INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

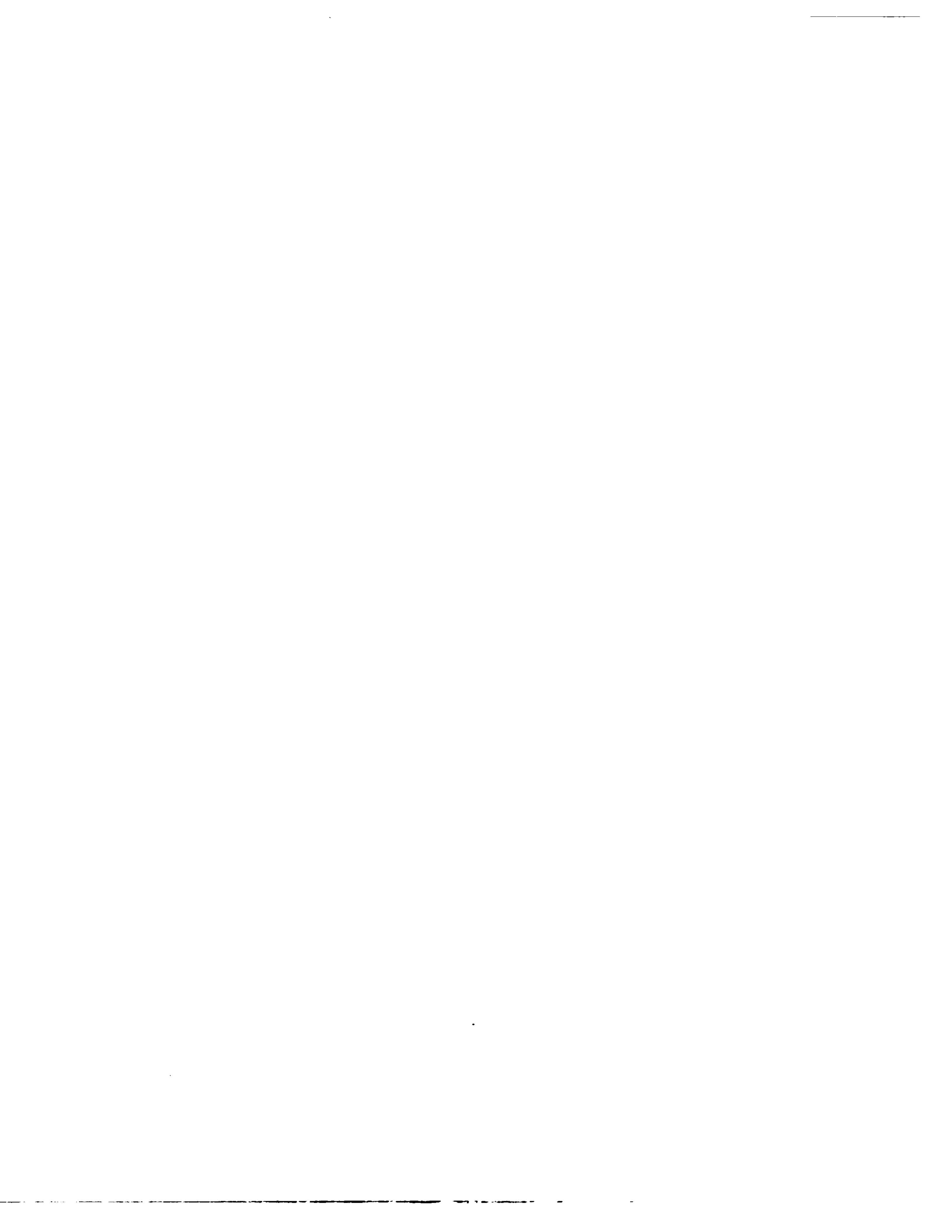
The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.
2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.
3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*
4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

*For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.

U·M·I Dissertation Information Service

University Microfilms International
A Bell & Howell Information Company
300 N. Zeeb Road, Ann Arbor, Michigan 48106



8629748

Thurm, Joseph

SOLVING LARGE FULL SETS OF LINEAR EQUATIONS ON A MICROCOMPUTER

City University of New York PH.D. 1986

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages __✓__
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _________ lacking when material received, and not available from school or author.
12. Page(s) _________ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received _______
16. Other ____

University
Microfilms
International

SOLVING LARGE FULL SETS
OF LINEAR EQUATIONS
ON A MICROCOMPUTER

by

Joseph Thurm

A dissertation submitted to the
Graduate Faculty in Computer Science
in partial fulfillment for the
degree of Doctor of Philosophy,
The City University of New York.

1986

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

5/27/86
date

Pat H. Sterbenz
Professor Pat Sterbenz
Chairman of Examining Committee

5/28/86
date

F S Beckman
Professor Frank Beckman
Executive Officer
Ph.D. Program in Computer Science

Professor Michael Anshel
Professor Theodore Brown
Professor Seymour Lipschutz

Supervisory Committee

The City University of New York

Abstract

Solving Large Full Sets of Linear Equations
on a Microcomputer

by

Joseph Thurm

Adviser: Professor Pat Sterbenz

The solution of $Ax=b$, where the entire matrix A cannot fit into RAM, is examined. Various methods are presented for partitioning A into submatrices, which are then stored on disk until needed. Based on the partitioning scheme, algorithms decomposing A into LU factors are developed and analyzed. Since these algorithms do explicit disk reads and writes for the submatrices of A, the selection of the algorithm which does the least amount of I/O is of paramount importance.

The row storage method stores several complete rows together in a submatrix, so the original matrix A is viewed as an array of blocks. The maximum size of each block is approximately one-half of available memory. Since all the decomposition algorithms for this storage method require two blocks to be in memory at once, the row storage method assumes that at least two complete rows will fit into memory at once.

The column storage method stores several complete columns together in a submatrix, so the matrix A is viewed as a vector of blocks. Again, the maximum size of each block is approximately one-half of available memory, and all the decomposition algorithms for this storage method require that at least two complete columns will fit into memory at once.

The submatrix storage method divides A into a matrix of square blocks. This storage method does not require two complete rows or columns to fit into memory at once. Some of the decomposition algorithms require that three blocks be in memory at once, limiting the size of a block to one-third of memory. Other decomposition algorithms only require two blocks to be in memory at once, allowing the blocks to grow to one-half of memory.

Based on the amount of memory available and the dimensions of A, the storage method and associated decomposition algorithm which do the least amount of I/O are determined from the results presented.

Table of Contents

Chapter 1
Overview 1

Chapter 2
Basic Storage Methods 5
Row Storage 6
Column Storage 7
Submatrix Storage 8

Chapter 3
Glossary of Variables 10
Common Variables 10
Row Storage 12
Column Storage 14
Submatrix Storage 16

Chapter 4
Decomposition Algorithms without Interchanges 17
Row Storage
Algorithm ROW#1 18
Algorithm ROW#2 21
Algorithm ROW#3 23
Column Storage
Algorithm COL#1 25
Comparing Algorithms ROW#3 with COL#1 28
Submatrix Storage
Algorithm THREESUB#1 29
Algorithm THREESUB#2 37
Algorithm THREESUB#3 41
Algorithm THREESUB#4 56
Algorithm THREESUB#5 63
Algorithm THREESUB#6 69
Algorithm THREESUB#7 76
Algorithm THREESUB#8 80
Algorithm THREESUB#9 87
Algorithm THREESUB#10 91
Comparing Algorithms THREESUB#1 thru THREESUB#10 95
Comparing Algorithms THREESUB#10 with COL#1 102
A Point Worth Reiterating 114

Table of Contents

Chapter 5	
A Variation on the Submatrix Storage Method	115
Basic Idea	115
The Hidden Vector	116
Common Variables	119
Decomposition Algorithms without Interchanges	
Algorithm TWOSUB#1	121
Algorithm TWOSUB#2	129
Algorithm TWOSUB#3	133
Algorithm TWOSUB#4	138
Algorithm TWOSUB#5	144
Comparing Algorithms TWOSUB#1 thru TWOSUB#5	157
Comparing Algorithms TWOSUB#5 with THREESUB#10	160
Comparing Algorithms TWOSUB#5 with COL#1	171
Chapter 6	
Solving for x	175
Applying the L Multiplier Values to b	175
Solving for x using the U Factors and b	176
Row Storage	177
Column Storage	179
Three Square Storage	181
Two Square Storage	190
Chapter 7	
Final Discussion	191
Chapter 8	
Future Research	197
Rectangular Storage Methods	197
Decomposition Algorithms With Interchanges	199
References	
Cited	201
Uncited	202
Sparse Matrix	207

Chapter 1

Overview of the Problem

The solution of simultaneous linear equations is of practical importance in many diverse fields.

Solving large sets of simultaneous equations by hand is tedious and time consuming. One of the first application programs written for the computer was the solution of Ax=b. Von Neumann and Goldstine (1949, 1953), and Forsythe (1953) are the earliest discussions of the computer solution of simultaneous linear equations. In these papers, computer memory size was the only limiting factor. Usually, the entire matrix had to fit into memory.

Bound by this constraint, much research was done in utilizing any special properties the matrix had to reduce the amount of storage needed. Since auxiliary storage was slow and expensive, there was minimal research done on using auxiliary storage. Barron and Swinnerton-Dyer (1960) is one of the few papers that deals with auxiliary tape storage solutions of Ax=b for full matrices.

However, problems involving sparse matrices, especially those arising from partial differential equations, did use auxiliary storage. A survey of the storage methods and associated solution algorithms for sparse matrices is presented in one of the appendices.

Bunch and Parlett (1971) present a variety of direct solutions for large symmetric indefinite matrices. Most of the methods involve storing the n by n matrix as a vector of length n*(n+1)/2 where the element (i,j) can be found as the element k of the vector, where k = (i*(i-1)/2) + j for i>=j. For i<j, the element (i,j) is identical to the element (j,i). Storing a matrix in this manner effects a savings of n*(n-1)/2 memory locations.

All of these methods assumed the reduced matrix would fit into storage. Therefore, storage was still a large limiting factor.

Large dense matrices with no memory saving characteristics remained a largely unsolved problem. The only solution appeared to be more memory.

As is well known, as computer technology advanced larger memories became more readily available and virtual memories became common. When large dense matrices were actually solved on paged memory machines, a tremendous amount of paging occurred. Much research was done on reducing the number of page faults. The research centered on organizing the loops in the algorithm to conform to page boundaries.

McKellar and Coffman (1969) present an algorithm based on several complete rows of the matrix per page. Moler (1972) presents an algorithm based on FORTRAN's column storage of matrices. Nugent and DuCroz (1981b) present an algorithm based on several complete columns of the matrix per page.

In the research done on page fault reduction, very little attention was given to the rewriting of updated pages. It was taken for granted that the operating system would rewrite updated pages as needed. Little thought was given to organizing the algorithm to minimize the number of rewrites needed.

Today as microcomputers become ever more common, another look at the problem of large matrix manipulation is warranted. Most microcomputer operating systems do not have virtual memory. Since the memory on most microcomputers is relatively small, many dense matrices do not fit into storage. Even though microcomputers are relatively slow in comparison to today's mainframes, and the solution of simultaneous equations on mainframes does take a considerable amount of time, analysis of large matrix manipulation algorithms on microcomputers does merit some attention. Typically, the slow computational problems are left for overnight processing, while microcomputers are used for fast computations during working hours.

This dissertation will examine algorithms to solve large dense matrices on unpaged limited-memory computers using random access auxiliary storage. We shall also examine algorithms which minimize the amount of I/O needed to solve large dense matrices. Even those using computers with lots of memory and virtual storage will find these algorithms valuable.

In addition, some of the algorithms examined have practical application in the field of parallel processing. The partitioning of the matrix and the looping patterns are crucial issues when more than one processing unit is manipulating the matrix.

Chapter 2

Basic Storage Methods

As stated previously, we shall consider the case where matrix A does not fit into main memory. Therefore, we must partition matrix A into blocks. These blocks are stored on disk and are brought into memory as needed. If a block is updated while in memory, the block must be rewritten to disk.

There are three basic ways to divide the matrix A.

Row Storage

Several complete consecutive rows are grouped together.

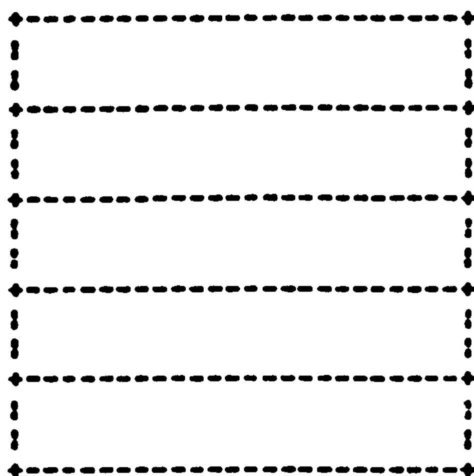


diagram 1.

This storage method assumes that at least one complete row will fit into memory at once.

Moreover, since some matrix decomposition algorithms involve two rows simultaneously, this method may only be viable when two complete rows fit into memory.

As can be seen from the diagram, the row storage method divides the matrix into a vector of blocks. Most of the algorithms presented for this storage method view the matrix A as an array of blocks, rather than as a matrix of individual elements.

The smallest possible block contains one complete row. The largest possible block contains half the matrix.

Column Storage

Several complete consecutive columns are grouped together.

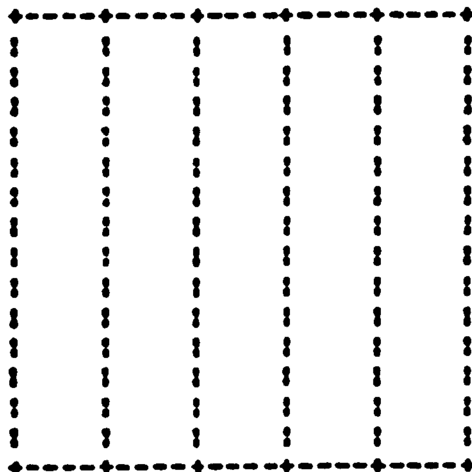


diagram 2.

This storage method also assumes that at least one complete column will fit into memory at once.

Just as in the row storage method, some matrix decomposition algorithms involve two columns simultaneously. Therefore, this method may only be viable when two complete columns fit into memory.

Continuing the analogy, the column storage method also divides the matrix into a vector of blocks. Most of the algorithms presented for this storage method view the matrix A as an array of blocks, rather than as a matrix of individual elements.

The smallest possible block contains one complete column. The largest possible block contains half the matrix.

Submatrix Storage

Several vertical row segments are grouped together.

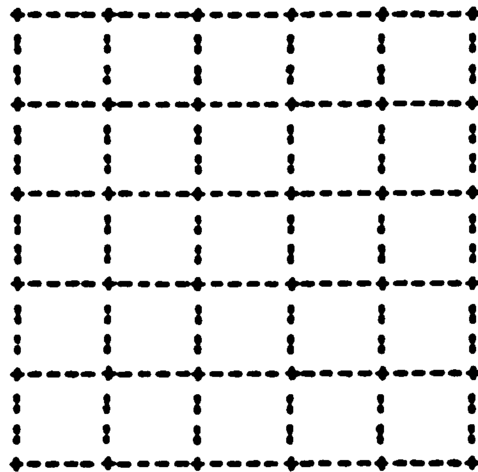


diagram 3.

This storage method does not assume that a complete row or column will fit into memory at once.

However, since most decomposition algorithms contain statements like:

A (i,j) = A (i,j) - (A (i,k) * A (k,j))

where (i,j), (i,k), and (k,j) may be in three different blocks, this storage method appears to be viable only when three complete blocks fit into memory at once.

In essence, the submatrix storage method divides the matrix into another matrix. The original matrix consisted of rows and columns of elements. The submatrix storage method creates a matrix consisting of rows and columns of blocks. In fact, most of the algorithms presented will view the matrix as a matrix of blocks and not of individual elements.

As has been noted, the row and column storage schemes require two full rows or columns to fit into memory at once.

Since the dimensions of A are given and are often quite large, there are matrices that cannot be solved on certain limited-memory computers.

As has been noted, the submatrix scheme requires three complete blocks to fit into memory at once.

On the other hand, the submatrix scheme will solve all matrices on any machine. The size of each block can be tailored to fit the given computer.

Chapter 3

Glossary of Variables To Be Used

Common Variables

For each of the three storage methods presented above and its associated matrix manipulation algorithms, there are several common variables. Before presenting any algorithms, the declarations and value assignments of these variables should be described.

Here are the common variables and their PL/I declarations. The names of the variables have been chosen to be descriptive of the values they will contain.

```
DCL MEMORY_FOR_DATA_SIZE_IN_BYTES         FIXED BIN (31,0);

DCL BYTES_PER_ELEMENT                     FIXED BIN (31,0);

DCL MEMORY_SIZE_IN_ELEM                   FIXED BIN (31,0);


DCL NUM_ELEM_PER_ROW_OF_MATRIX            FIXED BIN (31,0);
DCL NUM_ELEM_PER_COL_OF_MATRIX            FIXED BIN (31,0);

DCL NUM_ELEM_PER_ROW_OF_SUBMATRIX         FIXED BIN (31,0);
DCL NUM_ELEM_PER_COL_OF_SUBMATRIX         FIXED BIN (31,0);

DCL NUM_SUBMAT_PER_COL                    FIXED BIN (31,0);
DCL NUM_SUBMAT_PER_ROW                    FIXED BIN (31,0);

DCL FLOAT1                                F_OAT;
DCL FLOAT2                                FLOAT;
DCL FLOAT3                                FLOAT;

DCL TEMP1                                 FIXED BIN (31,0);
DCL TEMP2                                 FIXED BIN (31,0);
DCL TEMP3                                 FIXED BIN (31,0);

DCL BLOCK1 (NUM_ELTS_PER_ROW_OF_SUBMATRIX,
            NUM_ELTS_PER_COL_OF_SUBMATRIX)     FLOAT CTL;
DCL BLOCK2 (NUM_ELTS_PER_ROW_OF_SUBMATRIX,
            NUM_ELTS_PER_COL_OF_SUBMATRIX)     FLOAT CTL;
DCL BLOCK3 (NUM_ELTS_PER_ROW_OF_SUBMATRIX,
            NUM_ELTS_PER_COL_OF_SUBMATRIX)     FLOAT CTL;


GET LIST (NUM_ELEM_PER_ROW_OF_MATRIX,
          MEMORY_FOR_DATA_SIZE_IN_BYTES,
          BYTES_PER_ELEMENT);

MEMORY_SIZE_IN_ELEM    = FLOOR (MEMORY_FOR_DATA_SIZE_IN_BYTES /
                                 BYTES_PER_ELEMENT);

NUM_ELEM_PER_COL_OF_MATRIX    = NUM_ELEM_PER_ROW_OF_MATRIX;
```

Row Storage

For the row storage method, the number of elements in a row of a submatrix is equal to the number of elements in a row of the original matrix. Therefore, the number of complete rows contained in a block will be dependent on the amount of memory available. Since at least two blocks must be able to reside in memory, the number of rows contained in a block is determined by the number of complete rows that fit into half of memory.

After computing the dimensions of each block, the number of these blocks can be easily determined.

Here are the PL/I statements that reflect the discussion above.

```
TEMP1 = FLOOR (FLOAT(MEMORY_SIZE_IN_ELEM) / 2.);
TEMP2 = FLOOR (FLOAT(TEMP1) /
               FLOAT(NUM_ELEM_PER_ROW_OF_MATRIX));

NUM_ELEM_PER_ROW_OF_SUBMATRIX = NUM_ELEM_PER_ROW_OF_MATRIX;
NUM_ELEM_PER_COL_OF_SUBMATRIX = TEMP2;

NUM_SUBMAT_PER_COL  = CEIL
                        (FLOAT(NUM_ELEM_PER_COL_OF_MATRIX) /
                         FLOAT(NUM_ELEM_PER_COL_OF_SUBMATRIX));

NUM_SUBMAT_PER_ROW  = CEIL
                        (FLOAT(NUM_ELEM_PER_ROW_OF_MATRIX) /
                         FLOAT(NUM_ELEM_PER_ROW_OF_SUBMATRIX));

ALLOCATE BLOCK1;
ALLOCATE BLOCK2;
```

Clearly, NUM_SUBMAT_PER_ROW is 1, since NUM_ELEM_PER_ROW_OF_SUBMATRIX was set to NUM_ELEM_PER_ROW_OF_MATRIX.

Column Storage

For the column storage method, the number of elements in a column of a submatrix is equal to the number of elements in a column of the original matrix. Therefore, the number of complete columns contained in a block will be dependent on the amount of memory available. Since at least two blocks must fit in memory, the number of columns contained in a block is determined by the number of complete columns that fit into half of memory.

After computing the dimensions of each block, the number of these blocks can be easily computed.

Here are the PL/I statements that reflect the discussion above.

```
TEMP1 = FLOOR (FLOAT(MEMORY_SIZE_IN_ELEM) / 2.);

TEMP2 = FLOOR (FLOAT(TEMP1) /
               FLOAT(NUM_ELEM_PER_COL_OF_MATRIX));

NUM_ELEM_PER_COL_OF_SUBMATRIX = NUM_ELEM_PER_COL_OF_MATRIX;
NUM_ELEM_PER_ROW_OF_SUBMATRIX = TEMP2;

NUM_SUBMAT_PER_COL  = CEIL
                       (FLOAT(NUM_ELEM_PER_COL_OF_MATRIX) /
                        FLOAT(NUM_ELEM_PER_COL_OF_SUBMATRIX));

NUM_SUBMAT_PER_ROW  = CEIL
                       (FLOAT(NUM_ELEM_PER_ROW_OF_MATRIX) /
                        FLOAT(NUM_ELEM_PER_ROW_OF_SUBMATRIX));

ALLOCATE BLOCK1;
ALLOCATE BLOCK2;
```

Clearly, NUM_SUBMAT_PER_COL is 1, since NUM_ELEM_PER_COL_OF_SUBMATRIX was set to NUM_ELEM_PER_COL_OF_MATRIX.

Submatrix Storage

For the submatrix storage method, we shall begin by considering the case where all the submatrices are square. Since at least three blocks must reside in memory at once, the number of elements in a block depends on the number of elements that fit into one-third of memory. Thus the maximum dimension of a block is the square root of one-third of memory.

After determining the dimensions of each block, the number of these blocks can be easily determined.

Here are the PL/I statements for square blocks.

```
TEMP1 = FLOOR(MEMORY_SIZE_IN_ELEM / 3);
TEMP2 = FLOOR(SQRT (TEMP1));

NUM_ELEM_PER_ROW_OF_SUBMATRIX = TEMP2;
NUM_ELEM_PER_COL_OF_SUBMATRIX = TEMP2;

NUM_SUBMAT_PER_COL   = CEIL
                        (FLOAT(NUM_ELEM_PER_COL_OF_MATRIX) /
                         FLOAT(NUM_ELEM_PER_COL_OF_SUBMATRIX));

NUM_SUBMAT_PER_ROW   = CEIL
                        (FLOAT(NUM_ELEM_PER_ROW_OF_MATRIX) /
                         FLOAT(NUM_ELEM_PER_ROW_OF_SUBMATRIX));

ALLOCATE BLOCK1;
ALLOCATE BLOCK2;
ALLOCATE BLOCK3;
```

Chapter 4

Decomposition Algorithms Without Interchanges

The initial analysis will focus on decomposing the matrix A into LU factors without interchanges. In the following sections, algorithms are presented to decompose A into LU factors. A brief table outlining the numbering of the algorithms is presented below:

algorithm	storage method	comments
ROW#1	full row	McKellar and Coffman (1969).
ROW#2	full row	a modification of ROW#1.
ROW#3	full row	a modification of ROW#2.
COL#1	full column	Nugent and DuCroz (1981b).
THREESUB#1	submatrix	McKellar and Coffman (1969).
THREESUB#2	submatrix	a modification of THREESUB#1.
THREESUB#3	submatrix	McKellar and Coffman (1969). a modification of THREESUB#1.
THREESUB#4	submatrix	original work.
THREESUB#5	submatrix	a modification of THREESUB#4
THREESUB#6	submatrix	original work.
THREESUB#7	submatrix	a modification of THREESUB#6
THREESUB#8	submatrix	original work.
THREESUB#9	submatrix	a modification of THREESUB#8
THREESUB#10	submatrix	a modification of THREESUB#9

Algorithm ROW#1

Conceptually, the first algorithm to be presented should be McKellar and Coffman (1969). McKellar presented this algorithm in a discussion regarding the reduction of page faults in matrix operations. The algorithm has been rewritten to do explicit reads and writes using the row storage method described in diagram 1. It can be modified to use the column storage method by changing the loop limits.

Assume there are m complete rows per block and n blocks per matrix so the matrix has (m * n) variables in (m * n) equations. The first step of the first pass performs forward elimination of the rows of block #1 for variables 1 thru m. The second step of the first pass involves the elimination of the first m variables from the remaining n-1 blocks. The first step of the second pass performs forward elimination for variables m+1 thru 2m. The second step of the second pass eliminates the second m variables from the remaining n-2 blocks. The n th pass eliminates the last m variables from the n th block. There are n passes, each pass eliminating m variables.

The PL/I main routine for this algorithm is:

```
DO I = 1 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK  (I,BLOCK1);
   CALL SELF_REDUCE (BLOCK1);
   CALL WRITE_BLOCK (I,BLOCK1);

   DO J = I+1 TO NUM_SUBMAT_PER_COL;

      CALL READ_BLOCK  (J,BLOCK2);
      CALL DUAL_REDUCE (BLOCK2,BLOCK1);
      CALL WRITE_BLOCK (J,BLOCK2);

   END;

END;
```

READ_BLOCK (I,BLOCK) is a subroutine which does the actual read of block I from the disk into a memory block. WRITE_BLOCK (I,BLOCK) is a subroutine which does the actual rewrite of block I to disk from memory.

SELF_REDUCE (BLOCK) is a subroutine which eliminates m variables from a block. DUAL_REDUCE (BLOCK2,BLOCK1) is a subroutine which eliminates m variables from BLOCK2 using the values in BLOCK1. The variables eliminated in both subroutines are (((I-1) * m) + 1) thru (I * m).

In analyzing Algorithm ROW#1, it is clear that after reducing the first block, the remaining (N-1) blocks are reduced using the values in the first block. After reducing the second block, blocks 3 thru N are reduced using the values in block 2. After reducing block (N-1), only block N is reduced using the values in block (N-1). After reducing block N, no other blocks remain to be reduced. Therefore, in the Nth pass, only one block is read and rewritten. In the (N-1)th pass, only two blocks (N and (N-1)) are read and rewritten. In the first pass, N blocks are read and rewritten.

Hence, the number of blocks read in and rewritten by Algorithm ROW#1 is:

$$\sum_{I=1}^{N} I = \frac{N ** 2 + N}{2}$$

where N = NUM_SUBMAT_PER_COL.

