NASA Technical Memorandum 84228

On the Inversion of Block-Tridiagonals Without Storage Constraints

Marshal L. Merriam

March 1982

National Aeronautics and
Space Administration

NASA Technical Memorandum 84228

On the Inversion of Block-Tridiagonals Without Storage Constraints

Marshal L. Merriam, Ames Research Center, Moffett Field, California

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

Abstract. In many programs solving difference equations, problem size is restricted by the number of available memory cells. A strategy has been developed to permit trade-offs between the number of floating point operations required and the storage requirements for the solution of certain problems, such as block tridiagonal systems of equations. This is done by recomputing some intermediate results instead of storing them. Reducing the storage to the square root of the current requirement will roughly double the number of computations. Reducing the storage more than this tends to make the number of computations prohibitively large. In theory though, if m is the order of each sub-matrix in the block tridiagonal matrix, one can solve any linear system with only $5m^2 + 1$ temporary storage cells. In many cases m is a constant and quite small. For example, in solving a factored form of the three-dimensional Navier-Stokes equations, the size m of the block tridiagonals is 5. In fact, for block tridiagonals arising from finite difference solutions of equations of fluid flow, m is rarely more than 5. This method lends itself to efficient use on computers with parallel processing or vector processing architectures. On these computers the larger number of floating point operations is more than offset by the decrease in I/O and the increased percentage of vector operations made possible by this algorithm.

1. Introduction

The most widely used algorithm for solving general systems of linear equations is Gaussian elimination. Other methods have appeared which take advantage of the structure of certain problems, i.e., cyclic reduction for banded matrices with constant coefficients. Most methods thus far devised have had the objective of reducing the total number of floating point operations. The method described here does not.

Our basic objective is to minimize the overall time and cost of solving a given problem. When computers were slow and problems were small the way to do this was to minimize arithmetic. With the advent of supercomputers, however, other considerations have become important.

One such consideration is the ability to vectorize an algorithm. For a given computer, the speed of the vector hardware may exceed that of the scalar hardware by a factor of ten or more. A common way of finding long vectors in a tridiagonal solver is to solve many tridiagonals at once. The vector length then becomes the number of simultaneously solved systems. The storage requirements of such an approach exceed those of the scalar approach by a factor of the vector length.

Another consideration is the time spent in communication with secondary memory. The speed of the arithmetic units makes it possible to solve, in a reasonable amount of time, problems whose storage requirements exceed the capacity of primary memory. On most computers it is difficult to overlap the transfer time between primary and secondary memory. This becomes the dominant cost in some cases. In addition, programs which use secondary memory are often significantly more complex

than those which do not. Furthermore, data transfers between memory levels are hardware-dependent. Programs which do explicit transfers between memory levels are not portable for this reason. Realizing this, we turn our efforts toward an algorithm that requires less memory, even if it requires more arithmetic.

Recomputation is such an algorithm. It offers the user a trade-off between the number of arithmetic operations, time spent in scalar computation, and time spent on data transfers to secondary memories. A FORTRAN subroutine has been written which utilizes recomputation in the solution of block tridiagonal systems. The user can specify, with one parameter, exactly how much storage is available in primary memory and the subroutine will minimize arithmetic subject to this constraint. If there is enough storage the algorithm reduces to the standard, non-recomputing case. The minimum allowable space, not counting the solution vector, is 5 storage blocks, each an $m \times m$ matrix, plus one word. In the sense that this is independent of N, the number of block unknowns, we say that there are no storage constraints.

2. Method

The algorithm described here performs Gaussian elimination to solve a tridiagonal system of equations. This is equivalent to the Thomas algorithm [1]. We will deal only with scalar tridiagonal matrices, the extension to block tridiagonal matrices being relatively straightforward. The notation used is defined by the following equations:

$$
\begin{bmatrix}
b_1 & c_2 & & & & & \\
a_1 & \cdot & \cdot & & & & \\
 & \cdot & \cdot & \cdot & & & \\
 & & b_i & c_{i+1} & & & \\
 & & a_i & b_{i+1} & & & \\
 & & & & \cdot & \cdot & \\
 & & & & & \cdot & c_n \\
 & & & & & a_{n-1} & b_n
\end{bmatrix}
\times
\begin{bmatrix} q_1 \\ q_2 \\ \cdot \\ \cdot \\ \cdot \\ q_i \\ q_{i+1} \\ \cdot \\ \cdot \\ \cdot \\ q_{n-1} \\ q_n \end{bmatrix}
=
\begin{bmatrix} r_1 \\ r_2 \\ \cdot \\ \cdot \\ \cdot \\ r_i \\ r_{i+1} \\ \cdot \\ \cdot \\ \cdot \\ r_{n-1} \\ r_n \end{bmatrix}
\qquad (1)
$$

