Choleski, Gauss and LDL^T Factorizations

The row-oriented, sequential versions of the Choleski, Gauss and LDL^T 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:

where
$$[K]\{Z\} = \{F\} \tag{5.46}$$

$$[K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \tag{5.47}$$

and
$$\{F\} = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix} \tag{5.48}$$

The solution of Eqs. 5.46 through 5.48 is:

$$\{Z\} = \begin{Bmatrix} 1 \\ 1 \\ 1 \end{Bmatrix} \tag{5.49}$$

The basic ideas in the Choleski, Gauss and LDL^T 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, {Z}, can be solved by the familiar forward and backward substitution.

5.8.1 Choleski (U^{T}U) factorization

The stiffness matrix [K} of equation 5.47 can be converted into a Choleski upper-

triangular matrix, [U}, by appropriate "row operations":

$$[K1] = [K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \tag{5.50}$$

$$\rightarrow [K2] = \begin{bmatrix} \sqrt{2} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & -1 & 1 \end{bmatrix} \rightarrow [K3] = \begin{bmatrix} \sqrt{2} & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} \\ 0 & -1 & 1 \end{bmatrix} \tag{5.51}$$

$$\rightarrow [K4] = \begin{bmatrix} \sqrt{2} & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \rightarrow [K5] = \begin{bmatrix} \sqrt{2} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} \\ 0 & 0 & \frac{1}{\sqrt{3}} \end{bmatrix} \tag{5.52}$$

where

$$Row\ 1\ of\ [K2] = \frac{Row\ 1\ of\ [K]}{\sqrt{K1(1,1)}}$$

$$Row\ 2\ of\ [K2] = \frac{Row\ 1\ of\ [K2]}{\sqrt{2}} + Row\ 2\ of\ [K1]$$

$$Row\ 2\ of\ [K3] = \frac{Row\ 2\ of\ [K2]}{\sqrt{K2(2,2)}}$$

$$Row\ 3\ of\ [K4] = Row\ 2\ of\ [K3] * \sqrt{\frac{2}{3}} + Row\ 3\ of\ [K3]$$

$$Row\ 3\ of\ [K5] = \frac{Row\ 3\ of\ [K4]}{\sqrt{K4(3,3)}}$$

The multiplier constants, m_{ij}, used in the forward substitution (or updating the right-hand side vector of Eq. 5.46) are the same as terms in the factorized upper-triangular matrix such that:

$$m_{12} = u_{12} = -\frac{1}{\sqrt{2}},\ m_{13} = u_{13} = 0,\ m_{23} = u_{23} = -\frac{\sqrt{2}}{\sqrt{3}} \tag{5.53}$$

Another way to view this Choleski factorization process is to express Eq. 5.47 as

$$[K] = [U]^T[U] \tag{5.54}$$

For the data shown in Eq. 5.47, and the direct applications of Eqs. 5.16 and 5.17, Eq. 5.54 can be expressed as

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 & 0 \\ -\frac{1}{\sqrt{2}} & \frac{\sqrt{3}}{\sqrt{2}} & 0 \\ 0 & \frac{-\sqrt{2}}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{bmatrix} * \begin{bmatrix} \sqrt{2} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} \\ 0 & 0 & \frac{1}{\sqrt{3}} \end{bmatrix} \tag{5.55}$$

The second matrix on the right-hand side of Eq. 5.55 can be identified as the matrix [K5] presented in Eq. 5.52.

The forward solution (or updating the right-hand side vector F of Eq. 5.46) can be obtained by solving Eq. 4.4, from Chapter Four:

$$[U]^T\{y\} = \{F\}$$

for the vector {y} (or, the "updating" right-hand-side vector {F}). The multipliers shown in Eq. 5.53 turn our to be the same as the off-diagonal terms of the matrix $[U]^T$ in Eq. 5.55.

5.8.2 Gauss (with diagonal terms $L_{ii} = 1$) LU factorization

As in the Choleski method just described, the stiffness matrix, [K], of Eq. 5.47 can also be converted into a Gauss upper-triangular matrix by appropriate row operations.

$$[K1] = [K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \tag{5.56}$$

$$\Rightarrow [K2] = \begin{bmatrix} 2 & -1 & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & -1 & 1 \end{bmatrix} \Rightarrow [K3] = \begin{bmatrix} 2 & -1 & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \tag{5.57}$$

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

$$m_{12} = \frac{u_{12}}{u_{11}} = -\frac{1}{2} \tag{5.58}$$

$$m_{13} = \frac{u_{13}}{u_{11}} = \frac{0}{2} = 0 \tag{5.59}$$

$$m_{23} = \frac{u_{23}}{u_{22}} = \frac{-1}{\frac{3}{2}} = -\frac{2}{3} \tag{5.60}$$

Another way to view this Gauss factorization process is to express Eq. 5.56 as

$$[K] = [L][U] \tag{5.61}$$

or

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{5.62}$$

The three unknowns in the lower triangular matrix [L], and the six unknowns in the upper triangular matrix [U] of Eq. 5.62 can be obtained simply by enforcing the nine equality conditions (on both sides of Eq. 5.62). Thus, Eq. 5.62 can be expressed as:

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & -\frac{2}{3} & 1 \end{bmatrix} * \begin{bmatrix} 2 & -1 & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \tag{5.63}$$

The second matrix on the right-hand-side of Eq. 5.63 can be identified as the matrix [K3] presented in Eq. 5.57.

The forward solution can be obtained by solving

$$[L]\{y\} = \{F\}$$

for the vector {y} (or, the "updating" right-hand-side vector {F}). The multipliers shown in Eqs. 5.58 through 5.60 turn out to be the same as the off-diagonal terms of the lower triangular matrix [L] in Eq. 5.63.

5.8.3 Gauss (LU) factorization with diagonal terms $U_{ii} = 1$

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

$$[K1] = [K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \tag{5.64}$$

$$\Rightarrow [K2] = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & \frac{3}{2} & -1 \\ 0 & -1 & 1 \end{bmatrix} \Rightarrow [K3] = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & -1 & 1 \end{bmatrix} \tag{5.65}$$

$$\Rightarrow [K4] = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \Rightarrow [K5] = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & 0 & 1 \end{bmatrix} \tag{5.66}$$

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,

$$u_{11} = 2 \; ; \; u_{22} = \frac{3}{2} \; ; \; and \; u_{33} = \frac{1}{3} . \tag{5.67}$$

The multiplier m_{ij} is obtained from the factored matrix, [U], as:

$$m_{12} = u_{12} * u_{11} = -\frac{1}{2} * 2 = -1 \tag{5.68}$$

$$m_{13} = u_{13} * u_{11} = 0 * 2 = 0 \tag{5.69}$$

$$m_{23} = u_{23} * u_{22} = -\frac{2}{3} * \frac{3}{2} = -1 \tag{5.70}$$

Another way to view this Gauss factorization process is to express Eq. 5.64 as

$$[K] = [L][U]$$

or

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} * \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix} \tag{5.71}$$

The six unknowns in lower triangular matrix [L], and the three unknowns in the upper triangular matrix [U] of Eq. 5.71 can be obtained simply by enforcing the nine equality conditions on both sides of Eq. 5.71.

Thus, Eq. 5.71 can be expressed as

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ -1 & \frac{3}{2} & 0 \\ 0 & -1 & \frac{1}{3} \end{bmatrix} * \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & 0 & 1 \end{bmatrix} \tag{5.72}$$

The second matrix on the right-hand-side of Eq. 5.72 can be readily identified

as the matrix [K5] in Eq. 5.66.

The forward solution can be obtained by solving

$$[L]\{y\} = \{F\}$$

for the vector {y}. The diagonal terms (before scaling to the value 1) shown in Eq. 5.67 appear on the diagonals of the matrix [L] in Eq. 5.72. The multipliers shown in Eqs. 5.68 through 5.70 turn out to be the same as the off-diagonal terms of the lower triangular matrix [L] in Eq. 5.72.

5.8.4 LDL^T factorization with diagonal terms $L_{ii} = 1$

It is also possible to express Eq. 5.47 into the following form

$$[K] = [L][D][L]^T \tag{5.73}$$

In Eq. 5.73, [L] is a lower triangular matrix with unit values for its diagonals, and [D] is a diagonal matrix. Using the numerical data shown in Eq. 5.47, one can express Eq. 5.73 as:

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} D_1 & 0 & 0 \\ 0 & D_2 & 0 \\ 0 & 0 & D_3 \end{bmatrix} \begin{bmatrix} 1 & L_{21} & L_{31} \\ 0 & 1 & L_{32} \\ 0 & 0 & 1 \end{bmatrix} \tag{5.74}$$

Solving the above system of six simultaneous equations for the unknowns L_{21}, L_{31}, L_{32}, D_1, D_2 and D_3 (by expressing the six equality conditions for the upper triangular portions of the matrix on both sides of Eq. 5.74) one obtains

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & -\frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & \frac{3}{2} & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & 0 & 1 \end{bmatrix} \tag{5.75}$$

The product [P] of the first two matrices on the right-hand-side of Eq. 5.75 turns out to be

$$[P] = \begin{bmatrix} 2 & 0 & 0 \\ -1 & \frac{3}{2} & 0 \\ 0 & -1 & \frac{1}{3} \end{bmatrix} \tag{5.76}$$

It is interesting to see that matrix [P] in Eq. 5.76 is the same as the first matrix in Eq. 5.72 of Section 5.8.3. Furthermore, the last matrix of Eq. 5.75 is identical to the last matrix in Eq. 5.72. Thus, LDL^T strategy is essentially equivalent to $\bar{L}\ \bar{U}$ strategy (with $\bar{U}_{ii} = 1$, as discussed in Section 5.8.3), where $\bar{L} \equiv L * D$ and $\bar{U} \equiv L^T$.

The diagonal matrix D can be obtained as diagonal terms of the factored matrix (of LDL^T procedure, shown in Table 5.8). The multipliers (for the LDL^T algorithm) can

also be obtained directly from LDL^T matrix (= off-diagonal terms of the factored matrix U, in Table 5.8).

To simplify the discussions, assuming the matrix [K] (shown in Eq. 5.47) is fully populated, the "skeleton" LDL^T code is given in Table 5.8.

Table 5.8 LDL^T factorization

```
      DO 11 I = 1, N
      DO 22 K = 1, I-1
            xmult = u( ) / D( I ) = u(K,I) / u(K,K)
      DO 33 J = I,N (or I + Irowlength)
c     Irowlength = row length (or bandwidth) of the I^th row
            u(I,J) = u(I,J) - xmult * u(K,J)
      33    CONTINUE
            u(K, I) = xmult
      22    CONTINUE
      11    CONTINUE
```

Implementation of the algorithm given in Table 5.8 with the numerical data shown in Eq. 5.47 will lead to the following results

For I = 1, hence (temporarily) no change in 1st row
For I = 2, hence K = 1 → 1

$$xmult = \frac{u(1,2)}{u_{11}} = \frac{-1}{2}$$

Loop 33:

$$u_{2,2} = u_{2,2} - (xmult)(u_{1,2}) = 2 - \left(\frac{-1}{2}\right)(-1) = \frac{3}{2}$$

$$u_{2,3} = u_{2,3} - (xmult)(u_{1,3}) = -1 - \left(-\frac{1}{2}\right)(0) = -1$$

u(1, 2) = xmult = $\frac{-1}{2}$

For I = 3, hence K = 1 → 2
Now K = 1

$$xmult = \frac{u(1,3)}{u_{1,1}} = \frac{0}{2} = 0$$

Loop 33:

$$u_{3,3} = u_{3,3} - (xmult = 0) * (u_{1,3} = 0) = 1$$

$$u(1,3) = 0$$

Now K = 2

$$xmult = \frac{u(2,3)}{u_{2,2}} = \frac{-1}{\left(\frac{3}{2}\right)} = \frac{-2}{3}$$

Loop 33:

$$\left\{ u_{3,3} = u_{3,3} - \left(xmult = \frac{-2}{3} \right) \left(u_{2,3} = -1 \right) = \frac{1}{3}$$

$$u(2,3) = xmult = \frac{-2}{3}$$

Hence:

$$U = \begin{bmatrix} 2 & -\frac{1}{2} & 0 \\ & \frac{3}{2} & -\frac{2}{3} \\ & & \frac{1}{3} \end{bmatrix}$$

From the above results, one can identify:

$$[D] = \begin{bmatrix} 2 & 0 & 0 \\ 0 & \frac{3}{2} & 0 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \text{ and } [L]^T = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{2}{3} \\ 0 & 0 & 1 \end{bmatrix}$$

5.8.5 Similarities of Choleski and Gauss methods

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. 5.46) can always be recovered conveniently from the factored, upper triangular matrix, [U].
3) The methods can be modified 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 triangular 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.

To use the Gauss solution method (i.e., for non-positive-definite systems of equations), only two FORTRAN statements, labeled cgauss in Table 5.1 need to be changed.

The multiplier constants, xmult, and the column height information are utilized in the DO 2 loop in Table 5.1 to avoid operations with zeros outside the column height (or skyline). The parameter, k, of the DO 2 loop is illustrated in Table 5.1. For i= 6 (in DO 1 of Table 5.1) the index k (in DO 2) has the values from 2 to 5 as shown in Table 5.1.

Although [K] and [U] are two-dimensional arrays in Table 5.1, in the actual Choleski factorization code, both are stored in a one-dimensional array.

5.9 Factorization Based Upon "Look Backward" Versus "Look Forward" Strategies

The parallel-vector factorization (for row-by-row storage scheme) which has been discussed in Section 5.5 is based on the "look backward" strategy. This strategy is due to the fact that if we want to factorize the i^{th} (say the 17^{th}) row of a matrix (shown in Figure 5.9) then we need to "look backward" to utilize all previously factorized rows (say rows 1 through 14, according to the example shown in Figure 5.9).

The discussions on the "look forward" strategies can be started with a given symmetric, positive definite (stiffness) matrix [K]. We now are looking for a lower triangular matrix $[L_1]$, and a symmetric positive definite matrix $[K_1]$, such that

$$[K] = [L_1][K_1][L_1]^T \tag{5.77}$$

Similarly, we then are looking for another lower triangular matrix $[L_2]$, and a symmetric positive definite matrix $[K_2]$, such that

$$[K_1] = [L_2][K_2][L_2]^T \tag{5.78}$$

The above process will be repeated

$$[K_2] = [L_3][K_3][L_3]^T \tag{5.79}$$

until

$$[K_{N-1}] = [L_N][K_N][L_N]^T \tag{5.80}$$

If the matrix $[K_N]$, shown in Eq. 5.80, converges to an identity matrix $[I_N]$, then it can be easily shown that

$$[K] = [L] * [L]^T \tag{5.81}$$

where [L] is a lower triangular matrix with the same dimension as [K]. The proof of Eq. 5.81 can be easily shown, simply by substituting Eqs. 5.78 through 5.80 into Eq. 5.77, to obtain

$$[K] = [L_1][L_2]\ldots.[L_N][K_N][L_N]^T[L_{N-1}]\ldots.[L_1]^T \quad \textbf{(5.82)}$$

Since $[K_N]$ is an identity matrix, Eq. 5.82 can be expressed as

$$[K] = [L_1\ L_2\ .\ .\ .\ .\ L_N] * [L_1\ L_2\ .\ .\ .\ .\ L_N]^T \quad \textbf{(5.83)}$$

Equation 5.83 has the same form as Eq. 5.81, where

$$[L] = [L_1][L_2]\ .\ .\ .\ .[L_N] \quad \textbf{(5.84)}$$

Starting from the given matrix $[K]_{NxN} \equiv [K_0] \equiv [H_0]$, we can make the following partitions

$$[K] \equiv [K_0] = [H_0] = \begin{bmatrix} d_1 & v_1^T \\ v_1 & \bar{H}_1 \end{bmatrix} \quad \textbf{(5.85)}$$

The dimensions for various sub-matrices in Eq. 5.85 are given as

d_1 is a 1 x 1 submatrix (or scalar)
v_1 is a (N-1) x (1) submatrix (or a column vector)
v_1^T is a (1) x (N-1) submatrix (or a row vector)
$\bar{H}_1$ is a (N-1) x (N-1) submatrix

It will be proved shortly that the original, given matrix [K] can be expressed in the form of Eq. 5.77 if matrices $[L_1]$ and $[K_1]$ are defined according to the following formulas:

$$[L_1] = \begin{bmatrix} \sqrt{d_1} & 0 \\ \dfrac{v_1}{\sqrt{d_1}} & I_{N-1} \end{bmatrix} \quad \textbf{(5.86)}$$

where I_{N-1} is a square, identity matrix with the dimension N-1

$$[K_1] \equiv \begin{bmatrix} 1 & 0 \\ 0 & \bar{H}_1 - \dfrac{v_1 v_1^T}{d_1} \end{bmatrix} \quad \textbf{(5.87)}$$

With the above definition for matrices $[L_1]$ and $[K_1]$, the readers can verify (as a short exercise) easily the following equality

$$[K] \equiv \begin{bmatrix} d_1 & v_1^T \\ v_1 & \bar{H}_1 \end{bmatrix} = [L_1][K_1][L_1]^T \quad \textbf{(5.88)}$$

The repeated applications of Eqs. 5.86 and 5.87 in Eqs. 5.78 through 5.80 will

eventually lead to Eq. 5.82, and therefore Eq. 5.84 can be obtained. The following simple numerical example will clarify all the detailed steps discussed in Eqs. 5.77 through 5.88.

Step 1: Given

$$[K] = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \tag{5.89}$$

Hence:

$$v_1^T = \{-1, 0, 0\} \tag{5.90}$$

$$\bar{H}_1 = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \tag{5.91}$$

$$d_1 = 2 \tag{5.92}$$

Step 2: Compute (using Eqs. 5.86 and 5.87)

$$[L_1] = \begin{bmatrix} \sqrt{2} & 0 & 0 & 0 \\ \frac{-1}{\sqrt{2}} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.93}$$

$$[K_1] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1.5 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \tag{5.94}$$

Hence:

$$v_2^T = \{-1, 0\} \tag{5.95}$$

$$\bar{H}_2 = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \tag{5.96}$$

$$d_2 = 1.5 \tag{5.97}$$

Step 3: Compute (using Eqs. 5.86 and 5.87, again)

$$[L_2] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \sqrt{1.5} & 0 & 0 \\ 0 & \frac{-1}{\sqrt{1.5}} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.98}$$

$$\left[K_2\right] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \tag{5.99}$$

Hence:

$$v_3^T = \{-1\} \tag{5.100}$$

$$\bar{H}_3 = [1] \tag{5.101}$$

$$d_3 = \frac{4}{3} \tag{5.102}$$

Step 4: Compute (using Eqs. 5.86 and 5.87, again)

$$\left[L_3\right] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{2}{\sqrt{3}} & 0 \\ 0 & 0 & \frac{-\sqrt{3}}{2} & 1 \end{bmatrix} \tag{5.103}$$

Hence:

$$\left[K_3\right] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{4} \end{bmatrix} \tag{5.104}$$

$$v_4^T = \{0\} \tag{5.105}$$

$$\bar{H}_4 = [0] \tag{5.106}$$

$$d_4 = \frac{1}{4} \tag{5.107}$$

Step 5: Compute

$$[L_4] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{bmatrix} \tag{5.108}$$

$$[K_4] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.109}$$

The computation loop is ended in here.

Using Eq. 5.84, one has

$$[L] = [L_1][L_2][L_3][L_4] \tag{5.110}$$

Substituting Eqs. 5.93, 5.98, 5.103, and 5.108 into Eq. 5.110, one obtains

$$[L] = \begin{bmatrix} \sqrt{2} & 0 & 0 & 0 \\ \frac{-1}{\sqrt{2}} & \sqrt{1.5} & 0 & 0 \\ 0 & \frac{-1}{\sqrt{1.5}} & \frac{2}{\sqrt{3}} & 0 \\ 0 & 0 & \frac{-\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix} \tag{5.111}$$

Again, it can be left as a small exercise for the readers to verify that Eq. 5.81 will be satisfied, when the numerical data for Eqs. 5.111, and 5.89 are used.

Observing Eq. 5.111 carefully, one will notice that each column of the lower triangular matrix [L] can be obtained successively from the 1st, 2nd, 3rd and 4th columns of the matrices $[L_1]$, $[L_2]$, $[L_3]$, and $[L_4]$ (from steps 2 through 5) respectively.

This entire process (to obtain the final [L] matrix in Eq. 5.111) can be best summarized in Figs. 5.10 through 5.13

$$[L] = \begin{bmatrix} V & x & x & x \\ V & * & * & * \\ V & * & * & * \\ V & * & * & * \end{bmatrix}$$

Figure 5.10 1st column done initially (see V terms), "look forward" updating the 3x3 submatrix (see * terms)

$$[L] = \begin{bmatrix} x & x & x & x \\ x & V & x & x \\ x & V & * & * \\ x & V & * & * \end{bmatrix}$$

Figure 5.11 2nd column done (see V terms), "look forward" updating the 2x2 submatrix (see*terms)

$$[L] = \begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & V & x \\ x & x & V & * \end{bmatrix}$$

Figure 5.12 3rd column done (see V terms), "look forward" updating the 1x1 submatrix (see * term)

$$[L] = \begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & V \end{bmatrix}$$

Figure 5.13 4th (or last) column done (see V term)

The key differences between "Look Backward" and "Look Forward" parallel strategies for Choleski factorization are also summarized in Figures 5.14 and 5.15.

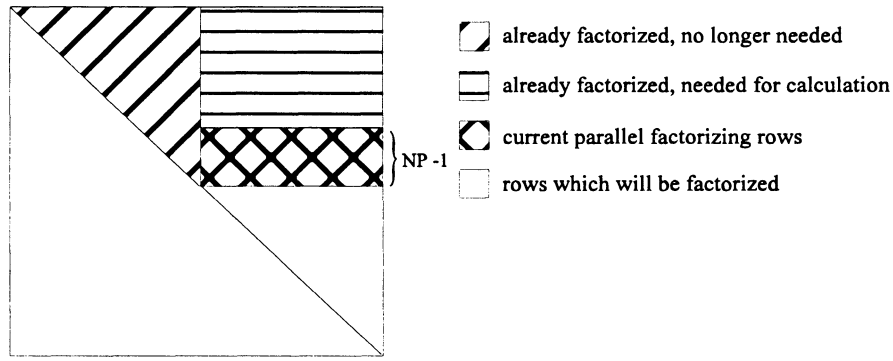


Figure 5.14 Look backward parallel factorization

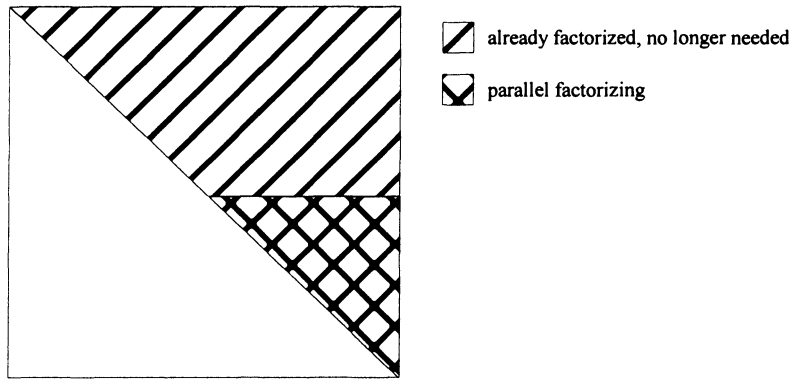


Figure 5.15 Look forward (outer product form) parallel factorization

5.10 Evaluation of Methods for Structural Analyses

To test the effectiveness of the parallel-vector solver, described in Sections 5.6 and 5.7, 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] {Z} = {F}), the residual vector,

$$\{R\} = [K]\{Z\} - \{F\} \tag{5.112}$$

is calculated, and the absolute error norm,

$$e_a = \sqrt{\{R\}^T \{R\}} \tag{5.113}$$

and the strain energy error norm,

$$e_s = \{Z\}^T [K] \{Z\} - \{Z\}^T \{F\} \tag{5.114}$$

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, result in small values for {R}, e_a and e_s in Eqs. 5.112 through 5.114.

The solution times using **pvs** code (see Section 5.11) for the SRB application were also obtained on Cray 2 supercomputers at NASA Ames and NASA Langley and compared with solution time for the skyline algorithm in a previously published paper[5.5].

In the following applications, code is inserted in **pvs** 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 **pvs** were run in a dedicated computer environment.

5.10.1 High speed research aircraft

To evaluate the performance of the and 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[5.6], shown in the upper right of Figure 5.16

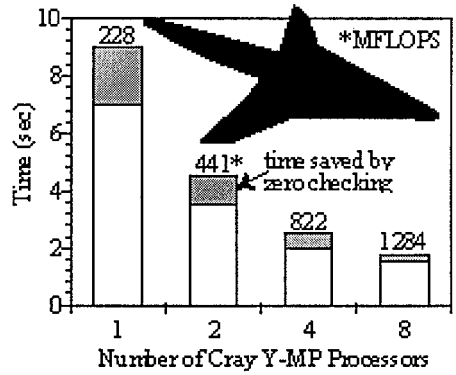


Figure 5.16 Effect of more processors on analysis time
(high speed research aircrafts)

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[5.7] finite element code. 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 initial number of non-zero in the stiffness matrix is 499,505. The number of non-zeros after factorization, including fills-in, increases to 5,579,839. The half-model is constrained along the plane of the fuselage centerline and subjected to upward loads at the wing tip 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. 5.113 and 5.114. 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 independent 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 **pvs**, 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 5.9.

Table 5.9 Matrix decomposition time (MFLOPS) for aircraft on Cray Y-MP

Processors	Sec (MFLOPS)	Sec (MFLOPS)
	(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 line 18 of Table 5.3) 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 Figure 5.16. The number above the bars (in MFLOPS) in Figure 5.16 show the increased computation rate as the number of processors increases.

5.10.2 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 have been calculated.

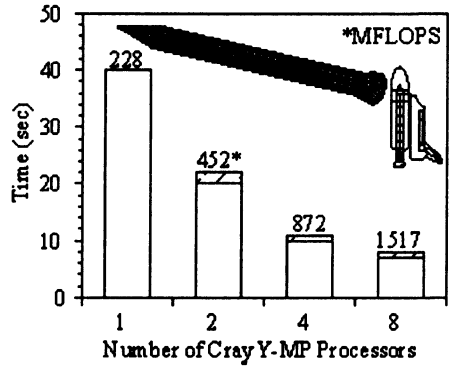


Figure 5.17 Effect of more processors on analysis time (space shuttle SRB)

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[5.8]. 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. The initial number of non-zeros in the stiffness matrix is 1,310,973. The number of non-zeros after factorization, including fills-in, increases to 21,090,396. A detailed description and analysis of this problem is given in references [5.8 and 5.9].

The calculated absolute and strain energy residuals for the static displacements are 0.00014 and 0.0017, respectively, from Eqs. 5.113 and 5.114. 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 **pvs**, 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 5.10. 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 1444MFLOPS. The CPU times are approximately 10 percent less than the elapsed times quoted on one processor.

Table 5.10 Matrix decomposition time (MFLOPS) (shuttle SRB on Cray Y-MP)

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 Figure 5.17. The corresponding computation rate for this matrix factorization, using eight processors on the Cray Y-MP is 1,517 MFLOPS. The previous recorded 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[5.10-5.11].

For structural analysis problems with a larger average column height, and bandwidth than the aircraft or SRB discussed, one can expect **pvs** 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 **pvs** for a structural matrix with an average bandwidth of 699 on the eight-processor Cray Y-MP[5.12-5.13].

The decomposition time for the Shuttle SRB matrix using **pvs**, is compared to the skyline algorithm[5.5] in Figure 5.18 (discussed in Chapter 4) for 1, 2 and 4 Cray 2 processors.

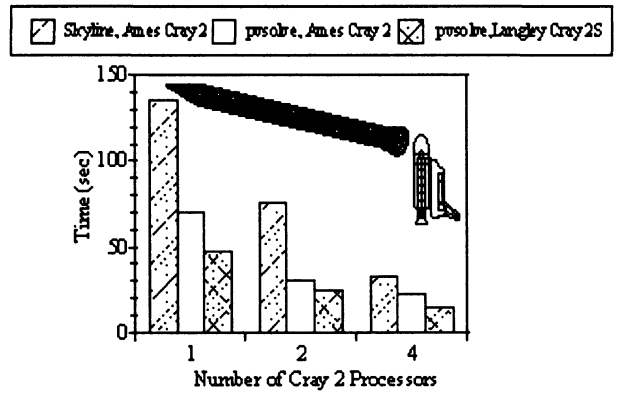


Figure 5.18 SRB decomposition time comparison

A reduction in decomposition time by a factor of 2 is shown for **pvs** in the figure for the Cray 2 at NASA Ames. An additional reduction in decomposition time of approximately 50 percent is shown for **pvs** 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 **pvs** using eight processors on the Cray Y-MP (six seconds in Figure 5.17) 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 Figure 5.18.

The above results have been obtained using loop unrolling to level 9 (see Section 5.4). 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 **pvs** 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.

5.11 Descriptions of Parallel-Vector Subroutine PVS

The input data and arguments required to call the equation solver, **pvs**, together with a simple 21-equation example are given in this section. The user should have a limited knowledge of parallel computing and the parallel FORTRAN language **Force**[5.4] **Pvs** contains a **Force** subroutine, PVS, which may be called by general purpose codes. It should be emphasized that parallel FORTRAN language FORCE is used here, however, others such as PVM, MPI etc... could also be used with minimum changes to the code. The information required by PVS to solve systems of simultaneous equations (i.e., [K] {Z} = {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, dimensional neq, containing the load vector, {F}. Upon return from subroutine PVS, b contains the displacement solution, {Z}.

maxa = an integer vector, dimensioned neq, containing the location of the diagonal terms of [K] in vector {a}, notice that maxa (neq) is equal to the total number of coefficients (including fill-ins, after factorization) of [K].

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 be solved (= degree of freedom).

nterms = the dimension of the vector, {a}, [= maxa(neq)], refer also to Eqs. 5.8 through 5.11, in Section 5.2.

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 factorization and forward/backward substitution, 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 Figure 5.19 are given in Table 5.11.

The line in Figure 5.19 represents the skyline defined by the column heights which extend up to the last nonzero in each column. The "extra zeros" outside the skyline are required to achieve level 9 loop unrolling. The DO 2 loop in Table 5.2 (see line 2) illustrates this for level 4 loop unrolling. The vector {a}, {b}, {maxa}, {icolh}, and

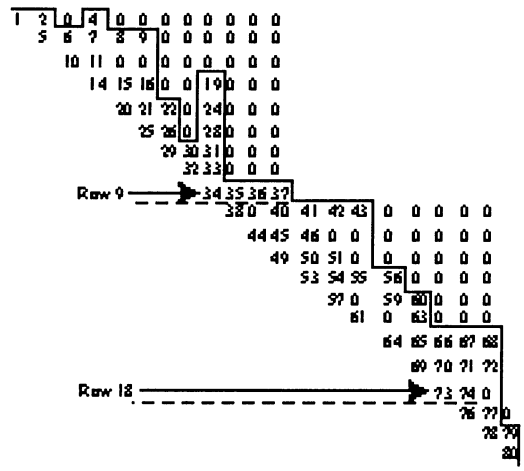


Figure 5.19 Example [K] matrix with 21 equations

{irowl} which are read by **pvs** are given in Table 5.11 (where neq=21 and nterms = 141)

Table 5.11 PVS input to solve [K] * {Z} = {F} (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	76.				
137	77				
138	0				
139	78.				
140	79				
141	80				

The Force subroutine PVS should be called twice; first to factor the matrix (iif = 2), and second to perform the forward/backward solution for displacements with error checking (iif = 5).

5.12 Parallel-Vector Equation Solver Subroutine PVS

For the complete listing of the FORTRAN source codes, instructions in how to incorporate this equation solver package into any existing application software (on any specific computer platform), and/or the complete consulting service in conjunction with this equation solver etc... the readers should contact:

Prof. Duc T. Nguyen
Director, Multidisciplinary Parallel-Vector Computation Center
Civil and Environmental Engineering Department
Old Dominion University
Room 135, Kaufman Building
Norfolk, VA 23529 (USA)
Tel = (757) 683-3761, Fax = (757) 683-5354
Email = dnguyen@odu.edu

5.13 Summary

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 zeros inside the stiffness 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 **pvs** 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.

5.14 Exercises

5.1 Verify Eq. 5.88

5.2 Given the following coefficient (stiffness) matrix [K]:

$$[K] = \begin{bmatrix} 3838. & 0. & 4. & 41. & 42. & 43. & 0. & 0. \\ & 4444. & 45. & 46. & 0. & 0. & 0. & 0. \\ & & 4747. & 48. & 49. & 0. & 0. & 0. \\ & & & 5050. & 51. & 52. & 53. & 0. \\ & & & & 5454. & 0. & 56. & 57. \\ SYM. & & & & & 5858. & 0. & 59. \\ & & & & & & 6060. & 61. \\ & & & & & & & 6565. \end{bmatrix}$$

Assuming "loop-unrolling" level 3 is used

(a) Construct the one-dimensional arrays a(-), maxa(-), icolh(-), and irowl(-), similar to the ones presented in Table 5.11?

(b) How many "real" words of computer memory are required by the array a(-) for this example?

5.3 Given the following coefficient (stiffness) matrix [K], and load (or right-hand-side) vector {F}:

$$[K] = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 8 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}$$

$$\{F\} = \begin{Bmatrix} 1 \\ 6 \\ 0 \\ 3 \end{Bmatrix}$$

(a) Find the factorized matrix [U] using a row-by-row Choleski algorithm?

(b) Find the "forward solution" of the system [K] {Z} = {F}?

(c) Find the "backward solution" of the system [K] {Z} = {F}?

5.4 Repeat problem 5.3, but using the row-by-row LDL^T algorithm?

5.5 For the data shown in problem 5.3, assuming the Choleski factorized matrix [U] has already been obtained (in problem 5.3a):

(a) Using (and modifying, if necessary) Table 5.4, write a computer FORTRAN program to obtain the "forward solution"?

(b) Using (and modifying, if necessary) Table 5.6, write a computer FORTRAN program to obtain the "backward solution"

5.6 For the data shown in problem 5.3, assuming the Choleski factorized matrix [U] has already been obtained (in problem 5.3a):

(a) Using (and modifying, if necessary) Table 5.5, write a computer FORTRAN program to obtain the "forward solution" with "loop-unrolling" level 2?

(b) Using (and modifying, if necessary) Table 5.7, write a computer FORTRAN program to obtain the "backward solution" with "vector-unrolling" level 2

5.7 Modifying the FORTRAN program(s) in problem 5.6, so that one-dimensional

(instead of two-dimensional) array can be used to store the factorized matrix [U].

Hints: You also need to use integer array MAXA(-) for diagonal locations, ICOLH(-) for column heights, IROWL(-) for variable row lengths, etc....

5.8 For the data shown in Problem 5.3, using (and modifying, if necessary) the LDL^T algorithm presented in Table 5.8, find the factorized matrix [U]?

5.9 Assuming the factorized matrix [U] has already been obtained (say, either by hand calculator, or by a computer program as have been done in Problem 5.8) from LDL^T algorithm (see Table 5.8)

(a) Write a FORTRAN subroutine to perform "forward solution"

(b) Write a FORTRAN subroutine to perform "backward solution"

5.10 For the coefficient (stiffness) matrix [K] data shown in Problem 4.1, assuming 3 processor (P_1, P_2, and P_3) are used in this example, and according to the following information

Processor Number	"Rows" of matrix [K] Which Belong to a Processor
P_1	1, 2, 3, 10, 11, 12, 19, 20, 21
P_2	4, 5, 6, 13, 14, 15
P_3	7, 8, 9, 16, 17, 18

Without any actual computation, and assuming the first eight rows of the factorized matrix [U] have already been completely factorized, identify which terms (if any) U_{ij} of the matrix [U] can be factorized

by processor P_1?

by processor P_2?

by processor P_3?

5.11 For the parallel-vector Choleski factorization algorithm presented in Section 5.6, what do you think will happen (say, in terms of parallel speed) when the number of processors (=NP) becomes very large?

(Hint: see Figure 5.8)

5.15 References

5.1 Agarwal, T.K., O.O. Storaasli and D.T. Nguyen, "A Parallel-Vector Algorithm for Rapid Structural Analysis on High-Performance Computers", *Proceedings of the AIAA/ASME/ ASCE/AHS 31st SDM Conference*, Long Beach, CA, AIAA paper No. 90-1149, April 2-4, 1990.

5.2 Bathe, K.J., *Finite Element Procedures*, Prentice-Hall, Inc., New York, (1996).

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

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

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

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

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

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

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

5.10 Ashcraft, C.C., R.G. Grimes, J.G. Lewis, B.W. Peyton, and H.D. Simon, "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.

5.11 Simon, H., P. Vu, and C. Yang, "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.

5.12 Storaasli, O.O., D.T. Nguyen, and T.K. Agarwal, "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.

5.13 Storaasli, 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.

6 Parallel-Vector Variable Bandwidth Out-of-Core Equation Solver

6.1 Introduction

For large-scale finite element based structural analysis, an out-of-core equation solver is often required since the in-core memory of a computer is very limited. For example, the Cray Y-MP has only 256 mega words incore memory compared to its 90 gigabytes of disk storage. Furthermore, in a multi-user environment, each user can only have 10 mega words of main memory, while 200 mega words of disk storage is available. A typical aircraft structure (High Speed Civil Transport Aircraft) needs 90 million (or mega) words of incore memory to store the stiffness matrix, which is not usually available in a multi-user environment.

This chapter presents vector and parallel out-of-core equation solution strategies which exploit features of the Cray type computers. For out-of-core solution strategies, considerable amount of input/output (I/O) is usually required. The input/output (I/O) time can be reduced by using a synchronous BUFFER IN and BUFFER OUT, which can be executed simultaneously with the CPU instructions. The parallel and vector capability provided by the supercomputers is also exploited to enhance the performance.

6.2 Out-of-Core Parallel/Vector Equation Solver (version 1)

To solve the following systems of linear equations

$$Ax = b \tag{6.1}$$

where A is a n x n symmetric, positive definite matrix, one first factorizes the matrix A into the product of two triangular matrices

$$A = U^T U \tag{6.2}$$

where U is the upper triangular matrix. Then, the solution vector x can be obtained through the standard forward/backward elimination

$$U^T y = b \quad (solve\ for\ y) \tag{6.3}$$

$$Ux = y \ (solve\ for\ x) \tag{6.4}$$

6.2.1 Memory usage and record length

The matrix A is stored in a one-dimensional array with row-oriented storage scheme [6.1-6.2], and each 8 rows of the matrix A have the same last column number (loop-unrolling level 8, or loop = 8), as has been described in Figure 5.26 of Chapter 5. Assuming A is written in a file on the disk, the file contains m records (each record contains one or more block-rows of A, here one block-row has 8 rows). The required in-core memory is assigned in the variable "mtot" and the maximum half bandwidth of A is "maxbw." To reduce the input/output (I/O) time during the solution procedure, one hopes to reduce the number of records "m," while the available in-core memory "istorv" should be capable to hold at least 3 records at any moment. When "istorv" and "maxbw" are given, the procedure (given in Table 6.1) is used to determine the record length and the number of records stay simultaneously in the main memory. In this parallel-vector out-of-core (version 1) strategy, the total required incore memories is (6* neq) + 1.1(maxbw)^2 where neq is the number of equations, or the number of unknowns in Eq. 6.1, and maxbw is the maximum half-bandwidth.

Table 6.1 Procedure to find record length and number of records

```
1. C     Definitions:
2. C     istorv     -----   in-core memory available.
3. C     mtot       -----   in-core memory required.
4. C     loop       -----   loop-unrolling level, here: loop = 8
5. C     maxbw      -----   maximum half-bandwidth.
6. C     icstore    -----   number of records stay in the memory at any time.
7. C     nloop      -----   number of "blocks" in a record, one block = 8 rows.
8.                  nloop = maxbw / (4 + loop)
9.                  if (nloop.le.1) nloop = 1
10. C *****         Find out the number of records required to be kept in the memory
*******
11.                 icstore = 2 + max (1, maxbw / (nloop*loop) + 1)
12. C*****          Find out the in-core memory required: mtot
13.                 mtot = loop*nloop*maxbw*icstore
14. C
15. 100  continue
16. C
17. C*****          Check if the available in-core memory is enough or not
**********
18.                 if (mtot.lt.istorv) then
19. C****           more blocks can be included in one record ***************
20.                 idelt = (istorv-mtot) / (loop*icstore*maxbw)
21.                 nloop = nloop + idelt
```

```
22.                   if (nloop.le.0) nloop = 1
23.                   icstore = 2 + max (1, maxbw/(nloop*loop) +1)
24.                   mtot = loop*nloop*maxbw*icstore
25.                   idelt = (istorv - mtot) / (loop*icstore*maxbw)
26.                   nloop = nloop + idelt
27.                   if (nloop.le.0) nloop = 1
28.                   icstore = 2 + max (1, maxbw/ (nloop*loop) + 1)
29.                   mtot = loop*nloop*maxbw*icstore
30.                   idelt = (istorv - mtot) / (loop*icstore*maxbw)
31.                   else
32. C*****            Too many blocks in one record, take some out!********
33.                   idelt = (istorv - mtot - 1) / (loop*maxbw) - 1
34.                   idelt = idelt / nloop - 1
35.                   endif
36.                   nloop = nloop + idelt
37.                   if (nloop.le. 0) nloop = 1
38. C*****            Check again if mtot < istorv or not ********************
39.                   icstore = 2 + max (1, maxbw / (nloop*loop) + 1)
40.                   mtot = loop*nloop*maxbw*icstore
41.                   mloop = nloop * loop
42.                   IF (MTOT.LT.ISTORV) go to 200
43.                   if (nloop.eq.1) then
44.                   write (*,*)'** Please increase the in-core memory to: ',mtot+1
45.                   stop
46.                   endif
47.                        go to 100
48. C******************** End of looking for nloop, icstore *******
49.         200 continue
```

The above procedure will adjust the number of blocks in a record automatically, and give the optimal values for "nloop" and "icstore." It will also determine the minimum incore memory required (=mtot+1) for a given "maxbw" (mtot is close to maxbw*maxbw).

The following numerical data will help the readers to clarify the procedure given in Table 6.1.

Assuming the maximum half-bandwidth maxbw = 600, and the incore memory available is istorv = 900,000, then from Table 6.1, one has

from line 8:	nloop = 50 blocks
from line 11:	icstore = 4 records
from line 13:	mtot = 960,000 words of required memory
from line 18:	because of this IF statement, the algorithm will jump to line 33
from line 33:	idelt = (900,000-960,000-1)/(8*600)-1=-13

from line 34:	idelt = -13/50 -1 =-1
from line 36:	nloop=49
from line 39:	icstore = 4
from line 40:	mtot = 940,800
from line 41:	mloop = 392
from line 42:	because of this IF statement, the algorithm will jump back to line 15, 18 and 33
from line 33:	idelt = (900,000-940,800-1)/(8*600)-1 = -9
from line 34:	idelt = -1
from line 36:	nloop = 48
from line 39:	icstore = 4
from line 40:	mtot = 921,600
etc... etc...:	the value of mtot will keep gradually decreasing, until it reaches mtot ≤ 900,000, then the algorithm will stop.

It is a helpful exercise to consider another data case, where maxbw = 600 and istorv = 980,000

6.2.2 A synchronous input/output on Cray computers

Considerable input/output (I/O) work is required for an out-of-core equation solver during the solution, which undoubtedly increases the solution time. Fortunately, the Cray computers offer BUFFER IN and BUFFER OUT [6.3] as an extension of the regular Fortran READ and WRITE statements. BUFFER IN and BUFFER OUT can perform several I/O operations concurrently, and I/O operations can be executed concurrently with CPU instructions. It is required that the files should be declared as unblocked.

A typical use of BUFFER IN and BUFFER OUT statements can be written as shown in Table 6.2

Table 6.2 Formats of BUFFER IN and BUFFER OUT statements

```
C******          To read a record from file (or unit number) id *********
                 call setpos (id, ilocate)
                 buffer in (id, m) (a(start), a (start+length-1))
      IF [unit (id). NE. -1.0] Go TO 99
C******          Calculation can be performed (with care!) during Buffer-In, by
C******          removing the above IF statement
C******          To write a record on file (or unit number) id ***********
                 call setpos (id, ilocate)
                 buffer out (id, m) (a(start), a (start+length-1))
      IF [unit (id). NE. -1.0] GO TO 99
C***** Calculation can be performed (with care!) during Buffer-Out, by removing the
C***** above IF statement
99               Write (6,*) 'error encountered during buffer In/Out'
```

```
C***** where
C***** id        --------  is a unit identified (or a file name) number
C***** ilocate   --------  is the beginning location of the record.
C***** m         --------  is a mode identifier (m=1 in this chapter).
C***** a         --------  is an array (in-core memory) to hold the record.
C***** start     --------  is the start location of the record in array a.
C***** length    --------  is the length of the record.
C***** Note:     in the above code, an IF check can be used to be sure that the code
C***** will go to the next statement only when Buffer-In/Out are "normally"
C*****           completed, or else the program will either wait or print error
                 message.
```

6.2.3 Brief summary for parallel-vector incore equation solver on the Cray Y-MP [6.1]

In order to facilitate the discussion of the out-of-core version of the equation solver in the next section, the parallel-vector incore version [6.1] is summarized here.

Since the major portion of the total solution time for solving systems of linear equations occurs during the factorization phase ($A=U^TU$). Parallel-vector (incore) factorization is summarized here. Using the Choleski method, the factorized upper triangular matrix U can be computed as

$$U_{ij} = \frac{A_{ij} - \sum_{k=1}^{i-1} U_{ki}\, U_{kj}}{U_{ii}} \qquad for\ i \neq j \tag{6.5}$$

$$U_{ii} = \left(A_{ii} - \sum_{k=1}^{i-1} U_{ki}^2 \right)^{1/2} \qquad for\ i = j \tag{6.6}$$

As an example, for i=5 and j=7, one has:

$$U_{57} = \frac{A_{57} - U_{15}U_{17} - U_{25}U_{27} - U_{35}U_{37} - U_{45}U_{47}}{U_{55}} \tag{6.7}$$

From the above formula, one can see that to update the term U_{57} (of row 5), one only needs information from columns 5 and 7. Therefore, to update the entire row# i, one needs the complete updated information (right above row i) as shown in Figure 6.1 (where the matrix A is assumed to be banded to simplify the discussion). Also from the above formula, one can see that if only rows 1 and 2 have been completed (even though rows 3 and 4 have not been completed), the term U_{57} (or the entire row 5) can be partially completed.

For example

$$U_{57}\ (\text{incomplete}) = A_{57} - U_{15}U_{17} - U_{25}\,U_{27} \tag{6.8}$$

The above observation will immediately suggest the parallel procedure for obtaining the factorized matrix U. Each processor will handle the updating of one row. Furthermore, to exploit the vector capability of Cray type computers, loop-unrolling technique [6.1] is used. In the above formula for U_{57}, assuming loop-unrolling level 2 is employed, what it means is that every 2 rows are grouped and processed by a processor. In the real computer code implementation, loop-unrolling level 8 is used to optimize the vector speed.

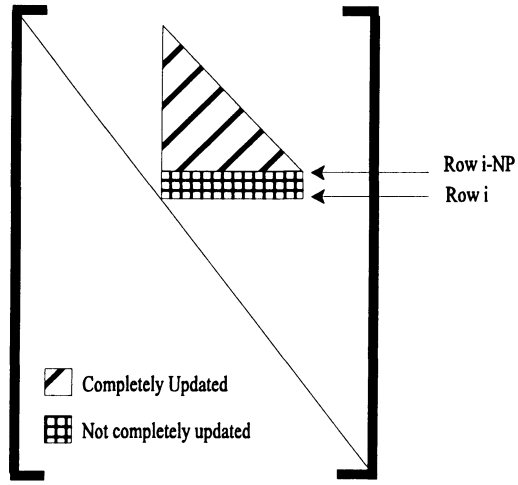


Figure 6.1 Information required to update row i (incore version)

6.2.4 Parallel-vector out-of-core equation solver on the Cray Y-MP

For an in-core solver, during the factorization of A, the rows of A will be updated (which now becomes the rows of U) and stored in the same locations of A (from row 1, row 2,..., to row N). For an out-of-core solver, however, since only a small part of A is currently stored in the memory, it is necessary to write (BUFFER OUT) the rows of U on the disk file, and to read (BUFFER IN) the other rows of A into the memory. In this proposed out-of-core solver, a record will be BUFFER OUT when it is completely updated, and a new record will be BUFFER IN as soon as the first row of a record is begun to be updated (see Figure 6.2). This can also be shown in Table 6.3 (to simplify the discussion, assuming NP = 1 processor).

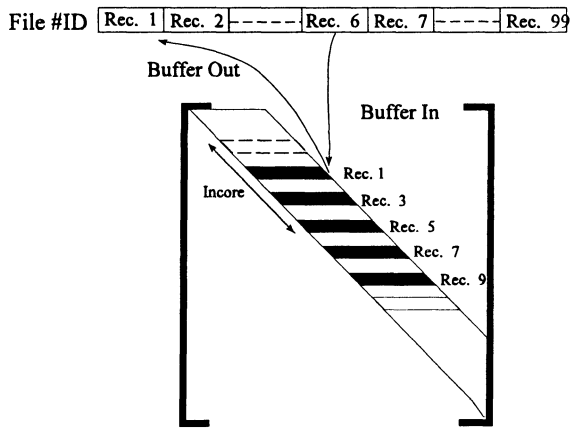


Figure 6.2 Parallel-vector out-of-core Choleski

In Figure 6.2, assuming the incore memory available is enough to hold only, say 7 records, the first two "unmarked" records are used to store some temporary working spaces. Information related to the descriptions of the (stiffness) matrix, such as: column heights, variable row-lengths (or variable row bandwidths), diagonal pointers, right-hand-side (or load) vector etc... (recall Eqs. 5.3, 5.4 & 5.7, in Chapter 5) will require an additional (6*neq) words of incore memory. The remaining incore memories will be used to hold, say 5 records at a time (=records 1, 2..., 5) of the stiffness matrix. The parallel-vector out-of-core factorization can be summarized in the following key steps (please also refer to Fig. 6.2):

Step 1: The first record (=record 1) can be completely factorized (according to the parallel factorized algorithm described in Section 5.6 of Chapter 5)
Step 2: The "completely factorized" record 1 is used to "partially factorized" records 2, 3..., 5 (recalled that 5 records can be resided in the core memory, at any moment)
Step 3: Buffer-out (or write) record 1 into auxiliary storage files
Step 4: Buffer-in (or read) record 6 into the core memory (overwrite the incore memory of record 1)
Step 5: Record 2 will be completely factorized (similar to step 1)
Step 6: The "completely factorized" record 2 is used to "partially factorized" records 3, 4,... , 6 (similar to step 2)
Step 7: Buffer-out record 2 (similar to step 3)
Step 8: Buffer-in record 7 into the core memory (overwrite the incore memory of record 2, similar to step 4)

The above 4-step cycle will be repeated until all records are completely factorized.

Table 6.3 Typical use of BUFFER IN and BUFFER OUT statements

```
1.          DO 1000 I = 1, n
2. C...     determine if "buffer in" is required
3.          if (I - ((I - 1) / (nloop*loop)) * (nloop*loop). eq. NP) then
4.          call setpos (id, ilocate)
5.          call buffer in (a(start), a(start + length-1)
6.          endif
7.c...      update the I-th row of A
8.          .....
9.          ....
10. c...    The I-th row of A has been updated
11.c...     determine if "buffer out" is required
12.         if (I - ((I -1) / (nloop*loop)) * (nloop*loop). eq.NP) then
13.         call setpos (id, ilocate1)
14.         call buffer out (a(start1), a(start1 + length1-1)
15.         endif
16.   1000 continue
```

In Table 6.3, assuming I = 64th row, NP = 1 processor, loop = 8 rows per block, and nloop = 1 block, then according to the algorithm in Tables 6.3 (with I = 64th row, or the last row of the record), one has

from line 1:	I = 64
from line 3:	Because of this IF statement, the algorithm will jump to line 7
from line 7:	row #64 (=last row of the current record) is being updated (or factorized)
from line 12:	Because of this IF statement, the algorithm will jump to line 16, and go back to line 1
from line 1:	I = 65 (=beginning row of the next record)
from line 3:	This IF statement will lead to line 4
from lines 4-6:	Buffer-in (or read) a new record into the core memory
from lines 7-10:	row #65 (=beginning row of the new record) is being updated (or factorized)
from lines 11-12:	This IF statement will lead to line 13
from lines 13-15:	Buffer-out (or write) the previous (completely factorized) record into auxiliary disk storage files
from line 16:	the next row, I=66, will be processed. The process will then be repeated.

In actual computer code implementation, the addresses such as ilocate, start, length, ilocate1, start1 and length1 should be properly defined to ensure a correct solution. In a parallel computer environment, only one processor will be assigned to deal with the I/O, while other processors will directly (and simultaneously) do the calculations. A similar I/O pattern is used in the forward/backward elimination phases.

Loop-unrolling level 8 is adopted in this work to enhance the vector performance of the solver. A parallel Fortran Language **Force** (**For**tran **C**oncurrent **E**xecution [6.4] is used here to develop a parallel version of the out-of-core solver. For a reference on the parallel/vector aspects of the in-core solver, see Ref. [6.1]. Algorithms discussed in this chapter can also be implemented in the PVM, or MPI environments [see sections 4.5, and 5.5 of Chapters 4, and 5, respectively].

6.3 Out-of-Core Vector Equation Solver (version 2)

6.3.1 Memory usage

The matrix A is stored in a one-dimensional array using a row-oriented storage scheme [6.1], and each 8 rows of the matrix have the same last column number (loop-unrolling level 8). It has been concluded in Refs. [6.1-6.3] that for Cray-type supercomputers, "saxpy" operations is faster than "dot product" operations, hence a row-oriented scheme is a more preferred choice as compared to skyline scheme. Assuming A is written in a file on disk, with [can be either "regular" disk, or solid state disk (ssd)], the file containing say, 10 records (or nblk = 10), and each record containing blocks of 8 rows. The last record contains the remaining data for the coefficient stiffness matrix [A] (see Figure 6.3). To simplify the discussion, it is further assumed the user wishes to declare the available incore memory is (IM) words, and 4 blocks of data for the coefficient stiffness matrix A can be brought into the core memory (or ntblk = 4) as shown in Figure 6.3. Some important information on the incore and out-of-core memory management schemes are defined in Table 6.4. It should also be emphasized here that this out-of-core strategy has the flexibility to use as little incore memory as (6 * neq)+(16* maxbw), or as much incore memory as specified by the user through the input data variable IM [see Table 6.4].

This kind of flexibility in using the incore memory will enhance the performance of the proposed out-of-core solve since the Input/Output (or I/O) time can be reduced when more incore memory are specified.

6.3.2 Vector out-of-core equation solver on the Cray Y-MP

For an in-core solver, during the factorization of A, the rows of A will be updated (which now becomes the rows of U) and stored in the same locations of A (from row 1, row 2,..., to row neq). For an out-or-core solver, however, since only a small part of A is currently stored in the memory, it is necessary to write (BUFFER OUT) the rows of U on the disk file, and to read (BUFFER IN) the other rows of A into the memory (see Figure 6.3). In this proposed out-of-core solver, a record will be BUFFER OUT when it is completely updated, and a new record will be BUFFER IN as soon as the first row of a record is begun to be updated (see Figure 6.3). This can also be shown (with key out-of-core strategies) in Table 6.5 (to simplify the discussion, assuming NP = 1 processor).

Table 6.4 Definition of important variables used in the memory management scheme

(a)	nbkl (say = 10) = total number of records to write (and read) on (and from) the disk.
(b)	Each record contains multiple of 8 rows of data for the coefficient matrix [A]. The last record, however, contains the remaining data for [A].
(c)	ntblk (input data, say = 4) = number of blocks of data of the coefficient matrix [A] that can be brought into the core memory.
(d)	Each block has enough incore-memory (multiple of 8 rows, plus a small, unused or left over incore memory) to hold the largest record of data [see record number 7 in Figure 6.3].
(e)	IM = user's input data for total available incore memory [see Figure 6.3].
(f)	IM for [A] = available incore memory (in words) to store ntblk blocks of the coefficient stiffness matrix [A]
(g)	$neqbk = \frac{IM\ for\ \ [A]}{ntblk}$ = number of words per block
(h)	$\max(nblk) = \frac{neq}{\left(\frac{neqbk}{maxbw * 8} * 8\right)}$ = maximum possible number of records
(i)	neq = total number of equations
(j)	maxbw = maximum bandwidth
(k)	$neqq = \frac{IM for [A]}{maxbw}$ = number of equations (based on available incore memory for [A])
(l)	$ntblkq = \frac{neqq}{8}$ = number of block-rows of equations (based on IM for [A])

Table 6.5 Major sketches of the vectorized out-of-core factorization

1.	C*****	ntblk = 4 (say, user input data)
2.	C*****	Loop through all records
3.		DO 1 N = 1, nblk (say = 10 records, and currently N = 7)
4.		KNM = N - ntblk + 1 (=7-4+1=4)
5.	C*****	To factorize current record N, one needs information on all previous records.
6.	C*****	Hence, loop 2 is required
7.	C*****	SRN = Starting Record Number (=known, or already computed value)
8.		DO 2 M = SRN, N-1 (currently, say SRN = 1)
9.	C*****	Find the (incore) block number nb that can be temporary used for BUFFER-IN
10.	C*****	information
11.	C*****	Note: nb≥ 1 and nb ≤ ntblk
12.	C*****	If current updated (or factorized) block is block #3, then nb = 4 (always
	C*****	= top block # of the current 4 blocks residing in the incore memory).
13.	C*****	If current updated (or factorized) block is block #2, then nb = 3
14.	C*****	If current updated (or factorized) block is block #1, Then nb = 2
15.	C*****	If current updated (or factorized) block is block #4, then nb = 1
16.	C*****	Some required info. Still stay in SSD (or in regular disk).
17.	C*****	Thus, if check is used to see if BUFFER-IN is required
18.		IF (M.LE.KNM) THEN
19	C*****	Read (or Buffer–IN) one record into the appropriate incore block # (say nb). Thus,
20.	C*****	Buffer-In data
21.	C*****	Will be stored in the coefficient matrix array A (1, nb)
22.		ELSE
23.	C*****	some required information are already stayed in the incore memory
24.		ENDIF
25.	C*****	"Partially" factorize current record #N using loop-unrolling technique [6.1-6.2]
26.	2	Continue
27.	C*****	Calculate the "Final" factorized terms in the current record #N, then
28.	C*****	BUFFER-OUT
29.	C*****	Since "partially" factorization has already been done using the previous record
30.	C*****	information, at this stage, we only need information from the current record #N
31.	1	Continue

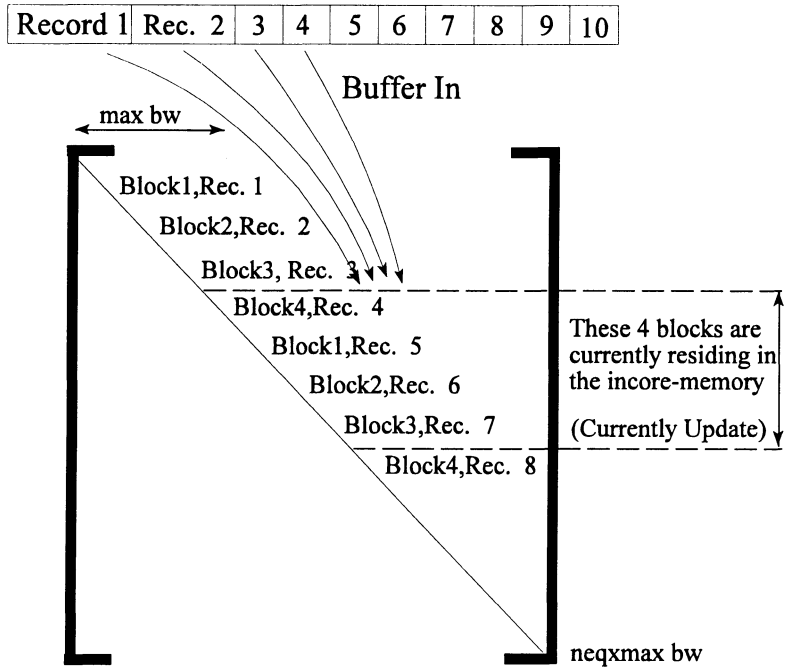


Figure 6.3 Out-of-core Choleski factorization (version 2)

Let's try to take a closer look at Table 6.5 and Figure 6.3, at the same time. In Fig. 6.3, we assume that records 4, 5, 6, & 7 are currently residing in the core memory, and record 7 is currently being factorized (also, refer to line 3 of Table 6.5). The formula shown in line 4 (KNM=4, in Table 3) will be used later on, in line 18 (of Table 6.5), to determine which records have already been resided in the core memory (thus, to avoid wasteful time to buffer-in these same records into the core memory). Since record N=7 is currently being factorized, previous completely factorized records (such as records 1, 2, ... 6) may be required, or to be more general, previous records SRN, SRN +1, SRN +2, ..., N-1 may be required (where SRN is the starting record number, which can be either record 1, or 2, or 4 depending on the problems). Statements on lines 9-15 are basically used to describe the rule to select the incore block number to be used as temporary incore storage space to hold those previously (& completely) factorized records. Line 18 stated that if the previous record #M (which is needed in order to factorize the current record N=7) is less than KNM (=4), then we need to buffer-in (or read) from the auxiliary storage disk space. On the other hand, if M is greater than KNM (=4), such as M = 5 or 6, then there is no need to buffer-in records #5 & 6 into the core memory (since these records #5 & 6 have already been resided in the core memory!) We just simply used the available records #5 & 6 to partially factorize record # N = 7. By the time we have completed loop 2 (from line 8 to line 26), all previous records have been used to partially factorize record N = 7. Hence, in line 27, the "completely factorized" record N can be obtained by using the information from the current record #N itself.

