RECEIVED BY DTIE OCT 4 1967

GEN. PRICES

OAK RIDGE NATIONAL LABORATORY

operated by

UNION CARBIDE CORPORATION

NUCLEAR DIVISION

for the

U.S. ATOMIC ENERGY COMMISSION

MASTER

ORNL - TM - 1969

COPY NO. -

10

DATE - September 5, 1967

The Effect on Program Execution Time of the Use of Fortran Multisubscripted Variables

J. G. Sullivan

Abstract

The computer time needed to execute a program written in Fortran can be improved considerably if the programmer understands the implications of the use of multidimensioned variables. The present discussion attempts to provide enough basic information so that a programmer may be able to produce programs which are efficient in their use of computer time.

NOTICE This document contains information of a preliminary nature and was prepared primarily for internal use at the Oak Ridge National Laboratory. It is subject to revision or correction and therefore does not represent a final report.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

- A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or
- B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

Introduction

The computer time required for the execution of a Fortran program depends on a number of factors. This discussion will concern itself with one of the most important of these factors, the use of subscripted variables and the implications (not always evident) of their use in Fortran programs. The purpose of this discussion is to provide programmers with a guide to improve the running time of programs by producing more "efficient" Fortran coding.

The methods suggested for use are, in general, independent of the compiler being used. Compilers vary in the efficiency and number of the program instructions they produce. The effectiveness of the methods outlined here will produce significant running time improvements for the majority of the compilers in general use but for some compilers a few of the methods may produce only marginal decreases in running time. It should, in any event, be clear to the programmer that "knowing" Fortran is not enough; the programmer must also be alert to which "Fortran" and which version of that Fortran he is using.

The criticisms and suggestions of Mrs. Nancy B. Alexander in the preparation of this report are greatly appreciated.

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

USE OF SUBSCRIPTS

For any Fortran programmer, the use of subscripted variables allows a facility for handling either quite simple or extremely complicated functional values. Elements of a two dimensional array, A, can be written and used in a manner reasonably close to the usual mathematical notation: e.g. $A(I,J)$ for A_{ij} . Computations using arrays are facilitated by the subscripting allowed and by the use of DO loops to change the indices in a regular manner.

The addition of two matrices is a simple coding problem involving two DO loops:

```
DO 1 I=1,N
DO 1 J=1,N
1 C(I,J)=A(I,J)+B(I,J)
```

Multiplication of a matrix by a scalar is a simple process.

```
DO 1 I=1,N
DO 1 J=1,N
1 A(I,J)=A(I,J)*SCALAR
```

The multiplication of two matrices is slightly more complicated, involving the use of three DO loops.

```
DO 1 K=1,N
DO 1 I=1,N
C(I,K)=0.
DO 1 J=1,N
1 C(I,K)=C(I,K)+A(I,J)*B(J,K)
```

Unfortunately, statements using variables with two or more subscripts, and which lie within a DO loop, are a potential hazard for the

programmer. To explain why this statement is made, a description of the implications of the appearance of a subscripted variable in a program must be given.

SINGLY DIMENSIONED VARIABLES

Computers have only sequential storage available to them and a one-dimensional array fits nicely into this machinery as the array can then be laid out in sequential storage cells in memory.* An array, A, dimensioned with length 5 is put in memory as:

A(1)

A(2)

A(3)

A(4)

A(5)

The computer instructions produced by a Fortran compiler when it runs across the variable A(I) is to perform an address calculation^x which adds a base address for the array, A, to the value of the index I. Because subscripts in Fortran may only run from 1 to a higher value, two possible ways to find the memory address of A(I) are available:

- 1) If A_1 is the computer memory address of the element A(1), then

$A_1 + 1$ is the address of the element A(2),

$A_1 + 2$ is the address of the element A(3), etc.

To find the address of A(I) the compiler would generate the addition of $A_1 + (I-1)$ so that if

I = 1 then the address of A(I) is $A_1 + (1-1)$ or A_1

I = 2 then the address of A(I) is $A_1 + (2-1)$ or $A_1 + 1$, etc.

* There are slight variations in the meanings of various terms used here from machine to machine but the differences are essentially trivial for this discussion.

Let us say that $A(1)$ is computer memory address, 100;

then

100: $A(1)$

101: $A(2)$

102: $A(3)$

103: $A(4)$

104: $A(5)$

and $A(I)$ is found by $A_1 + (I-1)$ or $100 + (I-1)$

$$I = 1 \quad A_1 + (1-1) = A_1 = 100$$

$$I = 2 \quad A_1 + (2-1) = A_1 + 1 = 101, \text{ etc.}$$

2) Because of the requirement of subtracting the constant 1 from the index I every time, a slight variation is usually introduced by compilers to save calculation time. A fictitious address A_0 corresponding to $A(0)$ (that is, the address $A_1 - 1$) is used as the base address in the addition to find variable addresses. Thus, conceptually, memory is laid out so that now the address of $A(I)$ is $A_0 + I$ and the subtraction has been eliminated from the address calculation.

