Some programming techniques for processing multi-dimensional matrices in a paging environment

by JAMES L. ELSHOFF
General Motors Research Laboratories
Warren, Michigan

INTRODUCTION

Although virtual memory systems are supposed to free the programmer from space management problems, the systems do not always succeed. In fact, programmers find that by ignoring the fact that real core is limited, the cost of executing their programs sometimes makes them unusable, not to mention some of the detrimental effects the program has on the throughput of the overall system. This problem seems to be especially prevalent when large matrices of data are involved. The data are usually referenced in a cyclical pattern and when the entire matrix will not fit in core, the number of page faults encountered during execution is maximized. The focus of this paper is to analyze programming techniques which will reduce the number of page faults in matrix operations and thereby improve program performance.

Program behavior in a paging environment has been studied from several points of view. Specifically, Brawn, Gustavson, and Mankin have concerned themselves with processing vectors in a paging environment. Moler and Dubrulle have looked at two separate matrix operations with respect to execution in a virtual memory environment. Several storage schemes and related operations for matrices were analyzed with respect to paging systems by McKellar and Coffman. Also, Guertin presented some programming examples to improve program behavior in a demand paging system.

The work presented in this paper was done on the premise that a programmer must be aware of how his program will reference the data during execution. The programmer will not be completely free of space management considerations in the design of his algorithms. The material presented deals with the mathematically simple problems of matrix addition, transposition, and multiplication. The methods of problem analysis and the programming guidelines are intended to give the working programmer new tools for doing a better job.

THE WORKING ENVIRONMENT

Although the material presented in this paper is directly extendable to matrices with more than two dimensions, the matrices used in the examples will all be two-dimensional for the sake of simplicity. The indices will refer to the row and column of the matrix from left to right. An M X N matrix A will have M rows numbered 1 through M and N columns numbered 1 through N. All matrices will be assumed to be stored rowwise. A 2 X 2 array A will have its elements mapped into linear virtual memory space in the order A(1,1), A(1,2), A(2,1), A(2,2). The order of the subscripts within the subscript list may be reversed throughout this paper for columnwise storage.

The paging algorithm executed by the operating system will be the least recently used LRU algorithm. This algorithm was chosen because most operating systems available either use this algorithm or an approximation to it. Also, the experimental results shown in the latter sections of the report were generated on a computer with LRU hardware. Note that this algorithm is used as a basis for the derivation of formulas and is not essential to the premises upon which this paper is founded.

The total number of page faults processed during a complete matrix operation will be used as a measure of the program performance. The CPU time required to perform the matrix operation will be considered to be constant. Implementation of some of the programming techniques described herein may increase program execution time due to additional loop controls, but this is considered to be negligible because the additional CPU time is measured in microseconds while the time to process a page fault is measured in tens of milliseconds.

The examples used are coded using PL/I DO statements for conciseness and readability. Except for matrix mapping functions, the programs being considered are really language independent. FORTRAN programmers may have to use IF loops instead of DO loops since they cannot specify negative increments on their DO statements. Furthermore, the programs shown are not written to minimize CPU time. Overlaying each two-dimensional matrix with a vector is an obvious method of reducing CPU time.

Table I lists some of the symbols and their respective meanings which are used throughout this paper. The notation $\lceil a \rceil$ will be used to signify the smallest integer greater than or equal to a and $\lfloor a \rfloor$ will signify the largest integer less than or equal to a.
In order to facilitate analysis, the first element of an array will be stored as the first word in a page. The matrix dimensions will satisfy the inequalities \( N \leq S < N^2 \). That is, at least one row of the matrix will fit in a page but not the whole matrix. The programming techniques hold when \( S < N \) but the formulas derived will not. In order to make the problem of interest, \( k < \sum_{i} p_i \) is also assumed. Furthermore, the executing code and the temporary variables are resident in real memory.

THREE PROGRAMMING TECHNIQUES

In this section three programming rules will be described which can be applied to multi-dimensional array operations in order to improve program performance. The circumstances under which each may be applied and the benefits that may be expected are presented.