Algorithm ROW#2

After examining the data flow (i.e., the pattern of reads and writes) of Algorithm ROW#1, it is clear that reversing the inner loop to proceed from NUM_SUBMAT_PER_ROW to I+1 instead of proceeding from I+1 to NUM_SUBMAT_PER_ROW will result in the next block to be reduced being in memory when the outer loop begins. This will result in a savings of NUM_SUBMAT_PER_ROW - 1 reads. This insight is mentioned in McKellar and Coffman (1969). The PL/I program for this algorithm is presented below.

```
CALL READ_BLOCK (1,BLOCK1);

DO I = 1 TO NUM_SUBMAT_PER_ROW;

   CALL SELF_REDUCE (BLOCK1);
   CALL WRITE_BLOCK (I,BLOCK1);

   DO J = NUM_SUBMAT_PER_ROW TO I+1 BY -1;

      CALL READ_BLOCK  (J,BLOCK2);
      CALL DUAL_REDUCE (BLOCK2,BLOCK1);
      CALL WRITE_BLOCK (J,BLOCK2);

   END;

END;
```

The same READ_BLOCK, WRITE_BLOCK, SELF_REDUCE and DUAL_REDUCE subroutines used in Algorithm ROW#1 are used in Algorithm ROW#2.

The number of writes remains the same as in Algorithm ROW#1 which was:

$$\frac{N ** 2 + N}{2}$$

where N = NUM_SUBMAT_PER_COL.

However, there are (N-1) less reads being done by Algorithm ROW#2 so the number of reads becomes

$$\frac{N ** 2 + N}{2} - (N - 1)$$

which is equal to

$$\frac{N ** 2 - N + 2}{2}$$

where N = NUM_SUBMAT_PER_COL.

Algorithm ROW#3

Let us now try to improve upon Algorithm ROW#2 by examining the pattern of reads and writes. Assume NUM_SUBMAT_PER_ROW is equal to 4. When I=1, then the inner loop varies J from 4 to 2 by -1. In the inner loop, BLOCK #2 is reduced by BLOCK #1 and rewritten to disk. Immediately thereafter in the outer loop, BLOCK #2 is reduced by itself and rewritten to disk. Clearly, the last write of BLOCK #2 to disk in the inner loop serves no purpose.

We can modify Algorithm ROW#2 and eliminate the extraneous writes from the inner loop. The extraneous write occurs when the inner loop is exited while the next block to be self reduced is in memory. So we must terminate the inner loop one pass earlier and handle as a special case the block to be saved for the next iteration of the outer loop. By doing so, the number of writes can be made to equal the number of reads.

The improved algorithm is presented below:

```
CALL READ_BLOCK (1,BLOCK1);

DO I = 1 TO NUM_SUBMAT_PER_ROW - 1;

   CALL SELF_REDUCE (BLOCK1);
   CALL WRITE_BLOCK (I,BLOCK1);

   DO J = NUM_SUBMAT_PER_ROW TO I+2 BY -1;

      CALL READ_BLOCK (J,BLOCK2);
      CALL DUAL_REDUCE (BLOCK2,BLOCK1);
      CALL WRITE_BLOCK (J,BLOCK2);

   END;

   CALL READ_BLOCK (J,BLOCK2);
   CALL DUAL_REDUCE (BLOCK2,BLOCK1);

END;

CALL SELF_REDUCE (BLOCK2);
CALL WRITE_BLOCK (I,BLOCK2);
```

The same READ_BLOCK, WRITE_BLOCK, SELF_REDUCE and DUAL_REDUCE subroutines used in Algorithm ROW#1 are used in Algorithm ROW#3.

The number of writes is equal to the number of reads, which is

$$\frac{N ** 2 - N + 2}{2}$$

where N = NUM_SUBMAT_PER_COL.

As mentioned previously, algorithms ROW#1 thru ROW#3 can be modified for the column storage method. The only changes will be in the loop parameters in the main routine and in the subroutines SELF_REDUCE and DUAL_REDUCE.

Algorithm COL#1

The data flow for algorithms ROW#1, ROW#2, and ROW#3 can be summarized as:

reduce block I
reduce the remaining blocks using the values in block I

Nugent and DuCroz (1981b), in a paper on the reduction of page faults in a virtual memory computer, suggest a different approach. For each block, apply the reductions from the previous blocks and then reduce that block. Following this idea, after reducing a block by itself, the algorithm does not have to reduce the rest of the matrix. This saves the complete rewriting of the remaining matrix in each pass which is done in algorithms ROW#1, ROW#2 and ROW#3.

This means that a block is rewritten to disk only if it has been completely reduced. A block will be read in many times. Before reducing block I, reductions from blocks 1 thru (I-1) must be applied. This may necessitate (I-1) disk reads.

Of course, provisions are made for the first block, since there are no reductions from previous blocks.

The PL/I program for this algorithm is presented below using the full column storage method.

```
DO I = 1 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK (I,BLOCK1);

   DO J = 1 TO I-1;

      CALL READ_BLOCK  (J,BLOCK2);
      CALL DUAL_REDUCE (BLOCK1,BLOCK2);

   END;

   CALL SELF_REDUCE (BLOCK1);
   CALL WRITE_BLOCK (I,BLOCK1);

END;
```

READ_BLOCK (I,BLOCK) is a subroutine which does the actual read of block I from the disk into a memory block. WRITE_BLOCK (I,BLOCK) is a subroutine which does the actual rewrite of block I to disk from memory.

SELF_REDUCE (BLOCK) is a subroutine which eliminates m variables from a block. DUAL_REDUCE (BLOCK1,BLOCK2) is a subroutine which eliminates m variables from BLOCK1 using the values in BLOCK2. The variables eliminated in both subroutines are (((i-1) * m)+ 1) thru (I * m).

The number of writes is simply the number of blocks or the value in NUM_SUBMAT_PER_COL. Each block is written only once.

Block 1 is read once, block 2 is read twice, block 3 is read thrice, so the number of reads is the sum of I from 1 to the number of blocks or

$$\frac{N ** 2 + N}{2}$$

where N = NUM_SUBMAT_PER_COL.

The algorithm above can be modified for the row storage method. The only changes will be in the loop parameters of the main routine and in subroutines SELF_REDJCE and DUAL_REDUCE.

Comparing Algorithms COL#1 and ROW#3

In algorithm ROW#3, the number of writes is equal to the number of reads, which is

$$\frac{N ** 2 - N + 2}{2}$$

where N = NUM_SUBMAT_PER_COL.

So, the total number of disk I/Os done by algorithm ROW#3 is

$$N ** 2 - N + 2$$

In algorithm COL#1, the number of writes is N and the number of reads is

$$\frac{N ** 2 + N}{2}$$

So, the total number of disk I/Os done by algorithm COL#1 is

$$\frac{N ** 2 + N}{2} + N$$

Clearly as the number of blocks increases, algorithm COL#1 is cheaper. Although there are (N - 1) more reads in algorithm COL#1 than in ROW#3, algorithm COL#1 does only N writes while algorithm ROW#3 does approximately (N ** 2)/2 writes.

Algorithm THREESUB#1

McKellar and Coffman (1969) present an algorithm which reduces the number of page faults in a virtual memory machine assuming that the original matrix A is partitioned into submatrix blocks.

Assume A is partitioned into an m by m matrix of blocks. The algorithm reduces BLOCK (I,I), then reduces the remaining blocks in the Ith column and in the Ith row, and finally it reduces the blocks in the submatrix bounded by row I+1 and column I+1.

Rewriting McKellar's algorithm to use explicit reads and writes, the PL/I main routine is:

```
DO I = 1 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK  (I,I,BLOCK1);
   CALL AUTO_REDUCE (BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK  (J,I,BLOCK2);
      CALL VERT_REDUCE (BLOCK2,BLOCK1);
      CALL WRITE_BLOCK (J,I,BLOCK2);

      CALL READ_BLOCK  (I,J,BLOCK2);
      CALL HORIZ_REDUCE(BLOCK2,BLOCK1);
      CALL WRITE_BLOCK (I,J,BLOCK2);

   END;


   DO J = I+1 TO NUM_SUBMAT_PER_COL;

      CALL READ_BLOCK (I,J,BLOCK1);

      DO K = I+1 TO NUM_SUBMAT_PER_ROW;

         CALL READ_BLOCK  (K,J,BLOCK3);
         CALL READ_BLOCK  (K,I,BLOCK2);
         CALL TRI_REDUCE  (BLOCK3,BLOCK2,BLOCK1);
         CALL WRITE_BLOCK (K,J,BLOCK3);

      END;

   END;

END;
```

READ_BLOCK (I,J,BLOCK) is a subroutine which does the actual read of BLOCK (I,J) from the disk to memory. WRITE_BLOCK (I,J,BLOCK) is a subroutine which does the actual write of BLOCK (I,J) from memory to disk.

Assume BLOCK2 is the block in memory which contains the values of the (J,I)th block (J>I) of the matrix and BLOCK1 is the block in memory which contains the values of the (I,I)th block of the matrix. VERT_REDUCE (BLOCK2, BLOCK1) is a subroutine which reduces BLOCK2 using the values in BLOCK1. BLOCK (I,I) is already in LU format. BLOCK (J,I) will be transformed into a complete block of L factors, since (J,I) is below the diagonal block (I,I).

Assume BLOCK2 is the block in memory which contains the values of the (I,J)th block (J>I) of the matrix and BLOCK1 is the block in memory which contains the values of the (I,I)th block of the matrix. HORIZ_REDUCE (BLOCK2, BLOCK1) is a subroutine which reduces BLOCK2 using the values in BLOCK1. BLOCK (I,I) is already in LU format. BLOCK (I,J) will be transformed into a complete block of U factors, since (I,J) is to the right of the diagonal block (I,I).

TRI_REDUCE (BLOCK3,BLOCK2,BLOCK1) is a subroutine which reduces disk BLOCK (K,J) using the L values stored in BLOCK (K,I) and the U values contained in BLOCK (I,J). TRI_REDUCE assumes that block (K,J) of the matrix A has been read into memory BLOCK3, block (K,I) of the matrix A has been read into BLOCK2, and block (I,J) of the matrix A has been read into BLOCK1.

Basically, the algorithm can be analyzed by examining the Ith pass of the algorithm as follows:

> Reduce block (I,I). Hence, each block on the diagonal is read and rewritten at the start of the Ith pass.
>
> Reduce the remaining blocks on the Ith row. They are the blocks (I,I+1) thru (I,N). Therefore, these blocks are read and rewritten in order to be reduced in the Ith pass. These blocks will also be read once in order to reduce other blocks beneath them during the Ith pass. As an example, block (1,3) is read when it is reduced by block (1,1) and again when it is used to reduce blocks (2,3), (3,3), and (4,3) thru (N,3).
>
> Reduce the remaining blocks in the Ith column. They are blocks (I+1,I) thru (N,I). Therefore, these blocks are read and rewritten in order to be reduced in the Ith pass. These blocks will also be read several times later in this pass in order to reduce other blocks residing on the same row.
>
> Reduce the remaining submatrix bounded by the (I+1)th row and column. These are the blocks (I+1,I+1) thru (I+1,N), (I+1,I+1) thru (N,I+1), (I+1,N) thru (N,N) and (N,I+1) thru (N,N). Therefore, these blocks will be read and rewritten during the Ith pass. As noted previously, in order to reduce these blocks, blocks from the Ith row and Ith column must be read into memory. The algorithm above reads in a block from the Ith row and Jth column and then proceeds to reduce the blocks in the Jth column from row I+1 thru row N using values read in from the Ith column.

Hence, the above-diagonal blocks in the Ith row are read twice (once directly and once for the remaining submatrix); while, the blocks below the diagonal in the Ith column are read (N-I) times for the remaining submatrix and once directly.

Therefore, for N passes we have:

> Upper Triangle: Any given (I,J) is read and rewritten once in passes 1, 2, 3 thru I-1. During the Ith pass, this block is read twice and rewritten once. Thus, each of these blocks is read I+1 times and rewritten I times.
>
> Diagonal: Any given (I,I) is read and rewritten once during passes 1 thru I-1 and once during pass I, for a total of I reads and writes.
>
> Lower Triangle: Any given (I,J) is read and rewritten once for passes 1 thru J-1. During pass J, (I,J) is read and rewritten after being reduced by the diagonal block. So block (I,J) is written J times. Block (I,J) is also read N-I times to help reduce the remaining submatrix. In total, block (I,J) is read (I-1) + 1 + (N-I) times, or N times.

Using the general idea of the algorithm, we can derive a formula for the number of writes.

In the first pass, block (1,1) is used to reduce the entire first row and first column of the matrix. Then, the remaining blocks are reduced using the values in the first row and first column of blocks. Therefore, in the first pass the entire N by N matrix is rewritten. Row 1 and column 1 are never again used.

In the second pass, block (2,2) is used to reduce the remaining N-1 blocks of the second row and the remaining N-1 blocks of the second column. Using the values of the second row and column, the remaining (N-2)*(N-2) blocks of the matrix are reduced and rewritten. Therefore, in the second pass (N-1)*(N-1) blocks are rewritten.

In the Ith pass, each block (K,L) with K>=I and L>=I is written once, so the Ith pass does (N+1-I)**2 writes.

Therefore, the formula for the number of writes is:

$$\sum_{I=1}^{N} (N+1-I)^{2} = \sum_{I=1}^{N} I^{2}$$

which is equal to

$$\frac{2 N^{3} + 3 N^{2} + N}{6}$$

Again, by using the above analysis, we can derive a formula for the number of reads:

Upper Triangle: Since each block (I,J) above the diagonal in the Ith row is read once for passes 1 thru I-1 and twice in pass I, it is read I+1 times. Noting that there are (N-I) blocks above the diagonal in the Ith row, we have

$$\sum_{I=1}^{N-1} (I+1) * (N-I)$$

Diagonal: Since the diagonal contains N blocks and each block (I,I) is read I times, we have

$$\sum_{I=1}^{N} I$$

Lower Triangle: Since each block (I,J) is read N times and there are (N*(N-1))/2 blocks below the diagonal, we do

$$N * \frac{N * (N-1)}{2} = \frac{N ** 3 - N ** 2}{2}$$

reads.

After combining the summations and simplifying, we have

$$\frac{4 * N ** 3 + 3 * N ** 2 - N}{6}$$

To illustrate the counts presented above, we consider a matrix which has been divided in a 10 by 10 matrix of blocks. The read and write counts by block for its reduction into LU format are presented below.

WRITE COUNTS BY BLOCK

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

READ COUNTS BY BLOCK

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

Algorithm THREESUB#2

After examining the data flow (i.e., the pattern of reads and writes) of Algorithm THREESUB#1, it is clear that this algorithm has three parts. First, reduce block (I,I). Then, reduce the blocks on row I and column I using block (I,I). Then, reduce the remaining blocks using the blocks in the Ith row and column. Algorithm THREESUB#1 reduces the remaining blocks by having the inner loops proceed from I+1 to N.

Reversing the inner loops to proceed from N to I+1 will result in the next diagonal block to be reduced being in memory when the outer loop begins. This will result in a savings of N-1 reads. Also, we can save the rewrite of the diagonal block in the inner loop since it will be reduced and rewritten at the beginning of the next pass. Thus, we save N-1 writes.

The PL/I main routine is:

```
CALL READ_BLOCK  (1,1,BLOCK1);
CALL AUTO_REDUCE (BLOCK1);
CALL WRITE_BLOCK (1,1,BLOCK1);

DO I = 1 TO NUM_SUBMAT_PER_COL-1;

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK  (J,I,BLOCK2);
      CALL VERT_REDUCE (BLOCK2,BLOCK1);
      CALL WRITE_BLOCK (J,I,BLOCK2);

      CALL READ_BLOCK  (I,J,BLOCK2);
      CALL HORIZ_REDUCE(BLOCK2,BLOCK1);
      CALL WRITE_BLOCK (I,J,BLOCK2);

   END;

   DO J = NUM_SUBMAT_PER_COL TO I+1 BY -1;

      CALL READ_BLOCK (I,J,BLOCK2);

      DO K = NUM_SUBMAT_PER_ROW TO I+1 BY -1;

         CALL READ_BLOCK  (K,J,BLOCK1);
         CALL READ_BLOCK  (K,I,BLOCK3);
         CALL TRI_REDUCE  (BLOCK1,BLOCK3,BLOCK2);
         IF ^ (K = I+1 & J = I+1)
            THEN CALL WRITE_BLOCK (K,J,BLOCK1);

      END;

   END;

   CALL AUTO_REDUCE (BLOCK1);
   CALL WRITE_BLOCK (I+1,I+1,BLOCK1);

END;
```

READ_BLOCK, VERT_REDUCE, HORIZ_REDUCE, WRITE_BLOCK, and TRI_REDUCE are the same subprograms as in algorithm THREESUB#1.

Therefore, the read formula for algorithm THREESUB#2 is derived by subtracting N-1 from algorithm THREESUB#1's read formula. This yields

$$\frac{4 * N ** 3 + 3 * N ** 2 - N}{6} - (N-1)$$

which is equal to

$$\frac{4 * N ** 3 + 3 * N ** 2 - 7 * N + 6}{6}$$

Similiarly, the write formula for algorithm THREESUB#2 is derived by subtracting N-1 from algorithm THREESUB#1's write formula. This yields

$$\frac{2 * N ** 3 + 3 * N ** 2 + N}{6} - (N-1)$$

which is equal to

$$\frac{2 * N ** 3 + 3 * N ** 2 - 5 * N + 6}{6}$$

By comparing the read/write counts of THREESUB#2 with the counts from THREESUB#1, we can immediately see the savings along the diagonal of N-1 reads and N-1 writes.

ALGORITHM THREESUB#2
WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	1	2	2	2	2	2	2	2	2
1	2	2	3	3	3	3	3	3	3
1	2	3	3	4	4	4	4	4	4
1	2	3	4	4	5	5	5	5	5
1	2	3	4	5	5	6	6	6	6
1	2	3	4	5	6	6	7	7	7
1	2	3	4	5	6	7	7	8	8
1	2	3	4	5	6	7	8	8	9
1	2	3	4	5	6	7	8	9	9

ALGORITHM THREESUB#2
READ COUNTS BY BLOCK

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

ALGORITHM THREESUB#1
WRITE COUNTS BY BLOCK

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

ALGORITHM THREESUB#1
READ COUNTS BY BLOCK

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

Algorithm THREESUB#3

After analyzing algorithm THREESUB#1, McKellar and Coffman (1969) suggest several revisions. Even though some of their suggestions deal with an optimal demand paging scheme with foreknowledge of future reads preventing certain key blocks from being swapped out, algorithm THREESUB#1 can be improved by using some of this foreknowledge by the loops and loop parameters.

Reviewing algorithm THREESUB#1, we observe the first inner loop reduces row I and column I. At the end of the loop, blocks (I,N) and (N,I) are in memory (where N is equal to NUM_SUBMAT_PER_ROW). The reduction of block (N,N) could be done at this point. By reading block (N-1,I), we can reduce block (N-1,N) since block (I,N) is already in memory. In fact, we can reduce the Nth column from N to I+1 by just reading the factors stored in the blocks of the Ith column. At this point, blocks(I+1,I) and (I,N) are in memory. By reading in block (I,N-1), block (I+1,N-1) can be reduced. In fact, the (N-1)th column can be reduced by proceeding from I+1 to N. The (N-2)th column is reduced from N to I+1. The (N-3)th column is reduced from I+1 to N. This first pass terminates when all columns l, where l > i, have been reduced.

In general, when the number of rows and columns of the remaining submatrix is even, block (I+1,I+1) is in memory at the start of the (I+1)st pass. Since block (I+1,I+1) is the next diagonal block, this saves one read and write per pass.

When the number of rows and columns of the remaining submatrix is odd, reducing the blocks in a slightly different pattern assures block (I+1,I+1) being in memory at the start of the (I+1)st pass. The Ith row is reduced from column (I+1) thru column N. Then, the Ith column is reduced from row N thru row (I+1). At this time, blocks (I,N) and (I+1,I) are in memory, so block (I+1,N) can be reduced. Column N is then reduced frow row I+1 thru row N. Column N-1 is reduced from row N to row I+1. The next column is reduced from row I+1 thru row N. The following column is reduced from row N thru row I+1. Following this alternating pattern, column (I+1) is reduced from row N thru (I+1). This results in block (I+1,I+1) being in memory at the start of pass I+1. This saves one read and write.

Hence, algorithm THREESUB#3 is really composed of two distinct looping patterns. At the beginning of each pass, the algoritm examines the dimensions of the remaining submatrix. If the dimensions are even, the algorithm follows a certain looping pattern. Otherwise, the algorithm follows a different looping pattern.

In order to illustrate the above loop patterns, assume the division of matrix A into a six by six submatrix of blocks. Pass 1 of the algorithm will proceed as indicated in diagram 4. Pass 2 of the algorithm will proceed as indicated in diagram 5 on the remaining five by five submatrix. Passes 3, 4, 5, and 6 are illustrated in diagrams 6, 7, 8, and 9, respectively. The number inside each block represents the sequence of block reductions.

1	2	4	6	8	10
3	36	27	26	17	16
5	35	28	25	18	15
7	34	29	24	19	14
9	33	30	23	20	13
11	32	31	22	21	12

diagram 4.

	1	2	3	4	5
	9	25	18	17	10
	8	24	19	16	11
	7	23	20	15	12
	6	22	21	14	13

diagram 5.

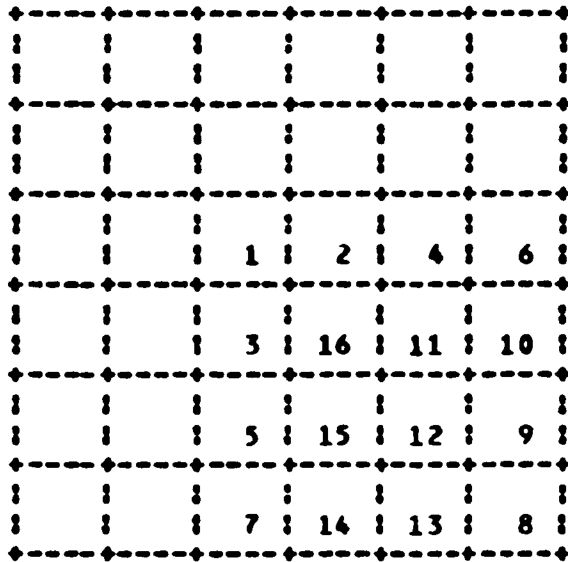


diagram 6.

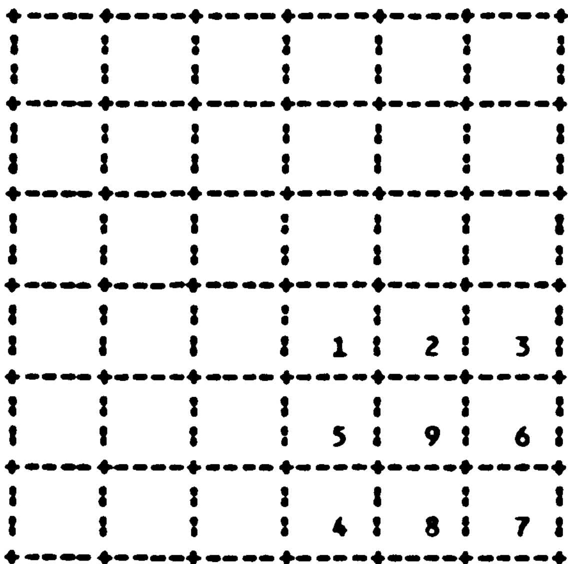


diagram 7.

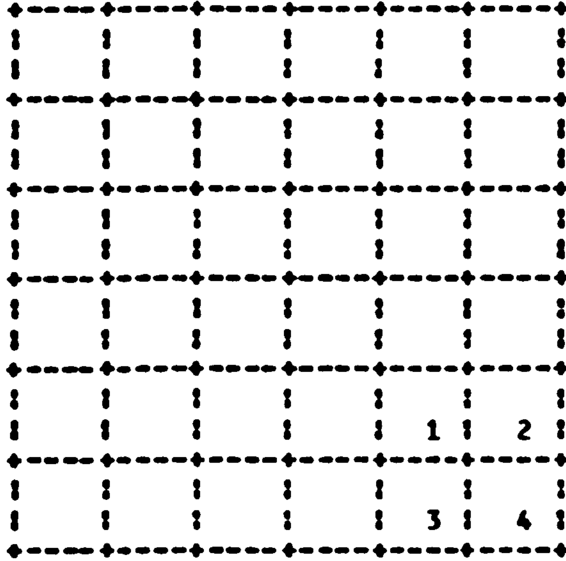


diagram 8.

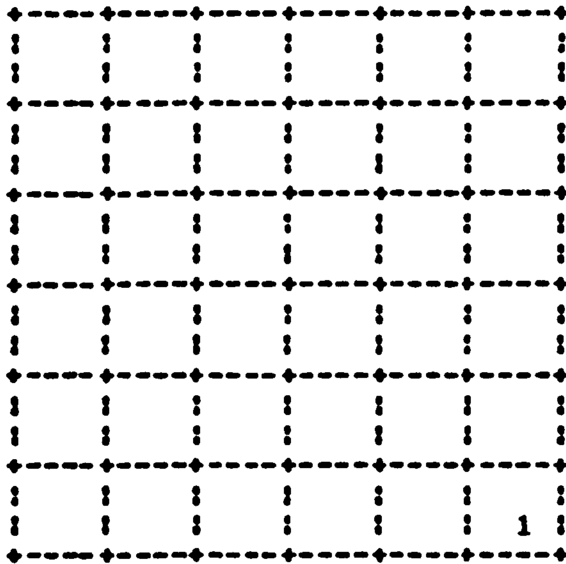


diagram 9.