99: $A(0)$

100: $A(1)$

101: $A(2)$

102: $A(3)$

103: $A(4)$

104: $A(5)$

Now the address of $A(1)$ can be expressed as $A_0 + 1$ where $A_0 = 99$ or $A_1 - 1$

address of $A(1)$ is $A_0 + 1 = 99 + 1 = 100$

address of $A(2)$ is $A_0 + 2 = 99 + 2 = 101$

address of $A(3)$ is $A_0 + 3 = 99 + 3 = 102, \text{ etc.}$

DOUBLY DIMENSIONED ARRAYS

Now let us examine how storage is arranged for arrays with two dimensions. Conventionally, in Fortran, multi-dimensioned arrays are stored in sequential memory locations and for an array dimensioned $A(3,2)$ storage is arranged:

100:A(1,1)

101:A(2,1)

102:A(3,1)

103:A(1,2)

104:A(2,2)

105:A(3,2)

Note that an easy rule to remember is that the left-most subscript runs through its range before the right-most is incremented.

The calculation of the general address $A(I,J)$ is more cumbersome and time consuming than for the case with the single subscript. First of all, the two-dimensional array, $A(3,2)$, is considered by Fortran compilers as if it were a single-subscripted array, $A'(6)$,

100:A(1,1) or $A'(1)$

101:A(2,1) or $A'(2)$

102:A(3,1) or $A'(3)$

103:A(1,2) or $A'(4)$

104:A(2,2) or $A'(5)$

105:A(3,2) or $A'(6)$

and the problem is to reduce the two dimensional subscript (in a unique manner) to a one dimensional subscript. Then the one-dimensional subscript is used as described earlier. Given a variable dimensioned $A(N,M)$, the method used

to find the element $A(I,J)$ is to let $I' = (J-1) * N + I$ and then find $A'(I')$. For dimensioned array $A(3,2)$,

(I,J) or $(J-1)*N+I = I'$
 $(1,1)$ or $(1-1)*N+1 = 1$
 $(2,1)$ or $(1-1)*N+2 = 2$
 $(3,1)$ or $(1-1)*N+3 = 3$
 $(1,2)$ or $(2-1)*N+1 = 3+1 = 4$
 $(2,2)$ or $(2-1)*N+2 = 3+2 = 5$
 $(3,2)$ or $(2-1)*N+3 = 3+3 = 6$

Example using $A'_0(I')$

	$A(I,J)$		$J*N+I-N=I'$	A'_0	$A'_0(I')$
99		$A'(0)$			
100	$A(1,1)$	$A'(1)$	$1*3+1-3=1$	99	100
101	$A(2,1)$	$A'(2)$	$1*3+2-3=2$	99	101
102	$A(3,1)$	$A'(3)$	$1*3+3-3=3$	99	102
103	$A(1,2)$	$A'(4)$	$2*3+1-3=4$	99	103
104	$A(2,2)$	$A'(5)$	$2*3+2-3=5$	99	104
105	$A(3,2)$	$A'(6)$	$2*3+3-3=6$	99	105

A constant term, $-N$, always appears in the address computation of $A'_0 + I' = A'_0 + J*N + I - N$. We define a "fictitious" address, A'_{-N} , where $(A'_{-N} = A'_0 - N)$ so that the address computation becomes $A'_{-N} + J*N+I$ or, in our notation, $A'_{-N}(I'')$, with $I'' = J*N+I$.

Example using $A'_{-N}(I'')$

	$A(I,J)$		$J*N+I=I''$	A'_{-N}	$A'_{-N}(I'')$
96		$A'(-3)$		A'_{-3}	$A'_{-3}(I'')$
97		$A'(-2)$			
98		$A'(-1)$			
99		$A'(0)$			
100	$A(1,1)$	$A'(1)$	$1*3+1=4$	96	100
101	$A(2,1)$	$A'(2)$	$1*3+2=5$	96	101
102	$A(3,1)$	$A'(3)$	$1*3+3=6$	96	102
103	$A(1,2)$	$A'(4)$	$2*3+1=7$	96	103
104	$A(2,2)$	$A'(5)$	$2*3+2=8$	96	104
105	$A(3,2)$	$A'(6)$	$2*3+3=9$	96	105

TRIPLY DIMENSIONED ARRAYS

Given a three-dimensional variable $A(L,M,N)$, to reduce the element $A(I,J,K)$ to $A'(I')$, I' is calculated by the formula:

$$I' = (K-1)*L*M + (J-1)*L + I$$

or, separating the constant portion,

$$I' = K*(L*M) + J*L + I - (L*M + L)$$

The base address corresponding to A_{-L*M-L} is used in order to eliminate the subtraction of the constant term, $L*M+L$, and

$$A'(I') = A'_{(-L*M-L)}(I') \quad \text{where} \quad (I' = K*(L*M) + J*L + I).$$