Ordering nested loops

Let \( A \) be an \( M \times N \) matrix and \( B \) be an \( N \) element vector. Write a program so that each element of \( B \) contains the sum of all of the elements in the corresponding column of \( A \), \( b_i = \sum j a_{ij} \). Each element of \( B \) is initialized to zero. An obvious solution to this problem is to write a loop which will sum each column. Then enclose the loop in a second loop which will traverse all the columns.

\[
\begin{align*}
&\text{DO COL = 1 TO N BY 1; /* Traverse each column */} \\
&\text{DO ROW = 1 TO M BY 1; /* Sum a column */} \\
&\text{B(COL) = B(COL) + A(ROW,COL);} \\
&\text{END;}
\end{align*}
\]

Now consider the reference pattern on matrix \( A \), \( A(1,1), A(2,1), A(3,1), \ldots \), which causes each page spanned by matrix \( A \) to be referenced on each pass through the outer loop. By interchanging the \( DO \) statements the elements in matrix \( A \) will be referenced in the order in which they are stored. Thus, all of the elements in a single page will be processed while the page is in core. Furthermore, each page is only required to be in core once.

\[
\begin{align*}
&\text{DO ROW = 1 TO M BY 1; /* Traverse each row */} \\
&\text{DO COL = 1 TO N BY 1; /* Add all row elements */} \\
&\text{B(COL) = B(COL) + A(ROW,COL);} \\
&\text{END;}
\end{align*}
\]

The minimum number of page faults \( F_{\text{min}} = p_A + p_B \), since each page of both matrices must be brought into core at least once. For any number of real memory pages \( k \), where \( 2 \leq k < F_{\text{min}} \), the first program will have \( F = k(N_A + p_B) \) page faults while the second program will have \( F = F_{\text{min}} \) page faults. Certainly the first program is related to \( N^2 \) in this case because of the LRU paging algorithm that is assumed. But even considering an optimum paging algorithm, the number of page faults \( F = (k-1) + N(p_A - k+2) \), where \( p_B = 1 \), page faults is the best that can be done. Thus, even when \( k = p_A \), an optimum paging algorithm cannot get \( F = F_{\text{min}} \) in the case of the first program.

Rule 1. Nest loops so that the innermost loop defines the subscript with the minimum distance between elements when all other subscripts are held constant.

Rule 1 is a further generalization of a rule published by Guertin. The rule may be applied iteratively to determine the second from innermost loop once the innermost loop is fixed, etc. The rule applies to most cases where nested loops are encountered and should always be considered by the programmer. In order to apply the rule, the programmer must understand the storage mapping algorithm of the language being used as well as the problem being solved. Guertin discusses many variations and applications of Rule 1. One of the variations is paraphrased here as an example. Consider the following program which is like the first except that no initialization of the vector \( B \) is assumed.

\[
\begin{align*}
&\text{DO COL = 1 TO N BY 1; /* Initialize B */} \\
&\text{B(COL) = 0;} \\
&\text{DO ROW = 1 TO N BY 1;} \\
&\text{DO COL = 1 TO N BY 1; /* Add all row elements */} \\
&\text{B(COL) = B(COL) + A(ROW,COL);} \\
&\text{END;}
\end{align*}
\]

If the nested loops are interchanged, the program will no longer execute properly. But \( B \) can be initialized to zero in a separate loop at only a small additional cost. Better yet, initialize \( B \) with the first row of \( A \) and regain the additional loop time by eliminating \( M \) additions.

\[
\begin{align*}
&\text{DO COL = 1 TO N BY 1;} \\
&\text{B(COL) = A(1,COL);} \\
&\text{DO ROW = 1 TO N BY 1;} \\
&\text{DO COL = 1 TO N BY 1;} \\
&\text{B(COL) = B(COL) + A(ROW,COL);} \\
&\text{END;}
\end{align*}
\]


table i—definitions of symbols

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A,B,C</td>
<td>name of a matrix</td>
</tr>
<tr>
<td>F</td>
<td>number of page faults</td>
</tr>
<tr>
<td>L,M,N</td>
<td>dimension of a matrix</td>
</tr>
<tr>
<td>p_a</td>
<td>number of pages spanned by matrix A</td>
</tr>
<tr>
<td>k</td>
<td>number of pages of real (core) memory available for data</td>
</tr>
<tr>
<td>r_a</td>
<td>number of rows of matrix A in one page</td>
</tr>
<tr>
<td>s</td>
<td>page size in words (matrix elements)</td>
</tr>
<tr>
<td>q_i</td>
<td>number of rows processed per pass through loop indexed by i</td>
</tr>
</tbody>
</table>
Processing multiple rows