After the forward elimination has been completed eq. (1) reduces to:

$$
\begin{bmatrix}
1 & c_2' & & & & & \\
 & \cdot & \cdot & & & & \\
 & & \cdot & \cdot & & & \\
 & & & 1 & c_{i+1}' & & \\
 & & & & 1 & \cdot & \\
 & & & & & \cdot & c_n' \\
 & & & & & & 1
\end{bmatrix}
\times
\begin{bmatrix} q_1 \\ q_2 \\ \cdot \\ \cdot \\ \cdot \\ q_i \\ q_{i+1} \\ \cdot \\ \cdot \\ \cdot \\ q_{n-1} \\ q_n \end{bmatrix}
=
\begin{bmatrix} q_1' \\ q_2' \\ \cdot \\ \cdot \\ \cdot \\ q_i' \\ q_{i+1}' \\ \cdot \\ \cdot \\ \cdot \\ q_{n-1}' \\ q_n' \end{bmatrix}
\qquad (2)
$$

The successive steps which are normally taken to solve this problem by Gaussian elimination are:

forward elimination

$$b_1' = b_1 \qquad \text{find } b_1' \tag{3a}$$

$$b_1' q_1' = r_1 \qquad \text{find } q_1' \tag{3b}$$

for i = 1 to n-1

$$b_i' c_{i+1}' = c_{i+1} \qquad \text{find } c_{i+1}' \tag{3c}$$

$$b_{i+1}' = b_{i+1} - a_i c_{i+1}' \qquad \text{find } b_{i+1}' \tag{3d}$$

$$b_{i+1}' q_{i+1}' = r_{i+1} - a_i q_i' \qquad \text{find } q_{i+1}' \tag{3e}$$

backward sweep

$$q_n = q_n' \qquad \text{find } q_n \tag{4a}$$

for i = n to 2 (backward iteration)

$$q_{i-1} = q_{i-1}' - c_i' q_i \qquad \text{find } q_{i-1} \tag{4b}$$

The storage problems with this method stem from the fact that one must compute all of the elements c' and q' before any of the elements q can be computed. The right-hand side is usually overwritten with the solution so that the same storage cell is occupied at various times by r_i, q_i', and finally q_i. Traditionally, both the c' vector and the right-hand side are stored for a total of $2n-1$ storage cells (a total of $(n-1)m^2 + nm$ for a block tridiagonal).

Notice that to compute c_{i+1}' we only need b_i' and c_{i+1}. Also, to compute b_{i+1}' we need only c_{i+1}' and the matrix elements a_i and b_{i+1}.

Schematically, this is shown in Fig. 1. The dashed boxes contain the original matrix entries. In many applications these require no storage, since they are either analytically known or can be recovered from other information contained in memory. We consider such applications here. The main consequence of this simplification is that if any element of the decomposition is known (i.e., b' or c'), then the forward elimination can be reinitiated at that point to get any subsequent decomposition element. This is the basis for the whole scheme. We save a few, selected, elements c' on the forward elimination and then execute the following sequence:

a. Execute the backward substitution in a conventional manner as far as possible.

b. When an element c'_i is needed and not available, recompute it by reinitiating the forward elimination starting with a stored element c'. The best choice is that element whose index is highest without exceeding i. All elements c' with higher indexes have been used already. These may be overwritten to save other indexes of c'. If no stored data remains, recompute from index 1. The element c'_1 is zero.

When the needed element has been recomputed resume step a.

Notice that the forward elimination in step b is analogous to the original forward elimination. The starting and ending indexes are different, as is the amount of available storage, but the form is the same. Thus, we may use the same selection process to decide which elements to save in steb b that we did on the original forward elimination. What follows is a description of and rationale for one such selection process.

Before any computing is done the available storage is divided into two arrays. The first array is for the temporaries. It is dimensioned C(IVEC,M,M,KMAX). The first subscript, typically 64, is the vector length which is also the number of simultaneously solved block tridiagonal systems. The next two subscripts are for referencing inside each block. The last index is the block number. This is the only subscript used by the selection process since all the tridiagonals are solved in parallel. In this discussion it is indexed by the variable K. Since on a given call to the selection routine, some of the array may be occupied, the index KMIN is needed. This is the lowest index of an unoccupied block. All storage cells are addressed by C(IVEC,M,M,K) such that KMIN$\leq$K$\leq$KMAX may be overwritten.

The second array contains pointers and is dimensioned IH(KMAX), indexed in the same way as the first array. The value of IH(K) is the integer I corresponding to the element c_i' which occupies C(, , ,K). The inputs to the selection algorithm are:

IL - The index of the first element c' to be recomputed.

IU - The index of the last element c' to be recomputed.

KMIN - An index to the pointer array IH. Lower indexes may not be used.