For an array dimensioned $A(2,3,4)$, ($L=2$, $M=3$, $N=4$), and
 $A(I,J,K)$, $I' = K*6 + J*2 + I$

	$A(I,J,K)$	$I + J*2 + K*6 = I'$	$A'_{(-L*M-L)}$ $A'_{(-8)}$	$A'_{(-L*M-L)}(I')$ $A'_{-8}(I')$
100	$A(1,1,1)$	$1 + 1*2 + 1*6 = 9$	91	100
101	$A(2,1,1)$	$2 + 1*2 + 1*6 = 10$	91	101
102	$A(1,2,1)$	$1 + 2*2 + 1*6 = 11$	91	102
103	$A(2,2,1)$	$2 + 2*2 + 1*6 = 12$	91	103
104	$A(1,3,1)$	$1 + 3*2 + 1*6 = 13$	91	104
105	$A(2,3,1)$	$2 + 3*2 + 1*6 = 14$	91	105
106	$A(1,1,2)$	$1 + 1*2 + 2*6 = 15$	91	106
107	$A(2,1,2)$	$2 + 1*2 + 2*6 = 16$	91	107
108	$A(1,2,2)$	$1 + 2*2 + 2*6 = 17$	91	108
109	$A(2,2,2)$	$2 + 2*2 + 2*6 = 18$	91	109
110	$A(1,3,2)$	$1 + 3*2 + 2*6 = 19$	91	110
111	$A(2,3,2)$	$2 + 3*2 + 2*6 = 20$	91	111
112	$A(1,1,3)$	$1 + 1*2 + 3*6 = 21$	91	112
⋮	⋮	⋮	⋮	⋮
123	$A(2,3,4)$	$2 + 3*2 + 4*6 = 32$	91	123

DISCUSSION

Because one of the objects of production type programs is to produce as many useful answers as possible in the least amount of machine time the use of subscripted variables and the compiler's treatment of them must be carefully

examined. Normally, in programs which use a significant amount of computer time, it is necessary to deal only with the use of subscripts in one subroutine or in a section of coding within one subroutine. Some compilers (e.g. IBM Fortran IV for the /360) do a remarkably good job of eliminating many of the multiplications necessary for the reduction of multi-dimensional subscripts to single subscripts inside DO loops. However, in almost all cases, a saving of machine time can be made by changing the Fortran coding to eliminate multiple subscripts and performing the calculation using a single subscript. In general it should be hoped that, no matter how well written a compiler may be, a programmer who knows the nature and probable paths to be taken in a routine can simplify the job of the compiler which is to produce machine coding which will perform the Fortran statement in the shortest amount of time.

In order to understand the possibilities and problems involved the following examples may be useful. On the left is the original Fortran coding and on the right is the pseudo-Fortran[†] coding implied. Examples 1,2,3 and 4 illustrate the effect of using subscripts as arguments in a CALL.

Example 1

DIMENSION A(4,5)	DIMENSION A(4,5)
CALL WORK (I,K)	CALL WORK (I,K)
	I'=I+J*4
A(I,J)=X	A'(I')=X

[†] By "pseudo-Fortran" we mean to indicate the actual calculations implied by the original Fortran statement. This seems an easier and more understandable way of explaining the various problems than by listing the computer's machine language which is really being generated. The notation, A'(I'), is meant to imply that the compiler has made use of the "dodge" to avoid the subtraction of a constant in producing an address.

Example 2

```
DIMENSION A(4,5)
```

```
A(I,J)=Y
```

```
CALL WORK (I,K)
```

```
A(I,J)=X
```

```
DIMENSION A(4,5)
```

```
I'=I+J*4
```

```
A'(I')=Y
```

```
CALL WORK (I,K)
```

```
I'=I+J*4
```

```
A'(I')=X
```

Note that in the "pseudo-Fortran" on the right, above, the $J*4$ multiplication is performed twice so that, in reality, a "smart" compiler might produce the following:

```
DIMENSION A(4,5)
```

```
J'=J*4
```

```
I'=I+J'
```

```
A'(I')=Y
```

```
CALL WORK (I,K)
```

```
I'=I+J'
```

```
A'(I')=X
```

Example 3

DIMENSION A(4,5)	DIMENSION A(4,5)
	I'=I+J*4
A(I,J)=Y	A'(I')=Y
CALL WORK(I,J)	CALL WORK(I,J)
	I'=I+J*4
A(I,J)=X	A'(I')=X

Since, under the rules of Fortran, the subroutine WORK may cause either or both of the values of the arguments to change, the compiler has no choice but to generate the coding for $I'=I+J*4$ both before and after WORK is called, so that we emphasize: implicit in the simple replacement operation, $A(I,J)=X$, is at least one multiplication and one addition. (On the 360/75 Model I the integer multiplication time, $2.8 \mu\text{sec}$, is a factor of four greater than the addition time, $.7 \mu\text{sec}$).