The Solid State Disk (SSD) storage available on the Cray-YMP can further significantly reduce the I/O time. For convenience, Table 6.6 summarizes the step-by-step procedure to run the developed out-of-core solver (where outof is the name of executable code, in is the name of the input data file, outputt is the name of output data file, the original coefficient stiffness matrix [A] is generated and stored in the file fort. 11, and the factorized of [A] is computed and stored in fort. 22).

Table 6.6 Procedure to execute out-of-core solver (version 2)

	On Fast Cray-YMP SSD (Note: ">" is the computer prompt sign)
Step 1	> ▽ srfs ▽ -r ▽ 15 mw ▽ $ FASTDIR
Step 2	> ▽ cp ▽ outof ▽ $ FASTDIR
Step 3	> ▽ cp ▽ in ▽ $ FASTDIR
Step 4	>▽ cd ▽ $ FASTDIR
Step 5	> ▽ assign ▽ -s ▽ unblocked ▽ u:11
Step 6	> ▽ assign ▽ -s ▽ unblocked ▽ u:22
Step 7	> ▽ outof ▽ < in ▽ >outputt
Step 8	(to release fast SSD to other users)
	> ▽ srfs ▽-r ▽ 0mw ▽ $FASTDIR (by specifying zero million words required.)
	>▽ srfs ▽ -i ▽ $FASTDIR
	>▽ srfs ▽-u

Version 2 of the out-of-core solver has been coded in a user's friendly, modular form as shown in Table 6.7.

Table 6.7 How to call the present out-of-core (version 2) subroutine "solve"??

	Subroutine solve (b, a, maxa, kbin, neq, neqr, nblk, ntblk, neqbk, ihu)
C...	implicit real * 8 (a-h, o-z)
C...	dimension b(neq), a(neqbk, ntblk), maxa(neq+1), kbin(5, nblk), ihu(neq)
C****	notes:
C****(a)	neqr =unused information (for future development)
C****(b)	kbin(1, nblk) = lowest equation (global) number in a block
C	kbin(2, nblk) = highest equation (global) number in a block
C	kbin(3, nblk) = (local) number of terms in a block
C	kbin(4, nblk) = (global) lowest coupled equation number (for every record)
C	kbin(5, nblk) = lowest coupled block number
C****(c)	ihu(neq) = same as kbin (4, nblk), but for every row
C****(d)	b(neq)= vector of known right hand side

C****(e)	a(neqbk, ntblk) = a 2-D array to store coefficient matrix in blocks
C****(f)	maxa(neq+1) = locations of diagonal pointers for the coefficient matrix
C****(g)	neq = number of equations
C****(h)	ntblk = number of blocks (for incore memory) of the coefficient matrix [a]
C****(i)	neqbk = number of words (memory) per block

The objective of the following section is to find the mapping (or, the corresponding location) of a general term of global stiffness matrix, such as K_{ij}, in a local stiffness matrix which is resided in four in-core memory blocks.

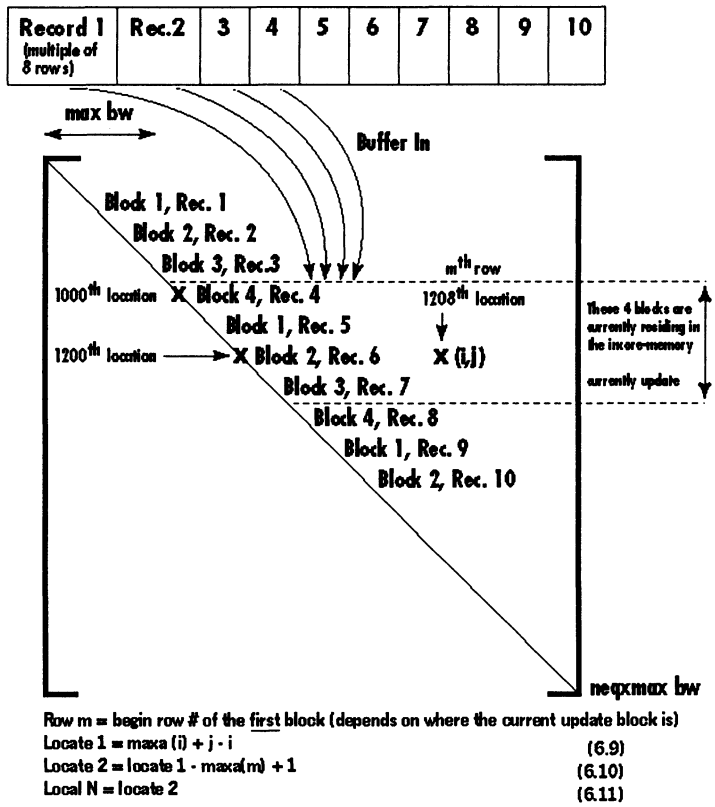


Figure 6.4 Mapping between global (K_{ij}) location and local (N) location (version 2)

In the above formulas (Eqs. 6.9-6.11), assuming m^{th} row = 100^{th} row, $K_{i,j} = K_{120,128}$, MAXA(i)= Maxa(120) = 1200, MAXA(m) = MAXA(100)= 1000, then

Locate 1 = Maxa(i) + j -i = Maxa(120) + 128 - 120 = 1208

Locate 2 = 1208 - Maxa(100) + 1 = 209

Local N = 209

Thus, the stiffness matrix term $K_{120,128}$ is located at the 1208th global location. Its corresponding local location is the 209th location.

A step-by-step procedure to generate and assemble element stiffness matrices (of structural applications) in an out-of-core fashion (version 2) is shown in Table 6.8.

Table 6.8 A step-by-step procedure to generate and assemble element stiffness matrices of **real** structures in an out-of-core (version 2) fashion

Step 1:	Knowing element connectivities and the global DOF associated with each element, we can find the so-called MAXA(-) array (diagonal pointer locations)
Step 2:	Knowing MAXA(-) array, we can find maximum and average bandwidth (say max bandwidth = 120, ave. bandwidth = 80 and neq = 88000)
Step 3:	Knowing Incore Memory (= IM) available for [K], and knowing the number of blocks (=ntblk = 4, usually) that the user wishes to partition the IM for [K], we can find the number of words per block (= neqbk) $$neqbk = \frac{IMfor\ [K]}{ntblk}$$
Step 4:	Based on the known max. bandwidth and neqbk, we can find how many block rows (each block row = 8 due to loop-unrolling) a record (on regular disk or SSD disk) can hold $$NBlkrows = \frac{neqbk}{(\text{max. bandwidth} * 8)}$$
Step 5:	Find how many rows (must be a multiple of 8 due to loop-unrolling) a record (on regular disk or SSD disk) can hold $$Nrows = Nblkrows * 8 \quad (\text{Say } Nrows = 1000)$$
Step 6:	Find the maximum record length (on regular disk or SSD disk) we can have Record Length = Nrows * maximum bandwidth (say record length = 120,000)
Step 7:	Find how many records (on regular or SSD disk) can we have. Nrecords=(neq*max. bandwidth)/record length (say=88000* 120/120,000=88)
Step 8:	Thus in this example, each record (on regular or SSD disk) can hold Nrows (=1000 rows). There are a total of Nrecords (=88) on the regular or SSD disk. Hence, record 1 will store information from row 1 to row 1000; record 2 will store information from row 1001 to row 2000; record 87 will store information from 86,001 to row 87,000; (last) record 88 will store information from remaining rows
Step 9:	Assuming a particular 9 x 9 element global stiffness is associated with the Global DOF {101, 102, 103, 104, 105, 106, 2078, 2079, and 2080}. Then, the element will be generated, assembled and "partially" stored in Record 1(correspond to global DOF #101 → 106) and also "partially" stored in Record 3 (correspond to global DOF 2078 → 2080).

6.4 Out-of-Core Vector Equation Solver (version 3)

The solution strategies used in this section is quite similar to the one discussed in the

previous section. However, this proposed out-of-core, vectorized equation solver strategy is believed to be more efficient than the previous one, as it will be explained in the following paragraphs.

There are a total of 10 records of the global stiffness matrix to be stored in the Solid State Disk (SSD), as shown in Fig. 6.5. A small fraction of the total Incore Memory available (=IM words) is used to store certain information about the stiffness matrix (such as column heights, variable row lengths, diagonal location pointers etc....). The remaining incore memory will be partitioned into 3 blocks (instead of 4 blocks as discussed in the previous section), as indicated in Figure 6.5. The key out-of-core equation solution strategies can be explained by referring to Fig. 6.6. To make the discussion more general, we assume that records 5, 6 and 7 are currently residing in the core memory, and record 7 is being factorized according to the following step-by-step procedure:

Step 1: Use record 5 to partially factorize record 7

Step 2: Use record 6 to partially factorize record 7, and simultaneously buffer-in (or read) record 1 into the core memory (and overwrite memory spaces previously occupied by record 5)

Step 3: Use record 1 to partially factorize record 7, and simultaneously buffer-in record 2 (and overwrite memory spaces previously occupied by record 6)

Step 4: Use record 2 to partially factorize record 7, and simultaneously buffer-in record 3 (and overwrite memory spaces previously occupied by record 5).

The above steps are repeated until all previous records have been used to factorize the current record 7.

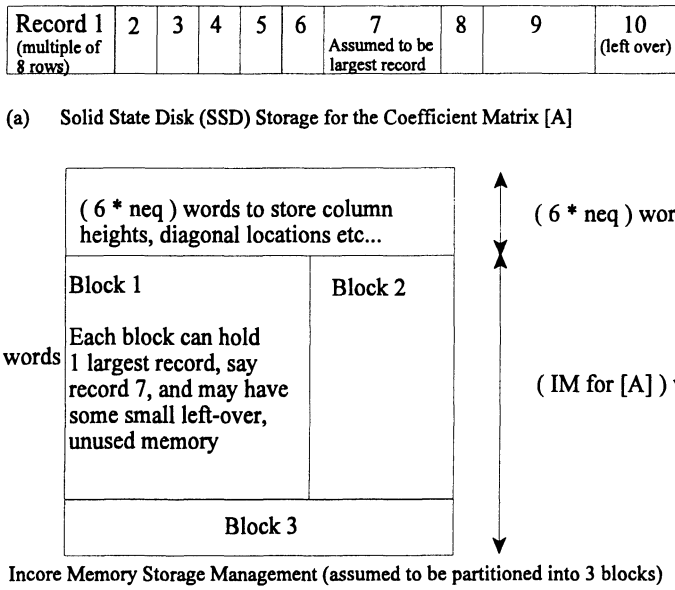


Figure 6.5 Out-of-core memory management scheme (version 3)

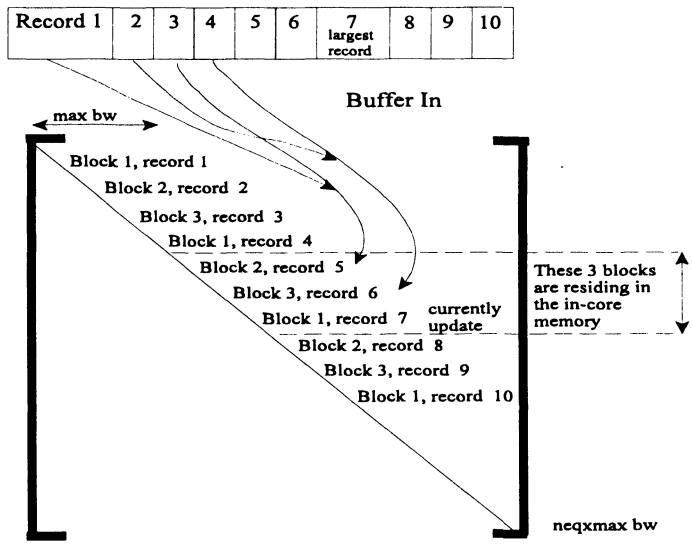


Figure 6.6 Out-of-core Choleski factorization

6.5 Application

6.5.1 Version 1 performance

To test the effectiveness of the proposed out-of-core parallel-vector equation solver

(pvsolve-ooc), described in Section 6.2, three 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 Civil Transport aircraft (HSCT) and the space shuttle Solid Rocket Booster (SRB). The aircraft and SRB models are selected as they were large, available finite-element models of interest to NASA. The characteristics of the stiffness matrix for each of the above practical finite element models is shown in Table 6.9.

Table 6.9 Characteristics of finite element models

	HSCT	Refined HSCT	SRB
Max Bandwidth	600	1272	900
Ave. Bandwidth	321	772	383
Matrix Terms	5,207,547	12,492,284	21,090,396
Non-zero Terms	499,505	373,752	1,310,973
No. Operations	171,425,520		9.2×10^9
No. Equations	16,146	16,152	54,870

In the following applications, code is inserted in pvsolve-ooc to calculate the CPU time for equation solution. The Cray timing (TSECND) is used to measure the time.

Example 1: High Speed Civil Transport Aircraft (HSCT) Application

To evaluate the performance of the parallel-vector out-of-core 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 [6.5]. Since the structure is symmetric, a wing-fuselage half model is used to investigate the overall deflection distribution of the aircraft. 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 time taken for a typical finite element code to generate the mesh, form the stiffness matrix and factor the matrix is 325 seconds on a Cray 2 (802 seconds on a CONVEX 220) of which the matrix factorization is the dominant part. Using pvsolve-ooc, the factorization for this aircraft application requires 6.98 and 1.01 seconds on one and eight Cray Y-MP processors, respectively, as shown in Table 6.10.

Table 6.10 Performance of pvsolve-ooc on Cray Y-MP

No of Processors	HSCT	Refined HSCT	SRB
1	6.98 (sec.)	43.87 (sec.)	31.26 (sec.)
2	3.50	20.00	15.53
4	1.85	10.00	7.80
8	1.01	5.71	4.21

Example 2: Refined Model for HSCT problem

More details and more realistic model of the HSCT structure than the crude model (in Example 1) is used. The characteristics of the resulted stiffness matrix is shown in Table 6.9. The numerical performance of the proposed parallel-vector out-of-core solver for this example is presented in Table 6.10.

Example 3: Space Shuttle Solid Rocket Booster (SRB) Application [6.6-6.7]

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

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. 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 bandwidth of 900 and an average bandwidth of 383. A detailed description and analysis of this problem is given in references [6.6-6.7]. The times required for a typical finite element code to generate the mesh, form the stiffness matrix and factor the matrix are about one-half hour on the Cray 2 (15 hours on a VAX 11/785) of which the matrix factorization is the dominant part. Using pvsolve-ooc, the factorization for this SRB problem requires 31.26 and 4.21 seconds on one and eight Cray Y-MP processors, respectively, (as shown in Table 6.10).

6.5.2 Version 2 performance

To evaluate the performance of the vector out-of-core Choleski solver discussed in Section III, a "simulated" structural static analysis has been performed on a 16,146 degree-of-freedom finite-element model of a high-speed aircraft concept [6.5]. Since the structure is symmetric, a wing-fuselage half model is used to investigate the overall deflection distribution of the aircraft. 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.

Since this is only a "simulated" high-speed aircraft model, the input data can be easily prepared with only a few defined variables: neq, (number of equation), avebw (average bandwidth), IM (Incore Memory Available), ntblk (number of incore memory blocks).

In the following numerical examples, the variables neq, avebw, and ntblk are set to be 16146, 321, and 5. However, the input parameter IM (specified Incore Memory available) will be varied in order to see how the solution time varies with the specified incore memory available. For this "simulated" aircraft model, 3 numerical cases will be studied. Since this is a simulated problem, maxbw= avebw = 321.

Case 1: IM = (6* neq) +(16 * maxbw)
Case 2: IM = (6 * neq) + (1.1 * maxbw^2)
Case 3: IM = (6 * neq) + (neq * avebw)

Thus, Case 1 uses the minimum incore memory, Case 2 uses the same minimum incore memory as in Section 6.2, and Case 3 uses complete incore memory. The performance of this simulated aircraft model in all 3 cases are summarized in Table 6.11. All results are accurate as compared to known solutions.

From the results presented in Table 6.11, one can see that there is no general trend on the wall clock time (wct). This es expected, since the wct is heavily depended on how busy the system is at the time the code was executed. The CPU time, however, is more stable and reliable. It shows that case 1 [where only a minimum (16 * maxbw) words of incore memory was used to store the coefficient matrix] requires the most CPU times during the factorization, forward and backward phases. Case 3 [where the equation solution is completely solved by incore memory] requires the least CPU time.

The incore memory used in Case 2 [where (1.1 * maxbw^2) words was used to store the coefficient matrix] is the same as used in Section 6.2. Thus, the trend in CPU time behaves as can be expected: more incore memory used will lead to less CPU solution time.

Table 6.11 CPU (and wall clock) Cray-YMP time (in seconds) performance of the "simulated" high speed civil transport aircraft model

		CASE 1	CASE 2	CASE 3
Factorization	CPU Time	14.29	8.38	8.31
	WCT Time	164.31	34.25	23.40
Forward	CPU Time	0.22	0.14	$7.15*10^{-2}$
	WCT Time	0.88	0.54	$7.15*10^{-2}$
Backward	CPU Time	0.27	0.17	$9.29*10^{-2}$
	WCT Time	1.67	0.49	$9.37*10^{-2}$

6.5.3 Version 3 performance

The 16,146 degree-of-freedom HSCT model will be used to evaluate the numerical

performance of version 3 out-of-core strategies.
Two cases have been considered in version 3 of the computer codes.

Case 1: The income memory provided is large enough to hold "all" arrays inside the core memory. Thus, in this case, only 1 block is used, with a total (integer and real) memory requirement is 5,304,434 words.

The factorization, forward and backward times (for a single Cray-YMP computer) are shown in Table 6.12 as 6.8985 seconds, 0.0433 seconds and 0.0743 seconds, respectively. The total CPU time (including "everything") is 7.0164 seconds, and the total wall-clock-time is 39.6494 seconds.

Case 2: The incore memory provided is less than half of the required incore memory. Thus, in this case, 13 blocks are used, with a total (integer and real) memory provided is only 1,999,988 words.

The factorization, forward and backward times (for a single Cray-YMP computer) are shown in Table 6.13 as 6.9577^{sec}, 0.0436^{sec}, and 0.0748^{sec}, respectively. The total CPU time (including "everything") is 7.0764^{sec}, and the total wall-clock-time is 40.7307 seconds.

Comparing the performances in the above 2 cases, one can see that the proposed out-of-core strategies are quite efficient, since the penalties for the overhead time (when out-of-core strategies are used) is quite small.

Table 6.12 Performance of (version 3) out-of-core solver on HSCT application (incore memory used = 5,304,434)

No. Equations	16,146
Non-zero Terms	499,505
Factorization (CPU) Time	6.90 seconds (Cray-YMP)
Forward (CPU) Time	0.0433 seconds
Backward (CPU) Time	0.0743 seconds
Total (CPU) Time	7.01636 seconds
No. Blocks Used	1
Incore Memory Used	5,304,434 real words

Table 6.13 Performance of (version 3) out-of-core solver on HSCT application (incore memory used = 1,999,988)

No. Equations	16,146
Non-zero Terms	499,505
Factorization (CPU) Time	6.96 seconds (Cray-YMP)
Forward (CPU) Time	0.0436 seconds
Backward (CPU) Time	0.0748 seconds
Total (CPU) Time	7.07636 seconds
No. Blocks Used	13
Incore Memory Used	1,999,988 real words

6.6 Summary

A parallel-vector out-of-core Choleski method (pvsolve-ooc) for the solution of large-scale structural analysis problems has been developed (see Section 6.2) and tested on Cray supercomputers. The method exploits both the parallel and vector capabilities of modern high- performance computers. To minimize computation time and memory requirement, BUFFER IN and BUFFER OUT statements are used for effective I/O operations. In this version (see Section 6.2) the total incore memory requirements is $(6*neq) + (1.1* maxbw^2)$. 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 phase have been found to be more effective to perform sequentially with loop unrolling and vector-unrolling, respectively.

The proposed parallel-vector Choleski method has been used to calculate the static displacements for three large-scale structural analysis problems; a high-speed air craft and the space shuttle solid rocket booster. The total equation solution time is small for one processor and is further reduced in proportion to the number of processors.

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 and low incore memory requirement of pvsolve-ooc 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, and safer designs.

The single-processor, vectorized out-of-core strategies developed in Sections 6.3 and 6.4, and the corresponding incore memory requirements are (6*neq) + (16*maxbw), and (6*neq) + (24*maxbw), respectively. In both proposed vectorized, out-of-core strategies, the required incore memory is significantly less than Version 1 (of Section 6.2). Version 3 (of Section 6.4) is more preferable than Version 2 (of Section 6.3), since with a little more incore memory requirement (24*maxbw, as compared to 16*maxbw), Version 3 is significantly faster than Version 2 (approximately 7 seconds CPU time as compared to approximately 14 seconds CPU time).

6.7 Exercises

6.1 Following the algorithms presented in Table 6.1, and using the data maxbw=600 and istorv=980,000, find the value of "mtot" after 4 iterations??

6.2 Given the following system of equations [A] $\{x\}$ = {b}
where [A] is symmetrical, positive definite matrix and is given as

$$[A] = \begin{bmatrix} 2 & -1 & & & & \\ -1 & 4 & -1 & & & \\ & -1 & 6 & -1 & & \\ & & -1 & 8 & -1 & \\ & & & -1 & 10 & -1 \\ & & & & -1 & 12 \end{bmatrix}$$

$$\{b\} = \begin{pmatrix} 1 \\ 2 \\ 4 \\ 6 \\ 8 \\ 11 \end{pmatrix}$$

Assuming each record in Fig. 6.5 (a) can only hold 1 row of [A], each of the (3) block shown in Fig. 6.5 (b) can only store 1 row of [A], and the first 3 rows of [A] are currently residing in the incore memory:

(a) Use the out-of-core strategies, shown in Fig. 6.6, to factorize [A]. Please also clearly indicate how and where to Buffer IN/Out the information
(b) Find the out-of-core forward solution
(c) Find the out-of-core backward solution

6.8 References

6.1 Agarwal, T.K., Storaasli, O.O. and Nguyen, D.T., "A Parallel-Vector Algorithm for Rapid Structural Analysis on High Performance Computers," Proceedings of the AIAA/ASME/ ASCE/AHS 31st SDM Conference, Long Beach, CA, April 2-4, 1990, AIAA paper #90-1149.

6.2 Qin, J., Agarwal, T.K., Nguyen, D.T., Storaasli, O.O. and Baddourah, M.A., "Parallel-Vector Out-of-Core Equation Solver For Computational Mechanics," presented at the 2nd *Symposium on Parallel Computational Methods for Large Scale Structural Analysis and Design*, Marriott Hotel, Norfolk, VA, sponsored by NASA Langley Research Center, Hampton, VA (February 24-25, 1993).

6.3 NAS User Guide, Version 6.0, NASA Ames Research Center, Moffett Field, CA, 1991.
6.4 Jordan, 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.
6.5 Robins, W.A., et al., "Concept Development of a Mach 3.0 High-Speed Civil Transport," NASA TM 4058, September 1988.
6.6 Knight, 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.
6.7 Knight, 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.

7 A Parallel-Vector Skyline Equation Solver for Distributed-Memory Computers

7.1 Introduction

It is now generally acknowledged that the most feasible and economical means of solving extremely large computational problems in the future will be with massively parallel computers and distributed memory. Though the relatively rapid growth in microprocessor technology over the last decade has lead to the development of massively parallel architectures capable of performing Giga arithmetic operations in a single second, the software required to efficiently solve large-scale problems remains a challenge to scientists and engineers today. A lot of effort has been made to develop efficient equation solvers on parallel computers [7.1], but most of them are either designed for computers with shared memory [7.2-7.6], or for special form of matrices [7.7-7.9], such as tridiagonal matrix, triangular matrix or banded matrix. Since in most scientific and engineering applications, the final systems of equations to be solved are large, symmetrical matrices with variable bandwidths, it is desirable to develop an efficient equation solver that can exploit such special features of the coefficient matrix.

In this chapter, an equation solver for symmetrical matrices with variable bandwidths is developed to solve large-scale problems on massively parallel computers and distributed memory, such as the Intel iPSC/860 hypercube, the IBM-SP2, or Meiko parallel computers [7.10-7.11].

7.2 Parallel-Vector Symmetrical Equation Solver [7.10]

7.2.1 Basic symmetrical equation solver

Systems of linear, symmetrical equations can be represented as

$$Ax = b \tag{7.1}$$

One way to solve Eq. (7.1) is first to decompose the coefficient matrix A into the product of two triangular matrices

$$A = U^T U \tag{7.2}$$

where U is an upper-triangular matrix which can be obtained by

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} u_{ki}\, u_{kj}}{u_{ii}} \qquad (i \neq j) \tag{7.3}$$

$$u_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2 \right)^{1/2} \qquad (i=j) \tag{7.4}$$

Then the unknown vector x can be solved through the forward/backward elimination, such as to solve

$$U^T y = b \tag{7.5}$$

for y, with

$$y_j = \frac{b_j - \sum_{i=1}^{j-1} u_{ij}\, y_i}{u_{jj}} \tag{7.6}$$

and to solve

$$Ux = y \tag{7.7}$$

for x, with

$$x_j = \frac{y_j - \sum_{i=j+1}^{n} u_{ji}\, x_i}{u_{jj}} \tag{7.8}$$

Since the number of operations involved in the factorization phase is much more than that in the forward/backward elimination phase, more efforts will be focused on developing an efficient parallel-vector algorithm for matrix factorization.

7.2.2 Parallel-vector performance improvement in decomposition

The efficiency of an equation solver on massively parallel computers with distributed memory is dependent on both of its vector performance and its communication performance. Since for the Intel/iPSC860 hypercube, the dot product performance is better than its saxpy performance, we have decided to adopt a skyline column storage scheme (column-by-column, from the diagonal term and up of a stiffness matrix[A]) to exploit dot product operations. Moreover, the skyline column storage scheme [7.6, 7.10-7.11] requires less memory than the row-storage scheme [7.5, 7.10]. To enhance its vector speed through vector-unrolling, the symmetrical matrix A is stored in a block-skyline scheme with block size equal to 4 (thus, each block consists of 4 columns). Figure 7.1 shows this storage scheme for multiprocessors (assuming NP = 3 processors are used). Thus, processor 1 stores column 9-12, 21-25 of the matrix A, while columns 13-16, 25-28 and columns 17-20 are held by processors 2 and 3, respectively (recalled Section 3.6 of Chapter 3).

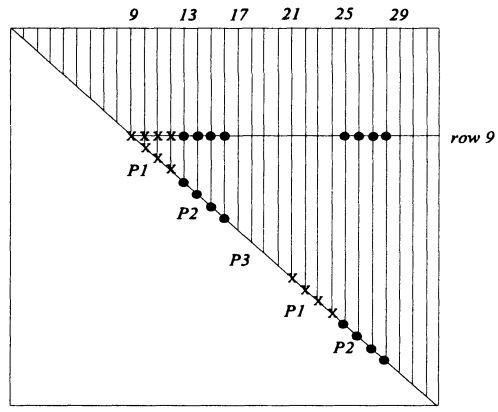


Figure 7.1 Block column storage scheme for matrix [A] (in a one-dimensional array)

The decomposition (or factorization) is processed for i=1, 5, 9,..., n, where for each i the corresponding rows (from i to i+3) can be updated concurrently, by multiprocessors in a row-by-row fashion. Thus, for i=1, rows 1 through 4 (or rows i through i+3) will be factorized by multiple processors. Similarly, for i=5, rows 5 through 8 will be factorized by multiple processors. Assuming the first eight rows of the matrix A (see Figure 7.1) have already been updated by multiple processors, and row 9 is currently being updated. Thus according to Figure 7.1, terms such as $U_{9,9}...U_{9,12}$ and $U_{9,21}...U_{9,24}$ are processed by processor 1. Similarly, terms such as $U_{9,13}...U_{9,16}$ and $U_{9,25}...U_{9,28}$ are handled by processor 2, while terms such as $U_{9,17}...U_{9,20}$ are executed by processor 3.

As soon as processor 1 completely updated column 9 (or more precisely, updated the diagonal term $U_{9,9}$, since the terms U_{19}, U_{29}, ...U_{89} have already been factorized earlier, it will send the entire column 9 (including its diagonal term) to all other processors. Then processor 1 will continue to update its other terms of row 9. At the same time, as soon as processors 2 and 3 receive column 9 (from processor 1), these processors will immediately update its own terms of row 9. In addition to the above parallel computation strategy, more vector speed can be obtained through the concept of "vector unrolling" which has been introduced in Ref. [7.6] for shared memory computers (such as the Cray-2 and Cray Y-MP). Referring to Figure 7.1 and Eqs 7.3 and 7.4 , one can see that having completed column 9, updating the remaining terms of row 9 (such as $U_{9,10}$, $U_{9,11}$ ....$U_{9,n}$) involve with the dot product between 2 columns (column 9 and columns 10, 11, ..., n).

Since column 9 in this example is used repeatedly in the dot product operations, it is desirable to keep column 9 to stay longer in the CPU (or fast memory). Thus, vector unrolling level 4 is used to enhance the vector speed. For example, the following dot product operations

SUM1 = SUM1 + (column 9) · (column 9)
SUM2 = SUM2 + (column 9) · (column 10)

SUM3 = SUM3 + (column 9) · (column 11)
SUM4 = SUM4 + (column 9) · (column 12)
are executed by processor 1, while the following dot product operation
SUM1 = SUM1 + (column 9) · (column 13)
SUM2 = SUM2 + (column 9) · (column 14)
SUM3 = SUM3 + (column 9) · (column 15)
SUM4 = SUM4 + (column 9) · (column 16)
are processed by processor 2 etc... .

A skeleton pseudo-code of the above parallel-vector Choleski factorization is shown in Table 7.1

It is also noticed that if the increment 4 in loop 100 (see Table 7.1) is changed into 1, then no vector unrolling is used.

To further improve the computational efficiencies, block-wise updating strategies are also employed. A block-wise updating (see Figure 7.2) means there are four rows being concurrently up-dated by multiprocessors. A block-wise updating also means that having completed all 4 columns (say 9, 10, 11, and 12 in Figure 7.1), processors 1 will send all these 4 columns to all other processors.

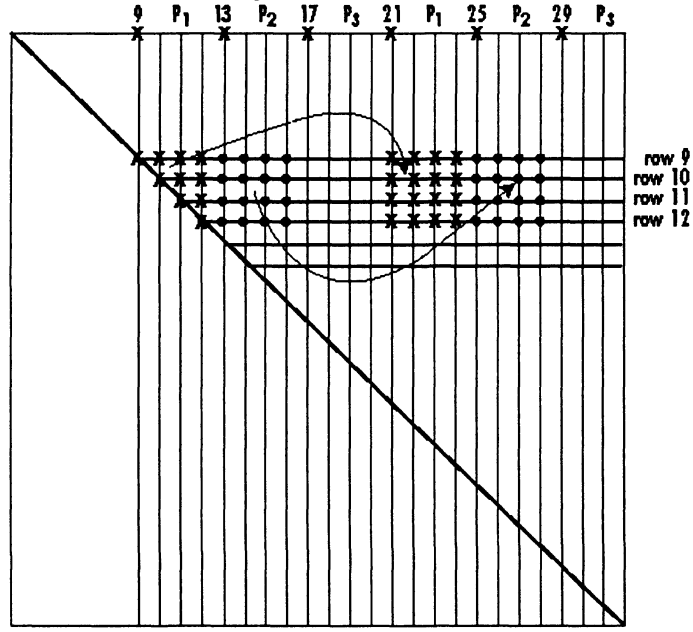


Figure 7.2 Block-wise (rows) factorization for matrix A (in a 1-D array)

In block-wise updating (for factorizing) strategies, each processor will independently factorize its appropriate terms in 4 (or more) consecutive rows. Assuming the first 8 rows of the matrix A (shown in Fig. 7.2) have already been completely factorized, then processor P_1 (which is the owner of columns 9-12) will factorize the following terms.

$$\begin{matrix} U_{9,9} & U_{9,10} & U_{9,11} & U_{9,12} \\ & U_{10,10} & U_{10,11} & U_{10,12} \\ & & U_{11,11} & U_{11,12} \\ & & & U_{12,12} \end{matrix}$$

and send its 4 completely factorized columns (9 through 12) to all other processors, before continuing to factorize other terms (of the 4 consecutive rows 9-12), such as

$$\begin{matrix} U_{9,21} & U_{9,22} & U_{9,23} & U_{9,24} \\ U_{10,21} & U_{10,22} & U_{10,23} & U_{10,24} \\ U_{11,21} & U_{11,22} & U_{11,23} & U_{11,24} \\ U_{12,21} & U_{12,22} & U_{12,23} & U_{12,24} \end{matrix}$$

At the same moment, processor P_2 (and processor P_3 etc...) will receive the 4 columns 9-12 (from P_1) and will factorize the following terms

$$\begin{matrix} U_{9,13} & U_{9,14} & U_{9,15} & U_{9,16} \\ U_{10,13} & U_{10,14} & U_{10,15} & U_{10,16} \\ U_{11,13} & U_{11,14} & U_{11,15} & U_{11,16} \\ U_{12,13} & U_{12,14} & U_{12,15} & U_{12,16} \end{matrix} \quad \textit{and} \quad \begin{matrix} U_{9,25} & U_{9,26} & U_{9,27} & U_{9,28} \\ \downarrow & & & \downarrow \\ U_{12,25} & \longrightarrow & & U_{12,28} \end{matrix} \quad \textit{by } P_2$$

and

$$\begin{matrix} U_{9,17} & U_{9,18} & U_{9,19} & U_{9,20} \\ \downarrow & & & \downarrow \\ U_{12,17} & \longrightarrow & & U_{12,20} \end{matrix} \quad \textit{and} \quad \begin{matrix} U_{9,29} & U_{9,30} & U_{9,31} & U_{9,32} \\ \downarrow & & & \downarrow \\ U_{12,29} & \longrightarrow & & U_{12,32} \end{matrix} \quad \textit{by } P_3$$

A skeleton pseudo-code for this block-wise updating is similar to the one discussed in Table 7.1, with few "minor" modifications, such as:

(a) Deleting the 2nd do-loop in Table 7.1 (thus, lines 5, 6 and 23 need be deleted)
(b) A new, different formula for index II need be defined before entering DO 400 loop (on line 7)
(c) Inside do 500 loop, sixteen (16) dot product operation need be done (instead of just 4 dot product operations as shown in Table 7.1)
(d) Lines 16, 18 and 20 (in Table 7.1) need be expanded for calculating 16 terms (instead of only 4 terms)
(e) Line 17 (of Table 7.1) needs be modified in order to send 4 columns (instead of sending just 1 column) to all other processors
(f) Line 26 (of Table 7.1) should be changed into: receive columns II, II+1, II+2, and II+3
(g) Line 27 (of Table 7.1) should be changed into: update the appropriate processor's terms of rows II, II+1, II+2 and II+3

Still another strategy can be employed to further enhance the performance of

parallel-vector skyline solver, which is illustrated in Figure 7.3. This parallel strategy will be referred to as "Twice" Block-wise Factorization algorithm. Basically, this additional improved strategy can be viewed as applying the previous idea of block-wise updating twice. Having received 4 columns 9-12 from processor P_1 (see Figure 7.3), processor P_2 will compute the 16 factorized terms (exactly the same ways as discussed earlier).

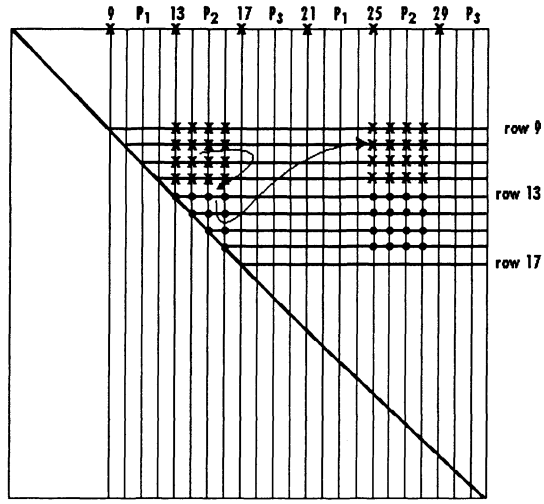


Figure 7.3 "Twice" block-wise (rows) factorization for matrix A (in a 1-D array)

$$U_{9,13} \longrightarrow U_{9,16}$$
$$\vdots \qquad\qquad \vdots$$
$$U_{12,13} \longrightarrow U_{12,16}$$

Processor P_2 then proceeds to the next 4 more rows (rows 13-16) to factorize the following terms

$$\begin{matrix} U_{13,13} & U_{13,14} & U_{13,15} & U_{13,16} \\ & U_{14,14} & U_{14,15} & U_{14,16} \\ & & U_{15,15} & U_{15,16} \\ & & & U_{16,16} \end{matrix}$$

(and send the completely factorized columns 13-16 to all other processors) before returning back to rows 9-12 to factorize its remaining terms, such as

$$U_{9,25} \longrightarrow U_{9,28}$$
$$\vdots \qquad\qquad \vdots$$
$$U_{12,25} \longrightarrow U_{12,28}$$

This enhancement will clearly help other processors to have less additional idle time. This so-called "Twice" Block-wise factorizing algorithm can be summarized in Figure 7.4

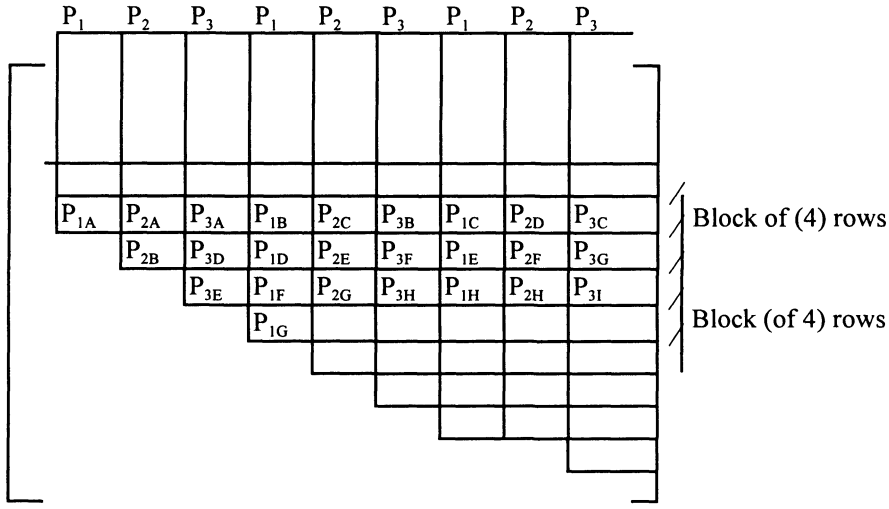


Figure 7.4 Flows of "twice" block-wise (rows) factorization algorithm

In Figure 7.4, the first subscript of P represents the processor number, and the second subscript of P represents the computation order. For example P_{2E} represents 16 factorized terms computed by processors P_2, will be followed by P_{2F} which represents the next 16 terms to be factorized by processor P_2. The convention we adopt in Fig. 7.4 is A is computed first, then B is computed next, then C, then D, then E etc...

To further improve the vector performance of the equation solver, we use a library subroutine DDOT as the kernel for dot product operations. Table 7.2 shows the vector performance for different options.

Table 7.1 A skeleton pseudo-code for parallel-vector Choleski factorization

```
1       Loop = 4
2   →   Parallel DO 100 i^th row = 1, n, 4          (Say i = 9)
3   C... me = processor number; NP = Number of Processors
4       If ("me" have the i^th column) then
5   →   DO 200 j = 0, 3          (this loop is required due to skipping occurs in
                                  the first loop)
6   II = (Global row #) = i + j
7   →   DO 400 column # JJ = II, n, NP*Loop  (The values of JJ maybe skipped,
                                              depending on the value of "me")
8       SUM1 = SUM2 = SUM3 = SUM4 = 0
9   →DO 500 row k = 1,2,. . . .II-1
10      SUM1 = SUM1 + U_k,II * U_k,jj
11      SUM2 = SUM2 + U_k,II * U_k,jj+1
12      SUM3 = SUM3 + U_k,II * U_k,jj+2
13      SUM4 = SUM4 + U_k,II * U_k,jj+3
14  500 Continue

15  If (II.Eq.JJ) then
16  u_ii,jj = (A_ii,jj - SUM1)^½
17  send column jj to all other processors

18  U_ii,jj+1 = (A_ii,jj+1 - SUM2) / U_ii,ii ;  U_ii,jj+2 = (A_ii,jj+2 - SUM3) / U_ii,ii ;  U_ii,jj+3 = (A_ii,jj+3 - SUM4) / U_ii,jj

19  else
20  U_ii,jj = (A_ii,jj - SUM1) / U_ii,ii ; U_ii,jj+1 = (A_ii,jj+1 - SUM2) / U_ii ; U_ii,jj+2 = (A_ii,jj+2 - SUM3) / U_ii ; U_ii,jj+3 = (A_ii,jj+3 - SUM4) / U_ii
21  Endif
22  400 Continue
23  200 Continue
24      Else
25  C... all other processors do the following
26      • receive column # II (from processor "me")
27      • update the appropriate, processor's terms of row ii
28          End if
29  100 continue
```

Additional explanations about Table 7.1 will be given in the following paragraphs

Line 1: Assuming unrolling level 4 is used, thus, every 4 columns are grouped together

Line 2: The increment of 4 for this do-loop is required, since every 4 columns are grouped together

Line 4: The processor (say processor "me") which owns the i^{th} column will

execute statements in lines 5 through 23. All other processors will execute statements in lines 25 through 27

Line 5: This loop is required in order to compensate the increment 4, used for the index i (on line 2)

Line 6: Global row number II is defined

Line 7: This do-loop (with the index JJ) is required to cover all terms in row II. Special care need be done to have appropriate starting point, ending point and increment value for the index JJ.
Assuming II = 9 (thus row 9 is being factorized), then JJ = (9, 10, 11, 12), (21, 22, 23, 24) etc... for processor P_1, JJ = (13, 14, 15, 16), (25, 26, 27, 28) etc... for processor P_2 and JJ = (17, 18, 19, 20), (29, 30, 31, 32) etc... for P_3
However, if II = 10 (or row 10 is being factorized), then JJ = (10, 11, 12), (21, 22, 23, 24) etc... for P_1
Thus, in actual computer coding, the "real" formulas (or algorithm) for the index JJ is more complicated than the "pseudo-formula" given on line 7 for the index JJ.

Lines 8-14: Four dot-product operations are required (for unrolling level 4), such as
column II • column JJ
column II • column (JJ+1)
column II • column (JJ+2)
column II • column (JJ +3)

Lines 15-18: If column #JJ has the same value as row # II, then we know the diagonal term (and its adjacent 3 off - diagonal terms) are being factorized (for example: $U_{9,9}$, $U_{9,10}$, $U_{9,11}$ and $U_{9,12}$ are being factorized).

Lines 19-21: If column #JJ has different value with row #II, then we know we are dealing with all off-diagonal terms of row II (for example: $U_{9,21}$, $U_{9,22}$, $U_{9,23}$, and $U_{9,24}$)

Lines 22-26: self -explained !

Line 27: All other processors (except processor "me") will factorized their appropriate (off-diagonal) terms of row #II. For example
Processor 2 will factorize ($U_{9,13}$...$U_{9,16}$), ($U_{9,25}$...$U_{9,28}$) etc...
Processor 3 will factorize ($U_{9,17}$...$U_{9,20}$), ($U_{9,29}$...$U_{9,32}$) etc...

Lines 28-29: self explained

Table 7.2 Vector performance with different options

Options	Time (seconds)
No vector-unrolling	44.8
Vector-unrolling level 4	35.8
Vector-unrolling + block-wise updating	22.2
DDOT + vector-unrolling + block-wise updating	20.2
DDOT + vector-unrolling without block-wise updating	14.5

Note: Decomposition of a 1000 x 1000 matrix on one processor (Intel iPSC/860)

Table 7.3 Communication schemes

	Single-send scheme		Double-send scheme
		1	DO 100 i = 1, n,4
1	DO 100 i = 1, n, 4	2	IF ("me" have the i^{th} column) THEN
2	If ("me" have the i^{th} column)	3	DO 200 j = 0,3
3	DO 200 j = 0, 3	4	update the $(i+j)^{th}$ column
4	update the $(i + j)^{th}$ column	5.1	send the $(i + j)^{th}$ column to the
5	send the $(i + j)^{th}$ column to		next processor
	all other processors (fan-out)	5.2	send the $(i + j)^{th}$ column to all
6	update portions of $(i + j)^{th}$ row		other processors
7	200 continue	6	update portions of $(i + j)^{th}$ row
8	ELSE	7	200 continue
9	DO 300 j = 0, 3	8	ELSE
10	receive the $(i + j)^{th}$ column	9	DO 300 j = 0,3
11	update portions of $(i + j)^{th}$ row	10	receive the $(i + j)^{th}$ column
12	300 continue	11	update portions of $(i + j)^{th}$ row
13	ENDIF	12	300 continue
14	100 continue	13	ENDIF
		14	100 continue

Table 7.4 Communication performance

Options	Time (seconds)
Single -send (with CSEND, CRECV)	231
Single-send (with ISEND, IRECV)	192
Single-send (DDOT + ISEND, IRECV)	120
Double-send (DDOT + ISEND, IRECV)	108
Sequential-send (DDOT + CSEND, CRECV)	104

Note: Decomposition of a 4000 x 4000 matrix on 16 processors (Intel iPSC/860)

Table 7.5 Sequential send (or RING) scheme

```
1   DO 100 i = 1, n, 4
2   IF ("ME" have the i^th column) Then
3   DO 200 J = 0, 3
4   { • Update the (i+j)th column
5   { • send (i+j)th column to next processor
6   { • update portions of (i+j)th row
7   200 continue
8   ELSE
9   DO 300 J = 0, 3
10  *Every processor receives info. from previous processor
11  * If (ME ≠ processor which owns column # [i - 1]) Then
12  • send info. to next processor which owns column # (i + 4)
        • with exception : last processor will send information to processor 0
13  Endif
14  * update portions of (i + j)^th row
15  300 continue
16      Endif
17  100 continue
```

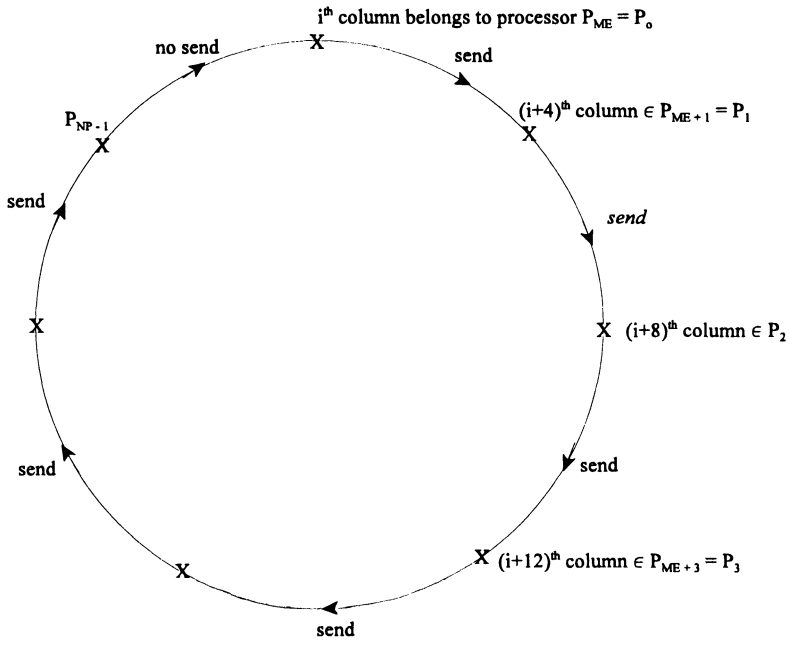


Figure 7.5 Sequential (or ring) sending message

7.2.3 Communication performance-improvement in factorization

On the Intel iPSC/860 hypercube, the maximum communication rate (massage-passing rate) between two nodes is only 0.35 MWORDS (in practice, it is only 0.1 MWORDS), which is much slower than its maximum arithmetic operation rate 60 MFLOPS (in practice, it is only 25 MFLOPS). The communication rate is even slower when more nodes are involved.

In general, a skyline (column storage scheme) equation solver on distributed memory computers needs to transfer the current updated column (say, stored in node i) to all other nodes, which is called fan-out. This fan-out becomes slower if the number of nodes increases. As these communication routines (for fan-out) are designed by the computer manufacturer, it is not practical for the users to make any changes in them. However, we have tried to overcome this difficulty by introducing the so-called double-send scheme. Table 7.3 shows both single fan-out scheme and the double-send scheme for factorization. Another simple scheme is called sequential-send (please refer to Table 7.5) which means each node (say, node i) only receive information from its preceding node (node i-1) and only send information to the next node (node i+1). Both double-send and sequential-send schemes are based on the idea that the closest node should get the message first. Table 7.4 presents communication performance with different message-passing techniques.

In Table 7.4, the sending and receiving messages (ISEND, IRECV) is better than (CSEND, CRECV). In simple language, CSEND can be interpreted as a person who drops (or sends) the mail in the mail box, he then waits in the mail box for a little while (say until the mailman arrived at the mail box), then he leaves the mailbox. On the other hand, ISEND can be interpreted as a person who drops (or sends) the mail in the mail box, and he immediately leaves the mail box. Depends on the applications and the problems at hands, proper use of the appropriated communication messages need to be observed.

Single send and double send schemes are self-explained in Table 7.3. Sequential (or ring) send scheme shown in Table 7.5, however, needs further explanations for better understanding.

(a) Lines 1 -8 (of Table 7.5) are essentially the same as lines 1 - 8 (of Table 7.3, double-send scheme), but with lines 5.2 deleted (from Table 7.3, double-send scheme)

(b) Lines 9, 10, 14-17 of Table 7.5 play the same role as lines 9-14 of Table 7.3.

(c) The IF statement provided on line 11 of Table 7.5 will make sure that processor "ME" (which owns the i^{th} column) will <u>not</u> receive its own message (from its previous neighbor processor) which it has sent (in a ring fashion) to all other processors, as clearly explained in Figure 7.5

(d) Line 12 of Table 7.5 will make sure that a processor will send the message (which it receives from the previous neighbor processor) to the next neighbor processor. For example, P_1 sends message (which it receives from P_o) to processor P_2. Processor P_2 sends message to P_3, P_3 sends message to P_4. Assuming P_4 is the last processor in this process, P_4 will <u>NOT</u> send the message to P_O (since P_0 is assumed to be the original processor which supposes to be a starting processor to send the message to all other processors!)

7.2.4 Forward/backward elimination

In linear static applications, factorization is the most time consuming portion of a finite element analysis. In many other applications (such as nonlinear static/dynamic, eigenvalue, design sensitivity analysis and optimization), however, the forward/backward elimination has to be done repeatedly. Thus, forward/backward solution time becomes quite important for the above applications.

A. Forward Elimination

Assuming the first 8 unknowns of the solution vector y in the forward elimination phase have already been calculated, and the forward solution for y_9 through y_{12} are sought. For simplicity, assuming the factorized lower triangular matrix U^T is full as shown in Figure 7.6. According to Eq. 7.6, one has:

$$y_9 = \frac{b_9 - \sum_{i=1}^{8} u_{i,9} y_i}{U_{9,9}} \tag{7.9}$$

or

$$y_9 = \frac{b_9 - (u_{1,9} y_1 + u_{2,9} y_2 + \ldots + u_{8,9} y_8)}{u_{9,9}} \tag{7.10}$$

Thus, one can clearly see that processor 1 (see Figure 7.6) can easily calculate unknowns y_9 through y_{12} since y_1 through y_8 have already been "completely" calculated. As soon as processor 1 finishes computing y_9 through y_{12}, it will broadcast these complete solutions (y_9 through y_{12}) to all other processors (fan out). This fan out process is illustrated by columns 9 → 12 below the diagonals as shown in Figure 7.6. Having broadcasted the solutions $y_9 \rightarrow y_{12}$ to all other processors, processor 1 continues to compute the "partially" updated solutions for portions of the remaining unknown solution vector $\vec{y}$.

For example, processor 1 (or P_1) will compute:

$$y_{21}(\textit{incomplete or partially complete}) = \frac{b_{21} - (U_{21,1} y_1 + \ldots + U_{21,12} y_{12}) - (unknown1)}{U_{21,21}} \tag{7.11}$$

Since the factorized matrix has been stored in the upper triangular portion, Eq (7.11) can be re-written as

$$y_{21}(incomplete) = \frac{b_{21} - (U_{1,21} y_1 + \ldots + U_{12,21} y_{12}) - (unknown1)}{U_{21,21}} \tag{7.12}$$

Similar expressions can also be written for y_{22} (incomplete), y_{23} (incomplete) and y_{24} (incomplete)

In Eq (7.11), we define

$$unknown1 = U_{13,21}\, y_{13} + \ldots\ldots + U_{20,21}\, y_{20} \tag{7.13}$$

At the same time, processor 3 (or P_3) will compute:

$$y_{17}\ (incomplete) = \frac{b_{17} - (U_{1,17} y_1 + \ldots\ldots + u_{12,17}) - (\ldots)}{U_{17,17}} \tag{7.14}$$

and will also compute:
y_{18} (incomplete) through y_{20} (incomplete), and y_{29} (incomplete) through y_{32} (incomplete).

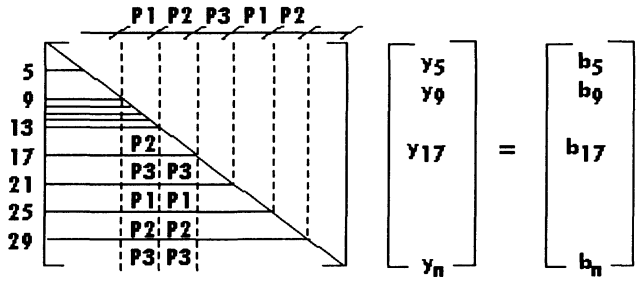


Figure 7.6 Parallel forward elimination

B. Backward Elimination:

Assuming the last 8 unknowns of the solution vector x in the backward elimination phase have already been calculated, and the backward solution for x_{n-8}, x_{n-9}, x_{n-10} and x_{n-11} are sought. To simplify the discussion, assuming n = 100 and the factorized upper triangular matrix U is full as shown in Figure 7.7. According to Eq. (7.7), one has:

$$x_{92} = \frac{y_{92} - \sum_{i=93}^{n=100} u_{92,i}\, x_i}{U_{92,92}} \tag{7.15}$$

$$x_{92} = \frac{y_{92} - (u_{92,93} x_{93} + u_{92,94} x_{94} + \ldots + u_{92,100} x_{100})}{U_{92,92}} \tag{7.16}$$

Thus, one can clearly see that processor 1 (see Figure 7.7) can easily calculate unknowns x_{92} through x_{89}, since x_{100} through x_{93} have already been "completely" calculated.

Having completed the final solution for x_{92} through x_{89}, processor 1 continues to compute the "partial" (or incomplete) solution for x_{88} through x_{85}. Processor 1 then send these partial solutions to the next processor (on its left neighbor, say processor 3) and processor 1 continues to find the partial solution for x_{84} through x_1.
For example:

$$x_i \text{ ("partial" solution)} = \frac{y_i - (known\ portion) - (unknown\ portion)}{U_{i,i}} \tag{7.17}$$

In Eq. (7.17), i = 84→1, and j = 88, the known and unknown portions are defined as

$$\text{known portion} = (u_{i,j+1}\, x_{j+1} + \ldots\ldots + u_{i,n}\, x_n) \quad \textbf{(7.18)}$$

$$\text{unknown portion} = (u_{i,\,i+1}\, x_{i+1} + u_{i,\,i+2}\, x_{i+2} + \ldots\ldots + u_{i,j}\, x_j) \quad \textbf{(7.19)}$$

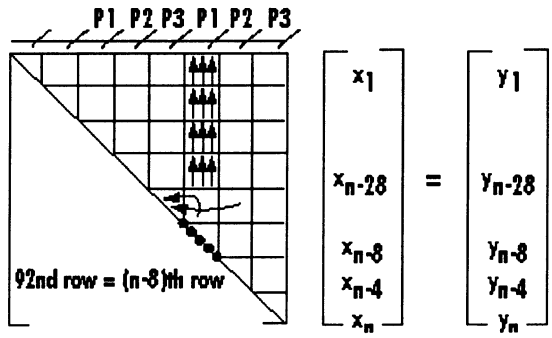


Figure 7.7 Backward elimination

Meanwhile, a "if check" is performed in order to determine the workloads for the remaining processor (not including processor 1). If a processor is adjacent to the left of processor 1 (say processor 3), it will receive the "partial" solutions (for example, x_{88} through x_{85}) from all other processors (fan in), say processors 1 and 2. All the other processors (not including processor 1 and the adjacent processor 3) will send the required information to processor 3. Therefore, processor 3 is now ready to compute the "final" solution for x_{88} through x_{85}.

The above backward solution strategies can also be conveniently cast in the following step-by-step procedure (also refer to Figure 7.8).

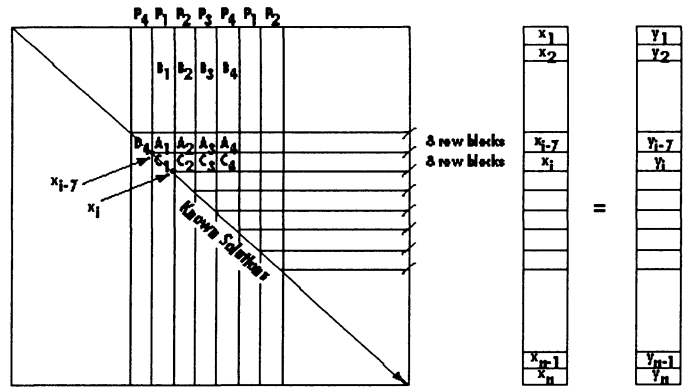


Figure 7.8 Backward solution

Processor P_1's tasks:

Step 1 : Solve for unknowns x_i, x_{i-1} .... x_{i-7} (refer to the triangle region C_1 in Fig. 7.8)

Step 2 : P_1 uses 8 x 8 block A_1 (see Fig. 7.8) together with the corresponding (known) portions of the $\vec{x}$ vector to compute the so called "known portion" as indicated in Eq. (7.18)

Step 3 : P_1 sends the "known portion" to its adjacent (left) neighbor processor P_4

Step 4 : P_1 continue to use (bigger) block B_1 (see Fig. 7.8) together with the corresponding (known) portions of the $\vec{x}$ vector to compute the so called "known portion" as indicated in Eq. (7.18)

The adjacent (left) neighbor processor P_4's tasks:

Step 0: P_4 receives "known portions" from all other processors

Step 1: solve for unknowns x_{i-8}, x_{i-9}, ..., x_{i-15} (refer to triangular region D_4 in Fig. 7.8). Then P_4 will perform tasks similar to steps 2-4 of processor P_1

Tasks to be done by each of the remaining processors (say P_2, P_3...)