KMAX - The size and largest allowable index of the pointer array IH.

IH - The pointer array. On output it contains indices of elements to save on the subsequent forward elimination.

The motivation behind choosing the following selection process is to minimize the number of recomputations subject to the storage

constraints. To this end we remember the cost of computing a given temporary and avoid overwriting any temporary with one that is cheaper.

First, we naively assume that a single computation will be enough. Thus we set IH(KMIN)=IL, IH(KMIN+1)=IL+1, and so on up to IH(KMAX). If IH(KMAX)≥IU then our assumption is correct and the selection process terminates. In this case the recomputing algorithm reduces to the Thomas algorithm.

If IH(KMAX)<IU then some recomputing is necessary. The temporaries corresponding to indexes in the pointer array are all equally expensive to recompute in the following sense: they all may be recovered with one computation per element by restarting the forward elimination (using the fact that $c'_1 = 0$). Thus we may overwrite them all. This may be noted by adding IH(KMAX)-IH(KMIN)+1 to each element in the pointer array. If this would yield IH(KMAX)>IU we should add IU-IH(KMAX) instead, so that IH(KMAX)=IU. Thus, the amount to be added is MIN(IH(KMAX)-IH(KMIN)+1,IU-IH(KMAX)). At this point, if IH(KMAX)=IU the selection process is completed.

Otherwise we must overwrite further, picking the indexes which are least expensive to recover. This time they are not all equally inexpensive to recompute. Recomputing $c'_{IH(KMIN)}$ would require three computations of c'_{IL}. This being the case, we choose not to overwrite IH(KMIN) yet. The temporaries corresponding to other elements of the pointer array can be computed at a total cost of two computations if $c'_{IH(KMIN)}$ is used as a starting point. We define here the variable KP, initially KMIN but now incremented to KMIN+1. The rationale of the previous paragraph is used to justify adding MIN(IH(KMAX)-IH(KP)+1,

IU-IH(KMAX)) to all IH(K) such that KP≤K≤KMAX. As before, if after this is done IH(KMAX)=IU, then the selection process is completed. If IH(KMAX)≠IU, then we can use the above argument to show that computing $c'_{IH(KP)}$ would be relatively expensive, implying three computations for $c'_{IH(KP-1)+1}$. We respond by incrementing KP and repeating the process until either the process is completed (i.e., IH(KMAX)=IU) or KP exceeds KMAX.

The second case states that even if one is willing to compute everything twice there is not enough storage. The temporaries corresponding to the elements in the pointer array would all cost two computations to recompute. Consequently, to complete the selection process we have to be willing to compute some elements three times. In terms of the variables in the selection process, this means resetting KP to KMIN. Otherwise everything is the same. Simply keep adding MIN(IH(KMAX)-IH(KP)+1,IU-IH(KMAX)) and incrementing or resetting KP as appropriate until IH(KMAX)=IU. A simple FORTRAN subroutine to accomplish this algorithm in less than 20 executable statements is given in the appendix. We will now illustrate the selection algorithm and the recomputing scheme by an example.

Example

Suppose n = 11 and there is only room to store 3 temporaries. The conventional Thomas algorithm requires 10 temporaries. The selection algorithm would proceed as follows:

1. Initially IL=2,IU=11,KMIN=1,KMAX=3

2. Naively assume that no recomputation is required. Set

   IH(1)=2

   IH(2)=3

   IH(3)=4

3. Set KP=KMIN=1. Then MIN(IH(KMAX)-IH(KP)+1,IU-IH(KMAX))
   =MIN(3,7)=3

   Adding this to all IH(K) from K=KP to K=KMAX gives

   IH(1)=5

   IH(2)=6

   IH(3)=7

4. Increment KP so that KP=2. NOW MIN(IH(KMAX)-IH(KP)+1,
   IU-IH(KMAX))=MIN(2,4)=2

   Adding this to all IH(K) from K=KP to K=KMAX gives

   IH(1)=5

   IH(2)=8

   IH(3)=9

5. Increment KP so that KP=3. Now MIN(IH(KMAX)-IH(KP)+1,
   IU-IH(KMAX))=MIN(1,2)=2

   Adding this to all IH(K) from K=KP to K=KMAX gives

   IH(1)=5

   IH(2)=8

   IH(3)=10

6. Increment KP so that KP=4. Since this exceeds KMAX we reset it so that KP=1. Now MIN(IH(KMAX)-IH(KP)+1, IU-IH(KMAX))=MIN(6,1)

   Adding this to all IH(K) from K=KP to K=KMAX gives

   IH(1)=6

   IH(2)=9

   IH(3)=11

7. Since [IH(KMAX)=IU] = [IH(3)=11], the selection process is completed. It says that we should save the elements c_6', c_9', and c_{11}' on the forward elimination.