One other factor which must always be borne in mind is that any variable name in a subroutine or main program which appears in any COMMON statement within that same subroutine or main program must be assumed to have a new value after a CALL statement is performed. Thus:

Example 4

DIMENSION A(4,5)	DIMENSION A(4,5)
COMMON I,J	COMMON I,J
A(I,J)=Y	I'=I+J*4
CALL WORK (M,N)	A'(I')=Y
	CALL WORK (M,N)
A(I,J)=X	I'=I+J*4
	A'(I')=X

Hence, after WORK is called, the compiler must assume that the value of J after the CALL may not be the same as before the CALL. WORK might not have any reference at all to COMMON in the version of WORK being used but since it would be possible to supply an alternate version of WORK which might use COMMON and change the values of I and J, all compilers must assume the "worst case" holds and, consequently, the only correct procedure is to recompute the value of I'.

Example 5

One more influence on compiler translation is the EQUIVALENCE statement. Consider

DIMENSION M(50),A(4,5)	DIMENSION M(50),A(4,5)
EQUIVALENCE (M(13),J)	EQUIVALENCE (M(13),J)
A(K,J)=Y	I'=J*4+K
M(L)=I	A'(I')=Y
A(N,J)=Z	M(L)=I
	I'=J*4+N
	A'(I')=Z

Notice that the "pseudo-Fortran" should not be written:

```

J'=J*4
I'=J'+K
A'(I')=Y
M(L)=I
I'=J'+N
A'(I')=Z

```


because only the "perfect" compiler, which is never going to be written, could hope to determine whether or not L was equal to 13 so that the statement $M(L)=I$ would or would not change the value of J . "Pseudo" code would have to look like this:

```

J'=J*4
I'=J'+K
A'(I')=Y
M(L)=I
IF(L-13) 1, 2, 1
2 J'=J*4
1 I'=J'+N
A'(I')=Z

```

It is obvious that a compiler could insert this checking but then one runs into an imponderable of the coding game: What percentage of the time will L be equal to 13? If you will take on faith the estimate that the coding to calculate $L-13$ and then to branch to the proper point will take about $1/3$ the time necessary for the multiplication to produce J' then it is clear that the percentage of the time that L is equal to 13 determines the "best" coding. Let us put relative times on each statement:

	Cost
IF(L-13) 1, 2, 1	1
2 J'=J*4	3

If the test for $L=13$ were not made then statement number 2 would always be executed and the "cost" for 100 passes through this section of coding would be 300 units of time. If the test for $L=13$ is put in, and L is never equal 13 then the "cost" is only 100 units of time (the cost of the test $\times$ 100). If L is always equal 13, then the "cost" is 400 units. The breakeven point is 300 units (the no-test "cost") so that $300 = 100 \times 1 + M \times 3$ where M is the number of times, out of 100, L is equal to 13. Solving for M , $M = 200/3$ or $66\frac{2}{3}$. So we see that if L is 13 more than $1/3$ of the time we have spent more than we have gained by putting the test in. Only the programmer knows if the test should be included and this sort of decision should not be left to a compiler.

We have seen that the appearance of a variable (used as a subscript) in either COMMON or EQUIVALENCE statements can cause a compiler to generate redundant coding in order to allow for the worst possible case. A simple way of "helping" the compiler is that in the first executable statements after entry to a subroutine all subscripts which appear in COMMON or EQUIVALENCE statements should be set equal to a comparable variable in the subroutine. This should be done in those cases where the subscript in COMMON or EQUIVALENCE is not to be changed during the execution of the subroutine.

BEFORE	AFTER
SUBROUTINE Q	SUBROUTINE Q
COMMON I	COMMON I
A(L,I)=Z	IX=I
CALL WORK(K,L)	A(L,IX)=Z
A(L,I)=Y	CALL WORK(K,L)
	A(L,IX)=Y

Example 6:

BEFORE

SUBROUTINE ABC(X,Z)	SUBROUTINE ABC(X,Z)
DIMENSION A(3,4)	DIMENSION A(3,4)
COMMON I,J	COMMON I,J
A(I,J)=Z	J'=J*3
CALL WORK(K)	I'=J'+I
A(K,J)=X	A'(I')=Z
RETURN	CALL WORK(K)
	J'=J*3
	I'=J'+K
	A'(I')=X
	RETURN