Step 0: P_2 (and also P_3 ...) will send its own "known portion" to processor P_4

Step 1: Waiting for its turn to compute subsequent unknowns. Then P_2 (and also P_3...) will perform tasks similar to steps 2-4 of processor P_1

The above process is repeated until the final solution for x_1 is found. It should also be noted here that each processor also holds a vector x of length n. At the end of the backward elimination process, each processor only has its own portion of the solution in a vector x. The system subroutine GDSUM is then used to effectively merge each individual processor's solution to obtain the final global solution (as shown in Figure 7.9).

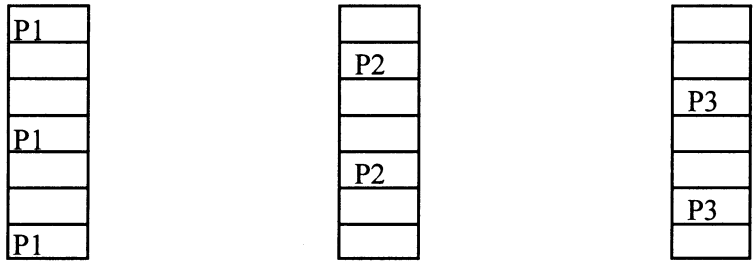


Figure 7.9 GDSUM is used to merge partial (processor) solution for final solution

The arithmetic operations in the forward elimination and in the backward elimination phases are essentially the same. In the forward elimination phase, the calculations are performed in parallel and the massages are passed by "fan-out". However, in the backward elimination phase, the calculations are mostly done in sequential and the communications are completed by "fan-in". Table 7.6 gives key ideas of the algorithm for forward/backward elimination.

Table 7.6 Algorithms for forward/backward elimination

Forward elimination	Backward elimination

DO 100 i = 1, n, 4
IF ("me" have the i^{th} row) THEN
 	
C ... For 1 processor
update y(i),y(i+1),y(i+2),y(i+3)
fan-out (or send to all)
y(i),y(i+1),y(i+2),y(i+3)
partially update y(j)(for i+3 < j < n)
for processor "me"' portion only
 	
ELSE
C ... For all other processors
receive y(i),y(i+1),y(i+2),y(i+3)
partially update y(j) (for i+3 < j < n)
 	
ENDIF
 	
100 continue
	DO 200 i = n, 1, - 4
	IF ("me" have the i^{th} column) THEN
	C ... For 1 processor
	update x(i),x(i-1),x(i-2),x(i-3)
	send partially updated x(i-4),x(i-5),x(i-6)
	x(i-7) to the next processor
	partially update x(j) (for 1 < j < i-7)
	ELSE
	C ... For 1 processor (the adjacent processor to "me")
	if ("me" have the $(i-4)^{th}$ column) then fan-in (or receive) x(i-4),x(i-5),x(i-6),x(i-7)
	else
	C ... For all other processors
	send information correspond to row (i-4), (i-5), (i-6) and (i-7) to processor which contains the $(i-4)^{th}$ column
	ENDIF
	ENDIF
	200 continue

7.3 Numerical Results and Discussions

Several numerical examples are run on the Intel iPSC/860 hypercube and on the MEIKO (Ref 7.11) parallel computers with the presented equation solver. Some results

are shown in Tables 7.7 through 7.10.

Table 7.7 Timings for solving 1000x1000 equations on the Intel iPSC/860 (Lagrange machine)

Nodes	1	2	4	8	16	32	64	128
Deco.	14.45	8.117	5.403	3.800	2.962	2.522	2.321	2.26
Forw.	0.231	0.156	0.123	.0870	.1086	.1789	.3243	.640
Back.	0.122	.0823	.0569	.0545	.0539	.0651	.0829	.108
Total	14.80	8.355	5.583	3.942	3.125	2.766	2.718	3.01

Note: Sequential-send scheme with vector-unrolling level 8.

Table 7.8 Comparison of equation solvers (n = 16152, nbw = 328, using 32 nodes on Intel iPSC/860 Gamma)

Intel Pro-solver (SES)	The present solver*
Decomposition: = 51.18 (sec)	Decomposition: = 25.85 (sec)
Forward: = 9 (sec)	Forward: = 0.8156 (sec)
Backward: = 62 (sec)	Backward: = 0.9401 (sec)
Total: = 122.18 (sec)	Total: =27.607 (sec)

* Sequential-send scheme with vector-unrolling level 8.

Two real structural problems are also solved by the proposed solver in order to evaluate its performance. The first one is a hinged-cylinder model with 1808 degrees of freedom (or n = 1808), average half bandwidth nbw = 200, and maximum half-bandwidth maxbw = 300. It should be noted here that for this problem, the block-skyline storage scheme only needs 254644 words of memory, which is about 70% of that required by a row storage scheme. The second one is an aircraft structure [7.5, 7.6] with n = 16148, nbw = 324, and maxbw=604. Numerical results for these 2 practical models are shown in Tables 7.9-7.10. Table 7.10 gives the total timing for solving the problem on 8, 16 and 32 nodes.

Table 7.9 Hinged-cylinder structure

When only one node is used, the required memory is 254644 < 1808*200.

Node(s)	1	2	4	8
Total Time (sec.)	3.225	2.482	2.806	3.045

Note: Double-send scheme with vector-unrolling level 4.

Table 7.10 Gamma computer timing (sec) for aircraft structures

Task	nodes			
	8	16	32	32*
Decomposition	35.9	30.7	28.7	26.0
Forward elimination	1.6	1.4	1.4	0.8
Backward elimination	1.4	1.5	1.7	1.0
Total	38.9	33.6	31.8	27.8

Note: Sequential send scheme with vector-unrolling level 4(* = level 8).

A 2-D Truss Structure with Multiple Bays and Stories: In this example, a 750 bay x 6 story (and a 1096 bay x 41 story) truss structure is shown in Figures 7.10. A horizontal force F is applied at node 100. The former (750 bay x 6 story) has 18006 elements. The resulted structural stiffness matrix has 9016 degree-of-freedom (or equations). The average bandwidth for this stiffness matrix is 1512.

The latter (1096 bay x 41 story) has 179785 elements. The resulted structural stiffness matrix has 89960 degree-of-freedom (or equations). The average bandwidth for this stiffness matrix is 2208.

In both the former and the latter, both the Intel Gamma Parallel Computer (with 128 processors) and the Intel Delta Parallel Computer (with 512 processors) were used to solve the resulted systems of simultaneous equations. It should be mentioned here that the Delta parallel computer has more processors as well as more memory per processor than the Gamma Computer. Due to the relatively large-size problems, at least 16 processors and 8 processors need to be used (for the 750 bay x 6 story structure) by the Gamma and Delta computer, respectively.

At least 256 processors need to be used (for the 1096 bay x 41 story) by the Delta computer.

The parallel-vector performance for the 750 bay x 6 story and the 1096 bay x 41 story are given in Tables 7.11 and 7.12, respectively.

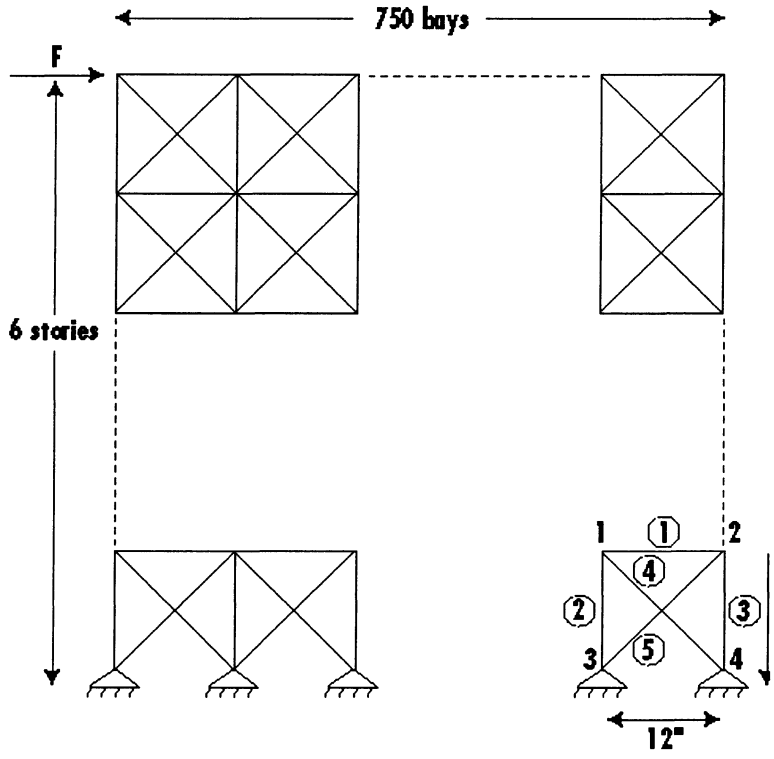


Figure 7.10 2-D truss structure

Table 7.11 Comparison of **MPFEA** finite element code on the Gamma and Delta computers (750 bays, 6 stories, 18006 els., NEQ = 9016, Ave. BW = 1512)

	8	16	32	64	128	256	512
Gamma fac.	-	80.20	61.383	57.018	48.081	-	-
forward	-	0.8228	0.6241	0.7751	1.2786	-	-
backward	-	0.5411	0.5405	0.6015	0.7090	-	-
Delta fac.	136.05	75.69	54.85	46.89	40.85	39.04	39.40
forward	1.265	0.795	0.603	0.735	0.596	0.552	0.575
backward	0.616	0.428	0.356	0.468	0.399	0.478	0.589

Table 7.12 Parallel Performance of MPFEA on 256 processors
(1096 bays, 41 stories, 179785 els, NEQ=89960, Ave. BW=2208)

Task(s)	CPU time (seconds)	Mflops
factorization	662.417	631.56
Forward elimination	4.9654	77.94
Backward elimination	3.1579	120.16
Others (overhead, etc.)	3.4624	-
Total	674.13	621.73

From the examples considered in the above sections, one can see that the vector performance is more than doubled through the use of vector-unrolling and the DDOT system subroutine. The communication performance is also improved by using the so-called double-send and sequential-send schemes. The sequential-send scheme is better than the double-send scheme when the number of processors increases. Further improvement in the performance can be expected if the users are allowed to make changes in the communication subroutines.

7.4 FORTRAN Call Statement to Subroutine Node

Based upon the parallel-vector algorithms discussed in the previous sections, a parallel Fortran subroutine "Node" has been written for the Intel type computers (massively parallel, distributed computers). The call statement to subroutine node, the meaning of various arrays (or arguments) in the call statement, the dimension requirements for each array and how to obtain various arguments in the subroutine are explained in the following sections. All real arrays are declared in double precision.

Subroutine node (nodes, iam, n, nbw, imod, a, z, y, x, maxa, irow, icolg, tem, kflag)

nodes = number of processors
iam = my node id#
n = degree-of-freedom
nbw = maximum bandwidth (include diagonal)
imod = 1 (for real problem)
a = stiffness matrix (dimension = nterms)
z = working array, z (nbw, 8)
y = load vector, y (n)
x = displacement vector, x (n)
maxa = diagonal locations, dimension $\geq \{[(\frac{n-1}{8}+1)/Nodes]+2\}*8$
*irow = i^{th} row length(include diagonal), dimension $\geq (\frac{n-1}{8}+1)$

*icolg = column height, dimension ≥ $(\frac{n-1}{8} + 1)$

*Notes: (a) All processors need to have these information

(b) Only the information of row-lengths (and column heights) of the last row (and last column) of each block is needed

tem = real array, dimension ≥ nbw

kflag = $\begin{cases} 1, \textit{ for factorization} \\ \textit{ELSE, for forward/backward} \end{cases}$

The above subroutine arguments can be better understood by referring to Figure 7.11 and the following comments

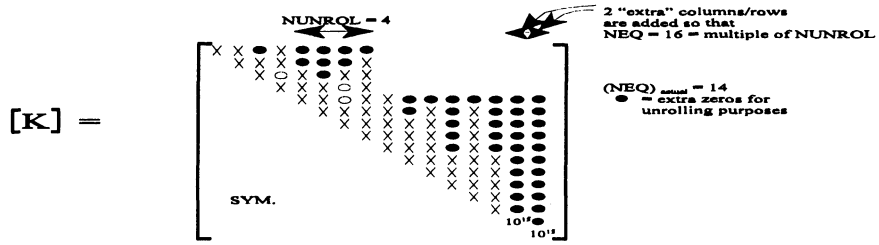


Figure 7.11 Massively distributed storage scheme for equation solver

Comments on Figure 7.11: In Figure 7.11, the following data is assumed

$\begin{cases} (NEQ)\ actual = 14 \\ NUNROL = 4 \\ NP\ (= No.\ of\ Processors) = 2 = (processor\ 0\ and\ 1) \\ Thus:\ NEQ = 16\ (= multiple\ of\ NUNROL) \end{cases}$

(a) column height : from diagonal upward (include diagonal term)

(b) row-length : include diagonal term.

(c) Since NUNROL = 4 is used in the equation (Intel) solver, each block (of 4) columns must have same level high➡ Extra ZEROES (see symbols ● as shown in Figure 7.11) need be added

(d) The last column height in each block (of 4) must be a multiple of NUNROL and must be ≥ NUNROL

(e) For the above example of [K], max NP = 4 (processors 0, 1, 2, 3), If NP > 4, then we'll have idle processors

(f) We need GLOBAL column height ➡ Icolh $\begin{pmatrix} 1 \\ 2 \\ \cdot \\ \cdot \\ \left[\frac{NEQ-1}{NUNROL}\right]+1 \end{pmatrix} = Icolh \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{Bmatrix} 4 \\ 8 \\ 8 \\ 12 \end{Bmatrix}$

where only global column height of the LAST column in each block (of 4 columns) need be calculated

(g) We also need GLOBAL row-length information

$$IROWL\begin{pmatrix} 1 \\ 2 \\ \vdots \\ \left[\frac{NEQ-1}{NUNROL}\right]+1 \end{pmatrix} = IROWL\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{Bmatrix} 5 \\ 9 \\ 5 \\ 1 \end{Bmatrix}$$

where only Global row-length of LAST row in each block (of 4 rows) need be calculated.

(h) We also need LOCAL (for each Processor) MAXA information

$$MAXA\begin{pmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ \frac{NEQ}{NP} \end{pmatrix} = MAXA\begin{Bmatrix} 1 \to 1 \\ 2 \to 2 \\ 3 \to 4 \\ 4 \to 7 \\ 5 \to 11 \\ 6 \to 16 \\ 7 \to 22 \\ 8 \to 29 \end{Bmatrix} \text{ For Processor 0}$$

$$\text{and } MAXA\begin{Bmatrix} 1 \to 1 \\ 2 \to 6 \\ 3 \to 12 \\ 4 \to 19 \\ 5 \to 27 \\ 6 \to 36 \\ 7 \to 46 \\ 8 \to 57 \end{Bmatrix} \text{ for processor 1}$$

For the complete listing of the FORTRAN source codes, instructions in how to incorporate this equation solver package into any existing application software (on any specific computer platform), and/or the complete consulting service in conjunction with this equation solver etc... the readers should contact:

Prof. Duc T. Nguyen
Director, Multidisciplinary Parallel-Vector Computation Center
Civil and Environmental Engineering Department
Old Dominion University
Room 135, Kaufman Building
Norfolk, VA 23529 (USA)
Tel = (757) 683-3761, Fax = (757) 683-5354
Email = dnguyen@odu.edu

7.5 Summary

The parallel-vector Choleski equation solver on distributed memory computers has been described and tested on several applications. Parallel, vector and communication strategies have also been discussed. Incore memory requirements and the number of

operations are small since the skyline storage scheme is used. Furthermore, the entire coefficient stiffness matrix is not resided in 1 processor. Rather, the entire coefficient matrix is splited and distributed across all processors. Thus, as the number of processors is increased, the incore memory requirement for each processor is decreased nearly in proportion with the number of processors used. The vector speed is enhanced by the vector unrolling technique. The parallel-vector performance of the proposed solver on several applications seems to be quite good. Furthermore, it should be emphasized here that the forward and backward solution of the present solver are quite effective, and therefore this solver can be nicely incorporated into other applications, such as eigen-values analysis, nonlinear analysis and structural optimization.

7.6 Exercises

7.1 Using the symmetrical coefficient (stiffness) matrix data shown in Figure 5.27 of Chapter 5, assuming 3 processors are available (P_0, P_1 and P_2) and using unroll level 4:
   (a) Find global column height (ICOLH) information???
   (b) Find global row - length (IROWL) information??
   (c) Find local diagonal locations (MAXA) information
   Hints: Read Section 7.4 of Chapter 7

7.2 In Fig. 7.1, one realizes that even though column-by-column fashion is used to store the coefficient matrix, but row-by-row fashion is used to factorize the matrix. What will be the problem(s) if we adopt the column-by-column factorization strategies???

7.3 In the algorithm presented in Table 7.1, only the "key ideas" behind the index JJ (of loop 400) are given and explained. Explain in greater details how the index JJ changes??

7.4 Given the size of the coefficient (stiffness) matrix is N (say N = 10,000 equations), the flop rate is FR (say FR = 10 MFLOPS per processor), the number of processors is NP (say NP =4), and the communication rate is CR (say CR = 0.1 Million Words per second). Using the Choleski factorization algorithm, and assuming the coefficient (stiffness) matrix is symmetrical and full.
   (a) Estimate the "purely" computational time (assuming no communication)
   (b) Estimate the "purely" communication time (assuming communication time will not overlap with computational time).

7.7 References

7.1 Ortega, J.M. andVoigt, R.G. "Solution of Partial Differential Equations on Vector and Parallel Computers" SIAM Review, Vol. 27, No. 2, 1985, pp. 149-240.

7.2 Qin, J., Gray, Jr., C.E., Mei, C. and Nguyen D.T., "A parallel-vector equation solver for unsymmetric matrices on supercomputers," Computing Systems in Engineering, Vol. 2, No. 2/3, 1991, pp. 197-201.

7.3 Henk A. van der Vorst, H.A., "Large tridiagonal and block tridiagonal linear systems on vector and parallel computers," Parallel Computing, Vol. 5, 1987, pp. 45-54.

7.4 Dongarra, J.J. and Johnson, L., "Solving banded systems on a parallel processor," Parallel Computing, Vol. 5, 1987, pp. 219-246.

7.5 Agarwal, T.K., Storaasli, O.O., and Nguyen, D.T., "A Parallel-Vector Algorithm For Rapid Structural Analysis on High-Performance Computers," Proceedings of the AIAA/ ASME/ ASCE/ AHS 31st

SDM Conference, Long Beach, CA, AIAA Paper No. 90-1149 (April 2-4, 1990).

7.6 Storaasli, O.O., Nguyen, D.T. and Agarwal, T.K., "The Parallel Solution of Large-Scale Structural Analysis Problems on Supercomputers," AIAA Journal, Vol. 28, No. 7, pp. 1211-1216 (July 1990).

7.7 Krechel, A., Plum, H.J. and Stuben, K., "Parallelization and vectorization aspects of the solution of tridiagonal linear systems," Parallel Computing, Vol. 14, 1990, pp. 31-49.

7.8 Heath, M.T., Romine, C.H., "Parallel solution of triangular systems on distributed-memory multiprocessors," SIAM J. Sci. Statist. Comput., Vol. 9, No. 3, 1988, pp. 558-588.

7.9 Hajj, I.N. and Skelboe, S., "A multilevel parallel solver for block tridiagonal and banded linear systems," Parallel Computing, Vol. 15, 1990, pp. 21-45.

7.10 Qin, J. and Nguyen, D.T., "A Parallel-Vector Equation Solver for Distributed Memory Computers," Proceedings of the 2nd Parallel Computational Methods for Large-Scale Structural Analysis and Design, sponsored by NASA LaRC, Marriott Hotel, Norfolk, VA (Feb. 24-25, 1993). Also, to appear in Computing Systems in Engineering.

7.11 Maker, B.N., Qin, J. and Nguyen, D.T., "Performance of NIKE3D with PVSOLVE on Vector and Parallel Computers," Computing Systems in Engineering Journal (1995).

8 Parallel-Vector Unsymmetrical Equation Solver

8.1 Introduction

Unsymmetric matrices are not uncommon in large-scale structural analysis. In panel flutter analysis, for example, one has to deal with unsymmetric equations due to the appearance of the unsymmetric aerodynamic influence matrix. When large deflections and unsteady third-order piston theory aerodynamics are considered in the flutter analysis, it is necessary to solve the unsymmetric equations incrementally and/or to solve the unsymmetric generalized eigen-problems interactively. Thus, an efficient and accurate unsymmetric equation solver plays an important role in structural analysis.

In this chapter, an efficient and accurate equation solver for unsymmetric matrices is presented. The proposed method exploits both parallel and vector capabilities provided by modern, high-performance supercomputers, such as the CRAY 2 and CRAY Y-MP. With minor changes in the computer coding, the proposed algorithms can also be implemented on distributed computers, such as the Intel Paragon, the IBM-SP2 multi-processor computers.

In order to optimize the vector performance, a special storage scheme is used to store the coefficient matrix A so that the loop-unrolling technique can be applied in most of the calculations. A parallel FORTRAN language[8.1] is adopted to develop a parallel code in a multiple processing computer environment, such as the CRAY 2, CRAY-J918, CRAY Y-MP and CRAY-C90.

8.2 Parallel-Vector Unsymmetrical Equation Solution Algorithms

8.2.1 Basic unsymmetric equation solver

Systems of unsymmetric linear equations can be represented as

$$A x = b \tag{8.1}$$

One way to solve Eq. 8.1 is first to decompose A into the product of two triangular matrices

$$A = L U \tag{8.2}$$

where U is an upper-triangular matrix and L is a lower-triangular matrix with unit diagonal elements. For r from 1 to n, the r^{th} row elements of U and the r^{th} column elements of L can be obtained by the following formulas:

$$U(r,i) = A(r,i) - \sum_{k=1}^{r-1} L(r,k) * U(k,i) \quad (i = r, \ldots, n) \tag{8.3}$$

$$L(i,r) = \frac{\left(A(i,r) - \sum_{k=1}^{r-1} L(i,k) * U(k,r) \right)}{U(r,r)} \quad (i = r+1, \ldots, n) \tag{8.4}$$

Then the unknown vector x is determined by the forward/backward elimination, for example, to solve

$$Ly = b \tag{8.5}$$

for y, with

$$y(i) = b(i) - \sum_{k=1}^{i-1} L(i,k) * y(k) \quad (i = 1, \ldots, n) \tag{8.6}$$

and to solve

$$Ux = y \tag{8.7}$$

for x, with

$$x(i) = \frac{\left(y(i) - \sum_{k=i+1}^{n} U(i,k) * x(k) \right)}{U(i,i)} \quad (i = n, \ldots, 1) \tag{8.8}$$

Since the number of operations involved in the decomposition (or factorization) is usually much more than that in the forward/backward elimination, emphasis will be placed on developing an efficient parallel-vector algorithm for the factorization. In the forward/backward elimination, however, it has been concluded in Ref. [8.2] that it is more efficient to use one processor with long vectors rather than introducing synchronization overhead on multiple processors.

In practice, both L and U can be stored in the same array previously used to store A. Moreover, do-loops in Eqs. 8.3 and 8.4 need to be rearranged to adopt the loop-unrolling technique, as the decomposition procedure can be simply described as:

For $I = 1, 2, 3, \ldots, n$
- step a. Find the I^{th} row of U.
- step b. Find the I^{th} column of L.

8.2.2 Detailed derivations for the [L] and [U] matrices

In order to better understand the derived formula shown in Eqs. 8.3 and 8.4, let us try to compute the factorized [L] and [U] matrices from the following given 3 x 3 unsymmetrical matrix [A], assuming to be a full matrix to simplify the discussion.

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{8.9}$$

The above unsymmetrical matrix A can be factorized as indicated in Eq. 8.2, or in the long form

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} * \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{8.10}$$

The nine (9) unknowns, according to a special ordering u_{11}, u_{12}, u_{13}; then l_{21}, l_{31}; then u_{22}, u_{23}; then l_{32}; and finally u_{33} from Eq. 8.10 can be found by simultaneously solving the following system of equations

$$\left.\begin{aligned} a_{11} &= u_{11} \\ a_{12} &= u_{12} \\ a_{13} &= u_{13} \\ a_{21} &= l_{21}\, u_{11} \\ a_{31} &= l_{31}\, u_{11} \\ a_{22} &= l_{21}\, u_{12} + u_{22} \\ a_{23} &= l_{21}\, u_{13} + u_{23} \\ a_{32} &= l_{31}\, u_{12} + l_{32}\, u_{22} \\ a_{33} &= l_{31}\, u_{13} + l_{32}\, u_{23} + u_{33} \end{aligned}\right\} \tag{8.11}$$

Thus, from Eq. 8.11, one obtains

$$\left.\begin{aligned} u_{11} &= a_{11} \\ u_{12} &= a_{12} \\ u_{13} &= a_{13} \\ l_{21} &= \frac{a_{21}}{u_{11}} \\ l_{31} &= \frac{a_{31}}{u_{11}} \\ u_{22} &= a_{22} - l_{21}\, u_{12} \\ u_{23} &= a_{23} - l_{21}\, u_{13} \\ l_{32} &= \frac{a_{32} - \left(l_{31}\, u_{12}\right)}{u_{22}} \\ u_{33} &= a_{33} - \left(l_{31}\, u_{13} + l_{32}\, u_{23}\right) \end{aligned}\right\} \tag{8.12}$$

It can be seen clearly that the nine unknowns shown in Eq. 8.12 can also be obtained by directly using Eqs. 8.3 and 8.4.

The ordering appeared in Eq. 8.12 suggests that the factorized matrices $[L]$ and $[U]$ can be found in the following systematic pattern:

step 1: The 1st row of the upper triangular matrix $[U]$ can be solved for the solution of u_{11}, u_{12} and u_{13}

step 2: The 1st column of the lower triangular matrix $[L]$ can be solved for the solution of l_{21} and l_{31}

step 3: The 2nd row of $[U]$ can be solved for the solution of u_{22} and u_{23}

step 4: The 2nd column of $[L]$ can be solved for the solution of l_{32}

step 5: The 3rd row of $[U]$ can be solved for the solution of u_{33}

For the case $r = 8$ and $i = 9$, Eqs. 8.3 and 8.4 become:

$$\left.\begin{aligned} u_{8,9} &= a_{8,9} - \left(l_{8,1}\,u_{1,9} + l_{8,2}\,u_{2,9} + \ldots + l_{8,7}\,u_{7,9}\right) \\ l_{9,8} &= \frac{a_{9,8} - \left(l_{9,1}\,u_{1,8} + l_{9,2}\,u_{2,8} + \ldots + l_{9,7}\,u_{7,8}\right)}{u_{8,8}} \end{aligned}\right\} \qquad (8.13)$$

Observing Eq. 8.13, one can see that to factorize the term $u_{8,9}$ of the upper triangular matrix $[U]$, one only need to know the factorized row 8 of $[L]$ and column 9 of $[U]$.

Similarly, to factorize the term $l_{9,8}$ of the lower triangular matrix $[L]$, only need to know the factorized row 9 of $[L]$ and column 8 of $[U]$.

8.2.3 Basic algorithms for decomposition of "full" bandwidths/column heights unsymmetrical matrix

To exploit the vector capability provided by supercomputers[8.3], it is necessary to arrange the data appropriately. To do this, matrix A is stored in a mixed row oriented and column oriented fashion. This storage scheme allows the use of the loop-unrolling technique in both steps a and b, described in section 8.2.1. Figure 8.1 shows how the coefficient matrix A is stored in one-dimensional array. In Figure 8.1, P_1, P_2, P_3. . . represent processor numbers. The basic FORTRAN code corresponding to steps a and b can be written as shown in Table 8.1.

Table 8.1 Basic algorithm for decomposition (full matrix)

```
      for     I = 1, 2, 3, . . . , n
              Do      1       K = 1, I-1
              Scalar = a(I, K)
c     do loop2 is used to update (or factorize) the Ith row of U due to the
      contribution of the Kth row (refer to Eq.8.3)
              Do 2  J = I, n
              a(I, J) = a(I, J) - scalar * a(K, J)
            2 Continue
```

c do loop 3 is used to "partially" update the I^{th} column of L due to the contribution of the K^{th} column (refer to the nominator of Eq.8.4)

Scalar = a(k, I)

```
      Do 3  J1 = I + 1, n
      a(J1, I) = a(J1, I) - scalar * a(J1, K)
3     Continue
```

1 Continue

c do loop 4 is used to compute the "final" update of the I^{th} column of L

```
      Do 4  J1  = I + 1, n
4     a(J1, I) = a(J1, I) / A(I, I)
```

Continue (for loop I)

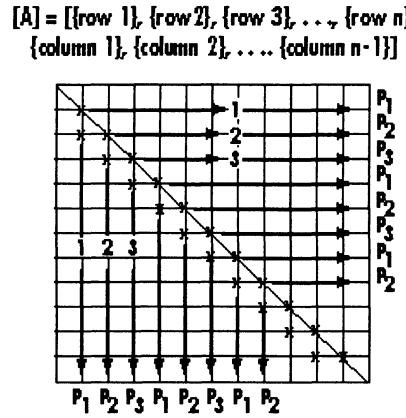


Figure 8.1 Storage scheme for unsymmetrical matrix A in a one-dimensional array

In order to better understand the basic algorithm shown in Table 8.1 for factorization of a full and unsymmetrical matrix, a 3 x 3 matrix [*A*] given in Eq. 8.9 of Section 8.2.2 will be used to verify Table 8.1.

For i = 1, then (please refer to Table 8.1)
- loop 1 will be skipped
- from loop 4

$$a(2,1) = \frac{a(2,1)}{a(1,1)} \equiv l_{21} \qquad (refer\ to\ Eq.\ 8.12)$$

$$a(3,1) = \frac{a(3,1)}{a(1,1)} \equiv l_{31} \qquad (refer\ to\ Eq.\ 8.13)$$

•notice:

The first row of [U] (such as u_{11}, u_{12} and u_{13}) are not required to calculate, because they are the same as the original matrix [A] ($u_{11} = a_{11}$, $u_{12} = a_{12}$ and $u_{13} = a_{13}$)

For i = 2, then

•from loop 2

$$a(2,2) = a(2,2) - a(2,1)*a(1,2) \equiv u_{22} \quad (refer\ to\ Eq.\ 8.12)$$

$$a(2,3) = a(2,3) - a(2,1)*a(1,3) \equiv u_{23} \quad (refer\ to\ Eq.\ 8.12)$$

•from loop 3

$$a(3,2) = a(3,2) - a(1,2) * a(3,1) \equiv partial\ solution\ for\ l_{32}$$

•from loop 4

$$a(3,2) = \frac{a(3,2)}{a(2,2)} \equiv final\ solution\ for\ l_{32}\ (refer\ to\ Eq.\ 8.14)$$

For i = 3, then

•from loop 2 (with K = 1)

$$a(3,3) = a(3,3) - a(3,1) * a(1,3) \equiv partial\ solution\ for\ u_{33}$$

•loop 3 will be skipped

•loop 4 will be skipped

•from loop 2 (with K = 2)

$$a(3,3) = a(3,3) - a(3,2)*a(2,3) \equiv final\ solution\ for\ u_{33}\ (refer\ to\ Eq.8.12)$$

•loop 3 will be skipped

•loop 4 will be skipped

Comments on Table 8.1:

(a) The operations in the innermost loops 2 and 3 are "saxpy" operations (a vector + a scalar * another vector), thus these operations can be done quite fast on vector computers, such as Cray-YMP, Cray-C90, Intel Paragon, or IBM-SP2 computers.

(b) In loop 2, the J^{th} column of U keeps changing, thus it is important to store the upper triangular matrix U according to a row-by-row fashion (see Figure 8.1). This will assure to have a stride 1 in vector computation.

(c) In loop 3, the $J1^{th}$ row of L keeps changing, thus it is important to store the lower triangular matrix L according to a column-by-column fashion (see Figure 8.1). This will assure to have a stride 1 in vector computation.

(d) The "scalar" defined in Table 8.1 is also referred to as "multiplier". In general, the average upper bandwidth or UBM of $[U]$ is different from the average lower bandwidth or LBW of $[L]$. Factorizing the I^{th} row of $[U]$ and the I^{th} column of $[L]$ can be done much more efficiently by skipping some operations when the multiplier is zero. Figures 8.2 and 8.3 show what information is truly needed to factorize the I^{th} row and the I^{th} column of the given unsymmetrical matrix $[A]$.

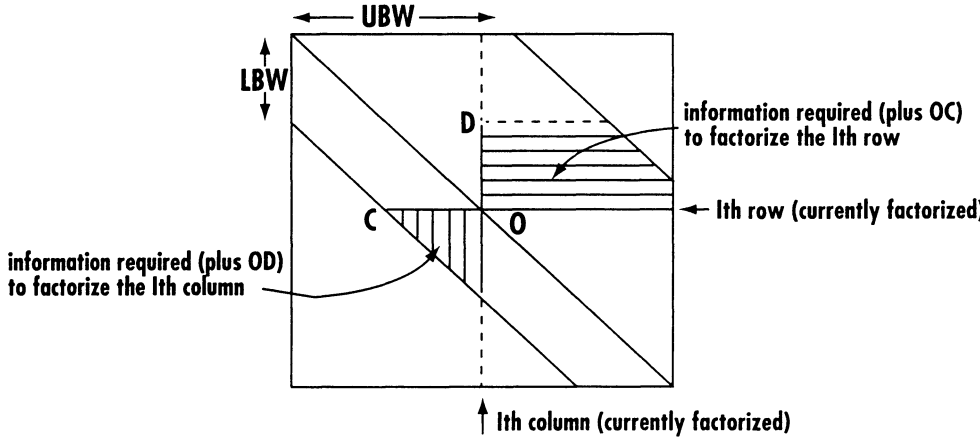


Figure 8.2 Unsymmetrical factorization (UBW > LBW, OD = OC)

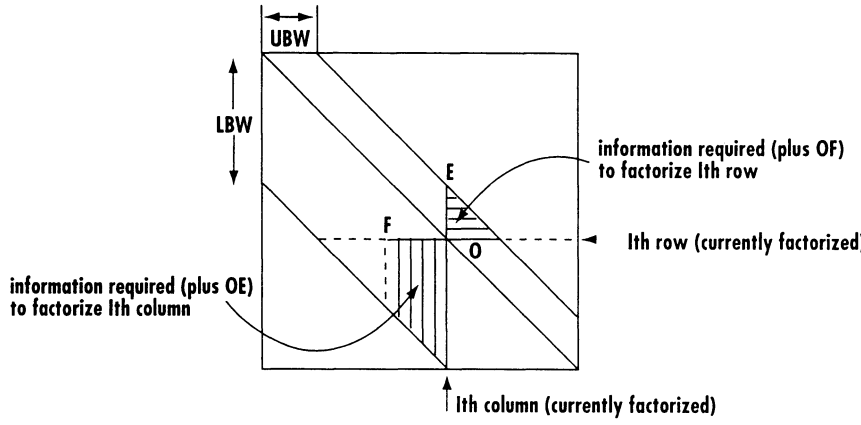


Figure 8.3 Unsymmetrical factorization (UBW < LBW, OF = OE)

8.2.4 Basic algorithm for decomposition of "variable" bandwidths/column heights unsymmetrical matrix

For many practical engineering applications, the unsymmetrical matrix is not full. Instead, the unsymmetrical matrix will have variable bandwidths and variable column heights as shown in Figure 8.4

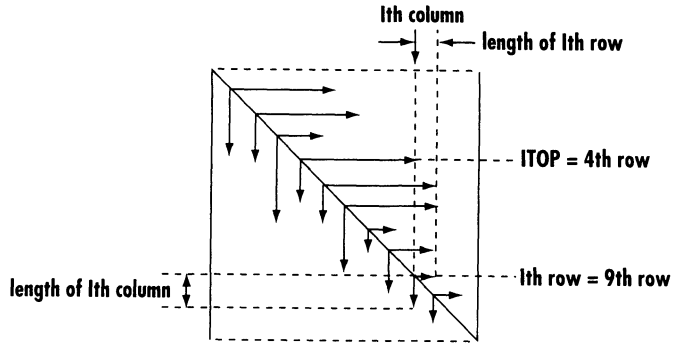


Figure 8.4 Unsymmetrical matrix with variable bandwidths/column heights

In this case, to avoid unnecessary operations with zero values, the algorithm given in Table 8.1 needs to be modified slightly as shown in Table 8.2.

Table 8.2 Basic algorithm for decomposition
(variable bandwidths and column heights)

```
        (For I = 1, 2, 3, . . . n)
c       ITOP is the row number of the top-most nonzero element of the I^th column
c       ITOP also represents the column number of the left-most nonzero element of
the I^th row
        DO 1    K = ITOP, I - 1
c       do-loop 2 is used to update the I^th row of U due to the contribution of the K^th
row
      2 { DO 2 J = I, K + length of the K^th row
          a(I, J) = a(I, J) - a(I, K) * a(K, J)
          CONTINUE }
c       do-loop 3 is used to update the I^th column of L due to the contribution of the
K^th column
      3 { DO 3 J1 = I + 1, K + length of the K^th column
          a(J1, I) = a(J1, I) - a(K, I) * a(J1, K)
          CONTINUE }
1       Continue
```

$$\begin{cases} DO\ 4\ J1\ =\ I\ +\ 1\ ,\ I\ +\ length\ of\ the\ I^{th}\ column \\ a(J1\ ,\ I)\ =\ \dfrac{a(J1\ ,\ I)}{a(I\ ,\ I)} \\ 4\ \ CONTINUE \end{cases}$$

Continue (for loop I)

Comparing the algorithms shown in Table 8.2 and Table 8.1, these two algorithms are quite similar, with the following key differences

(a) The starting value for the index K in loop 1 of Tables 8.1 and 8.2 are 1 and ITOP, respectively. This change is necessary to include the effects of column height of the I^{th} column of $[U]$ and/or row length of the I^{th} row of $[L]$. If the matrix is completely full, then ITOP = 1.

(b) The ending value for the index J in loop 2 of Table 8.1 and 8.2 are N and K + length of K^{th} row, respectively. This change is necessary to include the effects of having "variable bandwidths". If the matrix is full, then K + length of K^{th} row = N.

(c) The ending value for the index $J1$ in loop 3 and loop 4 of Table 8.1 and 8.2 are N and K + length of K^{th} column, or I + length of I^{th} column, respectively. This change is necessary to include the effects of "variable column heights". If the matrix is full, then K + length of K^{th} column = N or I + length of I^{th} column = N.

8.2.5 Algorithms for decomposition of "variable" bandwidths/column heights unsymmetrical matrix with unrolling strategies

The basic vector version of Table 8.2, with small modifications to include loop-unrolling level 6, is given in Table 8.3. It should be noted here that the compiler directives[8.3] are used to force the compiler to ignore potential vector dependencies in trying to vectorize the loop.

Table 8.3 Vector algorithm for factorization

```
        (For i = 1, 2, 3, . . ., n)
        DO 1 K = ITOP, I - 1, 6
CDIRS IVDEP
        DO 2 J = I, K + length of the K^th row
            a(I,J) = a(I,J) - a(I,K) * a(K + I,J)
        +       - a(I,K + 2) * a(K + 2,J) - a(I,K + 3) * a(K + 3,J)
        +       - a(I,K + 4) * a(K + 4,J) - a(I,K + 5) * a(K + 5,J)
2       Continue
CDIRS IVDEP
        DO 3 J1 = I + 1,K + length of the K^th column
        a(J1,I) = a(J1,I) - a(K,I) * a(J1,K) * a(K + 1,I) * a(J1,K + 1)
        +       - a(K + 2,I) * a(J1,K + 2) - a(K + 3,I) * a(J1,K + 3)
```

```
        +          - a(K + 4,I) * a(J1,K + 4) - a(K + 5,I) * a(J1,K + 5)
3       Continue
1       Continue
        DO 4 J1 = I + 1,I I + length of the I^th column
            a(J1,I) = a(J1,I) / a(I,I)
4       Continue
        Continue (for loop i)
```

From Table 8.2, one can clearly see that the previously factorized rows (please refer to loop 1) are used to partially factorize the current I^{th} row (please refer to loop 2) of the upper triangular matrix U. Thus, to improve the vector performance, one should try to increase the work loads of the innermost loop 2. This can be done by unrolling the outer loop 1. For example, a block of six, instead of just one, previously factorized rows are used to partially factorize the current I^{th} row (please refer to Table 8.3).

Similarly, the previously factorized columns (please refer to loop 1) are used to partially factorize the current I^{th} column (please refer to loop 3) of the lower triangular matrix L. A block of six instead of just one previously factorized columns are used to partially factorize the current I^{th} column (please refer to inside loop 3 of Table 8.3).

Thus, Table 8.3 can be obtained by simply making the following minor modifications to Table 8.2:

(a) The increment of loop 1 is changed from 1 to 6 (to consider a block of 6 rows/columns at a time)
(b) Expanding the FORTRAN statement inside loop 2 to include the effects of using 6 rows at a time to partially factorize the current I^{th} row of $[U]$.
(c) Expanding the FORTRAN statement inside loop 3 to include the effects of using 6 columns at a time to partially factorize the current I^{th} column of $[L]$.

8.2.6 Parallel vector algorithms for factorization

Assume the NP processors are specified during the execution. In a sequential code (Table 8.3), it is always known that before updating the I^{th} row and the I^{th} column, the previous $(I - 1)^{th}$ row and $(I - 1)^{th}$ column, have already been updated. In the multiple processors environment, however, only the previous $(I - NP)^{th}$ row and $(I - NP)^{th}$ column have been updated. Thus, the calculation of the contributions by rows and columns (from the $(I - NP + 1)^{th}$ to the $(I - 1)^{th}$) should be synchronized among the NP processors. It should be noted here that the parallel strategy employed here is quite similar to the one discussed in Chapter 5 for variable bandwidths symmetrical equation solvers.

The parallel FORTRAN language Force[8.1] is used in this work to develop a parallel code on multi-processors computer CRAY 2 and CRAY Y-MP. In Force, Presched DO allows all processors to execute the same statement simultaneously with a different do-loop index assigned to each processor. Produce $K = J$ assigns a value J to K and makes K "full". Copy K into L will store K into L only when K is "full" or else the processor has to wait. The combined use of Produce and Copy can provide communications among processors.

Table 8.4 gives a parallel algorithm for the factorization phase on supercomputers with multiple processors, such as CRAY 2 and CRAY Y-MP. By comparing Tables 8.2 through 8.4, it is helpful to identify the differences between the parallel and/or vector algorithm codes and the corresponding sequential one.

Table 8.4 Parallel algorithm for decomposition

```
      Presched DO 100 I = I, n
      DO 1 K = ITOP, I-NP, 6
CDIRS IVDEP
      Do 2 J = I,K + length of the K^th row
          a(I,J) = a(I,J) - a(I,K) * a(K,J) - a(I,K + 1) * a(K + 1,J)
     +          - a(I,K + 2) * a(K + 2,J) - a(I,K + 3) * a(K + 3,J)
     +          - a(I,K + 4) * a(K + 4,J) - a(I,K + 5) * a(K + 5,J)
2     CONTINUE
CDIRS IVDEP
      DO 3 J1 = I + 1,K + length of the K^th column
          a(J1,I) = a(J1,I) - a(K,I) * a(J1,K) - a(K + 1,I) * a(J1,K + 1)
     +          - a(K + 2,I) * a(J1,K + 2) - a(K + 3,I) * a(J1,K + 3)
     +          - a(K + 4,I) * a(J1,K + 4) - a(K + 5,I) * a(J1,K + 5)
3     CONTINUE
1     CONTINUE
      DO 10 K = I-NP + 1, I-1
      Copy Asyn(K) into KK
      DO 20 J = I,K + length fo the K^th row
          a(I,J) = a(I,J) - a(I,K) * a(K,J)
20    CONTINUE
      DO 30 J1 = I + 1,K + length of the K^th column
          a(J1,I) = a(J1,I) - a(K,I) * a(J1,K)
30    CONTINUE
10    CONTINUE
      DO 4 J1 = I + 1,K + length of the I^th column
          a(J1,I) = a(J1,I) / a(I,I)
4     CONTINUE
      Produce Asyn(I) = 1.0
100   End Presched DO
```

A careful comparison between Table 8.3, vector algorithm for factorization, and Table 8.4, parallel-vector algorithm for factorization, suggests that the latter can be obtained from the former with the following modifications:

(a) The outermost loop, for index *I*, is executed in parallel, instead of sequential mode, by using a "Presched" parallel Fortran statement

(b) Loop 1, for index *K*, in Table 8.3 is separated into 2 loops, loops 1 and 10, in Table 8.4. In Table 8.3, the index *K* goes from "*ITOP*" to "*I* - 1". In Table 8.4, the index *K* goes from "*ITOP*" to "*I* - *NP*", see loop 1, and then, from "*I* - *NP* + 1" to "*I* - 1" (see loop 10).

(c) The "copy" parallel Fortran statement inside loop 10 of Table 8.4, will assure that the previous K^{th} row have been completely factorized or else the processor will wait, and can now be safely used to partially factorize the current I^{th} row (see loop 20) and I^{th} column (see loops 30 and 4).

(d) The "Produce" parallel Fortran statement (after loop 4) is used to broadcast to all other processors that the I^{th} row/column have been completely factorized now.

It is important to recognize in here that parallel and vector factorization of the upper triangular matrix [*U*] in this chapter (unsymmetrical solver) is "quite similar" to the algorithms used in Chapter 5 (symmetrical, positive definite solver).

Furthermore, parallel and vector factorization of the lower triangular matrix [*L*] in this chapter is similar to the "image" of the algorithms used in obtaining the upper triangular matrix [*U*].

8.2.7 Forward solution phase [*L*] {*y*} = {*b*}

To simplify the discussions, let us consider a 6 x 6 full-system as shown in the following equation

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ L_{21} & 1 & 0 & 0 & 0 & 0 \\ L_{31} & L_{32} & 1 & 0 & 0 & 0 \\ L_{41} & L_{42} & L_{43} & 1 & 0 & 0 \\ L_{51} & L_{52} & L_{53} & L_{54} & 1 & 0 \\ L_{61} & L_{62} & L_{63} & L_{64} & L_{65} & 1 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{Bmatrix} \tag{8.14}$$

The forward solution for the unknown vector {y} can be proceeded as follows:

$$\begin{aligned} y_1 &= b_1 \\ y_2 &= b_2 - L_{21} y_1 \\ &\vdots \\ y_I &= b_I - \sum_{k=1}^{I-1} L_{I,k} y_k \end{aligned} \tag{8.15}$$

since the lower triangular matrix has been generated and stored in a column-by-column fashion (please see Figure 8.1), thus column 1 of [*L*] has stride 1. Furthermore, to

improve the vector performance, one should try to work with a long vector in the innermost do-loop. Thus, a good strategy is outlined in the following paragraphs:

step 1: Solve for the unknown y_1 (according to Eq. 8.15)
step 2: Use the first column of [L] and operate on the known scalar y_1 in order to update the right-hand-side vector {b}. Thus, the unknown y_2 can be found.
step 3: Use the second column of [L] and operate on the known scaler {b}. Thus, the unknown y_3 can be found.
step 4: Continue to do "similar" operations as mentioned in steps 2 and 3, until all unknowns of vector {y} are found.

The above step-by-step procedure can be simply coded as shown in Table 8.5.

Table 8.5 Basic algorithm for forward solution

```
c     solve for the first unknown (Note: Solution vector {y} will overwrite right-hand-
      side vector {b} to save computer memory)
      b(1) = b(1)
c     Try to solve the subsequent unknowns
      DO 1 I = 2, n, 1
      DO 2 J = I, n
c     Use the previously known solution to update the right-hand-side vector {b}
2     b(J) = b(J) - L(J,I-1) * b(I - 1)
c     Next solution is readily found
      b(I) = b(I)
1     continue
```

It should be mentioned at this time that inside loop 2 of Table 8.5, one has "saxpy" operations (a vector {b} ± scalar $b(I - 1)$ * another vector L), thus the innermost loop 2 can be executed very efficiently on the vector computers, such as the CRAY Y-MP, CRAY-C90, etc..

However, a careful observation of the above 4-step procedure and the data structure shown in Eq. 8.14 suggests that even better vector performance can be achieved by suing the "loop-unrolling" technique, with a simple modification to Table 8.5

The key idea in "loop-unrolling" technique is to add a more heavy work load ("saxpy" operations) into the innermost do-loop (see loop 2 of Table 8.5). A simple way to achieve this objective is to use 2 or more columns instead of just 1 column of matrix [L] and operate on previously known 2 (instead of just 1) solutions. Thus, a loop-unrolling algorithm for a forward solution can be shown in Table 8.6.

Table 8.6 Loop-unrolling (level 2) for forward solution

```
c       Solve the first 2 unknowns
        b(1) = b(1)
        b(2) = b(2) - L(2,1) * b(1)
c       For subsequent unknowns
        DO 1 I = 3, n, 2
        DO 2 J = I, n
        b(J) = b(J) - L(J,I-1) * b(I-1)
             - L(J,I-2) * b(I-2)
2       CONTINUE
c       Next 2 solutions can be found
        b(I) = b(I)
        b(I + 1) = b(I + 1) - L(I + 1, I) * b(I)
1       continue
```

Comments on Table 8.6:

(a) In actual computer implementation, loop-unrolling level 6 or 8 can be used, instead of just using level 2 (see the increment 2 in loop 1 of Table 8.6).

(b) For a general matrix with dimension n, the use of the loop-unrolling technique will require "special" treatments for the "left-over" columns of the matrix L.

(c) To simplify the discussions, the matrix system of equations shown in Eq. 8.14 is assumed to be "full". However, in actual computer implementation, variable column-heights of the lower triangular matrix [L], and variable row-length or bandwidth of the upper triangular matrix [U] can be accommodated to avoid unnecessary operations (on the zeros).

(d) In actual computer implementation, the lower and upper factorized matrices [L] and [U] will be stored in a 1-D array and the original matrix, which is also stored in a 1-D array, will be overwritten by [L] and [U] in order to save computer memory.

8.2.8 Backward solution phase [U] {x} = {y}

To simplify the discussions, let us consider the following 6 x 6 full system of equations

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} & u_{16} \\ 0 & u_{22} & u_{23} & u_{24} & u_{25} & u_{26} \\ 0 & 0 & u_{33} & u_{34} & u_{35} & u_{36} \\ 0 & 0 & 0 & u_{44} & u_{45} & u_{46} \\ 0 & 0 & 0 & 0 & u_{55} & u_{56} \\ 0 & 0 & 0 & 0 & 0 & u_{66} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{Bmatrix} \tag{8.16}$$

The backward solution for the unknown vector {x} can be proceeded as follows:

$$
\begin{aligned}
x_6 &= \frac{y_6}{u_{6,6}} \\
x_5 &= \frac{y_5 - u_{56} x_6}{u_{55}} \\
&\vdots \\
x_I &= \frac{\left(y_I - \sum_{k=I+1}^{N} u_{i,k} x_k \right)}{U_{I,I}}
\end{aligned}
\qquad \textbf{(8.17)}
$$

As an example, $x_2 = \dfrac{y_2 - \left(u_{23} x_3 + u_{24} x_4 + u_{25} x_5 + u_{26} x_6 \right)}{U_{2,2}}$

The operations involved in the above parenthesis are called "dot-product" operations. It is the dot product between the 2 vectors

$$\left\{ u_{23}, u_{24}, u_{25}, u_{26} \right\} \cdot \begin{Bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix}$$

Since the upper triangular matrix has been generated and stored in a row-by-row fashion (please refer to Figure 8.1), thus, each row of $[U]$ has stride 1. However, each column of $[U]$ has very undesirable stride (column stride of $[U]$ is greater than 1). Due to this reason, it is not efficient, in this case, to use loop-unrolling technique (for example, having found the unknown x_6, then using column 6 to operate on the scalar x_6 for the purpose of updating the right-hand vector $\{y\}$) as discussed in the previous section. The backward solution (please refer to Eq. 8.17) can be coded using "dot-product" operations, instead of "saxpy" operations as discussed in the forward solution phase, as shown in Table 8.7.

Table 8.7 Basic algorithm for backward solution

```
c       Solve the last unknown
        x(N) = y(N) / U(N,N)
c       For subsequent unknowns
        DO 1 I = N-1, 1, -1
c       Performing the summation (or dot-product) operations in Eq. 8.17
        DO 2 K = I + 1, N
2       SUM1 = SUM1  + U(I,K) * x(K)
        x(I) = y(I) - SUM1 / U(I,I)
1       continue
```

It should be mentioned here that the dot-product operations inside loop 2 of Table 8.7 can be vectorized quite well, on vector computers, since the row vector of [*U*] has stride 1 (recall that the matrix *U* is stored in a row-by-row fashion).

However, a careful observation of Eq. 8.17 and the storage scheme used for matrix [U] shown in Eq. 8.16 suggests that an even better vector-performance can be achieved by using the "vector-unrolling" technique, with simple modifications to Table 8.7.

The key idea in "vector-unrolling" technique is to add more work loads (dot-product operations) into the innermost do-loop (see loop 2 of Table 8.7). A simple way to achieve this objective is to use two (or more) rows, instead of just one row of matrix [*U*] and operate on the previously known two (or more) rows, instead of just one row of solutions. The results "vector-unrolling" (level 2 unrolling is assumed), algorithm for backward solution is illustrated in Table 8.8.

Table 8.8 Vector-unrolling algorithm for backward solution

```
c       Solve the last 2 (or more) unknowns
        x(N) = y(N) / U(N,N)
        x(N-1) = y(N-1) - U(N-1,N) * x(N) / U(N-1,N-1)
c       For subsequent unknowns
        DO 1 I = N-2, 1, -2
c       Performing 2 (or more) dot product operations in Eq. 8.17
        DO 2 K = I + 1, N
        SUM1 = SUM1 + U(I,K) * x(K)
2       SUM2 = SUM2 + U(I-1,K) * x(K)
        x(I) = y(I) - SUM1 / U(I,I)
        x(I-1) = y(I-1) - SUM2 - U(I-1,I) * x(I) / U(I-1,I-1)
1       continue
```

8.3 Numerical Evaluations

The numerical performance of the proposed unsymmetrical solver is presented in this section. Both "test" problems as well as practical engineering problems are considered.

To check the accuracy of the solution, a residual vector *r* is defined as

$$r = Ax - b \tag{8.18}$$

where *x* is the solution of Eq. 8.1 by the proposed solver. The (machine dependent) precision parameter *E* is defined as

$$1.0 + E > 1.0 \tag{8.19}$$

which means *E* is the smallest positive number that satisfies Eq. 8.19.[8.4] In this work the partial coefficient matrix *A* is automatically generated as

$$a(i,j) = \frac{1.0}{j} \qquad (for\ j > i) \tag{8.20}$$

$$a(i,j) = \frac{1.0}{j} + \frac{1.0}{(i+j)} \qquad (for\ j < i) \tag{8.21}$$

$$a(i,i) = i \qquad (for\ 1 < i < n) \tag{8.22}$$

$$b(i) = 1.0 \qquad (for\ 1 < i < n) \tag{8.23}$$

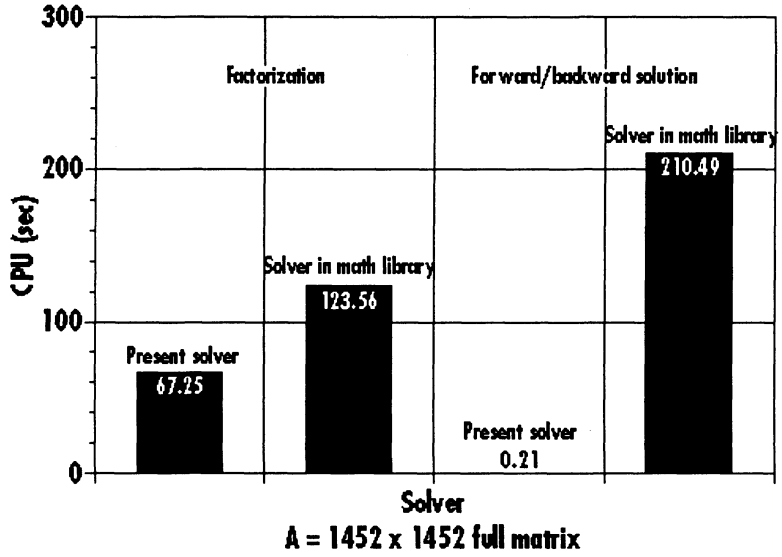


Figure 8.5 Different solvers on CONVEX C220

The solution is regarded as accurate as long as all the elements of the vector r remain less than $n * E$, i.e. $\|r\| < n * E$, where n is the largest absolute value of $a(i, j)$.

Example 1: In the example, matrix A and vector b are automatically generated with $n = 1452$, $NBWU = NBWL = n = 1452$. The CPU time for solving this equation on the CONVEX C220 computer is given in Fig. 8.5 and is compared with the time given by the equation solver form the library subroutines installed on the CONVEX C220. The machine precision parameter E for the CONVEX C220 has a value of $2.22024605 \times 10^{-16}$. The computed residual norm is $||r||_\infty = 1.0 \times 10^{-15}$, which is less than $n * E$.

Example 2: In this example, A is automatically generated with n = 2000, and the bandwidths on the CRAY Y-MP with 1, 2, 4 and 8 processors. The results are presented in Fig. 8.6 (for the CRAY Y-MP, the precision parameter is $E = 7.105427357601 \times 10^{-15}$).