Modifying algorithm THREESUB#1 to reflect these ideas, we have:

```
CALL READ_BLOCK (1,1,BLOCK1);

DO I = 1 TO NUM_SUBMAT_PER_COL - 1;

   CALL AUTO_REDUCE (BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);

   FLOAT1 = NUM_SUBMAT_PER_COL - I;
   FLOAT2 = FLOOR(FLOAT1 / 2.);
   FLOAT3 = FLOAT2 * 2;

   IF FLOAT1 ^= FLOAT3
      THEN DO;

            DO J = I+1 TO NUM_SUBMAT_PER_ROW;

               CALL READ_BLOCK (J,I,BLOCK2);
               CALL VERT_REDUCE (BLOCK2,BLOCK1);
               CALL WRITE_BLOCK (J,I,BLOCK2);

               CALL READ_BLOCK (I,J,BLOCK3);
               CALL HORIZ_REDUCE (BLOCK3,BLOCK1);
               CALL WRITE_BLOCK (I,J,BLOCK3);

            END;

            L = NUM_SUBMAT_PER_COL;

            DO WHILE (L > I) ;

               DO K = NUM_SUBMAT_PER_COL TO I+2 BY -1;

                  CALL READ_BLOCK (K,L,BLOCK1);
                  CALL TRI_REDUCE (BLOCK1,BLOCK2,BLOCK3);
                  CALL WRITE_BLOCK (K,L,BLOCK1);

                  CALL READ_BLOCK (K-1,I,BLOCK2);

               END;

               CALL READ_BLOCK (K,L,BLOCK1);
               CALL TRI_REDUCE (BLOCK1,BLOCK2,BLOCK3);
```

```
L = L - 1;
IF L > I
   THEN
     DO;

         CALL WRITE_BLOCK(K,L+1,BLOCK1);
         CALL READ_BLOCK (I,L,BLOCK3);

         DO K = I+1 TO NUM_SUBMAT_PER_COL-1;

            CALL READ_BLOCK (K,L,BLOCK1);
            CALL TRI_REDUCE
                 (BLOCK1,BLOCK2,BLOCK3);
            CALL WRITE_BLOCK(K,L,BLOCK1);

            CALL READ_BLOCK(K+1,I,BLOCK2);

         END;

         CALL READ_BLOCK (K,L,BLOCK1);
         CALL TRI_REDUCE
              (BLOCK1,BLOCK2,BLOCK3);

         L = L - 1;
         IF L > I
            THEN DO;
                   CALL WRITE_BLOCK
                        (K,L+1,BLOCK1);
                   CALL READ_BLOCK
                        (I,L,BLOCK3);
                 END;

     END;
END;

END;
```

```
ELSE DO;

        DO J = I+1 TO NUM_SUBMAT_PER_ROW;

           CALL READ_BLOCK (I,J,BLOCK3);
           CALL HORIZ_REDUCE(BLOCK3,BLOCK1);
           CALL WRITE_BLOCK (I,J,BLOCK3);

        END;

        DO J = NUM_SUBMAT_PER_ROW TO I+1 BY -1;

           CALL READ_BLOCK (J,I,BLOCK2);
           CALL VERT_REDUCE (BLOCK2,BLOCK1);
           CALL WRITE_BLOCK (J,I,BLOCK2);

        END;


        L = NUM_SUBMAT_PER_COL;

        DO WHILE (L > I) ;

           DO K = I+1 TO NUM_SUBMAT_PER_CO_-1;

              CALL READ_BLOCK (K,L,BLOCK1);
              CALL TRI_REDUCE(BLOCK1,BLOCK2,BLOCK3);
              CALL WRITE_BLOCK(K,L,BLOCK1);

              CALL READ_BLOCK(K+1,I,BLOCK2);

           END;

           CALL READ_BLOCK (K,L,BLOCK1);
           CALL TRI_REDUCE (BLOCK1,BLOCK2,BLOCK3);
```

```
                    L = L - 1;
                    IF L > I
                       THEN
                         DO;

                           CALL READ_BLOCK (I,L,BLOCK3);
                           CALL WRITE_BLOCK(K,L+1,BLOCK1);

                           DO K = NUM_SUBMAT_PER_CO_
                                                 TO  I+2 BY -1;

                              CALL READ_BLOCK (K,L,BLOCK1);
                              CALL TRI_REDUCE
                                   (BLOCK1,BLOCK2,BLOCK3);
                              CALL WRITE_BLOCK (K,L,BLOCK1);

                              CALL READ_BLOCK (K-1,I,BLOCK2);

                           END;

                           CALL READ_BLOCK (K,L,BLOCK1);
                           CALL TRI_REDUCE
                                (BLOCK1,BLOCK2,BLOCK3);

                           L = L - 1;
                           IF L > I
                              THEN DO;

                                     CALL WRITE_BLOCK
                                          (K,L+1,BLOCK1);
                                     CALL READ_BLOCK
                                          (I,L,BLOCK3);

                                   END;

                         END;
                 END;

              END;

  END;

  CALL AUTO_REDUCE (BLOCK1);
  CALL WRITE_BLOCK (I,I,BLOCK1);
```

READ_BLOCK, VERT_REDUCE, HORIZ_REDUCE, WRITE_BLOCK, and TRI_REDUCE are the same subprograms as in algorithm THREESUB#1.

A formula for the number of writes can be determined from the algorithm. Each non-diagonal block on column I and row I is written I times. A block is rewritten I-1 times after being reduced in passes 1 thru I-1 and after its final reduction in pass I. After pass I, a block above the present diagonal block is never reduced again. The diagonal blocks are rewritten one less time. That is, block (I,I) is rewritten I-1 times (except for block (1,1)). This is understandable since the algorithm is designed to finish a pass at the proper diagonal block and not rewrite it. There is no purpose in rewriting block (I,I) at the end of pass I-1, since block (I,I) will be the very next block reduced and rewritten in pass I. Since BLOCK (I,I) was not rewritten after being reduced in the previous pass, this algorithm saves N-1 writes over algorithm THREESUB#1.

Hence, the write count is N-1 less than that for algorithm THREESUB#1, so we have

$$\frac{2 * N ** 3 + 3 * N ** 2 + N}{6} - (N-1)$$

which may be rewritten as

$$\frac{2 * N ** 3 + 3 * N ** 2 - 5 * N + 6}{6}$$

where N = NUM_SUBMAT_PER_ROW.

NOTE: Algorithms THREESUB#3 and THREESUB#2 use the same number of writes.

The purpose of the algorithm is basically to save one read per column and to have the block needed for the next pass in memory at the end of the previous pass.

The formula for the number of reads can be determined by counting the number of reads in algorithm THREESUB#1 that will be saved by algorithm THREESUB#3. After each pass, the next diagonal block is in memory. This saves one read per pass, for a total of N-1 reads. During each pass, algorithm THREESUB#1 did two reads in order to reduce each block in the remaining submatrix bounded by row I+1 and column I+1. Algorithm THREESUB#3 saves one read per column of the remaining submatrix by the alternating column sweeps. Therefore, algorithm THREESUB#3 saves N-1 reads in pass 1, N-2 reads in pass 2, and 1 read in pass N-1, for a total of

$$\sum_{I=1}^{N-1} I = \frac{N * (N - 1)}{2}$$

This algorithm also saves one additional read per pass since the reduction of the first block in the remaining submatrix (either block (N,N) or (I+1,N)) does not require any reads. Thus, it saves another N-1 reads. This fact is graphically displayed by examining the subdiagonal and the last row of the read count table presented below. As you can see, the counts are approximately one-half the counts from algorithm THREESUB#1.

Thus, algorithm THREESUB#3 saves

$$2 * (N - 1) + \frac{N * (N - 1)}{2}$$

of the reads used by algorithm THREESUB#1, so the number of reads is

$$\frac{4 * N**3 + 3 * N**2 - N}{6} - 2 * (N-1) - \frac{N * (N-1)}{2}$$

which can be rewritten as

$$\frac{4 * N ** 3 - 10 * N + 12}{6}$$

where N = NUM_SUBMAT_PER_ROW.

In order to understand the differences between THREESUB#1 and THREESU3#3, a comparison of the read and write counts is in order. As an example, the matrix A has been divided in a 10 by 10 matrix of blocks. The read and write counts are presented below.

ALGORITHM THREESUB#3
WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	1	2	2	2	2	2	2	2	2
1	2	2	3	3	3	3	3	3	3
1	2	3	3	4	4	4	4	4	4
1	2	3	4	4	5	5	5	5	5
1	2	3	4	5	5	6	6	6	6
1	2	3	4	5	6	6	7	7	7
1	2	3	4	5	6	7	7	8	8
1	2	3	4	5	6	7	8	8	9
1	2	3	4	5	6	7	8	9	9

ALGORITHM THREESUB#3
READ COUNTS BY B_OCK

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

ALGORITHM THREESUB#1
WRITE COUNTS BY BLOCK

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

ALGORITHM THREESU3#1
READ COUNTS BY 3LOCK

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

As one can see from the write counts, there is a savings of N-1 writes along the diagonal. This is directly attributable to looping patterns of algorithm THREESUB#3, which are designed to finish a pass at the proper diagonal block and not rewrite it.

From the reads counts, we see the savings of N-1 reads along the diagonal.

We also note the read counts along the bottom row and the subdiagonal are one-half those of THREESUB#1.

Algorithm THREESUB#4

Algorithms THREESUB#1, THREESUB#2, and THREESUB#3 reduced BLOCK (I,I) and then reduced the remaining blocks using the factors from BLOCK (I,I). Algorithm THREESUB#4 will examine the idea of only reducing a column of blocks when one of its members is to be reduced.

After reducing BLOCK (1,1), BLOCKS (2,1) thru BLOCK (N,1) are reduced using the factors in BLOCK (1,1). This is the entire first pass.

Before reducing BLOCK (2,2), the factors stored in the blocks of column 1 must be applied. So BLOCK (1,2) is affected by BLOCK (1,1), BLOCK (2,2) by BLOCK (2,1), BLOCK (3,2) by BLOCK (3,1), BLOCK (4,2) by BLOCK (4,1), and BLOCK (N,2) is affected by BLOCK (N,1). Only then is BLOCK (2,2) reduced. Then, the remaining blocks in column 2 (BLOCKS (3,2), (4,2), thru (N,2)) are reduced using the factors in BLOCK (2,2).

Pass 3 begins with application of the L factors from the blocks in column 1 to the blocks in column 3 from rows 1 thru N. Then the L factors from column 2 are applied to column 3 (note: only rows 2 thru N need be applied). Finally, block (3,3) is reduced and blocks (4,3) thru (N,3) are reduced using the values in block (3,3).

The Nth column of blocks is untouched until the Nth pass where the factors from columns 1 thru N-1 are applied before the reduction of BLOCK (N,N).

Contrasting algorithms THREESUB#4 and THREESUB#1, the following ideas are apparent. Pass 1 of algorithm THREESUB#1 passes thru the most blocks, while pass 1 of algorithm THREESUB#4 passes thru the least blocks. Pass N of algorithm THREESUB#1 passes thru the least blocks, while pass N of algorithm THREESUB#4 passes thru the most blocks.

One would assume that the two algorithms should require the same amount of I/O. This is in fact the case.

Here is the PL/I main routine for algorithm THREESUB#4.

```
DO I = 1 TO NUM_SUBMAT_PER_COL;

   DO J = 1 TO I-1;

      CALL READ_BLOCK (J,J,BLOCK2);
      CALL READ_BLOCK (J,I,BLOCK1);
      CALL HORIZ_REDUCE (BLOCK1,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);

      DO K = J+1 TO NUM_SUBMAT_PER_ROW;

         CALL READ_BLOCK (K,I,BLOCK3);
         CALL READ_BLOCK (K,J,BLOCK2);
         CALL TRI_REDUCE (BLOCK3,BLOCK2,BLOCK1);
         CALL WRITE_BLOCK (K,I,BLOCK3);

      END;

   END;

   CALL READ_BLOCK (I,I,BLOCK1);
   CALL AUTO_REDUCE (BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK (J,I,BLOCK2);
      CALL VERT_REDUCE (BLOCK2,BLOCK1);
      CALL WRITE_BLOCK (J,I,BLOCK2);

   END;

END;
```

READ_BLOCK, VERT_REDUCE, HORIZ_REDUCE, WRITE_BLOCK, and TRI_REDUCE are the same subprograms as in algorithm THREESUB#1.

To derive the formula for the counts, we note that a block (I,J) above the diagonal ($I<J$) is read and written once in each pass, 1, 2, 3, thru I, for a total of I reads and writes. This yields a total of

$$\sum_{K=1}^{J-1} K = \frac{J * (J - 1)}{2}$$

reads and writes for the Jth column.

For the elements above the diagonal, we have

$$\sum_{J=1}^{N} \frac{J * (J-1)}{2} = \frac{N^{3} - N}{6}$$

reads and writes.

Each block (I,J) on or below the diagonal (I >= J) is read and written J times in the Jth pass, and read once in passes J+1, J+2, thru N. Thus, it is written J times and read N times. So for the blocks on or below the diagonal, there are

$$N * \sum_{J=1}^{N} J = \frac{N ** 2 * (N + 1)}{2}$$

reads and

$$\sum_{J=1}^{N} J * (N + 1 - J) = \frac{N * (N + 1) * (N + 2)}{6}$$

writes.

The number of reads is

$$\frac{N^{**}3 - N}{6} + \frac{N^{**}3 + N^{**}2}{2}$$

which is equal to

$$\frac{4 * N^{**}3 + 3 * N^{**}2 - N}{6}$$

The number of writes is

$$\frac{N^{**}3 - N}{6} + \frac{N^{**}3 + 3 * N^{**}2 + 2 * N}{6}$$

which is equal to

$$\frac{4 * N^{**}3 + 6 * N^{**}2 + 2 * N}{12}$$

where N = NUM_SUBMAT_PER_COL.

The read and write counts for algorithm THREESUB#4 are the same as those derived for algorithm THREESUB#1.

Confirming our hypothesis regarding the equivalence of algorithm THREESUB#4 and algorithm THREESUB#1, the read and write counts by block for a ten by ten submatrix are identical to those presented in the discussion of algorithm THREESUB#1.

WRITE COUNTS BY BLOCK

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

READ COUNTS BY BLOCK

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

Algorithm THREESUB#5

The fourth pass of algorithm THREESUB#4 can be summarized as follows:

> BLOCK (1,4) is reduced by BLOCK (1,1) and then BLOCK (1,4) is retained in memory and used to reduce BLOCK (2,4), BLOCK (3,4), thru BLOCK (N,4).
>
> BLOCK (2,4) is reduced by BLOCK (2,2) and then BLOCK (2,4) is retained in memory and used in the reduction of BLOCK(3,4), BLOCK (4,4), BLOCK (5,4), thru BLOCK (N,4).
>
> BLOCK (3,4) is reduced by BLOCK (3,3) and BLOCK (3,4) is used to reduce BLOCK (4,4), BLOCK (5,4), thru BLOCK (N,4).
>
> BLOCK (4,4) is reduced by itself and is used to reduce BLOCK (5,4), BLOCK (6,4), thru BLOCK (N,4).

After examining the data flow (i.e., the pattern of reads and writes) of algorithm THREESUB#4, it is clear that reversing the innermost loop to proceed from N to I+1 will result in some substantial savings.

By reversing the direction of one loop, the following data flow can be achieved:

BLOCK (1,4) is reduced by BLOCK (1,1) and then BLOCK (1,4) is retained in memory and used to reduce BLOCK (N,4), BLOCK (N-1,4), BLOCK (N-2,4), thru BLOCK (2,4).

BLOCK (2,4) is reduced by BLOCK (2,2) and then BLOCK (2,4) is retained in memory and used in the reduction of BLOCK (N,4), BLOCK (N-1,4), BLOCK (N-2,4), thru BLOCK (3,4).

BLOCK (3,4) is reduced by BLOCK (3,3) and BLOCK (3,4) is used to reduce BLOCK (N,4), BLOCK (N-1,4), BLOCK (N-2,4), thru BLOCK (4,4).

BLOCK (4,4) is reduced by itself and is used to reduce BLOCK (5,4), BLOCK (6,4), thru BLOCK (N,4).

Note that after applying all the reductions from column 1 to the 4th column, BLOCK (2,4) is in memory and does not have to be written to disk and reread. After applying all the reductions from column 2 to the 4th column, BLOCK (3,4) is in memory and does not have to be written and reread again. BLOCK (4,4) is in memory at the end of the innermost loop and does not have to read again to commence the self-reduction loop. Hence, in the fourth pass, algorithm THREESUB#5 has saved 3 reads and 3 writes. In general, the Kth pass of algorithm THREESUB#5 does (K-1) less reads and (K-1) less writes than algorithm THREESUB#4.

Here is the PL/I main routine for algorithm THREESUB#5.

```
DO I = 1 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK (1,I,BLOCK1);

   DO J = 1 TO I-1;

      CALL READ_BLOCK (J,J,BLOCK2);

      CALL HORIZ_REDUCE (J,I,BLOCK1,J,J,BLOCK2);

      CALL WRITE_BLOCK (J,I,BLOCK1);

      DO K = NUM_SUBMAT_PER_ROW TO J+1 BY -1;

         CALL READ_BLOCK (K,I,BLOCK3);
         CALL READ_BLOCK (K,J,BLOCK2);

         CALL TRI_REDUCE
              (K,I,BLOCK3,K,J,BLOCK2,J,I,BLOCK1);

         IF K ^= J+1
            THEN CALL WRITE_BLOCK (K,I,BLOCK3);

      END;

      BLOCK1 = BLOCK3;

   END;


   CALL AUTO_REDUCE (I,I,BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);


   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK (J,I,BLOCK2);
      CALL VERT_REDUCE (J,I,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK (J,I,BLOCK2);

   END;

END;
```

Since algorithm THREESUB#5 is merely an improvement over algorithm THREESUB#4, we can derive the read and write equations for algorithm THREESUB#5 by subtracting the savings for algorithm THREESUB#5 from the equations derived for algorithm THREESUB#4.

As we have seen, algorithm THREESUB#5 saves K-1 reads and writes in the Kth column from column 2 thru N. Thus, it saves a total of

$$\sum_{K=2}^{N} (K-1) = \sum_{K=1}^{N-1} K = \frac{N * (N - 1)}{2}$$

reads and writes.

Hence, the read count is

$$\frac{4 * N ** 3 + 3 * N ** 2 - N}{6} - \frac{N * (N - 1)}{2}$$

which reduces to

$$\frac{4 * N ** 3 + 2 * N}{6}$$

The write count is

$$\frac{4 * N ** 3 + 6 * N ** 2 + 2 * N}{12} - \frac{N * (N - 1)}{2}$$

which reduces to

$$\frac{4 * N ** 3 + 8 * N}{12}$$

where N = NUM_SUBMAT_PER_COL.

By comparing the read and write counts of THREESUB#5 with those generated by THREESUB#4, we can see that THREESUB#5 saves 1 read and 1 write per block in the upper triangle (except for row 1).

ALGORITHM THREESUB#5
WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	2	2	2	2	2	2	2	2	2
1	2	3	3	3	3	3	3	3	3
1	2	3	4	4	4	4	4	4	4
1	2	3	4	5	5	5	5	5	5
1	2	3	4	5	6	6	6	6	6
1	2	3	4	5	6	7	7	7	7
1	2	3	4	5	6	7	8	8	8
1	2	3	4	5	6	7	8	9	9

ALGORITHM THREESJB#5
READ COUNTS BY BLOCK

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

ALGORITHM THREESUB#4
WRITE COUNTS BY BLOCK

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

ALGORITHM THREESJB#4
READ COUNTS BY BLOCK

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

Algorithm THREESUB#6

Algorithms THREESUB#4 and THREESUB#5 have two distinct phases:

the application to column L of the reductions made to columns 1, 2, ..., L-1

the reduction of column L

When algorithm THREESUB#4 reduced block (3,4), block (3,4) was rewritten after being reduced by block (3,1), was again rewritten after the reduction by block (3,2) and was finally rewritten after being reduced by block (3,3). If a block were only rewritten to disk after being totally reduced, this could lead to tremendous savings in total I/O. Algorithm THREESUB#6 is designed to reduce the number of writes after partial reduction.

Algorithm THREESUB#6 is similar to algorithm THREESUB#4. No block in column L is touched until the Lth pass. However, algorithm THREESUB#6 will reduce BLOCK (K,L) by blocks (K,1), (K,2), (K,3),..., (K,L-1) before rewritting block (K,L) to disk. After the factors from the preceding columns of blocks have been applied to the Lth column, BLOCK (L,L) is reduced and the remaining blocks in the Lth column (blocks (K+1,L), (K+2,L),..., (N,L)) are reduced using BLOCK (L,L).

Here is the PL/I main routine:

```
CALL READ_BLOCK (1,1,BLOCK1);
CALL AUTO_REDUCE (BLOCK1);
CALL WRITE_BLOCK (1,1,BLOCK1);


DO J = 2 TO NUM_SUBMAT_PER_ROW;

   CALL READ_BLOCK (J, 1, BLOCK2);
   CALL VERT_REDUCE (BLOCK2,BLOCK1);
   CALL WRITE_BLOCK (BLOCK2);

END;

DO I = 2 TO NUM_SUBMAT_PER_COL;

   DO J = 1 TO I-1;

      CALL READ_BLOCK (J,I,BLOCK1);

      DO K = 1 TO J-1;

         CALL READ_BLOCK (J,K,BLOCK2);
         CALL READ_BLOCK (K,I,BLOCK3);
         CALL TRI_REDUCE (BLOCK1,BLOCK2,BLOCK3);

      END;


      CALL READ_BLOCK (J,J,BLOCK2);
      CALL HORIZ_REDUCE (BLOCK1,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);

   END;
```

```
    DO J = I TO NUM_SUBMAT_PER_ROW;

       CALL READ_BLOCK (J,I,BLOCK1);

       DO K = 1 TO I-1;

          CALL READ_BLOCK (J,K,BLOCK2);
          CALL READ_BLOCK (K,I,BLOCK3);
          CALL TRI_REDUCE (BLOCK1,BLOCK2,BLOCK3);

       END;

       CALL WRITE_BLOCK (J,I,BLOCK1);

    END;


    CALL READ_BLOCK (I,I,BLOCK1);
    CALL AUTO_REDUCE (BLOCK1);
    CALL WRITE_BLOCK (I,I,BLOCK1);


    DO J = I+1 TO NUM_SUBMAT_PER_ROW;

       CALL READ_BLOCK (J,I,BLOCK2);
       CALL VERT_REDUCE (BLOCK2,BLOCK1);
       CALL WRITE_BLOCK (J,I,BLOCK2);

    END;

END;
```

The write count is simply N ** 2 plus the number of blocks that are written twice. For column J > 1, there are (N+1-J) blocks (those on or below the diagonal) that are rewritten twice, so the number of writes is

$$N^{2} + \sum_{J=2}^{N} (N + 1 - J) = N^{2} + \sum_{K=1}^{N-1} K$$

which is

$$N^{2} + \frac{N (N - 1)}{2}$$

Note: We have eliminated the N**3 term from the write formula. This is truly an impressive savings.

Consider the reads for blocks above the diagonal. A block (J,I) above the diagonal is read when it is reduced, and then reread when each block below it in column I is reduced, so it is read N+1-J times. Since each block above the diagonal in row J is read N+1-J times, and there are N-J blocks above the diagonal in row J, there are (N-J)*(N-J+1) reads for blocks above the diagonal in row J. This yields a total of

$$\sum_{J=1}^{N-1} (N-J)*(N-J+1) = \sum_{K=1}^{N-1} K*(K+1) = \frac{N**3 - N}{3}$$

reads for blocks above the diagonal.

A block (J,I) on or below the diagonal is read once when L factors to its left are used, and again when column I is reduced by block (I,I), and once for each of the (N-I) blocks to its right in the Jth row, for a total of (N-I+2) reads. (An exception is the first column, which is not reduced by any columns to its left, so each block is read N-1+1, or N times.) Thus, for the first coluan there are N**2 reads. For blocks on or below the diagonal in columns 2 thru N, there are (N-I+2) reads for each of the (N+I-1) blocks for a total of (N-I+2)*(N+I-1) reads.

The total number of reads for blocks on or below the diagonal is

$$N^{2} + \sum_{I=2}^{N} (N-I+1)*(N-I+2) = N^{2} + \sum_{I=1}^{N-1} K(K+1)$$

which is

$$N^{2} + \frac{N^{3} - N}{3}$$

Thus, the total number of reads is

$$N^{2} + \frac{2 * (N^{3} - N)}{3}$$

which is

$$\frac{4 * N^{3} + 6 * N^{2} - 4 * N}{6}$$

As can be seen from the following read and write counts generated by THREESUB#6, the only blocks that are rewritten twice are in the lower triangle (excluding column 1). All other blocks are only written once.

WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	2	1	1	1	1	1	1	1	1
1	2	2	1	1	1	1	1	1	1
1	2	2	2	1	1	1	1	1	1
1	2	2	2	2	1	1	1	1	1
1	2	2	2	2	2	1	1	1	1
1	2	2	2	2	2	2	1	1	1
1	2	2	2	2	2	2	2	1	1
1	2	2	2	2	2	2	2	2	1
1	2	2	2	2	2	2	2	2	2

READ COUNTS BY BLOCK

10	10	10	10	10	10	10	10	10	10
10	10	9	9	9	9	9	9	9	9
10	10	9	8	8	8	8	8	8	8
10	10	9	8	7	7	7	7	7	7
10	10	9	8	7	6	6	6	6	6
10	10	9	8	7	6	5	5	5	5
10	10	9	8	7	6	5	4	4	4
10	10	9	8	7	6	5	4	3	3
10	10	9	8	7	6	5	4	3	2
10	10	9	8	7	6	5	4	3	2

Algorithm THREESUB#7

After analyzing algorithm THREESUB#6, we can save N-1 reads and N-1 writes of the diagonal blocks (except for (1,1)) by reversing the direction of one of the inner loops.