AFTER

SUBROUTINE ABC(X,Z)	SUBROUTINE ADC(X,Z)
DIMENSION A(3,4)	DIMENSION A(3,4)
COMMON I,J	COMMON I,J
JJ=J	JJ=J
A(I,JJ)=Z	JJ'=JJ*3
CALL WORK(K)	I'=JJ'+I
A(K,JJ)=X	A'(I')=Z
RETURN	CALL WORK(K)
	I'=JJ'+K
	A'(I')=X
	RETURN

This coding, once again, assumes a "smart" compiler which will save the multiplication but one could guard against variations in the compiler by the following:

<p>Example 7:</p> <pre> SUBROUTINE ABC(X,Z) DIMENSION A(12)[†] COMMON I,J JJ=3*(J-1) IJ=JJ+I A(IJ)=Z CALL WORK(K) KJ=JJ+K A(KJ)=X RETURN </pre> <p style="margin-left: 40px;">[†]from A(3,4)</p>	<p>Note that</p> <p>in this example</p> <p>the "pseudo-Fortran"</p> <p>is identical</p> <p>to the "real" Fortran</p> <p>and not even a</p> <p>"dumb" compiler</p> <p>can add the</p> <p>extra multiplication.</p>
--	---

(It must be admitted that one extra storage cell, JJ, has been added and that the "smart" compiler could, on some machines, keep JJ in a register and not have to store and load it but, in most instances, time will still have been saved.)

After seeing these examples, the following remark is bound to be made by someone: "But if I have to go to all this extra trouble what's the use? The machine time saved isn't that much!" The only proper reply to such a remark is to prove that it can indeed be worth the trouble! Generally speaking, there is almost always a gain to be made by reducing double and triple subscripting to a single subscript.

As one example, the simple task of setting a doubly dimensioned array, A, to zero is usually accomplished by:

```
DO 1 I=1,N
DO 1 J=1,N
1 A(I,J)=0.
```

In all compilers it is faster to use an array A, dimensioned M (where $M = N^2$), and code as:

```
DO 1 K=1,M
1 A(K)=0.
```

This might also be accomplished by:

```
DIMENSION A(10,10)
DIMENSION B(100)
EQUIVALENCE (A,B)
DO 1 K=1,100
1 B(K)=0.
```

Equivalencing B to A has the added advantage that in sections of coding where time must be saved, B can be used and in seldom used sections of coding A can be used for simplicity.

Let us consider the effect of DO loops on subscripts.

Example 8:

DIMENSION A(4,3)	DIMENSION A(4,3)
DO 1 I=1,4	I=1
DO 1 J=1,3	10 J=1
1 A(I,J)=0.	11 J'=J*4
	I'=J'+I
	A'(I')=0.
	J=J+1

```
IF(J.LT.4) GO TO 11
```

```
I=I+1
```

```
IF(I.LT.5) GO TO 10
```

"Smart" compilers can look through the pseudo code on the right and serve up the following delectable item:

```
DIMENSION A(4,3)
```

```
I=1
```

```
10 J'=0
```

```
11 I'=J'+I
```

```
A'(I')=0.
```

```
J'=J'+4
```

```
IF(J'.LT.12) GO TO 11
```

```
I=I+1
```

```
IF(I.LT.5) GO TO 10
```

"smart" programmers can do this:

```
DIMENSION A(12)
```

```
DO 1 I=1,12
```

```
1 A(I)=0.
```

```
DIMENSION A(12)
```

```
I=1
```

```
10 A(I)=0.
```

```
I=I+1
```

```
IF(I.LT.13) GO TO 10
```

The preceding discussion and examples were presented to show that the simple statement $A(I,J)=X$ can imply much more than appears on the surface. Fortran coding is like an iceberg: what appears above the surface (the Fortran statements) is only a small percentage of the total. It must be emphasized that each compiler has its own peculiarities, some can help the programmer and some can do him injury in the form of excessive calculation time. The present version of ORNL Fortran/360, for instance, does all subscript calculations on a statement by statement basis. A newer version will handle the EQUIVALENCE and COMMON checking of subscripts so as to become more nearly "perfect".

The IBM Fortran compiler does a very good job of moving the constant calculations for subscripts outside DO loops and avoiding multiplications but is dependent on the complexity of the DO loop (if there are subscripted variables with differing dimensions extra stores and loads of subscripts are introduced). One curious feature is that the same DO loops appearing at the beginning and at the end of a fair-sized program may have different machine language coding and require differing times for execution.