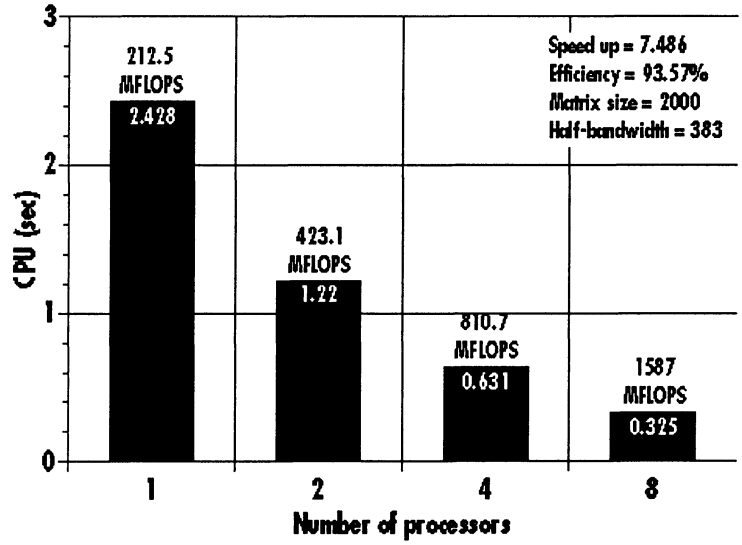


Figure 8.6 Factorization of A = LU on CRAY Y-MP

Example 3: The non-linear (large deflection and non-linear aerodynamics) 3-D panel flutter analysis (as shown in Fig. 8.7) by the finite element method similar to the method proposed in Mei and Gray[8.5] for 2-D panels, is used to evaluate the performance of the proposed parallel-vector unsymmetric equation solver for engineering applications on super-computers. The panel is modeled by (12 x 12) = 144 conforming rectangular elements. There are six degrees-of-freedom per node for each element. The element nodal dis-placements are: two in-plane displacements u and v, the transverse deflection w and its derivatives w_x, w_y and w_{xy} at each node. Thus, there is a total of 24 degrees-of-freedom per element. Due to the non-linear damping effects encountered in the non-linear aerodynamics, the configuration solution space is transformed to a state solution space. This, in effect, doubles to total number of active degrees-of-freedom[8.5]. The final coefficient matrix A is a 1452 x 1452 unsymmetric matrix, with its upper half-band-width $NBWU$ = 778 and lower half-bandwidth $NBWL$ = 727. It is required that during the flutter analysis, the coefficient matrix A should be updated, decomposed and the unknown vector should be found repeatedly. Numerical results are obtained on the CRAY Y-MP, using 1, 2, 4, 6 and 8 processors in a non-dedicated time and are shown in Figs. 8.8 and 8.9[8.6]. The elapsed time for the same size problem on the CRAY 2 (Voyager using 1, 2 and 4 processors is also presented in Fig. 8.10

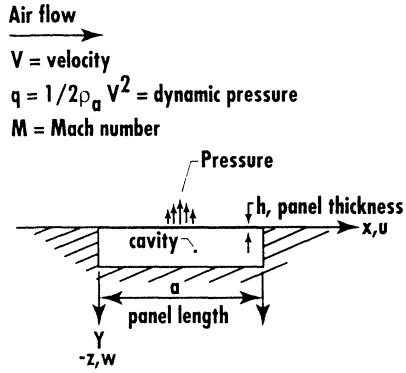


Figure 8.7 Finite element panel flutter analysis

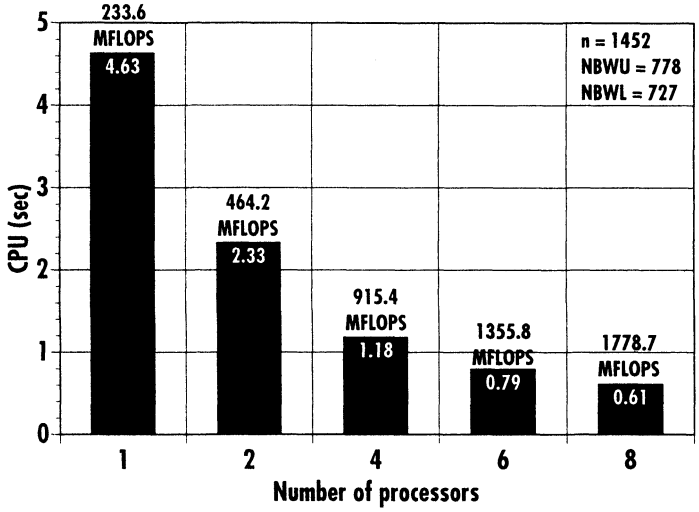


Figure 8.8 Panel flutter analysis on CRAY Y-MP (CPU time for factorization and forward/backward elimination)

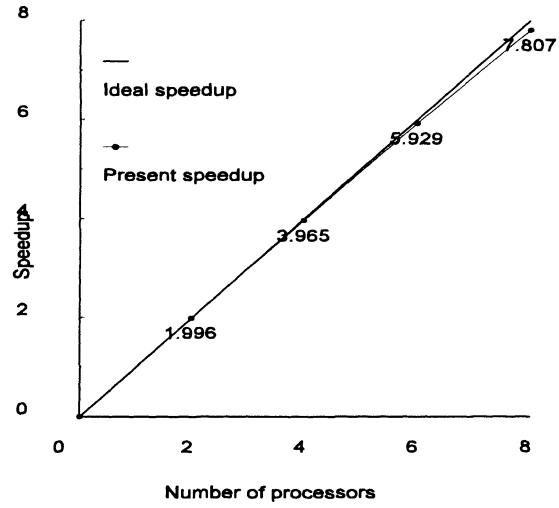


Figure 8.9 Speedup for panel flutter analysis on Cray Y-MP

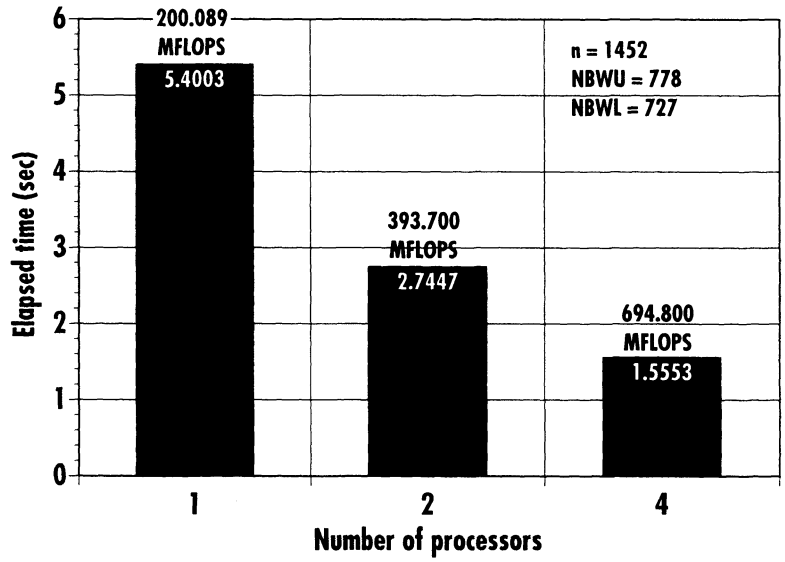


Figure 8.10 Elapsed time for factorization on the CRAY-2 (Voyager)

From the above numerical results it can be seen that the proposed equation solver is efficient both in the vector and parallel computer environment and gives accurate results to the machine precision.

8.4 A Few Remarks on Pivoting Strategies

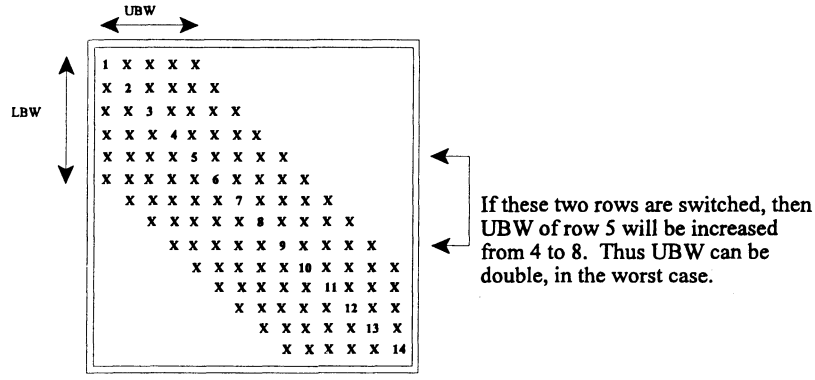
During the factorization phase for obtaining the lower triangular matrix [*L*] (see Eqs.8.4 and 8.12), if the diagonal term *U*(*r, r*), shown in Eq. 8.4, is close to zero, then numerical difficulties will occur. Pivoting strategies are, therefore, required in these cases. For the unsymmetrical matrix, with it supper bandwidth *UBW* and its lower bandwidth *LBW*, switching rows will, in the worst case, double the upper bandwidth and make the lower bandwidth to become full. Considering Fig. 8.11, for example, if the factorized diagonal term $U_{5,5}$ becomes zero, row number 5 can be switched to any row between rows number 6 and 9 (since we would also like to make sure that after switching rows, none of the diagonal terms are zero). Thus, the worst case will occur if row number 5 is switched with row number 9, since the upper bandwidth *UBW* of row number 5 will be increased from 4 (not including the diagonal term) to 8. Furthermore, when row number 5 is switched to row number 9, the lower bandwidth of column number 1 will be increased from 5 to 8. At a later stage of the factorization process, the factorized diagonal term of the "new" row number 4 may become zero, assuming that the "new" row number 9 needs to be switched with row number 14, then the maximum upper bandwidth *UBW* of the "new" row number 5 still remains to be 8. However, the maximum lower bandwidth of column number 1 will become full!

Three more remarks are in order:

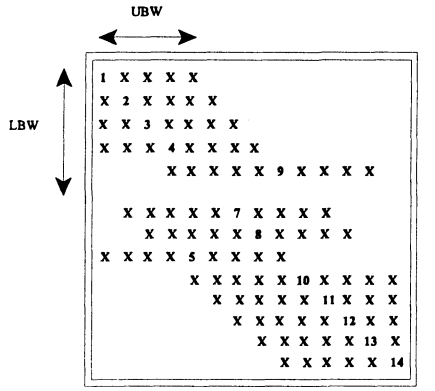
(a) when the rows of the (unsymmetrical) coefficient matrix [*A*] are switched, the corresponding rows of the right-hand-side vector {*b*} (see Eq. 8.1) need to be switched also.

(b) when the rows of the (unsymmetrical) coefficient matrix [*A*] are switched, the row-lengths and the diagonal pointer array *MAXA*(-) (please refer to Chapter 5) need to be re-defined.

(c) when the rows are switched, the total number of non-zero terms (including fills-in terms) upon completion of the factorized process can be predicted, and therefore memory allocations can be assigned (based on the worse situations where *UBW* can be doubled, and *LBW* can be full) even before performing the factorization phase.



(a) "Before" Switching rows # 5 and # 9



(b) "After" Switching rows # 5 and # 9

Figure 8.11 Effects of switching rows on UBW and LBW

8.5 A FORTRAN Call Statement to Subroutine UNSOLVER

Based upon the algorithms discussed in Tables 8.1 through 8.3 for solving a system of unsymmetrical equations, a vectorized version of subroutine UNSOLVER has been written for general users.

The subroutine's arguments will be explained in the following paragraphs.

Subroutine UNSOLVER (A, N, NBWU, NBWL, ISEG, NROL, IROWLU, IROWLL, $ TU, MAXU, MAXL, X, Y, B)

where:

A	=	A real, One-dimensional array (with the dimension N * N) to store the unsymmetrical matrix, as explained in Fig. 8.1
N	=	Number of equations
NBWU	=	Maximum upper-bandwidth (including diagonal term) of the unsymmetrical matrix
NBWL	=	Maximum Lower-bandwidth (excluding diagonal term) of the unsymmetrical matrix

ISEG = For a certain vector computer, such as the IBM-R600/590 Workstation, the vector performance increases with the vector length. However, the vector performance decreases when the vector length exceeds a certain threshold value, say 300, as an example. Thus, if the upper and/or lower bandwidth is large, say NBWU = 700, NBWL = 800, then the user should input the value of, say 300, for ISEG. Thus, in this example, the vector length of 700 or 800 will be broken into segments 300 + 300 + 100 or 300 + 300 + 200.

NROL = 16 (unrolling level number, with dimension N, to represent the row-length, or the variable bandwidth)

IROWLU = an integer, one-dimensional array, for each row including the diagonal term, of the upper-triangular matrix portion of the unsymmetrical matrix [A]

IROWLL = an integer, one-dimensional array, with the dimension N, to represent the column-height for each column, excluding the diagonal term, of the lower-triangular matrix portion of the unsymmetrical matrix [A]

TU = A real, one-dimensional working array, with the dimension 16 * N

MAXU = An integer, one-dimensional array, with dimension N, to represent the starting location for each row of the upper-triangular matrix portion of the unsymmetrical matrix [A]

MAXL = An integer, one-dimensional array, with dimension N, to represent the starting location of reach column of the lower-triangular matrix portion of the unsymmetrical matrix [A]

X = A real, one-dimensional array, with dimension N, to store the solution vector

Y = A real, one-dimensional working array, with dimension N

B = A real, one-dimensional array, with dimension N, to store the right-hand-side vector of a system of unsymmetrical equations.

Using the example shown in Figure 8.11(a), one has

N = 14

NBWU = 5 (including diagonal term)

NBWL = 5 (including diagonal term)

$$IROWLU \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{Bmatrix} \qquad IROWLL \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 4 \\ 3 \\ 2 \\ 1 \\ 0 \end{Bmatrix}$$

$$MAXU \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 6 \\ 11 \\ 16 \\ 21 \\ 26 \\ 31 \\ 36 \\ 41 \\ 46 \\ 51 \\ 55 \\ 58 \\ 60 \end{Bmatrix} \qquad MAXL \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 6 \\ 11 \\ 16 \\ 21 \\ 26 \\ 31 \\ 36 \\ 41 \\ 46 \\ 50 \\ 53 \\ 55 \\ 55 \end{Bmatrix}$$

8.6 Summary

A parallel-vector unsymmetric equation solver for supercomputers has been developed. Both tested problems and practical engineering problems have been used to evaluate the performance of the proposed solver in a parallel and vector computer environment. Results obtained to date have indicated that the proposed solver is fast and accurate. Finally, the developed solver has been demonstrated to interface with the existing finite elements' computer code with minimum effort and substantially improves the solution performance of non-linear iterative methods.

8.7 Exercises

8.1 Given the following system of unsymmetrical equations

$$[A]\{x\} = \{b\}$$

where

$$[A] = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -2 & 4 & -1 & 0 \\ 0 & -2 & 4 & -2 \\ 0 & 0 & -3 & 8 \end{bmatrix} \quad \textit{and} \quad \{b\} = \begin{Bmatrix} 1 \\ 1 \\ 0 \\ 5 \end{Bmatrix}$$

(a) Using a hand calculator, find the $[L]$ $[U]$ factorization of $[A]$
(b) Find the forward solution $\{y\}$ from $[L]$ $\{y\} = \{b\}$
(c) Find the backward solution {x} from $[U]$ $\{x\} = \{y\}$

8.2 Assuming the given matrix $[A]$ in problem 8.1 is full, and using the algorithm shown in Table 8.1 as the basic building block, write the FORTRAN computer program to perform the $[L]$ $[U]$ factorization. Also, verifying your FORTRAN program by using the same matrix $[A]$, given in Problem 8.1.

8.3 Modifying the FORTRAN computer program in Problem 8.2, so that the effects of the upper-bandwidth (*IUBW*) and lower-bandwidth (*ILBW*) can be exploited. Also, verifying your program by using the same matrix $[A]$, given in Problem 8.1

8.4 Suppose the entire given, unsymmetrical matrix $[A]$ is stored in a row-by-row fashion, as shown in Figure P8.4

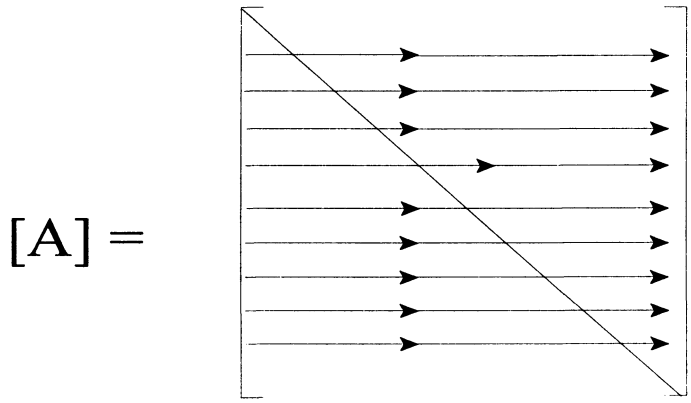


Figure P8.4 Row-by-row storage scheme for an unsymmetrical matrix

For "stride 1" operations, will you have SAXPY or DOT-PRODUCT operations for obtaining the

(a) Lower triangular matrix $[L]$?
(b) Upper-triangular matrix $[U]$?
(c) Forward solution $\{y\}$?
(d) Backward solution $\{x\}$?

Please explain your reason(s) in great detail!

8.5 Re-solve problem 8.4, but assuming the entire given, unsymmetrical matrix $[A]$ is stored in a column-by-column fashion, as shown in figure P8.5

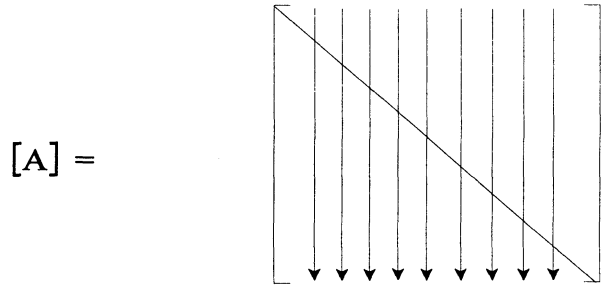


Figure P8.5 Column-by-column storage scheme for an unsymmetrical matrix

8.6 Re-solve problem 8.4, but assuming the entire given, unsymmetrical matrix [A] is stored as column-by-column fashion for the upper-triangular matrix [U] and row-by-row fashion for the lower-triangular matrix [*L*].

8.8 References

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

8.2 Agarwal, T.K., O.O. Storaasli and D.T. Nguyen, "A Parallel-Vector Algorithm for Rapid Structural Analysis on High-Performance Computers," Proceedings of the 31st Structures, Structural Dynamics and Materials Conference, Long Beach, California, pp.662-672, April 1990.

8.3 CRAY, "Mini Manual," CR-1, NASA, March 1989.

8.4 Hughes, T.J.R., *The Finite Element Method*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

8.5 Mei, C. and C.E. Gray, Jr., "A Finite Element Method for Large Amplitude Two-Dimensional Panel Flutter at Hypersonic Speeds," "Proceedings of the 30th Structures, Structural Dynamics and Materials Conference, Mobile, Alabama, pp.37-51, April 1989. Also to appear in *AIAA Journal* (1991).

8.6 Qin, J., C.E. Gray, Jr., C. Mei and D.T. Nguyen, "A Parallel-Vector Equation Solver for Unsymmetric Matrices on Super Computers," *Computing Systems in Engineering*, Vol. 2, No. 2/3, pp.197-201 (1991).

9 A Tridiagonal Solver for Massively Parallel Computers

9.1 Introduction

Efficient solution of large tridiagonal systems of linear equations is important in many engineering applications[9.1-9.2]. Many algorithms have been developed in the last two decades for efficiently solving large tridiagonal systems on vector and/or parallel computers (for a complete survey, see [9.3 - 9.5]). Among them, the cyclic reduction method[9.6] seems to be the most suitable one on vector computers[9.7-9.8]. To optimize the performance, Madsen and Rodrigue[9.9] suggested to use the cyclic reduction combined with the standard Gaussian elimination. Recently, Fabio[9.10] developed a tridiagonal solver using parallel cyclic reduction along with recursive Gaussian elimination, the maximum speedup is NP/4 (NP is the number of processors used). Similar work can also be found in Plum[9.11] and Hajj[9.12]. It is interesting to note that the maximum speedup in [9.12] is also bounded by NP/4.

For many engineering applications, we need to solve a large tridiagonal systems with a lot of right-hand-side[9.1-9.2] vectors, thus it is desirable to perform LU factorization only once and followed by repeated forward/backward substitutions. In this chapter, we develop an efficient tridiagonal solver for solving large systems of equations on massively parallel (distributed) computers.

9.2 Basic Sequential Solution Procedures for Tridiagonal Equations

Consider a tridiagonal linear system of equations

$$b_i x_{i-1} + a_i x_i + c_i x_{i+1} = y_i \quad i = 1, 2, 3, \ldots, n \tag{9.1}$$

or in the matrix form

$$Tx = y \tag{9.2}$$

with

$$T = \begin{bmatrix} a_1 & c_1 & & & & \\ b_2 & a_2 & c_2 & & & \\ & \cdot & \cdot & & & \\ & & \cdot & \cdot & & \\ & & & \cdot & \cdot & c_{n-1} \\ & & & & b_n & a_n \end{bmatrix} \tag{9.3}$$

$b_1 = c_n = 0$. The standard Gaussian elimination for solving these equations is to decompose T into T = LU by

$$d_1 = a_1^{-1} \tag{9.4}$$

$$\gamma_i = b_i * d_{i-1} \tag{9.5}$$

$$d_i = \left(a_i - \gamma_i c_{i-1}\right)^{-1} for\, i = 2, 3, \ldots, n \tag{9.6}$$

such that

$$L = \begin{bmatrix} 1 & & & & & \\ \gamma_2 & 1 & & & & \\ & \gamma_3 & 1 & & & \\ & & \cdot & \cdot & & \\ & & & \cdot & \cdot & \\ & & & & \gamma_n & 1 \end{bmatrix} \tag{9.7}$$

$$U = \begin{bmatrix} d_1^{-1} & c_1 & & & & \\ & d_2^{-1} & c_2 & & & \\ & & \cdot & \cdot & & \\ & & & \cdot & \cdot & \\ & & & & \cdot & c_{n-1} \\ & & & & & d_n^{-1} \end{bmatrix} \tag{9.8}$$

Then defining Ux = z and solve Lz = y by

$$z_1 = y_1 \tag{9.9}$$

$$z_i = y_i - \gamma_i z_{i-1} \; for \; i = 2, 3, ..., n \tag{9.10}$$

and solve Ux = z by

$$x_n = z_n * d_n \tag{9.11}$$

$$x_i = \left(z_i - c_i x_{i+1}\right) d_i \quad for \; i = n-1, n-2, \ldots, 1 \tag{9.12}$$

If the upper triangular matrix [U], shown in Eq. 9.8, is defined as

$$U = \begin{bmatrix} \alpha_1 & c_1 & & & & \\ & \alpha_2 & c_2 & & & \\ & & \cdot & \cdot & & \\ & & & \cdot & \cdot & \\ & & & & \cdot & c_{n-1} \\ & & & & & \alpha_n \end{bmatrix} \tag{9.13}$$

Then, the factorized Eqs. 9.4 through 9.6 can be expressed as

$$\alpha_1 = a_1$$

$$\gamma_i = \frac{b_i}{\alpha_{i-1}} \quad , for \; i = 2, 3, ..., n$$

$$\alpha_i = a_i - c_{i-1} * \gamma_i \; , for \; i = 2, 3, ..., n \tag{9.16}$$

However, Eqs. 9.9 and 9.10 for forward solution phase will remain to be unchanged. Finally, Eqs. 9.11 and 9.12 for backward solution phase will be changed into:

$$x_n = \frac{z_n}{\alpha_n} \tag{9.17}$$

$$x_i = \frac{\left(z_i - c_i * x_{i+1}\right)}{\alpha_i} \tag{9.18}$$

Remarks:

(1) Factorization, forward and backward equations, as shown in Eqs. 9.14 through

9.16, Eqs. 9.9 - 9.10, and 9.17 - 9.18, are recursive equations. Thus, these computations are highly sequential in nature.

(2) The total number of operations for factorization (see Eqs. 9.14 through 9.16), forward substitution (see Eqs. 9.9 and 9.10), and backward substitution (see Eqs. 9.17 and 9.18) are 3n, 2n, and 3n + 1, respectively. Thus, total number of operations are approximately 8n operations.

The following simple example is used to clarify the above derived Eqs. 9.4 through 9.6, 9.9 - 9.10, 9.11 and 9.12:

Given the following 3 x 3 tri-diagonal matrix and the right-hand-side vector $\vec{y} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix}$

$$T = \begin{bmatrix} a_1 & c_1 & 0 \\ b_2 & a_2 & c_2 \\ 0 & b_3 & a_3 \end{bmatrix} \tag{9.19}$$

One can factorize the above 3 x 3 matrix [T] as

$$[T] = [L][U] \tag{9.20}$$

or

$$\begin{bmatrix} a_1 & c_1 & 0 \\ b_2 & a_2 & c_2 \\ 0 & b_3 & a_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \gamma_2 & 1 & 0 \\ 0 & \gamma_3 & 1 \end{bmatrix} \begin{bmatrix} d_1^{-1} & c_1 & 0 \\ 0 & d_2^{-1} & c_2 \\ 0 & 0 & d_3^{-1} \end{bmatrix} \tag{9.21}$$

From Eq. 9.21, one can see that the five unknowns (γ_2, γ_3, d_1, d_2, and d_3) can be obtained by simultaneously solving the following five equations:

$$\left.\begin{array}{llll} a_1 = d_1^{-1} & or & d_1 = a_1^{-1} \\ b_2 = \gamma_2 d_1^{-1} & or & \gamma_2 = b_2 d_1 \\ a_2 = \gamma_2 c_1 + d_2^{-1} & or & d_2 = \left(a_2 - \gamma_2 c_1\right)^{-1} \\ b_3 = \gamma_3 d_2^{-1} & or & \gamma_3 = b_3 d_2 \\ a_3 = \gamma_3 c_2 + d_3^{-1} & or & d_3 = \left(a_3 - \gamma_3 c_2\right)^{-1} \end{array}\right\} \tag{9.22}$$

Eq. 9.22 can be readily identified as the same one given by Eqs. 9.4 through 9.6.

The forward solution of [L] {z} = {y} can be written as

$$\begin{bmatrix} 1 & 0 & 0 \\ \gamma_2 & 1 & 0 \\ 0 & \gamma_3 & 1 \end{bmatrix} \begin{Bmatrix} z_1 \\ z_2 \\ z_3 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} \tag{9.23}$$

From the 1st, 2nd then 3rd equation of Eq. 9.23, one obtains:

$$\left.\begin{aligned} z_1 &= y_1 \\ z_2 &= y_2 - \gamma_2 z_1 \\ z_3 &= y_3 - \gamma_3 z_2 \end{aligned}\right\} \qquad \textbf{(9.24)}$$

It is obvious that Eq. 9.24 can also be obtained directly from Eqs. 9.9 and 9.10.

The backward solution of [U] {x} = {z} can be written as

$$\begin{bmatrix} d_1^{-1} & c_1 & 0 \\ 0 & d_2^{-1} & c_2 \\ 0 & 0 & d_3^{-1} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} z_1 \\ z_2 \\ z_3 \end{Bmatrix} \qquad \textbf{(9.25)}$$

From the 3rd, 2nd then 1st equation of Eq. 9.25, one obtains

$$\left.\begin{aligned} x_3 &= z_3 d_3 \\ x_2 &= \left(z_2 - c_2 x_3\right) d_2 \\ x_1 &= \left(z_1 - c_1 x_2\right) d_1 \end{aligned}\right\} \qquad \textbf{(9.26)}$$

Eq. 9.26 can also be obtained directly from Eqs. 9.11 and 9.12.

In practice, the z and x vectors are all stored in the y vector and the γ and d vectors are stored in b and a vectors, respectively. Thus, the storage requirement for this algorithm is only 4*n and the number of operations needed is 9*n (Note: only n divisions are needed during the LU factorization, no divisions required for forward/backward eliminations). This scheme is faster than the one in Ref. [9.1, pp.115], even though only 8*n operations are needed there. Since all the above three equations (Eqs. 9.4 through 9.6, 9.9 - 9.10 and 9.11 - 9.12) are recursive, this algorithm usually can not be vectorized on most vector computers. That is why the cyclic reduction algorithm has been widely used[9.1, 9.2, 9.5, 9.7-9.10, 9.13].

9.3 Cyclic Reduction Algorithm

The key idea in the cyclic reduction algorithm is, through a sequence of row operations, to transform the original tridiagonal system into smaller tridiagonal systems. In order to better understand the details of the cyclic reduction algorithm, let us try to obtain the solution for the following 8 x 8 tridiagonal system:

$$\begin{bmatrix} 2 & -1 & & & & & & \\ -1 & 2 & -1 & & & & & \\ & -1 & 2 & -1 & & & & \\ & & -1 & 2 & -1 & & & \\ & & & -1 & 2 & -1 & & \\ & & & & -1 & 2 & -1 & \\ & & & & & -1 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_8 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 0 \\ 0 \\ . \\ . \\ . \\ 0 \end{Bmatrix} \qquad (9.27)$$

In the cyclic reduction algorithm, one can modify the "EVEN" row number of the augmented matrix

$$\left[\begin{array}{cccccccc|c} 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 2 & -1 & 0 & & & & 0 & 0 \\ 0 & -1 & 2 & -1 & & & & 0 & 0 \\ 0 & & -1 & 2 & -1 & & & 0 & 0 \\ 0 & & & -1 & 2 & -1 & & 0 & 0 \\ 0 & & & & -1 & 2 & -1 & 0 & 0 \\ 0 & & & & & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right] \qquad (9.28)$$

by the following row operations

Step (a) Multiply the "ODD" row number by a factor 0.5, this factor can be different for different tridiagonal systems, then the resulted odd row will be added to the "EVEN" row BELOW it in order to create new even rows.

Step (b) Multiply the "ODD" row number by a factor 0.5, then the resulted odd row will be added to the EVEN row ABOVE it in order to create new even rows.

Step (c) Obtain the reduced tridiagonal system from Step (b). This can be seen easily by extracting only EVEN rows/columns at the end of Step (b).

Step (d) Go back to Step (a) until the reduced tridiagonal system has the dimension 1 x 1.

Detailed implementations of the above 4-Step procedure is given in the following sections.

Step (a)

0.5**	2	-1	0	0	0	0	0	0	1
	-1	2	-1	0	0	0	0	0	0
0.5*	0	-1	2	-1	0	0	0	0	0
	0	0	-1	2	-1	0	0	0	0
0.5*	0	0	0	-1	2	-1	0	0	0
	0	0	0	0	-1	2	-1	0	0
0.5*	0	0	0	0	0	-1	2	-1	0
	0	0	0	0	0	0	-1	1	0

	2.0	-1.0	0	0	0	0	0	0	1.0
	0	1.5	-1.0	0	0	0	0	0	0.5
0.5*	0	-1.0	2.0	-1.0	0	0	0	0	0
➡	0	-0.5	0	1.5	-1.0	0	0	0	0
0.5*	0	0	0	-1.0	2.00	-1.0	0	0	0
	0	0	0	-0.5	0	1.5	-1.0	0	0
0.5*	0	0	0	0	0	-1.0	2.0	-1.0	0
	0	0	0	0	0	-0.5	0	0.5	0

Notes: 1. Original tridiagonal system size is $2^3 = 8$
2. Column #9 of the above matrix represents the right-hand-side vector {y} in Eq. 9.2

Step (b)

2	-1	0	0	0	0	0	0	1	
0	1	0	-0.5	0	0	0	0	0.5	
0	-1	2	-1	0	0	0	0	0	
0	-0.5	0	1	0	-0.5	0	0	0	
0	0	0	-1	2	-1	0	0	0	
0	0	0	-0.5	0	1	0	-0.5	0	
0	0	0	0	0	-1	2	-1	0	
0	0	0	0	0	-0.5	0	0.5	0	

➡

0.5*	1	-0.	0	0	0.5	x_2
	-0.5	1	-0.	0	0	x_4
0.5*	0	-0.	1	-0.5	0	x_6
	0	0	-0.	0.5	0	x_8

Note: Reduced Tridiagonal System size is $2^2 = 4$

Step (c)

	1	-0.5	0	0	0.5		1	-0.50	0	0	0.5	
	0	0.75	-0.50	0	0.25		0	0.5	0	-0.25	0.25	
0.5*	0	-0.5	1	-0.50	0	⇒	0	-0.50	1	-0.50	0	
	0	-0.2	0	0.25	0		0	-0.25	0	0.25	0	

⇓

	0.5	-0.25	0.25	0.5*	0.5	-0.25	0.25	x_4
	0	0.125	0.125	⇐	-0.25	0.25	0	x_8

Note: Reduced tridiagonal system size is $2^1 = 2$

Step (d)
From the above reduced 2 x 2 system, one obtains

$$0.125 * x_8 = 0.125$$

Hence:

$$x_8 = 1 \tag{9.29}$$

Note: Reduced tridiagonal system size is $2^0 = 1$

Having obtained the solution for x_8 from Eq. 9.29, we can substitute the value of x_8 back into step (c) to obtain

$$x_4 = 1 \tag{9.30}$$

Then, substituting the solutions for x_4 and x_8 back into step (b) to obtain

$$x_2 = 1 \quad and \quad x_6 = 1 \tag{9.31}$$

Finally, substituting all "even" solutions (x_2, x_4, x_6, and x_8) back into step (a) to obtain

$$x_1 = x_3 = x_5 = x_7 = 1 \tag{9.32}$$

A general formula for cyclic reduction of systems Tx = y with a tridiagonal matrix T can now be derived. Using the following elementary row operations:

$$-\alpha_{2i} * Row(2i-1) + Row(2i) - \beta_{2i} * Row(2i+1)$$

with $\alpha_{2i} = b_{2i}/a_{2i-1}$ and $\beta_{2i} = c_{2i}/a_{2i+1}$. Then the modified tridiagonal system of equations becomes:

$$\begin{aligned} &-(b_{2i-1}\alpha_{2i}) * x_{2i-2} + (a_{2i} - c_{2i-1}\alpha_{2i} - b_{2i+1}\beta_{2i})x_{2i} - c_{2i+1}\beta_{2i} * x_{2i+2} \\ &= y_{2i} - \alpha_{2i}y_{2i-1} - \beta_{2i}y_{2i+1} \quad for \quad i = 1, 2, \ldots, 2^{k-1} \end{aligned} \tag{9.33}$$

where $b_1 = c_n = 0$. In general, each step of a cyclic reduction reduces a (2**k) * (2**k) system to one of size 2**(k-1) * 2 ** (k-1), and after k steps we obtain one equation for the unknown.

As an example, for i = 1, then Eq. 9.33 becomes

$$-(b_1\alpha_2)x_0 + (a_2 - c_1\alpha_2 - b_3\beta_2)x_2 - c_3\beta_2 x_4 = y_2 - \alpha_2 y_1 - \beta_2 y_3$$

where:

$$\alpha_2 = \frac{b_2}{a_1}, \quad and \quad \beta_2 = \frac{c_2}{a_3}$$

Substituting the numerical values (using the data shown in Eq. 9.28) into Eq. 9.33, one obtains

$$(1)x_2 - (0.5\ x_4) = 0.5 \tag{9.34}$$

For i = 2, then Eq. 9.33 becomes

$$-(b_3\,\alpha_4)\,x_2 + (a_4 - c_3\,\alpha_4 - b_5\,\beta_4)\,x_4 - c_5\,\beta_4\,x_6 = y_4 - \alpha_4\,y_3 - \beta_4\,y_5$$

where:

$$\alpha_4 = \frac{b_4}{a_3}, \quad and \quad \beta_4 = \frac{c_4}{a_5}$$

Using the data shown in Eq. 9.28, one obtains

$$-0.5x_2 + 1x_4 - 0.5x_6 = 0 \qquad \textbf{(9.35)}$$

Similarly, for i = 4, then Eq. 9.33 becomes:

$$-[\,b_7\,\alpha_8\,]\,x_6 + [\,a_8 - c_7\,\alpha_8 - b_9\,\beta_8\,]\,x_8 - [c_9\,\beta_8\,]\,x_{10} = y_8 - \alpha_8\,y_7 - \beta_8\,y_9$$

where:

$$\alpha_8 = \frac{b_8}{a_7}, \quad and \quad \beta_8 = \frac{c_8}{a_9}$$

Substituting the numercial values (using the data shown in Eq. 9.28) into Eq. 9.33 one obtains:

$$-\left[(-1)*\left(\frac{-1}{2}\right)\right]x_6 + \left[1-(-1)\left(\frac{-1}{2}\right)-(0)(\beta_8)\right]x_8 = 0$$

$$-0.5x_6 + 0.5x_8 = 0 \qquad \textbf{(9.36)}$$

Eqs. 9.34 through 9.36 are exactly the same as the numerical results presented in step (b).

Some Remarks on Cyclic Reduction Algorithm:

(1) The operations involved in steps a, b, c, and d of the cyclic reduction algorithm are not recursive, they are essentially independent. Thus, better vector speed can be expected in the cyclic reduction algorithm.

(2) During the cyclic reduction steps, the stride (distance between two consecutive numbers) becomes larger and larger. Also, the vector lengths become shorter and shorter.

(3) Total number of operations is approximately in the order of (17*n) operations (see Homework Problem No. 9.2).

(4) Memories are required to store, for example, 8 x 8, then 4 x 4, then 2 x 2, etc. . . . reduced system of equations.

(5) Communications are needed whenever a row is updated by row(s) from other processors.

(6) Special attentions are required for handling multiple right-hand-side vectors.

9.4 Parallel Tridiagonal Solver by Using Divided and Conquered Strategies

To facilitate the discussions in this section, a tridiagonal (coefficient) matrix [T], with n = 16 degree-of-freedom (dof) is shown in the following equations

$$[T] = \begin{array}{c|cccccccccccccccc} & 1 & 2 & 3 & 4 & & 6 & & 8 & & 10 & & 12 & & 14 & & 16 \\ 1 & a_1 & c_1 & & & & & & & & & & & & & & \\ 2 & b_2 & a_2 & c_2 & & & & & & & & & & & & & \\ 3 & & b_3 & a_3 & c_3 & & & & & & & & & & & & \\ 4 & & & b_4 & a_4 & c_4 & & & & & & & & & & & \\ 5 & & & & b_5 & a_5 & c_5 & & & & & & & & & & \\ 6 & & & & & b_6 & a_6 & c_6 & & & & & & & & & \\ 7 & & & & & & b_7 & a_7 & c_7 & & & & & & & & \\ 8 & & & & & & & b_8 & a_8 & c_8 & & & & & & & \\ 9 & & & & & & & & b_9 & a_9 & c_9 & & & & & & \\ 10 & & & & & & & & & b_{10} & a_{10} & c_{10} & & & & & \\ 11 & & & & & & & & & & b_{11} & a_{11} & c_{11} & & & & \\ 12 & & & & & & & & & & & b_{12} & a_{12} & c_{12} & & & \\ 13 & & & & & & & & & & & & b_{13} & a_{13} & c_{13} & & \\ 14 & & & & & & & & & & & & & b_{14} & a_{14} & c_{14} & \\ 15 & & & & & & & & & & & & & & b_{15} & a_{15} & c_{15} \\ 16 & & & & & & & & & & & & & & & b_{16} & a_{16} \end{array} \quad \textbf{(9.37)}$$

Assuming there are four processors (NP = 4) available, and processors P_1, P_2, P_3 and P_4 store rows number 1 through 4, numbers 5 through 8, numbers 9 through 12, and numbers 13 through 16, respectively.

The parallel algorithms to solve tridiagonal system of equations can be conveniently described by the following step-by-step procedures:

Step 1(a):

Using elementary row operations to make the terms (b_2, b_3, b_4), (b_6, b_7, b_8), (b_{10}, b_{11}, b_{12}) and (b_{14}, b_{15}, b_{16}) become zeros (see Figure 9.1).

As an example, the terms b_2 through b_4 in Eq. 9.38 can be made to become zeros by using the following elementary row operations (by processor P_1):

"New" row 2 (with b_2 = 0) = "Old" pivot row 1 * (appropriate constant) + "Old" row 2

"New" row 3 (with b_3 = 0) = "New" pivot row 2 * (appropriate constant) + "Old" row 3

"New" row 4 (with b_4 = 0) = "New" pivot row 3 * (appropriate constant) + "Old" row 4

Simultaneously, processor P_4 can be used to perform elementary row operations to make the terms b_{14} through b_{16} in Eq. 9.38 to become zeros:

"New" row 14 (with $b_{14} = 0$) = "Old" pivot row 13 * (appropriate constant) + "Old" row 14
"New" row 15 (with $b_{15} = 0$) = "New" pivot row 14 * (appropriate constant) + "Old" row 15
"New" row 16 (with $b_{16} = 0$) = "New" pivot row 15 * (appropriate constant) + "Old" row 16

It is important to recognize that during the process to make the terms b_{14}, b_{15} and b_{16} to become zeros, three extra fill-in terms (see symbol F in Figure 9.1) are created.

Remarks:
(1) The elementary row operations need to be applied to the right-hand-side vector also. Thus, this algorithm is not efficient for multiple right-hand-side vectors.
(2) In this step, there are no communications among processors. Thus, 100% parallel computation can be achieved in this step.

Step 1(b)

Using elementary row operations to make the terms (c_{15}, c_{14}, c_{13}), (c_{11}, c_{10}, c_9), (c_7, c_6, c_5) and (c_3, c_2, c_1) become zeros (see Figure 9.2).

Remarks:
(1-2) Same remarks as have been mentioned in Step 1(a)
(3) Extra fills-in (see symbols F in Figure 9.2) are created during this process
(4) This step is quite similar to previous step 1(a).

Step 2:

Using elementary row operations to make the terms F_1, F_2, F_3, x_4, x_5, F_6, x_7, x_8, F_9, F_{10}, F_{11} and F_{12} (see Figure 9.3) to become zeros (according to the given orders F_1, then F_2, ... F_{12}).

Remarks:
(1) Some communications among the processors are required. For example, to make the term F_1 becomes zero (see Figure 9.3), one needs to perform the following elementary row operations
"New" row 13 (with $F_1 = 0$) = "Old" pivot row 12 * (appropriate constant) + "Old" row 13
Thus communications between processor P_3 and P_4 are necessary, since row 12 belongs to processor P_3 and row 13 belongs to processor P_4
(2) During the process to make F_1 term becomes zero, the extra fill-in term F_{12} is created. However, this newly created extra fill-in term F_{12} will also be made to zero (by using elementary row operations) at the end of this step!
(3) Using "New" row 13 (with $F_1 = 0$, and $F_{12} \neq 0$) as pivot row, the terms F_2 (in row 12) and F_3 (in row 9) can also be made to zeros (through elementary row operations).
(4) Then, using row 8 as a pivot row, the term x_4 (see Figure 9.3) can be made to zero. As a consequence, the extra fill-in term F_{10} is created. However, this newly created

term F_{10} will be made to zero, at a later time in this step.

(5) At the end of this step, one will obtain the matrix as shown in Figure 9.4

Step 3:

Using elementary row operations to make the terms (F_a, F_b), (F_c, F_d), (F_e, F_f, F_g), (F_h, F_i), (F_j, F_k) and (F_l, F_m, F_n) to become zeros (see Figure 9.4).

Remarks:

(1) Some communications among processors are required in this step. For example, to make the terms F_a and F_b (which belong to processor P_2) to become zeros, one needs to use pivot row number 4 (which belongs to processor P_1). Thus, some communications between P_1 and P_2 are necessary.

(2) The "orders" of those terms to be driven to zeros can be different. For example, the orders for the terms (F_a, F_b), (F_c, F_d) and (F_e, F_f, F_g) to be driven to zeros, can be changed into (F_e, F_f, F_g), (F_c, F_d) and (F_a, F_b).

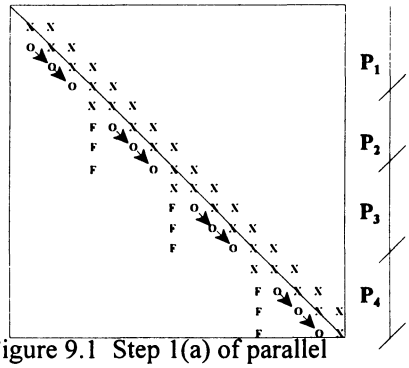


Figure 9.1 Step 1(a) of parallel divide & conquer

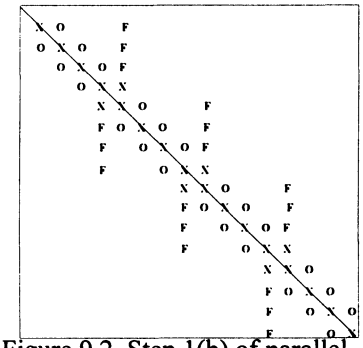


Figure 9.2 Step 1(b) of parallel divide & conquer

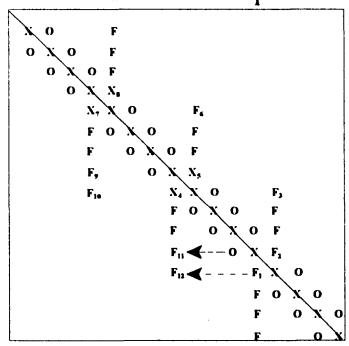


Figure 9.3 Step 2 of parallel divide and conquer

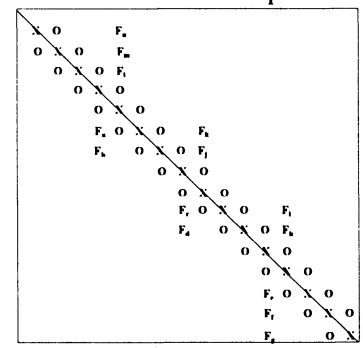


Figure 9.4 Step 3 of parallel divide and conquer

(3) At the end of this step, the matrix (shown in Figure 9.4) will become a diagonal matrix.

9.5 Parallel Factorization Algorithm for Tridiagonal System of Equations Using Separators

One way to do parallel computation for Eqs. 9.4 through 9.6, 9.9 through 9.12 is to uncouple the tridiagonal matrix T into T^*, so that the operations in Eqs. 9.4 through 9.6 are independent and can be done concurrently. We first define a separator as a diagonal element of T, say a_i ($1 < i < n$). The locations of the separators are determined so that they are equally distributed in T. Assuming NP processors are available, we need NP-1 separators to divide T into NP portions, so that each processor stores only one portion (each portion has roughly (n-NP)/NP equations) plus the NP-1 separators. The T^* matrix can be obtained from T matrix simply through relocating the rows and columns related to these NP-1 separators to the end of the matrix. In fact, for separator a_i, we only have to relocate four elements, i.e., c_i, b_i, b_{i+1} and c_{i-1}. For example, let NP = 4 and the NP-1 = 4 - 1 = 3 separators are located at i, j, and k, respectively. Then matrix T^* can be obtained by moving the i-th row and column to the (n+1)-th row and column, the j-th row and column to the (n+2)-th row and column, the k-the row and column to the (n+3)-th row and column, respectively. After renumbering, T^* will have the same size as T, as shown in Figure 9.5 (only the upper portion of T^* is shown here).

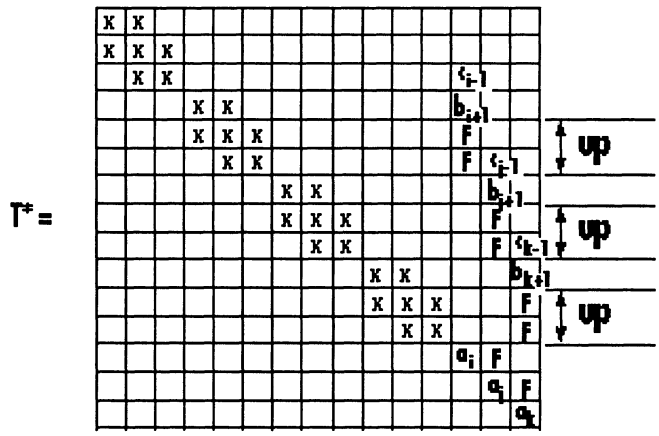


Figure 9.5 The uncoupled T* matrix

In Figure 9.5, F represents the fill-in elements in the LU factorization. The vectors "low" (not shown in Figure 9.5) and "up" are used to store these fill-in elements in L and U matrices, respectively. In practice, there is no need to renumber or to relocate the T matrix, the T^* matrix can be generated directly from the definitions. The LU factorization of T^* can be done in two steps:

a. LU factorization of the NP uncoupled portions can be done by the NP

processors concurrently without any communications. When NP > 1, the extra work required is to find out the fill-in elements.

b. When NP > 1, the LU factorization of the separators is done sequentially with communications among the NP processors.

As for the forward elimination, there are also two steps:

a. Forward elimination of the NP uncoupled portions can be done concurrently by the NP processors without any communications. When NP > 1, the extra work needed is to calculate the dot-product of the solution vector z and the fill-in vector "low".

b. When NP > 1, the solution portion corresponding to the separators can be found with communications among the NP processors.

Similarly, the backward substitution involves the following steps:

a. When NP > 1, find the solution portion corresponding to the separators first.

b. Backward substitution in the NP portions can be done concurrently without any communications. When NP > 1, the extra work to be done is the saxpy operations on the fill-in vector "up".

In order to better understand the above parallel algorithm, let us consider a 15 x 15 degree-of-freedom tridiagonal system as shown in Figure 9.6.

For parallel computational purposes, the coefficient tridiagonal matrix T should be partitioned (to simplify the discussions, assuming two processors are used) and factorized according to the following step-by-step procedure.

Step 1: Introducing the extra (artificial) degree-of-freedom number 16 (for matrix partitioning purposes) into the original tri-diagonal matrix T. The extra 16^{th} row and 16^{th} column have zero values everywhere, except 1 at the diagonal location (please refer to Figure 9.6)

Step 2: Switching row (and column) No. 8 with row (and column) No. 16 (please refer to Figure 9.7). Since there are 16 degree-of-freedom, and two processors are available, the separator should be approximately at degree-of-freedom No. 8.

Step 3: Removing the "artificial" 8^{th} row (and column). Thus, the final partitioned matrix can be shown in Figure 9.8. It should be noted here that if the tri-diagonal system is symmetric, then $c_1 = b_2$, $c_7 = b_8$, $c_8 = b_9$, etc..... Furthermore, there will be "fills-in" in the last column during the factorization. These "fills-in" are denoted by the symbol "F" in Figure 9.8. For a separator a_i (at location i = 8, as shown in Figure 9.7), then according to a more general case (as shown in Figure 9.5), c_{i-1}, (or c_7) and b_{i+1} (or b_9) terms need to be moved toward the end columns. These facts have been confirmed in Figure 9.8

$$
\begin{array}{cccccccccccccccc}
a_1 & c_1 & & & & & & & & & & & & & 0 & 1 \\
b_2 & a_2 & c_2 & & & & & & & & & & & & 0 & 2 \\
 & b_3 & a_3 & c_3 & & & & & & & & & & & 0 & 3 \\
 & & b_4 & a_4 & c_4 & & & & & & & & & & 0 & 4 \\
 & & & b_5 & a_5 & c_5 & & & & & & & & & 0 & 5 \\
 & & & & b_6 & a_6 & c_6 & & & & & & & & 0 & 6 \\
 & & & & & b_7 & a_7 & c_7 & & & & & & & 0 & 7 \\
 & & & & & & b_8 & a_8 & c_8 & & & & & & 0 & 8 \\
 & & & & & & & b_9 & a_9 & c_9 & & & & & 0 & 9 \\
 & & & & & & & & b_{10} & a_{10} & c_{10} & & & & 0 & 10 \\
 & & & & & & & & & b_{11} & a_{11} & c_{11} & & & 0 & 11 \\
 & & & & & & & & & & b_{12} & a_{12} & c_{12} & & 0 & 12 \\
 & & & & & & & & & & & b_{13} & a_{13} & c_{13} & 0 & 13 \\
 & & & & & & & & & & & & b_{14} & a_{14} & c_{14} & 0 & 14 \\
 & & & & & & & & & & & & & b_{15} & a_{15} & 0 & 15 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16
\end{array}
$$

Figure 9.6 Introducing extra degree-of-freedom # 16

$$
\begin{array}{cccccccccccccccc}
a_1 & c_1 & & & & & & & & & & & & & 0 & 1 \\
b_2 & a_2 & c_2 & & & & & & & & & & & & 0 & 2 \\
 & b_3 & a_3 & c_3 & & & & & & & & & & & 0 & 3 \\
 & & b_4 & a_4 & c_4 & & & & & & & & & & 0 & 4 \\
 & & & b_5 & a_5 & c_5 & & & & & & & & & 0 & 5 \\
 & & & & b_6 & a_6 & c_6 & & & & & & & & 0 & 6 \\
 & & & & & b_7 & a_7 & 0 & & & & & & & c_7 & 7 \\
 & & & & & & 0 & 1 & 0 & & & & & & 0 & 8 \\
 & & & & & & & 0 & a_9 & c_9 & & & & & b_9 & 9 \\
 & & & & & & & & b_{10} & a_{10} & c_{10} & & & & 0 & 10 \\
 & & & & & & & & & b_{11} & a_{11} & c_{11} & & & 0 & 11 \\
 & & & & & & & & & & b_{12} & a_{12} & c_{12} & & 0 & 12 \\
 & & & & & & & & & & & b_{13} & a_{13} & c_{13} & 0 & 13 \\
 & & & & & & & & & & & & b_{14} & a_{14} & c_{14} & 0 & 14 \\
 & & & & & & & & & & & & & b_{15} & a_{15} & 0 & 15 \\
0 & 0 & 0 & 0 & 0 & 0 & b_8 & 0 & c_8 & 0 & 0 & 0 & 0 & 0 & 0 & a_8 & 16
\end{array}
$$

Figure 9.7 After switching row (and column) # 8 with row (and column) # 16

$$
\begin{array}{cccccccccccccc}
a_1 & c_1 & & & & & & & & & & & & \\
b_2 & a_2 & c_2 & & & & & & & & & & & \\
 & b_3 & a_3 & c_3 & & & & & & & & & & \\
 & & b_4 & a_4 & c_4 & & & & & & & & & \\
 & & & b_5 & a_5 & c_5 & & & & & & & & \\
 & & & & b_6 & a_6 & c_6 & & & & & & & \\
 & & & & & b_7 & a_7 & & & & & & & c_7 \\
 & & & & & & & a_9 & c_9 & & & & & b_9 \\
 & & & & & & & b_{10} & a_{10} & c_{10} & & & & F \\
 & & & & & & & & b_{11} & a_{11} & c_{11} & & & F \\
 & & & & & & & & & b_{12} & a_{12} & c_{12} & & F \\
 & & & & & & & & & & b_{13} & a_{13} & c_{13} & F \\
 & & & & & & & & & & & b_{14} & a_{14} & c_{14} & F \\
 & & & & & & & & & & & & b_{15} & a_{15} & F \\
 & & & & & & b_8 & c_8 & & & & & & & a_8
\end{array}
$$

Figure 9.8 After deleting row (and column) # 8

Step 4: Assuming the tri-diagonal system is symmetrical (if it is not symmetrical, then the reader should refer to Chapter 8 for an unsymmetrical equation solver).

Based upon the pattern shown in Figure 9.8, and assuming two processors are used [i.e. processor one will store the first seven rows (or columns), and processor two will store the last eight rows (or columns)]. Then parallel factorization can be done more effectively in a column-by-column fashion (instead of row-by-row fashion).

In a column-by-column factorization strategy, even though column 7 has not been completely factorized, columns 8, 9 through 14 can still be proceeded independently for factorization. Only the last (or 15th) column factorization requires information on the factorization of column 7.

On the other hand, if factorization is conducted in a row-by-row fashion, then there will be more dependency on previous calculation and therefore, less parallel speed-ups can be expected. Referring to Figure 9.8, one can clearly see that processor 2 can not factorize row 8 (which contains the non-zero values of a_9, c_9, and b_9) unless row 7 (which contains the non-zero values of a_7 and c_7) has been completely factorized by processor 1.

Step 5: It should be emphasized at this time that factorization of columns 1 through 7 (by processor 1), and columns 8 through 14 (by processor 2) can be done concurrently without any communication required. However, factorization of the last (or 15^{th}) column by processor 2 will require some communication, since the factorized column 7 (possessed by processor 1)

is required.

Using the same 15 x 15 tridiagonal matrix (as shown in Figure 9.6) and assuming three processors (or NP = 3) are used (hence 2, or NP-1 separators are required), then each processor should have the following work loads

$$\frac{15 - (NP - 1)}{NP} = \frac{15 - 2}{3} = 4.333 \quad \text{rows (or columns) per processor.}$$

Thus, the work load partitioning for each processor should be:

Processor 1: row (or columns) 1 through 4
Processor 2: row (or columns) 6 through 9
Processor 3: row (or columns) 11 through 15

Rows (or columns) 5 and 10 are used as processor separators and these diagonal values are possessed by **ALL** processors.

The original 15 x 15 tridiagonal matrix can, therefore be partitioned as shown in Figure 9.9, or Figure 9.10 if one introduces (then removes) these two extra rows (and columns).

In actual computer implementation, there is no need to introduce (and then remove) the extra rows/columns as shown in Figure 9.9. Instead, Figure 9.9 can be directly and efficiently generated as shown in Figure 9.11. Assuming the tri-diagonal system is symmetrical, then processors one, two and three can be used to independently factorize columns 1 through 4, 5 through 8 and 9 through 13, respectively. Factorizing the last two columns (column numbers 14 and 15) will require some communications among processors.

In terms of storage assignments to different processors, processors one, two and three will store rows 1 through 4, 5 through 8, and 9 through 13, respectively. The two diagonal terms a_{10} and a_5 (see Figure 9.11) are stored by <u>all</u> processors. It should be mentioned here again, that Figure 9.11 has the same pattern as shown in the general Figure 9.5.

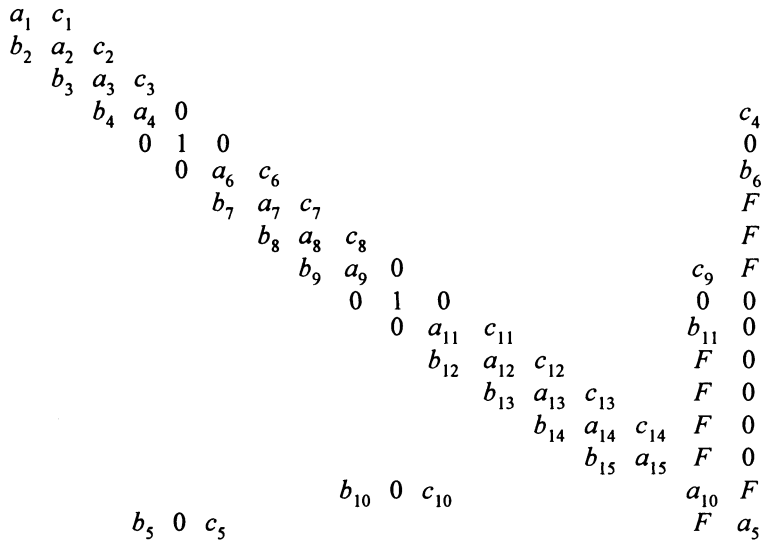


Figure 9.9 Row 5 becomes last row (or row n = 17); row 10 becomes next to last row (or row n-1 = 16); total "fills-in" = 8

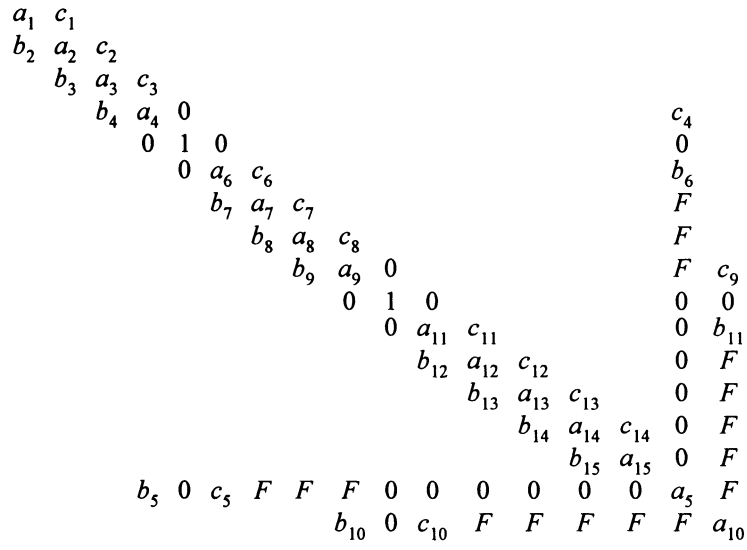


Figure 9.10 Row 5 becomes row n-1=16; row 10 becomes row n = 17; total "fills-in" = 8

$$
\begin{array}{cccccccccccccccc}
a_1 & c_1 & & & & & & & & & & & & & & c_4 \\
b_2 & a_2 & c_2 & & & & & & & & & & & & & b_6 \\
 & b_3 & a_3 & c_3 & & & & & & & & & & & & F \\
 & & b_4 & a_4 & 0 & & & & & & & & & & & F \\
 & & & 0 & a_6 & c_6 & & & & & & & & & c_9 & F \\
 & & & & b_7 & a_7 & c_7 & & & & & & & & b_{11} & 0 \\
 & & & & & b_8 & a_8 & c_8 & & & & & & & F & 0 \\
 & & & & & & b_9 & a_9 & 0 & & & & & & F & 0 \\
 & & & & & & & 0 & a_{11} & c_{11} & & & & & F & 0 \\
 & & & & & & & & b_{12} & a_{12} & c_{12} & & & & a_{10} & F \\
 & & & & & & & & & b_{13} & a_{13} & c_{13} & & & F & 0 \\
 & & & & & & & & & & b_{14} & a_{14} & c_{14} & & F & 0 \\
 & & & & & & & & & & & b_{15} & a_{15} & & F & 0 \\
 & & & & & & & b_{10} & c_{10} & F & F & F & F & a_{10} & F & F \\
 & & b_5 & c_5 & F & F & F & & & & & & & & F & a_5
\end{array}
$$

Figure 9.11 Directly partitioning (without introducing artificial rows 16 and 17)

For structural applications (tri-diagonal, or block tridiagonal matrices), since the matrix is generally symmetric, one can minimize the amounts of "fills-in" by adopting the following numbering scheme for the cantilever beam (shown in Figure 9.12)

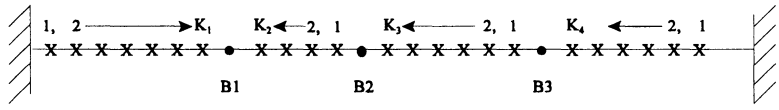


Figure 9.12 Special numbering scheme to minimize "fill ins"

In Figure 9.12, assuming four processors are used, and processors P_1, P_2, P_3 and P_4 will store K_1, K_2, K_3 and K_4 interior nodes, respectively. The nodes B_1, B_2 and B_3 represent boundary nodes (nodes which belong to two or more processors). Using the partitioning scheme discussed in the earlier sections, the partitioned matrix corresponds to Figure 9.12 can be given as shown in Figure 9.13

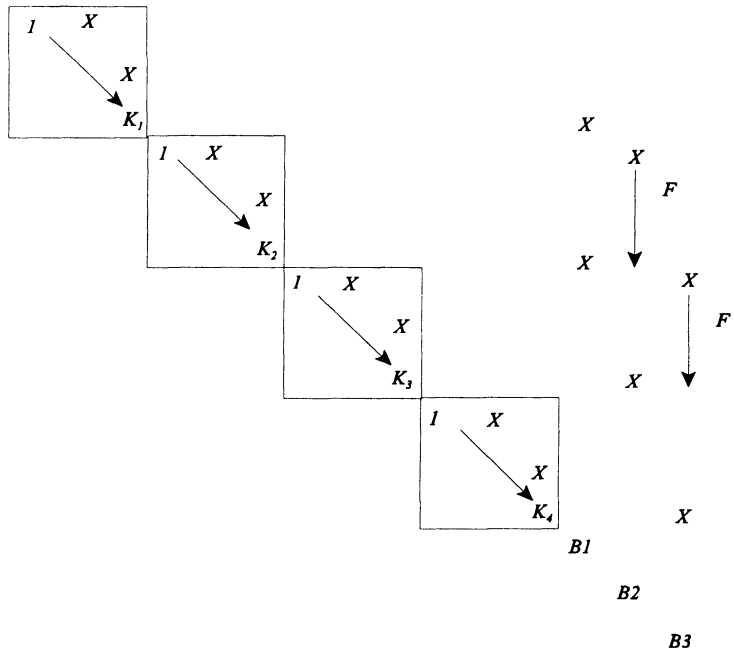


Figure 9.13 Minimizing "fill-in" of symmetrical tri-diagonal system by using proper node numbering system

9.6 Forward and Backward Solution Phases

To simplify the discussion, assuming we have a symmetrical tri-diagonal system. Thus, in Figure 9.11, the upper tridiagonal matrix is merely an image of the lower tridiagonal matrix. Furthermore, for the forward/backward solution phases, Figure 9.11 should be used only for the purpose of identifying the locations of non-zeros and "fills-in".