Here is the PL/I routine:

```
CALL READ_BLOCK (1,1,BLOCK1);
CALL AUTO_REDUCE (1,1,BLOCK1);
CALL WRITE_BLOCK (1,1,BLOCK1);

DO J = 2 TO NUM_SUBMAT_PER_ROW;

   CALL READ_BLOCK (J, 1, BLOCK2);
   CALL VERT_REDUCE (J,1,BLOCK2,1,1,BLOCK1);
   CALL WRITE_BLOCK (J,1,BLOCK2);

END;
```

```
DO I = 2 TO NUM_SUBMAT_PER_COL;

   DO J = 1 TO I-1;

      CALL READ_BLOCK (J,I,BLOCK1);

      DO K = 1 TO J-1;
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL READ_BLOCK (K,I,BLOCK3);
         CALL TRI_REDUCE (J,I,BLOCK1,J,K,BLOCK2,K,I,BLOCK3);
      END;

      CALL READ_BLOCK (J,J,BLOCK2);
      CALL HORIZ_REDUCE (J,I,BLOCK1,J,J,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);

   END;

   DO J = NUM_SUBMAT_PER_ROW TO I BY -1;

      CALL READ_BLOCK (J,I,BLOCK1);

      DO K = 1 TO I-1;
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL RFAD_BLOCK (K,I,BLOCK3);
         CALL TRI_REDUCE (J,I,BLOCK1,J,K,BLOCK2,K,I,BLOCK3);
      END;

      IF J ^= I
         THEN CALL WRITE_BLOCK (J,I,BLOCK1);

   END;

   CALL AUTO_REDUCE (I,I,BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;
      CALL READ_BLOCK (J,I,BLOCK2);
      CALL VERT_REDUCE (J,I,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK (J,I,BLOCK2);
   END;

END;
```

The write counts can be derived by subtracting N-1 from the write formula for THREESUB#6, which gives us:

$$N ** 2 \quad + \quad \frac{N * (N - 1)}{2} \quad - \quad (N-1)$$

which is equal to

$$\frac{3 * N ** 2 \; - \; 3 * N \; + \; 2}{2}$$

The read count formula is also derived by subtracting N-1 from the read formula for THREESUB#6, giving us:

$$\frac{4 * N ** 3 \quad + \quad 6 * N ** 2 \quad - \quad 4 * N}{6} \quad - \quad (N-1)$$

which is equal to

$$\frac{4 * N ** 3 \; + \; 6 * N ** 2 \; - \; 10 * N \; + \; 6}{6}$$

Here are the read and write counts by block for a matrix divided into a ten by ten submatrix of blocks:

ALGORITHM THREESUB#7
WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	2	1	1	1	1	1	1	1	1
1	2	2	1	1	1	1	1	1	1
1	2	2	2	1	1	1	1	1	1
1	2	2	2	2	1	1	1	1	1
1	2	2	2	2	2	1	1	1	1
1	2	2	2	2	2	2	1	1	1
1	2	2	2	2	2	2	2	1	1
1	2	2	2	2	2	2	2	2	1

ALGORITHM THREESUB#7
READ COUNTS BY BLOCK

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

ALGORITHM THREESUB#6
WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	2	1	1	1	1	1	1	1	1
1	2	2	1	1	1	1	1	1	1
1	2	2	2	1	1	1	1	1	1
1	2	2	2	2	1	1	1	1	1
1	2	2	2	2	2	1	1	1	1
1	2	2	2	2	2	2	1	1	1
1	2	2	2	2	2	2	2	1	1
1	2	2	2	2	2	2	2	2	1
1	2	2	2	2	2	2	2	2	2

ALGORITHM THREESUB#6
READ COUNTS BY BLOCK

10	10	10	10	10	10	10	10	10	10
10	10	9	9	9	9	9	9	9	9
10	10	9	8	8	8	8	8	8	8
10	10	9	8	7	7	7	7	7	7
10	10	9	8	7	6	6	6	6	6
10	10	9	8	7	6	5	5	5	5
10	10	9	8	7	6	5	4	4	4
10	10	9	8	7	6	5	4	3	3
10	10	9	8	7	6	5	4	3	2
10	10	9	8	7	6	5	4	3	2

By comparing these counts, we see that indeed the diagonal blocks (except (1,1)) are read and written one time less in this algorithm. Therefore, algorithm THREESUB#7 saves N-1 reads and N-1 writes.

Algorithm THREESUB#8

Algorithm THREESUB#8 will only write each block once. Algorithms THREESUB#6 and THREESUB#7 had two distinct parts. They first reduced a column by the previous columns and then reduced the column by itself. Algorithm THREESUB#8 will divide a column of blocks into three parts. The first part will be for all the blocks in a column above the diagonal. These blocks must be reduced by the previous factors. However, these blocks are already in final form and are not affected by the reduction of the diagonal block in this column. Then, the diagonal block (I,I) is reduced by all the previous blocks in the Ith row (i.e., blocks (I,K) where K<I), and then block (I,I) is self-reduced. After the diagonal has been reduced, all the blocks below the diagonal are processed. These blocks are reduced by the blocks in the previous columns and then by the diagonal block.

Here is the PL/I routine for this algorithm:

```
CALL READ_BLOCK  (1,1,BLOCK1);
CALL AUTO_REDUCE (1,1,BLOCK1);
CALL WRITE_BLOCK (1,1,BLOCK1);

DO J = 2 TO NUM_SUBMAT_PER_COL;
   CALL READ_BLOCK (J,1,BLOCK2);
   CALL VERT_REDUCE(J,1,BLOCK2,1,1,BLOCK1);
   CALL WRITE_BLOCK(J,1,BLOCK2);
END;
```

```
DO I = 2 TO NUM_SUBMAT_PER_COL;

   DO J = 1 TO I-1;
      CALL READ_BLOCK (J,I,BLOCK1);
      DO K = 1 TO J-1;
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL READ_BLOCK (K,I,BLOCK3);
         CALL TRI_REDUCE
              (J,I,BLOCK1,J,K,BLOCK2,K,I,BLOCK3);
      END;
      CALL READ_BLOCK (J,J,BLOCK2);
      CALL HORIZ_REDUCE (J,I,BLOCK1,J,J,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);
   END;

   CALL READ_BLOCK (I,I,BLOCK1);

   DO K = 1 TO I-1;
      CALL READ_BLOCK (K,I,BLOCK3);
      CALL READ_BLOCK (I,K,BLOCK2);
      CALL TRI_REDUCE (I,I,BLOCK1,I,K,BLOCK2,K,I,BLOCK3);
   END;

   CALL AUTO_REDUCE (I,I,BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK (J,I,BLOCK1);

      DO K = 1 TO I-1;
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL READ_BLOCK (K,I,BLOCK3);
         CALL TRI_REDUCE
              (J,I,BLOCK1,J,K,BLOCK2,K,I,BLOCK3);
      END;

      CALL READ_BLOCK (I,I,BLOCK2);
      CALL VERT_REDUCE (J,I,BLOCK1,I,I,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);

   END;
END;
```

The goal of the algorithm is to rewrite each submatrix once. In fact, the number of writes is exactly N**2.

In algorithm THREESUB#7, there were two distinct phases to the algorithm. In the first phase of the Ith pass, the factors from columns 1 thru I-1 were applied. In the second phase, the diagonal block (I,I) was reduced and each block beneath (I,I) in that column (BLOCK(I+1,I), BLOCK(I+2,I), thru BLOCK(N,I)) was reduced using the values stored in the diagonal block, BLOCK (I,I).

Reviewing the Ith pass of algorithm THREESUB#7, we see that each block beneath the diagonal is read once in the first phase and once in the second phase. However, the diagonal BLOCK (I,I) is only read once in the Ith pass.

Therefore, a block in the lower triangle was read twice in pass I and once in the each of the remaining passes I+1, I+2, thru N-1, and N. The diagonal block is read once in pass I and once in each of the remaining passes.

In order to accomplish the goal of reducing the number of writes to N**2, algorithm THREESUB#8 combined phases 1 and 2. Hence, after applying the factors from previous columns to a lower triangle block, this block is reduced by the diagonal block above it. This saves one read for each block below the diagonal. However, this requires that the diagonal block be reread for each block beneath it. The diagonal block cannot be retained in memory, since we must apply the reductions from the previous columns to the next block and only three blocks can be in memory at once.

Therefore, pass I does one less read for the N-I blocks beneath the diagonal, but it reads in the diagonal an extra N-I times. Thus, it performs exactly the same number of reads as algorithm THREESUB#7, which is

$$\frac{4 * N^{**}3 + 6 * N^{**} 2 - 10 * N + 6}{6}$$

The read and write counts for a ten by ten subdivision of a matrix bear out the analysis presented above:

WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

READ COUNTS BY BLOCK

10	10	10	10	10	10	10	10	10	10
10	17	9	9	9	9	9	9	9	9
10	9	15	8	8	8	8	8	8	8
10	9	8	13	7	7	7	7	7	7
10	9	8	7	11	6	6	6	6	6
10	9	8	7	6	9	5	5	5	5
10	9	8	7	6	5	7	4	4	4
10	9	8	7	6	5	4	5	3	3
10	9	8	7	6	5	4	3	3	2
10	9	8	7	6	5	4	3	2	1

It is easy to see that the algorithm writes each block only once, so the total number of writes is N ** 2.

As we can see by comparing the read table for THREESUB#8 with that for THREESUB#7, a given diagonal element is read N-I times more often in THREESJB#8 than in THREESUB#7. Also, notice that each element in the lower triangle is read one less time in THREESUB#8 than in THREESUB#7.

THREESUB#7
READ COUNTS BY BLOCK

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

A direct derivation for the read count formula is also possible.

A block (J,I) above the diagonal is read once when it is reduced, and again for each block below it, so it is read N-J+1 times. Since there are (N-I) blocks above the diagonal in row J, there are (N-J)*(N-J+1) reads for the blocks above the diagonal in row J. This yields a total of

$$\sum_{J=1}^{N-1} (N-J)(N-J+1) = \sum_{K=1}^{N-1} K(K+1) = \frac{N^{3} - N}{3}$$

Each block below the diagonal is read once when it is reduced by both the L factors to its left and by the diagonal block above it in the same column. Each block is also read once to reduce each of the blocks to its right. Block (J,I) where J<I is read N-I+1 times, so the total reads for the N-I blocks below the diagonal in column I is (N-I)(N-I+1). Thus, the number of reads for all blocks below the diagonal is

$$\sum_{J=1}^{N-1} (N-J)(N-J+1) = \sum_{K=1}^{N-1} K(K+1) = \frac{N^{3} - N}{3}$$

Finally, consider the diagonal block (I,I). Block (1,1) is read once to reduce itself and the rest of the first column. Block (1,1) is read again for each of the N-1 blocks to the right of it in the first row, so block (1,1) is read N times. Any other diagonal block (I,I) is read once when it is reduced. then, it is read again for each of the N-I blocks below it and once for each of the N-I blocks to the right of it, for a total of 2*(N-I)+1 times. Thus, the number of reads for the diagonal is

$$N + \sum_{I=2}^{N} (2*(N-I)+1) = N + \sum_{K=1}^{N-1} (2*K + 1)$$

which is

$$N^{2} - N + 1.$$

Thus, the total number of reads is

$$\frac{N^{3} - N}{3} + \frac{N^{3} - N}{3} + N^{2} - N + 1$$

which is

$$\frac{4 * N^{3} + 6 * N^{2} - 10 * N + 6}{6}$$

which is the same as the read count formula derived for algorithm THREESUB#7.

Algorithm THREESUB#9

After analyzing the data flow from algorithm THREESUB#8, it can be seen that BLOCK (1,1) is read twice in the initial phases of the algorithm, once when it reduced and once to reduce BLOCK (1,2). Also, BLOCK (1,2) is read N times in the second pass, once when it is reduced and once when it is used to reduce blocks beneath it. This is due to the loops necessary for passes 3 thru N. Hence, by removing pass 2 from the main loop and having a separate loop for the second column of blocks at the start of the main program, we can save N reads.

Here is the PL/I routine:

```
CALL READ_BLOCK  (1,1,BLOCK1);
CALL AUTO_REDUCE (1,1,BLOCK1);
CALL WRITE_BLOCK (1,1,BLOCK1);

DO J = 2 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK (J,1,BLOCK2);
   CALL VERT_REDUCE(J,1,BLOCK2,1,1,BLOCK1);
   CALL WRITE_BLOCK(J,1,BLOCK2);

END;


CALL READ_BLOCK (1,2,BLOCK2);
CALL HORIZ_REDUCE (1,2,BLOCK2,1,1,BLOCK1);
CALL WRITE_BLOCK (1,2,BLOCK2);

CALL READ_BLOCK (2,1,BLOCK1);
CALL READ_BLOCK (2,2,BLOCK3);
CALL TRI_REDUCE (2,2,BLOCK3,2,1,BLOCK1,1,2,BLOCK2);
CALL AUTO_REDUCE (2,2,BLOCK3);
CALL WRITE_BLOCK (2,2,BLOCK3);

DO J = 3 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK (J,1,BLOCK1);
   CALL READ_BLOCK (J,2,BLOCK3);
   CALL TRI_REDUCE (J,2,BLOCK3,J,1,BLOCK1,1,2,BLOCK2);
   CALL READ_BLOCK (2,2,BLOCK1);
   CALL VERT_REDUCE (J,2,BLOCK3,2,2,BLOCK1);
   CALL WRITE_BLOCK (J,2,BLOCK3);

END;
```

```
/** MAIN LOOP   PASSES 3 THRU N **/

DO I = 3 TO NUM_SUBMAT_PER_COL;

   DO J = 1 TO I-1;

      CALL READ_BLOCK (J,I,BLOCK1);

      DO K = 1 TO J-1;
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL READ_BLOCK (K,I,BLOCK3);
         CALL TRI_REDJCE
              (J,I,BLOCK1,J,K,BLOCK2,K,I,BLOC(3);
      END;

      CALL READ_BLOCK (J,J,BLOCK2);
      CALL HORIZ_REDUCE (J,I,BLOCK1,J,J,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);

   END;

   CALL READ_BLOCK (I,I,BLOCK1);

   DO K = 1 TO I-1;
      CALL READ_BLOCK (K,I,BLOCK3);
      CALL READ_BLOCK (I,K,BLOCK2);
      CALL TRI_REDUCE
           (I,I,BLOCK1,I,K,BLOCK2,K,I,BLOCK3);
   END;

   CALL AUTO_REDUCE (I,I,BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK (J,I,BLOCK1);

      DO K = 1 TO I-1;
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL READ_BLOCK (K,I,BLOCK3);
         CALL TRI_REDJCE
              (J,I,BLOCK1,J,K,BLOCK2,K,I,BLOC(3);
      END;

      CALL READ_BLOCK (I,I,BLOCK2);
      CALL VERT_REDUCE (J,I,BLOCK1,I,I,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);

   END;

END;
```

By modifying the looping patterns, algorithm THREESUB#9 saves one read of BLOCK (1,1) and N-1 reads of BLOCK (1,2) in passes 1 and 2. Hence, the read formula for algorithm THREESUB#9 is the read formula for algorithm THREESUB#8 minus N. So, we have

$$\frac{4 * N ** 3 + 6 * N ** 2 - 10 * N + 6}{6} - N$$

which is

$$\frac{4 * N ** 3 + 6 * N ** 2 - 16 * N + 6}{6}$$

By comparing the read counts from algorithm THREESUB#8 with those from algorithm THREESUB#9, we confirm that algorithm THREESUB#9 only reads BLOCK (1,2) once while algorithm THREESUB#8 does N reads. Also, THREESUB#9 reads BLOCK (1,1) only N-1 times while THREESUB#8 reads BLOCK (1,1) N times.

ALGORITHM THREESUB#8 READ COUNTS BY BLOCK									
10	10	10	10	10	10	10	10	10	10
10	17	9	9	9	9	9	9	9	9
10	9	15	8	8	8	8	8	8	8
10	9	8	13	7	7	7	7	7	7
10	9	8	7	11	6	6	6	6	6
10	9	8	7	6	9	5	5	5	5
10	9	8	7	6	5	7	4	4	4
10	9	8	7	6	5	4	5	3	3
10	9	8	7	6	5	4	3	3	2
10	9	8	7	6	5	4	3	2	1

ALGORITHM THREESJB#9 READ COUNTS BY BLOCK									
9	1	10	10	10	10	10	10	10	10
10	17	9	9	9	9	9	9	9	9
10	9	15	8	8	8	8	8	8	8
10	9	8	13	7	7	7	7	7	7
10	9	8	7	11	6	6	6	6	6
10	9	8	7	6	9	5	5	5	5
10	9	8	7	6	5	7	4	4	4
10	9	8	7	6	5	4	5	3	3
10	9	8	7	6	5	4	3	3	2
10	9	8	7	6	5	4	3	2	1

Algorithm THREESUB#10

After analyzing the data flow of passes 3 thru N of algorithm THREESUB#9, we can achieve a further savings of N-2 reads.

In pass K of algorithm THREESUB#9, BLOCK (1,K) is read and reduced by BLOCK (1,1). Then, BLOCKs (1,K), (2,1), and (2,K) are read. BLOCK (2,K) is reduced by the product of BLOCKs (2,1) and (1,K). Then, BLOCK (2,2) is read and used to reduce BLOCK (2,K). There is no need to reread BLOCK (1,K) in order to reduce BLOCK (2,K). Algorithm THREESUB#10 will save this unnecessary read of BLOCK (1,K). Since this savings will occur only in passes 3 thru N, algorithm THREESUB#10 saves N-2 reads.

Please note that BLOCK (1,K) cannot be saved in memory when reducing BLOCKs (3,K), (4,K), ..., (N,K), since we can only have three blocks in memory at once and any block below the second row must be reduced by the product of BLOCK (2,K) and (3,2) as well as the product of BLOCK (1,K) and (3,1).

Here is the PL/I routine:

```
CALL READ_BLOCK  (1,1,BLOCK1);
CALL AUTO_REDUCE (1,1,BLOCK1);
CALL WRITE_BLOCK (1,1,BLOCK1);

DO J = 2 TO NUM_SUBMAT_PER_COL;
   CALL READ_BLOCK (J,1,BLOCK2);
   CALL VERT_REDUCE(J,1,BLOCK2,1,1,BLOCK1);
   CALL WRITE_BLOCK(J,1,BLOCK2);
END;

IF NUM_SUBMAT_PER_COL > 1
   THEN DO;

          CALL READ_BLOCK (1,2,BLOCK2);
          CALL HORIZ_REDUCE (1,2,BLOCK2,1,1,BLOCK1);
          CALL WRITE_BLOCK (1,2,BLOCK2);

          CALL READ_BLOCK (2,1,BLOCK1);
          CALL READ_BLOCK (2,2,BLOCK3);
          CALL TRI_REDUCE
               (2,2,BLOCK3,2,1,BLOCK1,1,2,BLOCK2);
          CALL AUTO_REDUCE (2,2,BLOCK3);
          CALL WRITE_BLOCK (2,2,BLOCK3);

        END;

DO J = 3 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK (J,1,BLOCK1);
   CALL READ_BLOCK (J,2,BLOCK3);
   CALL TRI_REDUCE (J,2,BLOCK3,J,1,BLOCK1,1,2,BLOCK2);
   CALL READ_BLOCK (2,2,BLOCK1);
   CALL VERT_REDUCE (J,2,BLOCK3,2,2,BLOCK1);
   CALL WRITE_BLOCK (J,2,BLOCK3);

END;
```

```
DO I = 3 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK (1,I,BLOCK1);
   CALL READ_BLOCK (1,1,BLOCK2);
   CALL HORIZ_REDUCE (1,I,BLOCK1,1,1,BLOCK2);
   CALL WRITE_BLOCK (1,I,BLOCK1);
   CALL READ_BLOCK (2,I,BLOCK3);
   CALL READ_BLOCK (2,1,BLOCK2);
   CALL TRI_REDUCE (2,I,BLOCK3,2,1,BLOCK2,1,I,BLOCK1);
   CALL READ_BLOCK (2,2,BLOCK2);
   CALL HORIZ_REDUCE (2,I,BLOCK3,2,2,BLOCK2);
   CALL WRITE_BLOCK (2,I,BLOCK3);

   DO J = 3 TO I-1;
      CALL READ_BLOCK (J,I,BLOCK1);
      DO K = 1 TO J-1;
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL READ_BLOCK (K,I,BLOCK3);
         CALL TRI_REDUCE
              (J,I,BLOCK1,J,K,BLOCK2,K,I,BLOCK3);
      END;
      CALL READ_BLOCK (J,J,BLOCK2);
      CALL HORIZ_REDUCE (J,I,BLOCK1,J,J,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);
   END;

   CALL READ_BLOCK (I,I,BLOCK1);

   DO K = 1 TO I-1;
      CALL READ_BLOCK (K,I,BLOCK3);
      CALL READ_BLOCK (I,K,BLOCK2);
      CALL TRI_REDUCE (I,I,BLOCK1,I,K,BLOCK2,K,I,BLOCK3);
   END;

   CALL AUTO_REDUCE (I,I,BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;
      CALL READ_BLOCK (J,I,BLOCK1);
      DO K = 1 TO I-1;
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL READ_BLOCK (K,I,BLOCK3);
         CALL TRI_REDUCE
              (J,I,BLOCK1,J,K,BLOCK2,K,I,BLOCK3);
      END;
      CALL READ_BLOCK (I,I,BLOCK2);
      CALL VERT_REDUCE (J,I,BLOCK1,I,I,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);
   END;

END;
```

Since algorithm THREESUB#10 does N-2 reads less than algorithm THREESUB#9, we can derive the read formula for THREESUB#10 from that of THREESUB#9. So we have,

$$\frac{4 * N ** 3 + 6 * N ** 2 - 16 * N + 6}{6} - (N-2)$$

which is

$$\frac{4 * N ** 3 + 6 * N ** 2 - 22 * N + 18}{6}$$

By comparing read counts for THREESUB#10 with the counts from THREESUB#9, we can confirm the blocks in the first row from columns 3 thru N are read one less time. Thus, we save N-2 reads.

ALGORITHM THREESUB#9
READ COUNTS BY BLOCK

9	1	10	10	10	10	10	10	10	10
10	17	9	9	9	9	9	9	9	9
10	9	15	8	8	8	8	8	8	8
10	9	8	13	7	7	7	7	7	7
10	9	8	7	11	6	6	6	6	6
10	9	8	7	6	9	5	5	5	5
10	9	8	7	6	5	7	4	4	4
10	9	8	7	6	5	4	5	3	3
10	9	8	7	6	5	4	3	3	2
10	9	8	7	6	5	4	3	2	1

ALGORITHM THREESUB#10
READ COUNTS BY BLOCK

9	1	9	9	9	9	9	9	9	9
10	17	9	9	9	9	9	9	9	9
10	9	15	8	8	8	8	8	8	8
10	9	8	13	7	7	7	7	7	7
10	9	8	7	11	6	6	6	6	6
10	9	8	7	6	9	5	5	5	5
10	9	8	7	6	5	7	4	4	4
10	9	8	7	6	5	4	5	3	3
10	9	8	7	6	5	4	3	3	2
10	9	8	7	6	5	4	3	2	1

Comparison of Algorithms THREESUB#1 thru THREESUB#10

It is time to stop and compare these algorithms.

Algorithms THREESUB#1 thru THREESUB#10 use the submatrix storage method. This method requires that at least three blocks must be able to fit in memory at once. In fact, the size of the blocks is chosen with memory in mind, so that exactly 3 blocks fill fit into memory. Let us compare these algorithms.

Even though the read and write count equations are given in the sections describing each algorithm, they are repeated here for comparison purposes (using the same denominator).

Algorithm	Number of Reads	Number of Writes
THREESUB#1	$\frac{8*N**3+6*N**2-2*N}{12}$	$\frac{4*N**3+6*N**2+2*N}{12}$
THREESUB#2	$\frac{8*N**3+6*N**2-14*N+12}{12}$	$\frac{4*N**3+6*N**2-10*N+12}{12}$
THREESUB#3	$\frac{8*N**3-20*N+24}{12}$	$\frac{4*N**3+6*N**2-10*N+12}{12}$
THREESUB#4	$\frac{8*N**3+6*N**2-2*N}{12}$	$\frac{4*N**3+6*N**2+2*N}{12}$
THREESUB#5	$\frac{8*N**3+4*N}{12}$	$\frac{4*N**3+8*N}{12}$
THREESUB#6	$\frac{8*N**3+12*N**2-8*N}{12}$	$\frac{18*N**2-6*N}{12}$
THREESUB#7	$\frac{8*N**3+12*N**2-20*N+12}{12}$	$\frac{18*N**2-18*N+12}{12}$
THREESUB#8	$\frac{8*N**3+12*N**2-20*N+12}{12}$	$\frac{12*N**2}{12}$
THREESUB#9	$\frac{8*N**3+12*N**2-32*N+12}{12}$	$\frac{12*N**2}{12}$
THREESUB#10	$\frac{8*N**3+12*N**2-44*N+36}{12}$	$\frac{12*N**2}{12}$

As can be seen from the formulas above, algorithms THREESUB#8, THREESUB#9, and THREESUB#10 do the least number of writes. Even though these algorithms do more reads than several of the other algorithms, their total I/O count is less than all the others due to their large write count advantage. Since, algorithm THREESUB#10 does the least number of reads from this group, algorithm THREESUB#10 is the 'best' three square decomposition method.

A table showing the combined read and write counts is presented below:

Algorithm	Total I/O Count
THREESUB#1	$\frac{6*N**3 + 6*N**2}{6}$
THREESUB#2	$\frac{6*N**3 + 6*N**2 - 12*N + 12}{6}$
THREESUB#3	$\frac{6*N**3 + 3*N**2 - 15*N + 18}{6}$
THREESUB#4	$\frac{6*N**3 + 6*N**2}{6}$
THREESUB#5	$\frac{6*N**3 + 6*N}{6}$
THREESUB#6	$\frac{4*N**3 + 15*N**2 - 7*N}{6}$
THREESUB#7	$\frac{4*N**3 + 15*N**2 - 19*N + 12}{6}$
THREESUB#8	$\frac{4*N**3 + 12*N**2 - 10*N + 6}{6}$
THREESUB#9	$\frac{4*N**3 + 12*N**2 - 16*N + 6}{6}$
THREESUB#10	$\frac{4*N**3 + 12*N**2 - 22*N + 18}{6}$

After examining the table above, we can conclude that THREESUB#10 is the best algorithm for this storage method. The only exceptions are the cases in which N=2 or N=3. For N=2, THREESUB#3 requires 8 I/O operations, while THREESUB#10 requires 9 I/O operations. For N=3, THREESUB#3 requires 27 I/O operations, while THREESUB#10 requires 28 I/O operations.

A table listing the read and write counts for each algorithm for three sample matrices will reconfirm the advantages of algorithm THREESUB#10.

algorithm	matrix divided into 5 by 5 submatrices			matrix divided into 10 by 10 submatrices			matrix divided into 20 by 20 submatrices		
	read	write	total	read	write	total	read	write	total
SUB#1	95	55	150	715	385	1100	5530	2870	8400
SUB#2	91	51	142	706	376	1082	5511	2851	8372
SUB#3	77	51	126	652	376	1028	5302	2351	8153
SUB#4	95	55	150	715	385	1100	5530	2870	8400
SUB#5	85	45	140	670	340	1010	5340	2680	8020
SUB#6	105	35	140	760	145	905	5720	590	6310
SUB#7	101	31	132	751	136	887	5701	571	6272
SUB#8	101	25	126	751	100	851	5701	400	6101
SUB#9	96	25	121	741	100	841	5681	400	6081
SUB#10	93	25	118	733	100	833	5663	400	6063

Algorithms THREESUB#1, THREESUB#2, and THREESUB#3 reflect the same basic idea: reduce BLOCK (I,I) and then reduce the rest of the blocks.

Algorithms THREESUB#4 thru THREESUB#10 also have a common theme: reduce a block only when its column is being processed.

Algorithms THREESUB#6 thru THREESUB#10 succeed in producing a savings by reducing the number of writes. Algorithms THREESUB#8 thru THREESUB#10 write each block only once.

From the N**2 write algorithms, algorithm THREESUB#10 does the least number of reads.

Algorithm THREESUB#10 does do more reads than algorithms THREESUB#1 thru THREESUB#4. However, the read advantage is small in comparison to the write advantage enjoyed by algorithm THREESUB#10.

Hence, algorithm THREESUB#10 is the 'best' submatrix decomposition without interchanges algorithm, except for the special cases, N=2 and N=3.