In regard to the saving of calculation time one has to be aware of the nature of some calculations. Generally speaking, most lengthy calculations are the result of iterative or repetitive processes and there are usually sections of coding which can be identified as accounting for 75% or more of the total time. Consider a variable dimensioned (10,20,100). To perform computations to calculate each individual value these DO-loops (one for each subscript) could be used:

```
7 DO 1 I=1,10
8 DO 1 J=1,20
9 DO 1 K=1,100
```

and let us assume that between statements 7 and 8 there are i Fortran statements, between 8 and 9 there are j Fortran statements and between 9 and the DO loop end there are k statements. Then for one pass through the loops we are executing $10i + 200j + 20000k$ statements. Let us make the assumptions that $i=j=k=10$ and that a statement takes 5 microseconds. Then the total time for the loops would be $(100+2000+200000) \times 5 = 1,010,500$ microseconds or, approximately, 1 second. Notice that the interior loop, of course, dominates. Given complicated calculations (which are the rule rather than the exception) it can be seen, from the assumptions made, that removing one statement from the k -loop is equivalent to removing one hundred statements from the j -loop so we can conclude that anything which can be taken out of the k -loop means a very large relative savings in time. Take the following simple example:

Example 9:

```

      DO 1 I=1,10
        DO 1 J=1,20
          DO 1 K=1,100
            1 X(I,J,K)=A(I,J)*B(J,K)+C(I,K)

```

Change it to this:

```

      DO 1 I=1,10
        DO 1 J=1,20
          AIJ=A(I,J)
          DO 1 K=1,100
            1 X(I,J,K)=AIJ*B(J,K)+C(I,K)

```

One very important observation should be made here by programmers who might have the array A in bulk storage: The load time from this storage is $4. \mu\text{sec}$ versus $.7 \mu\text{sec}$ from fast storage. Instead of performing 20,000 loads from

bulk (=80 milliseconds) only 200 loads from bulk are made and 20,000 loads from fast storage are made (=14.8 milliseconds). In any case, the subscripting on A is taken out of the innermost loop and a gain has been made.

Consider the following example:

Example 10:

```
SUM=0.
DO 1 I=1,10
DO 1 J=1,20
DO 1 K=1,100
1 SUM=SUM+A(I,J)*B(J,K)*C(I,K)
```

In this example statement 1 will be executed 20,000 times. The "best" machine language generated for the statement will consist of indexing calculations (assume 1 μ sec.), a load of one of the variables (.7 μ sec), two floating-point multiplications (4.2 μ sec), an addition (.85 μ sec), an increment of indices (.8 μ sec) and a branch (1. μ sec). Total statement time would be $1. + .7 + 4.2 + .85 + .8 + 1. = 8.55 \mu$ sec. Total loop time would be $20,000 \times 8.55 = .171$ seconds.

```
SUM=0.
DO 1 I=1,10
DO 1 J=1,20
SUMA=0.
DO 2 K=1,100
2 SUMA=SUMA+B(J,K)*C(I,K)
1 SUM=SUM+SUMA*A(I,J)
```

The loop on statement 2 would now take 6.45 μ sec at most and the total time would be very close to $20,000 \times 6.45$ or .129 seconds. The outer loop multiplication by $A(I,J)$ can be neglected so long as the innermost loop is performed at least the same number of times as the outer loop (instead of 20,000 multiplies by $A(I,J)$ there are now only 200). The decrease in running time is from .171 to .129 seconds and the machine time for the loop has been decreased by about 25%.

It must be emphasized that this very simple example can be reduced in running time by 25%. More complicated examples can "see" a much greater decrease.

Let us isolate the following portion from example 9.

```
DO 1 J=1,20
```

```
DO 1 K=1,100
```

```
B(J,K)
```

and let us consider B' dimensioned 2000 instead of B dimensioned (20,100).

Let $B'(JK)$ correspond to $B(J,K)$, then $JK=(K-1)*20+J$ and :

```
DO 1 J=1,20
```

```
DO 1 K=1,100
```

```
JK=(K-1)*20+J
```

```
B'(JK)
```

Now note that JK varies with K as follows

$K=1$

$JK=J$

$K=2$

$JK=J+20$

$K=3$

$JK=J+40$

etc.

the J term is constant so far as the K-loop is concerned so, finally,
we may do the following:

DO 1 J=1,20	or	DO 1 J=1,20
JK=J-20		JK=J
DO 1 K=1,100		DO 1 K=1,100
JK=JK+20		B'(JK)
B'(JK)		JK=JK+20

Note further that if the value of K is not needed elsewhere inside the loop even more can be done with the example on the right, above:

```
DO 1 J=1,20
DO 1 JK=J,2000,20
B'(JK)
```

Note that now there is only one set of incrementing to be performed (on JK, instead of, as previously, on JK and K)

Now let us go back and work on the complete DO loops in this way:

Example: DIMENSION A(10,20),X(10,20,100),B(20,100),C(10,100)

```
DO 1 I=1,10
DO 1 J=1,20
AIJ=A(I,J)
DO 1 K=1,100
1 X(I,J,K)=AIJ*B(J,K)*C(I,K)
```

In the following set of Fortran the same variable names A,B,C, and X are used but with single dimensions. This is only for clarity and new variables say AX, BX, CX, XX would normally be used and equivalenced with the original multi-subscripted variables.