Let A and B be \( N \times N \) matrices and write a program to transpose B into A; \( a_{ij} = b_{ji} \).

```fortran
DO ROW = 1 TO N BY 1;
  DO COL = 1 TO N BY 1;
    A(ROW,COL) = B(COL,ROW);
  END;
END;
```

Rule 1 cannot be applied to this problem. Either A or B will have each of its pages referenced for each pass through the outer loop.

The number of page faults encountered in performing this transpose operation is \( F = p_A + Np_B \) in the working environment that has been described. Obviously the product \( Np_B \) causes the number of page faults to be high. \( N \) is the number of rows in the A matrix. If A has more than one row in a page, why not process all of the rows at one time? Let the rows in one page \( r_A \) be represented by the variable \( RPP \) and the following program results.

```fortran
DO ROWS = 1 TO N BY RPP;
  /\* Page of rows \*/
  MAXROW = MAX(ROWS + RPP-1,N);
  DO COL = 1 TO N BY 1;
    A(ROW,COL) = B(COL,ROW);
  END;
END;
```

The number of page faults is reduced to \( F = p_A + \lceil N/r_A \rceil p_B \) by processing multiple rows. For a small \( r_A \), \( p_A \) and \( p_B \) are very large and the reduction is significant. For a larger \( r_A \), the \( \lceil N/r_A \rceil \) factor is much smaller and the reduction is still significant.

A reasonable question to ask is why not process 2\( r_A \) rows during each pass through the outer loop. In this problem the number of page faults would be halved. But what if \( k = 2 \)? Then the second factor is halved but the first factor becomes \( 2\lceil N/r_A \rceil p_B \) and the expected gain becomes a loss. Processing multiple rows reduces the page faults as long as all of the rows being processed remain in real memory. Consider that letting \( RPP = N \) in the above example is tantamount to inverting the DO statements in the original program.

Rule 2. Process all the elements in a page which vary with respect to the subscript causing maximum paging while that page is in core and the other subscripts are held constant.

Rule 2 generally applies to those problems for which Rule 1 cannot minimize the page faults. Such problems can be characterized in several ways. (1) The same index may be used in different subscript positions within the loop. (2) A single element is referenced more than one time during the course of execution. (3) An extensive calculation is performed within the loop which results in insufficient real memory pages even after applying Rule 1.

There is no reason why Rule 2 cannot be applied to more than one index. McKellar and Coffman\(^4\) describe a matrix storage scheme which lends itself to matrix operations which apply Rule 2 to every subscript. For the storage scheme considered within this paper, Rule 2 has less effect on subscripts toward the right in the subscript lists since there are fewer elements per page which vary only in the right-hand subscripts.

Alternating matrix traversal direction

Consider the transpose problem used in the last section. The total number of page faults \( F = p_A + Np_B \) was reduced by attacking the factor \( N \), the number of rows in the A matrix. Another approach is to reduce the factor \( p_B \), the number of pages spanned by the B matrix. Since B is too large to fit in real memory, and since the algorithm references B in a cyclic manner, each page of B is removed from real memory between references. Thus, \( p_B \) can be effectively reduced by referencing an arbitrary page more than once while the page is in real memory. The cyclic reference pattern is broken in order to accomplish this end.

```fortran
COLSTART = 1;
COLEND = N;
COLDIFF = 1;
DO ROW = 1 TO N BY 1;
  DO COL = COLSTART TO COLEND BY COLDIFF;
    A(ROW,COL) = B(COL,ROW);
  END;
END;
```

The new program references the B matrix by going down the first column, \( B(1,1), B(2,1), \ldots, B(N,1) \), and up the second column, \( B(N,2), B(N-1,2), \ldots, B(1,2) \). The program continues to alternate the direction of column traversal until the matrix operation is complete. Since the pages referenced near the end of one column traversal are the same as those referenced at the beginning of the following column traversal, a number of page faults may be eliminated. With the LRU paging algorithm that has been assumed, \( F = p_A + p_A + (N-1)(p_B - k + 1) \). Thus, for each of \( N-1 \) column traversals the number of page faults that can be eliminated is the number of real memory pages available for the B matrix.