Comparison of Algorithms COL#1 and THREESUB#10

It is not as easy to compare algorithms COL#1 and THREESUB#10 as it was to compare THREESUB#1 thru THREESUB#10 or ROW#1 thru ROW#3 or ROW#3 versus COL#1. Algorithm THREESUB#10 uses submatrix storage and requires the blocks to be small enough to enable three blocks in memory at once. Algorithm COL#1 uses column storage and requires the blocks to contain complete columns and be small enough to allow two blocks in memory at once.

In order to compare these algorithms, solution of an actual matrix by each method must be contrasted using different memory sizes. As an example, we will look at a 1000 by 1000 matrix. Each element uses four bytes. We have selected memory sizes of 48000, 64000, 96000, 128000, and 256000 bytes available for data. The results are summarized in the table below.

algorithm COL#1

	48000	64000	96000	128000	256000
number of submatrices	167	125	84	63	32
size of each block	1000x6	1000x8	1000x12	1000x16	1000x32
number of reads	14028	7875	3570	2016	528
number of writes	167	125	84	63	32
total I/O	14195	8000	3654	2079	560

algorithm THREESUB#10

	48000	64000	96000	128000	256000
submatrix-shape	16x16	14x14	12x12	10x10	7x7
size of each block	63x63	73x73	89x89	103x103	146x146
number of reads	2391	1977	1255	733	255
number of writes	256	196	144	100	49
total I/O	2647	2173	1399	833	304

As can be seen, algorithm THREESUB#10 does much better than algorithm COL#1 with a small memory. As memory gets bigger, algorithm COL#1 catches up. In the following table, 512000, 768000, and 1024000 bytes are contrasted.

	512000	768000	1024000
algorithm COL#1			
reads	136	66	36
writes	16	11	8
total	152	77	54
algorithm THREESUB#10			
reads	93	53	53
writes	25	16	16
total	118	69	69

Algorithm COL#1 is better than algorithm THREESUB#10 in memories greater than 768000 bytes.

The poor performance of Algorithm COL#1 with a small memory size can be explained easily. Before reducing block N, the factors from the previous blocks were applied. Most of the elements in block N-1 were U factors, which were not needed. On the other hand, algorithm THREESUB#10 did not bring into memory as many unneeded values.

The conclusions drawn from the experiments can also be derived mathematically using the following argument:

> Assume the matrix has n elements per row and n elements per column.
>
> Assume memory can hold m elements of the matrix at once.

Since algorithm THREESUB#10 uses the submatrix storage scheme, the following statements follow:

> The m elements of memory must be divided into three, since this scheme requires the presence in memory of three subblocks simultaneously. Each submatrix is square, having x elements per submatrix row and x elements per submatrix column. Hence, x is approximately the square root of (m/3). So, memory can hold three subblocks at once or 3*x**2 elements.
>
> Since a subblock contains x elements per submatrix row, @ subblocks will be needed to cover a matrix row (where @ = n/x). Hence, the matrix can be viewed as being an @ by @ matrix of subblocks. Furthermore, the matrix can also be regarded as an @x by @x matrix of elements.

```
memory = 3 * x ** 2      elements
matrix = @x by @x        elements
```

Since algorithm COL#1 uses the column storage scheme, the following statements follow:

> The m elements of memory must be divided into two, since this scheme requires the presence in memory of two subblocks simultaneously. Each submatrix is rectangular, having y elements per submatrix row and n elements per submatrix column. Hence, y is approximately the quotient of (m/2) divided by n. So, memory can hold two subblocks at once or 2*y*n elements.
>
> Since a subblock contains y elements per submatrix row, T subblocks will be needed to cover a matrix row (where T = n/y). Hence, the matrix can be viewed as consisting of T blocks each y by n elements big. Furthermore, the matrix can also be regarded as a T * y * n matrix of elements.
>
> memory = 2 * y * n elements
> matrix = n by T * y elements

```
Algorithm THREESUB#10          Algorithm COL#1
--------------------          --------------

memory = 3 * x ** 2             memory = 2 * y * n
matrix = @x * @x                matrix = T * y * n
```

Since the matrix is n by n, and @x is approximately n; we derive the following relationships between the different storage schema:

```
Algorithm THREESUB#10          Algorithm COL#1
--------------------          --------------

memory = 3 * x ** 2             memory = 2 * y * Q * x
matrix = @x * @x                matrix = T * y * Q * x
```

Using memory as the common limiting feature, we have:

```
3 * x ** 2   =   2 * y * Q * x

 3 * x       =   y
 -----
 2 * Q
```

Now, using matrix row size as the common feature, we have:

$$Q * x = T * y$$

$$Q * x = T * \frac{3 * x}{2 * Q}$$

$$2 * Q * Q * x = T * 3 * x$$

$$\frac{2 * Q * Q}{3} = T$$

Now, T is the same as the N used for the number of subblocks used in the algorithm COL#1 read and write count formulas.

Also, Q is the same as the N used for the number of subblocks used in the algorithm THREESUB#10 read and write count formulas.

Rewriting the formulas for algorithms THREESUB#10 and COL#1 in terms of Q and T, we have:

algorithm	number of reads	number of writes
COL#1	$\frac{T * (T + 1)}{2}$	T
THREESUB#10	$\frac{4 * Q**3 + 6 * Q**2 - 22 * Q + 18}{6}$	Q ** 2

Converting the formula(e) for algorithm COL#1 into terms of Q, we have:

algorithm	number of reads	number of writes
COL#1	$\frac{4 * Q ** 4 + 6 * Q ** 2}{18}$	$\frac{2 * Q ** 2}{3}$
THREESUB#10	$\frac{4 * Q**3 + 6 * Q**2 - 22 * Q + 18}{6}$	$Q ** 2$

The total I/O formula for each algorithm is:

algorithm COL#1 $\frac{4 * Q ** 4 + 18 * Q ** 2}{18}$

algorithm THREESUB#10 $\frac{12 * Q ** 3 + 36 * Q ** 2 - 66 * Q + 54}{18}$

It is immediately apparent that the Q ** 4 term in the formula for algorithm COL#1 will dominate.

A table comparing the above formulae is presented below:

Q VALUE	COL#1 I/O COUNT	THREESUB#10 I/O COUNT
1.00	1.22	2.00
2.00	7.56	9.00
3.00	27.00	28.00
4.00	72.89	63.00
5.00	163.89	118.00
6.00	324.00	197.00
7.00	582.56	304.00
8.00	974.22	443.00
9.00	1539.00	618.00
10.00	2322.22	833.00

In this table, the last column (labeled THREESUB#10) gives the total I/O count for algorithm THREESUB#10 for the corresponding value of Q. For example, if Q = 1, the entire matrix fits into memory, so the I/O count is 2, one read and one write.

The second column is not exactly the I/O count for algorithm COL#1. Instead it is the value of the formula above, which was based on the assumption that T = (2./3.) * Q**2. Clearly, the I/O count for Q = 1 is 2, one read and one write. In general, it depends on the amount of rounding involved in computing T. For large values of Q, the effect of rounding is not as important, and THREESUB#10 is significantly better better than COL#1.

To see what happens for small Q, we first consider Q=2. This means that there are 4 blocks, so memory is large enough to hold 3/4 of the matrix, but not the entire matrix. Since COL#1 requires 2 blocks in memory, T must be 3. (T=2 is never used, because in that case the entire matrix fits into memory at once and T = 1.) With T=3, only 2/3 of the matrix need be in memory, and this will certainly fit, since memory can hold 3/4 of the matrix. With T=3, COL#1 yields 6 reads and 3 writes for a total I/O count of 9. Thus, THREESUB#10 and COL#1 do the same amount of I/O when Q=1 or Q=2. (But THREESUB#3 uses only 8 I/O operations in this case.)

Now suppose Q = 3, so THREESUB#10 splits the matrix into Q**2 or 9 blocks. Since three blocks must be in memory, memory will hold 1/3 of the matrix, but not 3/4 of the matrix. Then, depending on the size of memory, T can be anywhere from 3 (if memory will hold 2/3 of the matrix) up to 6 (if memory holds only 1/3 of the matrix). The corresponding I/O counts for COL#1 vary from 9 for T=3 to 27 for T=6. Thus, COL#1 is better than THREESUB#10 for Q=3.

We obtain the following table for small Q.

Q value	total I/O for SUB#10	fraction of matrix in memory lower bound		upper bound	T values from	to	total I/O COL#1 from to
1	2	1	to	1	1	1	2 to 2
2	9	3/4	to	1	3	3	9 to 9
3	28	1/3	to	3/4	3	6	9 to 27
4	63	3/16	to	1/3	7	11	35 to 77
5	118	3/25	to	3/16	11	17	77 to 190
6	197	3/36	to	3/25	17	24	190 to 325
7	304	3/49	to	3/36	24	33	325 to 594

Indeed for Q >= 7, THREESUB#10 is always better than COL#1.

To obtain a bound for the I/O count of COL#1, we consider the case in which THREESUB#10 uses Q blocks per row, but memory is almost large enough to allow it to use (Q-1) blocks per row. Then,

$$T \le \frac{2 * (Q-1) ** 2}{3}$$

so we use T = CEIL ((2*(Q-1)**2)/3) and compare the I/O counts for COL#1 using this T with those from THREESUB#10 using Q blocks per row. The results are given in the table below.

Q VALUE	T VALUE	COL#1 TOTAL I/O COUNT	THREESUB#10 TOTAL I/O COUNT
3	3	9.000000E+00	2.800000E+01
4	6	2.700000E+01	6.300000E+01
5	11	7.700000E+01	1.180000E+02
6	17	1.700000E+02	1.970000E+02
7	24	3.240000E+02	3.040000E+02
8	33	5.940000E+02	4.430000E+02
9	43	9.890000E+02	6.180000E+02
10	54	1.539000E+03	8.330000E+02
25	384	7.430400E+04	1.157800E+04
50	1601	1.284002E+06	8.815300E+04
75	3651	6.670377E+06	2.922230E+05
100	6534	2.135638E+07	6.863030E+05
150	14801	1.095570E+08	2.294453E+06
200	26401	3.485460E+08	5.412603E+06
250	41334	8.543118E+08	1.054075E+07
300	59601	1.776229E+09	1.817890E+07
400	106134	5.632372E+09	4.298520E+07
500	166001	1.377842E+10	8.383150E+07
1000	665334	2.213357E+11	6.686630E+08
1500	1498001	1.122006E+12	2.254495E+09
2000	2664001	3.548455E+12	5.341326E+09
2500	4163334	8.666681E+12	1.042916E+10

This table reinforces our findings that for very large Q, THREESUB#10 is orders of magnitude better than COL#1.

A Point Worth Reiterating

There is another advantage in using submatrix storage over row or column storage. If matrix A is very large in relation to memory, it is quite possible that two complete rows or columns cannot reside in memory simultaneously. Row and column storage cannot be used for these matrices.

Chapter 5

A Variation for the Submatrix Storage Method

Let us consider the initial presentation of the submatrix storage method. Most decomposition algorithms contain a statement like:

$$A\ (i,j) = A\ (i,j) - (\ A\ (i,k) * A\ (k,j)\ )$$

where (i,j), (i,k), and (k,j) may be in three different blocks. Therefore, it was natural to consider algorithms that hold three blocks in memory at once where the equation above can be processed in one logical step.

However, we now turn our attention to algorithms requiring only two blocks in memory at once. This submatrix storage method is only viable when at least two complete blocks fit into memory at once. Without two blocks in memory at once, the multiplication between block (i,k) and block (k,j) cannot be performed. If we have two blocks in memory simultaneously, we can form a product block, then add it to the target block.

This storage method does not assume that a complete row or column will fit into memory at once.

The TWOSUB class of algorithms which will be presented below holds two submatrices in memory at once. The TWOSUB submatrix is larger than the THREESUB submatrix discussed in the previous section.

The Hidden Vector

Since we are considering algorithms where the submatrices are square and two blocks will reside in memory at once, it would be expected that the number of elements in a block depends on the number of elements that fit into approximately one-half of memory. Thus, the number of elements in a row of a block is the square root of one-half of memory. Since each block is square, the number of elements per row of a block equals the number of elements per column of a block.

The above statements are ALMOST true.

Consider the following example:

```
                                   30 31 32
                                   33 34 35
                                   36 37 38

 10 11 12                          20 21 22
 13 14 15                          23 24 25
 16 17 18                          26 27 28
```

Here are three blocks, where each block contains 9 elements arranged in 3 by 3 submatrix.

Call the block containing the numbers 30 thru 38 A(K,J). Assume A(K,J) contains U values.

Call the block containing the numbers 10 thru 18 A(I,K). Assume A(I,K) contains L values.

Call the block containing the numbers 20 thru 28 A(I,J).

Assume the operation to be performed is the application of the L values stored in A(I,K) to A(I,J) using the U values in A(K,J).

If all three blocks are in memory at once, the operation can be performed easily. The new version of A(I,J) will be:

```
20-10*30-11*33-12*36 21-10*31-11*34-12*37 22-10*32-11*35-12*36
23-13*30-14*33-15*36 24-13*31-14*34-15*37 25-13*32-14*35-15*36
26-16*30-17*33-18*36 27-16*31-17*34-18*37 28-16*32-17*35-18*36
```

However, if only two blocks are to be in memory at once, this operation becomes somewhat more difficult. Assume we read A(K,J) into BLOCK1. Then, we read A(I,K) into BLOCK2. We must convert A(K,J) into:

```
-10*30-11*33-12*36  -10*31-11*34-12*37  -10*32-11*35-12*36
-13*30-14*33-15*36  -13*31-14*34-15*37  -13*32-14*35-15*36
-16*30-17*33-18*36  -16*31-17*34-18*37  -16*32-17*35-18*36
```

After we have done the above transformation, we can then read A(I,J) into BLOCK2, add BLOCK1 to BLOCK2, and rewrite BLOCK2 to disk.

Since we are not using an array processor, the transformation of A(K,J) is impossible without the use of an auxiliary vector of size 3 holding a column of A(K,J) while transforming that column into the column of products.

For example, we must store column 1 of A(K,J) in this vector. So, this vector will contain 30 33 36. Then, we replace column 1 of A(K,J) with

```
-10*30-11*33-12*36
-13*30-14*33-15*36
-16*30-17*33-18*36
```

Therefore, memory must be large enough to hold two submatrices of size n by n and a vector of size n.

Since the size of each submatrix is governed by the amount of available memory, we must allow for this hidden vector.

Common Variables

Using the results of the previous section, here are the PL/I statements needed for two square blocks.

```
GET LIST (NUM_ELTS_PER_ROW_OF_MATRIX,
          MEMORY_SIZE_IN_BYTES,
          BYTES_PER_ELEMENT);

MEMORY_SIZE_IN_ELTS  = FLOOR (MEMORY_SIZE_IN_BYTES /
                              BYTES_PER_ELEMENT);

NUM_ELTS_PER_COL_OF_MATRIX  = NUM_ELTS_PER_ROW_OF_MATRIX;
NUM_ELTS_IN_MATRIX          = NUM_ELTS_PER_ROW_OF_MATRIX *
                              NUM_ELTS_PER_COL_OF_MATRIX;

TEMP1 = FLOOR(MEMORY_SIZE_IN_ELTS / 2);
TEMP2 = FLOOR(SQRT (TEMP1));

DO WHILE ((TEMP2 + 2*(TEMP2*TEMP2)) > MEMORY_SIZE_IN_ELTS);
   TEMP2 = TEMP2 - 1;
END;

NUM_ELTS_PER_ROW_OF_SUBMATRIX = TEMP2;
NUM_ELTS_PER_COL_OF_SUBMATRIX = TEMP2;

NUM_ELTS_IN_SUBMATRIX   = TEMP2 * TEMP2;

NUM_SUBMATRIX_PER_MATRI = CEIL
                          (FLOAT(NUM_ELTS_IN_MATRIX ) /
                           FLOAT(NUM_ELTS_IN_SUBMATRIX ));

NUM_SUBMAT_PER_COL = CEIL
                     (FLOAT(NUM_ELTS_PER_COL_OF_MATRIX) /
                      FLOAT(NUM_ELTS_PER_COL_OF_SUBMATRIX));

NUM_SUBMAT_PER_ROW = CEIL
                     (FLOAT(NUM_ELTS_PER_ROW_OF_MATRIX) /
                      FLOAT(NUM_ELTS_PER_ROW_OF_SUBMATRIX));

ALLOCATE BLOCK1;
ALLOCATE BLOCK2;
```

Note that BLOCK3 was not allocated and does not occupy storage.

Also, note that the purpose of the DO WHILE loop is to make sure that there is enough memory available for the hidden vector after computing TEMP2. If there is not enough room for the hidden vector, we keep decreasing the dimensions of each submatrix by 1 until both submatrices and the hidden vector fit into memory.

Algorithm TWOSUB#1

Assume A is partitioned into an m by m matrix of blocks. The algorithm reduces BLOCK (I,I), then reduces the remaining blocks in the I th column and in the I th row, and then the submatrix bounded by row I+1 and column I+1. This algorithm is analogous to algorithm THREESUB#1.

The PL/I main routine is:

```
DO I = 1 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK  (I,I,BLOCK1);
   CALL AUTO_REDUCE (I,I,BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);


   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK (J,I,BLOCK2);
      CALL VERT_REDUCE(J,I,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK(J,I,BLOCK2);

      CALL READ_BLOCK (I,J,BLOCK2););
      CALL HORIZ_REDUCE(I,J,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK (I,J,BLOCK2);

   END;

   DO J = I+1 TO NUM_SUBMAT_PER_COL;

      DO K = I+1 TO NUM_SUBMAT_PER_ROW;
         CALL READ_BLOCK (I,K,BLOCK1);
         CALL READ_BLOCK (J,I,BLOCK2);
         CALL MULTIPLY (J,I,BLOCK2,I,K,BLOCK1);
         CALL READ_BLOCK (J,K,BLOCK1);
         CALL TRI_REDUCE (J,K,BLOCK1,J,K,BLOCK2);
         CALL WRITE_BLOCK (J,K,BLOCK1);
      END;

   END;

END;
```

READ_BLOCK (I,J,BLOCK) is a subroutine which does the actual read of BLOCK (I,J) from the disk to memory. WRITE_BLOCK (I,J,BLOCK) is a subroutine which does the actual write of BLOCK (I,J) from memory to disk.

VERT_REDUCE (BLOCK2,BLOCK1) is a subroutine that reduces BLOCK2 (which contains the values of the (J,I)th block of the matrix A, where J > I) using the values in BLOCK1 (which contains the reduced values of block (I,I) from the matrix A). BLOCK (I,I) will already be in LU format. BLOCK (J,I) will be transformed into a complete block of L, since (J,I) is vertically below (I,I).

HORIZ_REDUCE (BLOCK2,BLOCK1) is a subroutine that reduces BLOCK2 (which contains the block (I,J) of the matrix A, where J>I) using the factors stored in BLOCK1 (which contains block (I,I) of the matrix A). BLOCK (I,I) will already be in LU format. BLOCK (I,J) will be transformed into a complete block of U, since (I,J) is horizontally to the right of (I,I).

MULTIPLY(BLOCK1,BLOCK2) is a subroutine which produces a product block from the contents of BLOCK1 and BLOCK2. BLOCK1 will contain the product block.

TRI_REDUCE (BLOCK2,BLOCK1) is a subroutine which reduces BLOCK2 by adding the product block BLOCK1 to BLOCK2.

Basically, the algorithm can be summarized and analyzed by examining the Ith pass of the algorithm as follows:

Reduce block (I,I). Hence, the block on the diagonal is read and rewritten at the start of the Ith pass.

Reduce the remaining blocks on the Ith row. They are the blocks (I,I+1) thru (I,N). Therefore, these blocks are read and rewritten in order to be reduced in the Ith pass. These blocks will also be read once for each block beneath them in order to reduce the blocks beneath them during the Ith pass. As an example, block (1,3) is read when it is reduced by block (1,1) and again when it is used to reduce blocks (2,3), and read again to reduce (3,3), and read again to reduce (4,3),...., and read again to reduce (N,3).

Reduce the remaining blocks on the Ith column. They are blocks (I+1,I) thru (N,I). Therefore, these blocks are read and rewritten in order to be reduced in the Ith pass. These blocks will also be read several times later in this pass in order to reduce other blocks on the same row.

Reduce the remaining submatrix bounded by the (I+1)th row and column. These are the blocks (I+1,I+1) thru (I+1,N), (I+1,I+1) thru (N,I+1), (I+1,N) thru (N,N) and (N,I+1) thru (N,N). Therefore, these blocks will be read and rewritten during the Ith pass. As noted previously, in order to reduce these blocks, blocks from the Ith row and Ith column must be read into memory. The algorithm above reads in a block from the Ith row and Jth column and then proceeds to reduce the blocks in the Jth column from row I+1 thru row N using values read from the Ith column.

Since we are only allowing two square blocks in memory at once, we initially read two blocks, A(I,K) and A(K,J). One of these blocks becomes a product block. After the product block has been produced, we read in A(I,J), the block to be reduced, replacing the non-product block in memory. Then, block A(I,J) is reduced by the product block. This product block is specific to the particular block A(I,J). Therefore, we cannot use this product block for another block in that column. Thus, in order to reduce another block in that same column, we must read two blocks.

Hence, each of the non-diagonal blocks is read (N-1) times for the remaining submatrix and once for itself. Each diagonal block (J,J) is read and rewritten J times.

Using the general idea of the algorithm, we can derive a formula for the number of writes.

In the first pass, block (1,1) is used to reduce the entire first row and first column of the matrix. Then, the remaining blocks are reduced using the values in the first row and column of blocks. Therefore, in the first pass the entire N by N matrix is rewritten. Row 1 and column 1 are never again rewritten.

In the second pass, block (2,2) reduced the remaining N-1 blocks of the second row and the remaining N-1 blocks of the second column. Using the values of the second row and column, the remaining (N-2)*(N-2) blocks of the matrix are reduced and rewritten. Therefore, in the second pass (N-1)*(N-1) blocks are rewritten.

In the Ith pass, each block (K,L) with K>=I and L>=I is written once, so the Ith pass does (N+1-I)**2 writes. Therefore, the formula for the number of writes is:

$$\sum_{I=1}^{N} (N+1-I)^{2} = \sum_{I=1}^{N} I^{2}$$

After simplifying the summation, we have

$$\frac{2 N^{3} + 3 N^{2} + N}{6}$$

By using the above analysis, we can derive a formula for the number of reads:

Diagonal: Since the diagonal contains N blocks and each block (I,I) is read I times, we have

$$\sum_{I=1}^{N} I$$

Non-Diagonal: Since each block (I,J) is read N times and there are ((N**2) - N) non-diagonal blocks, we have

$$N * (N**2 - N) = N ** 3 - N ** 2$$

After simplifying and combining the summations, we have

$$\frac{2 * N ** 3 - N ** 2 + N}{2}$$

The write and read counts presented below by block verify the above analysis.

A matrix A has been divided in a 10 by 10 matrix of blocks. The read and write counts by block for its reduction into LU format are presented below.

WRITE COUNTS BY BLOCK

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

READ COUNTS BY BLOCK

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

Algorithm TWOSUB#2

After examining the data flow (i.e., the pattern of reads and writes) of Algorithm TWOSUB#1, it is clear that this algorithm has three parts. First, reduce block (I,I). Then, reduce the blocks on row I and column I using block (I,I). Then, reduce the remaining blocks using the blocks in the Ith row and column. Algorithm TWOSUB#1 reduces the remaining blocks by having the inner loops proceed from I+1 to N.

Reversing the inner loops to proceed from N to I+1 will result in the next diagonal block to be reduced being in memory when the outer loop begins. This will result in a savings of N-1 reads. Also, we can save the rewrite of the diagonal block in the inner loop since it will be reduced and rewritten at the beginning of the next pass. Thus, we save N-1 writes.

The PL/I main routine is:

```
CALL READ_BLOCK  (1,1,BLOCK1);

DO I = 1 TO NUM_SUBMAT_PER_COL;

   CALL AUTO_REDUCE (I,I,BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK (J,I,BLOCK2);
      CALL VERT_REDUCE(J,I,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK(J,I,BLOCK2);

      CALL READ_BLOCK (I,J,BLOCK2););
      CALL HORIZ_REDUCE(I,J,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK (I,J,BLOCK2);

   END;

   DO J = NUM_SUBMAT_PER_COL TO I+1 BY -1;

      DO K = NUM_SUBMAT_PER_ROW TO I+1 BY -1;

         CALL READ_BLOCK (I,K,BLOCK1);
         CALL READ_BLOCK (J,I,BLOCK2);
         CALL MULTIPLY (J,I,BLOCK2,I,K,BLOCK1);
         CALL READ_BLOCK (J,K,BLOCK1);
         CALL TRI_REDUCE (J,K,BLOCK1,J,K,BLOCK2);
         IF J ^= I+1 | K ^= I+1
            THEN CALL WRITE_BLOCK (J,K,BLOCK1);

      END;

   END;

END;
```

READ_BLOCK, VERT_REDUCE, HORIZ_REDUCE, MULTIPLY, WRITE_BLOCK, and TRI_REDUCE are the same subprograms as in algorithm TWOSUB#1.

Therefore, the read formula for algorithm TWOSUB#2 is derived by subtracting N-1 from algorithm TWOSUB#1's read formula.

$$\frac{2 * N ** 3 - N ** 2 + N}{2} - (N-1)$$

which is

$$\frac{2 * N ** 3 - N ** 2 - N + 2}{2}$$

Similiarly, the write formula for algorithm TWOSUB#2 is derived by subtracting N-1 from algorithm TWOSUB#1's write formula.

$$\frac{2 * N ** 3 + 3 * N ** 2 + N}{6} - (N-1)$$

which is

$$\frac{2 * N ** 3 + 3 * N ** 2 - 5 * N + 6}{6}$$

By comparing the read and write counts of algorithm TWOSUB#2 with the counts from algorithm TWOSUB#1, we see the savings of N-1 reads and write of the diagonal blocks.

ALGORITHM TWOSUB#2
WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	1	2	2	2	2	2	2	2	2
1	2	2	3	3	3	3	3	3	3
1	2	3	3	4	4	4	4	4	4
1	2	3	4	4	5	5	5	5	5
1	2	3	4	5	5	6	6	6	6
1	2	3	4	5	6	6	7	7	7
1	2	3	4	5	6	7	7	8	8
1	2	3	4	5	6	7	8	8	9
1	2	3	4	5	6	7	8	9	9

ALGORITHM TWOSJB#2
READ COUNTS BY BLOCK

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

ALGORITHM TWOSUB#1
WRITE COUNTS BY BLOCK

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

ALGORITHM TWOSUB#1
READ COUNTS BY B_OCK

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

Algorithm TWOSUB#3

Algorithms TWOSUB#1 and TWOSUB#2 reduced BLOCK (I,I) and then reduced the remaining blocks using the factors from BLOCK (I,I). Algorithm TWOSUB#3 will examine the idea of only reducing a column of blocks when one of its members is to be reduced.