9.6.1 Forward solution phase: [L] {z} = {y}

Considering Figure 9.11, with the assumption that all terms in the upper tridiagonal portion are zero. It can be seen obviously that the unknowns z_1 through z_4, z_5 through z_8 and z_9 through z_{13} can be found concurrently by NP (=3) processors. There is no communication required among processors for solving the first thirteen unknowns. However, the last two unknowns in the forward solution phase does need communications among processors. For example, to solve for the unknown z_{14}, one has

$$z_{14} = \frac{y_{14} - \left(\bar{b}_{10} z_8 + \bar{c}_{10} z_9 + \bar{F} z_{10} + \bar{F} z_{11} + \bar{F} z_{12} + \bar{F} z_{13}\right)}{\bar{a}_{10}} \qquad \textbf{(9.38)}$$

In Eq. 9.38, $\bar{b}_{10}$, $\bar{c}_{10}$, $\bar{a}_{10}$, *and* $\bar{F}$ represent the non-zero values of the

factorized tridiagonal matrix [T]. In actual computer implementation, only the upper triangular portion of the matrix T is computed and stored (in a column-by-column fashion). Thus, the operations shown in the parenthesis of Eq. 9.38 is basically involved with the dot product of the two vectors

$$\begin{Bmatrix} \bar{b}_{10} \\ \bar{c}_{10} \\ \bar{F} \\ \bar{F} \\ \bar{F} \\ \bar{F} \end{Bmatrix} \cdot \begin{Bmatrix} z_8 \\ z_9 \\ z_{10} \\ z_{11} \\ z_{12} \\ z_{13} \end{Bmatrix}$$

or, to be more precise (in actual computer implementation)

$$\begin{Bmatrix} \bar{c}_9 \\ \bar{b}_{11} \\ \bar{F} \\ \bar{F} \\ \bar{F} \\ \bar{F} \end{Bmatrix} \cdot \begin{Bmatrix} z_8 \\ z_9 \\ z_{10} \\ z_{11} \\ z_{12} \\ z_{13} \end{Bmatrix}$$

where again, the "over-bar" notations in the first vector symbolically represent the non-zero values of [T] after factorization.

Similarly, the last unknown (z_{15}) to be solved during the forward solution phase can be given as (please refer to Figure 9.11)

$$z_{15} = \frac{y_{15} - \left(\bar{b}_5 z_4 + \bar{c}_5 z_5 + \bar{F} z_6 + \bar{F} z_7 + \bar{F} z_8 \right)}{\bar{a}_5} \qquad \textbf{(9.39)}$$

Again, the operations shown in the parenthesis of Eq. 9.39 is involved with the dot product of two vectors

$$\begin{Bmatrix} \bar{b}_5 \\ \bar{c}_5 \\ \bar{F} \\ \bar{F} \\ \bar{F} \end{Bmatrix} \cdot \begin{Bmatrix} z_4 \\ z_5 \\ z_6 \\ z_7 \\ z_8 \end{Bmatrix}$$

or, to be more precise (in actual computer implementation)

$$\begin{Bmatrix} \bar{c}_4 \\ \bar{b}_6 \\ \bar{F} \\ \bar{F} \\ \bar{F} \end{Bmatrix} \cdot \begin{Bmatrix} z_4 \\ z_5 \\ z_6 \\ z_7 \\ z_8 \end{Bmatrix}$$

where again, the "over-bar" notations in the first vector symbolically represent the non-

zero values of [T] after factorization.

9.6.2 Backward solution phase: [U] {x} = {z}

Considering Figure 9.11, with the assumption that all terms in the lower triangular portion are zero.

The last two unknowns (involved with the separators) can be solved first as follows:

$$x_{15} = \frac{z_{15}}{\bar{a}_5} \tag{9.40}$$

where the "over-bar" notation appeared in the denominator of Eq. 9.40 symbolically represents the non-zero value [U] after factorization.

Having solved for the unknown x_{15}, the next unknown (x_{14}) can be computed as

$$x_{14} = \frac{z_{14}}{\bar{a}_{10}} \tag{9.41}$$

However, in actual computer implementation, the new right-hand-side vector {z} will be updated right after each unknown is solved. As an example, having solved for the unknown x_{15} (according to Eq. 9.40), the right-hand-side vector {z} can be updated as (please refer to Figure 9.11):

$$\{\bar{z}\}_{new} = \{z\}_{old} - x_{15} \begin{Bmatrix} 0 \\ 0 \\ 0 \\ \bar{c}_4 \\ \bar{b}_6 \\ \bar{F} \\ \bar{F} \\ \bar{F} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix} \tag{9.42}$$

The operations involved in Eq. 9.42 is called Saxpy (Summation of $\mathbf{a}\{\mathbf{x}\}+\{\mathbf{y}\}$) operations and can be done very fast on vector computers (such as the Cray-YMP, or Cray-C90 supercomputers).

Similarly, having solved for the unknown x_{14} (according to Eq. 9.41), the right-hand-side vector can be updated again as (please refer to Figure 9.11):

$$\{\bar{z}^*\}_{new} = \{\bar{z}\}_{new} - x_{14} \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \bar{c}_9 \\ \bar{b}_{11} \\ \bar{F} \\ \bar{F} \\ \bar{F} \\ \bar{F} \end{Bmatrix} \tag{9.43}$$

The remaining unknowns x_{13} through x_9, x_8 through x_5 and x_4 through x_1 can be solved independently (please refer to Figure 9.11) by NP (=3) processors without any processor communications involved.

9.7 Comparisons between Different Algorithms

The price paid for the parallel algorithm is the increased number of operations and the communications among the processors. First, 3*n extra operations are needed in the LU factorization to compute the fill-in elements (see Homework Problem 9.3). Then 4*n extra operations are needed for forward/backward substitutions (see Homework Problem 9.4). The total operations counts is 16*n, compared with the 9*n operations for the sequential Gaussian elimination. One may expect a maximum speed up of (9/16)*NP when NP processors are used. However, higher speedup is possible since some of the extra operations can be well vectorized, this can be seen in the following paragraphs.

The implementation of the parallel algorithm in Sections 9.5 and 9.6 can be done differently depending on the vector performance of the computers. For example, the LU factorization within each portion can be done by standard Gaussian elimination (Eqs. 9.4 through 9.6) or by cyclic reduction. Since we intend to develop a tridiagonal solver for multiple right-hand-side vectors, we focus on the performance of the forward backward substitution rather than on the LU factorization. Even though the Eqs. 9.9 through 9.12 are recursive, they can be executed at a rate of about 5 Mflops on an Intel iPSC/860 processor using single precision. Any other method seems to double the operation counts, so it needs a rate of 10 Mflops or higher to justify its use. Reference [9.14] gives a formulation for forward elimination by cyclic reduction that needs 5*n operations. Table 9.1 gives the performances of the Cyclic Reduction and the Gaussian elimination on Cray Y-MP and Intel iPSC/860, respectively.

Table 9.1 Cyclic reduction vs Gaussian elimination*

Computer	Cyclic Reduction	Gaussian Elimination
Cray Y-MP	8.309×10^{-3}	1.273×10^{-2}
iPSC/860	2.499×10^{-1}	5.713×10^{-2}

* bidiagonal matrix with n = 131072

It can be seen from Table 9.1 that cyclic reduction is faster than the serial Gaussian elimination on the Cray Y-MP, but not on the Intel iPSC/860. Thus, the implementation of the parallel algorithm on the Intel iPSC/860 computers can be described as (see Ref. 9.15):

1) The LU factorization is done by Gaussian elimination (Eqs. 9.4 through 9.6). The LU factorization of the separators is done by processor Zero, which receives necessary information from all the other processors, then the factored separators are sent to all the other processors.
2) The forward elimination is done using Eqs. 9.9 and 9.10, and each processor will find the solutions corresponding to the separators by passing information to each other.
3) Since all the processors have the solutions corresponding to the separators, the backward substitution can be done concurrently without any communications.

This parallel algorithm[9.15] requires 16*n operations and 5*NP communications, in which 7*n operations and 3*NP communications are needed for factorization.

9.8 Numerical Results

Two examples are shown in this section to demonstrate the efficiency of the present tridiagonal solver. The timings for one processor are corresponding to 9*n operations.

Example 9.1: The (unsymmetrical) tridiagonal systems to be solved have the following coefficients:

$a_i = 10$, $b_i = 2$, $c_i = 1$, $f_i = a_i + b_i + c_i$ $(i = 1, 2, 3, \ldots, n)$

$b_1 = 0$, $c_n = 0$,

$n = 38{,}400{,}000$

so that the solutions will be $x_i = 1$ (i = 1, 2, 3, . . . , n)

Table 9.2 gives the timings for solving this problem on NP processors. Since NP = 128 processors are needed to solve this problem, the timings for NP < 128 are for problem size of n = 300,000*NP, where 300,000 is the largest problem size which can be solved by a single processor.

Table 9.2 Timings for parallel solutions on Intel iPSC/860 (single precision)

Processors	1	2	4	8	16	32	64	128
Factorization	.362	.509	.510	.511	.512	.515	.5190	.526
Forward	.124	.156	.158	.158	.159	.160	.1629	.170
Backward	.216	.216	.216	.217	.217	.218	.2191	.219

Example 9.2: This tridiagonal systems of linear equations come from the finite element model of the one-dimensional truss, as shown in Figure 9.14. The solid dots, shown in Figure 9.14, represent the elements which are used as the separators in the solution. The structural parameters are:

E (Young Modulus) = 29,000 (k/in^2), A(area) = 4(in^2), L (Length) = 240 (in), and the load y = 10 (kips) acting on the last element.

There are totally 150,000*NP (i.e. n = 150,000*NP) one-dimensional truss elements. The resulted tridiagonal equations are symmetric but not diagonally dominant:

$$a_i = \frac{EA}{(Ln)}, \; b_i = c_i = -0.5\, a_i \qquad (i = 2, 3, 4, \ldots, n-1)$$

$$b_1 = c_n = 0, \; b_n = a_2, \; a_1 = a_2, \; c_1 = c_2, \; a_n = 0.5\, a_2$$

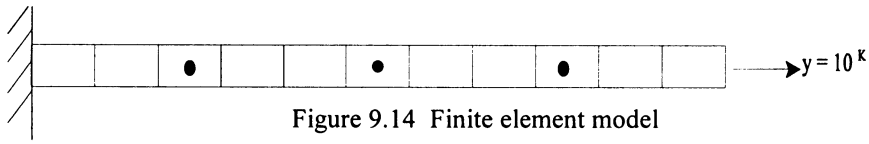


Figure 9.14 Finite element model

Table 9.3 Presents the timings for solving this problem, using up to 128 processors.

Table 9.3 Timings for solving example 9.2 (using double precision)

Processors	1	2	4	8	16	32	64	128
Factorization	.284	.366	.366	.367	.369	.372	.376	.387
Forward	.082	.108	.108	.109	.109	.111	.114	.121
Backward	.116	.127	.127	.127	.134	.136	.128	.129

9.9 A FORTRAN Call Statement to Subroutine Tridiag

Assuming that the original tridiagonal equations has been partitioned for parallel processing, the following FORTRAN subroutine will be called by each processor to simultaneously complete the computation.

Subroutine tridiag (iam, np, n, b, a, c, f, up, low, bond, bonu, bonl)
where:

iam	=	The identification number of each processor ($0 \le$ iam $\le$ np-1). This is an input variable
np	=	The number of processors. This is an input variable.
n	=	The size of the partition in each processor. This number can be different for different processors
b, a, c	=	Vectors (each has length n) to store the tridiagonal coefficient matrix. These are input vectors
f	=	Vector of length n+(np-1) to store the right-hand-side vector. This is an input vector
up, low	=	Vectors (each has length n) to store the upper (= up) and lower (= low) parts of fills-in, respectively. The array low should be declared as a real array. These are both input and output vectors
bond	=	Vector of length (np-1) to store the diagonals of the separators. This is an input vector
bonu, bonl	=	Vectors (each has length np-2) to store the upper (= bonu) and lower (= bonl) parts of fills-in around the separators, respectively. These are output variables. Referring to Figure 9.11, one should realize that vectors bonu, bonl and the diagonal of the separators (= a_{10}, a_5) together will also have the tridiagonal form

As an example, the matrix data shown in Figure 9.11 will be used to prepare for the subroutine tridiag.

For Processor P_0 (iam = 0, np = 3, n =4)

i^{th} location	**1**	**2**	**3**	**4**	**5**	**6**
b(i)	b_5	b_2	b_3	b_4		
a(i)	a_1	a_2	a_3	a_4		
c(i)	c_1	c_2	c_3	c_4		
f(i)	f_1	f_2	f_3	f_4	f_{14}	f_{15}
up(i)	0.	0.	0.	0.		
low(i)	0.	0.	0.	0.		
bond(i)	a_{10}	a_5				
bonu(i)	F		(Output array)			
bonl(i)	F		(Output array)			

For Processor P_1 (iam = 1, np = 3, n = 4)

i^{th} location	**1**	**2**	**3**	**4**	**5**	**6**
b(i)	b_{10}	b_7	b_8	b_9		
a(i)	a_6	a_7	a_8	a_9		
c(i)	c_6	c_7	c_8	c_9		
up(i)	b_6	0.	0.	0.		
low(i)	c_5	0.	0.	0.		
bond(i)	a_{10}	a_5				
bonu(i)	F		(Output array)			
bonl(i)	F		(Output array)			

For Processor P_2 (iam = 2, np = 3, n = 5)

i^{th} location	**1**	**2**	**3**	**4**	**5**	**6**
b(i)	0.	b_{12}	b_{13}	b_{14}	b_{15}	
a(i)	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	
c(i)	c_{11}	c_{12}	c_{13}	c_{14}	0.	
up(i)	b_{11}	0.	0.	0.	0.	
low(i)	c_{10}	0.	0.	0.	0.	
bond(i)	a_{10}	a_5				
bonu(i)	F	(Output array)				
bonl(i)	F	(Output array)				

For the complete listing of the FORTRAN source codes, instructions on how to incorporate this equation solver package into any existing application software (on any specific computer platform), and/or the complete consulting service in conjunction with this equation solver etc..., the readers should contact:

Prof. Duc T. Nguyen
Director, Multidisciplinary Parallel-Vector Computation Center
Civil & Environmental Engineering Dept.
Old Dominion University
Room 135, Kaufman Building
Norfolk, VA 23529 (USA)

Tel= (757) 683-3761, Fax = (757) 683-5354
Email= dnguyen@odu.edu

9.10 Summary

A parallel tridiagonal solver has been developed for solving large systems of linear equations on massively parallel computers. The FORTRAN implementation of this solver on Intel iPSC/860 computers is presented. Comparing with the standard sequential Gaussian elimination, this proposed parallel solver only requires 50% more memory. More importantly, there is no need to rearrange the data, and this feature makes the implementation easier on parallel computers. Even though the theoretical maximum speedup for the proposed parallel solver is (9/16)*NP, the practical speedup can be very close to NP since the extra operations can be executed much faster in a vector computer environment. More careful optimization of the code, such as using the 4K cache available on Intel iPSC/860 processors properly, will further improve the performance of the proposed solver.

9.11 Exercises

9.1 For $[A]_{n\times n}\{x\}_{n\times 1} = \{f\}_{n\times 1}$

where:

n	= 12					
a_1	= a_2	=	. . .	a_{12}	=	10
c_1	= c_2	=	. . .	c_{11}	=	+1
b_2	= b_3	=	. . .	b_{12}	=	-1

$$\{f\} = \begin{Bmatrix} 11 \\ 10 \\ 10 \\ \cdot \\ \cdot \\ \cdot \\ 10 \\ 9 \end{Bmatrix} \quad \text{and} \quad [A] = \begin{bmatrix} a_1 & c_1 & & & & & & \\ b_2 & a_2 & \cdot & & & & & \\ & \cdot & \cdot & \cdot & & & & \\ & & \cdot & \cdot & \cdot & & & \\ & & & \cdot & \cdot & \cdot & & \\ & & & & \cdot & \cdot & c_{11} & \\ & & & & & b_{12} & a_{12} & \end{bmatrix}$$

Using a hand calculator, performing parallel divide and conquer algorithm discussed in the text to find the solution for $\{\vec{x}\}$

Hint: $\bar{x} = \begin{Bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ \cdot \\ \cdot \\ \cdot \\ 1 \end{Bmatrix}$

9.2 Prove that the number of operations in the Cyclic Reduction algorithm is approximately in the order of 17*n, where n is the size of the tridiagonal system

9.3 Prove that 3*n "extra" operations are needed in the LU factorization to compute the fills-in elements (as shown in Figure 9.5)

9.4 Prove that 4*n "extra" operations are needed in the forward & backward solution phases (as described in Section 9.6)

9.5 Using the data shown in Figure 9.8, and assuming two processors are available (NP = 2). Prepare the necessary data for Subroutine TRIDIAG (as explained in Section 9.9)

9.12 References

9.1 Yoshihara, H. ed. "Computational Fluid Dynamics: Algorithms & Supercomputers, AGARD-AG-311", 1988.

9.2 Hockney, R.W., "Rapid Elliptic Solvers, in Numerical Methods in Applied Fluid Dynamics", ed. by B. Hunt, Academic Press, New York, 1980.

9.3 Miranker, W., "A Survey of Parallelism in Numerical Analysis", SIAM Rev. 13 (1971) 524-547.

9.4 Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra", SIAM Rev., 20 (1978) 740-777.

9.5 Ortega, J.M. and R.G. Voigt, "Solution of Partial Differential Equations on Vector and Parallel Computers", SIAM Rev., 27 (1985) 149-240.

9.6 Hockney, R., "A Fast Direct Solution of Poison's Equation Using Fourier Analysis", J.ACM, 12 (1965) 95-113.

9.7 Lambiotte, J. and R.G. Voigt, "The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computers", ACM Trans. Math. Software 1 (1975) 308-329.

9.8 Kershaw, D., "Solution of Single Tridiagonal Linear Systems and Vectorizations of the ICCG Algorithm on the Cray-1, in Parallel Computations", ed. by Garry Rodrigue, Academic Press. New York, 1982.

9.9 Madsen, N. and G. Rodrigue, "A Comparison of Direct Methods for Tridiagonal Systems on the CDC STAR-100", Lawrence Livermore National Laboratory, Preprint UCRL-76993, Rev. 1, Livermore, CA 1976.

9.10 Reale, F, "A Tridiagonal Solver for Massively Parallel Computer Systems", Parallel Computing 16 (190) 365-368.

9.11 Krechel, A., H.J. Plum and K. Stuben, "Parallelization and Vectorization Aspects of the Solution of Tridiagonal Linear Systems", Parallel Computing 14 (1990) 31-49.

9.12 Hajj, I.N. and S. Skelboe, "A Multilevel Parallel Solver for Block Tridiagonal and Banded Linear Systems", Parallel Computing 15 (1990) 21-45.

9.13 Gentzsch, W. K.W. Neves and H. Yoshihara, AGARD AGARDOGRAPH NO. 311, Computational Fluid Dynamics: Algorithms and Supercomputers (March 1988).

9.14 Axelsson, O. and V. Eijkhout, "A note on the Vectorization of Scalar Recursions", Parallel Computing 3 (1) (1986) 73-84.

9.15 Qin, J. and D.T. Nguyen, "A Tridiagonal Solver for Massively Parallel Computers," Advances in Engineering Software, Vol. 29, No. 3-6, pp. 395-397 (1998).

10 Sparse Equation Solver with Unrolling Strategies

10.1 Introduction

The solution of linear systems of equations on advanced parallel and/or vector computers is an important area of ongoing research. The development of efficient equation solvers is particularly important for static and dynamic (linear and non-linear) structural analyses, sensitivity and structural optimization, control-structure interactions, ground water flows, panel flutters, eigenvalue analysis etc.... [10.1-10.19]. Modern high-performance computers (such as Cray-YMP, Cray-C90, Intel Paragon, IBM-SP2) have both parallel and vector capability, thus algorithms that exploit parallel and/or vector capabilities are the most desirable.

In the past years, a lot of efforts have been devoted in the developments of efficient parallel and vector equation solvers on both shared and distributed memory computers which exploit the skyline and/or variable bandwidth of the coefficient matrix. On a single node computer processor with vectorized capability, however, it is generally safe to say that equation solvers which are based on sparse technologies are more efficient than ones which are based on skyline and/or variable bandwidth storage schemes [10.20-10.23]. Basic equation solution algorithms based on sparse technologies have been well documented in the literatures [10.20-10.23]. Few, limited research efforts have also been directed to the development of parallel sparse equation solvers [10.24-10.25]. In this chapter, however, emphasis will be placed on the development of efficient, fully vectorized sparse equation solver for single processor computers with vectorized capability (such as the Cray-YMP, Cray-C90, Intel Paragon, IBM-SP2, IBM-R6000/590 workstations, etc...).

Basic Choleski and LDL^T algorithms are briefly reviewed in Section 10.2. Different storage schemes for the coefficient matrix are presented in Section 10.3. Popular reordering algorithms are mentioned in Section 10.4. Sparse symbolic factorization is discussed in Section 10.5. Sparse numerical factorization and forward/backward solution phases are explained in Sections 10.6 and 10.7, respectively. Loop unrolling strategies to optimize the vector speed are introduced in Section 10.8. Numerical evaluations of the developed software are demonstrated in Section 10.9 through practical finite element models, such as 23155 degree-of-freedom (dof) Exxon Offshored Structure, 16146 dof High Speed Civil Transport (HSCT) aircraft, 55000 dof

Solid Rocket Booster (SRB) of the space shuttle and 256000 dof of an automobile. FORTRAN calls to sparse equation solver is explained in Section 10.10. Finally, conclusions are drawn in Section 10.11.

10.2 Basic Equation Solution Algorithms

The key to reduce the computation time for structural analysis is to reduce the time to solve the resulting linear system of equations. Using matrix notations, the linear system of equations can be conveniently expressed as

$$[K]\{z\} = \{f\} \tag{10.1}$$

For many engineering applications, the coefficient (stiffness) matrix [K] often has nice properties, such as symmetry, positive definite and sparse. In Eq. (10.1), the vectors {z} and {f} represent the unknown nodal displacement, and the known nodal load vectors, respectively.

10.2.1. Choleski algorithm

On sequential computers, direct methods based on Choleski algorithm are both accurate and fast in solving a wide range of structural analysis problems. These methods are sued in most commercial finite element codes. Choleski-based methods have also been found to be accurate and fast in solving structural analysis problems on parallel computers. The unknown vector {z} can be found in three distinct steps:

First Step: Factorization Phase
In this step, the coefficient matrix [K] can be factorized as

$$[K] = [U]^T [U] \tag{10.2}$$

where [U] is an upper triangular matrix

Second Step: Forward Solution Phase
One can substitute Eq. (10.2 into Eq. (10.1) to obtain

$$[U]^T \{y\} = \{f\} \tag{10.3}$$

where the intermediate unknown vector {y} can be readily identified as

Third Step: Backward Solution Phase

$$[U]\{z\} \equiv \{y\} \tag{10.4}$$

Having obtained the solution for the unknown vector {y} from Eq. (10.3), the original unknown vector {z} can be obtained by solving Eq. (10.4). For a single right-hand-side vector {f}, 95% (or more) of the total equation solution time will be spent in the first step. Thus, in this work, more emphasis will be placed on the development of efficient factorization schemes, which can fully exploit the vector capability of modern high-performance computers and workstations.

For a simple 3x3 symmetrical and positive definite stiffness matrix [K], Eq. (10.1) can be represented as

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} = \begin{bmatrix} u_{11} & 0 & 0 \\ u_{12} & u_{22} & 0 \\ u_{13} & u_{23} & u_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \qquad \textbf{(10.5)}$$

The unknown, factorized matrix [U] in Eq. (10.5) can be easily obtained by expressing the equalities between the upper triangular matrix (on the left-hand-side) and its corresponding terms on the right-hand-side of Eq. (10.5). For a general stiffness matrix with dimensions nxn, the factorized matrix [U] can be obtained from [10.15, 10.26-10.27]

$$U_{ii} = \left(K_{ii} - \sum_{k=1}^{i-1} U_{ki}^2 \right)^{1/2} \qquad for \qquad i > 1 \qquad \textbf{(10.6)}$$

and

$$U_{ij} = \frac{1}{U_{ii}} * \left(K_{ij} - \sum_{k=1}^{i-1} U_{ki}\, U_{kj} \right) \qquad for \qquad i,j > 1 \qquad \textbf{(10.7)}$$

Using the above Eqs. (10.6-10.7), and assuming the coefficient stiffness matrix is full (to simplify the discussions), the information required to factorize a general i^{th} row can be readily identified in Figure 10.1 (see the rectangular, cross-region right above the i^{th} row of Figure 10.1).

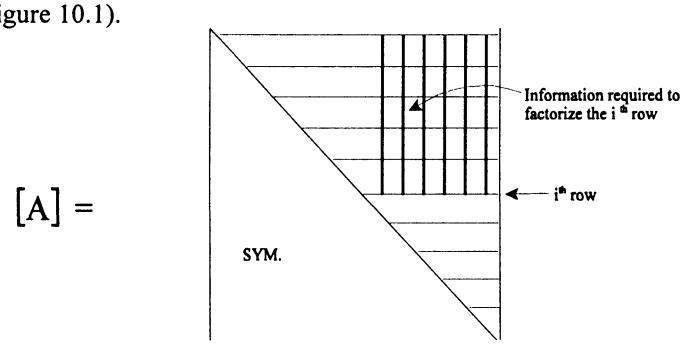


Figure 10.1 Information required to factorize the i^{it} row

10.2.2 LDL^T algorithm

The Choleski (or U^{T}U) factorization is efficient, however, its application is limited to the case where the coefficient stiffness matrix [K] is symmetrical and positive definite. With negligible additional computational efforts, the LDL^T algorithm can be used for broader applications (where the coefficient matrix can be either positive, or negative definite). In this algorithm, the given matrix [K] in Eq. (10.1) can be factorized as

$$[K] = [L][D][L]^T \qquad \textbf{(10.8)}$$

where [L] and [D] are lower triangular matrix (with unit values on the diagonal), and diagonal matrix, respectively. For a simple 3x3 symmetrical stiffness matrix, Eq. (10.8) can be explicitly expressed as

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} D_1 & 0 & 0 \\ 0 & D_2 & 0 \\ 0 & 0 & D_3 \end{bmatrix} \begin{bmatrix} 1 & L_{21} & L_{31} \\ 0 & 1 & L_{32} \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(10.9)}$$

The unknown L_{ij} and D_i can be easily obtained by expressing the equalities between the upper triangular matrix (on the left-hand-side) and its corresponding terms on the right-hand-side of Eq. (10.9). Since the LDL^T algorithm will be used later on to develop efficient, vectorized sparse algorithm, a pseudo-FORTRAN skeleton code is given in Table 10.1 (assuming the original given matrix [K] is symmetrical and full).

Table 10.1 Skeleton FORTRAN code for LDL^T
(assuming the matrix U is completely full)

1.	c	...	Assuming row 1 has been factorized earlier
2.			DO 11 I = 2, N
3.			DO 22 K = 1, I - 1
4.	c	...	Compute the multiplier (Note: U represents L^T)
5.			XMULT = $\dfrac{U(K,I)}{U(K,K)}$
6.			DO 33 J = I, N
7.			U (I, J) = U (I, J) - XMULT * U (K, J)
8.		33	CONTINUE
9.			U (K, I) = XMULT
10.		22	CONTINUE
11.		11	CONTINUE

As an example, implementation of the LDL^T algorithm, shown in Table 10.1, for a given, simple 3x3 stiffness matrix

$$[K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \quad \textbf{(10.10)}$$

will lead to the following factorized matrix

$$[U] = \begin{bmatrix} 2 & -\frac{1}{2} & 0 \\ & \frac{3}{2} & \frac{-2}{3} \\ & & \frac{1}{3} \end{bmatrix} \quad \textbf{(10.11)}$$

From Eq. (10.11), one can obtain

$$[D] \equiv [\textit{Diagonal of } U] = \begin{bmatrix} 2 & 0 & 0 \\ 0 & \frac{3}{2} & 0 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \qquad \textbf{(10.12)}$$

and

$$[L]^T \equiv \begin{bmatrix} \textit{Upper Triangular} \\ \textit{Portion of } U \end{bmatrix} = \begin{bmatrix} 1 & -\frac{1}{2} & 0 \\ & 1 & \frac{-2}{3} \\ & & 1 \end{bmatrix} \qquad \textbf{(10.13)}$$

For a simple 3x3 example shown in Eq. (10.10), the LDL^T algorithm presented in Table 10.1 can also be explained as

(a) Row #2_{new} = Row #$2_{original}$ - (xmult = $u_{1,2}/u_{1,1}$) * Row #1_{new}

(b) Row #3_{new} = Row #$3_{original}$ - (xmult1 = $u_{1,3}/u_{1,1}$) * Row #1_{new}
- (xmult2 = $u_{2,3}/u_{2,2}$) * Row #2_{new}

Using the data shown in Eq. (10.10), and following the LDL^T algorithm given in Table 10.1, one obtains:

2. For I = 2
3. K = 1
5. xmult = u_{12}/u_{11} = -1/2
6. J = 2
7. $$\begin{cases} u_{22} = u_{22} - (xmult)\left(u_{12}\right) = 2 - \left(\frac{-1}{2}\right)(-1) = \frac{3}{2} \\ J = 3 \\ u_{23} = u_{23} - (xmult)\left(u_{13}\right) = -1 - \left(-\frac{1}{2}\right)(0) = -1 \end{cases}$$
9. u_{12} = xmult =1/2
2. For I = 3
3. K = 1
5. $$xmult = \frac{u_{13}}{u_{11}} = \frac{0}{2} = 0$$
6. $$\begin{cases} J = 3 \\ u_{33} = u_{33} - (xmult = 0) * \left(u_{1,3} = 0\right) = 1 \end{cases}$$
9. u_{13} = 0
3. K = 2

5. $xmult = \dfrac{u_{23}}{u_{22}} = \dfrac{-1}{\left(\frac{3}{2}\right)} = \dfrac{-2}{3}$

6. J = 3

7. $u_{33} = u_{33} - \left(xmult = -\dfrac{2}{3} \right) \left(u_{23} = -1 \right) = \dfrac{1}{3}$

9. $u_{23} = xmult = \dfrac{-2}{3}$

Hence, upon exiting from Table 10.1, Eq. (10.11) will be obtained, and Eqs. (10.12-10.13) can be readily identified, accordingly.

10.3 Storage Schemes for the Coefficient Stiffness Matrix

Successful implementation of a sparse equation solution algorithm depends rather heavily on the reordering method used. While the Reversed Cuthill-Mckee (RCM), or Gipspoole-Stockmyer (GS)... reordering algorithms can be used efficiently in conjunction with skyline or variable bandwidth equation solution algorithms [10.22, 10.28-10.29], these reordering algorithms are not suitable for sparse equation solution algorithms. Designing efficient sparse-reordering algorithms is a big task itself, and is outside the scope of this chapter. For complete treatments on this subject, the readers are strongly recommended to popular textbooks and articles in the literatures [10.22, 10.30-10.36]. In this section, it is assumed that the best available sparse-reordering algorithm, such as Modified Minimum Degree (MMD), or Nested Dissection (ND) [10.22], has already been applied to the original coefficient matrix [K]. To facilitate the discussions in this section, assuming the resulted matrix [K] (after using MMD, or ND algorithm) takes the following form

$$[K] = \begin{matrix} 11. & 0 & 0 & 1. & 0 & 2. \\ & 44. & 0 & 0 & 3. & 0 \\ & & 66. & 0 & 4. & 0 \\ & & & 88. & 5. & 0 \\ SYM & & & & 110. & 7. \\ & & & & & 112. \end{matrix} \qquad \textbf{(10.14)}$$

For the data shown in Eq. (10.14), it can be easily shown (by referring to Eqs. 10.6-10.7, for example) that the factorized matrix [U] will have the following form:

$$[U] = \begin{matrix} x & 0 & 0 & x & 0 & x \\ & x & 0 & 0 & x & 0 \\ & & x & 0 & x & 0 \\ & & & x & x & F \\ & & & & x & x \\ & & & & & x \end{matrix} \qquad \textbf{(10.15)}$$

In Eq. (10.15), the symbols "x" and "F" represent the nonzero values after factorization. However, the symbol "F" also refers to "Fills-in" effect, since the original value of [K] at location F has zero entry.

For the same data shown in Eq. (10.14), if "skyline" equation solution is adopted, then the "fills-in" effect will take the following form:

$$\left[\bar{K}_s\right] = \begin{bmatrix} x & 0 & 0 & x & 0 & x \\ & x & 0 & F & x & F \\ & & x & F & x & F \\ & & & x & x & F \\ & & & & x & x \\ & & & & & x \end{bmatrix}$$

On the other hand, if "variable-bandwidth" equation solution is adopted, then the "fills-in" effect (on the data shown in Eq. 10.14) will have the following form:

$$\left[\bar{K}_b\right] = \begin{bmatrix} x & F & F & x & F & x \\ & x & F & F & x & F \\ & & x & F & x & F \\ & & & x & x & F \\ & & & & x & x \\ & & & & & x \end{bmatrix}$$

Thus, for the data shown in Eq. (10.14), the "sparse" equation solution is the best (in the sense of minimizing the number of arithmetic operations, and the required storage spaces in a sequential computer environment) and the "variable-bandwidth" equation solution is the worst one!

For practical computer implementation, the original stiffness matrix data, such as the one shown in Eq. (10.14), can be represented by the "sparse formats" as follows:

$$ISTARTROW = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 = N+1 \end{pmatrix} = \begin{Bmatrix} 1 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 7 \end{Bmatrix} \qquad \textbf{(10.16)}$$

$$ICOLNUM = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 = NCOEF \end{pmatrix} = \begin{Bmatrix} 4 \\ 6 \\ 5 \\ 5 \\ 5 \\ 6 \end{Bmatrix} \qquad \textbf{(10.17)}$$

$$DIAG = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 = N \end{pmatrix} = \begin{Bmatrix} 11. \\ 44. \\ 66. \\ 88. \\ 110. \\ 112. \end{Bmatrix} \qquad \textbf{(10.18)}$$

$$AK = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 = NCOEF \end{pmatrix} = \begin{Bmatrix} 1. \\ 2. \\ 3. \\ 4. \\ 5. \\ 7. \end{Bmatrix} \qquad \textbf{(10.19)}$$

The following definitions are used in Eqs. (10.16-10.19):

N	≡	Size of the original stiffness matrix [K]
NCOEF	≡	The number of non-zero, off-diagonal terms of the original stiffness matrix [K]. Only the upper triangular portion of [K] needs be considered.
ISTARTROW(i)	≡	Starting location of the first nonzero, off-diagonal term for the i^{th} row of [K]. The dimension for this integer array is N+1
ICOLNUM(j)	≡	Column numbers associated with each nonzero, off-diagonal terms of [K] (in a row-by-row fashion). The dimension for this integer array is NCOEF
DIAG(i)	≡	Numerical values of the diagonal term of [K]. The dimension for this real array is N
AK(j)	≡	Numerical values of the nonzero, off-diagonal terms of [K] (in a row-by-row fashion). The dimension for this real array is NCOEF

10.4 Reordering Algorithms

The reordering algorithm(s) used in any equation solver should be compatible to (or consistent with) the storage scheme and solution strategies used in the factorized, forward and backward solution phases. For example, if skyline, or variable bandwidth strategies are used, then either RCM, or GS reordering algorithms should be employed, since these reordering algorithms will try to minimize the bandwidths and/or the column heights of the factorized matrix [U]. These bandwidths and/or column heights minimization will help to reduce both memory requirement and also the number of arithmetic operations during the factorization phase. However, in sparse algorithms (for factorization), the concerned issues are not in the column heights (or bandwidths). Instead, efficient sparse factorization algorithms will require the number of "fills-in" to be minimized, in order to reduce both memory requirements and the number of arithmetic operations. Thus, it is quite likely to see that the "best" sparse reordering algorithm will lead to the "worse" performance, if it is used in conjunction with either skyline, or variable bandwidth strategies!

In this work, since fully vectorized sparse algorithms will be developed, either MMD or ND reordering algorithms [10.22] can be appropriately used.

10.5 Sparse Symbolic Factorization

The purpose of symbolic factorization is to find the locations of all nonzero (including "fills-in" terms), off-diagonal terms of the factorized matrix [U] (which has NOT been done yet!). Thus, one of the major goals in this phase is to predict the required computer memory for subsequent numerical factorization. The outputs from this symbolic factorization phase will be stored in the following 2 integer arrays (assuming the stiffness matrix data shown in Eq. 10.14 is used):

$$JSTARTROW \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 = N+1 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 3 \\ 4 \\ 5 \\ 7 \\ 8 \\ 8 \end{Bmatrix} \qquad \textbf{(10.20)}$$

$$JCOLNUM \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 = NCOEF2 \end{Bmatrix} = \begin{Bmatrix} 4 \\ 6 \\ 5 \\ 5 \\ 5 \\ 6 \\ 6 \end{Bmatrix} \qquad \textbf{(10.21)}$$

The following "new" definitions are used in Eqs. (10.20-10.21):

NCOEF2 ≡ The number of nonzero, off-diagonal terms of the factorized matrix [U]

JSTARTROW(i) ≡ Starting location of the first nonzero, off-diagonal term for the i^{th} row of the factorized matrix [U]. The dimension for this integer array is N+1

JCOLNUM(j) ≡ Column numbers associated with each nonzero, off-diagonal terms of [U] (in a row-by-row fashion). The dimension for this integer array is NCOEF2. Due to "fills-in" effects, NCOEF2 > > NCOEF.

As a rule of thumb for most engineering problems, the ratio of NCOEF2/NCOEF will be likely in the range between 7 and 20.

The key steps involved during the symbolic phase will be described in the following paragraphs:

Step 1: Consider each i^{th} row (of the original stiffness matrix [K])

Step 2: Record the locations (such as column numbers) of the original non-zero, off-diagonal terms

Step 3: Record the locations of the "fills-in" terms due to the contributions of some (not all) appropriated, previous rows j (where $1 \le j \le i-1$). Also, consider if current i^{th} row will have any immediate contribution to a "future" row

Step 4: Return to Step 1 for next row

A simple, but highly inefficient way to accomplish Step 3 (of the symbolic phase) will be identifying the nonzero terms associated with the i^{th} column. For

example, there will be no "fills-in" terms on row 3 (using the data shown in Eq. 10.14) due to "no contributions" of the previous rows 1 and 2. This fact can be easily realized by observing that the associated 3rd column of [K] (shown in Eq. 10.14) has no nonzero terms (and also by symbolically referring to Eqs. 10.6-10.7!)

On the other hand, if one considers row 4 in the symbolic phase, then the associated 4th column will have 1 nonzero term (on row 1). Thus, only row 1 (but not rows 2 and 3) may have "fills-in" contribution to row 4. Furthermore, since $K_{1,6}$ is nonzero (=2.), it immediately implies that there will be a "fills-in" terms at location $U_{4,6}$ of row 4.

A much more efficient way to accomplish step 3 of the symbolic phase is by creating 2 additional integer arrays, as defined in the following paragraphs

ICHAINL(i) ≡ Chained list for the i^{th} row. This array will be efficiently created to identify which previous rows will have contributions to current i^{th} row. The dimension for this integer, temporary array is N

LOCUPDATE(i) ≡ Updated starting location of the i^{th} row.

Using the data shown in Eq. (10.14), uses of the above 2 arrays in the symbolic phase can be described by the following step-by-step procedure:

Step 0: Initialize arrays ICHAINL (1, 2, ...N) = {0} and LOCUPDATE (1, 2, ..., N) = {0}

Step 1: Consider row i = 1

Step 2: Realize that the original nonzero terms occur in columns 4 and 6

Step 3: Since the chained list ICHAINL(i=1) = 0, no other previous rows will have any contributions to row 1

$$ICHAINL(4) = 1 \quad \textbf{(10.22)}$$

$$ICHAINL(1) = 1 \quad \textbf{(10.23)}$$

$$LOCUPDATE(i = 1) = 1 \quad \textbf{(10.24)}$$

Eqs. (10.22-10.23) indicate that "future" row i=4 will have to refer to row 1, and row 1 will refer to itself. Eq. (10.24) states that the updated starting location for row 1 is 1.

Step 1: Consider row i = 2

Step 2: Realizing the original nonzero term(s) only occurs in column 5

Step 3: Since ICHAINL(i=2) = 0, no other rows will have any contributions to row 2

$$ICHAINL(5) = 2 \quad \textbf{(10.25)}$$

$$ICHAINL(2) = 2 \quad \textbf{(10.26)}$$

$$LOCUPDATE(i = 2) = 3 \quad \textbf{(10.27)}$$

Eqs. (10.25-10.26) indicate that "future" row i = 5 will have to refer to row 2, and row 2 will refer to itself. Eq. (10.27) states that the updated starting location for row 2 is 3.

Step 1: Consider row i = 3

Step 2: The original nonzero term(s) occurs in column 5

Step 3: Since ICHAINL (i=3) = 0, no previous rows will have any contributions to row 3.

The chained list for "future" row i = 5 will have to be updated in order to include row 3 into its list.

$$ICHAINL(3) = 2 \quad \textbf{(10.28)}$$

$$ICHAINL(2) = 3 \quad \textbf{(10.29)}$$

$$LOCUPDATE(i = 3) = 4 \quad \textbf{(10.30)}$$

Thus, Eqs. (10.25, 10.29, 10.28) state that "future" row i = 5 will have to refer to rows 2, row 2 will refer to row 3, and row 3 will refer to row 2. Eq. (10.30) indicates that the updated starting location for row 3 is 4.

Step 1: Consider row i = 4

Step 2: The original nonzero term(s) occurs in column 5

Step 3: Since ICHAINL (i = 4) = 1, and ICHAINL (1) = 1 (please refer to Eq. 10.22), it implies row 4 will have contributions from row 1 only. The updated starting location of row 1 now will be increased by one, thus

$$LOCUPDATE(1) = LOCUPDATE(1) + 1 \quad \textbf{(10.31)}$$

Hence,

$$LOCUPDATE(1) = 1 + 1 = 2 \ (\textit{please refer to Eq. } 10.24) \quad \textbf{(10.32)}$$

Since the updated location of nonzero term in row 1 is now at location 2 (see Eq. 10.32), the column number associated with this nonzero term is column #6 (please refer to Eq. 10.17). Thus, it is obvious to see that there must be a "fills-in" term in column #6 of (current) row #4. Also, since $K_{1,6} = 2$ (or nonzero), it implies "future" row i=6 will have to refer to row 1.

Furthermore, since the first nonzero term of row 4 occurs in column 5, it implies that "future" row 5 will also have to refer to row 4 (in addition to refer to rows 2 and 3). The chained list for "future" row 5, therefore, has to be slightly updated (so that row 4 will be included on the list) as follows:

$$ICHAINL(4) = 3 \quad \textbf{(10.33)}$$

$$ICHAINL(2) = 4 \quad \textbf{(10.34)}$$

$$LOCUPDATE(i = 4) = 5 \quad \textbf{(10.35)}$$

Notice that Eq. (10.34) will override Eq. (10.29). Thus, Eqs. (10.25, 10.34, 10.33) clearly show that symbolically factorizing "future" row i = 5 will have to refer to rows 2, then 4 and then 3, respectively.

Step 1: Consider row i = 5

Step 2: The original nonzero term(s) occurs in column 6

Step 3: Since

$$ICHAINL(i = 5) = 2 \quad \textbf{(10.25, repeated)}$$

ICHAINL (2) = 4 **(10.34, repeated)**

ICHAINL (4) = 3 **(10.33, repeated)**

It implies rows #2 , then 4 and then 3 "may" have contributions (or "fills-in" effects) on row 5. However, since $K_{5,6}$ is originally a nonzero term, therefore, rows 2, 4, and 3 will NOT have any "fills-in" effects on row 5

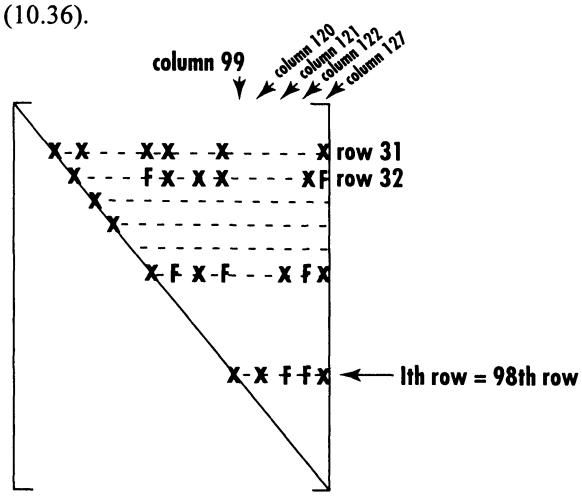
Step 1: There is no need to consider the last row i = N = 6, since there will be no "fills-in" effects on the last row! It is extremely important to emphasize that upon completion of the symbolic phase, the output array JCOLNUM(-) has to be re-arranged to make sure that the column numbers in each row should be in the increasing orders!

In this particular example (see data shown in Eq. 10.14), there is one fill-in taken place in row 4. However, there is no need to perform the "ordering" operations in this case, because the column numbers are already in the ascending order. This observation can be verified easily by referring to Eq. 10.15 . The original non-zero term of row 4 occurs in column 5 and the non-zero term (due to "fills-in") of row 4 occurs in column 6.

Thus, these column numbers are already in the ascending order!

In a general situation, however, "ordering" operations may be required as can be shown in Eq. (10.36).

[K] = **(10.36)**



During the sym4bolic factorization phase, assuming that only rows 31, 32 and 40 have their contributions to the current 98th row (as shown in Eq. 10.36). In this case, the original non-zero terms of row #98 occurs in columns 99, 120 and 127. However, the 2 "fills-in" terms occur in columns 122 and then 121 (assuming current row #98 will get the contributions from rows #32, 40, 31, respectively). Thus, the "ordering" operations are required in this case as shown in Eq., 10.36 (since column numbers 99, 120, 127, 122 and 121 are NOT in the ascending order yet!)

In subsequent paragraphs, more detailed discussions (including computer codings) about symbolic factorization will be presented. To facilitate the discussions, a specific stiffness matrix data is shown in Eq. (10.37)

$$K = \begin{bmatrix} x & & & x & & x & x \\ & x & & & x & & \\ & & x & & x & & \\ & & & x & x & F & F \\ & & & & x & x & F \\ & & & & & x & F \\ & & & & & & x \end{bmatrix} \qquad \textbf{(10.37)}$$

For this simple example, the "fills-in" effects (refer to symbols F) can be easily identified as shown in Eq. (10.37).

The symbolic codes (refer to Table 10.2) together with its explanations are described in the following paragraphs.

Table 10.2 FORTRAN listing of symbolic codes

```
      integer   isr(9),icn(9),jsr(9),jcn(66),ichain(9)                    001
      read(5,*) n,ncoef  !                                                002
      ncoef2=10*ncoef  !                                                  003
      read(5,*) (isr(i),i=1,n+1)  !                                       004
      read(5,*) (icn(i),i=1,ncoef)  !                                     005
      write(6,*) 'n,ncoef= ',n,ncoef  !                                   006
      write(6,*) 'isr(-)= ',(isr(i),i=1,n+1)                              007
      write(6,*) 'icn(-)= ',(icn(i),i=1,ncoef)  !                         008
      call symfact(n,isr,icn,jsr,jcn,ichain,ncoef,ncoef2) !               009
      stop  !                                                             010
      end  !                                                              011
C*********************************************************** !           012
      subroutine  symfact(n,isr,icn,jsr,jcn,ichain,ncoef,ncoef2)  !       013
       integer isr(1),icn(1),jsr(1),jcn(1),ichain(1) !                    014
C.....Purposes: Symbolic factorization  !                                 015
C       input: isr,   icn   structure of given matrix A in RR(U)U .!      016
```

```
C      n order of matrix A and of matrix U.  !                                  017
C       output:     jsr,jcn  structure of resulting matrix U in RR(U)U.  !      018
C     working space: ichain of dimension N.  Chained lists of rows  !           019
C                    associated with each column.  !                           020
C                    The array jsr is also used as the multiple !              021
C                    switch array.  !                                          022
C.....  !                                                                      023
C...This subroutine & all other subroutines in this file,EXCEPT numfa8,  !     024
C....has been verified by Duc T. Nguyen on March 9'96. Using !                 025
C..... the following .3 simultaneous equations:  !                             026
C.....  2  -1  0    x1      1  !                                               027
C..... -1   2 -1    x2      0  !                                               028
C.....  0  -1  1    x3      0  !                                               029
C------  !                                                                     030
      nm1=N-1  !                                                               031
      np1=N+1  !                                                               032
      DO 18 I=1,N  !                                                           033
      jsr(I)=0  !                                                              034
   18 ichain(I)=0  !                                                           035
      iount=1  !                                                               036
      DO 21 I=1,nm1  !                                                         037
      write(6,*) 'I= ',i  !                                                    038
      isrtem=icount  !                                                         039
      write(6,*) 'isrtem= ',isrtem  !                                          040
      jexceed=N+icount-I  !                                                    041
      write(6,*) 'jexceed= ',jexceed  !                                        042
      MIN=np1  !                                                               043
      write(6,*) 'MIN= ',min  !                                                044
      isra=isr(I)  !                                                           045
      write(6,*) 'isra= ',isra  !                                              046
      isrb=isr(I+1)-1  !                                                       047
      write(6,*) 'isrb= ',isrb  !                                              048
      IF(isrb.LT.isra)GO TO 30  !                                              049
      DO 25 J=isra,isrb  !                                                     050
      write(6,*) 'J= ',j  !                                                    051
      jcoln=icn(J)  !                                                          052
      write(6,*) 'jcoln= ',jcoln  !                                            053
      jcn(icount)=jcoln  !                                                     054
      write(6,*) 'jcn(',icount,')= ',jcn(icount)  !                            055
       icount=icount+1  !                                                      056
       write(6,*) 'icount= ',icount  !                                         057
       IF(jcoln.LT.MIN) then  !                                                058
       min=jcoln  !                                                            059
       write(6,*) 'MIN= ',min  !                                               060
       endif  !                                                                061
```

```
     Jsr(jcoln)=1 !                                              062
      write(6,*) 'jsr(',jcoln,')= ',jsr(jcoln) !                 063
  25  continue !                                                  064
  30  LAST=ichain(I) !                                            065
      write(6,*) 'LAST= ',last !                                  066
      IF(LAST.EQ.0)GO TO 66 !                                     067
      L=LAST !                                                    068
      write(6,*) 'L= ',l !                                        069
  79  L=ichain(L) !                                               070
      write(6,*) 'L= ',l !                                        071
      LH=L+1 !                                                    072
      write(6,*) 'LH= ',lh !                                      073
      iau=jsr(L) !                                                074
      write(6,*) 'iau= ',iau !                                    075
      ibu=jsr(LH)-1 !                                             076
      write(6,*) 'ibu= ',ibu !                                    077
      IF(LH.EQ.I) then !                                          078
      ibu=isrtem-1 !                                              079
      write(6,*) 'ibu= ',ibu !                                    080
      endif !                                                     081
      jsr(I)=I !                                                  082
      write(6,*) 'jsr(', i , ')= ', jsr(i) !                      083
      DO 84 J=iau,ibu !                                           084
      write(6,*) 'J= ',j !                                        085
      jcoln=jcn(J) !                                              086
      write(6,*) 'jcoln= ',jcoln !                                087
      IF(jsr(jcoln).EQ.I)GO TO 84 !                               088
      jcn(icount)=jcoln !                                         089
      write(6,*) 'jcn(', icount ,')= ',jcn(icount) !              090
      icount=icount+1 !                                           091
      write(6,*) 'icount= ',icount !                              092
      jsr(jcoln)=I !                                              093
      write(6,*) 'jsr(', jcoln ,')= ',jsr(jcoln) !                094
      IF(jcoln.LT.MIN) then !                                     095
      min=jcoln !                                                 096
      write(6,*) 'MIN= ',min !                                    097
      endif !                                                     098
   84 continue !                                                  099
      IF(icount.EQ.jexceed) GO TO 723 !                           100
      IF(L.NE.LAST)GO TO 79 !                                     101
   66 IF(MIN.EQ.np1)GO TO 322 !                                   102
  723 L=ichain(MIN) !                                             103
      write(6,*) 'L= ',l !                                        104
      IF(L.EQ.0) GO TO 875 !                                      105
      ichain(I)=ichain(L) !                                       106
```

```
      Write(6*) 'ichain('ichain(', i,')=',ichain(i)!                    107
       ichain(L)=I  !                                                   108
       write(6,*) 'ichain(', l ,')= ',ichain(l)  !                      109
       GO TO 322  !                                                     110
  875  ichain(MIN)=I  !                                                 111
       write(6,*) 'ichain(', min ,')= ',ichain(min)  !                  112
       ichain(I)=I  !                                                   113
       write(6,*) 'ichain(', i ,')= ',ichain(i)  !                      114
  322  jsr(I)=isrtem  !                                                 115
       write(6,*) 'jsr(', i ,')= ',jsr(i)  !                            116
   21  continue  !                                                      117
       jsr(N)=icount  !                                                 118
       write(6,*) 'jsr(', n ,')= ',jsr(n)  !                            119
       jsr(np1)=icount  !                                               120
       write(6,*) 'jsr(', np1 ,')= ',jsr(np1)  !                        121
       return  !                                                        122
      end  !                                                            123
```

Lines 1-12: Main program: input data are read, such as the size (N) of the coefficient matrix, the number of nonzeros (NCOEF) of the original coefficient matrix (see line 2), starting nonzero locations of each row (see line 4, array isr), column numbers associated with nonzero terms in each row (see line 5, array icn). The entire input data file for the stiffness matrix data (shown in Eq. 10.37) is given in Table 10.3

Lines 12-30: These lines are essentially self-explained, since most of them are comment statements. The following arrays need to be defined:
jsr(-), and jcn(-): Same definitions as arrays isr(-), and icn(-), respectively. However, arrays jsr(-) and icn(-) are associated with the factorized stiffness matrix, whereas arrays isr(-) and icn(-) are associated with the original stiffness matrix.
ichain(-): Chained lists associated with each row. This array has been explained as shown in Eqs. (10.22-10.23, 10.25-10.26).
NCOEF2: Total number of nonzeros of the factorized stiffness matrix (including "fills-in" terms)

Lines 31-36: Arrays jsr(-) and ichain(-) are initialized to zero. The counter "icount" is initialized to one. This integer variable will be increased by 1 whenever a nonzero term (either from the original stiffness matrix, or from the "fills-in") is detected. Thus, upon exiting from the symbolic factorization (or subroutine symfact, on line 13), the value of "icount" will be "NCOEF2".

Line 37: First major do-loop for symbolic factorization. The index I represents the current I^{th} row

Line 39: The temporary starting location for current I^{th} row is stored in variable isrtem

Line 41: Since "icount" represents the cumulative number of nonzero terms (up to and including the current I^{th} row), hence (N-I) represents the maximum possible number of columns in row I (see Figure 10.2)

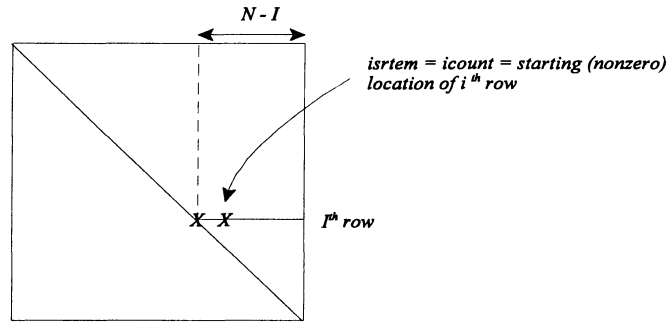


Figure 10.2 Interpretation of the variable "jexceed"

Line 43: The MINimum column number (=MIN) in all previous rows (and including the original nonzero terms of current row i) which have contributions to the current i^{th} row. The value of MIN is initialized to a large column number (such as NP1, or N + 1). For example, assuming I = 20, and MIN = 43, then the algorithm will prepare the chained list ichain(-) for future row #MIN.

Lines 45-47: Starting and ending locations of nonzero terms in the current I^{th} row

Line 50: In this do-loop, all nonzero terms of the I^{th} row will be considered

Line 52: Column number (of I^{th} row) of the original (before factorization) stiffness matrix is recalled.

Line 54: Column number of the (soon will be) factorized matrix is recorded.

Line 56: The counter "icount" is updated whenever a nonzero term and/or fills in term has been found

Lines 58-61: For structural engineering applications, the original stiffness matrix is always in order (for example, the column numbers, corresponding to nonzero terms in each row, is already in the increasing order), thus this if statement is not really required. However, for general applications, the original, coefficient (stiffness) matrix may not be in order yet, and therefore, the value of MIN may have to be updated (as shown in line 59)

Lines 62-64: Column # jcoln has already contributed to row #I. Later on, we do NOT want to include column # jcoln again for row I!