The recomputing algorithm would proceed as follows:

1. Initial forward sweep. Save c_6', c_9', and c_{11}'.
2. Backward sweep. Comput q_{11} and q_{10}. To compute q_9 we require c_{10}'.
3. Since q_{10} is already computed, c_{11}' is not needed. Resume forward sweep using c_9' and overwrite c_{11}' with c_{10}'.
4. Continue the backward sweep, using c_{10}' and c_9' to compute q_9 and q_8.
5. Resume forward sweep with c_6', overwriting c_9' and c_{10}' with c_7' and c_8'.
6. Continue the backward sweep, computing q_5, q_6, and q_7.
7. Resume forward sweep from index 1, overwriting c_6', c_7', and c_8' with c_3', c_4', and c_5'.
8. Continue backward sweep by computing q_2, q_3, and q_4.

9. Again resume forward sweep from index 1, overwriting c'_3 with c'_2.

10. Conclude the backward sweep by computing q_1.

Each forward sweep except the last used all the storage containing elements that were no longer needed. Temporaries c'_6, c'_9, and c'_{11} were computed only once. Temporary c'_2 was computed three times. All the rest were computed twice. The total cost was almost twice that of the conventional Thomas algorithm, yet the required storage was less than the square root of that required by the conventional algorithm. In larger problems it is often possible to reduce the storage requirements by a factor of ten while only doubling the arithmetic, a paying proposition if I/O is expensive. It is interesting to note that the minimum required storage space is five blocks, each an $m \times m$ matrix plus one word. This is extremely expensive, however, the computational effort being higher than the nonrecomputing case by a factor of roughly $n^2/2$. Three of the blocks are needed for the block elements A, B, and C. One is needed for the intermediate B' and the last is needed for the temporary C'. One additional word is needed for the pointer array IH to keep track of the one temporary. The flowchart in Fig. 2 illustrates how the selection process and the recomputation algorithm fit into the Thomas algorithm.

3. Discussion

The standard Thomas algorithm has the following operation count.

$$\begin{aligned} &\text{Multiplications} && N(7m^3/3 + 3m^2 - m/3) - 2m^2(m+1) \\ &\text{Additions} && N(7m^3/3 + 3m^2/2 - 5m/6) - 2m^2(m+1) \\ &\text{Divisions} && Nm\ . \end{aligned} \tag{5}$$

Here N is the number of block unknowns and m is the dimension of each block. If N >> 1 the second term in the addition and multiplication counts may be neglected. Using as a measure of relative cpu time the equivalency formula

$$1 \text{ add} = 1 \text{ multiply} = 1/4 \text{ divide} \tag{6}$$

the total number of operations becomes

$$N\{14m^3/3 + 9m^2/2 + 17m/6\} . \tag{7}$$

The variable T is defined as the total number of operations divided by the quantity in brackets. In this way the dependence on m of the results is largely removed. The variable KMAX, defined above, is a measure of the available storage. Finally N, also defined above, is a measure of the problem size. Figure 3 plots T vs N for various values of the parameter KMAX. This figure describes the common situation in which a computer has a limited total memory. This occurs when secondary memory is much slower than primary memory. Often the secondary memory is a disk or a standard tape drive. In the case of the current generation of micro-computers, it may even be a cassette. In such a case recomputing may allow the solution of problems that otherwise could not be solved at all. Figure 3 gives, at a glance, the cost of solving block tridiagonal systems as a function of problem size given a fixed amount of memory.

Another situation for which recomputing can be helpful is where a program has been written, the problem size is fixed, and the user wishes to modify the program in some way that requires more memory than the computer has. One way to get more memory is to reduce the amount of

space allocated for temporaries used in solving block tridiagonals. Depending on the problem this may free a significant amount of storage. In this way the user may avoid a complete rewrite which might otherwise be necessary to incorporate transfers to secondary memory. Depending on the accounting algorithm for the computer in question, recomputing may even be cheaper than transfers to secondary memory. Experience has shown, however, that recomputing rarely pays on a cost/run basis if any element c' is computed more than twice.