After reducing BLOCK (1,1), BLOCKS (2,1) thru BLOCK (N,1) are reduced using the factors in BLOCK (1,1). This is the entire first pass.

Before reducing BLOCK (2,2), the factors stored in the blocks of column 1 must be applied. So BLOCK (1,2) is affected by BLOCK (1,1), BLOCK (2,2) by BLOCK (2,1), BLOCK (3,2) by BLOCK (3,1), BLOCK (4,2) by BLOCK (4,1), and BLOCK (N,2) is affected by BLOCK (N,1). Only then is BLOCK (2,2) reduced. Then, the remaining blocks in column 2 (BLOCKS (3,2), (4,2), thru (N,2)) are reduced using the factors in BLOCK (2,2).

Pass 3 begins with application of the L factors from the blocks in column 1 to the blocks in column 3 from rows 1 thru N. Then the L factors from column 2 are applied to column 3 (note that only rows 2 thru N need be applied). Finally, block (3,3) is reduced and blocks (4,3) thru (N,3) are reduced using the values in block (3,3).

The Nth column of blocks is untouched until the Nth pass where the factors from columns 1 thru N-1 are applied before the reduction of BLOCK (N,N).

Contrasting algorithms TWOSUB#3 and TWOSUB#1, the following ideas are apparent. Pass 1 of algorithm TWOSUB#1 passes thru the most blocks, while pass 1 of algorithm TWOSUB#3 passes thru the least blocks. Pass N of algorithm TWOSUB#1 passes thru the least blocks, while pass N of algorithm TWOSUB#3 passes thru the most blocks.

One would assume that the two algorithms should require the same amount of I/O. However, there is a slight difference.

In TWOSUB#1, the Ith diagonal block was read once in the Ith pass and used to reduce all the blocks in the Ith row and in the Ith column WITHOUT having to be re-read. In the (I+1)th pass, BLOCK (I,I) is no longer referenced.

However, TWOSUB#3 must reference BLOCK (I,I) in passes (I+1) thru N. Hence, TWOSUB#3 must read each diagonal block N times.

The writes are the same, since each block is rewritten after it is reduced. Therefore, the number of writes is

$$\frac{2 * N ** 3 + 3 * N ** 2 + N}{6}$$

Again, each non-diagonal block is read N times, once each time it is reduced and once each time it is used to reduce. But the diagonal elements are treated differently, Thus, each element is read N times, for a total of N ** 3 reads.

Here is the PL/I main routine for algorithm TWOSUB#3.

```
DO I = 1 TO NUM_SUBMAT_PER_COL;

   DO J = 1 TO I-1;

      CALL READ_BLOCK (J,J,BLOCK1);
      CALL READ_BLOCK (J,I,BLOCK2);
      CALL HORIZ_REDUCE (J,I,BLOCK2,J,J,BLOCK1);
      CALL WRITE_BLOCK (J,I,BLOCK2);

      DO K = J+1 TO NUM_SUBMAT_PER_ROW;

         CALL READ_BLOCK (J,I,BLOCK1);
         CALL READ_BLOCK (K,J,BLOCK2);
         CALL MULTIPLY (K,J,BLOCK2,J,I,BLOCK1);
         CALL READ_BLOCK (K,I,BLOCK1);
         CALL TRI_REDUCE (K,I,BLOCK1,K,I,BLOCK2);
         CALL WRITE_BLOCK (K,I,BLOCK1);

      END;

   END;


   CALL READ_BLOCK (I,I,BLOCK1);
   CALL AUTO_REDUCE (I,I,BLOCK1);
   CALL WRITE_BLOCK (I,I,BLOCK1);


   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK (J,I,BLOCK2);
      CALL VERT_REDUCE(J,I,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK (J,I,BLOCK2);

   END;

END;
```

Confirming our hypothesis that TWOSUB#3 is only different from TWOSUB#1 in its read counts for the diagonal, we find that the read and write counts by block for a ten by ten submatrix are identical to those presented in the discussion of algorithm TWOSUB#1, except for the diagonal.

WRITE COUNTS BY BLOCK

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

READ COUNTS BY BLOCK

10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10

Hence, the read count is N ** 3.

The write count is the same as TWOSUB#1 which is:

$$\frac{2 * N ** 3 \quad + \quad 3 * N ** 2 \quad + \quad N}{6}$$

where N = NUM_SUBMAT_PER_COL.

Algorithm TWOSUB#4

The fourth pass of algorithm TWOSUB#3 can be summarized as follows:

BLOCK (1,4) is reduced by BLOCK (1,1) and then BLOCK (1,4) is transformed into a product block (from BLOCK (2,1)) and then used to reduce BLOCK (2,4). BLOCK(1,4) is re-read and transformed into a product block (from BLOCK (3,1)) and used to reduce BLOCK (3,4). etc. BLOCK (1,4) is re-read and transformed into a product block (from BLOCK(N,1)) and used to reduce BLOCK (N,4).

BLOCK (2,4) is reduced by BLOCK (2,2) and then BLOCK (2,4) is transformed into a product block (from BLOCK (3,2)) and then used to reduce BLOCK (3,4). BLOCK(2,4) is re-read and transformed into a product block (from BLOCK (4,2)) and used to reduce BLOCK (4,4). etc. BLOCK (2,4) is re-read and transformed into a product block (from BLOCK(N,2)) and used to reduce BLOCK (N,4).

BLOCK (3,4) is reduced by BLOCK (3,3) and then BLOCK (3,4) is transformed into a product block (from BLOCK (4,3)) and then used to reduce BLOCK (4,4). BLOCK(3,4) is re-read and transformed into a product block (from BLOCK (5,3)) and used to reduce BLOCK (5,4), etc. BLOCK (3,4) is re-read and transformed into a product block (from BLOCK(N,3)) and used to reduce BLOCK (N,4).

BLOCK (4,4) is reduced by itself and is used to reduce BLOCK (5,4), BLOCK (6,4), thru BLOCK (N,4).

After examining the data flow (i.e., the pattern of reads and writes) of algorithm TWOSUB#3, it is clear that reversing the innermost loop to proceed from N to I+1 will result in some substantial savings.

By reversing the direction of one loop, the following data flow can be achieved:

BLOCK (1,4) is reduced by BLOCK (1,1) and then BLOCK (1,4) is transformed into a product block (from BLOCK (N,1)) and then used to reduce BLOCK (N,4). BLOCK(1,4) is re-read and transformed into a product block (from BLOCK (N-1,1)) and used to reduce BLOCK (N-1,4), etc. BLOCK (1,4) is re-read and transformed into a product block (from BLOCK(2,1)) and used to reduce BLOCK (2,4).

BLOCK (2,4) is reduced by BLOCK (2,2) and then BLOCK (2,4) is transformed into a product block (from BLOCK (N,2)) and then used to reduce BLOCK (N,4). BLOCK(2,4) is re-read and transformed into a product block (from BLOCK (N-1,2)) and used to reduce BLOCK (N-1,4), etc. BLOCK (2,4) is re-read and transformed into a product block (from BLOCK(3,2)) and used to reduce BLOCK (3,4).

BLOCK (3,4) is reduced by BLOCK (3,3) and then BLOCK (3,4) is transformed into a product block (from BLOCK (N,3)) and then used to reduce BLOCK (N,4). BLOCK(3,4) is re-read and transformed into a product block (from BLOCK (N-1,3)) and used to reduce BLOCK (N-1,4), etc. BLOCK (3,4) is re-read and transformed into a product block (from BLOCK(4,3)) and used to reduce BLOCK (4,4).

BLOCK (4,4) is reduced by itself and is used to reduce BLOCK (5,4), BLOCK (6,4), thru BLOCK (N,4).

Please note that after applying all the reductions from column 1 to the 4th column, BLOCK (2,4) is in memory and does not have to be written to disk and re-read. After applying all the reductions from column 2 to the 4th column, BLOCK (3,4) is in memory and does not have to be written and re-read again. BLOCK (4,4) is in memory at the end of the innermost loop and does not have to read again to commence the self reduction loop. Hence, in the fourth pass, algorithm TWOSUB#4 has saved 3 reads and 3 writes. In general, the kth pass of algorithm TWOSUB#4 does (k-1) less reads and (k-1) less writes than algorithm TWOSUB#3.

Here is the PL/I main routine for algorithm TWOSUB#4.

```
DO I = 1 TO NUM_SUBMAT_PER_COL;

   CALL READ_BLOCK (1,1,BLOCK2);


   DO J = 1 TO I-1;

      CALL READ_BLOCK (J,I,BLOCK1);
      CALL HORIZ_REDUCE (J,I,BLOCK1,J,J,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);

      DO K = NUM_SUBMAT_PER_ROW TO J+1 BY -1;

         CALL READ_BLOCK (K,J,BLOCK2);
         CALL MULTIPLY (K,J,BLOCK2,J,I,BLOCK1);
         CALL READ_BLOCK (K,I,BLOCK1);
         CALL TRI_REDUCE (K,I,BLOCK1,K,I,BLOCK2);
         IF K ^= J+1
            THEN DO;
                   CALL WRITE_BLOCK (K,I,BLOCK1);
                   CALL READ_BLOCK  (J,I,BLOCK1);
                 END;

      END;

      CALL READ_BLOCK (J+1,J+1,BLOCK2);

   END;

   CALL AUTO_REDUCE (I,I,BLOCK2);
   CALL WRITE_BLOCK (I,I,BLOCK2);


   DO J = I+1 TO NUM_SUBMAT_PER_ROW;

      CALL READ_BLOCK (J,I,BLOCK1);
      CALL VERT_REDUCE(J,I,BLOCK1,I,I,BLOCK2);
      CALL WRITE_BLOCK(J,I,BLOCK1);

   END;

END;
```

Since algorithm TWOSUB#4 is merely an improvement over algorithm TWOSUB#3, we can derive the read and write equations for algorithm TWOSUB#4 by subtracting the savings for algorithm TWOSUB#4 from the equations for algorithm TWOSUB#3. We save

$$\sum_{I=2}^{N} (I-1) = \sum_{K=1}^{K-1} K = \frac{N * (N-1)}{2}$$

reads and writes.

Hence, the read count is

$$N ** 3 - \frac{N ** 2 - N}{2}$$

which reduces to

$$\frac{2 * N ** 3 - N ** 2 + N}{2}$$

The write count is

$$\frac{4 * N ** 3 + 6 * N ** 2 + 2 * N}{12} - \frac{N ** 2 - N}{2}$$

which reduces to

$$\frac{4 * N ** 3 + 8 * N}{12}$$

where N = NUM_SUBMAT_PER_COL.

Confirming our assumption that TWOSUB#4 saves one read and one write per block in the upper triangle over TWOSUB#3, we present the following read and write counts:

ALGORITHM TWOSUB#4
WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	2	2	2	2	2	2	2	2	2
1	2	3	3	3	3	3	3	3	3
1	2	3	4	4	4	4	4	4	4
1	2	3	4	5	5	5	5	5	5
1	2	3	4	5	6	6	6	6	6
1	2	3	4	5	6	7	7	7	7
1	2	3	4	5	6	7	8	8	8
1	2	3	4	5	6	7	8	9	9

ALGORITHM TWOSUB#4
READ COUNTS BY BLOCK

10	9	9	9	9	9	9	9	9	9
10	10	9	9	9	9	9	9	9	9
10	10	10	9	9	9	9	9	9	9
10	10	10	10	9	9	9	9	9	9
10	10	10	10	10	9	9	9	9	9
10	10	10	10	10	10	9	9	9	9
10	10	10	10	10	10	10	9	9	9
10	10	10	10	10	10	10	10	9	9
10	10	10	10	10	10	10	10	10	9
10	10	10	10	10	10	10	10	10	10

ALGORITHM TWOSUB#3
WRITE COUNTS BY BLOCK

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

ALGORITHM TWOSUB#3
READ COUNTS BY BLOCK

10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10

Algorithm TWOSUB#5

Algorithms TWOSUB#3 and TWOSUB#4 have two distinct phases:

the application to column L of the reductions made to columns 1, 2, ..., L-1

the reduction of column L

When algorithm TWOSUB#3 reduced block (3,4), block (3,4) was rewritten after being reduced by block (3,1), was again rewritten after the reduction by block (3,2), and was finally rewritten after being reduced by block (3,3). If a block were only rewritten to disk after being totally reduced, this would lead to tremendous savings in total I/O.

In the discussion of three square blocks, algorithms THREESUB#4 and THREESUB#5 were similiar in strategy to TWOSUB#3 and TWOSUB#4. Algorithm THREESUB#6 was then developed to reduce the number of writes, by doing all the reductions from columns 1 thru L-1 in one pass. Therefore, BLOCK (3,4) was written once after being reduced by BLOCKs (3,1) and (3,2) and rewritten again after being reduced by BLOCK (3,3). By using this strategy, algorithm THREESUB#6 eliminated the N**3 term from its write formula, an impressive savings.

However, this savings is NOT realized under the two square block scheme. In essence, algorithm THREESUB#6 held BLOCK (3,4) in memory while applying BLOCK(3,1)*BLOCK(1,4) and BLOCK(3,2)*BLOCK(2,4). With the two square block scheme, we cannot hold BLOCK (3,4) in memory since we need two blocks in order to form the product block.

Algorithm TWOSUB#5 is a blend of ideas developed in algorithms THREESUB#6 thru THREESUB#10. It is most similiar to algorithms THREESUB#6 and THREESUB#10, with the following important differences.

In order to save unneceesary reads and writes in column 2, a separate loop is used for column 2 (similiar to THREESUB#9).

In order to save unnecessary reads of BLOCK (1,1), BLOCKs (1,N), (1,N-1), (1,N-2), thru (1,2) are reduced outside of the main loop while BLOCK (1,1) is in memory (although this violates the idea that column L is untouched until pass L).

Here is the PL/I routine:

```
/* column 1 */

CALL READ_BLOCK  (1,1,BLOCK1);
CALL AUTO_REDUCE (1,1,BLOCK1);
CALL WRITE_BLOCK (1,1,BLOCK1);

DO J = 2 TO NUM_SUBMAT_PER_ROW;

   CALL READ_BLOCK (J,1,BLOCK2);
   CALL VERT_REDUCE(J,1,BLOCK2,1,1,BLOCK1);
   CALL WRITE_BLOCK(J,1,BLOCK2);

END;


/* column 2 and row 1 */

IF NUM_SUBMAT_PER_ROW > 1
   THEN DO;

          DO J = NUM_SUBMAT_PER_ROW TO 2 BY -1;

             CALL READ_BLOCK (1,J,BLOCK2);
             CALL HORIZ_REDUCE (1,J,BLOCK2,1,1,BLOCK1);
             CALL WRITE_BLOCK (1,J,BLOCK2);

          END;

          DO J = 2 TO NUM_SUBMAT_PER_COL;

             CALL READ_BLOCK (J,1,BLOCK1);
             CALL MULTIPLY (J,1,BLOCK1,1,2,BLOCK2);
             CALL READ_BLOCK (J,2,BLOCK2);
             CALL TRI_REDUCE (J,2,BLOCK2,J,2,BLOCK1);
             CALL WRITE_BLOCK (J,2,BLOCK2);
             IF J ^= NUM_SUBMAT_PER_COL
                THEN CALL READ_BLOCK (1,2,BLOCK2);

          END;

       END;
```

```
/* columns 3 thru N starting in row 2 */
/* (since row 1 was done above)     */

DO I = 3 TO NUM_SUBMAT_PER_ROW;
   DO J = 2 TO I-1;
      DO K = 1 TO J-1;
         CALL READ_BLOCK (K,I,BLOCK1);
         CALL READ_BLOCK (J,K,BLOCK2);
         CALL MULTIPLY   (J,K,BLOCK2,K,I,BLOCK1);
         CALL READ_BLOCK (J,I,BLOCK1);
         CALL TRI_REDUCE (J,I,BLOCK1,J,I,BLOCK2);
         IF K ^= J-1
            THEN CALL WRITE_BLOCK (J,I,BLOCK1);
      END;
      CALL READ_BLOCK (J,J,BLOCK2);
      CALL HORIZ_REDUCE(J,I,BLOCK1,J,J,BLOCK2);
      CALL WRITE_BLOCK (J,I,BLOCK1);
   END;

   DO J = 1 TO I-2;
      CALL READ_BLOCK (I,J,BLOCK1);
      CALL READ_BLOCK (J,I,BLOCK2);
      CALL MULTIPLY (I,J,BLOCK1,J,I,BLOCK2);
      CALL READ_BLOCK (I,I,BLOCK2);
      CALL TRI_REDUCE (I,I,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK (I,I,BLOCK2);
   END;

   CALL READ_BLOCK (I,J,BLOCK1);
   CALL READ_BLOCK (J,I,BLOCK2);
   CALL MULTIPLY (I,J,BLOCK1,J,I,BLOCK2);
   CALL READ_BLOCK (I,I,BLOCK2);
   CALL TRI_REDUCE (I,I,BLOCK2,I,I,BLOCK1);
   CALL AUTO_REDUCE(I,I,BLOCK2);
   CALL WRITE_BLOCK(I,I,BLOCK2);

   DO J = I+1 TO NUM_SUBMAT_PER_ROW;
      DO K = 1 TO I-1;
         CALL READ_BLOCK (J,K,BLOCK1);
         CALL READ_BLOCK (K,I,BLOCK2);
         CALL MULTIPLY   (J,K,BLOCK1,K,I,BLOCK2);
         CALL READ_BLOCK (J,I,BLOCK2);
         CALL TRI_REDUCE (J,I,BLOCK2,J,I,BLOCK1);
         IF K ^= I-1
            THEN CALL WRITE_BLOCK(J,I,BLOCK2);
      END;
      CALL READ_BLOCK (I,I,BLOCK1);
      CALL VERT_REDUCE (J,I,BLOCK2,I,I,BLOCK1);
      CALL WRITE_BLOCK (J,I,BLOCK2);
   END;
END;
```

By looking at the looping patterns of algorithm TWOSUB#5, we can derive the read formula in the following manner:

Initially, BLOCK (1,1) is read and is used to reduce ROW#1 and COL#1. Hence, it is read once.

Each of the blocks in ROW#1 is read once when being reduced by BLOCK (1,1) and once again to reduce each of the N-1 blocks beneath it. BLOCK (1,2) is the only exception, since it is in memory after being reduced by (1,1) and is available to help reduce (2,2) without being reread. Hence, BLOCK (1,2) is read N-1 times, while each one of blocks (1,3), (1,4), (1,5), thru (1,N) is read N times. So we have (N-2) blocks being read N times, and one block being read (N-1) times.

Each of the blocks in COL#1 (except for (1,1)) is read N times, once when it is reduced by (1,1) and once for each of the blocks in its row. That is, BLOCK (5,1) is read once when it is reduced by (1,1). BLOCK (5,1) is read to help reduce (5,2), read again to help reduce (5,3), read again for (5,4), ..., (5,N). So, BLOCK (5,1) is read N times. Therefore, we have (N-1) blocks being read N times.

So, for COL#1 and ROW#1, we have

BLOCK (1,1)	1	read
BLOCK (1,2)	N - 1	reads
BLOCKs(1,3) thru (1,N)	N * (N - 2)	reads
BLOCKs(2,1) thru (N,1)	N * (N - 1)	reads

which is a total of

2 * N ** 2 - 2 * N reads

For the remaining N-1 blocks of COL#2, we see each block being read once when it is reduced by the factors stored in COL#1 and once to help reduce each block to the right of it in the matrix (i.e., (K,3), (K,4), thru (K,N)). So each of these (N-1) blocks is read (N-1) times.

For the remaining N-2 blocks of ROW#2, we see each block being read once when it is reduced and being read once to help reduce each of the N-2 blocks beneath it. That is, BLOCK (2,5) is read once when it is reduced by the product of blocks (2,1) and (1,5) and then by (2,2). Then BLOCK (2,5) is read once to reduce (3,5), once for (4,5), once for (5,5), (6,5), thru (N,5). So, each of the remaining N-2 blocks in ROW#2 is read N-1 times.

For COL#2 and ROW#2, we have

COL#2	(N - 1) * (N - 1)	reads
ROW#2	(N - 1) * (N - 2)	reads

which is a total of

2 * N ** 2 - 5 * N + 3 reads.

For the elements on the diagonal from columns 3 thru N, we can see that BLOCK (J,J) is read J-1 times when it is reduced, once when it is used to reduce each block beneath it, and once to reduce each block to the right of it.

> For example (assuming N=10), BLOCK (5,5) is read four times when it is being reduced. Initially (5,5) is read when the product of (5,1) and (1,5) is applied to it. Then it is read again when the product of (5,2) and (2,5) is applied. It is read again when the product of (5,3) and (3,5) is applied. (5,5) is read again when the product of (5,4) and (4,5) is applied. It remains in memory (and is not reread) when it is reduced. So, block (5,5) was read 4 times in the process of reducing it.
>
> In pass 5, BLOCK (5,5) is read once to reduce (6,5), again for (7,5), again for (8,5), again for (9,5), and again for (10,5). So in pass 5, BLOCK (5,5) is read 5 times.
>
> In passes 6 thru N, BLOCK (5,5) is read once for (5,6), once for (5,7), once for (5,8), once for (5,9), and once for (5,10). So in each of passes 6 thru N, BLOCK (5,5) is read once for a total of 5 times.

In general, BLOCK (I,I) (3 <= I <= N) is read I-1 times when it is being reduced. In pass I, it is read N-I times for each of the blocks beneath it in the Ith column. In passes I+1 thru N, it is read once for each block to the right of it in the matrix for a total of N-I times. So, we have

$$\sum_{I=3}^{N} ((N-I) + (N-I) + (I-1))$$

which is

$$2 * N^{2} - 5 * N + 3 - \frac{N^{2} + N - 6}{2}$$

For each of the non-diagonal blocks, excluding rows 1 and 2 and columns 1 and 2, we can see each block being read N-1 times. Examining BLOCK (I,J), we see (I,J) being read I-1 times when it is being reduced and N-I times to reduce blocks in the Jth column beneath (I,J).

> For example (assuming N=10), BLOCK (6,10) is read 5 times when it is being reduced and 4 times to reduce blocks (7,10), (8,10), (9,10), and (10,10). Initially, BLOCK (6,10) is read when the product of (6,1) and (1,10) id applied to it. It is read again for (6,2)*(2,10). It is read again for (6,3)*(3,10). It is read again for (6,4)*(4,10). It is read again for (6,5)*(5,10) and held in memory when (6,6) is applied to it (without having to be reread). So, BLOCK (6,10) is read 5 times.
>
> After BLOCK (6,10) has been fully reduced, it is read once to reduce (7,10), once for (8,10), once for (9,10), and once for (10,10). So, BLOCK (6,10) is read 4 times.

So, we have (N-I) + (I-1) reads for each non-diagonal block (excluding columns 1 and 2 and rows 1 and 2). Since, (N-I) + (I-1) = N-1, we can rephrase the above statement. Each block (I,J), I ^= J, I > 2, J > 2, is read N-1 times. Since, there are N-3 blocks in each of columns 2 thru N satisfying I ^= J, I > 2, J > 2, each being read N-1 times, we have (N-3)*(N-2)*(N-1) reads which is

N ** 3 - 6 * N ** 2 + 11 * N - 6 reads.

Combining the read formulae derived, we have

$$\frac{2 * N ** 3 - N ** 2 - 3 * N + 6}{2}$$

The derivation of the write formula is done in a similiar manner.

Each of the blocks in column 1 is fully reduced in one pass and is written once. So, we have N writes.

Each of the blocks in row 1 (except (1,1) which was done in the first loop) is also fully reduced in the second loop and is written once. So, we have N-1 writes.

Each of the blocks in column 2 is also fully reduced in the second pass (except for (1,2) which was done together with the rest of row 1) and is written once. So, we have N-1 writes.

Each block (I,J) above the diagonal (I>1, J>2, I<J) is written I-1 times.

> For example, examine block (4,6). (4,6) must be written after (4,1)*(1,6) is applied to it. (4,6) must be written again after the application of (4,2)*(2,6). Finally, (4,6) is reduced by (4,3)*(3,6), remaining in memory while being reduced by (4,4), and is then rewritten for the last time. So, (4,6) is written 3 times.

So, we have

$$\sum_{J=3}^{N} \sum_{I=2}^{J-1} (I-1) = \sum_{J=3}^{N} \sum_{K=1}^{J-2} K$$

which is

$$\frac{N ** 3 - 3 * N ** 2 + 2 * N}{6}$$

Each block on the diagonal (I,I), starting from (3,3) thru (N,N), is written I-1 times.

> For example, examine (5,5). It is written after the application of (5,1)*(1,5). It is written again after the application of (5,2)*(2,5). It is written again after (5,3)*(3,5). However, it is NOT rewritten after the application of (5,4)*(4,5). (5,5) is held in memory while it is being reduced. Then, it is written. So, we have 4 writes.

So, we have

$$\sum_{I=3}^{N} (I-1) = \sum_{K=2}^{N-1} K = \frac{N ** 2 - N - 2}{2}$$

For the blocks (I,J) beneath the diagonal, (I>1, J>2, I>J), each block is written J-1 times. BLJCK (I,J) is written once when each factor from columns 1 tnru J-2 is applied to it. (I,J) remains in memory after the factor from column J-1 is applied to it. Then (J,J) is applied to (I,J). Then (I,J) is written. So, (I,J) is written J-1 times.

So, we have

```
     N         I-1                      N         I-2
  *******    ********                *******    ********
   *          *                       *          *
    *          *   (J-1)     =         *          *     K
   *          *                       *          *
  *******    ********                *******    ********
    I=3        J=3                     I=3        K=2
```

which is

```
N ** 3 - 3 * N ** 2 - 4 * N + 12
--------------------------------
               6
```

NOTE: If I > J and J > 2, then I > 2. Hence, the lower bound of the outer sum is 3.

Combining the write subformulae derived, we have

```
 2 * N ** 3  -  3 * N ** 2  + 13 * N  -  6
--------------------------------------------
                        6
```

Confirming our analysis above, here are the read and write counts by block for a matrix divided into a ten by ten submatrix of blocks:

WRITE COUNTS BY BLOCK

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	2	2	2	2	2	2	2	2
1	1	2	3	3	3	3	3	3	3
1	1	2	3	4	4	4	4	4	4
1	1	2	3	4	5	5	5	5	5
1	1	2	3	4	5	6	6	6	6
1	1	2	3	4	5	6	7	7	7
1	1	2	3	4	5	6	7	8	8
1	1	2	3	4	5	6	7	8	9

READ COUNTS BY BLOCK

1	9	10	10	10	10	10	10	10	10
10	9	9	9	9	9	9	9	9	9
10	9	16	9	9	9	9	9	9	9
10	9	9	15	9	9	9	9	9	9
10	9	9	9	14	9	9	9	9	9
10	9	9	9	9	13	9	9	9	9
10	9	9	9	9	9	12	9	9	9
10	9	9	9	9	9	9	11	9	9
10	9	9	9	9	9	9	9	10	9
10	9	9	9	9	9	9	9	9	9

Comparison of Algorithms TWOSUB#1 thru TWOSUB#5

It is time to stop and compare these algorithms.