Using the preceding discussion of $B(J,K)$ reduced to $B'(JK)$

let us make the first change:

Example 9a:

```

DO 1 I=1,10
DO 1 J=1,20
AIJ=A(I,J)
JK=J
DO 1 K=1,100
X(I,J,K)=AIJ*B(JK)*C(I,K)
1 JK=JK+20

```

Now let us perform the comparable operation on $C(I,K)$, transforming it to $C(IK)$.

```

DO 1 I=1,10
IK=I
DO 1 K=1,100
C(IK)
1 IK=IK+10

```

Example 9b:

```

DO 1 I=1,10
DO 1 J=1,20
IK=I†
AIJ=A(I,J)
JK=J
DO 1 K=1,100
X(I,J,K)=AIJ*B(JK)*C(IK)
IK=IK+10
1 JK=JK+20

```

[†]Note that $IK=I$ must be placed inside the J-loop.

Now let us perform the comparable operation on $X(I,J,K)$, transforming it to $X(IJK)$.

Example 9c:

```

DO 1 I=1,10
  IJ=I-10
  DO 1 J=1,20
    IK=I
    IJ=IJ+10
    AIJ=A(IJ)
    JK=J
    IJK=IJ
    DO 1 K=1,100
      X(IJK)=AIJ*B(JK)*C(IK)
      IK=IK+10
      JK=JK+20
    1 IJK=IJK+200

```

Finally, removing K from the innermost loop:

```

DO 1 I=1,10
  IJ=I-10
  DO 1 J=1,20
    IK=I
    IJ=IJ+10
    AIJ=A(IJ)
    JK=J
    IJMAX=IJ+19800
    DO 1 IJK=IJ,IJMAX,200
      X(IJK)=AIJ*B(JK)*C(IK)
      IK=IK+10
      JK=JK+20
    1 JK=JK+20
  (next column)

```

It is possible to reduce even this slightly (by working on the J loop in the same sort of way the K loop was simplified) but since the burden of the computation is on the innermost loop and since the example becomes tedious if there is nothing new to be learned we leave the simplification as an exercise for the reader. (The assembly language version could probably be reduced considerably no matter what the compiler.) This lengthy expansion may look frightening to the casual programmer but it should always be remembered that Fortran is a language devised to make machines more accessible to the scientist in a simple-minded way.

The above example shows the reduction of triple subscripts to single subscripts. It is also possible to make the relatively simple reduction of triple subscripts to double subscripts. This is not the "optimum" but does cause a reduction in running time.

Example:

BEFORE	AFTER
DIMENSION X(10,20,100)	DIMENSION X(10,20,100)
DO 1 J=1,20	DIMENSION XX(200,100)
DO 1 I=1,10	EQUIVALENCE(X,XX)
DO 1 K=1,100	DO 1 IJ=1,200
1 X(I,J,K)=X(I,J,K)+.5	DO 1 K=1,100
	1 XX(IJ,K)=XX(IJ,K)+.5

Following are examples of changing multiple subscripts to single subscripts within DO loops. Notice that the order in which each value of A is computed is preserved in the B computation.

Example 11:

BEFORE	AFTER
DIMENSION A(4,30)	DIMENSION A(4,30),B(120)
DO 1 J=1,30	EQUIVALENCE (A,B)
DO 1 I=1,4	DO 1 IJ=1,120
1 A(I,J)=A(I,J)+X	1 B(IJ)=B(IJ)+X

Example 12:

BEFORE	AFTER
DIMENSION A(4,30)	DIMENSION A(4,30),B(120)
	EQUIVALENCE (A,B)
DO 1 J=1,N	IJMAX=4*N
DO 1 I=1,4	DO 1 IJ=1,IJMAX
1 A(I,J)=A(I,J)+X	1 B(IJ)=B(IJ)+X

Example 13:

BEFORE	AFTER
DIMENSION A(4,30)	DIMENSION A(4,30),B(120)
	EQUIVALENCE (A,B)
DO 1 I=1,4	DO 1 I=1,4
DO 1 J=1,30	DO 1 IJ=I,120,4
1 A(I,J)=A(I,J)+X	1 B(IJ)=B(IJ)+X

Example 14:

BEFORE	AFTER
DIMENSION A(4,30)	DIMENSION A(4,30),B(120)
	EQUIVALENCE (A,B)
DO 1 I=1,M	DO 1 I=1,M
DO 1 J=1,30	DO 1 IJ=I,120,4
1 A(I,J)=A(I,J)+X	1 B(IJ)=B(IJ)+X

Example 15:

BEFORE	AFTER
DIMENSION A(4,30)	DIMENSION A(4,30),B(120)
	EQUIVALENCE (A,B)
DO 1 I=1,M	IJMAX=4*N
DO 1 J=1,N	DO 1 I=1,M
1 A(I,J)=A(I,J)+X	DO 1 IJ=I,IJMAX,4
	1 B(IJ)=B(IJ)+X

Example 16:

BEFORE	AFTER
DIMENSION A(10,20,30)	DIMENSION A(10,20,30),B(6000)
	EQUIVALENCE (A,B)
DO 1 K=1,30	DO 1 IJK=1,6000
DO 1 J=1,20	1 B(IJK)=B(IJK)+X
DO 1 I=1,10	
1 A(I,J,K)=A(I,J,K)+X	

(Notice that in both the double and triple subscript examples where the indexes vary most rapidly from left to right (Examples 11, 12, 16) and range sequentially in storage through the array that the reduction

is the simplest and, all other things being equal, DO loops in general should be nested so that the variation is most rapid from left to right).

Example 17:

BEFORE	AFTER
DIMENSION A(10,20,30)	DIMENSION A(10,20,30),B(6000)
	EQUIVALENCE (A,B)
DO 1 I=1,10	DO 1 I=1,10
DO 1 J=1,30	DO 1 IKJ=I,6000,10
DO 1 K=1,20	1 B(IKJ)=B(IKJ)+X
1 A(I,K,J)=A(I,K,J)+X	

Example 18:

BEFORE	AFTER
DIMENSION A(10,20,30)	DIMENSION A(10,20,30),B(6000)
	EQUIVALENCE (A,B)
DO 1 I=1,10	DO 1 I=1,10
DO 1 J=1,20	DO 1 IJ=I,200,10
DO 1 K=1,30	DO 1 IJK=IJ,6000,200
1 A(I,J,K)=A(I,J,K)+X	1 B(IJK)=B(IJK)+X

(Note the difference between example 16 and 18 made by the order in which the DO loops are written.)

It is clearly easier to debug programs containing statements like $C(I,K)=A(I,J)*B(J,K)+C(I,K)$ than the reduction of subscripts which is more complicated. It is also clear that programmers should use all the Fortran subscripting allowed until the correctness of the program has been demonstrated. But at this point most programmers stop programming and run the code for production purposes. Just a few more days used to refine the calculation can produce startling improvements in running time. One simple program in which 24 minutes out of 28 were spent in one subroutine was recoded using the techniques discussed above and the final Fortran version ran in 14 minutes out of a total of 18.

One point of overriding importance must be emphasized. The 14-minute Fortran version was rewritten in machine language instructions and the running time was reduced to 4 minutes! These numbers are reasonably typical for any fairly complicated calculation:

Fortran using multi-subscripted variables	24 minutes
Fortran reduced to basics	14 minutes
Fortran rewritten in assembly language	4 minutes

The question might be put concerning why compilers could not have produced a code which would run in 4 minutes rather than 24. Two factors are at work here. One is that better running time of Fortran programs can, of course, be achieved by making the compilers more complicated. However, the compiling time can increase by a factor of 10 with only slight reductions in program execution time. Historically, on the IBM 704 computer a few years ago there was a sense switch at the console one could set if optimizing were desired. Without optimizing, Fortran programs

would compile in a minute or two but with that sense switch set you could go to lunch and come back to watch the compiler in its final phases. The execution of the Fortran was generally improved a bit but there were even occasions where the compiler so outsmarted itself that the optimized version ran longer!

The second factor is that the programmer should be the best judge of which paths are most probable and the calculation should be designed to zip through these paths. The compiler tries to zip through all paths and ends up by marching through them all and zipping through none. Intelligence and the ability to take short cuts should still be the forte of programmers while the forte of compilers remains mechanical reliability in the translating process.

INTERNAL DISTRIBUTION

- 1-2. Central Research Library
- 3. Document Reference Section
- 4-8. Laboratory Records
- 9. Laboratory Records - Record Copy
- 10-24. Division of Technical Information Extension
- 25. Research and Development, ORO
- 26. ORNL Patent Office
- 27-76. J. G. Sullivan
- 77. A. A. Brooks
- 78. V. R. Cain
- 79. W. B. Gardner
- 80-730. ORCID Distribution List

EXTERNAL DISTRIBUTION

- 731. Wallace Givens, Argonne National Laboratory
Applied Mathematics Division, Argonne, Illinois
- 732. R. P. Leinius, University of Wisconsin Computing Center
1210 W. Dayton Street, Madison, Wisconsin
- 733. W. F. Miller, Stanford University
Computer Science Division, Stanford, California
- 734. J. A. Thompson, Control Data Corporation
4201 North Lexington Avenue, St. Paul 12, Minnesota