A situation sometimes arises in which the total computer time is fixed. From this constraint one may estimate the maximum number of times each element may be computed. We call this number P. Given P and the storage constraint KMAX there is a limit to the number of block unknowns we can solve for. We call this number NMAX. The question arises: What is the relationship between P,KMAX, and NMAX? Such information could be useful in deciding on a vector length or deciding if this algorithm would pay at all. It can be shown that the recursion relation for finding NMAX(P,KMAX) is

$$NMAX(P, \quad 1) = P + 1 \tag{8a}$$

$$NMAX(1,KMAX) = KMAX + 1 \tag{8b}$$

$$NMAX(P,KMAX) = NMAX(P,KMAX-1) + NMAX(P-1,KMAX) \tag{8c}$$

We notice immediately that NMAX(P,KMAX)=NMAX(KMAX,P), that is, the function NMAX is symmetrical about the line P=KMAX. Also, along a line where P (or KMAX) is a constant, the values of NMAX may be exactly fitted by a polynomial of degree P (or KMAX). The first few and the general case are given here.

$$P=1 \quad NMAX = 1 + KMAX \tag{9a}$$

$$P=2 \quad NMAX = 1 + 3/2\ KMAX + 1/2\ KMAX^2 \tag{9b}$$

$$P=3 \quad NMAX = 1 + 11/6\ KMAX + KMAX^2 + 1/6\ KMAX^3 \tag{9c}$$

$$P=z \quad NMAX = \sum_{m=1}^{z+1} \left(\left| (S_{z+1}^{(m)}) \right| * KMAX^{m-1}/z! \right) \tag{9d}$$

In the general case, $S_{z+1}^{(m)}$ are Stirling numbers of the first kind. Thus we see that the highest order term is always $KMAX^z/z!$. Equation (9d) is given without proof. In principal, however, one could substitute Eq. (9d) into (8c) to prove the equality.

The recomputation algorithm is arithmetically the same as the Thomas algorithm; hence, it has exactly the same stability properties and gives the same answer. Although the storage overhead is usually negligible it does require KMAX scalar temporaries. This should be compared with $KMAX*m^2*IVEC$ temporaries used in the rest of the computation or with $(N-1)*m^2*IVEC$ temporaries needed for the Thomas algorithm. The computational overhead is equally negligible, involving less than N floating point operations. If no recomputation is done, the recomputation algorithm costs virtually the same to use as a conventional Thomas algorithm.

4. Conclusions

It has proved useful to program the entire block tridiagonal solver as a subroutine which has, as an argument, the amount of available space. This substantially reduces the consequences of programming at the limit of primary memory. This alone helps increase productivity

through reducing the number of times programs are rewritten to free a tiny amount of storage.

Using recomputation, problem size can be substantially increased on computers where memory size is poorly matched to processor speed for this type of problem. Recomputation can free enough memory to allow the effective use of vector processing capabilities on machines like the Cray 1 and the CDC 7600. Furthermore, the extra computation required is largely made up of dot products, at which these machines are very efficient.

Surprisingly, execution speed can actually be increased through the use of recomputation. This can occur when disk latency becomes a substantial portion of the code's running time. It can also occur when recomputation is used to increase vector lengths. In both cases costs can be reduced by doing more arithmetic, keeping the job in core using recomputation. This algorithm was used on Illiac IV codes at Ames Research Center from 1977 until the Illiac was replaced in 1981 [2]. The required vector length of 64 and the small size of primary memory made recomputation a virtual necessity on the Illiac, allowing the solution of problems that otherwise could not have been solved. Any time a situation arises where it costs more to bring a problem in and out of core than it does to perform the arithmetic, or where many problems must be solved in parallel, recomputation is likely to pay.

Of course, this general approach is not limited to tridiagonal matrices. It can easily be extended to cover periodic and pentadiagonal matrices. For that matter, it can be used to solve dense or wide-banded matrices. It is not limited to Gaussian elimination but is

applicable to any method with a forward and backward sweep that saves intermediate results.

Larger and faster memories may temporarily reduce the need for recomputation but cannot remove its advantages. As long as there are substantial differences in speed between memory hierarchies, computers that don't match memory to processor speeds, computers with vector speeds significantly higher than scalar speeds, or problems which are computationally light relative to their size, there will be a need for recomputation schemes.

References

1. W. F. Ames, "Mathematics in Science and Engineering," Vol. 18, pp. 341-2, Academic Press, New York, N.Y., 1965.
2. J. Kim and P. Moin, "Large Eddy Simulation of Turbulent Channel Flow — Illiac IV Calculation," Proc. AGARD Symp. on Turbulent Boundary Layers, Experiment, Theory, and Modelling, the Hague, The Netherlands, Sept. 24-26, 1979; also, NASA TM-78619, Sept. 1979.