Rule 3. Let the increment of a faster varying subscript alternate its sign each time a more slowly varying subscript changes, when the more slowly varying subscript appears to the right of the faster varying subscript.
Rule 3, like Rule 2, should be applied after Rule 1. Rule 3 is based on a paging algorithm which keeps the most recently referenced pages in real memory. Although few paging systems have a true LRU paging algorithm, most systems do approximate the LRU algorithm. Consequently, benefits may not always be as great as indicated here, but results will be significant. Rule 3 produces positive results when applied to all subscripts except the leftmost. Rule 3 will not generally apply to the outermost loop of a set of nested loops.

Summary of rules

Three rules have been given which may be used in order to decrease the total number of page faults encountered while performing matrix operations. Each rule attack the problem from a different point of view and requires different knowledge on behalf of the programmer. Rule 1 eliminates page faults by aligning the reference pattern for the matrix elements with the storage mapping function; the programmer must know the storage mapping function. Rule 2 uses the programmer's knowledge of page size in the computer system in order to break a large problem into a series of smaller problems which generate fewer page faults. Finally, Rule 3 assumes a paging algorithm which approximates an LRU algorithm in order to reduce the number of page faults; consequently, the programmer must learn something about the paging strategy in the system.

THE TRANSPOSE OPERATION

An in-place matrix transpose operation is now analyzed with respect to the programming rules just given. The standard algorithm appearing in print

\[
\text{DO ROW = 1 TO N-1 BY 1;}
\text{DO COL = ROW TO N BY 1;}
\text{TEMP = A(ROW,COL);}
\text{A(ROW,COL) = A(COL,ROW);}
\text{A(COL,ROW) = TEMP; END;}
\]

will cause \(p_A\) page faults in processing the first row. For each additional row in the first page, \(p_A-1\) page faults will be incurred assuming the first page remains in core. When all processing has been completed on the first page, the rows on the second page will cause page faults on the remaining pages of the matrix in a similar manner. Finally, a point is reached where the remaining portion of the matrix will fit in real memory. The total number of page faults is given by

\[
F = p_A + \frac{1}{2} \left( \left\lceil \frac{r_A-q}{q} \right\rceil + 1 \right) \left( p_A^2 - p_A - k^2 + k \right).
\]

The second formula is derived from the case where the multiple rows being processed span an integral number of pages, \(a=r\), where \(a\) is a positive integer.

\[
F = \frac{1}{2} \left( \left\lceil \frac{p_A - k}{a} \right\rceil + 1 \right) \left( 2p_A - \left\lceil \frac{p_A - k}{a} \right\rceil \right)
\]

This formula also assumes that all of the multiple rows being processed will remain in real memory, \(a<k\). The number of page faults is given by

\[
F = k + \left( p_A - k - \left\lceil \frac{p_A - k}{a} \right\rceil \right) \left( \frac{p_A - k}{2} - 1 \right)
\]

when the matrix operation alternates the direction of loop traversal in order to reuse the matrix pages in real memory. All pages are assumed to be full in this case.

Finally, both the multiple row rule and the alternating direction rule can be applied in the same program. This combination results in the number of page faults being reduced to

\[
F = p_A + \frac{1}{2} \left\lceil \frac{p_A - k}{a} \right\rceil \left[ 2(p_A - k) - a \left\lceil \frac{p_A - k}{a} \right\rceil - 1 \right]
\]

where \(a\) is a positive integer such that \(ar=q\) and \(a<k\).