Lines 43-64: In summary, the main purpose for this segment of the code is to load the I^{th} row of the original coefficient (stiffness) matrix into the I^{th} row of the (soon to be) factorized matrix [U]

Line 65: The chained list array ichain(-) is used to find "which" previous rows will have contributions to the current I^{th} row. The row # of the last

previous row (which has contributions to current I^{th} row) is stored in the variable "LAST"

Line 67: If LAST = 0, it implies that the current I^{th} row will NOT receive any contributions from its previous rows (However, the "current" I^{th} row may have contributions to "future" J^{th} row, where J > I)

Line 68: The value of LAST is copied into a variable L

Line 70: Among all previous rows which have contributions to the current I^{th} row, L is the "current" previous row number to be considered

Line 74: Starting location for the L^{th} row (of the soon to be factorized matrix [U]) is stored in the variable "IAU"

Lines 72, 76: Ending location for the L^{th} row is stored in the variable "IBU". Since LH = L + 1, therefore, JSR(LH) will give the starting location of the $(L+1)^{th}$ row. Hence, JSR(LH) - 1 (≡ IBU) will give the ending location of the L^{th} row!

Lines 78-81: These statements can be better explained by referring to Figure 10.3

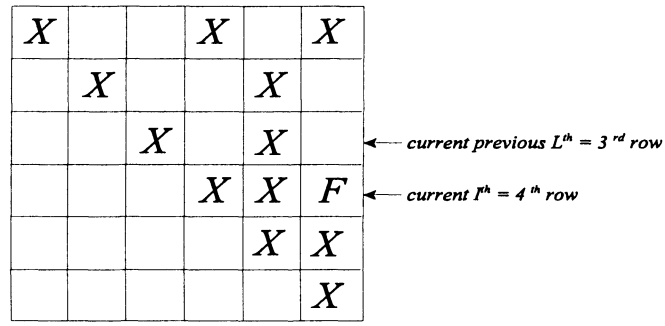


Figure 10.3 Special case for obtaining the ending location of the L^{th} row

Assuming the current I^{th} row to be the 4^{th} row, and assuming the "previous current" L^{th} row to be the 3^{rd} row, thus the starting location of the L^{th} row can be obtained from line 74 as:

$$IAU = JSR\left(3^{rd}\ row\right) \tag{10.38}$$

However, if we try to obtain the ending location of the L^{th} row from line 76, then:

$$IBU = JSR\left(4^{th}\ row\right) - 1 \tag{10.39}$$

Since we are currently considering row I = 4, the variable JSR (3^{rd} row) has already been properly defined. However, the variable JSR (4^{th} row) has not been properly defined yet. This variable JSR (4^{th} row) will have its properly defined value only when the current row I (=4) has been completely processed! Thus, special formula (shown on line 79) need be used to properly define the value of IBU whenever row #L is right above current row #I. Since ISRTEM represents the "proper" starting location of row #I = 4, hence ISRTEM-1 will give a proper ending location of row #L = 3.

Line 84: All nonzero terms of the "current previous" row #L are considered in this do-loop (with index J). Here, the index J represents the locations associated with nonzero terms (of row #L).

Line 86: Column number, associated with a nonzero location, is defined in variable JCOLN

Lines 82, 88: Referring to Figure 10.3, these two statements (together with the statement shown on line 62) will guarantee that such a nonzero term of previous row #L, which has contributions to the diagonal term of current row #I, will NOT be included in the JSR(-) and JCN(-) arrays

Furthermore, these two statements will also guarantee that the contributions of the factorized, fills-in term U_{46} on the term U_{56} (assuming current row #I = 5) will also NOT be included in arrays JSR(-) and JCN(-), since the "original" (before factorization) nonzero term K_{56} has already been existed!

Lines 89, 91, 93: These three statements play similar roles as earlier statements, which have already been explained on lines 54, 56 and 62, respectively.

Lines 95-98: These statements play similar roles as earlier statements, which have already been explained on lines 58-61. In here, however, ordering the column numbers (to make sure they are in increasing order) is usually required, regardless structural, or general applications (recalled the earlier descriptions related to Eq. 10.36)

Line 100: This statement, if satisfied, implies that current row #I is already full (please refer to the variable JEXCEED, defined earlier on line 41)

Line 101: This statement, if satisfied, implies that the next previous row (which has contributions to current I^{th} row) has to be considered

Lines 65-101: In summary, the main purpose of this segment of the code is to consider the possibilities for fills-in effects of previous rows on the current I^{th} row

Line 102: Only two places where the value for variable MIN is redefined: lines #59 and #96. This statement, is satisfied, will imply:

(a) Current row #I does NOT have any off-diagonal terms, and/or

(b) There are no previous rows which have contributions to row #I

Lines 103-123: In this segment of the code, the contribution of current row #I on "future" row #MIN is considered. Figure 10.4 (a, b, c, d) needs to be referred to for better understanding the logic behind this code segment. The original stiffness matrix (with fills-in terms denoted by the symbol "F") is shown in Figure 10.4 (a). Assuming the current row is row #I=5, and MIN = 7.

Line 103: Assuming L = ICHAIN(7) = 0, then the "if statement" (on line 105) will direct the code to line #111

Line 111: ICHAIN(7) = 5

Line 113: ICHAIN(5) = 5

Line 115: Starting location for the I^{th} row of the (soon to be) factorized matrix is recorded in array (JSR(-), before considering the next I^{th} row (see lines 117 & 37)

Lines 103, 111 & 113:

In summary, if L = 0, it implies that up until now, "current" row #I = 5 is the ONLY row which will have contributions to "future" row #MIN. Hence, the chained list for "future" row #MIN can now be established, accordingly (see Figure 10.4b).

Line 103: On the other hand, if one assumes L = ICHAIN(7) ≠ 0, say L = 3, then the "if statement" (on line 105) will direct the code to line 106

Line 106: ICHAIN(5) = ICHAIN(3)
Assuming ICHAIN (3) = 2, then ICHAIN(5) = 2

Line 108: ICHAIN(3) = 5, and then

Lines 110, 115: Starting location for the I^{th} row of the (soon to be) factorized matrix is recorded in array JSR(-), before considering the next I^{th} row (see lines 117 & 37)

Lines 103, 106, 108 & 110:

In summary, if L ≠ 0, it implies that the "current" row #I = 5 will be added to the chained lists for "future" row #MIN. The data shown in Figure 10.4a has clearly indicated that "previous" rows #2 & #3 had already been included on the chained lists for "future" row #7 (see Figure 10.4c). After including the "current" row #MIN, the chained list for "future" row #MIN is updated as shown in Figure 10.4d

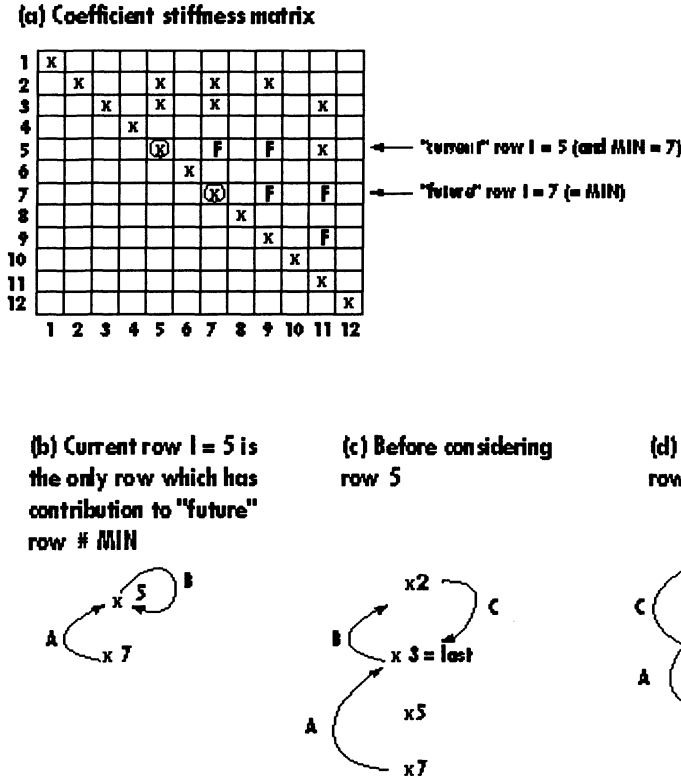


Figure 10.4 Contributions of "current" row # I on "future" row # MIN

The data file associated with Eq. 10.37 (also refer to lines #2,4, and 5 of Table 10.2) is given in Table 10.3

Table 10.3 Example data for symbolic factorization

7	7							
1	4	5	6	7	8	8	8	
4	6	7	5	5	5	6		

How to Obtain the Symbolic Factorized Matrix with Proper Orderings?? It has been explained in Eq. (10.36) that upon exiting from the symbolic factorization phase, the coefficient (stiffness) matrix (including new nonzero terms, due to fills-in effects) need to be ordered to make sure the column numbers (associated with nonzero terms)

in each row are in ascending order. This type of ordering operation is necessary before performing the numerical factorization.

To facilitate the discussions, let's consider an unordered, rectangular matrix (shown in Figure 10.5) which can be described by Eqs. (10.40-10.42)

1	2	3	4	5	6	= No. Columns (= NC)
		$A^{(3rd)}$		$B^{(1st)}$	$C^{(2nd)}$	1
$D^{(5th)}$			$E^{(4th)}$			2
		$F^{(6th)}$	$G^{(7th)}$			3
$H^{(10th)}$		$I^{(9th)}$	$J^{(8th)}$			4
	$K^{(11th)}$			$L^{(13th)}$	$M^{(12th)}$	5 = No. Rows (= NR)

Figure 10.5 An unordered, rectangular matrix
(numbers in parenthesis represent location numbers)

The starting location for each row in Figure 10.5 is given as

$$ISTARTROW \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 = NR + 1 \end{pmatrix} = \begin{Bmatrix} 1 \\ 4 \\ 6 \\ 8 \\ 11 \\ 14 \end{Bmatrix} \qquad \textbf{(10.40)}$$

The (unordered) column numbers associated with each row is given as

$$ICOLNUM \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 = ISTARTROW(6) - 1 \end{pmatrix} = \begin{Bmatrix} 5 \\ 6 \\ 3 \\ 4 \\ 1 \\ 3 \\ 4 \\ 4 \\ 3 \\ 1 \\ 2 \\ 6 \\ 5 \end{Bmatrix} \qquad \textbf{(10.41)}$$

The "numerical" values (for the matrix shown in Figure 10.5) can be given as

$$AK \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \end{pmatrix} = \begin{Bmatrix} B \\ C \\ A \\ E \\ D \\ F \\ G \\ J \\ I \\ H \\ K \\ M \\ L \end{Bmatrix} \qquad \textbf{(10.42)}$$

The "unordered" matrix, shown in Figure 10.5, can be made into an "ordered" matrix by the following 2-step procedure:

Step 1: Transposing The Given Unordered Matrix Once: If the matrix shown in Figure 10.5 is transposed, then one will obtain the following matrix (shown in Figure 10.6)

	Col. No.1	Col. No. 2	Col. No. 3	Col. No. 4	Col. No. 5
Row #1		$D^{(1st)}$		$H^{(2nd)}$	
Row #2					$K^{(3rd)}$
Row #3	$A^{(4th)}$		$F^{(5th)}$	$I^{(6th)}$	
Row #4		$E^{(7th)}$	$G^{(8th)}$	$J^{(9th)}$	
Row #5	$B^{(10th)}$				$L^{(11th)}$
Row #6	$C^{(12th)}$				$M^{(13th)}$

Figure 10.6 Transposing a given unordered matrix once

The starting locations and the associated column numbers for the matrix shown in Figure 10.6 are given as:

$$ISTROW\ TRANSPOSE\,1 \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{pmatrix} = \begin{Bmatrix} 1 \\ 3 \\ 4 \\ 7 \\ 10 \\ 12 \\ 14 \end{Bmatrix} \qquad \textbf{(10.43)}$$

$$ICOLN\ TRANSPOSE1 \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \end{pmatrix} = \begin{Bmatrix} 2 \\ 4 \\ 5 \\ 1 \\ 3 \\ 4 \\ 2 \\ 3 \\ 4 \\ 1 \\ 5 \\ 1 \\ 5 \end{Bmatrix} \quad \textbf{(10.44)}$$

It is important to realize from Figure 10.6 (and also from Eq. 10.44) that the matrix shown in Figure 10.6 is already properly ordered. For each row of the matrix shown in Figure 10.6, increasing the location numbers also associates with increasing the column numbers.

Step 2: Transposing the Given Unordered Matrix Twice: If the matrix shown in Figure 10.6 is transposed again (or the original matrix shown in Figure 10.5 is transposed twice), then one will obtain the matrix as shown in Figure 10.7

1	2	3	4	5	6	
		$A^{(1st)}$		$B^{(2nd)}$	$C^{(3rd)}$	1
$D^{(4th)}$			$E^{(5th)}$			2
		$F^{(6th)}$	$G^{(7th)}$			3
$H^{(8th)}$		$I^{(9th)}$	$J^{(10th)}$			4
	$K^{(11th)}$			$L^{(12th)}$	$M^{(13th)}$	5

Figure 10.7 Transposing a given unordered matrix twice

The starting locations and the associated column numbers (for the matrix shown in Figure 10.7) are given as:

$$IST ROW TRANSPOSE2 \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 = NR + 1 \end{pmatrix} = \begin{Bmatrix} 1 \\ 4 \\ 6 \\ 8 \\ 11 \\ 14 \end{Bmatrix} \quad \textbf{(10.45)}$$

$$ICOLNTRANSPOSE2 \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \end{pmatrix} = \begin{Bmatrix} 3 \\ 5 \\ 6 \\ 1 \\ 4 \\ 3 \\ 4 \\ 1 \\ 3 \\ 4 \\ 2 \\ 5 \\ 6 \end{Bmatrix} \qquad \textbf{(10.46)}$$

Thus, one concludes that an "ordered" matrix can be obtained from the original, "unordered" matrix simply by transposing the "unordered" matrix twice!

10.6 Sparse Numerical Factorization

It is generally safe to say that sparse numerical factorization is more complicated for computer coding implementation than its skyline, or variable bandwidth cases. Main difficulties are due to complex "book-keeping" (or index referring) process. The "key" ideas in this numerical phase are still basically involved the creation and usage of the 2 integer arrays ICHAINL(-) and LOCUPDATE(-), which have been discussed with great details in Section 10.5. There are two (2) important modifications that need to be done on the symbolic factorization, in order to do the sparse numerical factorization (to facilitate the discussion, please refer to the data shown in Eq. 10.14):

(a) For symbolic factorization purpose, there is no need to have any floating arithmetic calculation. Thus, upon completing the symbolic process for row 4, there are practically no needs to consider row 2 and/or row 3 for possible contributions to row 5. Only row 4 needs to be considered for possible contributions (or "fills-in" effects) to row 5 (since row 4, with its "fills-in," is already full).

For numerical factorization purpose, however, all rows 2, then 4 and then 3 will have to be included in the numerical factorization of row 5.

(b) For sparse numerical factorization, the basic skeleton FORTRAN code for LDL^T, shown in Table 10.1 of Section 10.2.2, can be used in conjunction with the chained list strategies (using arrays ICHAINL and LOCUPDATE) which have been discussed earlier in Section 10.5.

The skeleton FORTRAN code for sparse LDL^T is shown in Table 10.4. Comparing Table 10.1 and Table 10.4, one immediately sees the "major differences" only occur in the 2 do-loop indexes K and J, on lines 3 and 6, respectively.

Table 10.4 Pseudo FORTRAN skeleton code for sparse LDL^T factorization

1.	c	...	Assuming row 1 has been factorized earlier
2.			DO 11 I = 2, N

3.			DO 22 K = Only those previous rows which have contributions to current row I
4.1	c	...	Compute the multiplier (Note: U represents L^T)
5.1			XMULT = U (K, I) / U (K, K)
6.			DO 33 J = appropriated column numbers of row #K
7.1			U(I, J) = U(I, J) - XMULT * U(K, J)
8.		33	CONTINUE
9.1			U (K, I) = XMULT
10.		22	CONTINUE
11.		11	CONTINUE

A more detailed FORTRAN code for numerical factorization is shown in Table 10.5. The following are definitions of various input, output and temporary arrays used in Table 10.5

Input Arrays:

ISR(N+1), IU(N+1) = Starting location number of the first nonzero term in each row of the "original" matrix, and the factorized matrix, respectively

ICN(Ncoff), JCN(Ncof2) = Column numbers (associated only with nonzero terms) of the original matrix, and the factorized matrix, respectively.

AN(Ncoff) = Off-diagonal terms of the original (stiffness) matrix

AD(N) = Diagonal terms of the original (stiffness) matrix

Output Arrays:

UN(Ncof2) = Factorized, off-diagonal terms of the (stiffness) matrix

DI(N) = Inverse of the factorized diagonal terms of the matrix (this array is also used as the expanded accumulator)

Temporary Arrays:

Ichain(N) = Chained list of rows associated with a column

KUPP (N) = "Starting" location of a row

Table 10.5 Detailed numerical sparse factorization

```
   DO 10 J=1,N  !                                                    001
  10 ichain(J)=0  !                                                  002
C.....Begin of of 1-st (nested) loop: outer-most loop, for each i-th row  !
   DO 130 I=1,N  !                                                   003
  IH=I+1  !                                                         004
```

```
  ICUU=IU(I) !                                                    005
  IBUU=IU(IH)-1 !                                                 006
    IF(IBUU.LT.ICUU)GO TO 40 !                                    007
    DO 20 J=ICUU,IBUU !                                           008
 20  DI(jcn(J))=0. !                                             009
    ica=isr(I) !                                                  010
    iba=isr(IH)-1 !                                               011
    IF(iba.LT.ica)GO TO 40 !                                      012
    DO 30 J=ica,iba !                                             013
  30 DI(icn(J))=AN(J) !                                           014
  40 DI(I)=AD(I) !                                                015
    LAST=ichain(I) !                                              016
    IF(LAST.EQ.0)GO TO 90 !                                       017
    LN=ichain(LAST) !                                             018
C.....begin of 2-nd (nested) loop: considering all APPROPRIATED previous
C.....                    rows (any appropriated rows 1--->i-1)
  50 L=LN !                                                       019
    LN=ichain(L) !                                                020
    iucl=kupp(L) !                                                021
    iudl=IU(L+1)-1 !                                              022
    UM=UN(iucl)*DI(L) !                                           023
C.....begin of 3-rd (nested) inner-most loop: considering all APPROPRIATED
C.....                          columns (any columns i--->n)
    DO 60 J=iucl,iudl !                                           024
    JJ=jcn(J) !                                                   025
  60 DI(JJ)=DI(JJ)-UN(J)*UM !                                     026
    UN(iucl)=UM !                                                 027
    kupp(L)=iucl+1 !                                              028
    IF(iucl.EQ.iudl)GO TO 80 !                                    029
    J=jcn(iucl+1) !                                               030
    JJ=ichain(J) !                                                031
    IF(JJ.EQ.0)GO TO 70 !                                         032
    ichain(L)=ichain(JJ) !                                        033
    ichain(JJ)=L !                                                034
    GO TO 80 !                                                    035
  70 ichain(J)=L !                                                036
    ichain(L)=L !                                                 037
C.....the following go to statement is equivalent to 2-nd nested loop
C.....for factorization
```

```
 80 IF(L.NE.LAST)GO TO 50 !                                    038
 90 DI(I)=1./DI(I) !                                           039
    IF(IBUU.LT.ICUU)GO TO 120 !                                040
    DO 100 J=ICUU,IBUU !                                       041
100 UN(J)=DI(jcn(J)) !                                         042
    J=jcn(ICUU) !                                              043
    JJ=ichain(J) !                                             044
    IF(JJ.EQ.0)GO TO 110 !                                     045
    ichain(I)=ichain(JJ) !                                     046
    ichain(JJ)=I !                                             047
    GO TO 120 !                                                048
110 ichain(J)=I !                                              049
    ichain(I)=I !                                              050
120 kupp(I)=ICUU !                                             051
130 continue !                                                 052
```

Explanations for various statements in Table 10.5 are given in the following paragraphs

Lines 1-2: Initialize the array Ichain(-)

Line 3: The first DO-LOOP of the numerical factorization. The index I represents the I^{th} row which is currently being factorized

Lines 4-6: Find the starting (=ICUU) and ending (=IBUU) locations of the I^{th} row of the factorized matrix

Line 7: Check and see if the I^{th} row (of the factorized matrix) has no off-diagonal terms

Lines 8-9: The array DI(-) is initialized. For better efficiency, however, only those nonzero locations of the I^{th} row are included here

Lines 10-11: Find the starting (=ICA) and ending (=IBA) locations of the I^{th} row of the "original" (or unfactorized) matrix

Line 12: Check and see if the I^{th} row (of the original matrix) has no off-diagonal terms

Lines 13-14: Copy Row #I (of the original matrix) into a temporary array DI(-)

Line 15: Copy the I^{th} diagonal term (of the original matrix) into a temporary array DI(-)

Line 16: The "last" previously factorized row which has contributions to the currently factorized I^{th} row. Strictly speaking, this should be considered as the "first" previously factorized row! As an example, suppose the currently factorized row #6 will require [according to array Ichain(-)] the information from the previously factorized rows #5, 4, 1, respectively, then LAST = 1. Based upon skyline and/or variable bandwidth factorization, the currently factorized row #6

should receive the contributions from the previously factorized row #1 first, then row #4 and row #5, respectively!

Line 17: If LAST = 0, then it implies that there are NO previously factorized rows which have contributions to the current I^{th} row. Referring to the data shown in Eq. (10.14), as an example, for I^{th} row = 3^{rd} row, there are NO previous rows (such as rows 1 & 2) which have contribution to row 3, hence LAST = 0. On the other hand, for the current I^{th} row = 4^{th} row (see Eq. 10.14), it will receive the contribution from the previously factorized row #1, hence LAST = 1

Lines 18-19: The "first" previously factorized row #L is defined as

L = LN, or L = Ichain (LAST)

Thus, L can be considered as the "current previous" row # which has contributions to the currently factorized row #I. Note that line #19 (L = LN) is the beginning of the 2^{nd} loop of the numerical factorization (which has contributions to the currently factorized row #I)

Line 20: The "next" previously factorized row #LN is defined

Lines 21-22: Starting and ending (nonzero) location of row #L, respectively

Line 23: Compute the multiplier factor (also see Table 10.1 in Section 10.2.2 for the expression xmult = $u_{K,I}/u_{K,K}$). Note here DI(-) contains the inverse (or reciprocal) value of diagonal term (see line 39)

Lines 24-26: The 3^{rd} do loop of numerical factorization is used to partially update the current row #I (due to the contributions from the previously factorized row #L). The partially updated current row #I is stored temporarily under the array DI(-)

Line 27: This statement plays the same role as the statement u(K, I) = xmult in Table 10.1

Line 28: The "next" (non-zero) starting location for row #L is computed (See Figure 10.8)

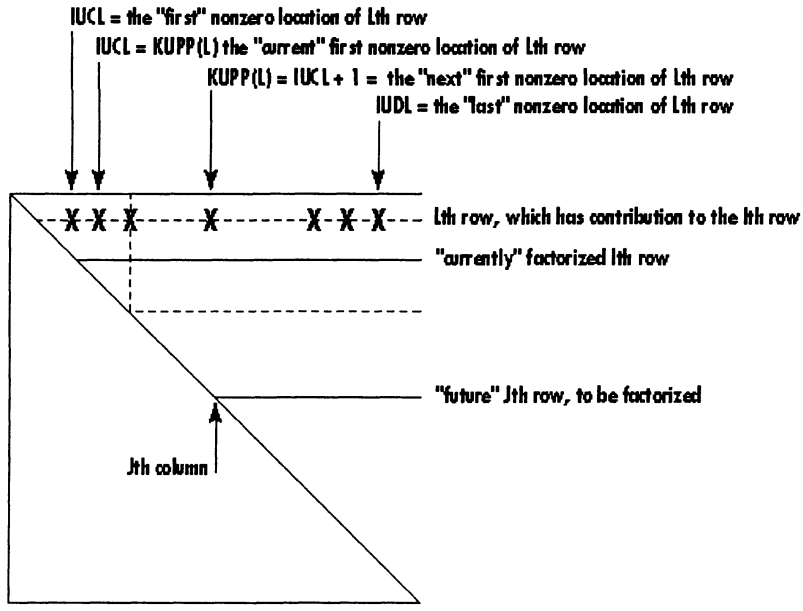


Figure 10.8 The "current" first nonzero location of the L^{th} row, which has contribution to the currently factorized I^{th} row

Line 29: If IUCL = IUDL, then (accordingly to Figure 10.8) it implies row #L has no more nonzero terms, and therefore row #L has absolutely no more use in the subsequent rows (such as rows #I+1, I+2, ... , N).

Line 30: Find the column number (=J) which corresponds to the "next" (or "new") first (nonzero) location of row #L

Line 31: Get the information (=JJ) about column J, see if any nonzero rows in this column

Lines 32, 36-37: If JJ = 0, it implies that row #L will be the only row (so far) which will have the contributions into the "future" (to be factorized) row #J. Thus, the "future" factorized row #J will have to refer to the "previously factorized" row L (see statement on line #36)

Lines 33, 34: If JJ ≠ 0, then it implies that row #L will be added to the "chained list" of those (previously factorized) rows which will have contributions to "future" row #I (see Figure 10.9)

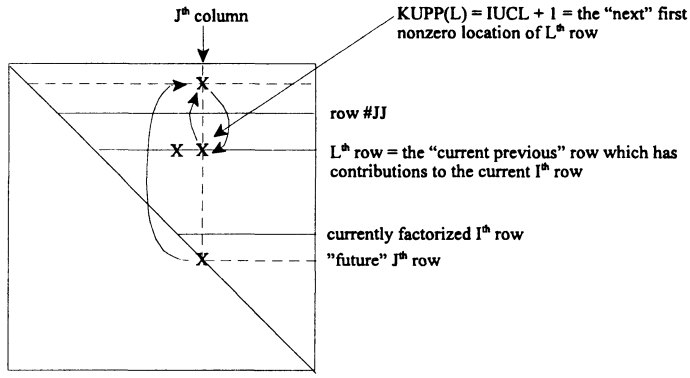


Figure 10.9 Previously factorized L^{th} row is being added to the list of rows which will have contributions to the future J^{th} row

Rows #JJ and #L will have the contributions to the "future" row #J, hence row #L (or, the "current previous" row which will have contributions to "future" row #J) will be added on the chained list array Ichain(-). In other words, before row #L entering into the picture, row #JJ is assumed to be the only row who will contribute to "future" row #J. Thus, the chained list array will be:

IP(J) = JJ
IP(JJ) = JJ

Now, since row #L also has contributions to future row #J, hence the chained list array will be updated as

IP(J) = JJ
IP(JJ) = L
IP(L) = JJ

The evolution of the chained list array IP(-) to include those (previously factorized) rows (such as row #JJ, and then subsequently row #L) which will have the contributions to "future" row #J is shown in Figure 10.10

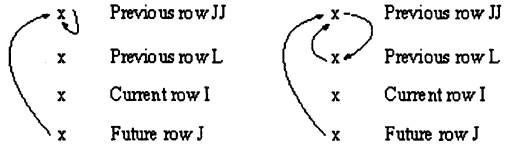


Figure 10.10 Evolution of previous rows #JJ and #L to future row #J

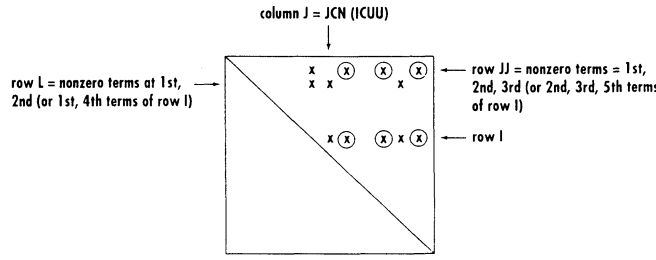


Figure 10.11 Contributions from the previously factorized rows #JJ and #L to current I^{th} row

Line 38: If L ≠ LAST, it implies the previously factorized row #L is not yet the last row which have contributions to the current I^{th} row (in other words, there are more previously factorized row(s) which have contributions to the current I^{th} row). In this case, the algorithm will go back to line #19 to consider the contributions of the "new" previously factorized row (="new" row L) on the current I^{th} row

Line 39: The inverse (or reciprocal) of diagonal term [= 1/DI(I) or 1/u(K,K) as shown in Table 10.1]. This information will be used in calculating the multiplier constant as shown on line 23

Line 40: If IBUU < ICUU, then it implies there are NO MORE nonzero terms in row #I of the factorized matrix

Lines 41-42: The "temporary" array DI(-), which has been used to store the updated (or factorized) I^{th} row (see line 26) is now copied into a "permanent" array UN(-).

This approach is more convenient than introducing the "permanent" array UN(-) directly on line 26, because row #JJ will have the contributions on the 2^{nd}, 3^{rd} and 5^{th} off-diagonal terms of the current I^{th} row, whereas row #L will have the contributions on the 1^{st} and 4^{th} off-diagonal terms of the current I^{th} row (see Figure 10.11)

Lines 43-50: Essentially followed the same logic as already explained in lines 30-37. In lines 30-37, the chained list Ichain(-) array was updated so that "earlier" rows which contribute to the "current" row I and "future" row(s) J were recorded. However, in lines 43-50, the Ichain(-) array is updated so that the "current" row I which has contributions to "future" row(s) J is recorded.

Line 51: The "current" first (nonzero) location of row #I is recorded in array KUPP(I)

Line 52: Go back to line #3 to consider the next row

10.7 Forward and Backward Solutions

For a single right-hand-side vector {f}, the combined forward and backward solution time is very small as compared to the numerical factorization time. However, for multiple right-hand-side vector {f}, or for cases where the vector {f} needs to be

modified repeatedly (such as in eigenvalue analysis, structural dynamics, nonlinear finite element analysis, electro-magnetic engineering applications, etc....), the forward and backward solution time has to be considered more seriously.

10.7.1 Forward substitution phase

In the forward substitution phase, the intermediate vector {y} can be solved from Eq. (10.3)

$$[U]^T \{y\} = \{f\} \qquad \textbf{(10.3, repeated)}$$

For the Choleski method, $[U]^T = [L]$ where [L] is a lower triangular matrix. Thus, Eq. (10.3) can be rewritten as

$$[L] \{y\} = \{f\} \qquad \textbf{(10.47)}$$

For the data shown in Eq. (10.15), Eq. (10.47) has the following form

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 & 0 & 0 \\ 0 & L_{22} & 0 & 0 & 0 & 0 \\ 0 & 0 & L_{33} & 0 & 0 & 0 \\ L_{41} & 0 & 0 & L_{44} & 0 & 0 \\ 0 & L_{52} & L_{53} & L_{54} & L_{55} & 0 \\ L_{61} & 0 & 0 & L_{64} & L_{65} & L_{66} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{Bmatrix} \qquad \textbf{(10.48)}$$

The first unknown y_1 can be found easily as

$$y_1 = \frac{f_1}{L_{11}} \qquad \textbf{(10.49)}$$

As soon as y_1 has been solved, the right-hand-side vector {f} can be updated by taking the first column of [L] (or for the actual implementation, the first row of [U]) to operate on the variable y_1. This type of operation is NOT time consuming since row 1 is quite sparse (only three nonzero terms appeared in the first column of matrix [L]). Thus, only three terms (the first, the fourth and the sixth terms) of the vector {f} need to be updated. Then, the next unknown, y_2, can be solved and this process can be repeated until all unknowns of the vector {y} are solved.

It should be emphasized here that in actual computer implementation, the intermediate array {y} is not needed and the forward solution phase will be overwritten on the original vector {f}.

10.7.2 Backward substitution phase

For the data shown in Eq. (10.15), Eq. (10.4) will take the following form

$$\begin{bmatrix} U_{11} & 0 & 0 & U_{14} & 0 & U_{16} \\ 0 & U_{22} & 0 & 0 & U_{25} & 0 \\ 0 & 0 & U_{33} & 0 & U_{35} & 0 \\ 0 & 0 & 0 & U_{44} & U_{45} & U_{46} \\ 0 & 0 & 0 & 0 & U_{55} & U_{56} \\ 0 & 0 & 0 & 0 & 0 & U_{66} \end{bmatrix} \begin{Bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{Bmatrix} \qquad \textbf{(10.50)}$$

The last unknown z_6 can be found easily as

$$z_6 = \frac{y_6}{U_{66}} \tag{10.51}$$

Once z_6 has been solved, one may attempt to (follow the same logic mentioned in Section 10.7.1) update the right-hand-side vector {y} by taking the sixth column of matrix [U] to operate on the variable z_6 . This kind of operation although is valid, however, is NOT preferred in practical computer implementation. This conclusion can be drawn because in practice, the upper triangular matrix [U] is stored in a 1-D array {A} according to a row-by-row fashion. Thus, it is neither convenient, nor efficient to locate nonzero terms of column 6 and multiply with variable z_6 in order to update the vector {y}.

In general, assuming z_n, z_{n-1},..., z_{i+1} have been solved, then the next unknown z_i can be obtained by simply operating a few non-zero terms of the i^{th} row (of the [U] matrix) on the unknown variables z_{i+1}, z_{i+2},...., z_{n-1}, z_n.

As an example, for the data shown in Eq. (10.15) and assuming that variables z_6, z_5, and z_4 have been solved, then the unknown variable z_3 can be found as

$$z_3 = \frac{y_3 - U_{35} * z_5}{U_{33}} \tag{10.52}$$

10.8 Sparse Solver with Improved Strategies

In this section, several strategies which can be used to improve the performance of the developed sparse solver will be discussed.

10.8.1 Finding master (or super) degree-of-freedoms (dof)

To simplify the discussion, assuming that upon completion of the symbolic phase, the stiffness matrix [K] will have the following form

$$[K] = SYM. \quad \begin{array}{ccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & \\ x & x & x & & & x & & & x & x & x & & x & x & 1 \\ & x & x & & & x & & & x & x & x & & x & x & 2 \\ & & x & & & x & & & x & x & x & & x & x & 3 \\ & & & x & x & & x & x & & & & & x & & 4 \\ & & & & x & & x & x & & & & & x & & 5 \\ & & & & & x & x & x & F & F & F & & x & F & 6 \\ & & & & & & x & x & x & x & F & & x & x & 7 \\ & & & & & & & x & x & x & F & & x & x & 8 \\ & & & & & & & & x & x & F & & x & x & 9 \\ & & & & & & & & & x & F & & x & x & 10 \\ & & & & & & & & & & x & x & x & x & 11 \\ & & & & & & & & & & & x & x & x & 12 \\ & & & & & & & & & & & & x & x & 13 \\ & & & & & & & & & & & & & x & 14 \end{array} \tag{10.53}$$

In Eq. (10.53) the stiffness matrix [K] has 14 dof. The symbols "x" and "F" refer to the original nonzero terms, and the nonzero terms due to "Fills-in," respectively. It can be seen that rows 1-3 have the same nonzero patterns (by referring to the enclosed "rectangular" region, and ignoring the fully populated "triangular" region of rows 1-3).

Similarly, rows 4-5 have the same nonzero patterns. Rows 7-10 have the same nonzero patterns. Finally, rows 11-14 also have the same nonzero patterns. Thus, for the data shown in Eq. (10.53), the "Master" (or "Super") dof can be generated as

$$MASTER \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 = N \end{Bmatrix} = \begin{Bmatrix} 3 \\ 0 \\ 0 \\ 2 \\ 0 \\ 1 \\ 4 \\ 0 \\ 0 \\ 0 \\ 4 \\ 0 \\ 0 \\ 0 \end{Bmatrix} \tag{10.54}$$

According to Eq. (10.54), then the "master" (or "super") dof are dof #1 (which is followed by 2 "slave" dof), dof #4 (which is followed by 1 slave dof), dof #6 (which has no slave dof!), dof #7 (which is followed by 3 slave dof), and dof #11 (which is followed by 3 slave dof).

10.8.2 Sparse matrix (with unrolling strategies) times vector

In our developed sparse equation solver, upon obtaining the solutions, the user has the option to compute the **r**elative **e**rror **n**orm (see R.E.N. in Section 10.9). For the error norm computation, one needs to have efficient sparse matrix (with unrolling strategies) vector multiplication.

To facilitate the discussions, let us consider the coefficient (stiffness) matrix as shown in Figure 10.12. This 14 dof matrix is symmetrical, and it has same nonzero patterns as the one considered earlier in Eq. (10.53). The master/slave dof for this matrix has been discussed and given in Eq. (10.54). The input data file associated with Figure 10.12 follows exactly the same sparse numerical factorization procedures discussed earlier in arrays ISR(-), ICN(-), AN(-) and AD(-).

The sparse matrix-vector [A] * {x} multiplication (with unrolling strategies) can be described by the following step-by-step procedures (please also refer to Figure 10.12)

Step 0.1:Multiplications between the given diagonal terms of [A] and vector {x}

Step 0.2:Consider the first "master" dof. According to Figure 10.12 (and Eq. 10.54), the first master dof is at row #1, and this master dof has 2 associated slave dof. In other words, the first 3 rows of Figure 10.12 have the same off-diagonal, nonzero patterns

Step 1: The first three rows (within a rectangular box) of the given matrix [A] (shown in Figure 10.12) operate on the given vector {x}

Step 2: The first 3 columns (within a rectangular box) of the given matrix [A] (shown in Figure 10.12) operate on the given vector {x}

Step 3: The upper and lower triangular portions (right next to the first 3 diagonal terms of the given matrix [A]) operate on the given vector {x}, according to the orders a, then b, and finally c (as shown in Figure 10.12)

Step 4: The row number corresponds to the next "master" dof can be easily computed (using the master/slave dof information, provided by Eq. 10.54).

If the next "master" dof number exceeds N (where N = total number of dof of the given matrix [A]), then stop, or else return to step 0.2 (where the "first" master dof will be replaced by the "second" master dof, etc ....)

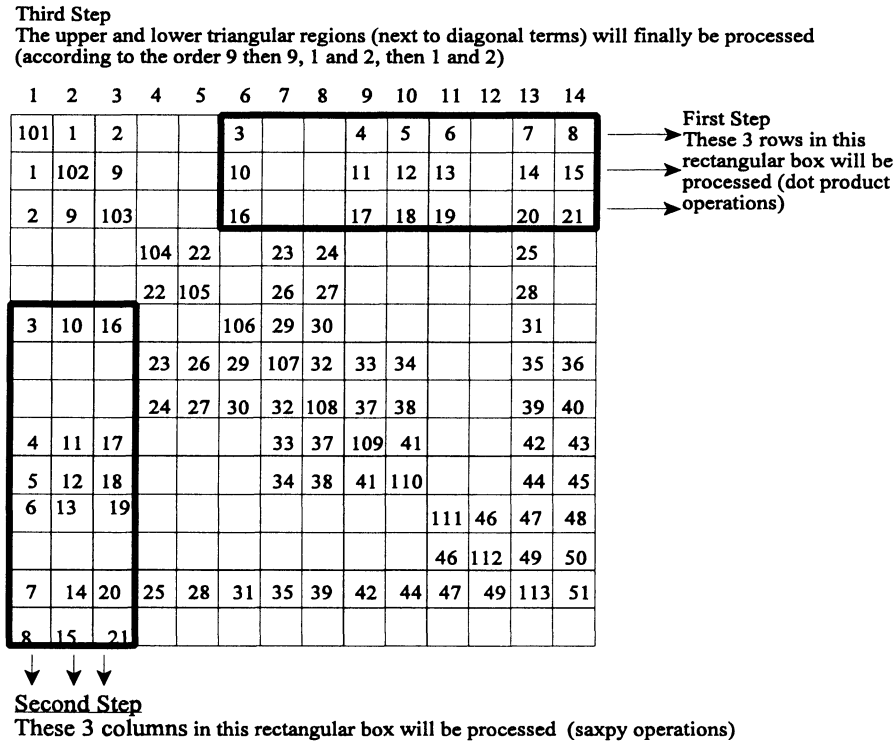


Figure 10.12 Space matrix - vector multiplications with unrolling strategies

The above step-by-step procedures (for sparse matrix-vector multiplication with master/slave dof and unrolling strategies) have been implemented into the FORTRAN computer code, shown in Table 10.6.

Explanations for major block of FORTRAN statements (in Table 10.6) will be given in the following paragraphs (see subroutine multspa)

Lines 8-9: Step 0.1 is implemented

Line 11: The first master dof (or supernode) is considered

Line 15: Unrolling level 4 strategies are assumed

Line 16: The total number of master and slave rows (corresponding to the current master row) is identified in variable "jj". For the example data shown in Figure 10.12, we have:

JJ = isupern (1) = 3

However, for the sake of discussion, let us assume JJ = 11

Line 17: Since level 4 unrolling strategies are used (see line 15), here we want to find how many blocks (each block contains 4 rows) to be associated with the above 11 rows (which correspond to the current master row). Thus:

KK = (11/4) * 4 = (2) * 4 = 8

For our actual data (shown in Figure 10.12), then: KK = (3/4) * 4 = (0) * 4 = 0

Thus, the first 8 rows (out of the total 11 rows) will be considered first (see lines 19-68). The number of remaining rows, in general, can be either 1, or 2, or 3, and is calculated in line 69.

Thus:

LEFTOV (= LEFT OVER) = 11 - 8 = 3 (see line 69)

Depending on the number of remaining rows (1, 2, or 3), the code will branch to line 133, 108, or 72, accordingly. For the actual data shown in Figure 10.12, since JJ = 3 (see line 16), hence lines 19-68 will be skipped. Furthermore, the remaining rows can be calculated (according to line 69) as:

LEFTOV = 3 - 0 = 3

Thus, the code will branch to line 72

Lines 72-76: The row number corresponds to the last row (of the remaining rows for the current master/slave rows) is calculated in line 74. The row numbers of the preceeding 2 rows are calculated in lines 75-76.

Line 77: The total number of nonzero terms (corresponding to the last row) is calculated.

Lines 78-80: The starting locations for each of the remaining rows are calculated

Lines 86-89: Step 1 is implemented (please also refer to Figure 10.12)

Lines 90-93: Step 2 is implemented (refer to Figure 10.12)

Lines 99-100: Step 3a is implemented (refer to Figure 10.12)

Line 103: Step 3b is implemented (refer to Figure 10.12)

Lines 105-106: Step 3c is implemented (refer to Figure 10.12)

The FORTRAN statements used in lines 19-68, or lines 108-132, or lines 133-147 follow the "same logic" as explained in lines 86-106.

Lines 150-151: Step 4 is implemented.

Table 10.6 Master dof and sparse matrix times vector

```
subroutine multspa(n,istartr,kindx,coefs,diag,rhs,answer,isupern) !   001
   implicit real*8(a-h,o-z) !                                         002
   common/junk1/lastrmi(8) !                                          003
C...purpose: <sparse, and symmetric> matrix times vector !            004
```

```
C.. with UNROLLING capability  !                                    005
   dimension diag(1),kindx(1),coefs(1),rhs(1),istartr(1)  !         006
   $,answer(1),isupern(1)  !                                        007
C...dimension kptrs(1)
C...starting with diagonal multiplications
    do 3 i=1,n  !                                                   008
 3  answer(i)=diag(i)*rhs(i)  !                                     009
C   write(6,*) 'MVSPARU: diag*rhs= ',(answer(i),i=1,n)
C.....first supernode  !                                            010
    i=1  !                                                          011
C... subsequent supernodes  !                                       012
 1000 continue  !                                                   013
C.....number of rows (equations) in the i-th supernode  !           014
     nunrol=4  !                                                    015
     jj=isupern(i)  !                                               016
    kk=(jj/nunrol)*nunrol  !                                        017
C---------------------------------------------------------------
    lastrow=i-1  !                                                  018
C....write(6,*) 'MVSPARU:jj,kk,lastrow= ',jj,kk,lastrow
    do 31 j=1,kk,nunrol  !                                          019
C....find the last row # in a block (each block = "nunrol" rows)  !  020
    lastrow=lastrow+nunrol  !                                       021
    lastrm1=lastrow-1  !                                            022
    lastrm2=lastrow-2  !                                            023
    lastrm3=lastrow-3  !                                            024
C.....
    lastrmi(1)=lastrm3  !                                           025
    lastrmi(2)=lastrm2  !                                           026
    lastrmi(3)=lastrm1  !                                           027
C....ii=kptrs(lastrow)
    ii=istartr(lastrow+1)-istartr(lastrow)  !                       028
C.....if(ii.eq.0) go to ???
    icount=istartr(lastrow)-1  !                                    029
    icaum1=istartr(lastrm1)  !                                      030
    icaum2=istartr(lastrm2)+1  !                                    031
    icaum3=istartr(lastrm3)+2  !                                    032
    do 32 k=1,ii    !                                               033
    icount=icount+1  !                                              034
    icaum1=icaum1+1  !                                              035
```

```
      icaum2=icaum2+1  !                                                  036
      icaum3=icaum3+1  !                                                  037
      jcoln=kindx(icount)  !                                              038
C.....upper portions (vector unrolling)  !                                039
      answer(lastrow)=answer(lastrow)+coefs(icount)*rhs(jcoln)  !         040
      answer(lastrm1)=answer(lastrm1)+coefs(icaum1)*rhs(jcoln)  !         041
      answer(lastrm2)=answer(lastrm2)+coefs(icaum2)*rhs(jcoln)  !         042
      answer(lastrm3)=answer(lastrm3)+coefs(icaum3)*rhs(jcoln)  !         043
C.... write(6,*) 'MVSPARU:icount,icaum1,icaum2,icaum3,jcoln,lastrow= '
C.... write(6,*) icount,icaum1,icaum2,icaum3,jcoln,lastrow
C.... write(6,*) 'MVSPARU: answer(lastrow),answer(lastrm3)= '
C.... write(6,*) answer(lastrow),answer(lastrm3)
C.....lower portions (loop unrolling)  !                                  044
      answer(jcoln)=answer(jcoln)+coefs(icount)*rhs(lastrow)  !           045
     $                    +coefs(icaum1)*rhs(lastrm1)  !                  046
     $                    +coefs(icaum2)*rhs(lastrm2)  !                  047
     $                    +coefs(icaum3)*rhs(lastrm3)  !                  048
C....write(6,*) 'MVSPARU: answer(jcoln)= ',answer(jcoln)
 32 continue  !                                                           049
C....now,take care of 2 little "FULL" (upper & lower) triangular portions  !  050
C....the following FORTRAN statements can be applied in several places,   051
C....thus these statements can be placed in a form of a subroutine !     052
C....call fulltri(nunrol,istartr,kindx,answer,coefs,rhs)  !               053
C....subroutine fulltri(nunrol,istartr,kindx,answer,coefs,rhs)  !         054
C.....implicit real*8(a-h,o-z)
C.....common/junk1/lastrmi(8)
C.....dimension istartr(1),kindx(1),answer(1),coefs(1),rhs(1)
C*********
      do 33 l=1,nunrol-1  !                                               055
      nterms=nunrol-l  !                                                  056
      ithrow=lastrmi(l)  !                                                057
      icount=istartr(ithrow)-1  !                                         058
      do 34 m=1,nterms  !                                                 059
      icount=icount+1  !                                                  060
      jcoln=kindx(icount)  !                                              061
C....upper row  !                                                         062
      answer(ithrow)=answer(ithrow)+coefs(icount)*rhs(jcoln)  !           063
C.... write(6,*) 'MVSPARU: ithrow,answer(-)= ',ithrow,answer(ithrow)
C.....lower column (=symmetry with upper row)  !                          064
```

```
      answer(jcoln)=answer(jcoln)+coefs(icount)*rhs(ithrow) !                  065
C.... write(6,*) 'MVSPARU: jcoln,answer(-)= ',jcoln,answer(jcoln)
  34  continue !                                                               066
  33  continue !                                                               067
  31  continue !                                                               068
C*********
C-------------------------------------------------------------
      leftov=jj-kk !                                                           069
C.... write(6,*) 'MVSPARU: leftov= ',leftov
      if(leftov.eq.0) go to 789 !                                              070
      go to(10,20,30),leftov !                                                 071
  30 continue !                                                                072
C....vectorize by unrolling (level 3) for left over rows (of a supernode) !   073
      lastrow=i+jj-1 !                                                         074
      lastrm1=lastrow-1 !                                                      075
      lastrm2=lastrow-2 !                                                      076
C....ii=kptrs(lastrow)
      ii=istartr(lastrow+1)-istartr(lastrow) !                                 077
C.....if(ii.eq.0) go to ???
      icount=istartr(lastrow)-1 !                                              078
      icaum1=istartr(lastrm1) !                                                079
      icaum2=istartr(lastrm2)+1 !                                              080
      do 2 k=1,ii !                                                            081
      icount=icount+1 !                                                        082
      icaum1=icaum1+1 !                                                        083
      icaum2=icaum2+1 !                                                        084
      jcoln=kindx(icount) !                                                    085
C....upper portion (vector unrolling) !                                        086
      answer(lastrow)=answer(lastrow)+coefs(icount)*rhs(jcoln) !               087
      answer(lastrm1)=answer(lastrm1)+coefs(icaum1)*rhs(jcoln) !               088
      answer(lastrm2)=answer(lastrm2)+coefs(icaum2)*rhs(jcoln) !               089
C.... write(6,*) 'MVSPARU: check point # 1'
C.... write(6,*) 'MVSPARU: answer(lastrow),answer(lastrm2)'
C....  write(6,*) answer(lastrow),answer(lastrm2)
C....  lower portion (loop unrolling) !                                        090
      answer(jcoln)=answer(jcoln)+coefs(icount)*rhs(lastrow) !                 091
     $                    +coefs(icaum1)*rhs(lastrm1) !                       092
     $                    +coefs(icaum2)*rhs(lastrm2) !                       093
C... write(6,*) 'MVSPARU: jcoln,answer(jcoln)= ',jcoln,answer(jcoln)
```

```
 2  continue  !                                                          094
C.....now,take care of 2 little "FULL" (upper & lower) triangular portions  !  095
C.....call fulltri(nunrol,istartr,kindx,answer,coefs,rhs)  !             096
      icaum1=istartr(lastrm1)  !                                        097
      jcoln=kindx(icaum1)  !                                            098
      answer(lastrm1)=answer(lastrm1)+coefs(icaum1)*rhs(jcoln)  !       099
      answer(jcoln)=answer(jcoln)+coefs(icaum1)*rhs(lastrm1)  !         100
      icaum2=istartr(lastrm2)  !                                        101
      jcoln=kindx(icaum2)  !                                            102
      answer(lastrm2)=answer(lastrm2)+coefs(icaum2)*rhs(jcoln)  !       103
     $                        +coefs(icaum2+1)*rhs(jcoln+1)  !           104
      answer(jcoln)=answer(jcoln)+coefs(icaum2)*rhs(lastrm2)  !         105
      answer(jcoln+1)=answer(jcoln+1)+coefs(icaum2+1)*rhs(lastrm2)  !   106
C....write(6,*) 'MVSPARU: check point # 11'
      go to 789  !                                                      107
 20  continue  !                                                       108
C.....vectorize by unrolling (level 2) for left over rows (of a supernode)  !  109
      lastrow=i+jj-1  !                                                 110
      lastrm1=lastrow-1  !                                              111
C... ii=kptrs(lastrow)
      ii=istartr(lastrow+1)-istartr(lastrow)  !                         112
C.....if(ii.eq.0) go to ???
      icount=istartr(lastrow)-1  !                                      113
      icaum1=istartr(lastrm1)  !                                        114
      do 12 k=1,ii  !                                                   115
      icount=icount+1  !                                                116
      icaum1=icaum1+1  !                                                117
      jcoln=kindx(icount)  !                                            118
C....upper portions  !                                                  119
      answer(lastrow)=answer(lastrow)+coefs(icount)*rhs(jcoln)  !       120
      answer(lastrm1)=answer(lastrm1)+coefs(icaum1)*rhs(jcoln)  !       121
C.....lower portions  !                                                 122
      answer(jcoln)=answer(jcoln)+coefs(icount)*rhs(lastrow)  !         123
     $                        +coefs(icaum1)*rhs(lastrm1)  !            124
 12  continue  !                                                       125
C.... write(6,*) 'MVSPARU: check point # 21'
C.....now,take care of 2 little "FULL" (upper & lower) triangular portions  !  126
C.....call fulltri(nunrol,istartr,kindx,answer,coefs,rhs)  !             127
      icaum1=istartr(lastrm1)  !                                        128
```

```
      jcoln=kindx(icaum1)  !                                              129
      answer(lastrm1)=answer(lastrm1)+coefs(icaum1)*rhs(jcoln)  !         130
      answer(jcoln)=answer(jcoln)+coefs(icaum1)*rhs(lastrm1)  !           131
      go to 789  !                                                        132
  10  continue  !                                                         133
C....NO vectorize by unrolling (level 1) for left over rows (of a supernode)  134
C....write(6,*) 'MVSPARU: check point # 31'
      lastrow=i+jj-1  !                                                   135
C.... ii=kptrs(lastrow)
      ii=istartr(lastrow+1)-istartr(lastrow)  !                           136
C....if(ii.eq.0) go to ???
      icount=istartr(lastrow)-1  !                                        137
      do 13 k=1,ii  !                                                     138
      icount=icount+1  !                                                  139
      jcoln=kindx(icount)  !                                              140
C....upper portion  !                                                     141
      answer(lastrow)=answer(lastrow)+coefs(icount)*rhs(jcoln)  !         142
C.....lower portion  !                                                    143
      answer(jcoln)=answer(jcoln)+coefs(icount)*rhs(lastrow)  !           144
  13  continue  !                                                         145
C.... write(6,*) 'MVSPARU: check point # 41'
C.....for this case (left over 1 row from unrolling),there is NO 2 FULL  !  146
C.....little triangular portions  !                                       147
      go to 789  !                                                        148
C.....
C.....find the row (equation) number of the next supernode  !             149
 789 i=i+jj  !                                                            150
C....  write(6,*) 'MVSPARU: check point # 41'
C....  write(6,*) 'MVSPARU: i= ',i
      if(i.lt.n) go to 1000  !                                            151
      return
      end
```

10.8.3 Modifications for the chained list array ICHAINL(-)

The chained list strategies discussed earlier in Section 10.5 need to be modified in order to include the additional information provided by the MASTER dof (refer to, for example, Eq. 10.54). The major modification that needs to be done can be accomplished by simply making sure that the chained list array ICHAINL(-) will be pointing only toward the Master dof (and not toward the slave dof!)

10.8.4 Sparse numerical factorization with unrolling strategies

The vector unrolling, and loop unrolling strategies that have been successfully introduced earlier in Refs [10.1-10.2] for skyline and variable bandwidth equation solvers, can also be effectively incorporated into the developed sparse solver (in conjunction with the Master dof strategies).

Referring to the stiffness matrix data shown in Eq. (10.53), for example, and assuming the first 10 rows of [U] have already been completely factorized, thus our objective now is to factorize the current i^{th} (= 11^{th}) row.

By simply observing Eq. (10.53), one will immediately see that factorizing row #11 will require the information from the previously factorized row numbers 1, 2, 3, 6, 7, 8, 9, and 10 (not necessarily to be in the stated increasing row numbers!) in the "conventional" sparse algorithm. Using "loop-unrolling" sparse algorithm, however, the chained list array ICHAINL(-) will point only to the "master" dof #6, #7 and #1.

The skeleton FORTRAN code LDL^T (with full matrix) shown in Table 10.1 (refer to Section 10.2.2) should be modified as shown by the pseudo, skeleton FORTRAN code in Table 10.7.

Table 10.7 Pseudo FORTRAN skeleton code for sparse LDL^T factorization with unrolling strategies

1	c	....	Assuming row 1 has been factorized earlier
2			DO 11 I = 2, N
3			DO 22 K = Only those previous "master" rows which have contributions to current row I
4	c	....	Compute the multiplier(s) (Note: U represents L^T)
4			NSLAVEDOF = MASTER(I) - 1
5			XMULT = U (K, I) / U (K, K)
5			$XMUL_m$ = U (K + m, I) / U (K + m, K + m)
5	c	....	m = 1, 2, ... NSLAVEDOF
6			DO 33 J = appropriated column numbers of "master" row #K
7			U(I, J) = U(I, J) - XMULT * U(K, J)
7			- $XMUL_m$ * U(K + m, J)
8		33	CONTINUE
9			U(K, I) = XMULT
9			U(K+m, I) = $XMUL_m$
10		22	CONTINUE
11		11	CONTINUE

More detailed computer codings for sparse LDL^T factorization with "loop-unrolling" (level 8) strategies are given in subroutine numfa8, shown in Table 10.8.