Appendix

```
      SUBROUTINE CHOOSE(IL,IU,KMIN,KMAX,IH)
C     THIS SUBROUTINE IS DESIGNED TO CHOOSE THE OPTIMUM INDICES
C     AT WHICH TO SAVE DECOMPOSITION ELEMENTS C'.
C
C     ARGUMENT DESCRIPTION
C
C     IL          INPUT-THE LOWEST INDEX FOR WHICH C'(I) MUST BE COMPUTED.
C
C     IU          INPUT-THE HIGHEST INDEX FOR WHICH C'(I) IS NEEDED.
C
C     KMIN        INPUT-C(KMIN-1) CONTAINS THE DECOMPOSITION ELEMENT
C                 NEEDED TO RESTART THE FORWARD SUBSTITUTION.  INITIALLY
C                 KMIN=1.
C
C     KMAX        INPUT-THE TOTAL NUMBER OF ELEMENTS C'(I) WHICH CAN BE
C                 STORED. ON THIS CALL TO CHOOSE WE CAN PICK UP TO
C                 KMAX-KMIN+1 ELEMENTS.
C
C     IH          OUTPUT-AN ARRAY OF LENGTH KMAX WHICH CONTAINS THE
C                 INDICES OF THE ELEMENTS C'(I) WE WISH TO STORE.
C
      DIMENSION IH(KMAX)
      DO 10 K = KMIN,KMAX
           IH(K)=K-KMIN+IL
   10 CONTINUE
C                 THIS BRANCH IS TAKEN IF THERE IS SUFFICIENT SPACE TO
C                 AVOID RECOMPUTING.
      IF (IH(KMAX) .GT. IU) GOTO 6
C
C
      KK=KMIN
    2 IF ( (2*IH(KMAX)+1-IH(KK)) .GE. IU ) GOTO 4
C                 THIS SECTION IS EXECUTED IF OVERWRITNG ALL THE CHEAP
C                 SPACE IS NOT SUFFICIENT.
      KC=IH(KMAX)-IH(KK)+1
      DO 3 K = KK,KMAX
       IH(K)=IH(K) + KC
    3 CONTINUE
      PRINT*,(IH(J),J=1,20)
      KK=KK+1
C                 WHEN THIS TEST PASSES ANOTHER LEVEL OF RECOMPUTATION IS
C                 REQUIRED.
      IF (KK .GT. KMAX) KK=KMIN
      GOTO 2
C                 THIS SECTION IS EXECUTED SO THAT IH(KMAX) IS IU.  IT
C                 ALSO ENSURES THAT IF SOME ELEMENTS MUST BE RECOMPUTED
C                 MORE TIMES THAN OTHERS THEY ARE DONE LAST, WHEN THERE
C                 IS MORE SPACE.
    4 CONTINUE
      IC=IU-IH(KMAX)
      DO 5 K = KK,KMAX
           IH(K) = IH(K) + IC
    5 CONTINUE
C
    6 RETURN
      END
```

Figure Captions

Fig. 1. Data dependencies in a tridiagonal.

Fig. 2. The Thomas algorithm with recomputation.

Fig. 3. A summary of tradeoffs offered by recomputation.