Table II—Summary of Transpose Operation Data

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Paging Buffer Size (k)</th>
<th>Expected Page Faults</th>
<th>Measured Page Faults</th>
<th>Measured Expected Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>15</td>
<td>445</td>
<td>430</td>
<td>1.011</td>
</tr>
<tr>
<td>Multiple Rows</td>
<td>14</td>
<td>119</td>
<td>148</td>
<td>1.243</td>
</tr>
<tr>
<td>Alternating Direction</td>
<td>15</td>
<td>131</td>
<td>144</td>
<td>1.099</td>
</tr>
<tr>
<td>Combination</td>
<td>15</td>
<td>35</td>
<td>64</td>
<td>1.828</td>
</tr>
</tbody>
</table>
Some Programming Techniques for Processing Multi-Dimensional Matrices in a Paging Environment

Figure 1—Comparison of transposition algorithms for a matrix spanning twenty pages

The parameters, which did not fall within the constraints under which the formulas were derived, were (1) the last page spanned by the matrix was not full and (2) a page did not exactly contain an integral number of rows. Table II summarizes the results from these runs.

Additional statements had to be added to the standard program when the programming rules were applied. Table III shows the additional CPU time required by the new programs and also the less total CPU time, memory, and channel resources that are needed.

An arbitrary thrashing level was indicated by a superimposed line on Figure 1. The line represents approximately one page fault each 10 milliseconds in its present location. This line is of special interest since the distance between the line and a curve above the line is directly proportional to the length of time the program will thrash. The duration of such detrimental effects is shown in Table III. The duration would be even worse if this program were a customer in a timesharing system where the program would be regularly removed from real memory. Thus, the 35 percent increase in CPU time for the transpose operation shown in Table III is negligible when compared to the overall reduction of 78 percent in total CPU time and 86 percent in elapsed time.

Another experiment was performed with the transpose operation. The purpose was to determine the validity of the formula for the multiple row transpose operation since the $rA - q/q$ factor and the $fPA - k/a$ factor introduce error when the result of the division is not an integer. Figure 2 shows the error introduced. The error for the formula with the $fPA - k/a$ factor is nearly constant and is attributable to...
the fact that a page does not contain an integral number of rows. No attempt will be made to apply this result to other multiple row formulas; however, please note that error bounds are easily calculated on each of these formulas if such a comparison is desired.

A program which will perform an in-place transpose with both the multiple row rule and the alternating direction rule applied is shown as Example 1. Although the program may seem complicated initially, it really is easily understood. Figure 3 shows the manner in which the array is referenced with respect to the outer loop. All of the elements in the areas shown with a 1 are interchanged during the first pass through the DO WHILE loop. The second pass interchanges the elements in the areas marked with a 2. The operation continues until the elements in the areas marked L are interchanged. The variable names from the program shown in Figure 3 relate to the first pass through the DO WHILE loop.

\[
\begin{align*}
\text{STARTROW} &= 1; \\
\text{ENDROW} &= N; \\
\text{DIFF} &= 1; \\
\text{DO WHILE} & \left( (\text{STARTROW-ENDROW}) \times \text{DIFF} < 0 \right); \\
& \text{IF DIFF > 0 /* DETERMINE LAST ROW FOR THIS LOOP */} \\
& \text{THEN LASTROW} = \min (\text{STARTROW} + \text{RPP} \times \text{DIFF}, \text{ENDROW}) - \text{DIFF}; \\
& \text{ELSE LASTROW} = \max (\text{STARTROW} + \text{RPP} \times \text{DIFF}, \text{ENDROW}) - \text{DIFF}; \\
& \text{DO COL} = \text{STARTROW} + \text{DIFF} \text{ TO LASTROW BY DIFF;} \\
& \text{IF DIFF > 0 /* DO NOT CROSS DIAGONAL */} \\
& \text{THEN TLASTROW} = \min (\text{COL-DIFF}, \text{MAXROW}); \\
& \text{ELSE TLASTROW} = \max (\text{COL-DIFF}, \text{MAXROW}); \\
& \text{DO ROW} = \text{STARTROW} \text{ TO TLASTROW BY DIFF;} \\
& \text{TEMP} = A(\text{ROW},\text{COL}) \\
& A(\text{ROW},\text{COL}) = A(\text{COL},\text{ROW}) \\
& A(\text{COL},\text{ROW}) = \text{TEMP}; \\
& \text{END;} \\
& \text{END;} \\
& \text{TEMP} = \text{STARTROW}; /* ALTER DIRECTION */ \\
& \text{STARTROW} = \text{ENDROW}; \\
& \text{ENDROW} = \text{TEMP} + \text{RPP} \times \text{DIFF}; \\
& \text{DIFF} = -\text{DIFF}; \\
& \text{END;} \\
\end{align*}
\]