Table 10.8 Detailed computer codings for numerical sparse factorization with unrolling strategies

```
      subroutine numfa8(n,isr,icn,ad,an,iu,jcn,di,un,ichain,kupp,isupd,iopf,   !    001
     $ ME,maxnp) !                                                                 002
C.....purpose: numerical factorization !                                           003
C.....this portion of numerical factorization has unrolling level 8 !             004
      implicit real*8(a-h,o-z) !                                                   005
      dimension isr(*),icn(*),ad(*),an(*),iu(*),jcn(*),di(*),un(*) !               006
      dimension ichain(*),kupp(*),isupd(*) !                                       007
C.....DEFINITIONS !                                                                008
C.....input:      isr,icn,an,ad   given matrix A in RR(U)U. !                      009
C.....            iu,jcn      structure of resulting matrix U in !                 010
C.....                        RR(U)O. !                                            011
C.....            n           order of matrices A and U. !                         012
C..... output:    un          numerical values of the nonzeros of !                013
C.....                        matrix U in RR(U)O. !                                014
C.....            di          inverse of the diagonal matrix D. !                  015
C..... working space: ichain      of dimension N. Chained lists of rows !          016
C.....                        associated with each column. is differnt !           017
C.....                        from the one in symbolic !                           018
C.....            kupp        of dimension N. Auxiliary pointers to !              019
C.....                        portions of rows. !                                  020
C.....            di          is used as the expanded accumulator. !               021
      DO 10 J=1,N !                                                                022
   10 ichain(J)=0 !                                                                023
      DO 130 I=1,N !                                                               024
      write(6,*) '***>    I =      <***',i !                                       025
      IH=I+1 !                                                                     026
      icuu=IU(I) !                                                                 027
      ibuu=IU(IH)-1 !                                                              028
       write(6,*) 'IH, icuu, ibuu= ',ih,icuu,ibuu !                                029
      IF(ibuu.LT.icuu)GO TO 40 !                                                   030
      DO 20 J=icuu,ibuu !                                                          031
   20 DI(jcn(J))=0. !                                                              032
      IAA=isr(I) !                                                                 033
      IAB=isr(IH)-1 !                                                              034
```

```
    write(6,*) 'IAA,IAB= ',iaa,iab  !                                    035
   IF(IAB.LT.IAA)GO TO 40  !                                             036
   DO 30 J=IAA,IAB  !                                                    037
30 DI(icn(J))=AN(J)  !                                                   038
40 DI(I)=AD(I)  !                                                        039
   LAST=ichain(I)  !                                                     040
    write(6,*) 'LAST= ',last  !                                          041
   IF(LAST.EQ.0)GO TO 90  !                                              042
   LN=ichain(LAST)  !                                                    043
    write(6,*) 'LN= ',ln  !                                              044
   loop=8  !                                                             045
50 L=LN  !                                                               046
    write(6,*) 'L= ',l  !                                                047
   LN=ichain(L)  !                                                       048
   m= min(i-l,isupd(l))  !                                               049
   iend=(m/loop)*loop  !                                                 050
   isbegin=l  !                                                          051
   isend=l+iend-1  !                                                     052
   write(6,*) 'LN,m,iend,isbegin,isend= ',ln,m,iend,isbegin,isend  !     053
   IUCL=kupp(isbegin)  !                                                 054
   icu1=iucl  !                                                          055
   IUDL=IU(isbegin+1)-1  !                                               056
    write(6,*) 'IUCL,icu1,IUDL= ',iucl,icu1,iudl  !                      057
   kupp(l)=iucl + 1  !                                                   058
   length=IUDL-IUCL+1  !                                                059
    write(6,*) 'kupp(',L,')= ',kupp(l)  !                                060
    write(6,*) 'length= ',length  !                                      061
   do is= isbegin,isend,8  !                                             062
   IUC2=IU(is+2)-length  !                                               063
   IUC3=IU(is+3)-length  !                                               064
   IUC4=IU(is+4)-length  !                                               065
   IUC5=IU(is+5)-length  !                                               066
   IUC6=IU(is+6)-length  !                                               067
   IUC7=IU(is+7)-length  !                                               068
   iuc8=IU(is+8)-length  !                                               069
    write(6,*) 'is,IUCL,IUC2,...,IUC8= ',is,IUCL,IUC2,IUC3,IUC4  !      070
  $                              ,IUC5,IUC6,IUC7,IUC8
   UM1=UN(IUCL)*DI(is)  !                                                071
   UM2=UN(IUC2)*DI(is+1)  !                                              072
```

```
      UM3=UN(IUC3)*DI(is+2) !                                          073
      UM4=UN(IUC4)*DI(is+3) !                                          074
      UM5=UN(IUC5)*DI(is+4) !                                          075
      UM6=UN(IUC6)*DI(is+5) !                                          076
      UM7=UN(IUC7)*DI(is+6) !                                          077
      UM8=UN(IUC8)*DI(is+7) !                                          078
      DO 68 J=IUCL,IUDL !                                              079
      JJ=jcn(J) !                                                      080
   68 DI(JJ)=DI(JJ)-UN(J)*UM1-un(iuc2-iucl+j)*um2 !                    081
     +     -un(iuc3-iucl+j)*um3-un(iuc4-iucl+j)*um4 !                 082
     +     -un(iuc5-iucl+j)*um5-un(iuc6-iucl+j)*um6 !                 083
     +     -un(iuc7-iucl+j)*um7-un(iuc8-iucl+j)*um8 !                 084
      UN(IUCL)=UM1 !                                                   085
      un(iuc2)=um2 !                                                   086
      un(iuc3)=um3 !                                                   087
      un(iuc4)=um4 !                                                   088
      un(iuc5)=um5 !                                                   089
      un(iuc6)=um6 !                                                   090
      un(iuc7)=um7 !                                                   091
      un(iuc8)=um8 !                                                   092
      iucl=iu(is+9)-length !                                           093
      iudl=iucl+length-1 !                                             094
       write(6,*) 'IUCL,IUDL= ',iucl,iudl !                            095
      enddo !                                                          096
C.....loop of level 7,6,5,4,3,2,1
      iloop=m-iend !                                                   097
       write(6,*) 'iloop = m-iend = ',iloop !                          098
      if (iloop.eq.0) go to 77 !                                       099
      go to(1,2,3,4,5,6,7)iloop !                                      100
      go to 77 !                                                       101
C@@@@@@@@@@@@@@@@
    1 is=isend+1 !                                                     102
      UM1=UN(IUCL)*DI(is) !                                            103
      DO 61 J=IUCL,IUDL !                                              104
      JJ=jcn(J) !                                                      105
   61 DI(JJ)=DI(JJ)-UN(J)*UM1 !                                        106
      UN(IUCL)=UM1 !                                                   107
       write(6,*) 'is,IUCL= ',is,IUCL !                                108
      go to 77 !                                                       109
```

```
C@@@@@@@@@@
 2   is=isend+1  !                                                    110
     IUC2=IU(is+2)-length   !                                         111
     UM1=UN(IUCL)*DI(is)  !                                           112
     UM2=UN(IUC2)*DI(is+1)  !                                         113
     DO 62 J=IUCL,IUDL  !                                             114
     JJ=jcn(J)  !                                                     115
  62 DI(JJ)=DI(JJ)-UN(J)*UM1-un(iuc2-iucL+j)*um2  !                   116
     UN(IUCL)=UM1  !                                                  117
     un(IUC2)=um2  !                                                  118
      write(6,*) 'is,IUCL,IUC2= ',is,IUCL,IUC2  !                     119
     go to 77  !                                                      120
C@@@@@@@@@@@
 3   is=isend+1  !                                                    121
     IUC2=IU(is+2)-length  !                                          122
     IUC3=IU(is+3)-length  !                                          123
     UM1=UN(IUCL)*DI(is)  !                                           124
     UM2=UN(IUC2)*DI(is+1)  !                                         125
     UM3=UN(IUC3)*DI(is+2)  !                                         126
     DO 63 J=IUCL,IUDL  !                                             127
     JJ=jcn(J)  !                                                     128
  63 DI(JJ)=DI(JJ)-UN(J)*UM1-un(iuc2-iucl+j)*um2  !                   129
    +     -un(iuc3-iucl+j)*um3  !                                    130
     UN(IUCL)=UM1  !                                                  131
     un(iuc2)=um2  !                                                  132
     un(iuc3)=um3  !                                                  133
      write(6,*) 'is,IUCL,IUC2,...,IUC3= ',is,IUCL,IUC2,IUC3  !      134
     go to 77  !                                                      135
C@@@@@@@@@@@@
 4   is=isend+1  !                                                    136
     IUC2=IU(is+2)-length  !                                          137
     IUC3=IU(is+3)-length  !                                          138
     IUC4=IU(is+4)-length  !                                          139
     UM1=UN(IUCL)*DI(is)  !                                           140
     UM2=UN(IUC2)*DI(is+1)  !                                         141
     UM3=UN(IUC3)*DI(is+2)  !                                         142
     UM4=UN(IUC4)*DI(is+3)  !                                         143
     DO 64 J=IUCL,IUDL  !                                             144
     JJ=jcn(J)  !                                                     145
```

```
 64 DI(JJ)=DI(JJ)-UN(J)*UM1-un(iuc2-iucl+j)*um2 !                    146
  +     -un(iuc3-iucl+j)*um3-un(iuc4-iucl+j)*um4 !                  147
    UN(IUCL)=UM1 !                                                  148
    un(iuc2)=um2 !                                                  149
    un(iuc3)=um3 !                                                  150
    un(iuc4)=um4 !                                                  151
     write(6,*) 'is,IUCL,IUC2,...,IUC4= ',is,IUCL,IUC2,IUC3,IUC4 !  152
    go to 77 !                                                      153
C@@@@@@@@@@@@@@
 5   is=isend+1 !                                                   154
    IUC2=IU(is+2)-length !                                          155
    IUC3=IU(is+3)-length !                                          156
    IUC4=IU(is+4)-length !                                          157
    IUC5=IU(is+5)-length !                                          158
    UM1=UN(IUCL)*DI(is) !                                           159
    UM2=UN(IUC2)*DI(is+1) !                                         160
    UM3=UN(IUC3)*DI(is+2) !                                         161
    UM4=UN(IUC4)*DI(is+3) !                                         162
    UM5=UN(IUC5)*DI(is+4) !                                         163
    DO 65 J=IUCL,IUDL !                                             164
    JJ=jcn(J) !                                                     165
 65 DI(JJ)=DI(JJ)-UN(J)*UM1-un(iuc2-iucl+j)*um2 !                   166
  +     -un(iuc3-iucl+j)*um3-un(iuc4-iucl+j)*um4 !                  167
  +     -un(iuc5-iucl+j)*um5 !                                      168
    UN(IUCL)=UM1 !                                                  169
    un(iuc2)=um2 !                                                  170
    un(iuc3)=um3 !                                                  171
    un(iuc4)=um4 !                                                  172
    un(iuc5)=um5 !                                                  173
    go to 77 !                                                      174
C@@@@@@@@@@@@@@
 6   is=isend+1 !                                                   175
    IUC2=IU(is+2)-length !                                          176
    IUC3=IU(is+3)-length !                                          177
    IUC4=IU(is+4)-length !                                          178
    IUC5=IU(is+5)-length !                                          179
    IUC6=IU(is+6)-length !                                          180
    UM1=UN(IUCL)*DI(is) !                                           181
    UM2=UN(IUC2)*DI(is+1) !                                         182
```

```
    UM3=UN(IUC3)*DI(is+2) !                                          183
    UM4=UN(IUC4)*DI(is+3) !                                          184
    UM5=UN(IUC5)*DI(is+4) !                                          185
    UM6=UN(IUC6)*DI(is+5) !                                          186
    DO 66 J=IUCL,IUDL !                                              187
    JJ=jcn(J) !                                                      188
  66 DI(JJ)=DI(JJ)-UN(J)*UM1-un(iuc2-iucl+j)*um2 !                   189
   +     -un(iuc3-iucl+j)*um3-un(iuc4-iucl+j)*um4 !                  190
   +     -un(iuc5-iucl+j)*um5-un(iuc6-iucl+j)*um6 !                  191
    UN(IUCL)=UM1 !                                                   192
    un(iuc2)=um2 !                                                   193
    un(iuc3)=um3 !                                                   194
    un(iuc4)=um4 !                                                   195
    un(iuc5)=um5 !                                                   196
    un(iuc6)=um6 !                                                   197
    go to 77 !                                                       198
C@@@@@@@@@@
 7   is=isend+1 !                                                    199
    IUC2=IU(is+2)-length !                                           200
    IUC3=IU(is+3)-length !                                           201
    IUC4=IU(is+4)-length !                                           202
    IUC5=IU(is+5)-length !                                           203
    IUC6=IU(is+6)-length !                                           204
    IUC7=IU(is+7)-length !                                           205
    UM1=UN(IUCL)*DI(is) !                                            206
    UM2=UN(IUC2)*DI(is+1) !                                          207
    UM3=UN(IUC3)*DI(is+2) !                                          208
    UM4=UN(IUC4)*DI(is+3) !                                          209
    UM5=UN(IUC5)*DI(is+4) !                                          210
    UM6=UN(IUC6)*DI(is+5) !                                          211
    UM7=UN(IUC7)*DI(is+6) !                                          212
    DO 67 J=IUCL,IUDL !                                              213
    JJ=jcn(J) !                                                      214
  67 DI(JJ)=DI(JJ)-UN(J)*UM1-un(iuc2-iucl+j)*um2 !                   215
   +     -un(iuc3-iucl+j)*um3-un(iuc4-iucl+j)*um4 !                  216
   +     -un(iuc5-iucl+j)*um5-un(iuc6-iucl+j)*um6 !                  217
   +     -un(iuc7-iucl+j)*um7 !                                      218
    UN(IUCL)=UM1 !                                                   219
    un(iuc2)=um2 !                                                   220
```

```
      un(iuc3)=um3  !                                                  221
      un(iuc4)=um4  !                                                  222
      un(iuc5)=um5  !                                                  223
      un(iuc6)=um6  !                                                  224
      un(iuc7)=um7  !                                                  225
      go to 77  !                                                      226
C@@@@@@@@@@@@@@@@@
 77   continue  !                                                      227
     if(icu1.eq.iudl) go to 80  !                                      228
     j=jcn(icu1+1)  !                                                  229
     JJ=ichain(J)  !                                                   230
      write(6,*) 'j=jcn(icu1+1)= ',j  !                                231
      write(6,*) 'JJ=ichain(J)= ',jj  !                                232
     IF(JJ.EQ.0)GO TO 70  !                                            233
     ichain(L)=ichain(JJ)  !                                           234
      write(6,*) 'ichain(', L , ')= ', ichain(L)  !                    235
     ichain(JJ)=L  !                                                   236
      write(6,*) 'ichain(', JJ , ')= ', ichain(JJ)  !                  237
     GO TO 80  !                                                       238
  70 ichain(J)=L  !                                                    239
      write(6,*) 'ichain(', J , ')= ', ichain(J)  !                    240
     ichain(L)=L  !                                                    241
      write(6,*) 'ichain(', L , ')= ', ichain(L)  !                    242
C.....
  80 IF(L.NE.LAST)GO TO 50  !                                          243
  90 DI(I)=1.d0/DI(I)  !                                               244
    IF(ibuu.LT.icuu)GO TO 120  !                                       245
     DO 100 J=icuu,ibuu  !                                             246
 100 UN(J)=DI(jcn(J))  !                                               247
      write(6,*) 'isupd(i)= ', isupd(i)  !                             248
     if(isupd(i).eq.0) go to 130  !                                    249
     J=jcn(icuu)  !                                                    250
      write(6,*) 'J=jcn(icuu)= ',j  !                                  251
     JJ=ichain(J)  !                                                   252
      write(6,*) 'JJ=ichain(j)= ',jj  !                                253
     IF(JJ.EQ.0)GO TO 110  !                                           254
     ichain(I)=ichain(JJ)  !                                           255
      write(6,*) 'ichain(', i , ')= ', ichain(i)  !                    256
     ichain(JJ)=I  !                                                   257
```

```
     write(6,*) 'ichain(', jj , ')= ', ichain(jj)  !                    258
    GO TO 120  !                                                        259
 110 ichain(J)=I  !                                                     260
     write(6,*) 'ichain(', j , ')= ', ichain(j)  !                      261
    ichain(I)=I  !                                                      262
     write(6,*) 'ichain(', i , ')= ', ichain(i)  !                      263
 120 continue  !                                                        264
    kupp(I)=icuu  !                                                     265
     write(6,*) 'kupp(', i , ')= ', kupp(i)  !                          266
 130 continue  !                                                        267
C.....Store output results [arrays un(-) and di(-)] on fort*.files
 999  return  !                                                        268
     end  !                                                             269
```

Since the "key ideas" in sparse numerical factorization with unrolling strategies has already been explained (please refer to Table 10.7), only "major differences" between the detailed codes shown in Table 10.5 and Table 10.8 will be explained in the following paragraphs (please refer also to subroutine numfa8, in Table 10.8).

Line 24: The current I^{th} row is being factorized

Line 45: Loop-unrolling level 8 is assumed

Lines 51-52: The beginning (isbegin) and ending (isend) row numbers of a group of rows which have the same paterns (or same column numbers) of nonzero terms are identified.

Lines 54-57: The starting location for the beginning row number (isbegin) is stored in variable ICU1 (see Figure 10.13). The ending location for the beginning row number (isbegin) is stored in variable IUDL (see Figure 10.13).

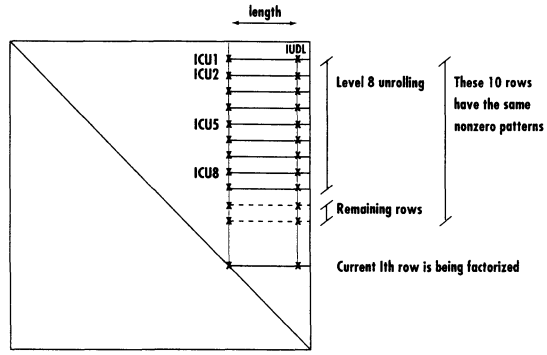


Figure 10.13 Level 8 unrolling with some remaining rows

Line 59: Knowing the starting and ending locations for the beginning row number (isbegin), the length can be computed (see Figure 10.13).

Lines 62-69: In Figure 10.13, the current row #I is assumed to require the information from the previously factorized row numbers 1 through 10. Furthermore, it is assumed that the first 10 rows will all have the same nonzero patterns. Since loop-unrolling level 8 is used, the first 8 rows will be treated first, and then, the remaining 2 rows will be treated separately.

The starting locations for rows 2 through 8 are computed in lines 63 through 69 (please also refer to Figure 10.13). The 8 multiplier factors UM1 through UM8 are computed in lines 71 through 78.

The inner-most do-loop 68 (see line 79) is expanded to make sure that the contributions from the first 8 rows are all included in factorizing the current row #I (see lines 81-84). Finally, the factorized terms U(1, I), U(2, I), ... , U(8, I) are updated in lines 85 through 92.

It is important to realize that, although the computer codes shown in Table 10.8 look much more complicated than the earlier version (without using unrolling strategies), it still looks very similar to the basic, skeleton code shown earlier in Table 10.7. In fact, lines 71 through 78 in Table 10.8 are completely equivalent to lines 5.1-5.2 in Table 10.7. Lines 81-84 in Table 10.8 are completely equivalent to lines 7.1-7.2 in Table 10.7, and lines 85-92 in Table 10.8 are completely equivalent to lines 9.1-9.2 in Table 10.7.

The starting and ending locations for the first remaining rows (such as row #9, shown in Figure 10.13) are calculated in lines 93 and 94, respectively.

Line 97: The number of remaining rows (such as rows 9-10, shown in Figure 10.13) of the same (nonzero patterns) group is calculated and stored in variable "iloop".

Lines 99-100: Since loop-unrolling level 8 is assumed in the codes, the remaining rows (of the same nonzero patterns group) can only be 0, or 1, 2, ... , 7. Thus, the code will branch to lines 227, or 102, or 110, or 121, or 136, or 154, or 175, or 199, respectively.

For the example shown in Figure 10.13, since the number of remaining rows iloop = 2, hence the code will branch to line 110.

Lines 111-118: These statements play the "same roles" as those which have already been explained earlier, in lines 63-92.

Lines 229-242: The starting location for the previously factorized row #L is updated to the next location (see ICU1 + 1, on line 229), and the corresponding column number is identified by the variable J (see line 229). The chained lists for "future" row #J (which will require the previously factorized row #L) is prepared in lines 234-242.

This segment of the codes (lines 229-242) play the same roles as described earlier in Table 10.5 (lines 33-37) for LDL^T sparse numerical factorization without using "loop-unrolling" strategies.

Line 249: Check to see if the current row #I is a master, or slave row (or dof). If row #I is a slave dof, then the code will branch to line 267 (to

consider the next row I). However, if row #I is a master dof, then the column number (corresponds to the first nonzero term of current row #I) is identified in the variable J (see line 250). The chained lists for "future" row #J (which will require the currently factorized row #I) is prepared in lines 251-262.

This segment of the codes (lines 250-262) play the same roles as described without using "loop-unrolling" strategies

Line 265: The currentearlier in Table 10.5 (lines 43-50) for LDL^T sparse numerical factorization first nonzero term of row #I is recorded in array kupp(-)

10.8.5 Out-of-core sparse equation solver with unrolling strategies

For extremely large-scale applications, the available incore memory of even a supercomputer may not be large enough to store the entire coefficient (stiffness) matrix. Thus, one needs to assemble the coefficient matrix in a block-by-block fashion, where a block may contain several rows (refer to Figure 10.14)

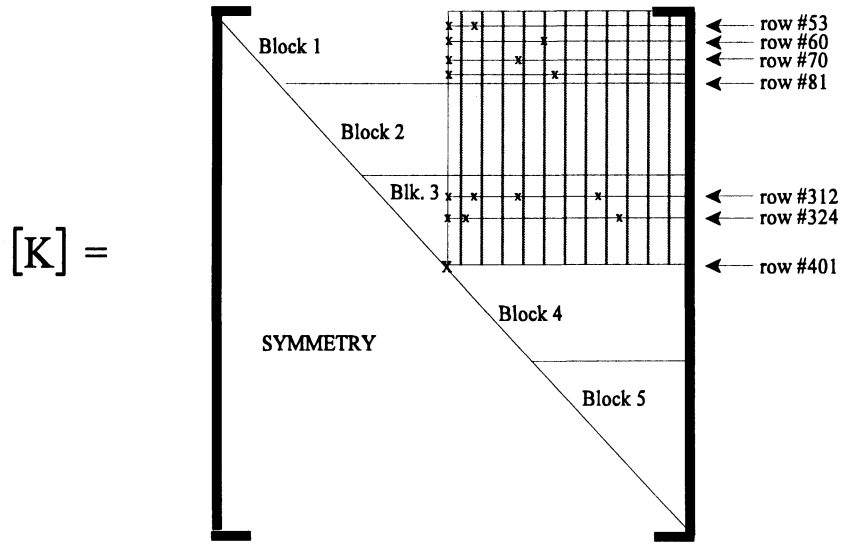


Figure 10.14 Sparse coefficient (stiffness) matrix is stored in blocks (of rows) in auxiliary storages.

The number of rows in each block will be determined by the available incore memory and the sparsity of the matrix [K]. The available incore memory will be partitioned into 2 blocks (A and B) as shown in Figure 10.15(a).

Incore Memory Available for [K]	
Block A	Block B
To store a block of "currently" factorized row(s)	To store a block of "previously" factorized rows

(a) 2-block partitioning scheme

Incore Memory Available for [K]		
Block A	Block B	Block C
"Currently" factorized row(s)	"Previously" factorized row(s)	

(b) 3-block partitioning scheme

Figure 10.15 Partitioning schemes for the available incore memory

Obviously, the size of block A (or block B) should be large enough to hold the largest block shown in Figure 10.14. Block A is used to store "currently" factorized row(s), while block B is used to store some (or all) "previously" and completely factorized rows, which have contributions to currently factorized row(s). As an example, considering the factorization of the first row of block #4 (or, say row #401 as shown in Figure 10.14), and assuming the sequence of previously factorized rows (which have contributions to the current row #401) to be in the following orders (using the chained list array ICHAINL(-) discussed in Section 10.5): rows #324, 312, 81, 70, 60 and 53.

Thus, both previous blocks #3 and #1 will have to be brought (one at a time) to the core memory (see block B of Figure 10.15a). In order to reduce the I/O time, whenever a block of rows (say, block #3 of Figure 10.14) is read into a core memory (and resided in block B, see Figure 10.15a), this block of rows' information should also be used to "partially" factorize other "future" rows (if possible), in addition to factorize the "current" i^{th} (=401^{st}) row.

On some high-performance computers, where Buffer-In/Buffer-Out capabilities (to do I/O and computation at the same time) are available (such as the Cray-YMP, Cray C90, and Intel Paragon), alternative strategies (such as the one shown in Figure 10.15b) should be considered. Using the 3-block partitioning scheme (as shown in Figure 10.15b), factorizing row #401 can be done in the following fashions:

Step 1: Buffer-In block #3 from the auxiliary storage into block B of the incore memory

Step 2: Using block #3 to partially factorize the current row (say, row #401). While these arithmetic computation is taking place, one can also buffer-in block #1 from the auxiliary storage into block C of the incore memory.

Step 3: When block #3 has been completely utilized to partially factorize the current row, then

(a) Use block #1 to factorize current row #i;

(b) buffer-out block #3 to the auxiliary storage

The above 3-step procedure can be repeated, until the current row #i is completely factorized. Thus, the 3 incore memory blocks A, B & C should always be occupied: block A is used to stored "currently" factorized row(s), blocks B & C are used to store the "immediately needed," previously factorized rows, and the "soon needed," previously factorized rows. These out-of-core strategies can be conveniently represented in Figure 10.16

	Incore Memory Blocks		
	Block A	Block B	Block C
Time = t_1	"Currently" factorized rows	"Immediately" needed rows	"Soon" needed rows (Buffer-In)
Time = $t_2 > t_1$	"Currently" factorized rows	"Soon" needed rows (Buffer-In)	"Immediately" needed rows
Time = $t_3 > t_2$	"Currently" factorized rows	"Immediately" needed rows	"Soon" needed rows (Buffer-In)
Time = $t_4 > t_3$	"Currently" factorized rows	"Soon" needed rows (Buffer-In)	"Immediately" needed rows

Figure 10.16 Out-of-core sparse factorization using three in-core-memory blocks

10.9 Numerical Performance of the Developed Sparse Equation Solver

Based upon the discussions in previous sections, several practical finite element models (such as Exxon Off-Shore Structure, High Speed Civil Transport Aircraft, Space Shuttle Solid Rocket Booster, and Automobile Structure) are used to evaluate the performance of the developed sparse solver. Since the codes have been written in standard FORTRAN language (and without using any library subroutines), it can be ported to different computer platforms (such as SUN-Sparc, IBM-R6000/590, Intel Paragon, Cray-C90 etc ... .) with no (or minimum) changes to the codes. The accuracy of the developed codes for solving system of linear equations can be measured by the Relative Error-Norm (=R.E.N.) which can be computed as

$$R.E.N. = \frac{\| K * Z - f \|}{\| f \|} \qquad \textbf{(10.55)}$$

Example 1: Exxon Off-Shore Structure

The finite element model for the Exxon model (refer to Figure 10.17) has been used extensively in earlier research works [10.37-10.39]. The resulted system of linear

equations from the Exxon model has 23,155 dof. The number of nonzero terms of the original stiffness matrix is 809,427. Using the Nested-Dissection (ND) algorithm, the number of nonzero terms (including "fills-in" terms) is 10,826,014. The relative error norm (or R.E.N., defined in Eq. 10.55) and the wall-clock time is presented and explained in Table 10.9.

It should be noted here that on the IBM-R6000/590 workstation, vector capability is available. Thus, the time improvements by using "master" dof, and "loop unrolling" strategies should be visible. Even though no efforts have been spent to utilize the optimized compiler, "loop unrolling" strategies do help to reduce the wall clock time by nearly a factor of 2 (wall clock time is dropped from 302.50 sec to 179.10 sec).

Table 10.9 Numerical performance of 4 practical finite element models

Example No.	Total No. dof	Total No. Nonzeros of [K] Before (and after) Factorization	R.E.N.	Time (in seconds)
1. Exxon	23155	809,427 (10,826,014)	$4.97 * 10^{-11}$	657.50 (a) 179.10 (b) 302.50 (c)
2. HSCT	16152	373,980 (2,746,286)	$2.01 * 10^{-6}$	2.25 (d)
3. SRB	54870	1,308,185 (11,987,067)	$2.28*10^{-9}$	12.5 (d)
4. Car	263096	6,267,099 (36,744,123)	$5.83 * 10^{-9}$	44.78 (e)

Notes:

(a) Sun-Sparc 20: time includes I/O, symbolic factorization, numerical "unrolling" factorization, forward/backward solution, R.E.N. computation

(b) IBM-R6000/590: (Peak Performance = 266 MFLOPS per node) same discriptions as in (a)

(c) IBM-R6000/590: same descriptions as in (b), but NOT using "unroll" strategies

(d) Single Cray-C90 processor: same description as in (a)

(e) Single Cray-C90 processor: (Peak performance = 980 MFLOPS per node)
Symbolic factorization = 2.09 seconds
Time to find "master" dof = 0.42 seconds
Numerical factorization = 41.33 seconds
Forward & backward solution = 0.94 seconds

(f) Peak performance (per node) on the Paragon and Cray-YMP are 75 MFLOPS, and 333 MFLOPS, respectively

Example 2: High Speed Civil Transport (HSCT) Aircraft
The finite element model for the HSCT aircraft (refer to Figure 10.18) has been used extensively in earlier research works [10.1-10.2, 10.7]. The resulted system of linear equations from the HSCT model has 16,152 dof. The number of nonzero terms of the original stiffness matrix is 373,980. Using the Modified Minimum Degree (MMD) algorithm, the number of nonzero terms (including "fills-in" terms) is 2,746,286. The relative error norm (or R.E.N., defined in Eq. 10.55) and the wall clock time is presented and explained in Table 10.9.

The maximum value of the unknown vector (or maximum displacement) is 0.1134 and is occurred at the 27th DOF. The timing for error-norm check, reading input files, symbolic factorization, ordering, numerical factorization and forward/backward solutions are presented in Table 10.10. The flop-rates are also shown in Table 10.10.

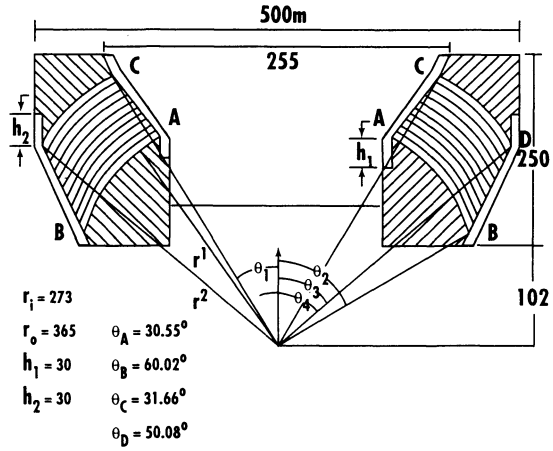


Figure 10.17 TLP flexjoint geometry parameters

Figure 10.18 High speed civil transport (HSCT) aircraft

Example 3: Solid Rocket Booster (SRB) of Space Shuttle
The finite element model for the SRB (refer to Figure 10.19) has been used extensively in earlier research works [10.1-10.2, 10.7]. The resulted system of linear equations from the SRB model has 54,870 dof. The number of nonzero terms of the original stiffness

matrix is 1,308,185. Using the Modified Minimum Degree (MMD) algorithm, the number of nonzero terms (including "fills-in" terms) is 11,987,067. the relative error norm (or R.E.N., defined in Eq. 10.55) and the wall clock time is presented and explained in Table 10.9.

The maximum value of the unknown vector (or maximum displacement) is -2.0619 and is occurred at the 47041^{th} DOF. The timing for error-norm check, reading input files, symbolic factorization, ordering, numerical factorization and forward/backward solutions are presented in Table 10.11. The flop-rates are also shown in Table 10.11.

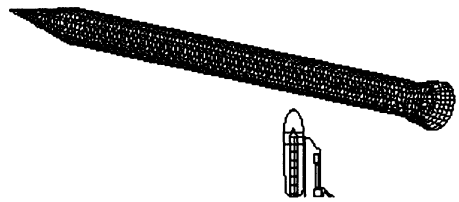


Figure 10.19 Solid rocket booster (SRB) of space shuttle

Table 10.10 Sparse (incore) solver for the HSCT aircraft model

neq	= 16152
before fill-in, ncoff	= 373980
after fill-in, ncof2	= 2746286
Maximum displacement	= 0.1133645326777 at the 27-th dof
The summation of the displacements	= 90.66547130756
Absolute error norm ‖Ax-b‖	= 4.029842721613E-4
Relative error norm ‖ Ax-b ‖ / ‖ b ‖	= 2.0149213608065E-6
Time for error norm check	= 2.6892126197993E-2
Time for reading files	= 7.579180090071
Time for symbolic factorization	= 0.147645148167
Time for reordering	= 0.123889385358
Time for numerical factorization	= 1.890624158667
Time for forward/backward solution	= 6.0497589419981E-2
Total time	= 2.252237772942
Total operations in factorization	= 781071623
Total operations for forward/backward	= 11017444
MFLOPS for factorization	= 406.7645686811

MFLOPS for forward/backward	= 182.1137685919

Table 10.11 Sparse (incore) solver for the space shuttle SRB model

neq	= 54870
before fill-in, ncoff	= 1308185
after fill-in, ncof2	= 11987067
Maximum displacement	= -2.061863838374 at the 47041-th dof
The summation of the displacements	= 13569.65122618
Absolute error norm ‖Ax-b‖	= 1.729938340402E-2
Relative error norm ‖ Ax-b ‖ / ‖ b ‖	= 2.2804218761662E-9
Time for error norm check	= 9.1727875142851E-2
Time for reading files	= 37.15652628432
Time for symbolic factorization	= 0.588571548624
Time for reordering	= 0.528286334526
Time for numerical factorization	= 11.02692366666
Time for forward/backward solution	= 0.2415057814171
Total time	= 12.501004192
Total operations in factorization	= 5416379330
Total operations for forward/backward	= 48058004
MFLOPS for factorization	= 491.1958671645
MFLOPS for forward/backward	= 198.9931823495

Example 4: Large-Scale Car Model

The finite element model for an automobile (see Figure 10.20) is used in this work [10.40] to evaluate the performance of the developed sparse algorithm on large-scale problems. The resulted system of linear equations from this car model has 263,096 dof. The number of nonzero terms of the original stiffness matrix is 6,267,099. Using the Modified Minimum Degree (MMD) algorithm, the number of nonzero terms (including "fills-in terms) is 36,744,123. The relative error norm (R.E.N., defined in Eq. 10.55) and the wall-clock time are presented and explained in Table 10.9. Other detailed results are presented in Table 10.12.

It is interesting to notice that for the example shown in Eq. 10.53, the integer array "master" (shown in Eq. 10.54) has 9 zero values.

Since more zero values in array "Master" indicate that more "slave" dof exists in the finite element model (and thus, better vector speed can be expected from unrolling techniques). For this automobile finite element model, there are 227,975 zero values. The numbers of 1, 2, 3, 4, 5, 6, 7, 8, and larger than 8 are 3493, 2095, 2085, 275, 1398, 20816, 193, 86, and 4679, respectively.

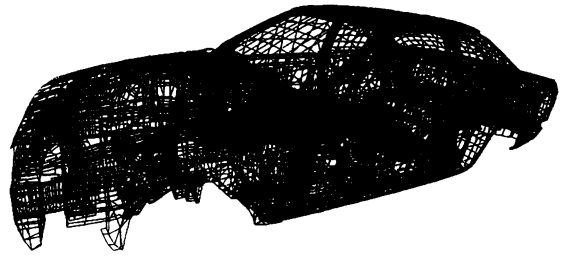


Figure 10.20 Finite element analysis of an automobile

Table 10.12 Sparse (incore) solver for the automobile model

neq	= 263096
before fill-in, ncoff	= 6267099
after fill-in, ncof2	= 36744123
Maximum displacement	= 7.8424204546417E-6 at the 78977-th dof
The summation of the displacements	= 1.5413413603372E-4
Absolute error norm ‖Ax-b‖	= 5.8339128095829E-7
Relative error norm ‖ Ax-b ‖ / ‖ b ‖	= 5.8339128095829E-9
Time for error norm check	= 0.439017051888
Time for reading files	= 4.0042494810001E-3
Time for symbolic factorization	= 2.099430224316
Time for reordering	= 1.802684694906
Time for numerical factorization	= 41.33246903718
Time for forward/backward solution	= 0.9438044399822
Total time	= 46.60164401246
Total operations in factorization	= 20750449948
Total operations for forward/backward	= 147239584
MFLOPS for factorization	= 502.0375126715
MFLOPS for forward/backward	= 156.0064540518

10.10 FORTRAN Call Statements to SPARSE Equation Solver

Based upon the discussions in the earlier sections, FORTRAN computer codes for the developed sparse equation solver has been written and presented in Table 10.13. The required input data, and a sequence of FORTRAN call statements have been presented in the main program, shown in Table 10.13 Explanations of the main

program are given in the comment statements (inserted inside Table 10.13), and in the following paragraphs:

Lines 1-5:	Dimensions for various arrays are declared
Line 8:	Input variables N, NCOEF
N	= Size of the coefficient (stiffness) matrix (= number of rows)
NCOEF	= Number of nonzero, off-diagonal (upper triangular) terms of the "original" (stiffness) matrix
Lines 9-13:	Input arrays (see explanations above)
.Line 14:	To perform the symbolic factorization (see Section 10.5)
Lines 16-17:	To perform the orderings for the matrix (see Section 10.5, Eqs. 10.40-10.46)
Line 19:	To perform the sparse numerical factorization (with unrolling strategies)
Line 22:	To perform the Forward and Backward solution phases

Table 10.13 Calling sequences for sparse equation solver with unrolling level 8

```
1    Cmain Program to Test NUMFA8*
2            implicit real *8 (a-h, o-z)
3            real *8 ad(100), an(10000), di(100), un(10000), b(100), x(100)
4            integer isr(100), icn(10000), iu(100), jcn(10000), ichain(100).
5          $ isupd(100), kupp(100), jsrt(100), jcnt(10000)
     C-----------------------------------------
6            me=1
7            maxnp=1
8            read(5,*) n,ncoef !number of rows, # nonzero off-diagonal terms
9            read(5,*) (isr(i), i=1, n+1) ! Starting locations of the first nonzero of
             each row
10           read (5,*) (icn(i), i=1, ncoef) ! Column numbers (for each row)
11           read (5, *) (ad(i), i=1,n) ! Diagonal values
12           read(5,*) (an(i), i=1, ncoef) !off-diagonal values
13           read (5,*) (b(i),i=1,n)! Values of RHS vector
     c-------------------------------------------------------------------
14           call symfact (n, isr, icn, iu, jcn, ichain, ncoef2, ME, isupd, maxnp)
15           write (6,*) 'passed symbolic factorization !"
16           call transa (n, n, iu, jcn, jsrt, jcnt)
17           call transa (n, n, jsrt, jcnt, iu, jcn)
18           write (6,*) 'passed transa twice'
19           call numfa8 (n, isr, icn,ad, an, iu, jcn, di, un, ichain, kupp, isupd, iopf,
20         $ ME,maxnp)
21           write (6,*) 'passed numerical factorization!'
```

```
call fbe (n, iu, jcn, di, un, b, x, iopb, isupd)
write(6,*) 'passed forward/backward !'
stop
end
```

10.11 Summary

Detailed discussions for the proposed sparse equation solver have been presented in this chapter. Both incore and out-of-core strategies have been explained. While the very basic, key ideas for sparse equation solution is rather straight forward, effective sparse solution can only by achieved by careful implementation of various components in a sparse solution process. Successful sparse codes will require the combinations of best available reordering algorithm(s), efficient usages of the "chained list" and optimum utilization of the vector speed (saxpy and/or dot operations) offered by modern high-performance computers. It has been demonstrated that loop-unrolling techniques can be used effectively in conjunction with sparse algorithms to fully exploit the vector capability of high-performance computers. Numerical results presented in this chapter (on medium to large-scale practical finite element models) have clearly indicated that the developed sparse algorithms and software are accurate and highly efficient.

10.12 Exercises

10.1 For the given (stiffness matrix) data shown in Figure 5.27 of Chapter 5, and using the sparse storage schemes described in Section 10.3
- (a) Define the integer array ISTARTROW(-), as explained earlier in Eq. (10.16)
- (b) Define the integer array ICOLNUM(-), as explained earlier in Eq. (10.17)
- (c) Define the real array DIAG(-), as explained earlier in Eq. (10.18)
- (d) Define the real array AK(-), as explained earlier in Eq. (10.19)

10.2 Using same data as in problem 10.1:
- (a) Without any actual computations, identify all possible fills-in factorized terms from Figure 5.27
- (b) Define the integer array JSTARTROW(-), as explained earlier in Eq. (10.20)
- (c) Define the integer array JCOLNUM(-), as explained earlier in Eq. (10.21)
- (d) After symbolic factorization phase, do we need to perform "ordering" phase for row #15 (as explained in Eqs. 10.40-10.46)? Explain your reason(s)?

10.3 For the data shown in Figure 5.27, assuming loop-unrolling level 9 is used. Find the integer array MASTER(-), as explained earlier in Eq. (10.54)?

10.4 Re-do problem 10.3, if loop-unrolling level 4 is used?

10.5 For the data shown in Figure 5.27, without using loop-unrolling strategies (thus, loop-unrolling level 1 is assumed!), following the steps 0-3 (shown in Eqs. 10.22-10.35) to define arrays ICHAINL(-), LOCUPDATE(-), only for the first 5 rows of figure 5.27?

10.6 Re-do problem 10.5, if loop-unrolling level 4 is used?

10.13 References

10.1 Pissanetzky, S., "Sparse Matrix Technology," Academic Press, Inc., London (1984).

10.2 Agarwal, T.K., Storaasli, O.O., and Nguyen, D.T., "A Parallel-Vector Algorithm for Rapid Structural Analysis on High Performance Computers," to appear in Computers and Structures Journal.

10.3 Nguyen, D.T., Storaasli, O.O., Carmona, E.A., Al-Nasra, M., Zhang, Y., Baddourah, M.A. and Agarwal, T.K., "Parallel-Vector Computation for Linear Structural Analysis and Nonlinear Unconstrained Optimization Problems," Computing Systems in Engineering, An International Journal, Vol.2, No.2/3, September 1991, (Pergamon Press), pp.175-182.

10.4 Qin, J., Gray,$^{Jr.}$, C.E., Mei, C. and Nguyen, D.T., "A Parallel-Vector Equation Solver for Unsymmetric Matrices on Supercomputers," Computing Systems in Engineering, An International Journal, Vol.2, No.2/3, September 1991 (Pergamon Press), pp.197-202.

10.5 Zhang, Y. and Nguyen, D.T., "Parallel-Vector Sensitivity Calculations in Linear Structural Dynamics," Computing Systems in Engineering Journal, Vol.3, No.1-4, pp.365-378, (1992).

10.6 Belvin, W.K., Maghami, P.G. and Nguyen, D.T., "Efficient Use of High Performance Computers for Integrated Controls and Structures Design," Computing Systems in Engineering Journal, Vol.3, No.1-4, pp.181-188, (1992).

10.7 Qin, J. and Nguyen, D.T., "A Parallel-Vector Equation Solver for Distributed Memory Computers," Computing Systems in Engineering journal, Vol.5, No.1, (1994).

10.8 Maker, B.N., Qin, J. and Nguyen, D.T., "Performance of NIKE3D with PVSOLVE on Vector and Parallel Computers," to appear in Computing Systems in Engineering Journal.

10.9 Qin, J. and Nguyen, D.T., "A Parallel-Vector Simplex Algorithm on Distributed Memory Computers," to appear in Optimization Journal.

10.10 Akan, A.O., Qin, J., Nguyen, D.T., and Basco, D.R., "Parallel Computation for Groundwater Flow," Proceedings of the International Groundwater Management Symposium, San Antonio, TX (August 1995)

10.11 Qin, J., Nguyen, D.T. and Zhang, Y., "Parallel-Vector Lanczos Eigen-Solver for Structural Vibration Problems," Proceedings of the 4th International Conference on Recent Advances in Structural Dynamics, Institute of Sound and Vibration Research, University of Southamption, Southampton, England (July 15-18, 1991).

10.12 Bailey, D.H., Barszcz, E., Dagum, L., and Simon, H.D., "NAS Parallel Benchmark Results 10-93," NASA Ames Research Center Report, ARC275, Moffett Field, CA.

10.13 Sporzynski, Steven R., "Vector/Parallel Skyline Matrix Routines for the IBM-3090," Technical Report, Washington Systems Center, IBM Corp., 18100 Frederick Pike, Gaithersburg, MD 20879 (May 1990).

10.14 Zheng, D. and Chang, T.Y.P., "Parallel Cholesky Method on MIMD with Shared Memory," Computers and Structures, Vol.56, No.1, pp.25-38 (1995).

10.15 Tong, Pin, Rossettos, John N., "Finite Element Method: Basic Technique and Implementation," the MIT Press, Cambridge, Massachusetts, and London, England.

10.16 Ortega, J.M., "Introduction to Parallel and Vector Solution of Linear Systems," Plenum Press (1988).

10.17 Khan, A.I. and Topping, B.H.V., "Parallel-Finite Element Analysis Using the Jacobi-Conditioned Conjugate Gradient Algorithm," Information Technology for Civil and Structural Engineers," B.H.V. Topping & A.I. Khan (Eds.), Civil-Comp. Press, Edinburgh, 245-255 (1993).

10.18 Khan, A.I., and Topping, B.H.V., "A Transputer Routing Algorithm for Nonlinear or Dynamic Finite Element Analysis," Engineering Computations, Vol.11, pp.549-564 (1994).

10.19 Topping, B.H.V. and Khan, A.I., "Parallel Computations for Structural Analysis, Re-Analysis and Optimization," Optimization of Large Structural Systems, Vol.II, pp.767-792 (G.I.N. Rozvany, Ed., 1993 Kluwer Academic Publishers, Netherlands).
10.20 Duff, I.S. and Stewart, G.W. (Editors), "Sparse Matrix Proceedings 1979," SIAM (1979)
10.21 Duff, I.S., Grimes, R.G. and Lewis, J.G., "Sparse Matrix Test Problems," ACM Trans. Math Software, 15, pp.1-14 (1989).
10.22 George, J.A. and Liu, W.H., "Computer Solution of Large Sparse Positive Definite Systems," Prentice-Hall, Englewood Cliffs, NJ (1981).
10.23 Damhaug, A.C., Mathisen, K.M., and Okstad, K.M., "The Use of Sparse Matrix Methods in Finite Element Codes for Structural Mechanics Applications," Department of Structural Engineering, The Norwegian Institute of Technology, N-7034 Troudheim, Norway (1993).
10.24 Noor, A.K., "Parallel Processing In Finite Element Structural Analysis," in Parallel Computations and Their Impact on Mechanics, ASME, pp.253-277, A.K. Noor (Ed.) 1987.
10.25 Law, K.H. and Mackay, D.R., "A Parallel Row-Oriented Sparse Solution Method for Finite Element Structural Analysis," Inter. Journal for Num. Meth. in Engr., Vol.36, pp.2895-2919 (1993).
10.26 Bathe, K.J., Finite Element Procedures, Prentice-Hall (1996).
10.27 Golub, G.H. and VanLoan, C.F., "Matrix Computations," Johns Hopkins University Press, Baltimore, MD, Second Edition (1989).
10.28 Cuthill, E. and McKee, J., "Reducing the Bandwidth of Sparse Symmetric Matrices," Proc. 24th Nat'l. Conf. Assoc. Comput. Mach., ACM Publ., pp.157-172 (1969).
10.29 Crane, H.L., Jr., Gibbs, N.E., Poole, Jr., W.G., and Stockmeyer, P.K., "Algorithm 508: Matrix Bandwidth and Profile Reduction," ACM Trans. on Math. Software, 2, pp.375-377 (1976).
10.30 Lewis, J.G., Peyton, B.W. and Pothen, A., "A Fast Algorithm for Reordering Sparse Matrices for Parallel Factorization," SIAM J. Sci. Statist. Comput., 6, pp.1146-1173 (1989).
10.31 Liu, J.W.H., "Reordering Sparse Matrices for Parallel Elimination," Tech. Report 87-01, Computer Science, York University, North York, Ontario, Canada (1987).
10.32 Pothen, A., Simon, H.D. and Liou, K-P., "Partitioning Sparse Matrice with Eigenvectors of Graphic," Siam J. Matrix, Vol.II, No.1, pp.430-452 (1990).
10.33 George, J.A., "Nested Dissection of a Regular Finite Element Mesh," Siam J. Numer. Anal., 15, pp.1053-1069 (1978).
10.34 Gilbert, J.R. and Zmijewski, E., "A Parallel Graph Partitioning Algorithm for a Message Passing MultiProcessor," Inter. J. Parallel Programming, 16, pp.427-449 (1987).
10.35 Leiserson, C.E. and Lewis, J.G., "Orderings for Parallel Sparse Symmetrix Factorization," Third SIAM Conference on Parallel Processing for Scientific Computing (1987).
10.36 Simon, H.D., Vu, P. and Yang, C., "Performance of a Supernodal General Sparse Solver on the Cray-YMP: 1.68 GFLOPS with Autotasking," Applied Mathematics Technical Report (SCA-TR-117), Boeing Computer Service, Scientific Computing and Analysis Division, G-8910, M/S 7L-21, P.O. Box 24346, Seattle, Washington, 98124-0346, USA.
10.37 Wang, S.M., Chang, T.Y.P. and Tong, P., "Nonlinear Deformation Responses of Rubber Components by Finite Element Analysis," Computational Mechanics `95: Theory and Applications Proceedings of the International Conference on Computational Engineering Science, July 30-August 3, 1995, Hawaii, USA (Volume 2, pp.3135-3140).
10.38 Chang, T.Y.P., Saleeb, A.F. and Li, G., "Large Strain Analysis of Rubber-like Materials Based on a Perturbed Lagrangian Variational Principal," J. Comput. Mech., Vol.8, pp.221-233, (1991).
10.39 Gunderson, R.H., "Fatigue Life of TLP Flexelements," 24th Annual OTC Conference, Houston, Texas, May 4-7, 1992.
10.40 Storaasli, O.O., NASA Langley Research Center, Hampton, VA (Private Communication).
10.41 Ng, E. and Peyton, B.W., "Block Sparse Choleski Algorithm On Advanced Uniprocessor Computer," SIAM J. of Sci. Comput., Volume 14, pp. 1034-1056, 1993.

11 Algorithms for Sparse-Symmetrical-Indefinite and Sparse-Unsymmetrical System of Equations

11.1 Introduction

For certain important classes of engineering and science applications [11.1-11.5], the coefficient matrix of the system of linear equations is no longer "positive definite." Instead, it can be symmetric (or unsymmetric) and/or "indefinite" matrix. For these problems, pivoting strategies [11.6-11.10] are often required in order to avoid numerical difficulties. For symmetric, positive definite matrix [11.11-11.19], since pivoting strategies are not required, thus it is relatively easy to accurately "predict" the amounts of fill-in terms during the factorization process. In fact, upon completion the symbolic factorization (as discussed in Section 10.5), the exact dimensions for the (soon to be) factorized matrix can be determined, hence exact memory allocations can be assigned to the factorized matrix.

However, when pivoting strategies are required, one has to switch row(s) and column(s) of the matrix. These actions will change the fill-in patterns of the coefficient matrix. Furthermore, since pivoting strategies may be required at any stages during the factorization process, it is quite difficult to have a precise prediction on the total numbers of fill-in terms "before" entering the numerical factorization phase! For these reasons, it is proposed in this work that symbolic and numerical factorization will be executed simultaneously in a row-by-row fashion.

11.2 Basic Formulation for Indefinite System of Linear Equations

Without losing any generality, it is assumed that the original coefficient (stiffness) matrix has a zero value (at the first diagonal location) as shown in Eq. (11.1).

[K] =

O	X	O	X	O	X					X				
	X	O	O	X	O		X		X					
		X	O	X	O							X		
			X	X	O			X	X	X				
				X	X						X			
					X						X			
						O	X	O	X	O	X			
	S	Y	M.				X	O	O	X	O			
								X	O	X	O	X		
									X	X	O			
										X	X		X	
											X	X		
												X		
													X	
														X

(11.1)

In Eq. (11.1), the symbol "x" represents a nonzero value, and row #1 is referred to as "sick" row (please notice a zero value on the diagonal term of row #1). Equation (11.1) can be symbolically represented as

$$[K] = \begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \qquad \textbf{(11.2)}$$

Where, using the data shown in Eq. (11.1), one can identify:

$$[A_{11}] = \begin{bmatrix} o & \times \\ \times & \times \end{bmatrix} = a\ 2\times2\ \textit{symmetrical, square submatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} \qquad \textbf{(11.3)}$$

$$[A_{12}] = a\ 2\times13\ \textit{rectangular submatrix} = [A_{21}]^T \qquad \textbf{(11.4)}$$

$$[A_{22}] = a\ 13\times13\ \textit{symmetrical, square submatrix} \qquad \textbf{(11.5)}$$

Case 1: Two-by-Two (2×2) Block Pivoting, With Remaining Block Factorization
Assuming the submatrix $[A_{11}]$ is non-singular, Eq. (11.2) can be factorized (in the partitioned form) as:
In Eq. (11.5a), the following sub-matrices are defined:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \qquad \textbf{(11.5a)}$$

$[L_{11}]$ is a 2 x 2 identity matrix
$[L_{22}]$ is a J x J identity matrix (where J=13, according to the assumed dimensions shown in Eqs.11.2-11.5).
$[L_{21}]$ is a J x 2 rectangular matrix

$[D_{11}]$ is a 2 x 2 non-diagonal matrix
$[D_{22}]$ is a J x J non-diagonal matrix

In this case, the unknowns are $[L_{21}]$, $[D_{11}]$ and $[D_{22}]$. Expanding the right-hand-side of Eq. (11.5a), and realizing that $[L_{11}]$ and $[L_{22}]$ are identity matrices, one has:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} D_{11} & D_{11} L_{21}^T \\ L_{21} D_{11} & L_{21} D_{11} L_{21}^T + D_{22} \end{bmatrix} \quad \textbf{(11.5b)}$$

Comparing both sides of Eq. (11.5b), one obtains:

$$D_{11} = A_{11} \quad \textbf{(11.5c)}$$

$$L_{21}^T = D_{11}^{-1} A_{12} \quad \textbf{(11.5d)}$$

$$D_{22} = A_{22} - L_{21} D_{11} L_{21}^T \quad \textbf{(11.5e)}$$

Substituting Eq. (11.5d) into Eq. (11.5e), one obtains:

$$D_{22} = A_{22} - A_{21} D_{11}^{-1} A_{21}^T \quad \textbf{(11.5f)}$$

Thus, either Eq. (11.5e), or Eq. (11.5f) can be used to compute the unknown sub-matrix D_{22}.

Rotation Matrix [R]:

The sub-matrix $[D_{11}]$, defined in Eq. (11.5c), can be made a diagonal matrix $[D_{11}^*]$ by the following transformation:

$$\begin{bmatrix} D_{11} & 0 \\ O & D_{22} \end{bmatrix} = \begin{bmatrix} R & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} D_{11}^* & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} R^T & 0 \\ 0 & I \end{bmatrix} \quad \textbf{(11.5g)}$$

In the above equation, $[R] = [\Phi]$ = [Eigen matrix of D_{11}], such that $R^TR = I = RR^T$ (or $R^T = R^{-1}$). Substituting Eq. (11.5g) into Eq. (11.5a), one has:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11}R & 0 \\ L_{21}R & L_{22} \end{bmatrix} \begin{bmatrix} D_{11}^* & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} R^T L_{11}^T & R^T L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \quad \textbf{(11.5h)}$$

Since L_{11} and L_{22} are identity matrices, hence Eq. (11.5h) becomes:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} RD_{11}^* R^T & RD_{11}^* R^T L_{21}^T \\ L_{21} RD_{11}^* R^T & L_{21} RD_{11}^* R^T L_{21}^T + D_{22} \end{bmatrix} \quad \textbf{(11.5i)}$$

Comparing both sides of Eq. (11.5i), one has:

$$D_{11}^* = R^T A_{11} R \quad \textbf{(11.5j)}$$

$$L_{21}^T = R[D_{11}^*]^{-1} R^T A_{12} \quad \textbf{(11.5k)}$$

or

$$L_{21} = A_{21}R[D_{11}^{*}]^{-1} R^{T} \qquad \textbf{(11.5l)}$$

$$D_{22} = A_{22}-L_{21}RD_{11}^{*} R^{T}L_{21}^{T} \qquad \textbf{(11.5m)}$$

Example 11.1:
The system of equations [A] {x} = {b} is given, where:

$$[A] = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \textit{ and } \{b\} = \begin{Bmatrix} -2 \\ 0 \\ 0 \\ 1 \end{Bmatrix}.$$

Our objectives here are simply to find the LDL^{T} (without using the rotations matrix [R]) factorization of the given matrix [A]. From Eq. (11.5 c), one has:

$$[D_{11}]=[A_{11}] = \begin{bmatrix} 0 & -1 \\ -1 & 2 \end{bmatrix} \bullet \textit{Thus,} \quad D_{11}^{-1} = \frac{\begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}}{(-1)} = \begin{bmatrix} -2 & -1 \\ -1 & 0 \end{bmatrix} \qquad \textbf{(11.5n)}$$

From Eq. (11.5d), one obtains

$$L_{21}^{T} = [D_{11}]^{-1}\,[A_{12}] = \begin{bmatrix} -2 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \qquad \textbf{(11.5p)}$$

From Eq. (11.5f), one has

$$[D_{22}] = [A_{22}]-[A_{21}]\,[D_{11}]^{-1}\,[A_{21}]^{T}$$

or

$$[D_{22}] = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}-\begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} -2 & -1 \\ -1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

or

$$D_{22} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \qquad \textbf{(11.5q)}$$

The LDL^{T} factorization of A, in the partitioned form, can be expressed as shown in Eq. (11.5a):

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}\begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix}\begin{bmatrix} L_{11}^{T} & L_{21}^{T} \\ 0 & L_{22}^{T} \end{bmatrix}$$

In the above matrix equations, submatrices [L_{11}] and [L_{22}] are identity matrices.

Furthermore, the above matrix equations are satisfied when all submatrices (in left-hand-side and right-hand-side) of the above equations are replaced by their numerical values.

Example 11.2:
Using the same data as shown in Example 11.1, and using the rotation matrix [R] to factorize the given matrix [A] according to Eq. (11.5h), one has:
[T] = Eigen matrix of $[D_{11}]$, such that $T^T T = [I] = TT^T$

Hence from Eq. (11.5n), one obtains: $\det\begin{vmatrix} 0-\lambda & -1 \\ -1 & 2-\lambda \end{vmatrix} = 0 = -2\lambda + \lambda^2 - 1$. The 2 roots (or eigen values) can be computed as :

$$\lambda = \frac{2 \pm \sqrt{8}}{2} = \frac{2 \pm 2\sqrt{2}}{2} = 1 \pm \sqrt{2}. \quad \textit{Therefore,} \quad \boxed{\begin{matrix} \lambda_1 = 1 - \sqrt{2} \\ \lambda_2 = 1 + \sqrt{2} \end{matrix}}$$

- *For* $\lambda = \lambda_1 = 1 - \sqrt{2}$

From the eigen-equations $[D_{11}]\{\phi\} = \lambda\{\phi\}$, one has

$$\begin{bmatrix} \sqrt{2}-1 & -1 \\ -1 & 1+\sqrt{2} \end{bmatrix} \begin{Bmatrix} \phi_1^1 \\ \phi_2^1 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \qquad \textbf{(11.5r)}$$

Letting $\phi_1^1 = 1$, Eq. (11.5r) can be solved and normalized as

$$\phi^{(1)} = \begin{Bmatrix} 1 \\ \sqrt{2} \quad -1 \end{Bmatrix} * \left(\frac{1}{1.0824} \right)$$

$$\textit{or } \phi^{(1)} = \begin{Bmatrix} \frac{1}{1.0824} \\ \frac{0.4142}{1.0824} \end{Bmatrix}$$

- *For* $\lambda = \lambda_2 = 1 + \sqrt{2}$

Similar to Eq. (11.5r), one obtains

$$\begin{bmatrix} -1-\sqrt{2} & -1 \\ -1 & 1-\sqrt{2} \end{bmatrix} \begin{Bmatrix} \phi_1^2 \\ \phi_2^2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \qquad \textbf{(11.5s)}$$

Letting $\phi_1^2 = 1$, Eq. (11.5s) can be solved and normalized as

$$\phi^{(2)} = \begin{Bmatrix} 1 \\ -1-\sqrt{2} \end{Bmatrix} * \left(\frac{1}{2.6131} \right)$$

$$or\ \phi^{(2)} = \begin{Bmatrix} \phi_1^2 \\ \phi_2^2 \end{Bmatrix} = \begin{Bmatrix} \frac{1}{2.6131} \\ \frac{-2.4142}{2.6131} \end{Bmatrix}$$

The rotation matrix [R] which consists of the eigen-vectors of $[D_{11}]$ can be given as

$$[R] \equiv \Phi = \begin{bmatrix} \frac{1}{1.0824} & \frac{1}{2.6131} \\ \frac{0.4142}{1.0824} & \frac{-2.4142}{2.6131} \end{bmatrix} = \begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix} \equiv T$$

Since:

$$\Phi^T\Phi = I = \Phi\Phi^T = T^TT = TT^T$$

Therefore, from Eqs. (11.5j) and (11.5n), one has:

$$D_{11}^* = R^T A_{11} R = \begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix}$$

$D_{11}^* = \begin{bmatrix} -0.4142 & 0 \\ 0 & 2.4143 \end{bmatrix}$. Thus, the inverse of D_{11}^* can be obtained as:

$$[D_{11}^*]^{-1} = \begin{bmatrix} -2.4143 & 0 \\ 0 & 0.4142 \end{bmatrix}$$

From Eq. (11.5k), one obtains:

$$L_{21}^T = \begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix} \begin{bmatrix} -2.4143 & 0 \\ 0 & 0.4142 \end{bmatrix} \begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix}$$

$L_{21}^T = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$. *Therefore, one computes*: $R^T L_{21}^T = \begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.9239 & 0 \\ 0.3827 & 0 \end{bmatrix}$

From Eq. (11.5 m), one obtains

$$D_{22} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix} \begin{bmatrix} -0.4142 & 0 \\ 0 & 2.4143 \end{bmatrix} \begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

$$D_{22} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$$

Thus, from Eq. (11.5h), one does verify that the following equation is correct:

$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 0.5239 & 0.3827 & 0 & 0 \\ 0.3827 & -0.9239 & 0 & 0 \\ 0.9239 & 0.3827 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0.4142 & 0 & 0 & 0 \\ 0 & 2.4143 & 0 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0.9239 & 0.3827 & 0.9239 & 0 \\ 0.3827 & -0.9239 & 0.3827 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Case 2: Two-by-Two (2x2) Block Pivoting, with Remaining Row-by-Row Factorization
Assuming the submatrix $[A_{11}]$ is non-singular, then Eq. (11.2) can be factorized as indicated in Eq. (10.9), or as shown in the following partitioned form (see Figure 11.1):

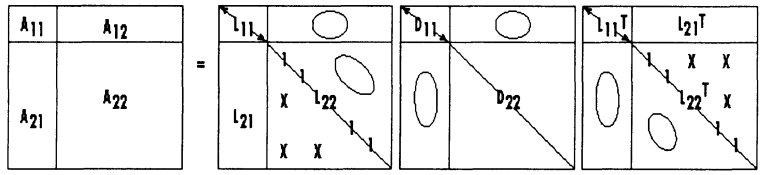


Figure 11.1 LDL^T with 2x2 block pivoting

In Fig. 11.1, the submatrix $[L_{11}]$ is a 2x2 identity matrix, the 2x2 submatrix $[D_{11}]$ is "NOT" a diagonal matrix, the JxJ submatrix $[D_{22}]$ is a diagonal matrix (where J = n-2, and n is the total number of degree-of-freedoms of the matrix [K], in Eq. 11.2), the JxJ submatrix $[L_{22}]$ is a lower triangular matrix with unit values on its diagonal, and the Jx2 submatrix $[L_{21}]$ is a rectangular matrix. Thus, in this case, the unknown sub-matrices are L_{21}^T, D_{11}, D_{22} and L_{22}.