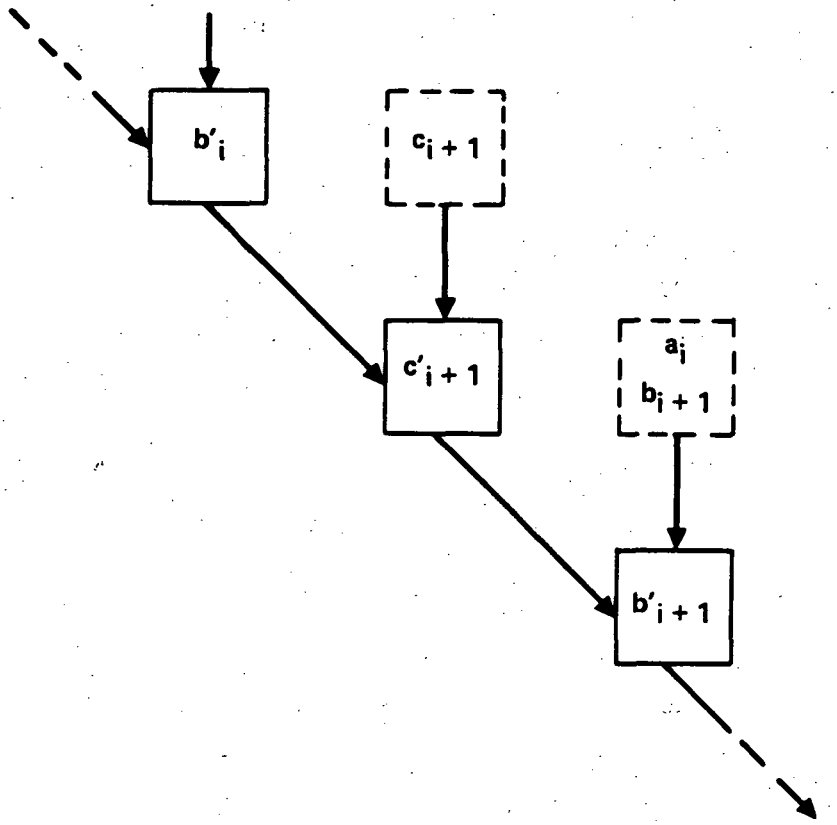

Fig. 1

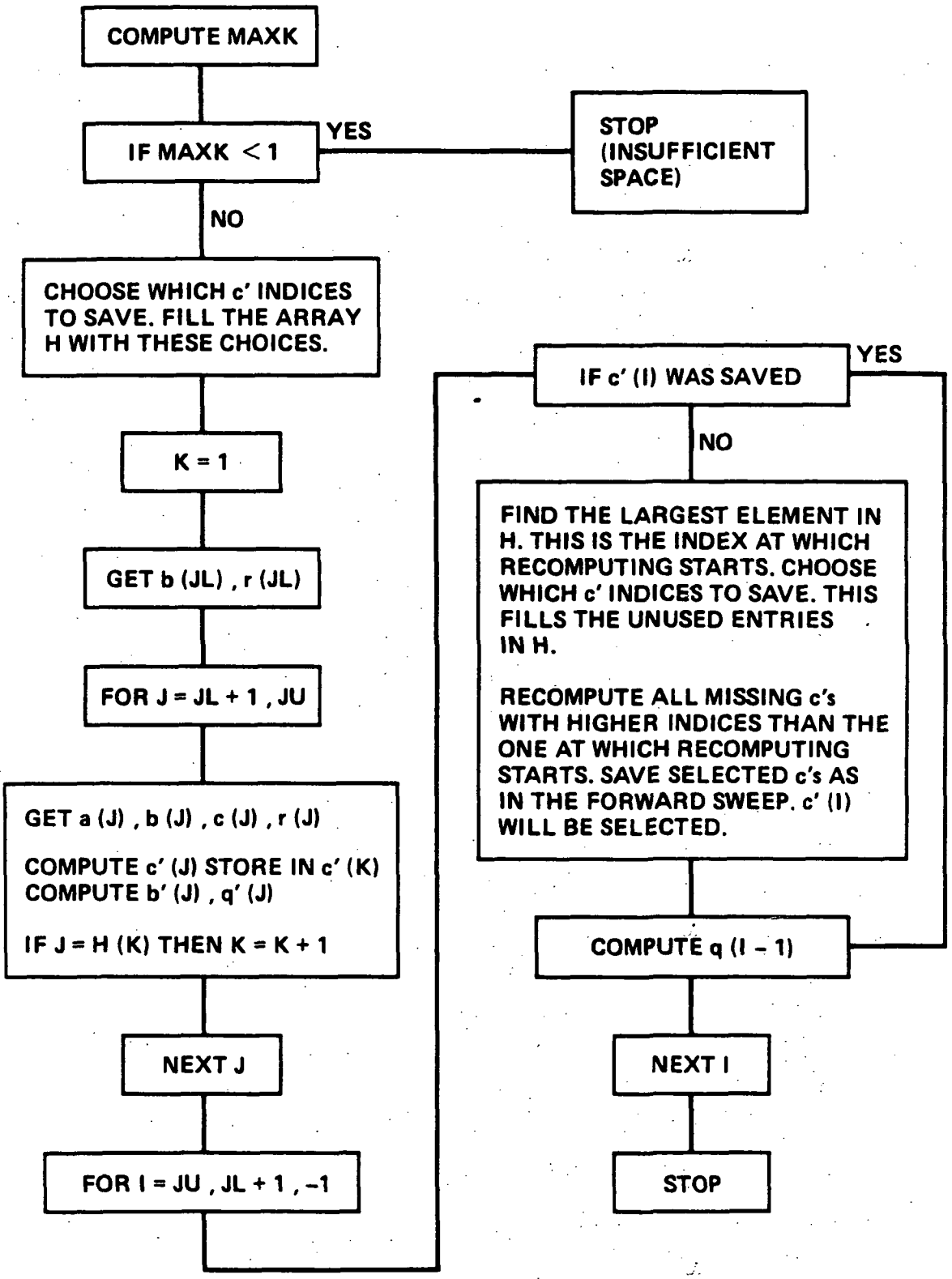
COMPUTE MAXK
IF MAXK < 1
YES
STOP (INSUFFICIENT SPACE)
NO
CHOOSE WHICH c' INDICES TO SAVE. FILL THE ARRAY H WITH THESE CHOICES.
K = 1
GET b (JL) , r (JL)
FOR J = JL + 1 , JU
GET a (J) , b (J) , c (J) , r (J)
COMPUTE c' (J) STORE IN c' (K)
COMPUTE b' (J) , q' (J)
IF J = H (K) THEN K = K + 1
NEXT J
FOR I = JU , JL + 1 , -1
IF c' (I) WAS SAVED
YES
NO
FIND THE LARGEST ELEMENT IN H. THIS IS THE INDEX AT WHICH RECOMPUTING STARTS. CHOOSE WHICH c' INDICES TO SAVE. THIS FILLS THE UNUSED ENTRIES IN H.
RECOMPUTE ALL MISSING c's WITH HIGHER INDICES THAN THE ONE AT WHICH RECOMPUTING STARTS. SAVE SELECTED c's AS IN THE FORWARD SWEEP. c' (I) WILL BE SELECTED.
COMPUTE q (I - 1)
NEXT I
STOP