Algorithms TWOSUB#1 thru TWOSUB#5 use the submatrix storage method. This method requires that two blocks must be able to fit in memory at once. In fact, the size of a block is chosen with memory in mind. Let us compare these algorithms.

Even though the read and write count equations are given in the sections describing each algorithm, they are repeated here for comparison purposes using the same denominator.

Algorithm	Number of Reads	Number of Writes
TWOSUB#1	$\frac{6N^3 - 3N^2 + 3N}{6}$	$\frac{2N^3 + 3N^2 + N}{6}$
TWOSUB#2	$\frac{6N^3 - 3N^2 - 3N + 6}{6}$	$\frac{2N^3 + 3N^2 - 5N + 6}{6}$
TWOSUB#3	$\frac{6N^3}{6}$	$\frac{2N^3 + 3N^2 + N}{6}$
TWOSUB#4	$\frac{6N^3 - 3N^2 + 3N}{6}$	$\frac{2N^3 + 4N}{6}$
TWOSUB#5	$\frac{6N^3 - 3N^2 - 9N + 12}{6}$	$\frac{2N^3 - 3N^2 + 13N - 6}{6}$

A table showing total I/O operations is presented below:

algorithm	total I/O operations
TWOSUB#1	$\frac{8*N**3 + 4*N}{6}$
TWOSUB#2	$\frac{8*N**3 - 9*N + 12}{6}$
TWOSUB#3	$\frac{8*N**3 + 3*N**2 + N}{6}$
TWOSUB#4	$\frac{8*N**3 - 3*N**2 + 7*N}{6}$
TWOSUB#5	$\frac{8*N**3 - 6*N**2 + 4*N + 6}{6}$

As can be seen from the formulas above, algorithm TWOSUB#5 does the least number of I/O operations.

A table listing the read and write counts for each algorithm for three sample matrices will reconfirm the advantages of algorithm TWOSUB#5.

algorithm	matrix divided into 5 by 5 submatrices			matrix divided into 10 by 10 submatrices			matrix divided into 20 by 20 submatrices		
	read	write	total	read	write	total	read	write	total
TWOSUB#1	115	55	170	955	385	1340	7810	2870	10680
TWOSUB#2	111	51	162	946	376	1322	7791	2851	10642
TWOSUB#3	125	55	180	1000	385	1385	8000	2870	10870
TWOSUB#4	115	45	160	955	340	1295	7810	2680	10490
TWOSUB#5	107	39	146	937	304	1241	7772	2509	10281

As can be seen, algorithm TWOSUB#5 is the 'best' submatrix decomposition without interchanges algorithm that uses two blocks in memory at once.

Comparison of Algorithms TWOSUB#5 and THREESUB#10

It is not as easy to compare algorithm TWOSUB#5 with algorithm THREESUB#10 as it was to compare TWOSUB#1 thru TWOSUB#5. Algorithm THREESUB#10 uses submatrix storage that requires the blocks to be small enough to enable three blocks to be in memory at once. Although TWOSUB#5 also uses submatrix storage, its blocks are larger since this method only requires that two blocks be in memory at once.

We can compare the two algorithms mathematically using the following argument:

> Assume the matrix has n elements per row and n elements per column.
>
> Assume memory can hold m elements of the matrix at once.

If we adopt the version of the submatrix storage scheme used in algorithm THREESUB#10, the following statements follow:

> The m elements of memory must be divided into three, since this scheme requires the presence in memory of three subblocks simultaneously. Each submatrix is square, having x elements per submatrix row and x elements per submatrix column. Hence, x is approximately the square root of (m/3). So, memory can hold three subblocks at once or 3*x**2 elements.
>
> Since a subblock contains x elements per submatrix row, Q subblocks will be needed to cover a matrix row. Hence, the matrix can be viewed as being a Q by Q matrix of subblocks. Furthermore, the matrix can also be regarded as a Qx by Qx matrix of elements.

```
memory = 3 * x ** 2        elements
matrix = Qx by Qx          elements
```

If we adopt the version of the submatrix storage scheme used in algorithm TWOSUB#5, the following statements follow:

> The m elements of memory must be divided into two, since this scheme requires the presence in memory of two subblocks simultaneously. Each submatrix is square, having y elements per submatrix row and y elements per submatrix column. Hence, y is approximately the square root of (m/2). So, memory can hold two subblocks at once or 2*y**2 elements.
>
> Since a subblock contains y elements per submatrix row, T subblocks will be needed to cover a matrix row. Hence, the matrix can be viewed as being a T by T matrix of subblocks. Furthermore, the matrix can also be regarded as a Ty by Ty matrix of elements.

```
memory = 2 * y ** 2        elements
matrix = Ty by Ty          elements
```

Algorithm THREESUB#10	Algorithm TWOSUB#5
memory = 3 * x ** 2	memory = 2 * y ** 2
matrix = Qx * Qx	matrix = Ty * Ty

We note that: $3 * x ** 2 = 2 * y ** 2$

Solving for x: $x = SQRT (2./3.) * y$

Approximating SQRT (2./3.) by .815, we have

$$x = .815 * y$$

We also note that: $Qx = Ty$

Substituting for x: $Q * .815 * y = T * y$

Dividing by y: $Q * .815 = T$

Now, T is the same as the N used for the number of subblocks in the read and write formulas for algorithm TWOSUB#5.

Now, Q is the same as the N used for the number of subblocks in the read and write formulas for algorithm THREESUB#10.

Rewriting the formulas for algorithms THRE=SUB#10 and TWOSUB#5 in terms of Q and T, we have:

```
THREESUB#10   4 * Q ** 3 + 12 * Q ** 2 - 22 * Q + 18
              --------------------------------------
                                6

TWOSUB#5      8 * T ** 3 -  6 * T ** 2 +  4 * T +  6
              --------------------------------------
                                6
```

Converting the TWOSUB#5 formula into terms of Q, we have:

```
8*(.815*Q) ** 3 -  6*(.815*Q) ** 2 +  4*(.815*Q) +  6
-----------------------------------------------------
                         6
```

which is

```
   4.328 * Q ** 3 - 3.984 * Q ** 2 + 3.260 * Q + 6
   ----------------------------------------------
                         6
```

Assymptotically, THREESUB#10 is better, because the coefficient of Q**3 is slightly smaller, but for small (or even most reasonable) values of Q, the comparison will depend on how well the algorithms use the available memory.

For Q=1, the entire matrix fits into memory at once, and both methods use 2 I/O operations, 1 read and 1 write.

If Q=2, the entire matrix will not fit into memory, so T is also 2. Then THREESUB#10 uses 9 I/O operations and so does TWOSUB#5.

Now suppose Q = 3, so THREESUB#10 splits the matrix into 9 blocks. Since 3 blocks must be in memory at once, memory will hold 1/3 of the matrix but not 3/4 of the matrix. Then, depending on the size of memory, T can either be 2 or 3. If memory will hold 1/2 of the matrix, T is 2, and TWOSUB#5 does 9 I/O operations. If memory does not hold 1/2 of the matrix, T is 3, and TWOSUB#5 does 30 I/O operations.

We obtain the following table for small Q:

Q value	total I/O for THREESUB#10	fraction of matrix in memory lower bound		upper bound	T values from	to	total I/O TWOSUB#5 from	to
1	2	1	to	1	1	1	2 to 2	
2	9	3/4	to	1	2	2	9 to 9	
3	28	1/3	to	3/4	2	3	9 to 30	
4	63	3/16	to	1/3	3	4	30 to 73	
5	118	3/25	to	3/16	4	5	73 to 146	
6	197	3/36	to	3/25	5	5	146 to 146	
7	304	3/49	to	3/36	5	6	146 to 257	

As can be seen, even for small Q, THREESUB#10 can do less I/O than TWOSUB#5. The degree to which each storage method utilizes the available memory is the deciding factor.

To obtain a better comparison of the I/O count for TWOSUB#5 with that of THREESUB#10, we note that for a given value of Q in THREESUB#10, the corresponding value of T for TWOSUB#5 lies between

CEIL (.815 * (Q-1)) and CEIL (.815 * Q).

This gives us the following table of values:

Q VALUE	THREESUB#10 I/O COUNTS	T VALUES FROM	T VALUES TO	TWOSUB#5 I/O COUNTS FROM	TWOSUB#5 I/O COUNTS TO
3	28	2	3	9	30
4	63	3	4	30	73
5	118	4	5	73	146
6	197	5	5	146	146
7	304	5	6	146	257
8	443	6	7	257	414
9	618	7	8	414	625
10	833	8	9	625	898
11	1092	9	9	898	898
12	1399	9	10	898	1241
13	1758	10	11	1241	1662
14	2173	11	12	1662	2169
15	2648	12	13	2169	2770
16	3187	13	14	2770	3473
17	3794	14	14	3473	3473
18	4473	14	15	3473	4286
19	5228	15	16	4286	5217
20	6063	16	17	5217	6274
21	6982	17	18	6274	7465
22	7989	18	18	7465	7465
23	9088	18	19	7465	8798
24	10283	19	20	8798	10281
25	11578	20	21	10281	11922
26	12977	21	22	11922	13729
27	14484	22	23	13729	15710
28	16103	23	23	15710	15710
29	17838	23	24	15710	17873
30	19693	24	25	17873	20226
31	21672	25	26	20226	22777
32	23779	26	27	22777	25534
33	26018	27	27	25534	25534
34	28393	27	28	25534	28505
35	30908	28	29	28505	31698
36	33567	29	30	31698	35121
37	36374	30	31	35121	38782
38	39333	31	31	38782	38782
39	42448	31	32	38782	42689
40	45723	32	33	42689	46850
41	49162	33	34	46850	51273
42	52769	34	35	51273	55966
43	56548	35	36	55966	60937
44	60503	36	36	60937	60937
45	64638	36	37	60937	66194
46	68957	37	38	66194	71745
47	73464	38	39	71745	77598
48	78163	39	40	77598	83761

Q VALUE	THREESUB#10 I/O COUNTS	T VALUES FROM	T VALUES TO	TWOSUB#5 I/O COUNTS FROM	TWOSUB#5 I/O COUNTS TO
51	93452	41	42	90242	97049
52	98959	42	43	97049	104190
53	104678	43	44	104190	111673
54	110613	44	45	111673	119506
55	116768	45	45	119506	119506
56	123147	45	46	119506	127697
57	129754	46	47	127697	136254
58	136593	47	48	136254	145185
59	143668	48	49	145185	154498
60	150983	49	49	154498	154498
61	158542	49	50	154498	164201
62	166349	50	51	164201	174302
63	174408	51	52	174302	184809
64	182723	52	53	184809	195730
65	191298	53	53	195730	195730
66	200137	53	54	195730	207073
67	209244	54	55	207073	218846
68	218623	55	56	218846	231057
69	228278	56	57	231057	243714
70	238213	57	58	243714	256825
71	248432	58	58	256825	256825
72	258939	58	59	256825	270398
73	269738	59	60	270398	284441
74	280833	60	61	284441	298962
75	292228	61	62	298962	313969
76	303927	62	62	313969	313969
77	315934	62	63	313969	329470
78	328253	63	64	329470	345473
79	340888	64	65	345473	361986
80	353843	65	66	361986	379017
81	367122	66	67	379017	396574
82	380729	67	67	396574	396574
83	394668	67	68	396574	414665
84	408943	68	69	414665	433298
85	423558	69	70	433298	452481
86	438517	70	71	452481	472222
87	453824	71	71	472222	472222
88	469483	71	72	472222	492529
89	485498	72	73	492529	513410
90	501873	73	74	513410	534873
91	518612	74	75	534873	556926
92	535719	75	75	556926	556926
93	553198	75	76	556926	579577
94	571053	76	77	579577	602834

Q VALUE	THREESUB#10 I/O COUNTS	T VALUES FROM	T VALUES TO	TWOSUB#5 I/O COUNTS FROM	TWOSUB#5 I/O COUNTS TO
100	6.863030E+05	81	82	7.020820E+05	7.4890E+05
200	5.412603E+06	163	163	5.747870E+06	5.7370E+06
300	1.817890E+07	244	245	1.930967E+07	1.4831E+07
400	4.298520E+07	326	326	4.608858E+07	4.8358E+07
500	8.383150E+07	407	408	8.972681E+07	9.9023E+07
1000	6.686630E+08	815	815	7.211275E+08	7.1275E+08
1500	2.254495E+09	1222	1223	2.431565E+09	2.7541E+09
2000	5.341326E+09	1630	1630	5.771674E+09	5.1674E+09
2500	1.042916E+10	2037	2038	1.126554E+10	1.8214E+10

By inspecting the table, we find that for all Q >= 78, THREESUB#10 does less I/O than TWOSUB#5. For Q <=77, sometimes THREESUB#10 is better, sometimes TWOSUB#5 is better.

To prove that THREESUB#10 is always better than TWOSUB#5 for values >= 78, we ignore the use of the ceiling function in the lower bound for T, so we use

T = .815 * (@ - 1)

Then, a lower bound for the I/O count for TWOSUB#5 is

$$\frac{4.328 * (@-1) ** 3 - 3.984 * (@-1) ** 2 + 3.260 * (@-1) + 6}{6}$$

Upon comparing the I/O formula for THREESUB#10 with this formula, we find that for @ >= 87, THREESUB#10 does less I/O than the lower bound on what TWOSUB#5 does. Therefore, we know that THREESUB#10 is better for @ >= 87, so we don't have to consider @ greater than 100, which are not shown in this table. Then this table, which takes the ceiling into account, shows that THREESUB#10 is better for all @ >= 78.

Comparison of Algorithms COL#1 and TWOSUB#5

It is not as easy to compare algorithms COL#1 and TWOSUB#5 as it was to compare TWOSUB#1 thru TWOSUB#5 or ROW#1 thru ROW#3 or ROW#3 versus COL#1. Algorithm TWOSUB#5 uses submatrix storage and requires the blocks to be small enough to enable two blocks in memory at once. Algorithm COL#1 uses column storage and requires the blocks to contain complete columns and be small enough to allow two blocks in memory at once.

We can use the following mathematical arguments:

Assume the matrix has n elements per row and n elements per column.

Assume memory can hold m elements of the matrix at once.

Since both TWOSUB#5 and COL#1 require two blocks to be in memory, in each case the block contains about m/2 elements. That is, the blocks are approximately the same size. Since TWOSUB#5 uses Q**2 blocks, while COL#1 uses T blocks, we see that

Q ** 2 = T

if we ignore the fact that Q and T must be integers.

Now, T is the same as the N used for the number of subblocks used in the algorithm COL#1 read and write count formulas.

Now, Q is the same as the N used for the number of subblocks used in the algorithm TWOSUB#5 read and write count formulas.

Rewriting the total I/O formulas for algorithms TWOSUB#5 and COL#1 in terms of Q and T, we have:

COL#1

$$\frac{T ** 2 + 3 * T}{2}$$

TWOSUB#5

$$\frac{8 * Q ** 3 - 6 * Q ** 2 + 4 * Q + 6}{6}$$

Converting the formulas for algorithm COL#1 into terms of Q, we have:

COL#1

$$\frac{Q ** 4 + 3 * Q ** 2}{2}$$

It is immediately apparent that the Q ** 4 term in in the formula for algorithm COL#1 will dominate.

When Q=1, the entire matrix fits into memory, so both COL#1 and TWOSUB#5 do one read and one write.

When Q=2, the entire matrix does not fit into memory. When Q=2, there are 4 blocks, where two blocks fit into memory at once. So, memory is large enough to fit 1/2 of the matrix, but not the entire matrix. Therefore, T may be 3 or 4 (T cannot be 2, because if T=2, then the entire matrix would fit into memory). If memory can hold 2/3 of the matrix, then T = 3 and COL#1 does 6 I/O operations. If memory holds less than 2/3 of the matrix, then T=4 and COL#1 does 14 I/O operations.

Now suppose Q=3; then there are 9 blocks. Since, two blocks are in memory at once, memory must be large enough to contain 2/9 of the matrix, but not big enough to contain 1/2 the matrix. Therefore, T ranges from 4 to 9. If T=4, COL#1 does 14 I/O operations. If T=9, COL#1 does 54 I/O operations.

We obtain the following table for small Q.

Q value	total I/O for TWOSUB#5	fraction of matrix in memory lower bound		fraction of matrix in memory upper bound	T values from	T values to	total I/O for COL#1 from		total I/O for COL#1 to
1	2	1	to	1	1	1	2	to	2
2	9	1/2	to	1	3	4	9	to	14
3	30	2/9	to	1/2	4	9	14	to	54
4	73	2/16	to	2/9	9	16	54	to	152
5	146	2/25	to	2/16	16	25	152	to	350
6	257	2/36	to	2/25	25	36	350	to	702

As can be seen from the table above, if Q >= 5, then TWOSUB#5 is always the better algorithm. In the cases where Q < 5, it depends on the number of blocks COL#1 will need. At these small values of Q, memory utilization plays the deciding role.

Chapter 6

Solving for x

Until this point, we have analyzed only the decomposition of a matrix A into LU factors. This is only part of the solution of Ax=b. After decomposing A, the L factors must be applied to any given right hand side of values (known as b). Then, we solve for x using the U factors and b.

Applying the Factors to b

The arithmetic operations involved in applying the factors to b can be summarized as:

$$b(j) = b(j) - m(j,k) * b(k)$$

for k = 1,2,3,....,(j-1)

Solving for x using the U Factors and b

The arithmetic operations involved in solving for x using the U factors can be summarized as:

$$x(j) = \frac{b(j) - \sum_{k=j+1}^{n} m(j,k) * x(k)}{m(j,j)}$$

for j = n, n-1, n-2, ..., 3, 2, 1

Row Storage

For the row storage method, this phase is quite simple. In the row storage method, several complete rows are stored together. Hence, each block of A will contain at least one row. Since the size of each block was determined to allow two blocks in memory at once, there is enough room in memory for one block of A and the entire set of b. Since b(n) is affected by b(1), it makes sense to store all the b's together and keep the b's in memory for the duration of this phase. Also, all the factors for a given b(k) are in one block of A. Therefore, if each block of A contains m complete rows, then block (k) will have all the factors necessary for b((k-1)*m+1) thru b(k*m).

Note that we must apply the L factors to b(1), then b(2), thru b(n).

After applying the L factors, we reverse the process and solve for x(n), x(n-1), x(n-2), thru x(1) using the U values.

Here is the algorithm:

```
Read in the b vector

DO I = 1 TO NUM_SUBMAT_PER_COL;
   CALL READ_BLOCK (I,BLOCK1);
   CALL APPLY_MULT (I,BLOCK1);
END;

CALL SOLVE_X (N,BLOCK1);

DO I = N-1 TO 1 BY -1;
   CALL READ_BLOCK (I,BLOCK1);
   CALL SOLVE_X (I,BLOCK1);
END;

Write out the b vector
```

There are only 2*N-1 reads of the matrix and one read and write for the vector b.

Note: We read most of the matrix in twice. In the multiplier loop, many U values were read in. In the solution phase, many L values were read in. Only BLOCK (N) was not read twice, since it is in memory at the end of the multiplier loop and need not be reread for the solution loop.

Note: This storage method is only viable if two complete rows fit into memory.

Column Storage

For the column storage method, this phase is not quite as simple. In the column storage method, several complete columns are stored together. Hence, each block will contain at least one column. Since the size of each block was determined to allow two blocks in memory at once, there is enough room in memory for one block and the entire set of b. Since b(n) is affected by b(1), it makes sense to store all the b's together and keep the b's in memory for the duration of this phase.

At this point, the similiarity to the row method stops. In the row method, b(j) was modified by the contents of row j in one pass. In the column method, only part of row j is in memory at once. Therefore, b(j) is only partially transformed in a single pass.

Although the main program below appears to be the same, the logic of APPLY_MULT and SOLVE_X is very different.

The algorithm for column storage is:

```
Read in the b vector

DO I = 1 TO NUM_SUBMAT_PER_COL;
   CALL READ_BLOCK (I,BLOCK1);
   CALL APPLY_MULT (I,BLOCK1);
END;

CALL SOLVE_X (N,BLOCK1);

DO I = N-1 TO 1 BY -1;
   CALL READ_BLOCK (I,BLOCK1);
   CALL SOLVE_X (I,BLOCK1);
END;

Write out the b vector
```

There are only 2*N-1 reads of the matrix and one read and write for the vector b.

Note: We read most of the matrix in twice. In the multiplier loop, many U values were read in. In the solution phase, many L values were read in. Only BLOCK (N) was not read twice, since it is in memory at the end of the multiplier loop and need not be reread for the solution loop.

Note: This storage method is only viable if two complete columns fit into memory.

Three Square Blocks

For the three square storage method, if the entire vector b fits into 2/3 of memory, then our algorithm is quite simple.

We first read in the lower triangular blocks (those containing the L values) and apply them to b. Then, we read in the upper triangular blocks and apply the U values to b. Hence, we read and write b only once.

Here is the algorithm:

```
read in the b vector

DO I = 1 TO NUM_SUBMAT_PER_COL;

   DO J = 1 TO I-1;
      CALL READ_BLOCK (I,J,BLOCK1);
      CALL FULL_BLOCK_MULT (I,J,BLOCK1);
   END;

   CALL READ_BLOCK (I,I,BLOCK1);
   CALL HALF_BLOCK_MULT (I,I,BLOCK1);

END;

CALL HALF_BLOCK_BACKSUB
     (NUM_SUBMAT_PER_COL,NUM_SUBMAT_PER_CO_,BLOCK1);

DO I = NUM_SUBMAT_PER_COL - 1  TO  1 BY -1;

   DO J = NUM_SUBMAT_PER_COL TO I+1 BY -1;
      CALL READ_BLOCK (I,J,BLOCK1);
      CALL FULL_BLOCK_BACKSUB (I,J,BLOCK1);
   END;

   CALL READ_BLOCK (I,I,BLOCK1);
   CALL HALF_BLOCK_BACKSUB (I,I,BLOCK1);

END;

write the b vector
```

When applying the factors to b, we have two distinct cases:

1. The block contains only L values or U values. Hence, the subroutine FULL_BLOCK_MULT applies all the values in the block to b as multipliers. The subroutine FULL_BLOCK_BACKSUB uses all the values in the block to solve for x.

2. The block is on the diagonal and contains both L and U values. Therefore, HALF_BLOCK_MULT will only apply the L values of the block to b, while HALF_BLOCK_BACKSUB will use the remaining values of the block (the U factors) to solve for x.

This algorithm does one read and one write for the vector b.

In the multiply phase, each block in the lower triangle (including the diagonal) is read once and applied to an appropriate b block. No matrix block needs to be written. Since there are (N * (N+1)) / 2 blocks in the lower triangle, we have

$$\frac{N ** 2 + N}{2}$$

reads.

In the solution phase, each block in the upper triangle is read once. This includes the diagonal, except for (N,N). So, we have another

$$\frac{N ** 2 + N}{2} - 1$$

reads.

So combining the matrix reads, we have N ** 2 + N - 1 reads of matrix blocks. In addition, b is read and written, given a total of N**2 + N + 1 I/O operations.

Note: In this method, we have read the matrix a little bit more than once. This is significantly better than the row or column algorithms, which read the matrix twice.

Note: Since the diagonal blocks contain both L and U factors, they are read twice. Hence, we have N**2 + N-1 (the N-1 being the extra reads for the diagonal), rather than just N ** 2.

Note: This entire algorithm is based upon the vector b fitting in 2/3 of memory.

For the three square storage method, if b does not fit into 2/3 of memory, then our algorithm is much more complex. In this storage method, an entire column need not fit into memory at once. Hence, we must divide the b values into subblocks.

In the multiplier phase, if a given subblock of matrix A contains the element A(k,l), we must have b(l) in memory in order to apply A(k,l) to b(k).

In the solution phase, if a given subblock of matrix A contains the element A(l,k), we must have x(k) in memory in order to apply A(l,k) to b(l).

Hence, we shall need one block of A (containing the L factors) in memory along with two blocks of b values.

In this storage method, memory is divided into three blocks. Each block of the matrix A contains x ** 2 elements, where x ** 2 is approximately one-third of memory. Each row of the submatrix contains x elements. Therefore, the number of submatrices is (n/x). Since there are only n elements of b, the number of blocks needed to contain b is n/(x**2). Hence, there are x matrix blocks for each b block.

When applying the factors to b, we have three cases:

1. The entire block contains L values or U values. This block can be combined with a b block directly. As an example, the b block contains the values 1 thru (x**2). The matrix block contains the rows k thru k + x. Hence, we can apply all the values in the matrix to b using only the values in this b block. The routines FULL_BLOCK_MULT and FULL_BLOCK_BACKSUB handle these blocks.

2. The entire block contains L values or U values. This block needs another b block in order to transform a given b block. As an example, the b block contains the values (x**2 + 1) thru (2*x**2). The matrix block contains the rows (x**2+1) thru (x**2+x) and the columns 1 thru x. In order to apply the matrix block to the b block, we also need the b block containing the values 1 thru x**2. The routines TRI_BLOCK_MULT and TRI_BLOCK_BACKSUB handle these blocks.

3. The block is on the diagonal. Half the block contains L values. Half the block contains U values. This block can be directly appied to the a block without using any other b block. The routines HALF_BLOCK_MULT and HALF_BLOCK_BACKSUB will handle these blocks.