Example 1—In Place Transpose Processing Multiple Rows and Alternating Directions

**MATRIX MULTIPLICATION**

In order to further investigate the programming rules given earlier, consider a simple program for performing matrix multiplication. Let A, B, and C be LXM, M\times N, and L\times N matrices, respectively. A program which computes \(C = A \times B\) is shown.

\[
\begin{align*}
\text{DO ROW} &= 1 \text{ TO L BY 1}; \\
\text{DO COL} &= 1 \text{ TO N BY 1}; \\
C(\text{ROW},\text{COL}) &= 0; \\
\text{DO INNER} &= 1 \text{ TO M BY 1}; \\
C(\text{ROW},\text{COL}) &= C(\text{ROW},\text{COL}) + A(\text{ROW},\text{INNER}) \times B(\text{INNER},\text{COL}); \\
\text{END}; \\
\text{END}; \\
\end{align*}
\]

By the reordering rule the DO statements with the COL and INNER indices should be interchanged if possible. The index COL never appears in any subscript position but the rightmost; therefore, the DO statement controlling the COL index should be innermost. Furthermore, the INNER and COL indices are only used together when the B matrix is referenced, and INNER is to the left of COL in that case. The number of page faults is reduced to

\[
F = p_A + Lp_B + p_C
\]

**TABLE IV—Matrix Multiply Page Fault Formulas**

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
<th>Page Fault (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>(p_A + \frac{[L^3]}{q} Np_a + p_c)</td>
<td>(p_A + \frac{[L^3]}{q} p_a + p_c)</td>
<td>(p_A + k - 2 + LN(p_a - k + 2) + p_c)</td>
</tr>
<tr>
<td>X</td>
<td>(p_A + \frac{[L^3]}{q} p_a + p_c)</td>
<td>(p_A + k - 2 + LN(p_a - k + 2) + p_c)</td>
<td>(p_A + k - a + c + \frac{[L^3]}{q} N(p_a - k + a + c) + p_c)</td>
</tr>
<tr>
<td>X</td>
<td>(p_A + k - a + c + \frac{[L^3]}{q} (p_a - k + a + c) + p_c)</td>
<td>(p_A + k - a + c + \frac{[L^3]}{q} (p_a - k + a + c) + p_c)</td>
<td>(p_A + k - a + c + \frac{[L^3]}{q} (p_a - k + a + c) + p_c)</td>
</tr>
</tbody>
</table>
by interchanging the DO statements and adding an appropriate loop to initialize the C matrix.

The DO statements controlling the ROW and INNER indices should not be interchanged since ROW is always in the leftmost position. Calculating the number of expected page faults verifies this fact, since interchanging the DO statements results in more page faults.

\[ F = M_{pa} + L p_b + p_c \]

The multiple row rule and alternating direction rule may also be applied to the matrix multiplication operation. Table IV shows the formulas for the number of page faults expected when different combinations of the rules are applied. The variables a and c represent the number of pages required to hold q rows of the A and C matrices respectively, where q is the number of rows being processed during each pass through the outermost loop. All of the formulas only hold for the condition a+c<k; that is, all the rows of A and C being processed during one outer loop traversal remain in real memory.

Each of these formulas has been evaluated for a matrix multiplication. Each matrix is assumed to be 101 X 101, L=M=N=101, spanning 20 pages, \( p_a = p_b = p_c = 20 \). The multiple row algorithms will process five rows at a time, q = 5 and a = c = 1. The amount of real memory varies. The results are displayed on the semi-logarithmic graph in Figure 4. For matrix multiplication the number of page faults can be reduced from over 200,000 to near 100 by application of the programming techniques described.

The matrix multiplication algorithms were programmed and executed. The environment was like that described for the transpose operation in Section IV. Tables V and VI show the measurements made. For those algorithms generating more than 2500 page faults, partial runs were completed and the results extrapolated.