Fig. 2

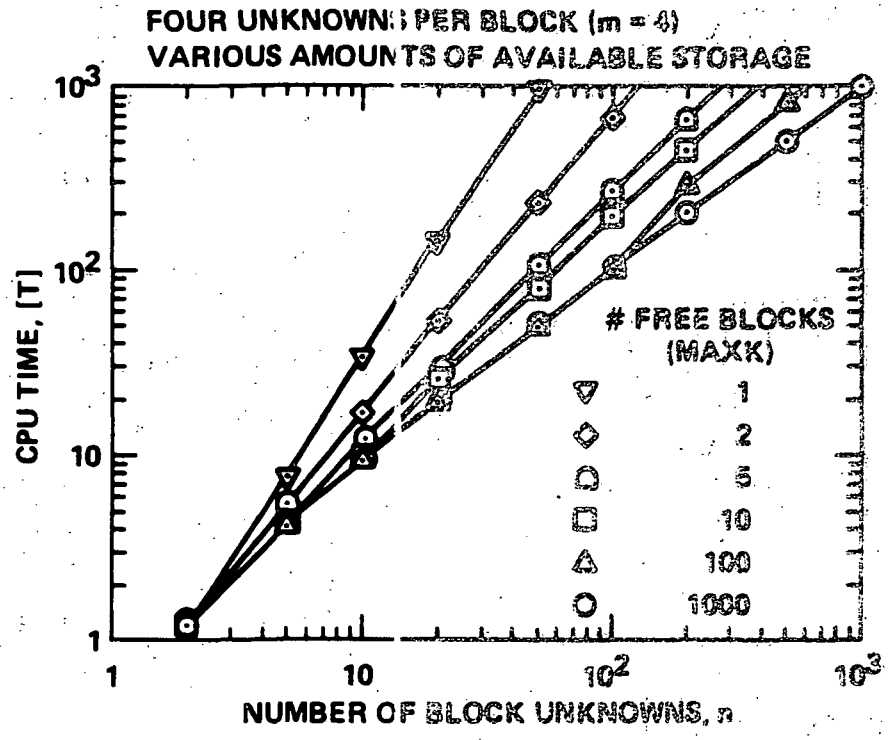
FOUR UNKNOWNS PER BLOCK (m = 4)
VARIOUS AMOUNTS OF AVAILABLE STORAGE
CPU TIME, [T]
NUMBER OF BLOCK UNKNOWNS, n
FREE BLOCKS (MAXK)
1
2
5
10
100
1000
1
10
10^2
10^3

Fig. 3

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
TM-84228		

4. Title and Subtitle	5. Report Date
ON THE INVERSION OF BLOCK-TRIDIAGONALS WITHOUT STORAGE CONSTRAINTS	March 1982
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Marshal L. Merriam	A-8848
	10. Work Unit No.
9. Performing Organization Name and Address	T-9517
NASA Ames Research Center Moffett Field, Calif. 94035	11. Contract or Grant No.
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Memorandum
National Aeronautics and Space Administration Washington, D.C. 20546	14. Sponsoring Agency Code 505-31-11-02-00-21

15. Supplementary Notes

Point of Contact: Marshal L. Merriam, Ames Research Center, M.S. 202A-1, Moffett Field, CA, (415) 965-6417 or FTS 448-6417.

16. Abstract

In many programs solving difference equations, problem size is restricted by the number of available memory cells. A strategy has been developed to permit trade-offs between the number of floating point operations required and the storage requirements for the solution of certain problems such as block tridiagonal systems of equations. This is done by recomputing some intermediate results instead of storing them. Reducing the storage to the square root of the current requirement will roughly double the number of computations. Reducing the storage more than this tends to make the number of computations prohibitively large. In theory though, if m is the order of each sub-matrix in the block tridiagonal matrix, one can solve any linear system with only $5m^2 = 1$ temporary storage cells. In many cases m is a constant and quite small. For example, in solving a factored form of the three-dimensional Navier-Stokes equations, the size m of the block tridiagonals is 5. In fact, for block tridiagonals arising from finite difference solutions of equations of fluid flow, m is rarely more than 5. This method lends itself to efficient use on computers with parallel processing or vector processing architectures. On these computers the larger number of floating point operations is more than offset by the decrease in I/O and the increased percentage of vector operations made possible by this algorithm.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Block Tridiagonals Storage Recompute	Unlimited Subject Category - 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	27	A03

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