In following algorithm, the number of blocks used to store the b vector is NUM_BLOCKS_OF_RHS, which is n/(x**2). Each block of b values contains x**2 elements. Since each block of A values has x elements per column, there are x matrix blocks for each b block. Hence, RATIO_A_BLOCKS_TO_RHS_BLOCKS = x.

```
/* applying the multipliers */

DO I = 1 TO NUM_BLOCKS_OF_RHS;

   A = (I-1) * RATIO_A_BLOCKS_TO_RHS_BLOCKS + 1;
   B = I * RATIO_A_BLOCKS_TO_RHS_BLOCKS;

   IF B > NUM_SUBMAT_PER_COL
      THEN B = NUM_SUBMAT_PER_COL;

   CALL READ_VALUE (I,VALUE1);

   DO J = 1 TO I-1;

      CALL READ_VALUE (J,VALUE2);

      X = (J-1) * RATIO_A_BLOCKS_TO_RHS_BLOCKS + 1;
      Y = J * RATIO_A_BLOCKS_TO_RHS_BLOCKS;

      IF Y > NUM_SUBMAT_PER_COL
         THEN Y = NUM_SUBMAT_PER_COL;

      DO L = A TO B;
         DO K = X TO Y;
            CALL READ_BLOCK (L,K,BLOCK);
            CALL TRI_BLOCK_MULT (VALUE1,VALUE2,BLOCK);
         END;
      END;

   END;

   DO L = A TO B;
      DO K = A TO L-1;
         CALL READ_BLOCK (L,K,BLOCK);
         CALL FULL_BLOCK_MULT (VALUE1,BLOCK);
      END;

      CALL READ_BLOCK (L,L,BLOCK);
      CALL HALF_BLOCK_MULT (VALUE1,BLOCK);

   END;

   CALL WRITE_VALUE (I,VALUE1);

END;
```

```
/* back substitution --- solving for x */

DO I = NUM_BLOCKS_OF_RHS TO 1 BY -1;

   A = (I-1) * RATIO_A_BLOCKS_TO_RHS_BLOCKS + 1;
   B = I * RATIO_A_BLOCKS_TO_RHS_BLOCKS;

   IF B > NUM_SUBMAT_PER_COL
      THEN B = NUM_SUBMAT_PER_COL;

   CALL READ_VALUE (I,VALUE1);

   DO J = NUM_BLOCKS_OF_RHS TO I+1 BY -1;

      CALL READ_VALUE (J,VALUE2);

      X = (J-1) * RATIO_A_BLOCKS_TO_RHS_BLOCKS + 1;
      Y = J * RATIO_A_BLOCKS_TO_RHS_BLOCKS;

      IF Y > NUM_SUBMAT_PER_COL
         THEN Y = NUM_SUBMAT_PER_COL;

      DO L = B TO A BY -1;
         DO K = Y TO X BY -1;
            CALL READ_BLOCK (L,K,BLOCK);
            CALL TRI_BLOCK_BACKSUB (VALUE1,VALUE2,BLOCK);
         END;
      END;

   END;

   DO L = B TO A BY -1;

      DO K = B TO L+1 BY -1;
         CALL READ_BLOCK (L,K,BLOCK);
         CALL FULL_BLOCK_BACKSUB (VALUE1,BLOCK);
      END;

      CALL READ_BLOCK (L,L,BLOCK);
      CALL HALF_BLOCK_BACKSUB (VALUE1,BLOCK);

   END;

   CALL WRITE_VALUE (I,VALUE1);

END;
```

By analyzing the algorithms, we see that each b block is written twice (once in the L phase and once in the U phase). There are Z blocks of b values, where Z = (N/x).

In the L phase, b block(1) is read Z times, b block(2) is read Z-1 times, b block(3) is read Z-2 times, ..., b block (Z) is read once. Hence, the number of b block reads is

$$\sum_{I=1}^{Z} (Z-I+1) = \frac{Z^{2} + Z}{2}$$

In the U phase, b block(1) is read once, b block (2) is read 2 times, b block(3) is read 3 times, ..., b block (Z) is read Z times. Hence, the number of b block reads is

$$\sum_{I=1}^{Z} I = \frac{Z^{2} + Z}{2}$$

So, the total number of b block I/O operations is Z**2 + 3*Z.

As in the previous algorithm, there are N**2 + N submatrix reads and no submatrix writes. This algorithm did not save the read of block (N,N), hence its total I/O count is N ** 2 + N rather than N ** 2 + N - 1.

Therefore, the total number of I/O operations is
N**2 + N + Z**2 + 3*Z.

Two Square Storage

In this storage method, the matrix A was divided into subblocks, where two subblocks fit into memory at once.

If the b vector fits into 1/2 of memory, then we can use the algorithm presented in the previous section for the three square method, when b fit into 2/3 of memory. That algorithm read and wrote b once, and did N ** 2 + N - 1 reads of the matrix A, where N = NUM_SUBMAT_PER_COL.

In case b does not fit into 1/2 of memory, then we must partition the vector b into subblocks, where each b block uses approximately one-fourth of memory. Hence, two b blocks and one A subblock fit into memory at once. Therefore, the second algorithm used for the three square method is applicable. Using that algorithm, we did N**2 + N reads of the matrix A, where N was equal to (n/x) where x**2 = (m/2). We also did Z**2+3*Z reads and writes, where Z is the number of b blocks. Using two b blocks in memory at once, each b block has (m/4) elements. Since there are n elements in b, Z = n/(m/4).

Note: If we must partition b, then the two square storage method has more b blocks than the corresponding three square method. Also note that there are less A blocks than in the three square storage method.

Chapter 7

Final Discussion

After having done all this analysis on the various storage methods, we can present the following decision algorithm:

Assume we are given an n by n matrix A.

Assume we can store m elements in memory at once.

Compute the number of blocks (i.e., Q ** 2) that the three-square storage method would need. Call this value SQ3. Using SQ3 and the formulae for THREESUB#10, compute the number of I/O operations needed. Call this value SQ3I/O.

If Q >= 78, we will use the three-square storage method and algorithm THREESUB#10.

If Q <= 77, we continue the decision making process.

Compute the number of blocks (i.e., Q ** 2) that the two-square storage method would need. Call this value SQ2. Using SQ2 and the formulae for TWOSUB#5, compute the number of I/O operations needed. Call this value SQ2I/O.

If Q >= 7, compare SQ3I/O with SQ2I/O and use the storage method and related algorithm yielding the least amount of I/O.

If Q < 7, we continue the decision making process.

Compute the number of blocks that the column storage method would need. Call this value COL. Using COL and the formulae for COL#1, compute the number of I/O operations needed. Call this value COLI/O.

Compare COLI/O, SQ3I/O and SQ2I/O. Choose the method yielding the least amount of I/O.

Finally, if Q = 2, use THREESUB#3.

In general, we can conclude that the three-square method is best only for extremely large matrices or cases where memory is severely limited.

More typically, the choice will be between the column method and the two-square method. If SQ2 (the number of blocks needed for the two-square method) is greater than 25 (Q = 5), then TWOSUB#5 is the preferred method. If SQ2 is less than 25, the choice depends on the memory utilization. Sometimes, column storage using COL#1 will do less I/O operations. Other times, two-square storage using TWOSUB#5 will do less I/O operations.

Note: The analysis has ignored:

Size of Program

Clearly, a larger program will reduce the amount of memory available for the storage of matrix elements.

Temporary Variables

Clearly, different programs will use different numbers and types of temporaries, thereby affecting the amount of memory available for the storage of matrix elements.

Hidden Vectors

The two-square storage method requires a vector that can hold one column of a subblock. This vector is used in the decomposition of A, the application of L to b, and in the solution of x using U. Clearly, this reduces the amount of memory available for the matrix A and reduces the size of each subblock.

Integer Number of Blocks When computing the number of blocks, certain divisions and square roots must be taken. These arithmetic operations do not always yield integers. Hence, values are either rounded or truncated. Therefore, memory can sometimes accomodate more elements than the storage method will use. This is unavoidable, since the amount of memory left is usually not enough to increase the size of each subblock.

Spillage

As an example, assume that a 100 by 100 matrix is presented to the two-square method, and memory can contain two 6 by 6 subblocks; the two-square method yields a 17 by 17 matrix of subblocks. This is enough to handle a 102 by 102 matrix. Although our storage method and algorithm will handle a somewhat larger matrix, we cannot 'shave' the last row and column of blocks and achieve any savings.

Transmission Time

Throughout the analysis, the concern has been to reduce the number of I/O operations, ignoring the fact that blocks of different sizes have different transmission times. The assumption has been that transmission time is insignificant when compared to seek time.

Language, Computer, Operating System

By ignoring these factors, one avoids the discussion about which storage methods and algorithms will do best in a given environment. Although our analysis is true in general, there may be a given language, computer, and operating system combination that may not agree with the results achieved in this paper.

Here are some examples of possible problems:

Stack computers may not have the problem of temporaries.

Different languages may use different numbers of temporaries.

Different operating systems will use different buffering schemes and I/O algorithms for random access file operations.

Certain operating systems may eliminate large amounts of seek time using cache memory, hence increasing the significance of the transmission time.

The usage of cache memory may introduce the need to weigh the size of the subblock with its effect on the utilization of the cache.

Operating systems that use segmentation or paging may require different algorithms having loops that are wholly contained within segments or pages.

Operating systems having 'look ahead' paging may use different algorithms that utilize these features.

Certain languages are not efficient on certain computers under certain operating systems.

Computers with array processors may avoid the problem of the 'hidden vector', since they may process the various rows and columns in parallel.

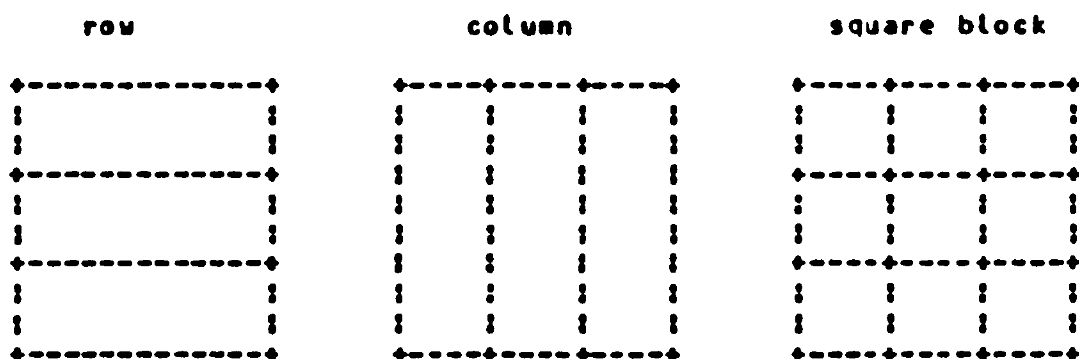
Note: The analysis has focused upon the number of I/O operations performed during the decomposition of A into LU factors. During the application of the L factors to b, I/O operations were done. During the solving for x using the U factors, I/O operations were also done. For the row and column storage methods, the number of I/O operations done was approximately 2*N. For the submatrix methods, the number of I/O operations was approximately (1/2) * N **2. These terms are insignificant in comparison with the N**3 and N**4 counts generated during the decomposition phase. Hence, these small counts have not been included in the comparisons.

Chapter 8

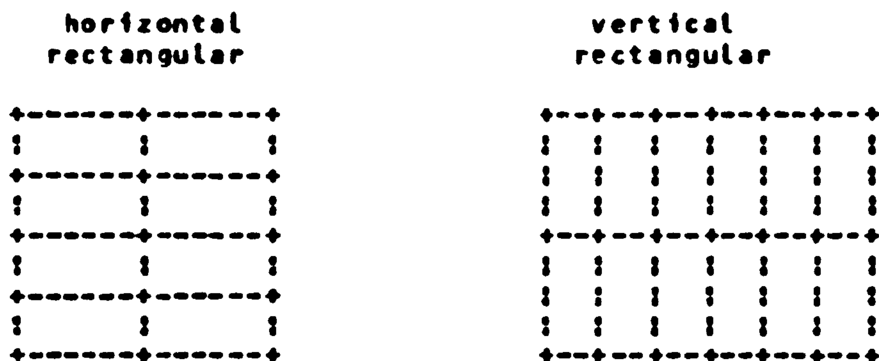
Future Research

Rectangular Storage Methods

In this thesis, we have examined three storage schemes. They were:



In future research, we expect to study two hybrid methods, which are:



These two methods partition A into non-square submatrices. One method has more elements per row of the submatrix, while the other has more elements in the column of the submatrix. Therefore, the number of submatrices comprising a row of the original matrix is different than the number of submatrices needed to comprise a column of the original matrix.

We conjecture that the horizontal rectangle scheme method lies between the square submatrix and the row storage scheme. It would be satisfying to develop decomposition algorithms for this storage scheme whose formulae would lie in between those of the square and row. What we would like to find is the following: the squarer the block, the closer to the square I/O counts; the more rectangular, the closer to the row I/O counts. This method may be appealing in cases where the row method is desired, but two complete rows cannot fit into memory at once.

Similiarly, the vertical rectangular scheme should lie in between the square submatrix and the column storage scheme.

Decomposition Algorithms With Interchanges

In principle, all the decomposition algorithms developed for the column storage method are easily modified to accomodate pivoting. A vector of size n must be used in order to record the swaps. The addition of this vector will probably reduce the number of columns fitting into a block, hence increasing the number of blocks needed to store the matrix and the algorithms will do more I/O. However, if memory can only accomodate two full columns, then the swap vector will preclude the usage of any of our previously developed column algorithms.

In order to implement pivoting for the row and the submatrix schemes in an efficient manner, one should avoid sweeping the remaining matrix looking for the maximal element. The paper by Barron and Swinnerton-Dyer (1960) speculates about a new pivoting strategy that addresses this problem. This strategy uses the largest element uncovered so far in choosing the pivot. More analysis is needed to determine the numerical accuracy of this strategy.

Pivoting introduces complications that make the square blocks methods unattractive. The simplest approaches require so much additional I/O that they are not attractive. Although the technique adopted by Barron and Swinnerton-Dyer (1960) could be applied to square blocks, it would be attractive only if we used a scheme analogous to the row scheme, and this would mean we could not achieve the savings of THREESUB#10 or TWOSUB#5. In general, if pivoting is required, further research is needed to determine an efficient square block algorithm. The papers discussing the square block method (McKellar and Coffman (1969), Nugent and Du Croz (1981)) have also ignored the cases where pivoting is required.

References

Cited References

Barron, D.W., Swinnerton-Dyer, H.P.F., "Solutions of Simultaneous Linear Equations using a Magnetic-Tape Store", Computer Journal, vol. 3, No. 1, pg. 28-33 (April 1960).

Bunch, J.R., annd Parlett, B.N., "Direct Methods for Solving Symmetric Indefinite Systems of Linear Equations", SIAM J. Numer. Anal., 8(1971), 639-655.

Du Croz, J.J., Nugent, S.M., Reid, J.K., Taylor, D.B., "Solving Large Full Sets of Linear Equations in a Paged Virtual Store", ACM. Trans. Math. Software, Vol. 7, No. 4, pg. 527-535 (December 1981).

Du Croz, J.J., Reid, J.K., Taylor, D.B., "Algorithm 578: Solution of Real Linear Equations in a Paged Virtual Store (F4)", ACM. Trans. Math. Software, Vol. 7, No. 4, pg. 537-541 (December 1981).

Forsythe, G.E., Moler, C.B., "Computer Solution of Linear Algebraic Equations.", Prentice-Hall, Englewood Cliffs, N.J., 1967.

Goldfarb, D., Private Communication, May, 1983.

Goldstine, H.H., Von Neumann, J., "Numerical Inverting of Matrices of High Order. II.", Bull. Amer. Math. Soc., Vol. 57, pg. 901-924 (May, 1953).

McKellar, A.C., Coffman, E.G., "Organizing Matrices and Matrix Operations for Paged Memory Systems", Commun. ACM., Vol. 12, No. 3, pg. 153-165 (March 1969).

Moler, C.B., "Algorithm 423, Linear Equation Solver.", Commun. ACM., Vol. 15, No. 4, pg. 274 (April 1972).

Moler, C.B., "Matrix Computations with Fortran and Paging.", Commun. ACM., Vol. 15, No. 4, pg. 268-270 (April 1972).

Von Neumann, J., Goldstine, H.H., "Numerical Inverting of Matrices of High Order.", Bull. Amer. Math. Soc., Vol. 53, pg. 1021-1099 (November, 1947).

Uncited References

Aho, A., Hopcroft, J.E., and Ullman, J.D., "The Design and Analysis of Computer Algorithms.", Addison-Wesley, Reading, MA , 1974.

Bartels, R., Golub, G.H., "The Simplex Method of Linear Programming Using LU Decomposition.", Commun. ACM., Vol. 12, No. 5, pg. 266-268 (May 1969).

Bowdler, H.J., Martin, R.S., Peters, G., and Wilkinson, "Solution of Real and Complex Systems of Linear Equations.", Numer. Math., Vol. 8, pg. 217-234 (1966).

Bunch, James. Ph.D. Thesis. University of California at Berkeley. 1969.

Bunch, J.R., "Analysis of the Diagonal Pivoting Method", SIAM J. Numer. Anal., 8(1971), 656-680.

Burden, R.L., Faires, J.D., and Reynolds, A.C., "Numerical Analysis.", Prindle, Weber and Schmidt, Boston Mass., 1978.

Businger, P, Golub, G.H., "Linear Least Squares Solutions by Householder Transformations", in Handbook Series Linear Algebra, Numer. Math., Vol. 7, pg. 269-276 (1965).

Chartes, B.A., and Geuder, J.C., "Computing Error Bounds for Direct Solution of Linear Equations", JACM, Volume 14, January, 1967, pp. 63-71.

Cline, A.K., Moler, C.B., Stewart, G.W., and Wilkinson, J.H., "An Estimate for the Condition Number of a Matrix", SIAM J. Numer. Anal., 16, pg. 368-375 (1979).

Coffman, E.G., Varian, L.C., "Further Experimental Data on the Behavior of Programs in a Paging Environment.", Commun. ACM., Vol. 11, No. 7, pg. 471-474 (July, 1968).

Dongarra, J.J., Bunch, J.R., Moler, C.B., and Stewart, G.W., "Linpack Users Guide", SIAM, Philadelphia, P.A., 1979.

Faddeev, D.K., and Faddeeva, V.N., "Computational Methods of Linear Algebra.", W. H. Freeman, San Francisco, 1963.

Fischer, P.C., Probert, R.L., "Storage Reorganization Techniques for Matrix Computation in a Paging Environment", Commun. ACM., Vol. 22, No. 7, pg. 405-415 (1979).

Ford, B., "Parameterization of the Environment for Transportable Numerical Software.", ACM. Trans. Math. Software, Vol. 4, No. 2, pg. 100-103 (June 1978).

Forsythe, G.E., "Algorithm 16: Crout with Pivoting.", Commun. ACM., Vol 3, No. 9, pg. 507-508 (September 1960).

Forsythe, G.E., "Today's Computational Methods of Linear Algebra.", SIAM Rev., Vol. 9, No. 3, pg. 489-515.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B., "Computer Methods for Mathematical Computations.", Prentice-Hall, Englewood Cliffs, N.J., 1977.

Fox, P.A., Hall, A.D., Schryer, N.L., "The PORT Mathematical Subroutine Library.", ACM. Trans. Math. Software, Vol. 4, No. 2, pg. 104-126 (June 1978).

Golub, G.H. and Reinsch, C., "Singular Value Decomposition and Least Squares Solutions", Numerische Mathematik, 14(5), 1970, 403-420.

Hellerman, H., "Addresing Multi-dimensional Arrays.", Commun. ACM., Vol. 5, No. 4, pg. 205-207 (April 1962).

Henderson, D.S., and Wassying, A., "A New Method for the Solution of Ax = b", Numer. Math., 29, pg. 287-289 (1978).

International Mathematical and Statistical Libraries, Library 2, Ed. 5 IMSL, Houston, Tex.

Kalaba, R. and Rasakhod, N. "Analytical Aspects of Computational Linear Algebra", Appl. Math. and Comput., Vol. 8, No. 2, pg. 83-109 (1981).

Larsen, L.J., "A Modified Inversion Procedure for Product Form of the Inverse Linear Programming Codes.", Commun. ACM., pg. 382-383.

Lawson, C.L. and Hanson, R.J., "Solving Least Squares Problems", Prentice-Hall Inc., Englewood Cliffs, N.J., 1974.

Lawson, C.L., Hanson, R.J., and Kincaid, D.R. "Basic Linear Algebra Subprograms for Fortran Usage", ACM Trans. Math. Software. Vol. 5, No. 3, pg. 308-323 (1979).

Marshall, C.P., "Inversion of Matrices by Partitioning.", J. ACM., Vol. 16, pg. 303-314 (April 1969).

Martin R.S., Peters, G., and Wilkinson, J.H., "Symmetric Decomposition of a Positive Definite Matrix", Numerische Mathematik, 7(5), 1965, 362-383.

Mayoh, B.H., "A Graph Technique for Inverting Certain Matrices.",.

Moler, C.B., "Algorithm 423, Linear Equation Solver.", Commun. ACM., Vol. 15, No. 4, pg. 274 (April 1972).

Moler, C.B., "Linear Equation Solver", Comm. ACM., 15, pg. 274 (1972).

Moler, C.B., "Matrix Computations with Fortran and Paging.", Commun. ACM., Vol. 15, No. 4, pg. 268-270 (April 1972).

Randell, B., and Kuehner, C.J., "Dynamic Storage Allocation Systems.", Commun. ACM., Vol. 11, No. 5, pg. 297-306 (May 1968).

Rice, J.R., "Matrix Computations and Mathematical Software", McGraw Hill N.Y. 1981.

Rice, J.R., Mathematical Software. Academic Press, N.Y. 1971. Skeel, R.D. "Iterative Refinement Implies Numerical Stability for Gaussian Elimination", Math. Comput. Vol. 35, No. 151, pg. 817-832 (1980).

Skeel, R.D., "Effect of Equilibration on Residual Size for Partial Pivoting", SIAM J. Numer. Anal. Vol. 18, No. 3, pg. 449-454 (June 1981).

Skell, R.D., "Scalling for Numerical Stability in Gaussian Elimination", J. Assoc. Comput. Mach., Vol. 26, No. 3, pg. 494-526 (1979).

Smith, B.T., et. al. , "Matrix Eigensystem Routines - EISPACK Guide. Springer-Verlag, N.Y., 2nd ed., 1976.

Steinberg, D.J., "Computational Matrix Algebra.", McGraw-Hill, N.Y., 1974.

Stewart, G.W., "Introduction to Matrix Computations", Academic Press, N.Y., 1973.

Steward, D.V., "On an Approach to Techniques for the Analysis of the Structure of Large Systems of Equations.", SIAM Rev., Vol. 4, No. 4, pg. 321-342 (October 1962).

Strassen, V., "Gaussian Elimination is not Optimal.", Numer. Math., Vol. 13, pg. 354-356 (1969).

Swift, G., "A comment on matrix inversion by partition." SIAM Rev., Vol. 2 , pg. 132-133 (1960).

System 360 Matrix Language (MATLAN) Aplication Description, IBM H20-0479 Program Description Manual, IBM H20-0564.

Wassyng, A., "Solving Ax = b: A Method With Reduced Storage Requirements", SIAM J. Numer. Anal. Vol. 19, No. 1, pg. 197-204 (February 1982).

Wilkinson, J.H., "Error Analysis of Direct Methods of Matrix Inversion.", J. ACM., Vol. 8, No. 3, pg. 281-330 (July 1961).

Wilkinson, J.H., "Householder's Method for Symmetric Matrices.", Numer. Math., Vol. 4, pg. 354-361 (1962).

Wilkinson, J.H., "Rounding Errors in Algebraic Processes", Prentice-Hall Series in Automatic Computation, Englewood Cliffs, N.J., 1963.

Wilkinson, J.H., and Reinsch, C., "Handbook for Automatic Computation", Vol. 2, Linear Algebra, Springer, Berlin, 1971.

Wilkinson, J.H., "The Solution of Ill-Conditioned Linear Equations", Mathematical Methods for Digital Computers, Editors, A. Ralston and H. Wilf, John Wiley, New York, 1967, Chapter 3.

Young, O.M., Gregory, R.J., "A Survey of Numerical Mathematics.", Addison-Wesley, Reading Mass., 1973.

Zlatev, Z., Wasniewski, J., and Schaumburg, K., "Comparison of Two Algorithms for Solving Large Linear Systems.", SIAM J. Sci. Stat. Comput., Vol. 3, No. 4, pg. 486-501 (December 1982).

Sparse Matrix References

Duff, I.S., "A Survey of Sparse Matrix Research", Proc. IEEE, Vol. 65, No. 4, pg. 500-535 (1977).

Eisenstat, S.C., Schultz, M.H., and Sherman, A.H. "Algorithms and Data Structures for Sparse Symmetric Gaussian Elimination", SIAM J. Sci. and Stat. Comput. Vol. 2, No. 2, pg. 225-237 (1981).

Forsythe, G.E., Moler, C.B., "Two fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition", ACM. Trans. Math. Software, 4, pg. 250-269 (1978).

George, A. "A Minimal Storage Implementation of the Minimum Degree Algorithm, SIAM J. Numer. Anal. Vol. 17, No. 2, pg. 282-299 (1980).

George, A., and Liu, J.W., "Computer Solution of Large Sparse Positive Definite Systems.", Prentice-Hall, Englewood Cliffs, N.J., 1981.

George, A., and Liu, J.W.H., "The Design of a User Interface for a Sparse Matrix Package", ACM. Trans. Math. Soft., Vol. 5, No. 2, pg. 139-162 (1979).

Grimes, R.G. and Lewis, J.G. "Condition Number Estimation for Sparse Matrices", SIAM J. Sci. and Stat. Comput. Vol. 2, No. 4, pg. 384-388 (1981).

Martin, R.S., Wilkinson, J.H., "Solution of Symmetric and Unsymmetric Band Equations and the Calculation of Eigenvectors of Band Matrices.", Numer. Math., Vol. 9, pg. 279-301 (1967).

Nocedal, J. "Updating Quasi-Newton Matrices with Limited Storage", Math. Comput. Vol. 35, No. 151, pg. 773-782 (1980).

Pease, M.C., "Matrix Inversion Using Parallel Processing.", J. ACM., Vol. 14, No. 4, pg. 757-764 (October 1967).

Pooch, U.W., and Nieder, A., "A Survey of Indexing Techniques for Sparse Matrices.", Computing Surveys, Vol. 5, No.2, pg. 109-133 (June 1973).

Rose, D.J., "An Algorithm for Solving a Special Class of Tridiagonal Systems of Linear Equations.", Commun. ACM. , Vol. 12, No. 4, pg. 234-236 (April 1969).

Rose, D.J., Willoughby, R.A., "Sparse Matrices and their Applications.", Plenum Press, N.Y. 1972.

Rosen, J.B., "Primal Partitioning Programming for Block Diagonal Matrices.", Numer. Math., Vol. 6, pg. 250-260 (1964).

Schreiber, R., "A New Implementation of Sparse Gaussian Elimination", ACM. Trans. Math. Soft., Vol. 8, No. 3, pg. 256-276 (September 1982).

Sherman, A.H., "Algorithms for Sparse Gaussian Elimination With Partial Pivoting", ACM. Trans. Math. Soft., Vol. 4, No. 4, pg. 330-338 (1978).

Tewarson, R.P., "Computations with Sparse Matrices.", SIAM Rev., Vol. 12, No. 4, pg. 527-543 (October 1970).

Weil, R.L., Kettler, P.C., "Rearranging Matrices to Block-Angular Form for Decomposition (and other) Algorithms.", Management Science, Vol. 18, No. 1, pg. 98-108 (September 1971).

Wilkinson, J.H., "Calculation of the Eigenvectors of a Symmetric Tridiagonal Matrix by Inverse Iteration.", Numer. Math., Vol. 4, pg. 368-376 (1962).