Table VI illustrates several significant points about the application of the programming rules to matrix multiplication. The increase in problem CPU time is about the same for each rule or combination of rules. The elapsed time to complete the specified matrix multiply was reduced from 5.4 hours to 4.2 minutes by applying the rules. The programmer should consider all three rules. After applying the reordering rule, the programmer could be satisfied with a 99.9 percent decrease in page faults in exchange for a 12.7 percent increase in problem CPU time. By continuing with the other two rules, the problem CPU time is increased an additional 0.2 percent while the page faults are reduced by another 88.4 percent.

The matrix multiplication program which resulted when all three rules were applied to the standard program and then used to generate the data shown in Tables V and VI is shown as Example 2.

```plaintext
INSTART = 1;
INEND = M;
INDIFF = 1;
DO BASEROW = 1 TO L BY RRP;
   LASTROW = MAX(BASEROW + RRP-1, L);
   DO ROW = BASEROW TO LASTROW BY 1;
      DO COL = 1 TO N BY 1;
         C(ROW,COL) = C(ROW,COL) + A(ROW,INNER) * B(INNER, COL);
      END;
   END;
DO INNER = INSTART TO INEND BY
   DO COL = 1 TO N BY 1;
      C(ROW,COL) = C(ROW,COL) + A(ROW,INNER) * B(INNER, COL);
   END;
   INDIFF = -INDIFF;
END;

Example 2—Matrix Multiplication With All Rules Applied

REMARKS ON APPLICABILITY AND SUMMARY

In this paper the problem of performing matrix operations on large matrices is being considered from the point of view of the application programmer. The motivation is to reduce the number of page faults encountered while performing the
matrix operation in order to improve the performance of the application program. Three rules have been given which may be applied by the application programmer to a source program in order to reduce the number of page faults. (1) Nest loops so that matrix elements are referenced in the same order as they are stored. (2) Process all the rows in one page while the page is in real memory. (3) Alternate the direction of traversing a matrix to reuse pages not purged from real memory.

When virtual memory was first introduced, one of its major advantages was said to be that of allowing the programmer to work in a real memory environment without concern about overlays. Shortly thereafter, material began to appear discussing the locality and compactness of a program. Some papers actually discussed program design in terms of the average number of real memory pages an operating system would allocate to the application. Clearly, the freedom of the programmer is abridged when space considerations must be made.

Applying the rules described in this paper does not really diminish the freedom of the programmer but does allow the programmer to get better performance from the application program by using additional knowledge. The programmer may use knowledge of (1) the matrix mapping function of the language, (2) the word size and page size of the machine, (3) the paging algorithm, or (4) some combination of these items in order to reduce the number of page faults generated by the program.

The data presented show that using the rules will not only improve the performance of the application but may also greatly lessen the demands on the resources of the system. For a slight increase in problem CPU time, reductions can be realized in total CPU time, elapsed time or real memory costs, and channel time. Yet, when both a FORTRAN library and a PL/I library on a paging system were checked, the standard matrix transpose operation and the standard matrix multiplication operation used as examples in this paper were programmed. Several other matrix operations that were checked could have been easily improved in an obvious manner. In addition to program libraries, individuals concerned with program performance should be aware of the code executed when a program refers to all of the elements of matrix by a simple reference to the matrix by name. PL/I has several of these matrix operations defined. Also, several languages allow the notation A(I,* ) to refer to all of the elements in row I of matrix A. Program performance may be improved by explicitly writing the loop controls to access all of the elements in the row.

In an early part of this paper, the class of problems was restricted to those in which at least one row of a matrix could be contained in a page. That restriction was for the purpose of deriving formulas only. The programming techniques apply to matrices of any size. In fact, the programming rules may be applied to the problem of folding in processing large matrices in a non-paging environment.

The importance of applying a rule so that the number of page faults depends on the amount of real memory available should not be overlooked. For example, only the alternating direction rule introduced this dependency into the matrix multiplication operation. All of the matrix multiplication algorithms not employing alternating directions would have performed the same in three pages of real memory as in any larger number of pages until the point at which all of the data would fit in real memory. On the other hand, those algorithms, which depend on the amount of real memory available, had better performance for each page of real memory allocated to them.

BIBLIOGRAPHY