Expanding the right-hand-side of Figure 11.1, one obtains:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} D_{11} & D_{11}L_{21}^T \\ L_{21}D_{11} & L_{21}D_{11}L_{21}^T + L_{22}D_{22}L_{22}^T \end{bmatrix} \quad \textbf{(11.6)}$$

Equating both sides of Eq. (11.6), one has

$$D_{11} = A_{11} \quad \textbf{(11.7)}$$

$$L_{21}^T = D_{11}^{-1} A_{12} \quad \textbf{(11.8)}$$

$$L_{22}\, D_{22}\, L_{22}^T = A_{22}^* \quad \textbf{(11.9)}$$

where

$$A_{22}^* \equiv A_{22} - L_{21} D_{11} L_{21}^T \quad \textbf{(11.10)}$$

The basic procedures for a 2x2 pivoting strategies for sparse factorization can be summarized in the following step-by-step algorithms:

Step 1: Eq. (11.7) is used to compute $[D_{11}]$
Step 2: Eq. (11.8) is used to compute $[L_{21}^T]$
Step 3: Eq. (11.10) is used to compute $[A_{22}^*]$

Step 4: Knowing $[A_{22}^*]$, its triple product $L_{22}D_{22}L_{22}^T$ can be computed in an usual fashion as explained earlier in Section 10.6.

11.3 Rotation Matrix [R] Strategies

Since the 2 x 2 submatrix $[D_{11}]$, shown in Figure 11.1, is a "non-diagonal" matrix, factorizing the subsequent rows (after the "sick" row) are not convenient. These inconveniences are mainly due to the facts that the previous rows which have the contributions to the currently factorized i^{th} row can be processed 1 row at a time, with the exception of the "sick" 2x2 block (which contains the sick row and the next row)! Thus, it is desirable to uncouple the "sick" 2x2 block. In other words, one would like to transform the non-diagonal matrix $[D_{11}]$ into a diagonal matrix $[D_{11}^*]$, through the following transformation.

$$\begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \equiv \begin{bmatrix} R & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} D_{11}^* & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} R^T & 0 \\ 0 & I \end{bmatrix} \tag{11.11}$$

In Eq. (11.11), [R] is the 2x2 rotation matrix whose columns are the normalized eigenvectors of the 2x2 matrix $[D_{11}]$. The rotation matrix [R] is also normalized so that

$$[R]\,[R]^T = [I] = [R]^T\,[R] \tag{11.12}$$

From a geometry view point, the rotation matrix [R] can also be expressed as

$$[R] = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

In the above form, only 1 parameter (say, angle θ) needs be defined for recovering the matrix [R]. Expanding the right-hand-side of Eq. (11.11), one obtains

Thus:
$$\begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} = \begin{bmatrix} RD_{11}^* R^T & 0 \\ 0 & D_{22} \end{bmatrix} \tag{11.13}$$

$$D_{11} = R\, D_{11}^*\, R^T \tag{11.14}$$

or
$$D_{11}^* = R^T D_{11} R \tag{11.15}$$

Now, substituting Eq. (11.11) into the right-hand-side of the equation shown in Figure 11.1, one obtains

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \equiv \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} R & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} D_{11}^* & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} R^T & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \tag{11.16}$$

Expanding the right-hand-side of Eq. (11.16), one has

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \equiv \begin{bmatrix} L_{11}R & 0 \\ L_{21}R & L_{22} \end{bmatrix} \begin{bmatrix} D_{11}^* & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} R^T L_{11}^T & R^T L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \tag{11.17}$$

where:

$[L_{11}] = [I]_{2x2}$
[R] = Eigen-matrix of $[D_{11}]$
$[D_{11}^*]$ and $[D_{11}]$ are defined in Eqs. (11.14) and (11.15)
$[L_{22}]$ and $[D_{22}]$ are defined in Eq. (11.9)
$[L_{21}]$ is defined in Eq. (11.8)

Example 11.3: Resolve the same example 11.1, using the formulation in case 2 with Rotation matrix [R]. From Eq. (11.7),

$$[D_{11}]=[A_{11}]=\begin{bmatrix}0 & -1\\ -1 & 2\end{bmatrix} \bullet \textit{Thus, } [\Phi = \textit{eigen-vectors of } D_{11}]=\begin{bmatrix}0.9239 & 0.3827\\ 0.3827 & -0.9238\end{bmatrix}=[R]$$

From Eq. (11.5j), one has $[D_{11}^*]=[\Phi]^T [D_{11}] [\Phi]=\begin{bmatrix}-0.4142 & 0\\ 0 & 2.4143\end{bmatrix}$

From Eq. (11.8), one has $[L_{21}^T]=[D_{11}]^{-1} A_{12}$, *where* $[D_{11}]^{-1} = \begin{bmatrix}-2 & -1\\ -1 & 0\end{bmatrix}$

$$L_{21}^T=\begin{bmatrix}-2 & -1\\ -1 & 0\end{bmatrix}\begin{bmatrix}0 & 0\\ -1 & 0\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix} \bullet \textit{Therefore, } R^T L_{21}^T = \begin{bmatrix}0.9239 & 0.3827\\ 0.3827 & -0.9238\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}$$

$$R^T L_{21}^T = \begin{bmatrix}09239 & 0\\ 0.3828 & 0\end{bmatrix}$$

From Eq. (11.10), one has $A_{22}^*=A_{22}-L_{21}D_{11}L_{21}^T$

$$A_{22}^* = \begin{bmatrix}2 & -1\\ -1 & 1\end{bmatrix} - \begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}0 & -1\\ -1 & 2\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}$$

$$A_{22}^* = \begin{bmatrix}2 & -1\\ -1 & 1\end{bmatrix}$$

From Eq. (11.9), A_{22}^* can be factorized (in a row-by-row fashion, in actual computer implementation) as:

$$A_{22}^*=L_{22}D_{22}L_{22}^T$$

$$or \begin{bmatrix}2 & -1\\ -1 & 1\end{bmatrix} = \begin{bmatrix}1 & 0\\ -0.5 & 1\end{bmatrix}\begin{bmatrix}2 & 0\\ 0 & 0.5\end{bmatrix}\begin{bmatrix}1 & -0.5\\ 0 & 1\end{bmatrix}$$

From Eq. (11.17), one has:

$$\begin{bmatrix}0 & -1 & 0 & 0\\ -1 & 2 & -1 & 0\\ 0 & -1 & 2 & -1\\ 0 & 0 & -1 & 1\end{bmatrix} = \begin{bmatrix}0.9239 & 0.3827 & 0 & 0\\ 0.3827 & -0.9239 & 0 & 0\\ 0.9239 & 0.3828 & 1 & 0\\ 0 & 0 & -0.5 & 1\end{bmatrix}\begin{bmatrix}-0.4142 & 0 & 0 & 0\\ 0 & 2.4143 & 0 & 0\\ 0 & 0 & 2 & 0\\ 0 & 0 & 0 & 0.5\end{bmatrix}\begin{bmatrix}0.9239 & 0.3827 & 0.9239 & 0\\ 0.3827 & -0.9239 & 0.3828 & 0\\ 0 & 0 & 1 & -0.5\\ 0 & 0 & 0 & 1\end{bmatrix}$$

In actual computer coding, the rotation matrix R need not be stored in the upper-left

portion of [L], or $[L]^T$. Instead, only the row (or column) location of the "sick" row, and one rotation angle θ need to be stored, since these information will be used later during the Forward and Backward solution phases.

Recalled: [R] ≡ [Φ = Eigen Matrix] = $\begin{bmatrix} 0.9239 & 0.3827 \\ 0.3827 & -0.9239 \end{bmatrix}$

Thus, we only need to store rotation angle θ =22.5° $\begin{pmatrix} \cos 22.5° & = 0.9239 \\ \sin 22.5° & = 0.3827 \end{pmatrix}$ to recover the rotation matrix [R].

Example 11.4:
The system of equations [A] {x} = {b} is given, where:

$$[A] = \begin{bmatrix} 2 & 10 & 0 & 7 \\ 10 & 0 & 1 & 4 \\ 0 & 1 & 1 & 8 \\ 7 & 4 & 8 & 9 \end{bmatrix} \text{ and } \{b\} = \begin{Bmatrix} 19 \\ 15 \\ 10 \\ 28 \end{Bmatrix}$$

It is noted that [A] is an indefinite matrix, since the eigenvalues of [A] have both positive and negative eigen-values (according to MATLAB solutions!)

(a) Question 1: Find the LDL^T factorization of A, by using the algorithm shown in Table 10.1

(b) Question 2: Find the LDL^T factorization of A, by using the algorithm shown in Table 10.1, and by "pretending" row #2 is sick.

Solution for question 1 of this example is described in the following paragraphs (also refer to Table 10.1):

Step 1: Factorize row #1 (temporarily assume row #1 is not changed!)

Step 2: Factorize row #2 (thus, I =2)

K = 1

$$\text{XMULT} = \frac{u_{1,2}}{u_{1,1}} = \frac{10}{2} = 5$$

$$\begin{cases} J = ②,③,④ \\ u_{2,2} = 0 - (5) * (u_{1,2} = 10) = -50 \\ u_{2,3} = 1 - (5) * (u_{1,3} = 0) = 1 \\ u_{2,4} = 4 - (5) * (u_{1,4} = 7) = -31 \\ u_{1,2} = 5 \end{cases}$$

2	10	0	7
	-50	1	-31
		1	8
			9

Step 3: Factorize row #3 (thus, I = 3)

K = ①,②

$$\text{XMULT} = \frac{u_{1,3}}{u_{1,1}} = 0$$

$$\text{XMULT} = \frac{u_{2,3}}{u_{2,2}} = \frac{1}{-50}$$

$$\begin{cases} J = ③,④ \\ u_{3,3} = 1 - (0) * (\times) = 1 \\ u_{3,4} = 8 - (0) * (\times) = 8 \\ u_{1.3} = 0 \end{cases}$$

2	10	0	7
	-50	1	-31
		1	8
			9

$$\begin{cases} u_{3,3} = 1 + \left(\frac{+1}{50}\right) * (u_{2,3} = 1) = \frac{51}{50} = 1.02 \\ u_{3,4} = 8 - \left(\frac{+1}{50}\right) * (u_{2,4} = +31) = 7.38 \end{cases}$$
$$u_{2,3} = -0.02$$

2	10	0	7
	-50	-0.02	-31
		1.02	7.38
			9

Step 4: Factorize row # 4 (thus, I = 4)
K = ①, ②, ③

$$\text{XMULT} = \frac{u_{1,4}}{u_{1,1}} = \frac{7}{2} = 3.5$$

$$\begin{cases} J = ④ \\ u_{4,4} = 9 - (3.5)(u_{1,4} = 7) = -15.5 \\ u_{1,4} = 3.5 \end{cases}$$

2	5	0	3.5
	-50	-0.02	-31
		1.02	7.38
			-15.5

$$\text{XMULT} = \frac{u_{2,4}}{u_{2,2}} = \frac{-31}{-50} = 0.62$$

$$\begin{cases} J = 4 \\ u_{4,4} = -15.5 + (0.62)(u_{2,4} = +31) = 3.72 \\ u_{2,4} = 0.62 \end{cases}$$

2	5	0	3.5
	-50	-0.02	0.62
		1.02	7.38
			3.72

$$\text{XMULT} = \frac{u_{3,4}}{u_{3,3}} = \frac{7.38}{1.02} = 7.24$$

$$\begin{cases} J = ④ \\ u_{4,4} = 3.72 - (7.24)(u_{3,4} = 7.38) = -49.71 \\ u_{3,4} = 7.24 \end{cases}$$

2	5	0	3.5
	-50	-0.02	0.62
		1.02	7.24
			-49.71

Hence:

$$\begin{bmatrix} 2 & 10 & 0 & 7 \\ 10 & 0 & 1 & 4 \\ 0 & 1 & 1 & 8 \\ 7 & 4 & 8 & 9 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 \\ 0 & -0.02 & 1 & 0 \\ 3.5 & 0.62 & 7.24 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & -50 & 0 & 0 \\ 0 & 0 & 1.02 & 0 \\ 0 & 0 & 0 & -49.79 \end{bmatrix} \begin{bmatrix} 1 & 5 & 0 & 3.5 \\ 0 & 1 & -0.02 & 0.62 \\ 0 & 0 & 1 & 7.24 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Solution for question 2 of this example is described in the following paragraphs (also refer to Table 10.1)

Step 1: Factorize row #1 (temporarily assume row #1 is not changed !)
Step 2: Factorize row #2 (thus, I = 2)
K = 1

$$\text{XMULT} = \frac{u(1,2)}{u(1,1)} = \frac{10}{2} = 5$$

$$\begin{cases} J = ②,③,④ \\ u_{2,2} = 0 - (5) * u_{1,2} = -50 \\ u_{2,3} = 1 - (5) * u_{1,3} = 1 \\ u_{2,4} = 4 - (5) * (u_{1,4} = 7) = -31 \\ u_{1,2} = 5 \end{cases}$$

2	10	0	7
	-50	1	-31
		1	8
			9

Step 3: At this stage, we "pretend" row #2 is the sick row!

Step 4: Factorize row #3 (thus, I =3), by considering contribution from row #1 only!
K = 1 (note: K = 2 = "sick" row is <u>not</u> considered)

$$\text{XMULT} = \frac{u_{1,3}}{u_{1,1}} = 0$$

$$\begin{cases} J = ③,④ \\ u_{3,3} = 1 - (0) * (\times) = 1 \\ u_{3,4} = 8 - (0) * (\times) = 8 \\ u_{1,3} = 0 \end{cases}$$

2	10	0	7
	-50	1	-31
		1	8
			9

Step 5: Since the determinant of a 2x2 block is $\begin{vmatrix} -50 & 1 \\ 1 & 1 \end{vmatrix} \neq 0$, hence the 2x2 block is non-singular (and no switching rows/columns required)
Step 6: Perform the 2x2 pivoting with rotation strategies

Since $D_{11} = A_{11} = \begin{bmatrix} -50 & 1 \\ 1 & 1 \end{bmatrix}$. *Therefore,* $[R] = [\Phi] = \begin{bmatrix} 0.9998 & 0.0196 \\ -0.0196 & 0.9998 \end{bmatrix} = [\textit{Eigen matrix of } D_{11}]$

From Eq. (11.15), one has

$$D_{11}^{*} = \Phi^{T} D_{11} \Phi = \begin{bmatrix} -50.0196 & 0 \\ 0 & 1.0196 \end{bmatrix}$$

$$D_{11}^{-1} = \begin{bmatrix} -0.0196 & 0.0196 \\ 0.0196 & 0.9804 \end{bmatrix} \; ; \; A_{12} = \begin{Bmatrix} -31 \\ 8 \end{Bmatrix}$$

From Eq. (11.8), one has $L_{21}^{T} = D_{11}^{-1} * A_{12} = \begin{Bmatrix} 0.7647 \\ 7.2353 \end{Bmatrix}$

Furthermore, $R^{T} A_{21}^{T} = R^{T} A_{12} = \begin{Bmatrix} -31.1508 \\ 7.3910 \end{Bmatrix}$

Thus, currently we have

2	10	0	7
	-50.0196	0	-31.1508
		1.0196	7.3910
			9

Step 7: Factorize row #4 (using contributions from previous rows) according to the

normal procedure.
Using previous rows K = ①, ②, ③

$$\text{MULT} = \frac{u_{1,4}}{u_{1,1}} = \frac{7}{2} = 3.5$$

$$\begin{cases} J=④ \\ u_{4,4}=9-(3.5)(u_{1,4}=7)=-15.5 \\ u_{1,4}=3.5 \end{cases}$$

2	10	0	3.5
	-50.0196	0	-31.1508
		1.0196	7.3910
			-15.5

$$\text{MULT} = \frac{u_{2,4}}{u_{2,2}} = \frac{31.1508}{50.0196} = 0.6228$$

$$\begin{cases} J=④ \\ u_{4,4}=-15.5+(0.6228)(u_{2,4}=+31.1508)=3.90 \\ u_{2,4}=0.6228 \end{cases}$$

2	10	0	3.5
	-50.0196	0	0.6228
		1.0196	7.3910
			3.9

$$\text{MULT} = \frac{u_{3,4}}{u_{3,3}} = \frac{7.3910}{1.0196} = 7.2489$$

$$\begin{cases} J=④ \\ u_{4,4}=3.9-(7.2489)(u_{3,4}=7.3910)=-49.6768 \\ u_{3,4}=7.2489 \end{cases}$$

2	10	0	3.5
	-50.0196	0	0.6228
		1.0196	7.2489
			-49.6768

Hence:

$$\begin{bmatrix} 2 & 10 & 0 & 7 \\ 10 & 0 & 1 & 4 \\ 0 & 1 & 1 & 8 \\ 7 & 4 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 0.9998 & 0.0196 & 0 \\ 0 & -0.0196 & 0.9998 & 0 \\ 3.5 & 0.6728 & 7.2489 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & -50.0196 & 0 & \\ 0 & 0 & 1.01096 & 0 \\ 0 & 0 & 0 & -49.6768 \end{bmatrix} \begin{bmatrix} 1 & 5 & 0 & 3.5 \\ 0 & 0.9998 & -0.0196 & 0.6728 \\ 0 & 0.0196 & 0.9998 & 7.2489 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

11.4 "Natural" 2x2 Pivoting

The submatrix $[A_{11}]$, shown in Eq. (11.3) and in Figure 11.1, has been assumed to be non-singular. In practical computer implementation, the following procedure can be used for determining whether or not submatrix $[A_{11}]$ can pass the "non-singular" test.

To assure that submatrix $[A_{11}]$ is "invertible," in general, we would like to see the determinant of $[A_{11}]$ to be relatively large. In other words, since the diagonal term (of submatrix $[A_{11}]$) on the "sick" row is very small (nearly zero), we prefer to have (please refer to Eq. 11.3):

$$|a_{11}| < \max |a_{1r}|, \; r = 2, 3 \ldots N \qquad \textbf{(11.18)}$$

The inequality shown in Eq. (11.18) is necessary, since if required, we can move $\max|a_{1r}|$ into the location of a_{12} (to achieve a large determinant of submatrix $[A_{11}]$). To be even safer, the inequality Eq. (11.18) can be modified to

$$|a_{11}| < \alpha * \max |a_{1r}| \tag{11.19}$$

The factor α has been suggested in Refs. [11.1, 11.3] as

$$\alpha = (1+\sqrt{17})/8 \tag{11.20}$$

Thus, the diagonal term on the "sick" row of submatrix $[A_{11}]$ will be much smaller than its off-diagonal term.

To make the discussion more general, the "sick" row may occur at any row (such as the k^{th} row) instead of assuming to occur in row 1. Thus, Eq. (11.19) can be generalized to:

$$|a_{kk}| < \alpha * \max |a_{kr}|, \tag{11.21}$$

Equation (11.21), however, can only be applied for a "single" row. In order to make it applicable also to a "2x2 block" row, the "concept" can be generalized to:

$$|[A_{11}]| < \alpha * \max \begin{Bmatrix} a_{kr} \\ a_{k+1,r} \end{Bmatrix} \tag{11.22}$$

The left-hand-side of Eq. (11.22) "looks" like a 2x2 matrix, but the right-hand-side "looks" like a vector. Thus, for dimensional "compatibility," Eq. (11.22) should be modified as

$$|[A_{11}]| * \begin{Bmatrix} 1 \\ 1 \end{Bmatrix} < \alpha * \max \begin{Bmatrix} a_{kr} \\ a_{k+1,r} \end{Bmatrix} \tag{11.23}$$

$$\text{or} \quad \begin{Bmatrix} 1 \\ 1 \end{Bmatrix} < \alpha * \left|[A_{11}]^{-1}\right| * \begin{Bmatrix} \gamma \\ \mu \end{Bmatrix} \tag{11.24}$$

where $\gamma \equiv \max |a_{kr}|$

$\mu \equiv \max |a_{k+1,r}|$ (11.25)

From Eq. (11.3), one has

$$[A_{11}] = \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} \tag{11.26}$$

Hence:

$$\left|[A_{11}]^{-1}\right| = \left(\frac{1}{D}\right) * \begin{bmatrix} |a_{22}| & |-a_{12}| \\ |-a_{12}| & |a_{11}| \end{bmatrix} \tag{11.27}$$

The scalar D, shown in Eq. (11.27), is the determinant of submatrix $[A_{11}]$.
Eq. (11.27) can be symbolically represented as

$$\left|[A_{11}]^{-1}\right| = \begin{bmatrix} |b_{11}| & |b_{12}| \\ |b_{12}| & |b_{22}| \end{bmatrix} \tag{11.28}$$

where

$$|b_{11}| \equiv \frac{|a_{22}|}{D} \tag{11.29}$$

$$|b_{22}| \equiv \frac{|a_{11}|}{D} \tag{11.30}$$

$$|b_{12}| \equiv \frac{|-a_{12}|}{D} \tag{11.31}$$

Substituting Eq. (11.28) into Eq. (11.24), one obtains

$$\begin{Bmatrix}1\\1\end{Bmatrix} < \alpha * \begin{bmatrix}|b_{11}| & |b_{12}|\\ |b_{12}| & |b_{22}|\end{bmatrix} * \begin{Bmatrix}\gamma\\ \mu\end{Bmatrix} \tag{11.32}$$

$$\begin{Bmatrix}1\\1\end{Bmatrix} < \alpha * \begin{bmatrix}|b_{11}|\gamma + & |b_{12}|\mu\\ |b_{12}|\gamma + & |b_{22}|\mu\end{bmatrix} \tag{11.33}$$

or $1<\alpha\,(|b_{11}|\gamma + |b_{12}|\mu)$ **(11.34)**

$1<\alpha\,(|b_{12}|\gamma + |b_{22}|\mu)$ **(11.35)**

11.5 Switching Row(s) and Column(s) During Factorization

Up to this point, it has been assumed that one only has a single "sick" row. In other words, the 2x2 block rows (which consists of a sick row, and its neighboring row) is non-singular. If this is not the case, then row(s) and column(s) switching are necessary for maintaining the numerical stabilities. In this work, the criteria used to determine whether a row (or a block 2x2 rows) is sick are similar (but not exactly the same) to the ones used in Ref. [10.27], and are illustrated in Figure 11.2 and Fig. 11.3.

Remarks:

1. Point F (on the "sick" row) is assumed to have the maximum value.
2. Point G (on the column which passes through point F) is assumed to have the maximum value on column FLT, or (due to symmetry) on column FLJ
3. The good region #1, bounded by ABCDEFQO (except the "sick" row) is assumed to have been factorized in a "normal" row-by-row fashion (no switching rows and columns are required in this region). It is also assumed that the factorized region BCDE is zero. Thus, triangular region HIN will "not" be effected by those factorized rows in the good region #1.
4. Since the row(s) switching only occurs between the "sick" row, and the r^{th} row, the arrays IA(-) and JA(-) correspond to the triangular region LJN will not be effected. It is noted here that arrays IA and JA have the same -definitions as arrays ISTARTROW and ICOLNUM in Eqs. 10.16, and 10.17, respectively.
5. After switching row(s) and column(s), the array IA(-) and JA(-) need to be constructed, which correspond to the "new" sub-matrix ODNU (thus, the "sick" row will always appear as the 1^{st} row of the "new" coefficient matrix!).

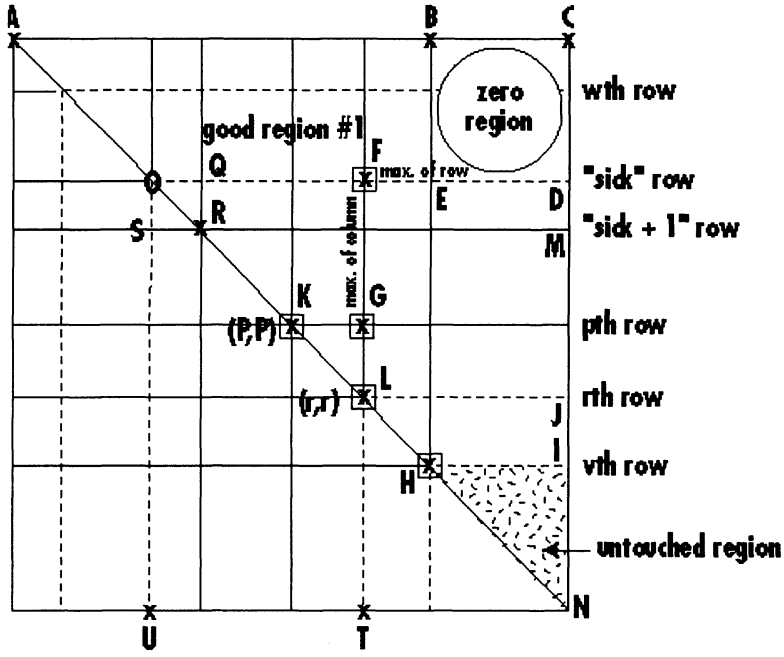


Figure 11.2 Switching row(s) and column(s) during factorization (row #r is inside the partially factorized region OQDIH) .

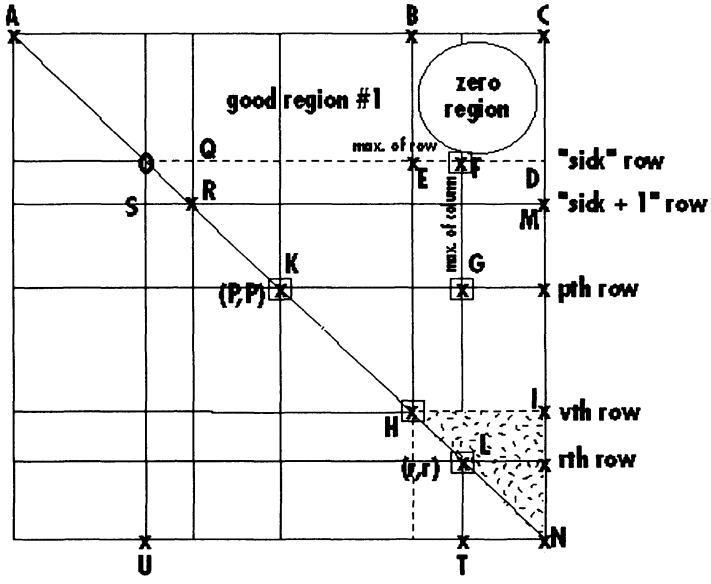


Figure 11.3 Switching row(s) and column(s) during factorization (row # r is outside the partially factorized region OQDIH)

Case 1: Only need to switch 1 row and 1 column

This is the case where point Q (on the "sick" row) has a (small) nonzero value (see Figure 11.2). Switching row (and column) may not be required if point Q already has a relatively large value. When a row (and a column) switching is required, row (and column) number (sick +1) will be switched with row (and column) number (r). As a consequence, the maximum value of the "sick" row (such as point F) will be moved to the desirable location (such as point Q).

Case 2: Need to switch 2 rows and columns

This is the case where point Q (on the "sick" row) has a near zero value. This option is needed if case 1 has failed (for example, the maximum value on the "sick" row, point F in Fig. 11.2 or in Fig. 11.3, is too small). In this case, one will switch a row (and column) number (sick) with row (and column) number (p). Then, row (and column) number (sick +1) will be switched with row (and column) number (r). The objective here is to move the maximum value (see point G) of column FGLT (or column FGLJ, due to symmetry) to the location Q (on the sick row). It is also preferred to have small, or near zero values for the diagonal of row # (sick) and row # (sick +1). Thus, compromised strategies can be enforced to ensure the 2x2 block (= row # "sick," and row # "sick+1", after rows/columns switching occur) will have the following form (for numerical stability purpose), as shown in Figure 11.4.

b	a	⇨ "sick" row
a	b	⇨ "sick +1" row
a≡ prefer to be a large number b≡prefer to be a near zero (small) number		

Figure 11.4 A preferable 2x2 block matrix after rows and columns switching

The eigenvalues of the 2x2 block matrix, shown in Figure 11.4, can be computed as

$$(b-\lambda)^2-a^2=0 \qquad \textbf{(11.36)}$$

If b is a small (say, near zero) number, then the above equation can be approximated as

$$\lambda^2 \approx a^2 \qquad \textbf{(11.37)}$$

or $\lambda_1 \approx +a$, and $\lambda_2 = -a$ **(11.38)**

The 2 eigenvalues of the 2x2 block matrix (see Fig. 11.4), therefore , are preferred to have nearly same magnitude, with opposite signs, for numerical stability purpose. It is also noticed that the determinant of the 2x2 block is not zero (or not nearly zero).

It should also be mentioned here that compromised strategies (trade off between computational time, and solution accuracy) can be used by specifying the constraint that the switched rows are preferable to be close to each other.

Case 3: Switching 1 or 2 rows/columns (#sick, and/or # sick +1) with 1 or 2 rows/columns (# j, and/or # k) which are "closest" to row numbers "sick" and

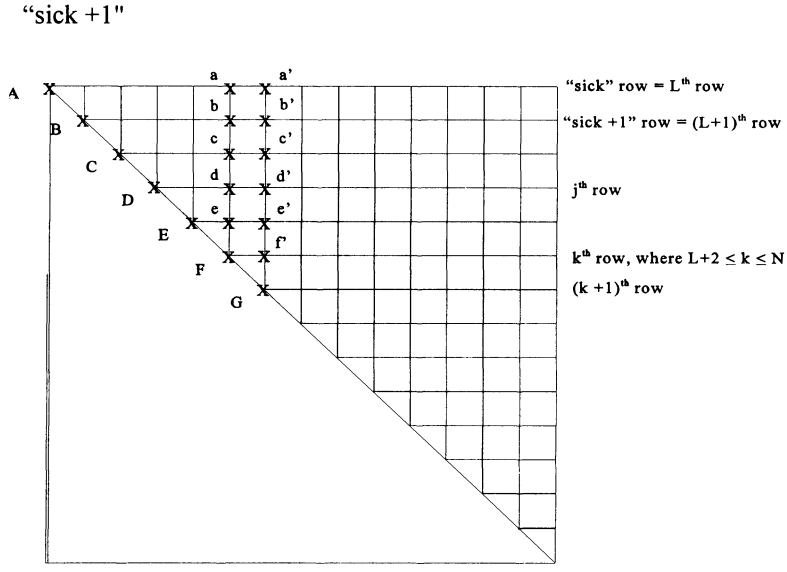


Figure 11.5 Switching rows / columns which are closest to each other

Considering the k^{th} row, shown in Figure 11.5, a vertical line is then drawn from point F. An "If" check is used to see if the 2x2 block matrix FaA is "invertible," if not, then the 2x2 block matrix FbB, or block matrix FcC, FdD, FeE etc... are checked.

Assuming the 2x2 block matrix FdD is invertible, then rows # j and k will be switched with rows # L and # L+1, respectively.

On the other hand, assuming the 2x2 block FaA is invertible, then in this case, one only needs to switch row/column # k with row/column # (L+1). However, if the 2x2 block FbB is invertible, then in this case, one only needs to switch row/column # k with row/column # L.

If all 2x2 submatrices FaA, FbB,..., FeE (associated with k^{th} row) are NOT invertible, then we'll have to consider the next row (i.e., # row k+1), and the same process is repeated (i.e., to check and see if there is any 2x2 block Ga'A, or Gb'B, or ...Gf'F is invertible...)

Since the distance between rows # j and/or # k are closest possible to rows # L and/or # L+1,
one may expect minimum fill-in term will be created due to these row(s)/column(s) switching.

Another strategy has also been considered in our work, which is essentially based upon the super node (or master node) idea, presented earlier in Section 10.8.1. The "sick" row/column (and/or its next row/column) will be switched with the r^{th} row/column (and/or the p^{th} row/column), where we prefer to see the swapped rows/columns to have similar non-zero patterns.

Assuming the "sick" row/column will be switched with the r^{th} row/column, and these 2 rows/columns have similar (say 90% or more) non-zero patterns. If this is the case, then we can expect the extra "fills" created during the rows/columns switching will

be minimized.

11.6 Simultaneously Performing Symbolic and Numerical Factorization

For symmetrical and positive definite system of equations, it is relatively simple to predict the exact fill-in terms, before actually performing the numerical factorization. In this case, one will perform the symbolic factorization phase for the entire matrix, and then perform the numerical factorization phase for the whole matrix. For indefinite system of equations, however, it is more feasible to adopt the strategy of simultaneously performing the symbolic and numerical factorization in a row-by-row fashion. The main difficulties are due to the fact that row(s) and column(s) switching may be required at any stages during the numerical factorization process for indefinite system of equations. The direct consequence is that the fill-in patterns will be changed whenever row(s) and column(s) switching occur.

11.7 Restart Memory Managements

The focus of this section is to discuss the memory management schemes, in conjunction with the developed 2x2 pivoting strategies for sparse, indefinite system of equations. Figures 11.6(a) and 11.6(b) show the memory management schemes, which correspond to Figures 11.2 and 11.3, respectively

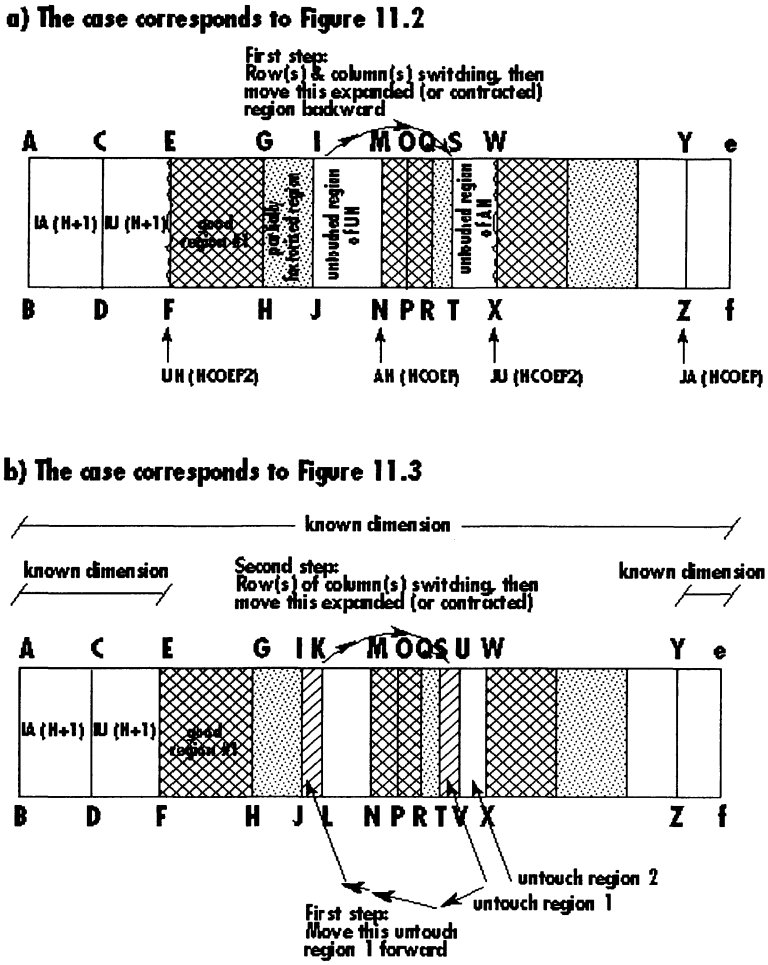


Figure 11.6 Memory management movements

Some Remarks About Figures 11.6(a) and 11.6(b):

[1] We assume the total dimension (or total incore memory) available for equation solution is known.

[2] The known, fixed dimensions for integer arrays IA, and IU are placed at the beginning (see region AEFB) of the total vector (see region AefB). The remaining memories (see region EFef) will be divided into 2 segments (see segment EFWX, and segment WXef), where the first segment (=EFWX) is twice as big as the second segment (=WXef). The reason is because arrays UN (NCOEF2) and AN (NCOEF) in the first segment are "real" arrays, while arrays JU (NCOEF2) and JA (NCOEF) in the second segment are "integer" arrays.

[3] Arrays AN(NCOEF), and JA (NCOEF) are placed at the end of 1st segment (=segment EFWX), and at the end of 2nd segment (=segment WXef), respectively. The dimensions for these 2 arrays are known. In Figure 11.6 (a)

and Fig. 11.6(b), the beginning of arrays UN, AN, JU and JA are indicated by the lines EF, MN, WX and YZ, respectively.

[4] In Figure 11.6(b), which corresponds to Figure 11.3, the arrays UN (NCOEF2), AN (NCOEF), JU(NCOEF2) and JA(NCOEF) each consists of 3 regions:

(a) Good region (see region EGHF of Fig. 11.6(b), or region ABCDFQO of Fig 11.3) consists of rows#1 through # (sick-1), which can be factorized in a normal row-by-row fashion.

(b) Partially factorized region (see region GIJH of Fig. 11.6(b), or region OQFDIH of Fig. 11.3) consists of rows # (sick+1) through #(v). These rows have been "partially" factorized by (appropriated) previous rows #1 through # (sick-1)

(c) Untouched region (see region SWXT, or region IKLJ in Fig. 11.6(b), or region HIN in Fig. 11.3) consists of rows #(v+1) through #(N). These rows will not be influenced by the factorized rows in the good region (see Fig. 11.3). The untouched region can be further partitioned into 2 sub-regions, such as shown in sub-region STVU (see Fig. 11.6(b), or sub-region HIJL (see Fig. 11.3), and subregion UVXW (see Fig. 11.6(b), or sub-region LJN (see Fig. 11.3)

[5] Row(s) and column(s) switching may occur anywhere between row # (sick) and row # (r). Due to these rows/columns switching, the region GIKLJH (see Fig. 11.6(b)) may be either enlarged, or shrinked (since the fill-in patterns can be changed for better or worse!), and this region (=GHLK, see Fig. 11.6(b)) will be moved backward to the region OPVU (see Fig. 11.6(b). The edge KL (of region GHLK) should be coincided with the edge UV (of region OPVU)

[6] The edge OP (of Fig. 11.6(b)) will be the starting location for the new, reduced matrix ODNU (shown in Fig. 11.3)

[7] The memory management schemes presented in Fig. 11.6(a) (which corresponds to Fig. 11.2, where r^{th} row is inside the partially factorized region OQDIH) is quite similar (and more simpler) as compared to Fig. 11.6(b). The key difference between Fig. 11.6(a) and Fig. 11.6(b) is from the fact that since row(s)/column(s) switching only occur between row #(sick), and row # (r), therefore, the entire untouched region HIN (see Fig. 11.2) will not be divided into 2 subregions, as mentioned earlier in remark #4(c).

[8] As the sparse numerical factorization process continues to progress, the "good region # 1" (see Fig. 11.6(a), or Fig. 11.6(b) will continue to grow, while the region OWXP of array AN (see Fig. 11.6(a), or Fig. 11.6(b), or region ODNU of Figs. 11.2 and 11.3 will continue to shrink.

[9] As the region OWXP (which is part of array AN) continues to shrink (see Fig. 11.6(a), or Fig. 11.6(b)). The region MOPN (which is also part of array AN) will continue to grow. Furthermore, this region (=MOPN) can be used for the expanded array UN. Thus, the unused and growing region MOPN of array AN can be re-used (by the expanded array UN) to save the computer memory.

11.8 Major Step-by-Step Procedures for Mixed Look Forward/Look Backward, Sparse LDL^T Factorization, Forward and Backward Solution with 2x2 Pivoting Strategies

The row-by-row sparse LDL^T factorization can be accomplished based upon either look forward, or look backward strategies. In using "Look Forward Factorization" strategies,

upon completing the currently factorized row, such as the w^{th} row (shown in Fig. 11.2), the factorized w^{th} row will be used to partially update (or factorize) all appropriately remaining rows. Then, the w^{th} row (and all its previous rows) will never be used (or referred to) again! In using "Look Backward Factorization" strategies, the currently being factorized row, such as the w^{th} row (shown in Fig. 11.2), will have to refer to its previously factorized rows. Furthermore, subsequent factorized rows (after w^{th} row) may still have to refer to the w^{th} row again.

Major step-by-step procedures can be summarized in the following paragraphs:

Step 1: From row #1 until the "sick" row, row-by-row "look backward factorization" strategies can be used. Assuming the "sick row" occurs at row I=10 (refer to Table 10.1). Thus, upon completely factorized by its appropriately previous rows, the sick row #10 has $U_{10,10} = 0$. Hence, factorizing the next row (say row I=11, see Table 10.1) will have the problem of dividing by zero (or $U_{10,10}$), unless appropriated pivoting strategies are used.

Step 2: Check to see if 2x2 blocks (consists of "sick" row and "sick+1" row) is non-singular (and stable) or not. If necessary, check to see if 1x1, or 2x2 pivoting strategies are required (see Sections 11.1-11.4)

Step 3: Using the completely factorized rows [from row #1 until $(sick-1)^{th}$ row] and the "look forward factorization" strategies to partially factorize all remaining rows [such as from row #(sick+1) until the last row #N]

Step 4: Performing the appropriated row(s) and column(s) switching operations, and memory management operations (as shown in Figures 11.6(a) & 11.6(b))

Step 5: "New" definitions for arrays IA_{new}, JA_{new}, AN_{new}, etc... are defined for the "new" (& reduced) stiffness matrix ODNU (see Fig. 11.2, or Fig. 11.3). In this "new" stiffness matrix, the "sick" row maybe placed at either the first row, or at some intermediate row (if the sick row has been switched with another row).

Step 6: Repeat the procedures (until all rows of the original stiffness matrix have been completely factorize) by returning back to step 1.

Step 7: Forward and backward substitution phases can be done in a similar fashion as explained in Chapter 10, with 2 special attentions:

(a) When row/column switching occurs during factorization phase, the corresponding row switching need be done also for the RHS (Right-Hand-Side) vector

(b) Operations involved the rotation matrix [R] during factorization need also be done for the RHS vector.

11.9 Numerical Evaluations

In order to evaluate the performance (in terms of computational time, solution accuracy and memory requirements) of the proposed sparse solver with pivoting strategies for symmetrical, indefinite system of equations, 5 examples (ranging from 51 to 15,367 unknown degree-of-freedoms) are used in this study.

The numerical results are presented in Tables 11.1 and 11.2. The improved performance can be achieved by applying the MMD re-ordering algorithm (to minimize the fill-in terms) and by moving all zero diagonal terms (of the original stiffness matrix) toward the bottom right of the original stiffness matrix.

In Table 11.1, both Cray-YMP (single processor) computer and the IBM-R6000/590 workstation are used in this study, and are shown in column 1. The total number of equation (NEQ, or the total number of degree-of-freedom), and the total number of nonzero coefficients (NCOEF) before factorization are shown in column 2.

For all 5 structural examples considered in this section, the resulting linear system of indefinite equations, shown in Eq. 10.1 can be expressed in the following form:

$$\begin{bmatrix} A & B \\ B^T & 0 \end{bmatrix} \begin{Bmatrix} x \\ \lambda \end{Bmatrix} = \begin{Bmatrix} b \\ c \end{Bmatrix} \tag{11.39}$$

In Eq. (11.39), the vector $\vec{x}$ can be referred to as the "displacement" vector, where as the vector $\vec{\lambda}$ (which corresponds to the zero diagonal terms of the coefficient stiffness matrix) can be referred to as the "Lagrange multiplier" vector. The bottom right submatrix of the coefficient stiffness matrix (shown in Eq. 11.39) is a "zero" submatrix.

The 3rd column of Table 11.1 represents the summation of the (absolute) unknown displacement vector $\vec{x}$. The maximum (absolute) displacement component is printed in column #4. The relative "displacement and Lagrange multiplier" error norm can be calculated as (refer to Eq. 10.55, or Eq. 11.39)

$$R.E.N. = \frac{\|KZ-f\|}{\|f\|} \tag{11.40}$$

and is shown in column #6 (of Table 11.1). The 3 controlled parameters α, β (or factored parameter) and γ (or tuning parameter) are presented in column #7. Finally, memory storage requirements are given in column #8. Impacts of the input controlled parameters α, β and γ on the algorithm performance are shown in Table 11.2 (Ncoef2 represents total number of nonzero coefficients after factorization).

Table 11.1 Comparison of different indefinite sparse solvers

	NEQ NCOEF	SUM (DISPL)	MAX DISPLACEMENT	CPU TIME (SECONDS)	RELATIVE ERROR (DISP + λ)	CONTROL PARAMETER α - Factor - Twin
BOEING		2.265 $*10^{-2}$	1.999 $* 10^{-3}$	0.041	7.0 $* 10^{-14}$	
CRAY	51	2.26499 $*10^{-2}$	1.999 $* 10^{-3}$	0.0036	1.152 $* 10^{-13}$	0.1-1-1
STRETCH	218	2.2649 $*10^{-2}$	1.999 $* 10^{-3}$	0.0	1.8 $* 10^{-15}$	0.1-1-1
BOEING		3.1596564	0.15253658	0.245	4.034 $* 10^{-10}$	
CRAY	247	3.1596564	0.15253658	0.021	1.06757 $* 10^{-9}$	0.1-1-1
STRETCH	2009	3.1596564	0.15253658	0.0099≈0.01	7.06289 $* 10^{-12}$	0.1-1-1
BOEING		29.68462	0.20289318	2.351	3.26 $* 10^{-10}$	
CRAY	1440	29.68462	0.20289318	0.571	1.184 $* 10^{-9}$	0.1-1-1
STRETCH	22137	29.68462	0.20289318	0.299	6.728 $* 10^{-12}$	0.1-1-1
BOEING		34.7033	9.31212 $* 10^{-2}$	7.736	9.97 $* 10^{-11}$	
CRAY	2430	34.672262	9.3111810 $* 10^{-2}$	6.136	1.2695 $* 10^{-11}$	0.1-1-1
STRETCH	75206	34.700716	9.312068 $* 10^{-2}$	8.389	4.4253 $* 10^{-13}$	
BOEING		512.35	0.205696	35.77	4.384 $* 10^{-11}$	
CRAY	15367	N/A	N/A	36.62	2.73 $*10^{-9}$	0.1-1-1
STRETCH	286044	512.35488	0.2056969	≈76 sec	1.70566 $* 10^{-13}$	0.1-1-1

Table 11.2 Parameters study for an indefinite sparse solver

CONTROL PARAMETERS	NEQ NCOEF	SUM (DISPL)	MAX DISPL	CPU SECONDS	RELATIVE ERROR (DISP + λ)	NUMBER OF 2X2 PIVOTING	NUMBER OF DIAG. INTER-CHANGE	Ncoef2
0.1-1.-1.	51 218	2.2649 * 10^{-2}	1.999 * 10^{-3}	0.0	1.8 * 10^{-15}			
0.1-1.-1.	247 2009	3.159564	0.1525 3658	0.0099	7.06289 * 10^{-12}			
0.1-.1-.1	1440 22157	29.68462	0.20289318	0.299	6.728 * 10^{-12}			
0.1-.1-.1 0.01-.1-.1 0.	2430 75206	34.700716 34.698586	9.312068 * 10^{-2} 9.312042 * 10^{-2}	8.389 7.1599	4.4253 * 10^{-3} 8.7521 * 10^{-12}	32	134	540109
0.01-1-1 0.005-.075-.1 0.1-1-1	15367 286044	512.35488 512.35488 512.35488	0.2056969 0.2056969 0.2056969	76 sec	9.922983 * 10^{-12} 1.1545 * 10^{-11} 1.70566 * 10^{-13}	45 37 143	446 432 623	6221192 6194548 6452162

11.10 Some Remarks on Unsymmetrical-Sparse System of Linear Equations

For several important engineering and science applications, such as thermal-structural analysis, linear programming, computational fluid dynamics etc..., unsymmetrical-sparse system of linear equations can arise very naturally. Parallel-vector algorithms for full, banded and variable bandwidths (for unsymmetrical system of equations) have been discussed with great details in Chapter 8. Also, algorithms for sparse, symmetrical

system of equations (with and without pivoting strategies) have been discussed with great details in Chapter 10, (without pivoting), and Sections 11.1 through 11.7 (with pivoting strategies). The sparse technologies discussed in Chapter 10 can be directly used in conjunction with unsymmetrical-banded algorithms discussed in Chapter 8 for designing efficient unsymmetrical-sparse algorithms. The resulting unsymmetrical-sparse algorithms will have the following key components:

(a) The sparse, upper triangular portion of a given unsymmetrical coefficient, matrix will be factorized essentially in the same row-by-row fashions as described in Chapter 10.

(b) The sparse lower triangular portion of a given unsymmetrical coefficient matrix will be factorized in a column-by-column fashion [as the image, with respect to the main diagonal, of step (a) above]

(c) It is assumed that the given coefficient matrix is unsymmetrical with respect to the numerical values, but is still symmetrical with respect to the non-zero locations (see Figures 11.7 - 11.8)

$$A = \begin{bmatrix} a & o & b & o & o \\ o & d & e & o & f \\ c & g & i & j & o \\ o & o & k & l & o \\ o & h & o & o & m \end{bmatrix}$$

Figure 11.7 Matrix [A] is symmetrical in locations, but unsymmetrical in numerical values

$$B = \begin{bmatrix} a & o & \bullet & b & o \\ o & d & e & o & f \\ c & g & i & j & o \\ o & o & k & l & o \\ o & h & o & o & m \end{bmatrix}$$

Figure 11.8 Matrix [B] is unsymmetrical in both locations and numerical values

(d) The restrictions stated in (c) can be easily removed by sacrificing some additional computer memory. For example matrix [B] in Fig. 11.8 can still be considered as symmetrical in locations, if we consider the term $B_{1,3} = \bullet$, and $B_{4,1} = 0$ as a non-zero term with its numerical value to be equal to zero!!

(e) The MMD reordering and the symbolic algorithms (discussed in Chapter 10) can still be applied to the unsymmetrical (with respect to numerical values only!) matrix [A] of the type as shown in Fig. 11.7. However, upon exiting the MMD reordering algorithm, special attention need be focused on the movements (utilizing the integer array PERM obtained upon exiting from the MMD algorithm) of non-zero terms (of the original unsymmetrical matrix [A]) to different locations.

(f) In item (e), the integer array PERM represents the "mapping" between the "old" numbering system (before applying MMD algorithm) and the "new" numbering system (after exiting MMD algorithm). For example: PERM (i) = j means the "old" i^{th} row/column of [A] becomes the "new" j^{th} row/column.

The performance of the unsymmetrical sparse solvers (based on the key ideas

presented in this section) is shown in Table 11.3, for the aircraft HSCT finite element model.

It should be emphasized here that the HSCT finite element model will result in system of "symmetrical" simultaneous equations where both unsymmetrical (see Table 11.3) and symmetrical (see Table 11.4) sparse equation solvers are used.

As can be expected, the computer core memory and the numerical factorization time required by the unsymmetrical sparse solver (refer to Table 11.3) are both higher than the ones required by the symmetrical sparse solver (refer to Table 11.4).

Table 11.3 Performance of "unsymmetrical" sparse solver for HSCT aircraft model on IBM- R6000/590 workstation

Problem Characteristics			
Number of Equations	=>	NEQ	= 16146
Non-Zero before fill in	=>	NCOEF	= 999010
Non-Zero after fill in	=>	NCOEF2	= 6034566
Loop Unrolling Level	=>	LOOP	= 8
Memory			
Total Integer memory	= 3613667		
Total real memory	= 7114306		
Total memory	= 10727973		
Error Norm Check			
MAX ABS DISPL AT DOF 522	= 0.447440400042149411		
SUMMATION OF ABS DISPLACEMENTS	= 301.291343623234013		
THE ABSOLUTE ERROR IS || Ax-b ||	= 0.19243162876536217 5E-06		
THE RELATIVE ERROR IS || AX-b || / ||b |	= 0.136069709614759900E-08		
Timing			
-TIME READ fort. FILES	= 0.00000000000000000E+00		
-TIME SYMFACT	= 0.479999989271163940		
-TIME TRANSA	= 2.03999995440244675		
-TIME SUPNODE before N	= 0.179999995976686478		
-TIME NUMFA	= 28.7299993578344584		
-TIME FBE	= 0.319999992847442627		
-TIME SUPNODE After N	= 0.169999996200203896		
-TIME MULTSPA	= 0.599999986588954926E-01		
-TIME ERROR NORM	= 0.00000000000000000E+00		
-TIME MMD REORDERING	= 0.1281261444E-01		

Table 11.4 Performance of "symmetrical" sparse solver for HSCT aircraft model on IBM-R6000/590 workstation

Problem Characteristics		
Number of Equations	=> NEQ	= 16146
Non-Zero before fill in	=> NCOEF	= 499505
Non-Zero after fill in	=> NCOEF2	= 3017283
Loop Unrolling Level	=> LOOP	= 8
MEMORY		
Total Integer Memory	= 3613667	
Total real memory	= 3581372	
Total memory	= 7195039	
ERROR NORM CHECK		
ABS DISPL AT DOF 522	= 0.447440400042149411	
SUMMATION OF ABS DISPLACEMENTS	= 301.291343623234013	
THE ABSOLUTE ERROR IS || Ax-b ||	= 0.1924316228765362175E-06	
THE RELATIVE ERROR IS || AX-b || / || b ||	= 0.136069709614759900E-08	
TIMING		
-TIME READ fort. FILES	= 0.00000000000000000E+00	
-TIME SYMFACT	= 0.479999989271163940	
-TIME TRANSA	= 2.06999995373189449	
-TIME SUPNODE Before N	= 0.169999996200203896	
-TIME NUMFA	= 16.8799996227025986	
-TIME FBE	= 0.309999993070960045	
-TIME SUPNODE After N	= 0.169999996200203896	
-TIME MULTSPA	= 0.399999991059303284E-01	
-TIME ERROR NORM	= 0.00000000000000000E+00	
-TIME MMD REORDERING	= 0.2609395981E-01	

For the complete listing of the FORTRAN source codes, instructions on how to incorporate this equation solver package into any existing application software (on any specific computer platform), and/or the complete consulting service in conjunction with this equation solver etc..., the readers should contact:

Prof. Duc T. Nguyen
Director, Multidisciplinary Parallel-Vector Computation Center
Civil & Environmental Engineering Dept.
Old Dominion University

Room 135, Kaufman Building
Norfolk, VA 23529 (USA)

Tel= (757) 683-3761, Fax = (757) 683-5354
Email= dnguyen@odu.edu

11.11 Summary

An alternative formulation and new computational strategies have been developed for solving general system of sparse-symmetrical- indefinite, and sparse-unsymmetrical equations. Rotational matrix has been used to uncouple the 2x2 block diagonal matrix, and therefore, greatly enhance the FORTRAN computer coding implementation. Mixed "look backward factorization" and "look forward factorization" strategies have also been employed. The computational efficiency, and the solution accuracy have been validated by solving 5 indefinite system of equations (ranging from 51 to 15,367 unknown degree-of-freedom). Further numerical performance improvements have been realized by using MMD reordering algorithm (to minimize the number of fill-in) and by pushing all zero diagonal terms of the original stiffness matrix toward the bottom right of the coefficient matrix.

11.12 Exercises

11.1 Given the following system of equations [A] {x} = {b}, where:

$$A = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 3 & -1 & 0 \\ 0 & -1 & 3 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \text{ and } \{b\} = \begin{Bmatrix} -1 \\ +1 \\ +1 \\ +1 \end{Bmatrix}$$

(a) Following the procedure explained in Example 11.1, find the LDL^T factorization of [A]

(b) Find the forward and backward solution phases

11.2 Resolve the previous problem, and by using the rotation matrix [R] (see Example 11.2)

11.3 Resolve the problem 11.1, by using the rotation matrix [R] and following the procedure explained in Example 11.3

11.13 References

11.1 M.A. Aminpour, J.B. Ransom, and S.L. McCleary, "A Coupled Analysis Method for Structures With Independently Modelled Finite Element Subdomains," IJNM in Engr., Vol. 38, pp. 3695-3718 (1995).

11.2 J.B. Ransom, S.L. McCleary, and M.A. Aminpour, "A New Interface Element for Connecting Independently Modeled Substructures," AIAA/ASME/ASCE/AHS SDM Conference Proceedings, AIAA-93-1503-CP. (1993).

11.3 J.M Housner, M.A. Aminpour, and S.L. McCleary, "Some Recent Developments In Computational Structural Mechanics," Proc. Int. Conf. Computational Engineering Science, ICES Publications, Atlanta, GA, pp. 376-381 (1991).

11.4 I.S. Duff, and J.K. Reid, "MA47: A Fortran Code for Direct Solution of Indefinite Sparse Symmetric

Linear Systems," RAL Report #95-001 (Jan. 1995).
11.5 Z. Johan, T.J.R. Hughes, K.K. Mathur and S.L. Johnson, "A Data Parallel Finite Element Method For Computational Fluid Dynamics On the Connection Machine System," Comput. Methods Appl. Mech. Engr. 99, 113-134 (1992).
11.6 J.R. Bunch, and K. Kaufman, "Some Stable Methods for Calculating The Inertia and Solving Symmetric Linear System," Math. Comp., 31, pp. 162-179 (1977).
11.7 I.S. Duff, A.M. Erisman, and J.K. Reid, Direct Methods for Sparse Matrices, Monographs On Numerical Analysis, Oxford Science Publications (1989).
11.8 J.A. George, and W.H. Liu, Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Englewood Cliffs, N.J. (1981).
11.9 G.H. Golub, and Charles F. Van Loan, Matrix Computations, 2nd Edition, The Johns Hopkins University Press (1989).
11.10 K.H. Law and D.R. Mackay, "A Parallel Row-Oriented Sparse Solution Method for Finite Element Structural Analysis," IJNM in Engr., Vol. 36, 2895-2919 (1993).
11.11 Esmond G. Ng, and B.W. Peyton, "Block Sparse Choleski Algorithm On Advanced Uniprocessor Computer," SIAM J. of Sci. Comput., Volume 14, pp. 1034-1056 (1993).
11.12 D.T. Nguyen, J. Qin, T.Y.P. Chang, and P. Tong, "Efficient Sparse Equation Solver with Unrolling Strategies for Computational Mechanics," CEE Report #96-001, Old Dominion University, Norfolk, VA (1996).
11.13 H. Simon, P. Vu, and C. Yang, "Performance of a Supermodal General Sparse Solver on The Cray-YMP: 1.68 GFLOPS with Autotasking," Applied Mathematics Technical Report, Boeing Computer Services, SCA-TR-117 (March 1989).
11.14 T. Belytscho, E.J. Plaskacz, J.M. Kennedy and D.M. Greenwell, "Finite Element Analysis on the Connection Machine," Computer Methods Appl. Mech. Engr. 81, 27-55 (1990).
11.15 A.K. Noor, "Parallel Processing In Finite Element Structural Analysis*, in Parallel Computations and Their Impact on Mechanics, ASME, pp. 253-277, A.K. Noor (Ed.), (1987).
11.16 A.I. Khan and B.H. V. Topping, "A Transputer Routing Algorithm for Nonlinear or Dynamic Finite Element Analysis," Engineering Computations, Vol. 11, pp. 549-564 (1994).
11.17 D.T. Nguyen, O.O. Storaasli, E.A. Carmona, M. Al-Nasra, Y. Zhang, M.A. Baddourah and T.K. Agarwal, "Parallel-Vector Computation for Linear Structural Analysis and Nonlinear Unconstrained Optimization Problems," Computing Systems in Engineering, An Inter. Journal, Vol. 2, No. 2/3, pp. 175-182 (Sept. 1991).
11.18 K.N. Chiang and R.E. Fulton, "Structural Dynamic Methods for Concurrent Processing Computer," Computers and Structures, 36(6), 1031-1037 (1990).
11.19 D. Zhang and T.Y.P. Chang, "Parallel Cholesky Method on MIMD with Shared Memory," Computers and Structures, Vol. 56, No. 1, pp. 25-38 (1995).

Index